CodeWarrior®

Debugger User Guide

metrowerks ®

How to Contact Metrowerks:

U.S.A. and international	Metrowerks Corporation P.O. Box 334 Austin, TX 78758 U.S.A.
Canada	Metrowerks Inc. 1500 du College, Suite 300 Ville St-Laurent, QC Canada H4L 5G6
Ordering	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`
CompuServe	`goto Metrowerks`

Table of Contents

Table of Contents

DBG

Table of Contents

1

Introduction

Welcome to the CodeWarrior Debugger manual.

NOTE: On occasion a CodeWarrior product ships with an earlier version of the IDE than reflected in this user guide. In that case, your IDE will not have the new features described in this manual. You can identify new features by referring to "What's New" on page 13.

In some cases a patch may become available to update the tools. You can visit the Metrowerks website at `http://www.metrowerks.com` for information on available patches.

Overview of the Debugger Manual

A debugger is software that controls program execution so that you can see what's happening internally as your program runs. You use a 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.

This manual describes MW Debug, the source-level debugger for the Metrowerks CodeWarrior software development environment. To a great extent the same debugger works for all supported target chips, operating systems, and languages (C, C++, Pascal, Java, and assembly language). This manual often refers to MW Debug as "the CodeWarrior debugger," or simply as "the debugger."

This manual describes the common functionality of the debugger for all platforms. There may be some minor differences, either additions or unimplemented features, on a per-target basis. You should read the debugging chapter or chapters of the appropriate *Targeting* manual to cover the specific differences for your target.

The other topics in this chapter are:

- Read the Release Notes—important last minute information
- Manual Conventions—important information on conventions used in this manual
- What's New—a short review of new features
- System Requirements—hardware and software requirements
- Installing MW Debug—putting it together
- 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.

Manual Conventions

This section describes the different conventions used in this manual.

Typographical conventions

This manual uses some style conventions to make it easier to read and use:

Notes, warnings, tips, and beginner's hints

An advisory statement or **NOTE** may restate an important fact, or call your attention to a fact which may not be obvious.

A **WARNING** given in the text may call attention to something such as an operation that, if performed, could be irreversible, or flag a possible error that may occur.

A **TIP** can help you become more productive with the CodeWarrior IDE. Impress your friends with your knowledge of little-known facts that can only be learned by actually reading the fabulous manual!

A **For Beginners** note may help you better understand the terminology or concepts if you are new to programming.

Typeface conventions

If you see some text that appears in a `different` typeface (as the word `different` does in this sentence), you are reading file or folder names, source code, keyboard input, or programming items.

Text **formatted like this** means that the text refers to an item on the screen, such as a **menu command** or **control** in a dialog box.

If you are using an on-line viewing application that supports hypertext navigation, such as Adobe Acrobat, <u>text formatted like this</u> means you can click on it to view another topic or related information. For example, click this text, <u>"Overview of the Debugger Manual" on page 9</u>, to jump to a section that describes the rest of this manual.

Host Conventions

CodeWarrior runs on several different host platforms, each of which may have several versions of its operating system. For purposes of this manual, a generic platform identifier is used to identify the host platform, regardless of operating system.

The specific versions of the operating system that host CodeWarrior are:

- **Windows**—desktop versions of the WIndows operating system that are Win32 compliant, such as Windows 95 or Windows NT.

- **Mac OS**—desktop versions of Mac OS, System 7.1 or later.

Figure Conventions

CodeWarrior runs on several different host platforms. The visual interface on these hosts is nearly identical in all significant respects. When discussing a particular interface element such as a dialog box or window, the screenshot may come from any host. If you are using CodeWarrior on a different host, you should have no difficulty understanding the picture.

However, there are occasions when dialog boxes or windows are unique to a particular host. In that case, a screenshot from each unique host will appear and be clearly identified so that you can see how CodeWarrior works on your preferred host.

Keyboard Conventions

The default keyboard shortcuts for CodeWarrior on some platforms are very similar. However, keyboards and shortcuts do vary across host platforms. For example, a typical keyboard for a Windows machine has an Alt key, but a typical keyboard for a Mac OS computer calls the same key the Option key.

To handle these kinds of situations, CodeWarrior documentation identifies and uses the following paired terms in the text:

- Enter/Return—the "carriage return" or "end of line" key. This is not the numeric keypad Enter key, although in almost all cases that works the same way.

- Backspace/Delete—the Windows Backspace key and the Mac OS Delete key. In most cases, CodeWarrior maps these keys the same way. This is the key that (in text editing) causes the character before the insertion point to be erased. (This is not the Delete/Del, the "forward delete" key.)

- Ctrl/Command—the Windows Ctrl (control) key and the Mac OS Command key. In most cases, CodeWarrior maps these keys the same way. (This is not the Mac OS keyboard Control key.)

- Alt/Option—the Windows Alt key and the Mac OS Option key. In most cases, CodeWarrior maps these keys the same way.

For example, you may encounter instructions such as "Press Ctrl/ Command S to save a file," or "Alt/Option click the Function pop-up menu to see the functions in alphabetical order." Use the appropriate key as it is labeled on your keyboard.

Some combinations of key strokes require multiple modifier keys. In those cases, key combinations are shown connected with hyphens. For example, Ctrl-Alt/Option-Shift-Command.

In certain cases the cross-platform variation in keyboard shortcuts is more complex. In those cases, you will see more detailed instructions on how to use a keyboard shortcut for your host platform. In all cases the host and shortcut will be clearly identified.

What's New

There are a few new features that habe been added to the debugger.

CodeWarrior IDE and the Integrated Debugger

The CodeWarrior IDE now has MW Debug integrated into itself to provide seamless interaction between the programming and debugging of your source code. Some of the benefits of this integration include:

- Reduced memory requirements. With only one application running, memory demands are significantly reduced.
- Increased productivity. Since you don't have to switch back and forth between the IDE and Debugger to step through your code, set breakpoints, etc., a more efficient use of time is achieved, increasing your productivity.
- The integrated debugger fully supports x86, PowerPC, 68K, and Java debugging. No longer are separate debuggers required to debug each platform, the integrated debugger handles them all.

Once you enable the integrated debugger, choose **Debug** from the **Project** menu to activate and use the integrated debugger. You can pause the program at any time to set breakpoints, view variables or memory, step into or out of routines, as well as perform many other debugging tasks.

NOTE: Some versions of the CodeWarrior IDE do not ship with the integrated debugger. In those cases, debugging support is provided by the external MW Debug application or other third party debuggers. See your platform's Targeting manual for additional information on debugging a specific target.

To make use of the external debugger, ensure that MW Debug is running, then choose **Switch to MW Debugger**.

System Requirements

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

Windows

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.

Mac OS

MW Debug requires a Motorola 68020 processor or better, or a PowerPC 601 or better processor. MW Debug needs approximately 2 MB of disk space and 1.5 MB of RAM.

MW Debug requires System 7.1 or later (for 68K Macintosh) or System 7.1.2 or later (for Power Macintosh), and color QuickDraw. CFM-68K is required on 68K systems.

The watchpoints feature requires a Motorola 68020 or better processor with Virtual Memory turned on, or a PowerPC 601 processor or better with Virtual Memory either on or off. The watchpoints feature also requires System 7.5 or later. See "Watchpoints" on page 86 for more information.

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.

We strongly recommended that you use the CodeWarrior Installer to install the debugger to make sure you have all the required files.

Mac OS

MW Debug and the Debugger Plugins folder should be located in the (Helper Apps) folder or you may not be able to launch the debugger directly from a project.

In order to work, the debugger also requires the presence of a few system extensions in the Extensions subfolder of your System folder. The installer automatically places these items in your Extensions folder. If you install or remove any of them, you must restart your computer for the changes to take effect. Before using the debugger, make sure that the correct extensions are installed for your current target and platform, as shown in Figure 1.1:

Figure 1.1   Extra files required by MW Debug (Mac OS)
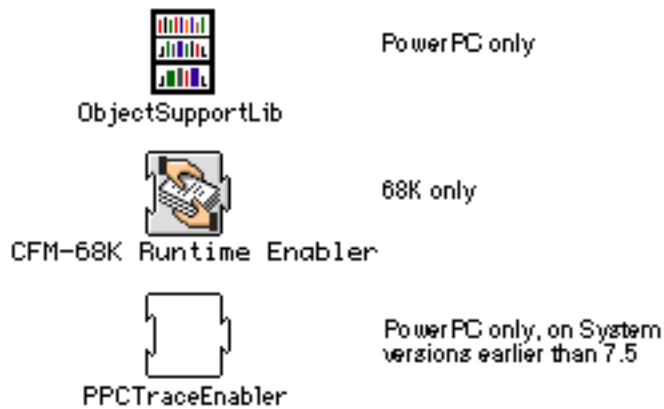
- ObjectSupportLib—a shared library needed for debugging PowerPC object code.
- CFM-68K Runtime Enabler—required to use the debugger shared library. You can find a version of the debugger on the Tools CD that does not use the shared library.
- PPCTraceEnabler—needed for debugging PowerPC object code on System versions earlier than 7.5.

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 19. To become familiar with the debugger interface, see "What You See Overview" on page 27.

If you don't know how to control program execution, set break-points, or modify variables, read "Basic Debugging Overview" on page 59, and "Expressions Overview" on page 103.

For reference on menu items in the debugger, see "Debugger Menus Overview" on page 133.

No matter what your skill level, if you have problems using the de-bugger, consult "About Troubleshooting" on page 149. 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 the *Targeting* manual for your target.

2

Getting Started

This chapter discusses how to prepare for debugging, and contains background information on symbolics files. Other 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 the code generated by a particular target within your CodeWarrior project file, you must make sure both the target and the individual source files within it are set up for debugging. When they are, CodeWarrior generates symbolics information that the debugger requires.

This section discusses each of these topics:

- Setting Up a Target for Debugging
- Setting Up a File for Debugging
- Generating Symbolics Information

Setting Up a Target for Debugging

To prepare a target for debugging, make sure it is the current target. Then choose the **Enable Debugger** command from the Project menu in the CodeWarrior IDE. When debugging is enabled for a target,

the menu item changes to **Disable Debugger**. Choosing **Disable Debugger** turns off debugging for the target and changes the menu item back to **Enable Debugger**.

The **Enable Debugger** command sets items in the Project window and the settings panels for the current target to tell the compiler and linker to generate debugging information. In response, the compiler and linker generate a symbolics file containing information for debugging at the source-code level.

See the *CodeWarrior User Guide* for more information on target settings.

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 25 for more information.

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 target.

Figure 2.1   Accepting changes set by Enable Debugger
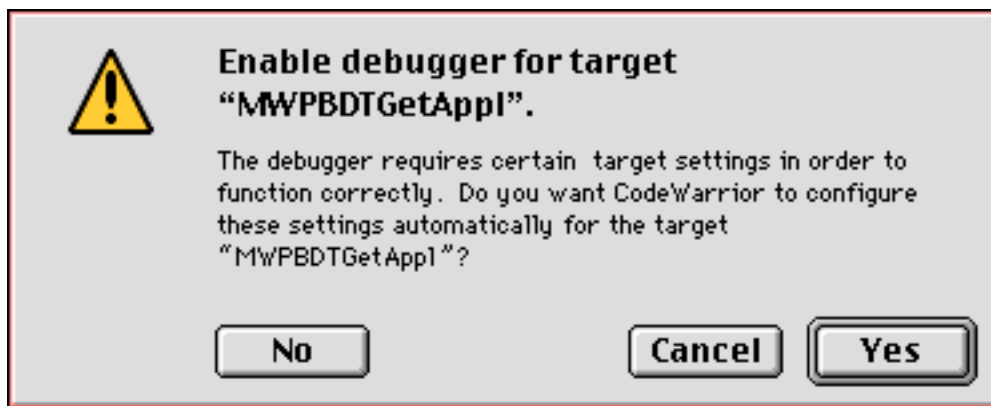
Setting Up a File for Debugging

After you have enabled debugging for the current target, 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 group names, a mark indicates that debugging is on for every file in the group and no mark means that debugging is off for one or more files in the group.

Figure 2.2   Setting debugging in the Project window
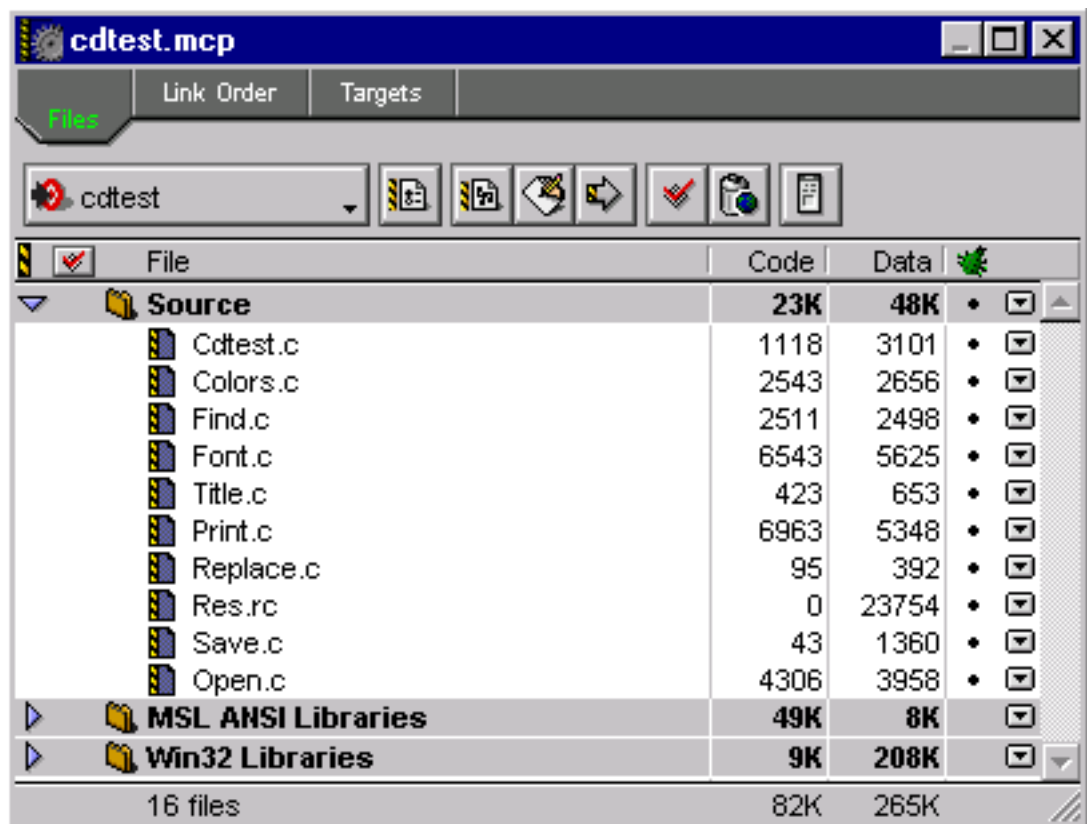
To turn debugging on or off for a file, click in the debug column for that file. Clicking next to a group name turns debugging on or off for all files in the group. If a file cannot be debugged (because it is not a source file) you cannot turn debugging on for that file.

Generating Symbolics Information

To generate symbolics information, both the current target and source files within that target must be prepared for debugging. See Setting Up a Target for Debugging and Setting Up a File for Debugging for information on how to do this.

Once the current target and its source files are prepared, choose the **Make** command from the CodeWarrior IDE's Project menu to compile and link your final code.

For more information on compiling and linking, see the *CodeWarrior IDE User Guide* as well as the Targeting manual for your particular target.

Launching the Debugger

There are two ways you can launch the debugger: directly from the IDE, or separately as you would any other application on your host platform. One or both of these mechanisms is available to you depending upon the chip and operating system you are targeting, and the kind of code you are creating.

You can always launch the debugger as a separate application. When you do, you will typically be asked to locate a symbolics file.

Some targets and projects also allow the debugger to be launched directly from the IDE.

This section discusses the details of launching the debugger using either technique. The topics are:

- Launching MW Debug from the IDE (Mac OS)
- Launching the Debugger Directly

Either process requires a symbolics file. See ["Preparing for Debugging" on page 19](#) for information on how to create a symbolics file.

Using the Integrated Debugger

To run the integrated debugger from within the IDE, you must have debugging enabled. If the debugger is currently disabled, choose **Enable Debugger** before you can choose the **Debug** command.

The IDE's Project menu has an item that toggles between **Run** and **Debug,** depending upon whether the debugger is enabled. If the **Debug** command is enabled, then the IDE can launch the debugger directly for your target and project. The debugger will open the required symbolics file automatically, or ask you to locate it.

The IDE enables the **Debug** command only for targets that generate executable code (such as an application).

Launching MW Debug from the IDE (Mac OS)

To launch the debugger from the IDE, the MW Debug application must be in the same directory as the CodeWarrior IDE, it must already be running in the background, and you need a source file open.

If the debugger is currently disabled, choose **Enable Debugger** before you can choose the **Debug** command

When you enable debugging, the **Run** command on the Project menu changes to **Debug**. This command compiles and links your code, then launches it through the debugger. (See ["Preparing for Debugging" on page 19](#) for more information on this topic.)

Choose **Switch to MW Debugger** to begin debugging your application in the external debugger.

The IDE enables the **Debug** command only for targets that generate executable code (such as an application).

If your current target generates a library or other form of shared code, you can still debug your source files. However, you must

launch the application that uses your code separately, and launch the debugger directly. See <u>"Launching the Debugger Directly" on page 24</u> for more information.

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 symbolics file for the debugger to work with. You can launch the debugger in any of three ways:

- Double-click a symbolics file.
- Double-click the MW Debug icon. You'll see the standard Open File dialog box allowing you to choose a symbolics file.
- Drag and drop a symbolics file onto the MW Debug icon.

Launching the debugger directly is frequently required. There are many targets or kinds of code that the IDE cannot launch. For example, if you are writing an application plug-in, the plug-in cannot run on its own. In addition, the CodeWarrior IDE has no idea what application must be running to invoke the plug-in.

Using the plug-in example, the steps you would follow to debug the plug-in would typically be:

1. **Launch the debugger directly, as described above.**

2. **Open the symbolics file for your plug-in code.**

3. **Set a breakpoint in your code.**

   (For more information, see <u>"Breakpoints" on page 79.</u>)

4. **Launch the application that uses the plug-in.**

5. **Do whatever is necessary in the application to invoke the code in the plug-in.**

   When execution of the plug-in code reaches your breakpoint, the debugger takes control, and you can debug your plug-in code.

Symbolics Files

A project's symbolics file contains information the debugger needs 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 display the source code that corresponds to your object code. When you stop in the debugger to examine what's going on, the debugger shows you the source.

You may also view the corresponding assembly-language instructions and memory addresses. See "Viewing source code as assembly" on page 34.

CodeWarrior supports several different symbolics formats appropriate for a variety of targets. Among the formats supported are:

Format	Principal Target
CodeView	Win32
DWARF	Embedded systems
SYM	Mac OS

See "Preparing for Debugging" on page 19 for information on how to set up a project and source files to create a symbolics file.

See the Codewarrior *IDE User Guide* for more information on generating symbolic information, including information on compiler and linker settings.

For more information on target-specific symbolic information, see the corresponding *Targeting* manual.

3

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
- Watchpoint Window
- Log Window
- Variable Window
- Array Window
- Memory Window
- Register Window
- 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.

The *Program window* (Figure 3.1) displays debugging information about the source-code file containing the currently running routine. It has four primary areas:

- Stack Crawl Pane at the top left
- Variables Pane at the top right
- Debugger Toolbar 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 Variables panes. You can also use the arrow keys or Tab to navigate the items in either of these panes when it is the active pane.

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 37 for details on the contents of the Browser window.

Figure 3.1   Parts of the Program window

To display a routine in the Source Pane, click the name of the routine in the Stack Crawl Pane.

To examine local variables and any referenced global variables, use the Variables Pane.

The Source Pane displays the currently executing source code and any breakpoints.
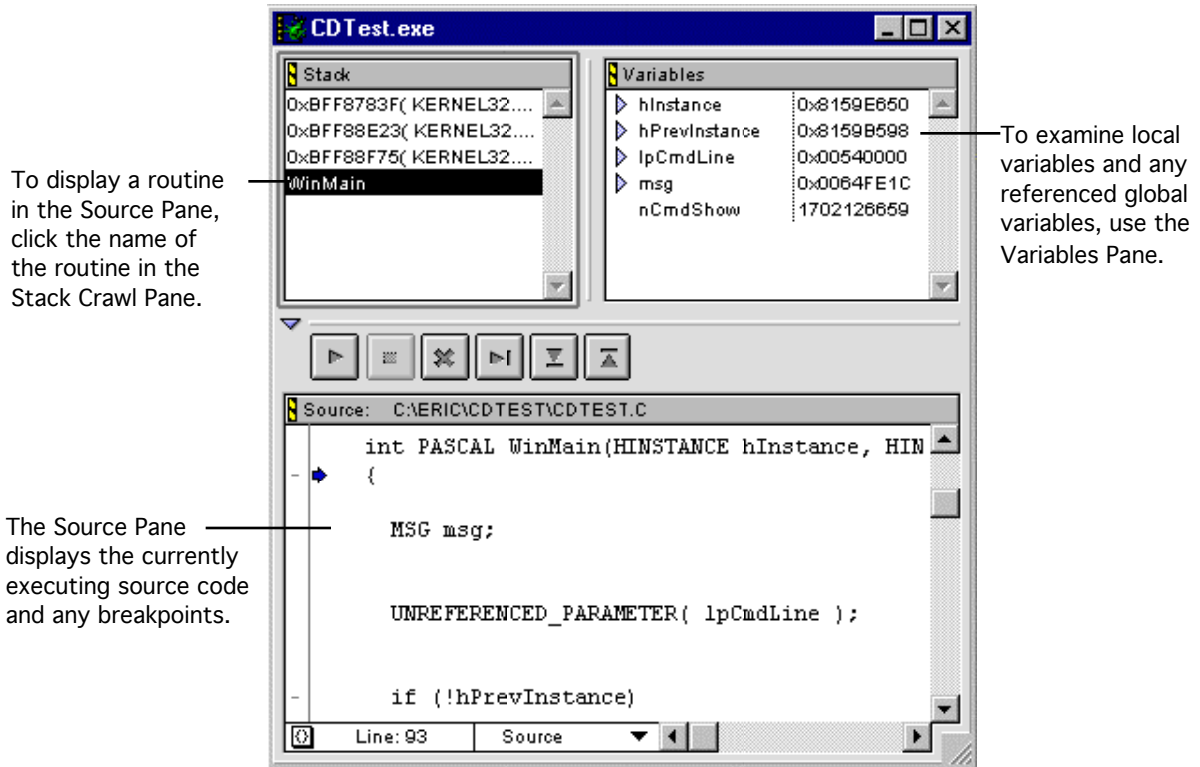
Stack Crawl Pane

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

NewBall() is on display
in the Source Pane. It
was called by main().

To view main(), click its
name in the Stack Crawl
Pane. It will be displayed
in the Source Pane.
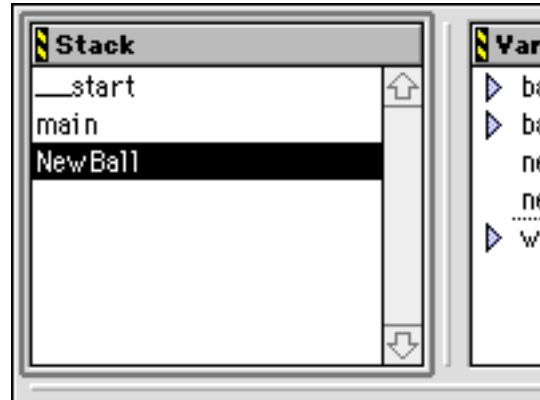
Variables Pane

The *variables pane* ([Figure 3.3](#)) shows a list of the currently executing
routine's local variables.

Mac OS   The variables pane also displays any global variables the
routine refers to. Local and global variables are separated by a
dashed line.

The Variables pane lists the variables in outline form. Click the dis-
closure triangle next to an entry to show or hide the entries inside it.

For example, in [Figure 3.3](#), clicking the disclosure triangle next to
variable msg hides its members. Click the disclosure triangle again
to redisplay the members. You can dereference multiple levels of
pointers to get directly to the data by pressing the Ctrl/Option key
while expanding an entry; this feature is useful for expanding a
handle to a structured type and viewing the structure's members.

See also ["Expand" on page 141](#) and ["Collapse All" on page 141.](#)

Figure 3.3   Variables pane
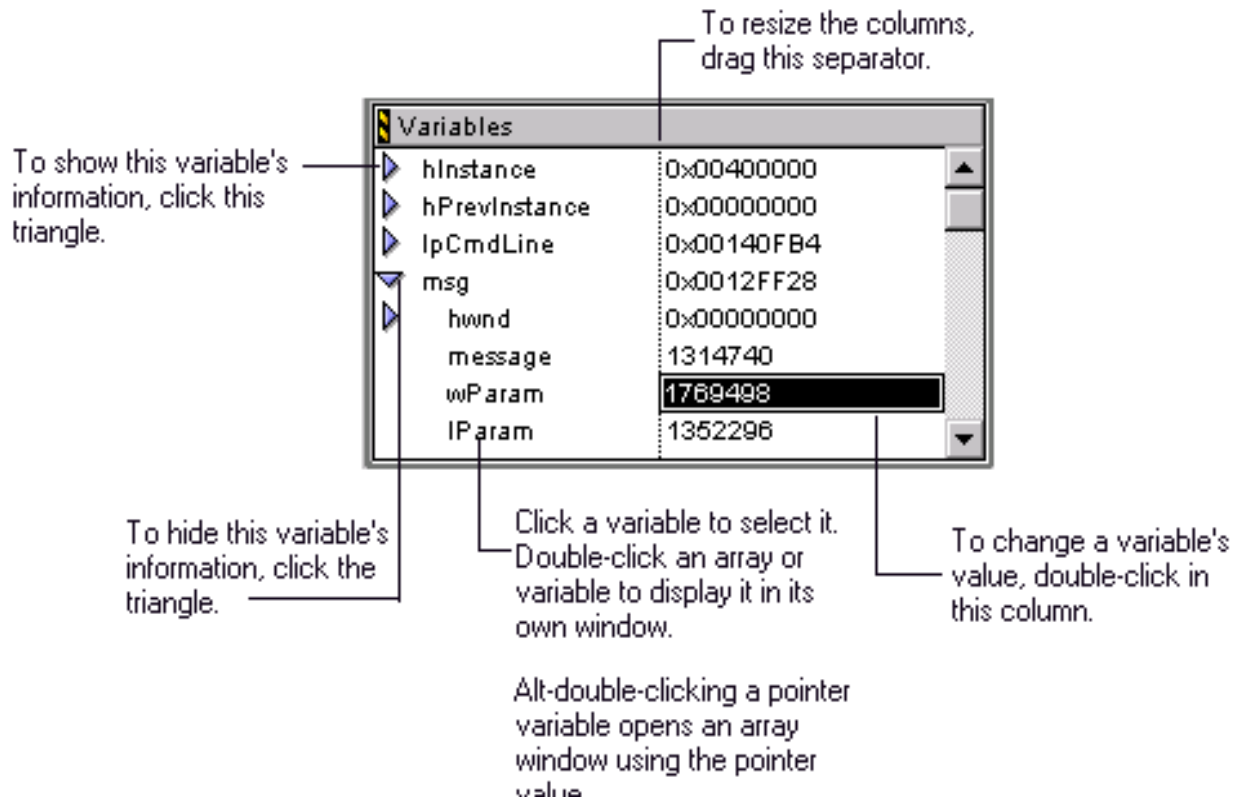
NOTE: If you are viewing assembly code, no register or memory will be displayed in the Variables pane. Instead, use the register and FPU register windows ("Register Window" on page 52) to view the contents of the central-processor and floating-point registers. (Some targets do not have an FPU, and the FPU register window is not available.)

Debugger Toolbar

The control buttons (Figure 3.4) are 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 Process Window Toolbar
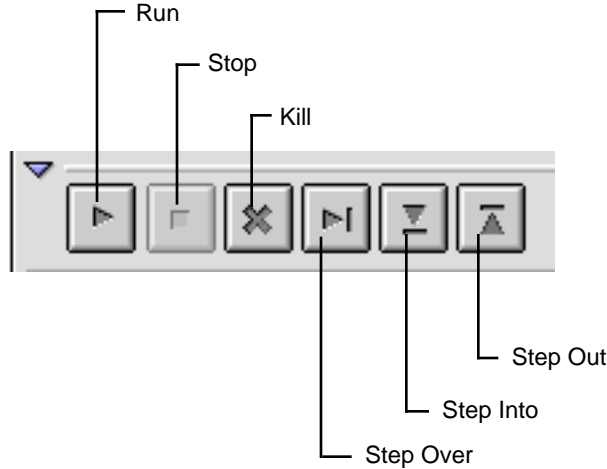
Click the disclosure triangle to the left of the buttons to show or hide them.

Mac OS   Smaller versions of the control buttons are available as a separate toolbar (Figure 3.5). Choose **Show/Hide Toolbar (Mac OS)** from the **Window** menu to display or hide the toolbar.

Figure 3.5   Debugger Toolbar

See also <u>"Basic Debugging Overview" on page 59.</u>

Source Pane

The *source pane* displays the contents of a source-code file. The debugger takes the source code directly from the current target'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 3.6), using the font and color specified in the CodeWarrior IDE's Editor preferences panel.

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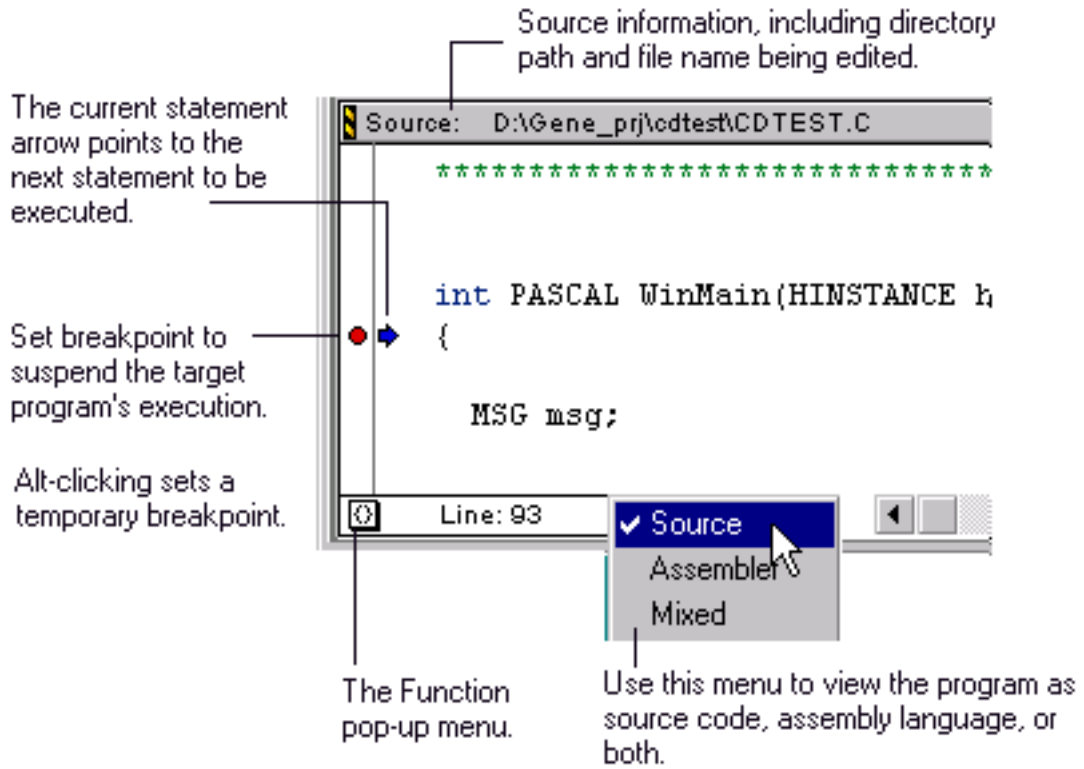
If there are two or more routine calls on a single line, each routine is executed separately as one step before the current-statement arrow moves to the next line. When this happens, the arrow is dimmed whenever the program counter is within, but not at the beginning of, a source-code line.

Figure 3.7   Source and assembly views
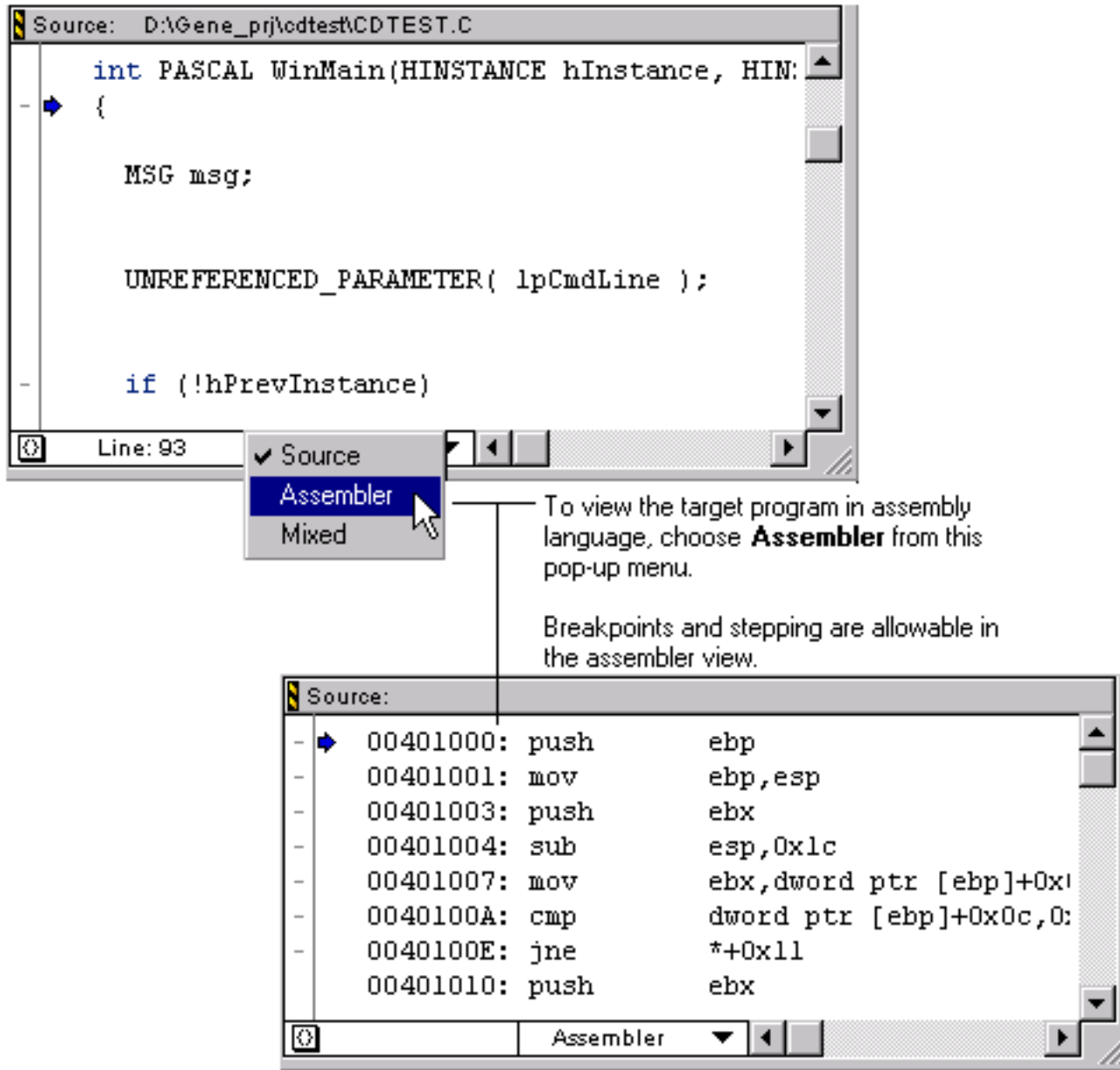
Viewing source code as assembly

To view your source code as assembly language, click the *source pop-up menu* at the bottom of the Program window. Choosing **Assem-**

bler displays the contents of the source pane as assembly code (Figure 3.7). When viewing assembly code, the debugger still lets you step through the code and set breakpoints.


NOTE: If you are viewing assembly code, no register or memory will be displayed. Instead, use the register and FPU register windows ("Register Window" on page 52) to view the contents of the central-processor and floating-point registers. (Some targets do not have an FPU, and the FPU register window is not available.)

Viewing source with mixed assembly

To view your source code and assembly language at the same time, click the *source pop-up menu* at the bottom of the Program window. Choosing **Mixed** displays the source code of the current routine intermixed with assembly code (Figure 3.8). The source that produced the assembly instructions appears before the assembly itself. When viewing code in the mixed view, the debugger makes the assembly code "live." This means you can set breakpoints and step through source code, but only for assembly language instructions. Notice you cannot set breakpoints on source lines as shown in Figure 3.8.

Figure 3.8   Viewing mixed code
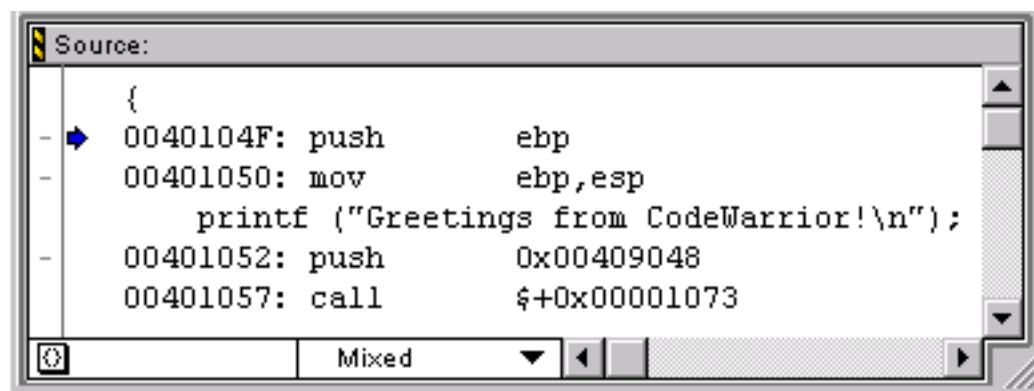
If no source is available for the code, then the display reverts to Assembly. There is no syntax hiliting for this view, so all the text appears plain.

Viewing source with mixed assembly is not available for Java.

Figure 3.9   Function pop-up menu
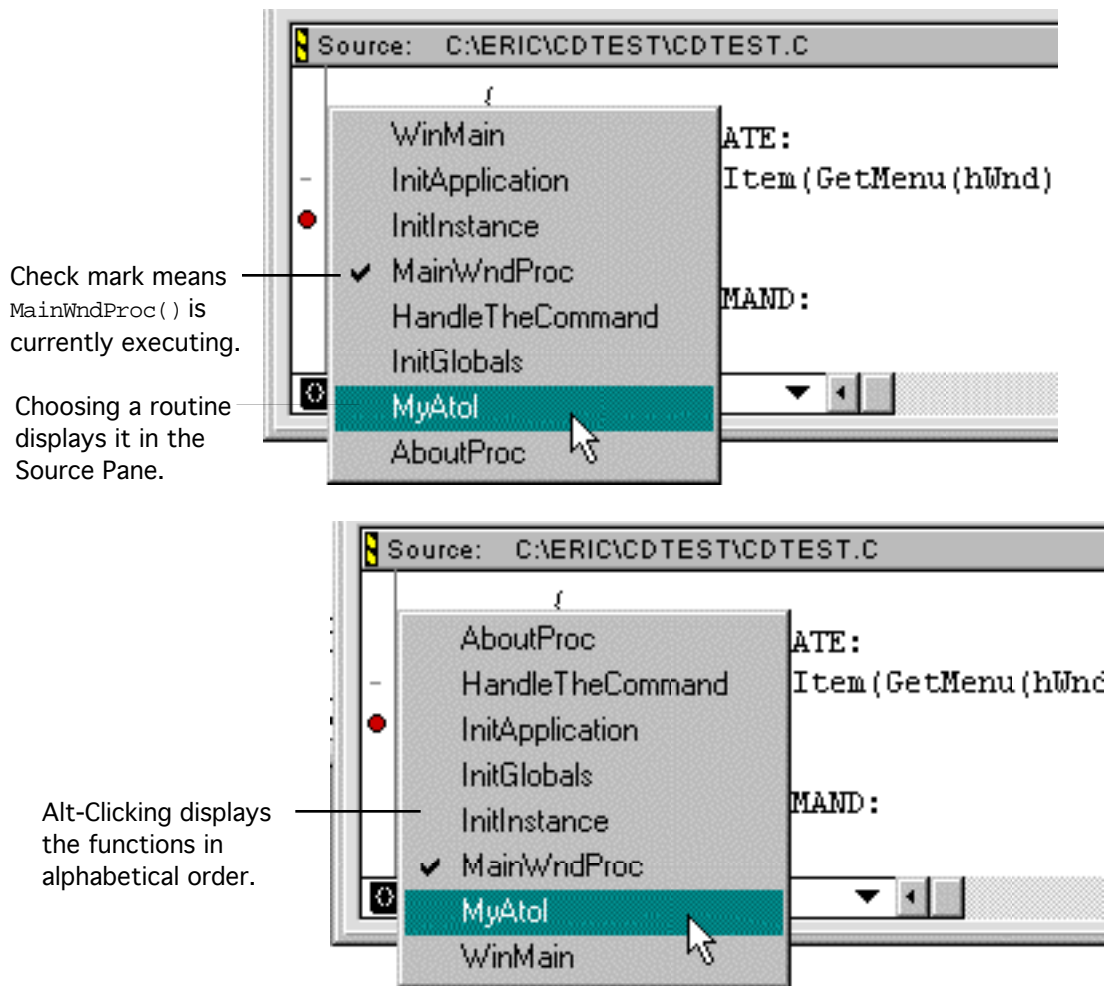
Check mark means
`MainWndProc()` is
currently executing.

Choosing a routine
displays it in the
Source Pane.

Alt-Clicking displays
the functions in
alphabetical order.

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 (). Selecting a routine in the function menu displays it in the source pane.

Press the Alt/Option key, then click the Function menu to display the menu sorted alphabetically.

Browser Window

The *Browser window* ([Figure 3.10](#)) somewhat resembles the Program window in both appearance and functionality, but displays different information. The Browser window lets you view any file in the current target, 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 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
- *Browser Source Pane* on 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 28 for details on the contents of the Program window.

Figure 3.10   Browser window

Click a file name to display the file's routines in the Function Pane. Click Global Variables to display all global variables in the Globals Pane.

Click a routine in the Function Pane to display it in the Source Pane.

View global and static variables in the Globals Pane.

The Source Pane displays the source code of the routine selected in the Function Pane.
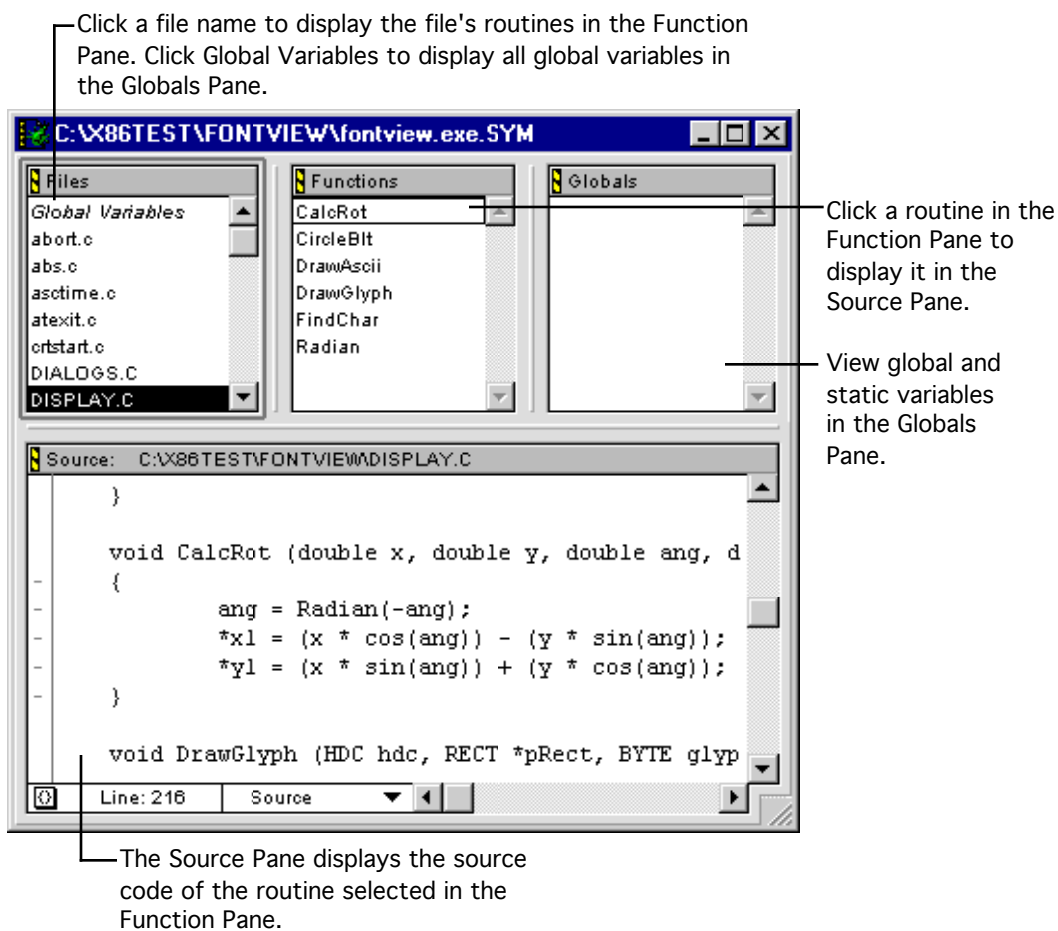
File Pane

The *file pane* in the Browser window (Figure 3.11) displays a list of all source files associated with the current target 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.

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 40 and "Breakpoints" on page 79.

Figure 3.11   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 SillyBalls.c, is displayed in the source pane.
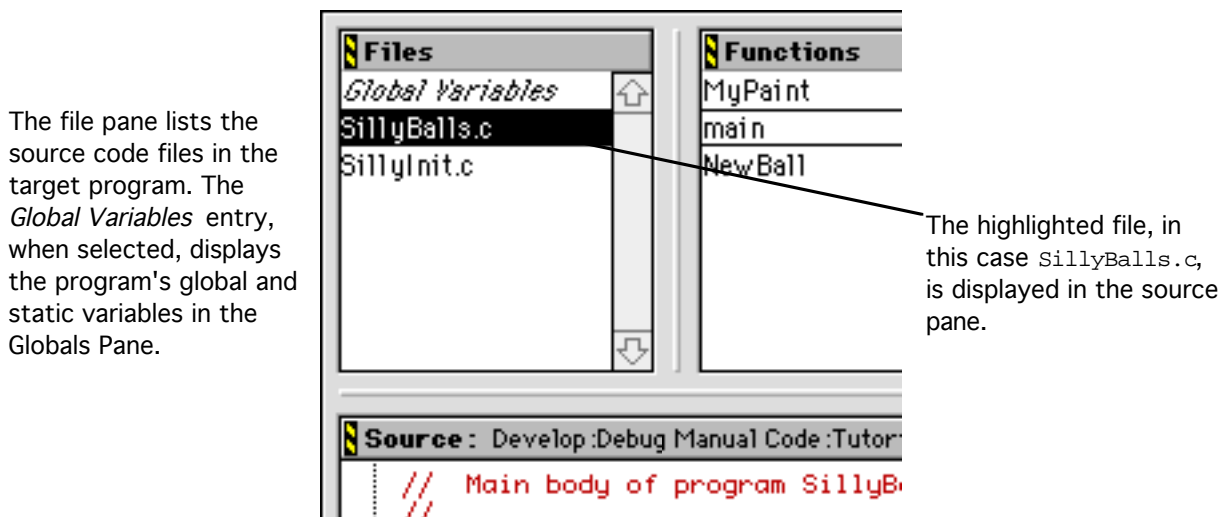
Function Pane

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 3.12   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.
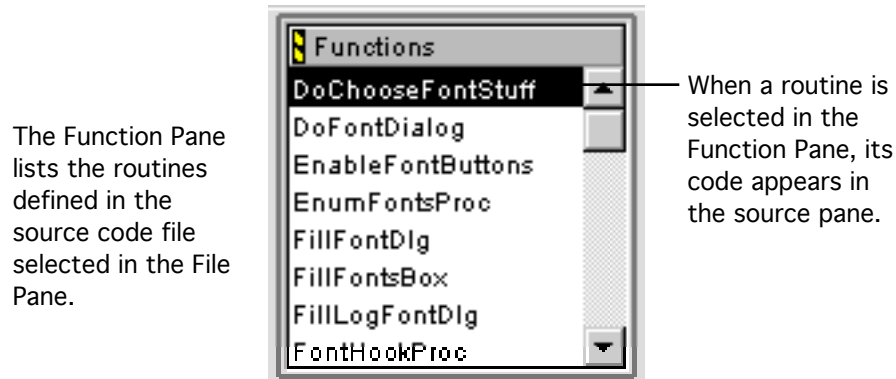
NOTE If your code is written in C++ or Object Pascal, the **Sort functions by method name in browser** option in the **Preferences** dialog (see "Settings" on page 118) alphabetizes function names of the form *className::methodName* by method name instead of by class name. Since most C++ source files 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 3.13). 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 3.13   Globals pane

To resize column widths,
drag this separator.

Click a variable to select it.
Double-click an array or
variable to display it in its
own window.

Alt-double-click a pointer
variable to open an array
window using the pointer
values.

```
 Globals
 ▷  ccThread2        0x0040F0DE ▲
 ▽  cf               0x0040DB82
      lStructSize     0
 ▷    hwndOwner       0x00000000
 ▷    hDC             0x00000000
 ▷    lpLogFont       0x00000000
      iPointSize      0
      Flags           0          ▼
```

Placing globals in a separate window

To display a global variable in its own window, double-click the
variable's name in the globals pane; a new variable window will ap-
pear 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 Win-
dow** command from the Data menu. A global displayed in its own
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 48, "Array Window" on page
48, and "Expression Window" on page 44.

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 Win-
dows** command from the Window menu.

See also "Close All Variable Windows" on page 147.

Browser Source Pane

The *browser source pane* allows you to browse the contents of the source-code file selected in the file pane (Figure 3.14). You can use it to set breakpoints in any file listed in the file pane. Notice, however, that the source browser pane does not show the currently executing statement; to view the current statement or local variables, use the Program window instead.

The source pane displays code in the font and colors specified in CodeWarrior IDE Editor's preferences panel. 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 34). Choose **Assembler** to display the contents of the source pane as assembly code, as shown earlier in Figure 3.7. You can set breakpoints in assembly code, just as you can in source code. Choose **Mixed** to display source code intermixed with assembly language, as shown earlier in Figure 3.8.

See also "Source Pane" on page 32, "Changing Font and Color" on page 79, and "Breakpoints" on page 79.

Figure 3.14   Browser source pane

The source pane will display the message "Source text or disassembly not available" when a library file is selected.

The breakpoint column lets you set breakpoints that suspend the target program's execution.

```
Source:   C:\ERIC\CDTEST\FONT.C

    void DoChooseFontStuff(HWND hwnd, LPCHOOSEFONT pcf)
-   {
-      BOOL bRet = FALSE ;

-      if (IsDlgButtonChecked(hwnd, ID_PRELOADEDFONT) ==
       {
-          pcf->hInstance = GetFontDlgHandle() ;
-          wsprintf(szTemp, szLongFilter, pcf->hInstance) ;
   Line: 1392    Source    ▼
```

Clicking this icon displays a list of routines defined in the file selected in the File Pane.

Click this menu to switch between source and assembly language views.

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.

Press the Alt/Option key, then click the Function menu to display the menu sorted alphabetically as shown earlier in <u>Figure 3.9</u>.

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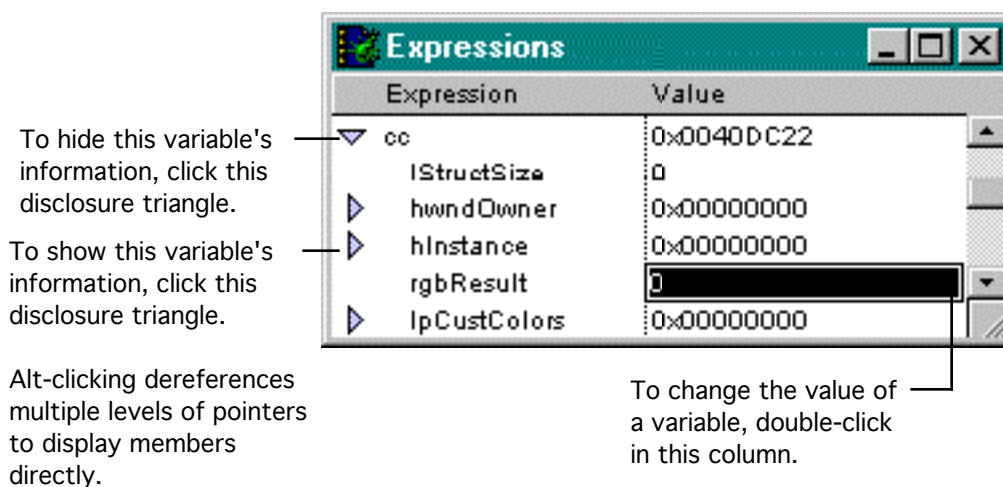

Expression Window

The *expression window* ([Figure 3.15](#)) 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.

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.

Figure 3.15   Expression window

To hide this variable's information, click this disclosure triangle.

To show this variable's information, click this disclosure triangle.

Alt-clicking dereferences multiple levels of pointers to display members directly.

To change the value of a variable, double-click in this column.

To remove an item from the expression window, select the item and press the Backspace/Delete 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 147, "Copy to Expression" on page 142, and "Using the Expression Window" on page 97.

Breakpoint Window

The *breakpoint window* (Figure 3.16) lists all breakpoints in your current target, 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 3.16   **Breakpoint window**
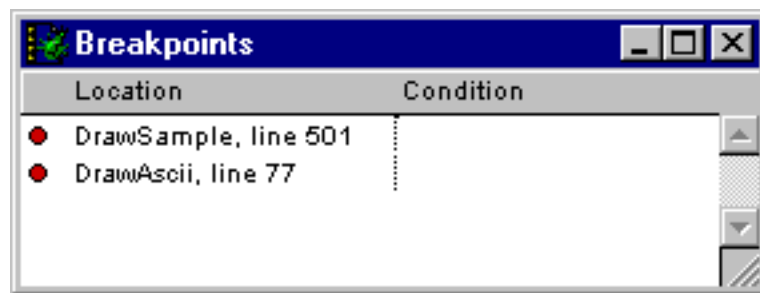
There is a breakpoint marker to the left of each listing. A circle indicates that the breakpoint is active, a dash that it is inactive. Clicking a breakpoint marker toggles the breakpoint on or off while remembering its position in the target program. Double-clicking a breakpoint listing activates the 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 79, "Show/Hide Breakpoints" on page 147, and "Conditional Breakpoints" on page 82.

Watchpoint Window

The *watchpoint window* (Figure 3.17) lists all watchpoints in your current target by memory address. To open the watchpoint window, choose **Show Watchpoints** from the **Window** menu. When the watchpoint window is open, the command changes to **Hide Watchpoints** and is used to close the window. (You can, of course, also close the watchpoint window by simply clicking its close box.)

Figure 3.17   Watchpoint window
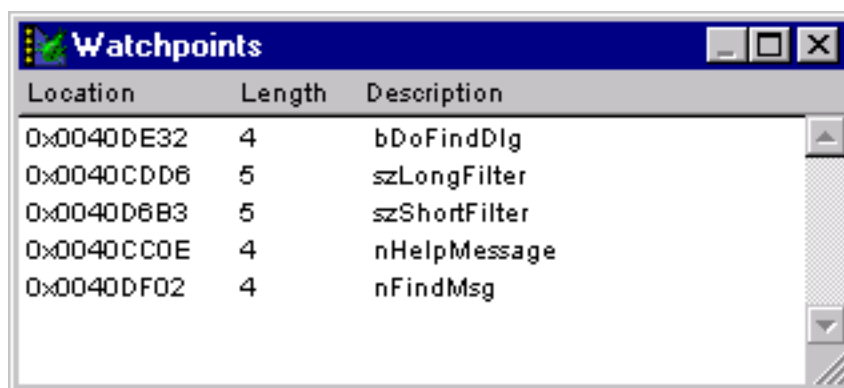
You can clear a watchpoint by selecting it with the mouse and doing any of the following:

- Choose **Clear Watchpoint** from the Data menu.
- Choose **Clear** from the Edit menu.
- Press the Backspace/Delete key.

See also "Watchpoints" on page 86 and "Show/Hide Watchpoints" on page 147.

Log Window

The *Log window* ([Figure 3.18](#)) displays messages as your program makes calls to system DLL's or starts new tasks.

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 3.18   Log window
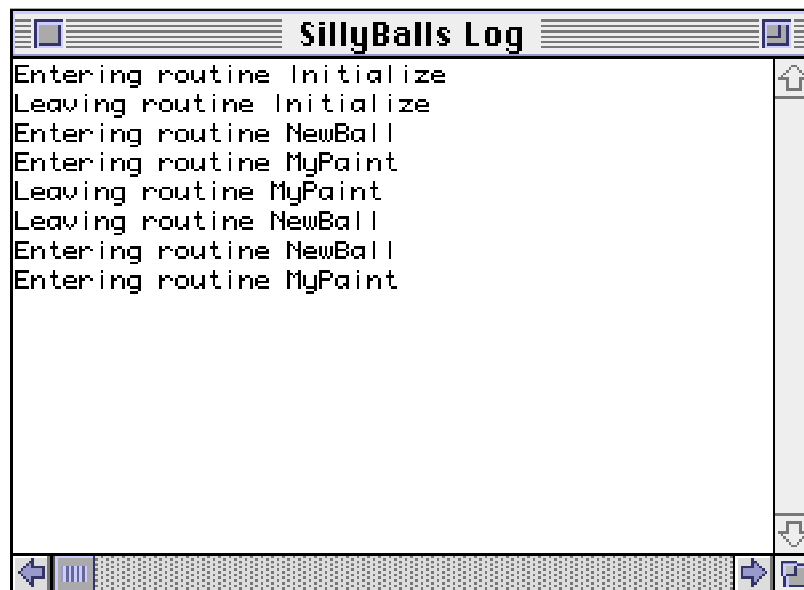
Variable Window

A *variable window* ([Figure 3.19](#)) 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 3.19   A variable window

To change a variable's value, double-click in this column.

Array Window

An *array window* ([Figure 3.20](#)) 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 infor-

mation pane shows more information about the array. From the ex-
panded 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.

Figure 3.20   Anatomy of an array window
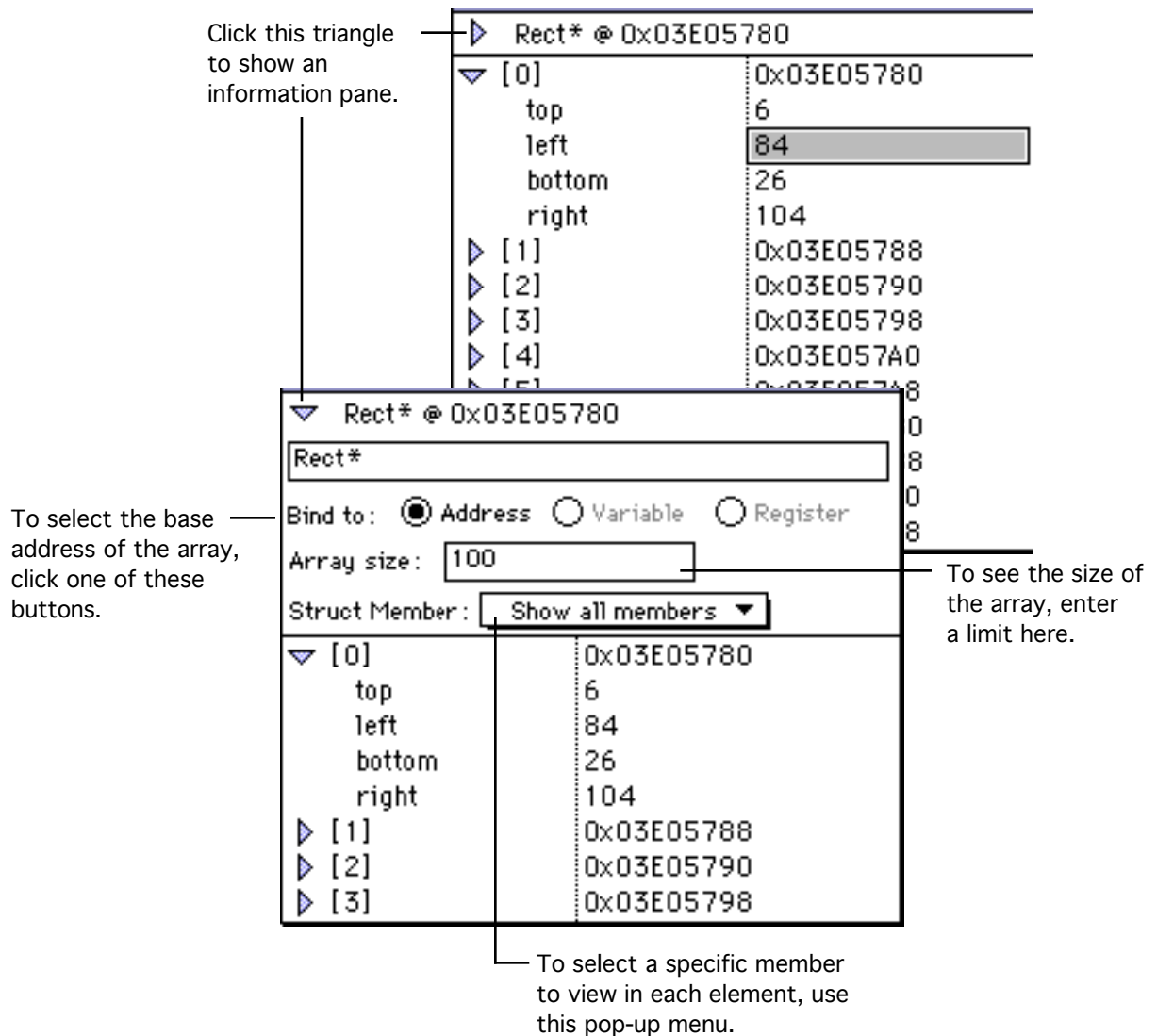
The array's contents are listed sequentially, starting at element 0. If
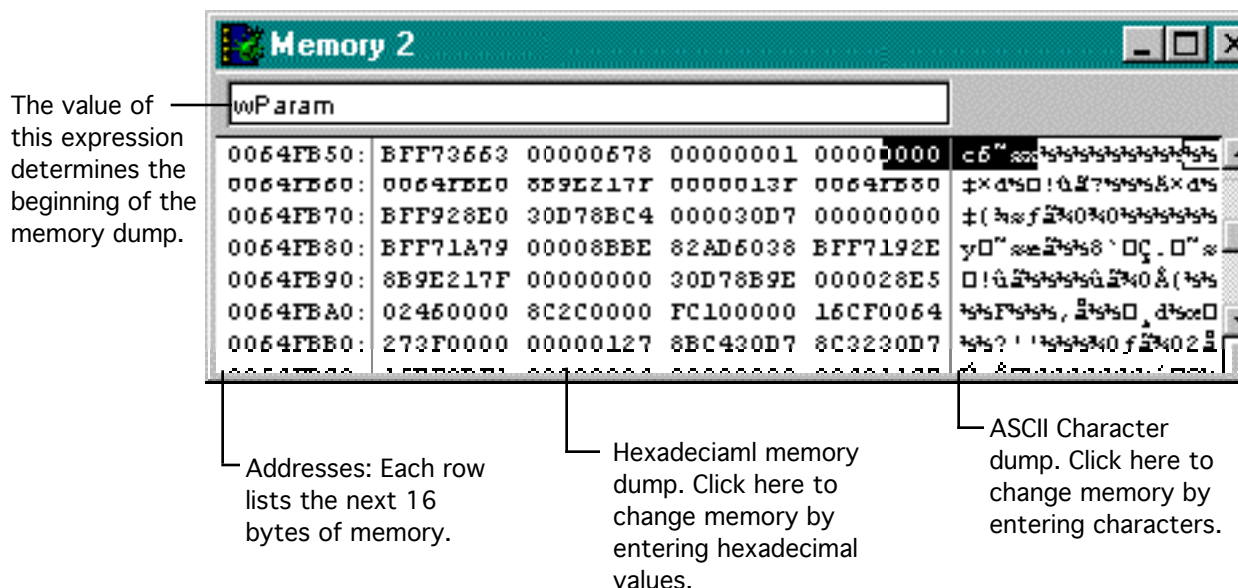array elements are cast as structured types, an arrow appears to the

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

See also <u>"Open Array Window" on page 142</u> and <u>"View Memory As" on page 144.</u>

Memory Window

A *memory window* displays the contents of memory in hexadecimal and corresponding ASCII character values ([Figure 3.21]). To open a memory window, select a variable, routine name, or expression representing the desired base address in the program, browser, or expression window and choose **View Memory** from the Data menu. To close the memory window, click its close box.

Figure 3.21   A memory window

The value of this expression determines the beginning of the memory dump.

Addresses: Each row lists the next 16 bytes of memory.

Hexadeciaml memory dump. Click here to change memory by entering hexadecimal values.

ASCII Character dump. Click here to change memory by entering characters.

NOTE The **View Memory as** command opens an array window (<u>"Array Window" on page 48</u>) displaying memory as an array of data of a type you specify.

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.

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/Delete, Tab, Enter, and so forth) do not work. New data you type overwrites what is already in memory.

Mac OS

If the expression is a pointer-sized register, then the register's contents are used as the base address. For example,

```
®A0
```

will show memory beginning at the address contained in the 68K register A0. Floating-point registers cannot be used.

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 Window

A *register window* displays CPU registers (Figure 3.22) 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.)

NOTE: The appearance of the Register window will change depending upon the target processor.

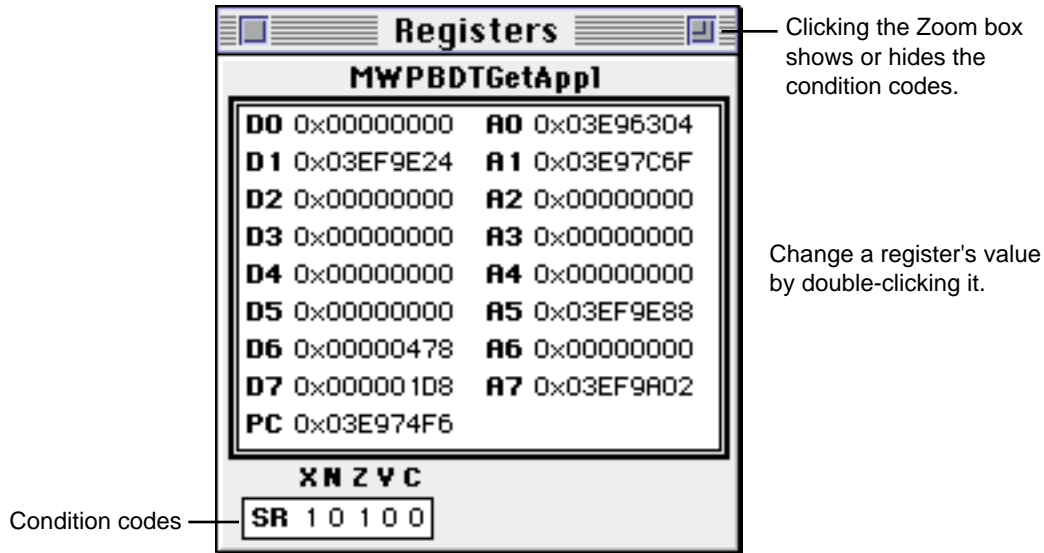**Figure 3.22   A CPU register window**
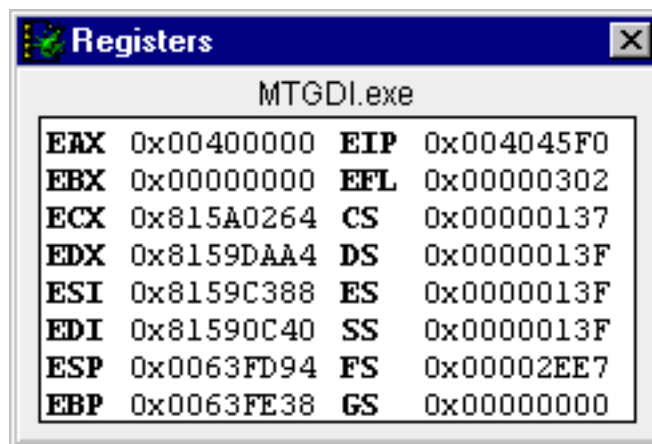
Double-click a registers value to change it.

For some targets, an FPU (floating -point unit) register is also available. If an FPU is available, you can display its registers as well. Choose **Show FPU Registers** from the **Window** menu. When a register window is open, the corresponding command changes to **Hide FPU Registers** and is used to close the window.

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

Figure 3.23   A CPU register window (Mac OS)

Clicking the Zoom box shows or hides the condition codes.

Change a register's value by double-clicking it.

Condition codes

Mac OS   Click a register window's zoom box to display the full set of registers. Toggle status and condition registers between 0 and 1 by double-clicking, or by selecting the register and pressing Return or Enter.

NOTE: Changing the value of a register is a very dangerous thing to do. It could corrupt your data, memory, or cause a crash.
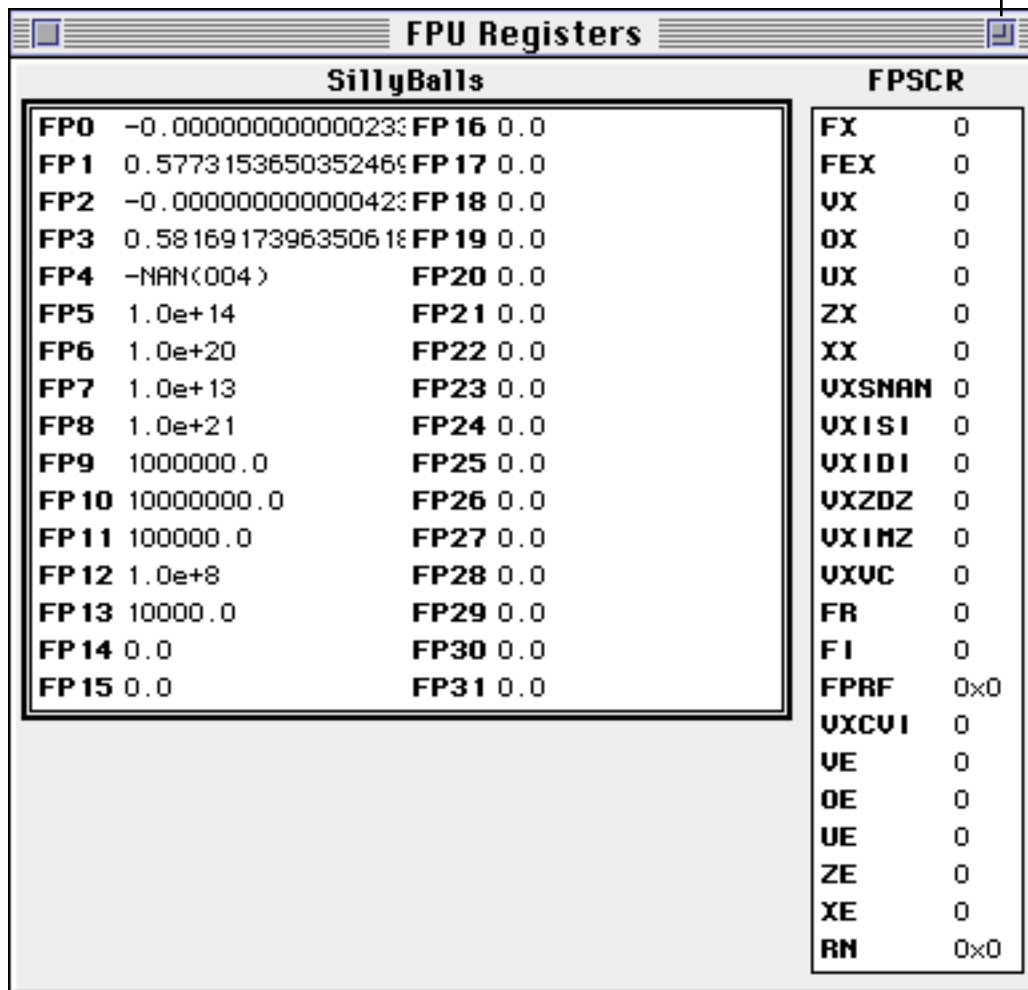
Figure 3.24   An FPU register window (Mac OS)

Clicking the zoom box alternates the window
between this full view and a small view.

FPU Registers	
SillyBalls	**FPSCR**

	SillyBalls		FPSCR	
FP0	-0.00000000000023	FP16 0.0	FX	0
FP1	0.5773153650352469	FP17 0.0	FEX	0
FP2	-0.00000000000042	FP18 0.0	VX	0
FP3	0.5816917396350618	FP19 0.0	OX	0
FP4	-NAN(004)	FP20 0.0	UX	0
FP5	1.0e+14	FP21 0.0	ZX	0
FP6	1.0e+20	FP22 0.0	XX	0
FP7	1.0e+13	FP23 0.0	VXSNAN	0
FP8	1.0e+21	FP24 0.0	VXISI	0
FP9	1000000.0	FP25 0.0	VXIDI	0
FP10	10000000.0	FP26 0.0	VXZDZ	0
FP11	100000.0	FP27 0.0	VXIMZ	0
FP12	1.0e+8	FP28 0.0	VXVC	0
FP13	10000.0	FP29 0.0	FR	0
FP14	0.0	FP30 0.0	FI	0
FP15	0.0	FP31 0.0	FPRF	0x0
			VXCVI	0
			VE	0
			OE	0
			UE	0
			ZE	0
			XE	0
			RN	0x0

See also "Show / Hide Registers" on page 147 and "Show / Hide FPU Registers" on page 147.

Process Window

The *Process window* ([Figure 3.25](#)) lists processes currently running including some hidden processes. The process window also lists tasks 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.)

Figure 3.25   Process window
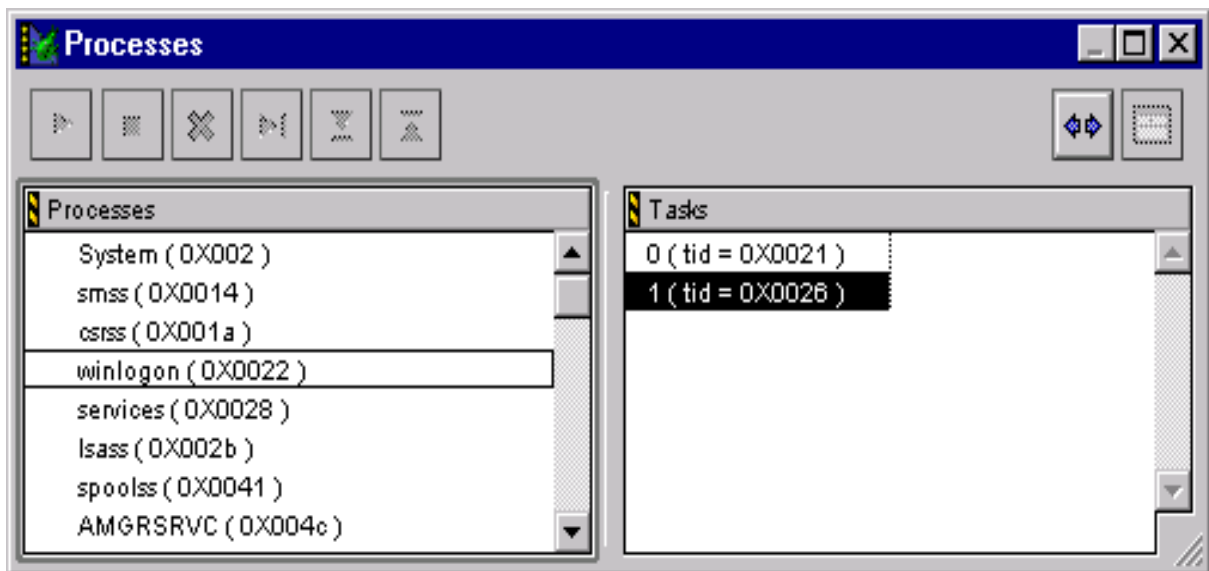
The Process window has two panes and a tool bar:

- [Process Pane](#)—displays currently running processes
- [Tasks Pane](#)—displays tasks running in the selected process
- [Process Window Toolbar](#)—allows to run, stop, or kill processes and tasks under the debugger's control

**See also** ["Show/Hide Processes" on page 146.](#)

Process Pane

The Process pane lists all active processes. A process under the debugger's control has a checkmark next to its entry in the window. To set debugger control for a process, click the checkmark column. Double-clicking a process name activates that process.

> **TIP:** If you turn on debugger control for a process, you can untarget the process without killing it. To untarget the process, simply click the checkmark column again and click the **Resume** button in the subsequent dialog.

Figure 3.26   Process pane
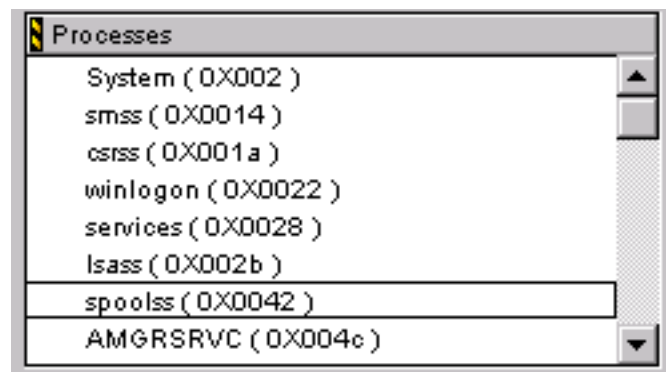
Tasks Pane

The tasks pane lists all the active tasks for a given process. Only tasks from the program under the debugger's control will be shown. Double-clicking a task name activates a Program window with the code for that task. You can also choose a task and then use the Program window button in the top right corner.

There are two columns in the task pane. The first column displays the task ID. The second columns shows the task state. A task can be either running, stopped, or crashed.

**Figure 3.27   Tasks pane**
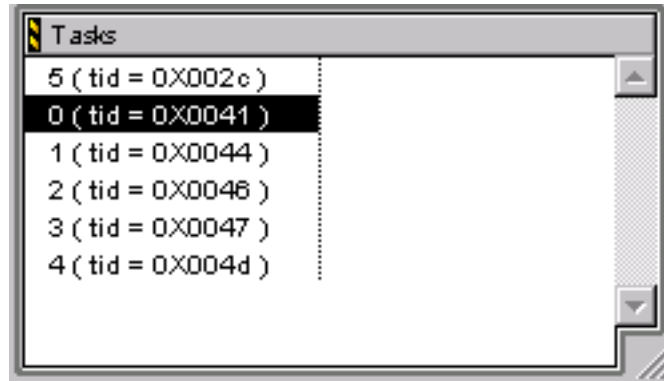
Process Window Toolbar

The Process Window Toolbar ([Figure 3.28](#)) has controls to **Run**, **Stop**, and **Kill** a process under the debugger's control. These controls have no effect on any other active processes or tasks.

The **Step Over**, **Step Into**, and **Step Out** buttons work the same as they do in the Program window. Clicking any of these three buttons will activate a Program window showing the current statement arrow.

Figure 3.28   Process Window Toolbar
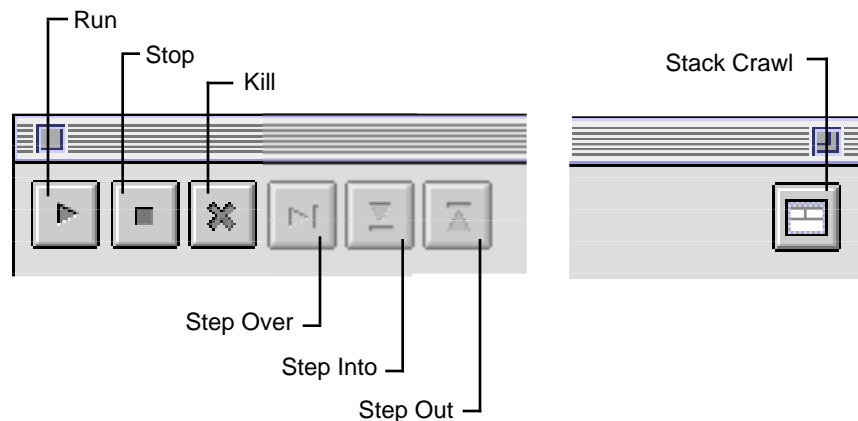
The Program window button will show the Program window for the selected process or task. If a process is selected, the Program window for that process is brought to the front. If a task is selected, the Program window for that task is brought to the front. You can have multiple Program windows open at a time.

See also "Debugger Toolbar" on page 31.

4

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
- Watchpoints—stopping execution when the contents of a memory location are changed
- Viewing and Changing Data—seeing your variables and modifying them at will
- Editing Source Code—editing source code while in a debugger session.

To learn how to prepare a target for debugging or launch the debugger, see *"Getting Started Overview" on page 19.* This chapter also assumes you are familiar with the information about the debugger interface found in *"What You See Overview" on page 27.* For information on how to set the debugger's preferences, see *"Preferences" on page 138.*

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 target under debugger control, brings the program window to the foreground, and stops the program at the first line.

Figure 4.1   Where is file?

Figure 4.2   Where is file? (Mac OS)
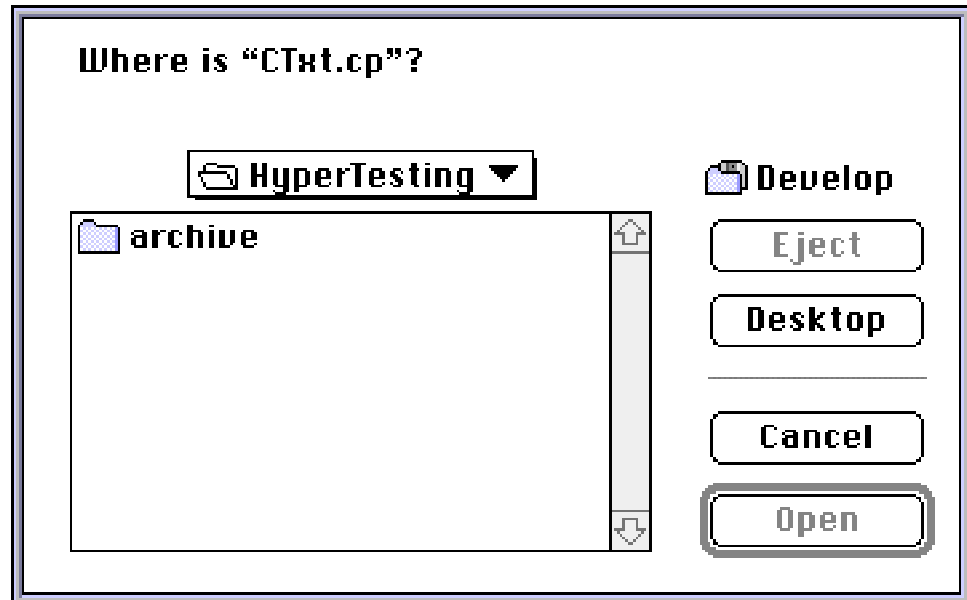
Occasionally, the debugger may ask you for the location of a partic-ular file. The dialog, shown in Figure 4.1, appears.

You may see this dialog either upon startup (the debugger is look-ing for the file with the main entry point) or by clicking on a specific file in the Browser Window.

This can happen under the following situations:

- a file has been moved to a different directory
- you've received the project from another person on your team and the paths are different
- you've selected a file belonging to a compiled library and you do not have the source files

The last case is most common if a library you are using in your tar-get has been compiled with debug symbols turned on. This is espe-cially true with some of the libraries distributed with CodeWarrior.

Once you have found the file, the debugger will remember the file location, even between debug sessions.

See also <u>"Launching MW Debug from the IDE (Mac OS)" on page 23</u> and <u>"Launching the Debugger Directly" on page 24.</u>

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 target 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 control buttons, keyboard, or choose the appropriate command from the Control menu. <u>Table 4.1</u> lists the control buttons along with their menu and keyboard equivalents.

Table 4.1   Button and Key commands

Button	Menu Command	Keyboard Equivalent
▷	Run	Ctrl/Command R
▣	Stop	Ctrl/Command . (period)
✖	Kill	Ctrl/Command K
▶❘	Step Over	Ctrl/Command S

Button	Menu Command	Keyboard Equivalent
⊼	Step Into	Ctrl/Command T
⊻	Step Out	Ctrl/Command U

This section discusses:

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

Figure 4.3   The current-statement arrow

The current-statement arrow points to the next statement to execute.
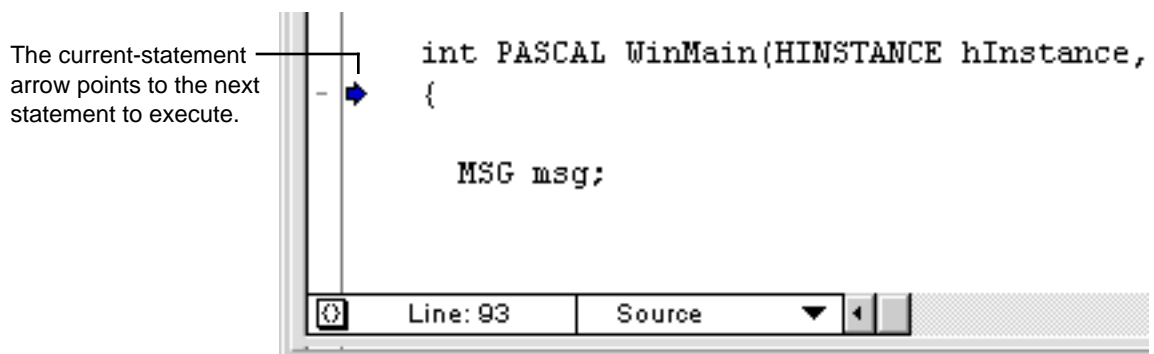
Current-Statement Arrow

The *current-statement arrow* in the program window (Figure 4.3) 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.
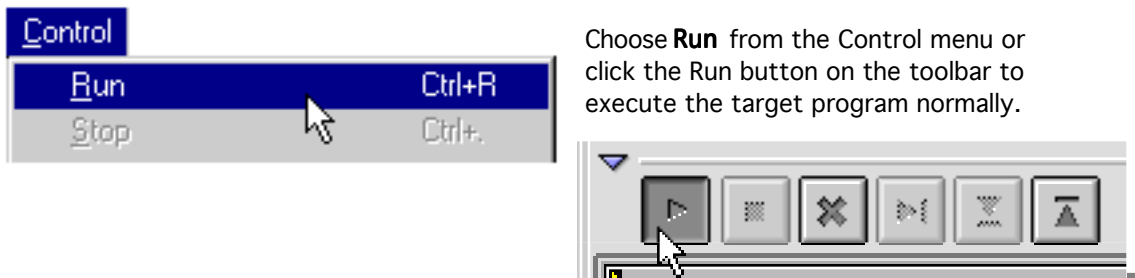
Running Your Code

If the target has been launched but execution has been stopped, use the **Run** command ([Figure 4.4](#)) to restart your program. When you do, the program resumes execution at the current-statement arrow.

Figure 4.4   The Run command

Choose **Run** from the Control menu or click the Run button on the toolbar to execute the target program normally.

If the target 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 target 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 4.5](#)). 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/Option key while opening the symbolics file.You can also change the **Automatically launch applications when SYM file opened** preference (see "Program Control" on page 124). One use for this feature is to debug C++ static constructors, which are executed before entering the program's main routine.

Figure 4.5   Starting the execution of the target program

When a program is first launched from the debugger, execution automatically stops at the program's main entry point.
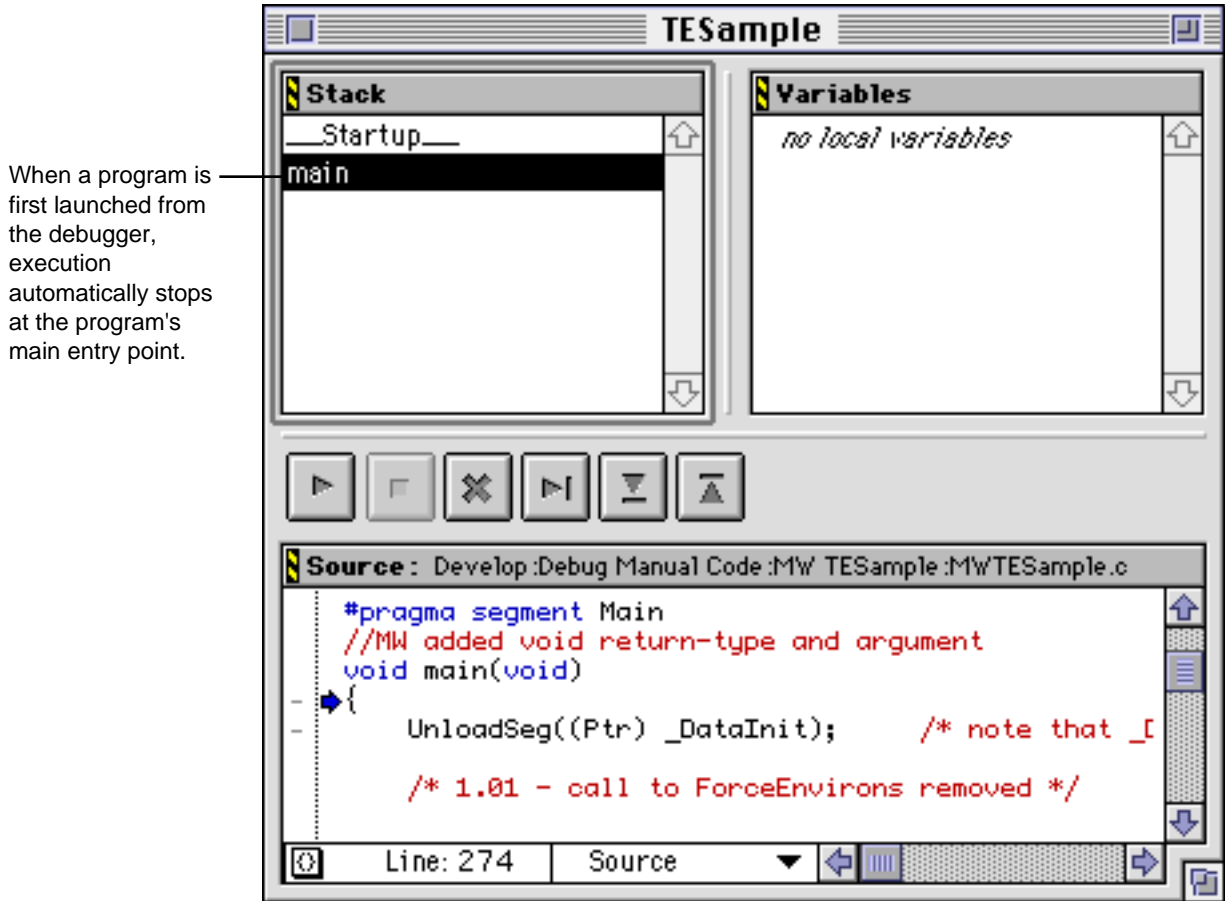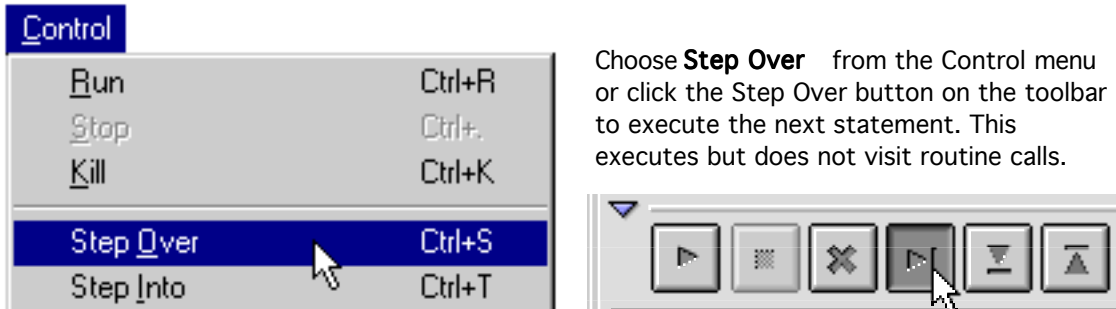

Stepping a Single Line

To execute one statement, use the **Step Over** command ([Figure 4.6](#)). 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 4.6   The Step Over command

Choose **Step Over** from the Control menu or click the Step Over button on the toolbar to execute the next statement. This executes but does not visit routine calls.
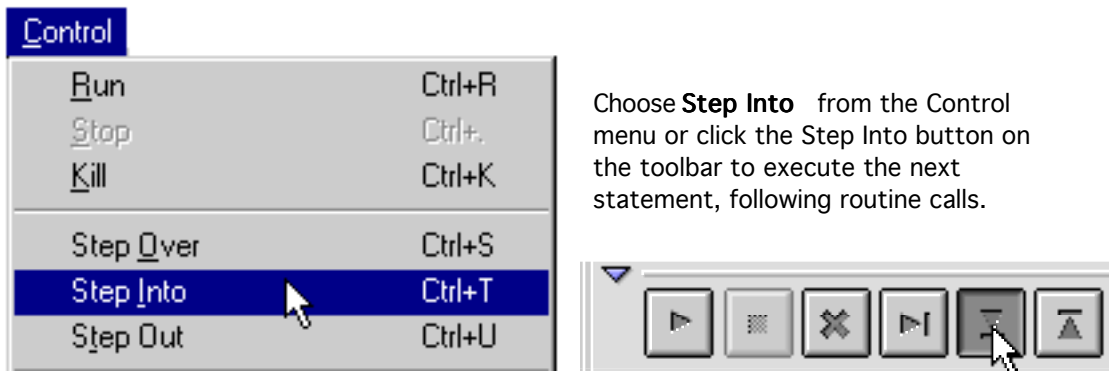
Stepping Into Routines

Sometimes you want to follow execution into a called routine. This is known as *tracing* code). To execute one statement at a time and follow execution into a routine call, use the **Step Into** command (Figure 4.7).

Figure 4.7   The Step Into command

Choose **Step Into** from the Control menu or click the Step Into button on the toolbar to execute the next statement, following routine calls.
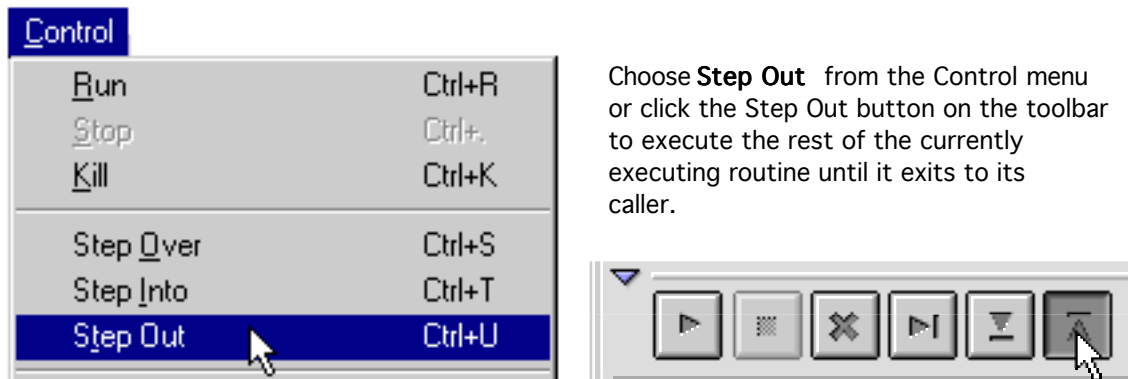
Step Into moves the current-statement arrow down one statement, unless the current statement contains a routine call. When **Step Into** reaches a routine call, it follows execution into the routine being called.

Stepping Out of Routines

To execute statements until the current routine returns to its caller, use the **Step Out** command (Figure 4.8). **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. See "Call-Chain Navigation" on page 71.

Figure 4.8 **The Step Out command**

Choose **Step Out** from the Control menu or click the Step Out button on the toolbar to execute the rest of the currently executing routine until it exits to its caller.

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 4.9). 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.
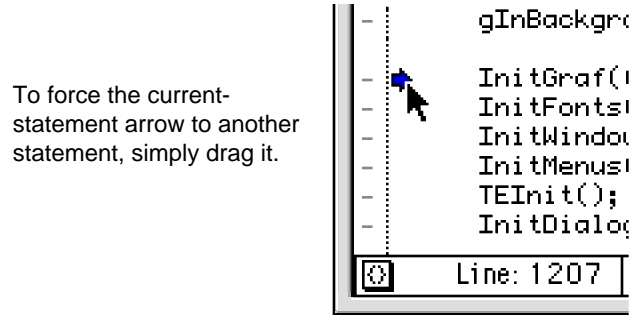
Figure 4.9   Dragging the current-statement arrow

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

To move the current-statement arrow without potentially corrupt-
ing the run-time environment, Alt/Option click a statement in the
breakpoint column (Figure 4.10). Alt/Option and clicking sets a
temporary breakpoint: Execution proceeds normally until the cur-
rent-statement arrow reaches the temporary breakpoint, then stops.
(See "Breakpoints" on page 79.)

Figure 4.10   Setting a temporary breakpoint

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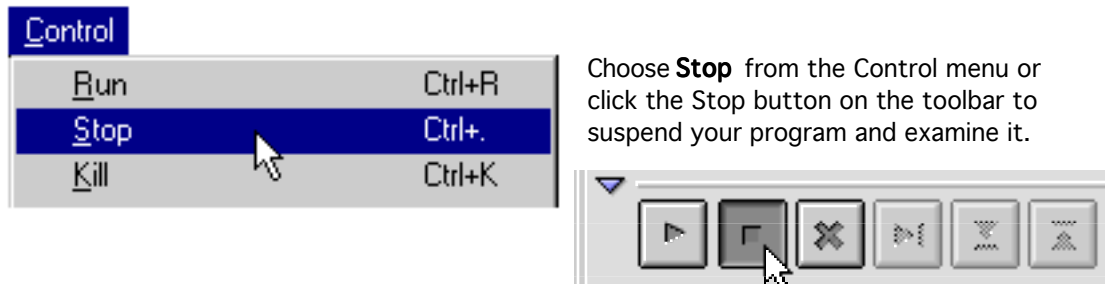in the breakpoint column.
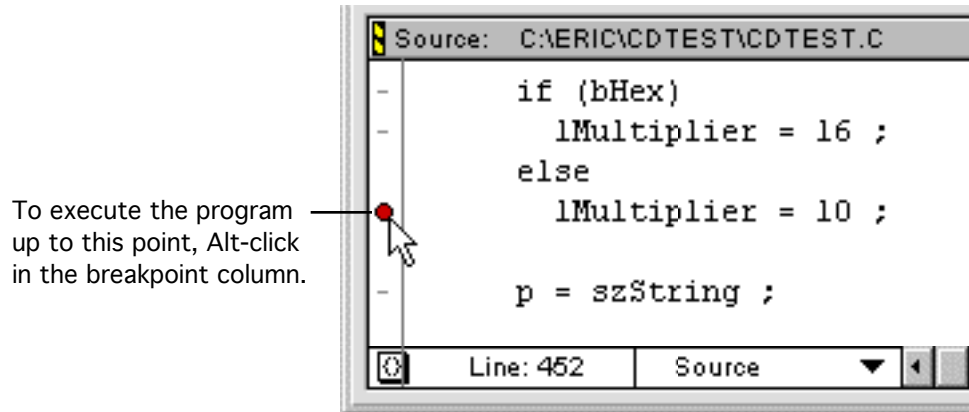
Stopping Execution

While your program is running, you may wish to use the **Stop** com-
mand (Figure 4.11) to suspend execution and explore with the de-
bugger. This stops execution at some point where the operating sys-
tem surrenders control to other processes such as the host debugger.

You can then step through your code from that point, or use the **Run** command to resume execution.

Figure 4.11   The Stop command

Choose **Stop** from the Control menu or click the Stop button on the toolbar to suspend your program and examine it.

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)

NOTE: The Stop command is not available for some targets because it is dependent on operating system services. For details on any particular target, see the corresponding Targeting manual.

TIP: (Mac OS) If your program hangs in an infinite loop, you can regain control by typing the combination Command-Control- / from your keyboard. This will interrupt the program and put you in the debugger so you can try to figure out what's going on.

TIP: (Win 95/NT) 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 4.12](#)) 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 4.12   The Kill command

Choose **Kill** from the Control menu or click the Kill button on the toolbar to stop your program.
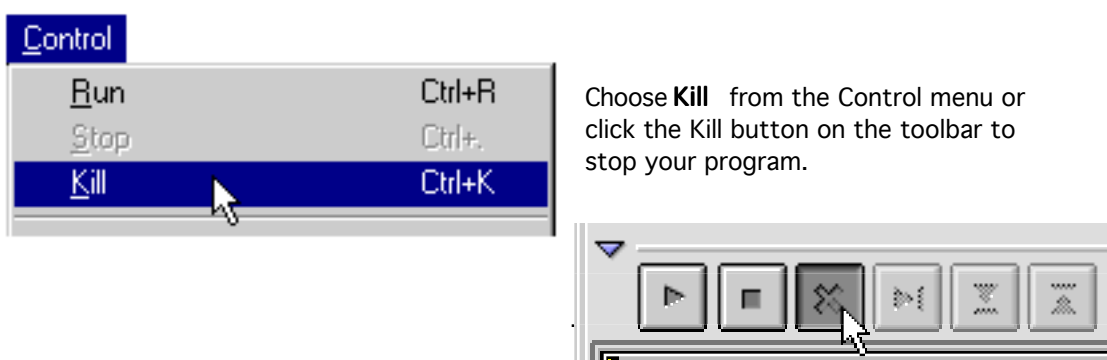
Killing the program is not the same as stopping. Stopping only suspends execution temporarily: you can resume from the point at which you stopped. Killing permanently terminates the program.

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 though code
- [Call-Chain Navigation](#)—moving to active routines
- [Browser Window Navigation](#)—moving to code in the browser window
- [Source-Code Navigation](#)—moving to code in your source files
- [Using the Find Dialog](#)—finding occurrences of specific definitions, variables, or routine calls
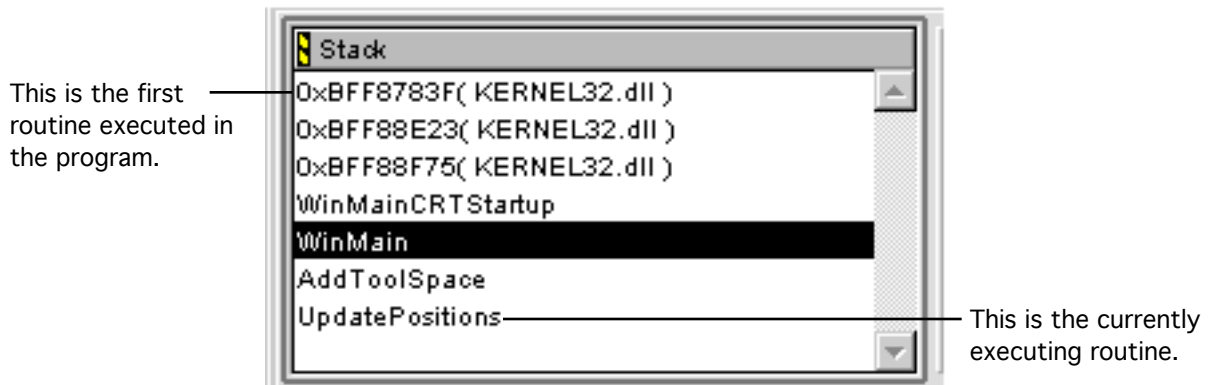
Linear Navigation

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 65, "Stepping Into Routines" on page 66, and "Stepping Out of Routines" on page 67.
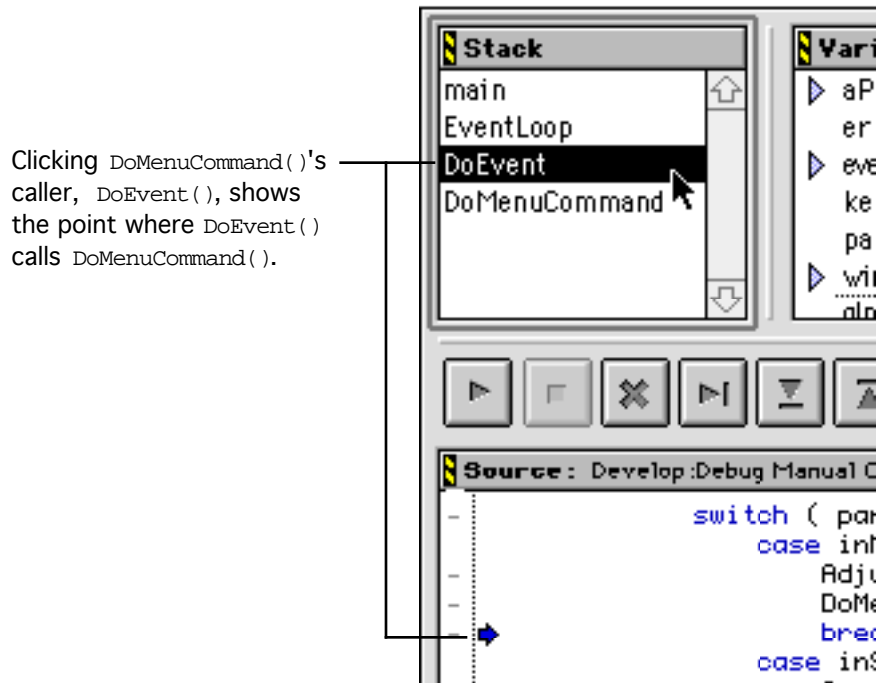
Call-Chain Navigation

The chain of routine calls is displayed in the stack crawl pane of the program window (Figure 4.13). 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 4.13   The stack crawl pane

This is the first routine executed in the program.


```
Stack
0xBFF8783F( KERNEL32.dll )
0xBFF88E23( KERNEL32.dll )
0xBFF88F75( KERNEL32.dll )
WinMainCRTStartup
WinMain
AddToolSpace
UpdatePositions
```

This is the currently executing routine.

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 4.14).

Figure 4.14   Finding a routine's point of call

Clicking `DoMenuCommand()`'s
caller, `DoEvent()`, shows
the point where `DoEvent()`
calls `DoMenuCommand()`.

Browser Window Navigation

You can use the browser window to jump to any location in your source code. To view a specific routine:
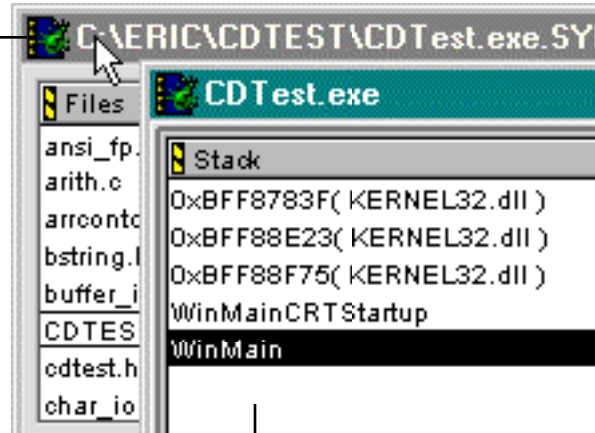
1.   **Make the browser window active ([Figure 4.15](#)).**

Figure 4.15   Activating the browser window
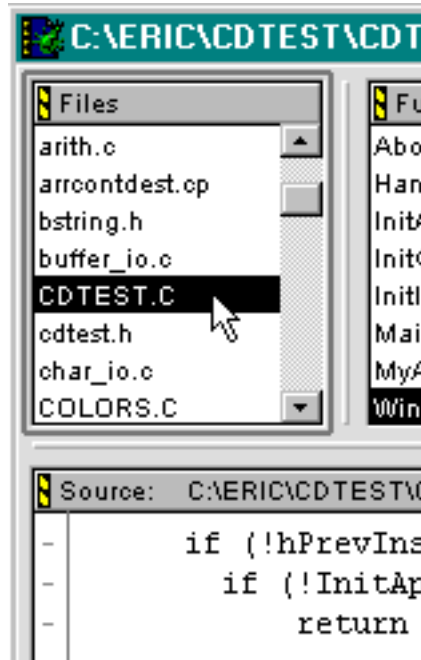
Click in the Browser
Window to make it
active.

The Browser Window
lets you view any
file in the target
project.

Program window

2.   In the browser window's file pane, select the file where the routine is defined ([Figure 4.16](#)).

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 4.16   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 4.17).

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 79.)

Figure 4.17   Choosing a routine to view
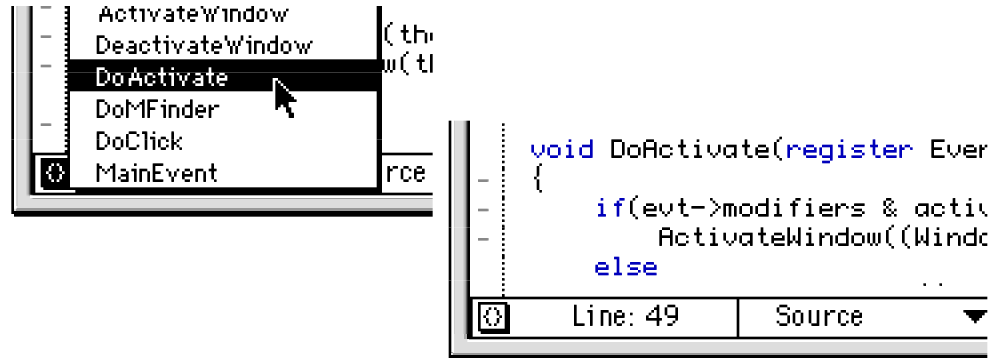

Source-Code Navigation

You can display the routine you want to view in the debugger's browser window by first using the CodeWarrior integrated development environment (either C/C++ or Pascal) to open and search your source code. Then switch to the debugger to view the same code in the browser window.

To display a specific routine or file while using the CodeWarrior IDE, and then display that code in the debugger's browser window:

1. **Within the CodeWarrior environment, open the source-code file that contains the desired routine. The file must be a project file.**

2. **Place the insertion point at the statement you want to appear in the browser window.**

    You can use the function pop-up menu to display a specific routine (Figure 4.18), or use any other technique in the source-code editor to locate the desired code.

Figure 4.18   Selecting a function from C/C++ or Pascal

Choose a routine in
C/C++ or Pascal to
set a breakpoint in.
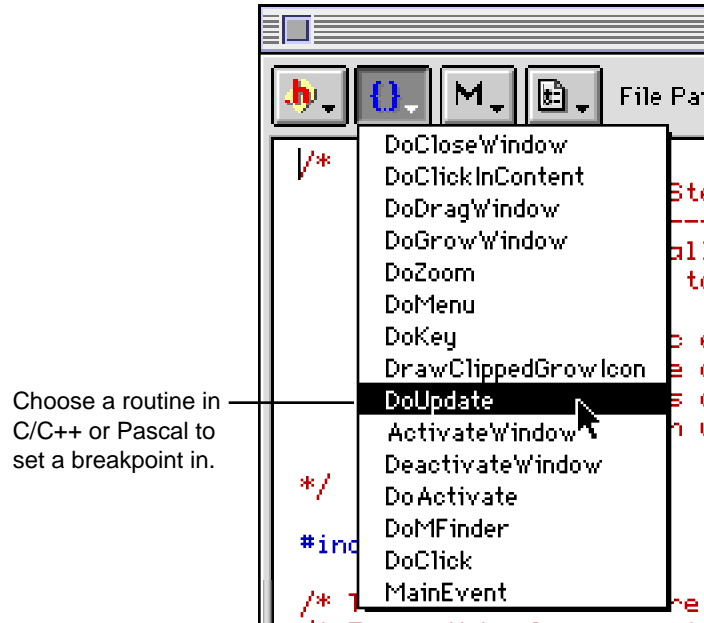
TIP: You can use the **Find** and **Find Next** commands on the Edit
menu to quickly move to the code you want to look at in the
browser window.

---

3.   **Choose the Switch To MW Debugger command from the File
menu ([Figure 4.19](#)).**

The debugger becomes the active application. The browser window
displays the statement at which you set the editor's insertion point.
You can return to the Editor at any time by choosing the **Edit** *file-
name* command from the debugger's File menu ([Figure 4.20](#)).
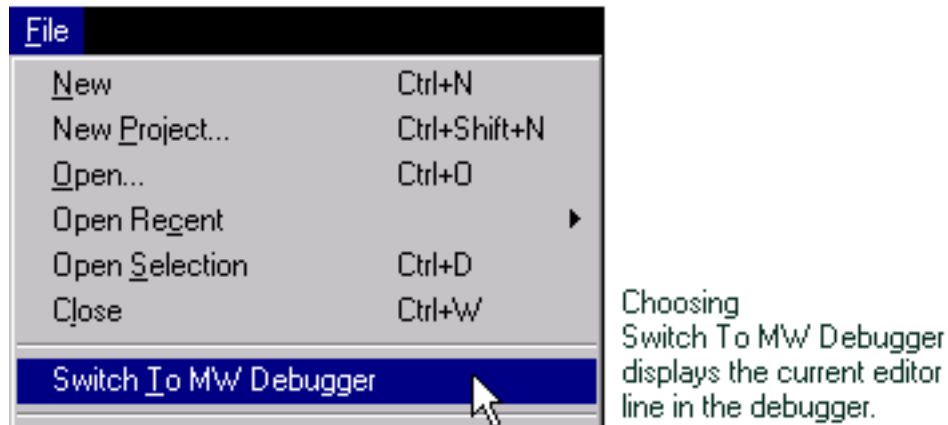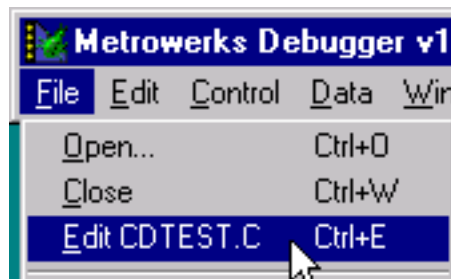
Figure 4.19   Switching to the debugger

Figure 4.20   Returning to the CodeWarrior development environment

Using the Find Dialog

The Find dialog box (Figure 4.21) 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 4.21   Find dialog
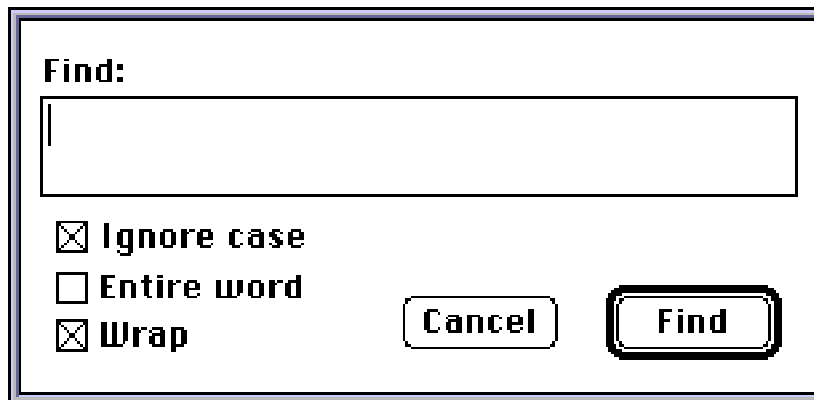
The Find dialog box contains the following items:

- **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.

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/Cmd G (find next) or Ctrl/Cmd H (find selection).

Changing Font and Color

The debugger displays source code in the font and color specified in the CodeWarrior IDE's Editor preference panel.

To change the font and syntax coloring of source code in the debugger:

1. **Launch the CodeWarrior IDE.**
2. **Make sure no editor window or project is open.**
3. **Choose Preferences from the Edit menu.**
4. **Choose the Editor preference panel.**
5. **Set the font and syntax coloring preferences.**
6. **Click OK to close the Preferences dialog box.**

The debugger will use these font and syntax-coloring settings when displaying source code.

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.
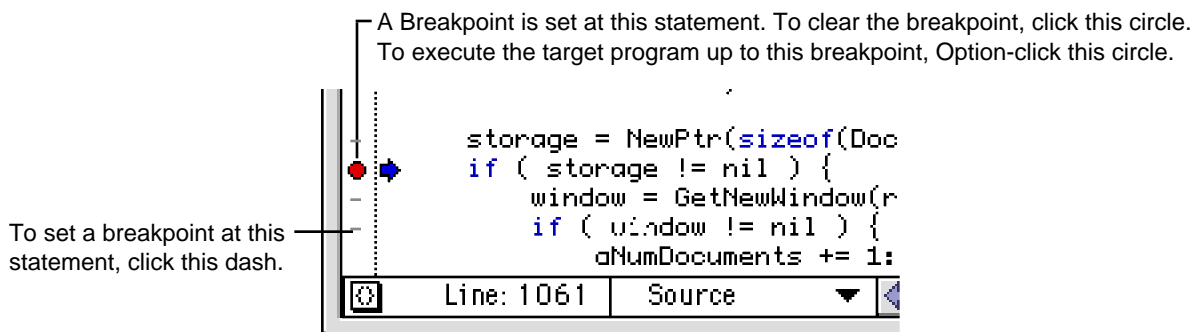
This section discusses:

-
-
-
-
-
-

Setting Breakpoints

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 4.22). 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.

Figure 4.22   Setting breakpoints

A Breakpoint is set at this statement. To clear the breakpoint, click this circle. To execute the target program up to this breakpoint, Option-click this circle.

To set a breakpoint at this statement, click this dash.

```
        storage = NewPtr(sizeof(Doc
        if ( storage != nil ) {
            window = GetNewWindow(r
            if ( window != nil ) {
                aNumDocuments += 1:
    Line: 1061      Source      ▼
```

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/Option click the breakpoint dash to the left of the desired statement. When you resume execution, it will proceed to that statement and stop.

NOTE: If there is already a regular breakpoint at the statement, Alt/Option 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 4.23). 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 4.23   Displaying the breakpoint window
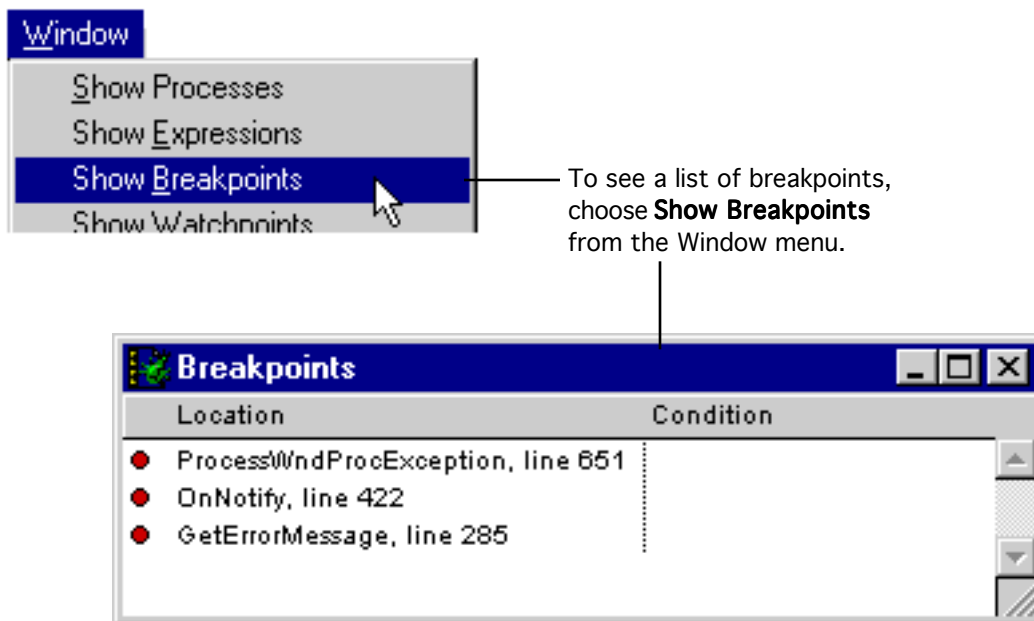
To see a list of breakpoints, choose **Show Breakpoints** from the Window menu.

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

See Also

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 4.24, the debugger will stop execution at line 120 in the `NewBall()` routine if and only if the variable `newTop` is greater than six.

Figure 4.24   **Creating a conditional breakpoint**

> **NOTE:** 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.

Impact of Optimizing Code on Breakpoints

To enable you to set breakpoints accurately, the debugger relies on a direct correspondence between source code and object code. Optimizing your code can disrupt this relationship and cause problems setting breakpoints..

If there is no breakpoint dash to the left of a line of source code in the debugger, you cannot set a breakpoint at that line. The potential causes are:

- symbolics information is disabled for that line
- the routine containing the line is unused and was therefore deadstripped by the linker

- the code has been optimized and the final object code no longer corresponds to the original source code, the subject of this topic

For example, the PowerPC compiler will let you set a breakpoint when the start of a source statement corresponds to the start of a "basic block" (no real need to understand that term) that has at least one instruction in it.

Normally, when Debug Info is turned on for a source file, the compiler will contrive to start a new basic block at each source statement that actually generates some code. For example:

```
- int i = 1;
- if (i)
        {
            int k;
-       int j = 1;
-         i = j;
        }
```

lines like { will not permit a breakpoint because there isn't a unique object code address for that source line, since they generate no instructions.

Once you start turning on the optimizer, things break down. For example, when **Instruction Scheduling** is enabled, the compiler no longer starts a new basic block for each source statement. That allows the scheduler the maximum flexibility for reordering instructions within the block. The different instructions that correspond to a source statement will no longer be consecutive, they'll be intermingled with the instructions from other source statements. In the example above, you'll get breakpoints like this:

```
- int i = 1;
        if (i)
      {
              int i;
-         int j = 1;
              i = j;
      }
```

At optimization level 3 or 4, the source statements you write may not even appear in the generated code. For example, given a loop like this:

```
i = 0;
j = 0;
while (i < 10)
{
        j = j + 1;
        i = i + 1;
}
```

The compiler will translate this into the source-equivalent of this:

```
j = j + 1; // duplicate 10 times
j = j + 1;
...
j = j + 1;
```

or even into this:

```
j = 10;
```

and totally eliminate any instructions corresponding to the while loop.

For best symbolic debugging results, you want to turn optimizations off or use optimizations that are "debug safe." Different optimizations are available for different targets. See the Targeting man-

ual for details. In a typical situation, you would turn **Peephole** and **Global Optimizers** OFF, **Instruction Scheduling** OFF, and **Don't Inline** ON. Then you should get a breakpoint marker at every "meaty" statement.

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 62.

Watchpoints

A *watchpoint* is a location or region of memory that you designate for the debugger to keep an eye on for you. Whenever a new value is written to that area of memory, the debugger will suspend execution of the target program and notify you with an alert message on your screen (Figure 4.25). After dismissing the alert, you can proceed to examine the call chain, inspect or change variables, step through your code, or use any of the debugger's other facilities. (In particular, from the debugger level, you *can* change the contents of the location that triggered the watchpoint without triggering it again.) Use the **Run** command (or the Run button on the toolbar) to continue execution from the watchpoint.

Mac OS   Watchpoints require System version 7.5 to run, and will not work on 68K machines unless virtual memory is enabled. They are also known to be incompatible with Speed Doubler, and possibly with RAM Doubler as well.

Figure 4.25   Watchpoint alert
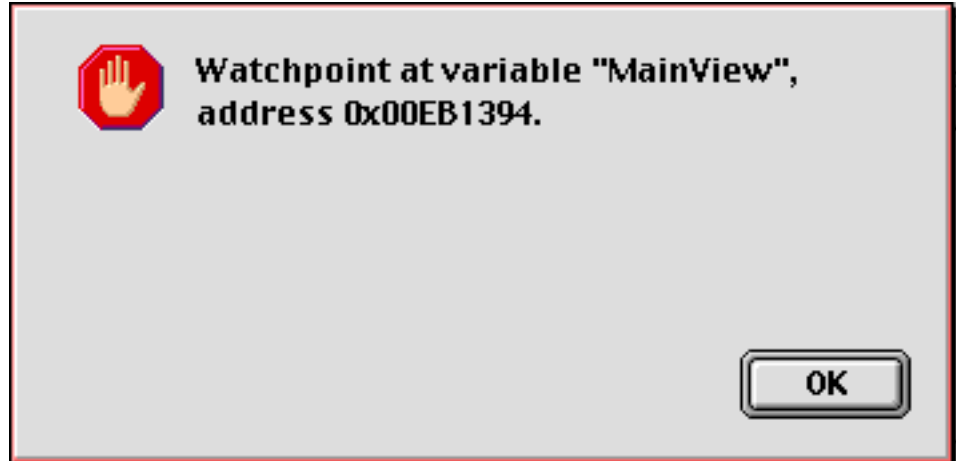
Setting Watchpoints

You can set a watchpoint in any of the following ways:

- Select a variable, in a variable window or in the globals pane of the browser window, and choose **Set Watchpoint** from the Data menu.

- Drag a variable from another window into the watchpoint window.

- Select a range of bytes in a memory window and choose **Set Watchpoint** from the Data menu.

Variables or memory locations on which a watchpoint has been set are underlined in red in the symbolics, variable, or memory windows.

See Also "Use Syntax Coloring in Source Display" on page 121.

WARNING!   There are some significant restrictions on where in memory you can place a watchpoint. You can use them only on global variables or on objects allocated from your application heap. You cannot set a watchpoint on a stack-based local vari-

able, on a variable being held in a register, or anywhere in low memory or the system heap.

Clearing Watchpoints

You can clear a watchpoint in any of the following ways:

- After triggering the watchpoint, choose the **Clear Current Watchpoint** command from the Data menu.
- Select a variable, in a variable window or in the globals pane of the browser window, and choose **Clear Watchpoint** from the **Data** menu.
- Select a range of bytes in a memory window and choose **Clear Watchpoint** from the Data menu.
- Select an existing watchpoint in the watchpoint window and
  - Choose **Clear Watchpoint** from the Data menu
  - Choose **Clear** from the Edit menu
  - Press the Backspace/Delete key

All watchpoints are automatically cleared when the target program terminates or is killed by the debugger.

Viewing Watchpoints

To see a list of all watchpoints currently set in your program, choose the **Show Watchpoints** command from the **Window** menu. A window appears that lists the address and length of each watchpoint (Figure 4.26).

Figure 4.26   Viewing the watchpoint window
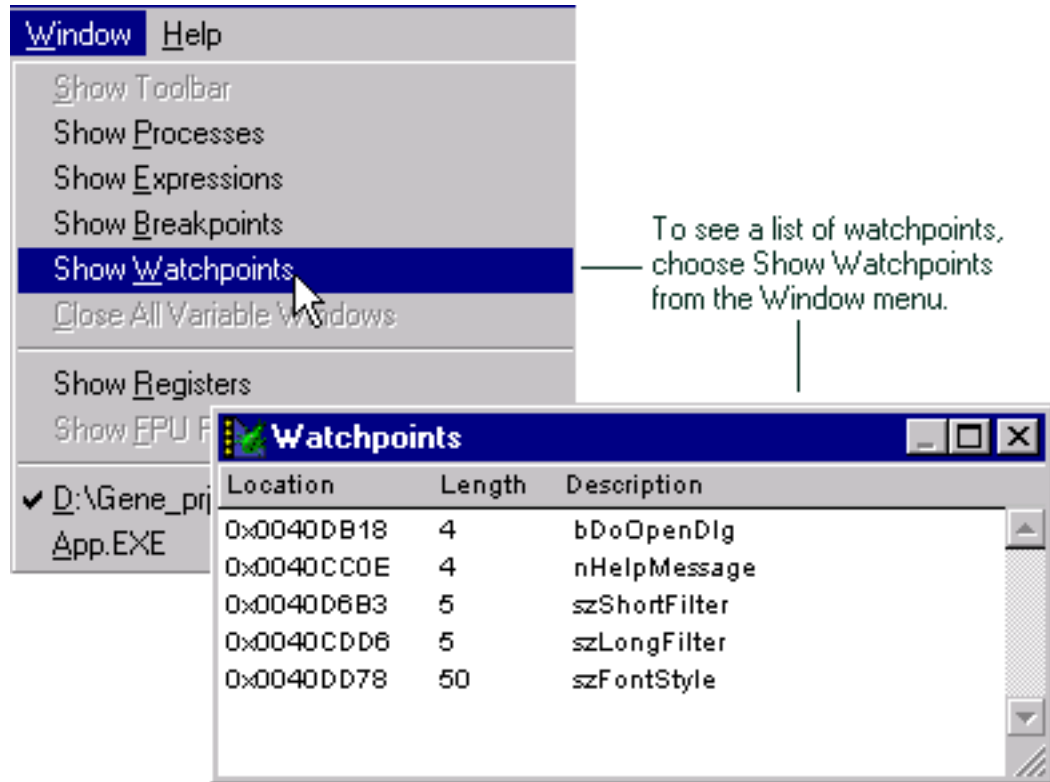
See Also "Watchpoint Window" on page 46.

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](#)

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 4.27](#)). 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 141,](#) [“Collapse All” on page 141,](#) and [“Variables Pane” on page 30.](#)

Figure 4.27   Viewing local variables

These are local variables defined in the currently executing routine.
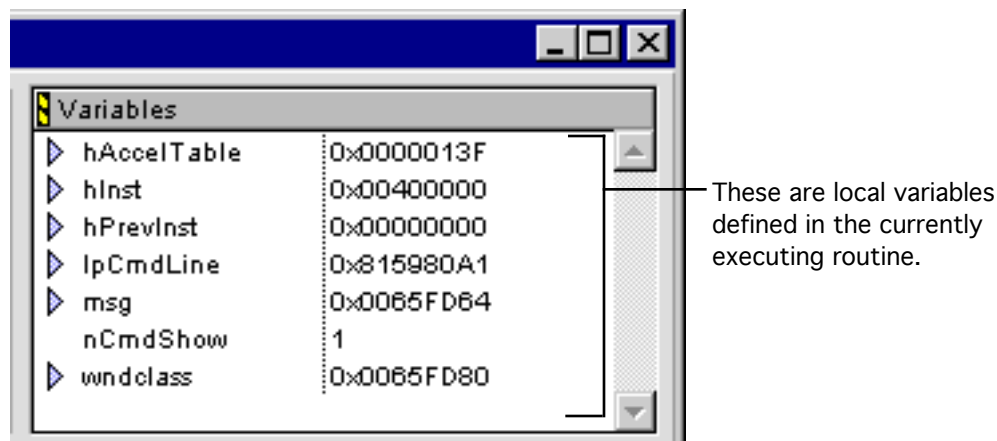
Viewing Global Variables

Global variables appear in the program and browser windows (Figure 4.28). In the program window, they appear below a dotted line in the Variables pane. In the browser window, they appear in the globals pane when you select the Global Variables item in the file pane.

See also "Variables Pane" on page 30 and "Globals Pane" on page 40.

Global variables appear in the browser window (Figure 4.28). They appear in the globals pane when you select the *Global Variables* item in the File pane.

See also "Globals Pane" on page 40.

Figure 4.28 **Viewing global variables in the browser window**

Click *Global Variables* in the browser window
to view all global variables in the program.
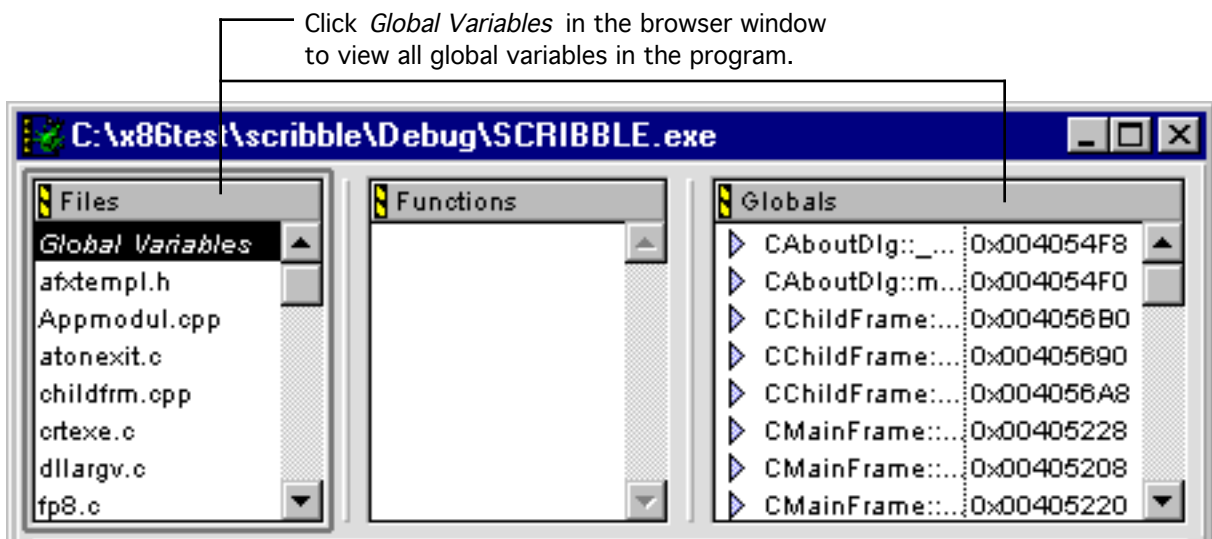
Putting Data in a New Window

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 4.29   Putting a variable in its own window
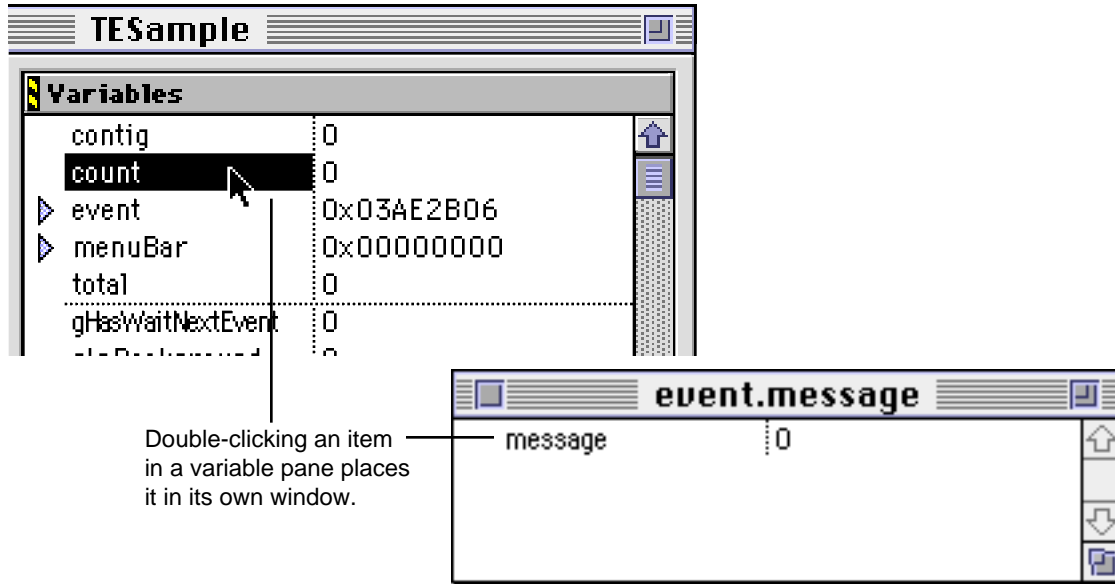
To place a variable or memory location in its own window, double-click its name (Figure 4.29) or select the name and choose the **Open Variable Window** command from the Data menu. If the variable is an array, use the **Open Array Window** command instead. To view the memory the variable occupies as a memory dump, use either the **View Memory** or **View Memory as** command.

See also "Variable Window" on page 48, "Array Window" on page 48, and "Memory Window" on page 50.

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 4.30).

TIP: To show data types automatically, select the **In variable panes, show variable types by default** preference in the **Prefer-**

ences dialog box. See "Settings" on page 118 for more information.

Figure 4.30   Viewing data types

Choose **Show Types** from the Data menu.

Items in the pane appear with their data types.
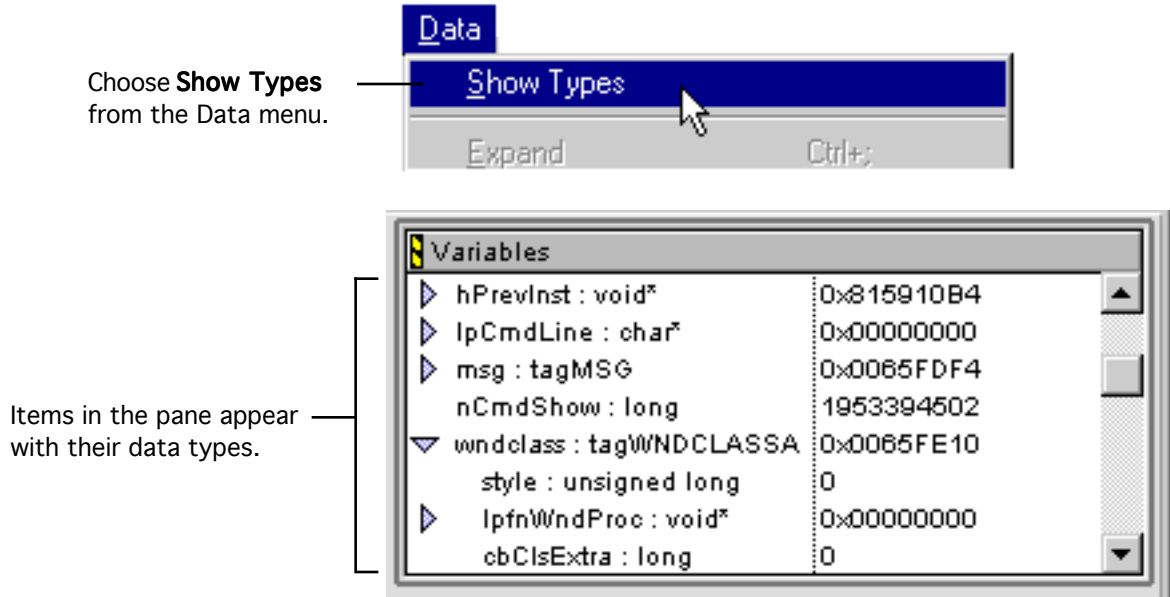
Viewing Data in a Different Format

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 4.31).

Figure 4.31   Selecting a data format

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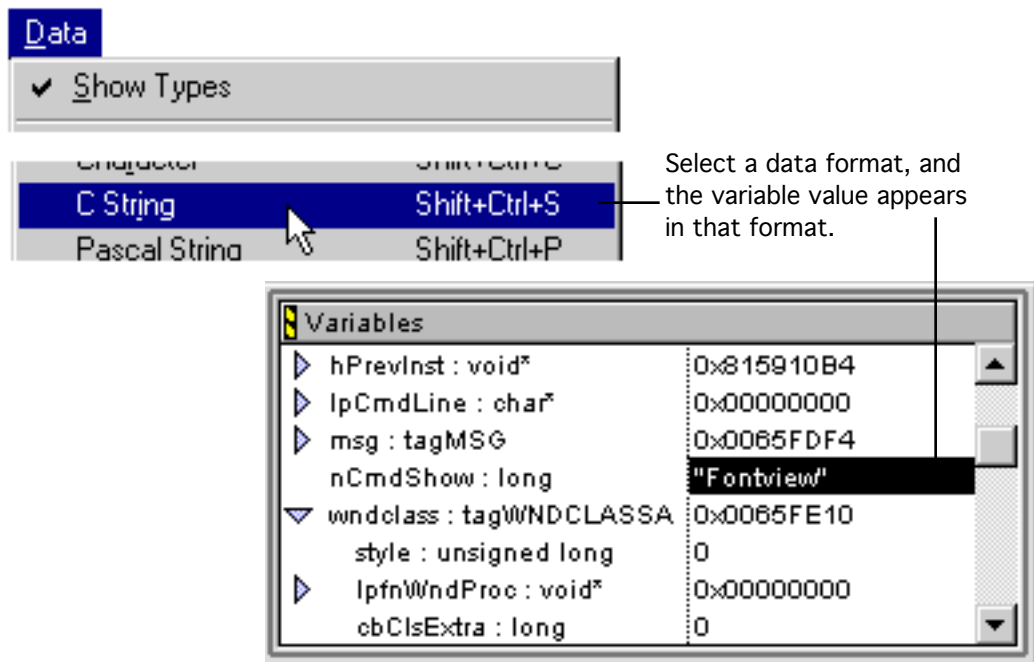
Not all formats are available for all data types. For example, if a variable is an integral value (such as type `short` or `long`), you can view it in signed or unsigned decimal, hexadecimal, or even as a character or string, but not in floating-point, `Fixed`, or `Fract` format.

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.**

2.   Choose View as from the Data menu.

A dialog box appears ([Figure 4.32](#)).

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 4.32   Selecting a data type
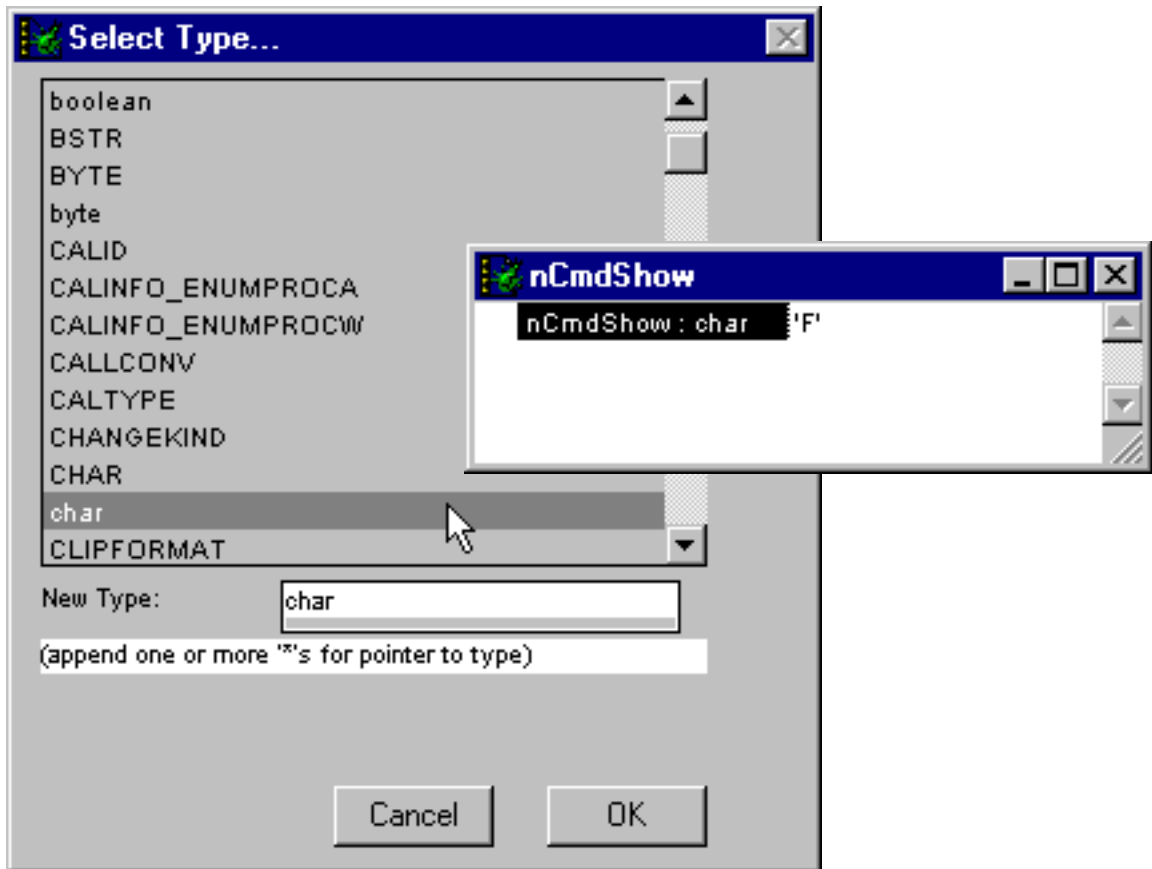
Changing the Value of a Variable

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/Return and type the new value (Figure 4.33).

Figure 4.33   Changing a variable value
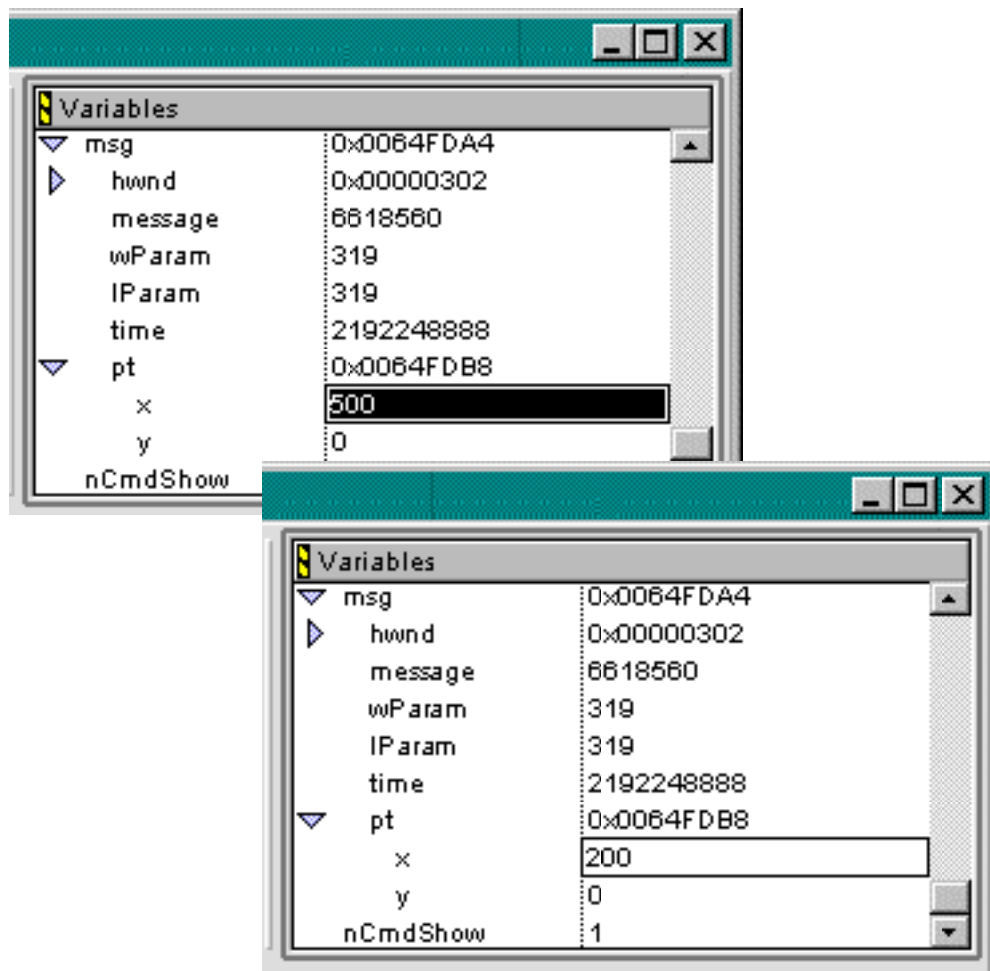
Variable values can be entered in any of the following formats:

- decimal

- 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 44.

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. If you choose **View Memory As,** a dialog box appears for selecting a data type (Figure 4.34); 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 4.34   Choosing a data type to view memory

Selecting a data type enters it in the **New Type** field.

To make it a pointer, add an asterisk (*) after the name of the data type. Use two asterisks for handles.
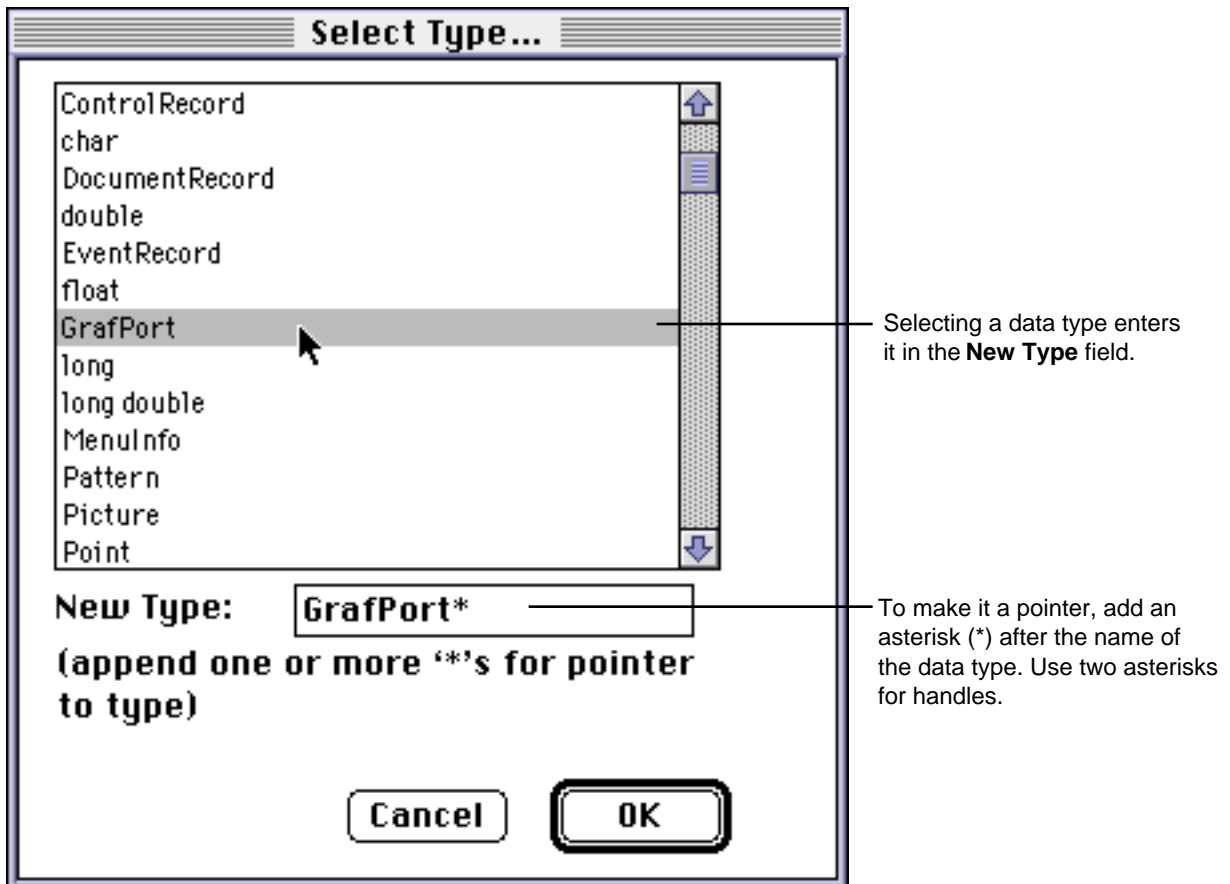
4. **Click the OK button.**

A new window appears ([Figure 4.35](#)) displaying the contents of memory starting at the address referenced by the pointer.

Figure 4.35   Viewing memory as a specified data type
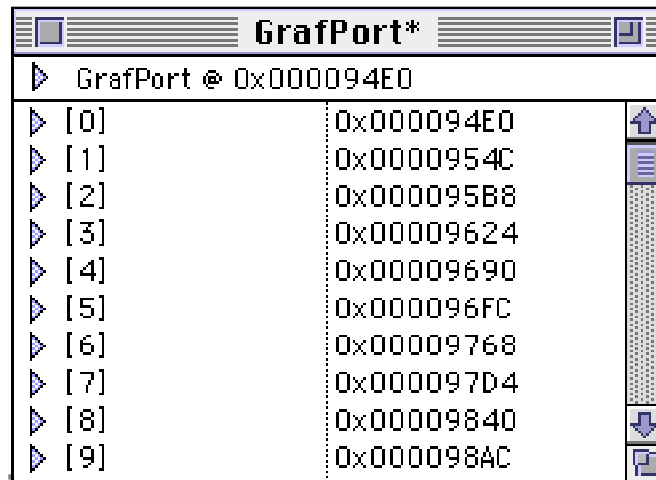
NOTE You can use this technique to view the contents of the stack. If your target processor stores the stack pointer in a particular register, select the value of that register. Then follow the steps above.

See Also ["Memory Window" on page 50.](#)

Viewing Processor Registers

To view the contents of the processor's registers, choose **Show Registers** or **Show FPU Registers** from the **Window** menu ([Figure 4.36](#)). (Some targets do not have an FPU, and the FPU register window is not available for them.)

Figure 4.36   Viewing processor registers
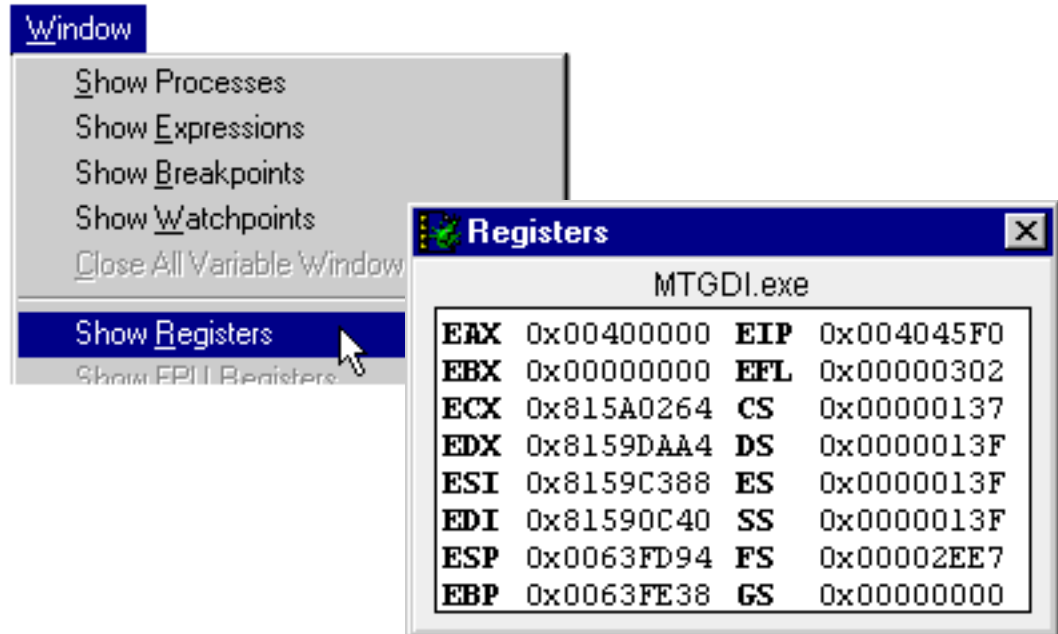
See Also *"Register Window" on page 52.*

Editing Source Code

You cannot edit code directly in the debugger. However, you can use the debugger to open the file so that you can modify your code. In the Files pane of the Browser window you can:

- double-click a file name
- select a file name and choose the Edit filename item in the debugger's File menu.

When you do, the IDE opens the file in an Editor window, and you can edit it.

Windows   You can specify which editor opens a the file. See *"Win32 Settings" on page 127.*

5

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.

This chapter discusses how expressions are treated and used in the debugger. The topics in this chapter are:

- How Expressions are Interpreted
- Using Expressions
- Example Expressions
- Expression Syntax

How Expressions are Interpreted

This section discusses how the debugger interprets expressions in each window. The topics are:

- [Expressions in the Expression Window](#)
- [Expressions in the Breakpoint Window](#)
- [Expressions in the Memory Window](#)

Expressions 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.**

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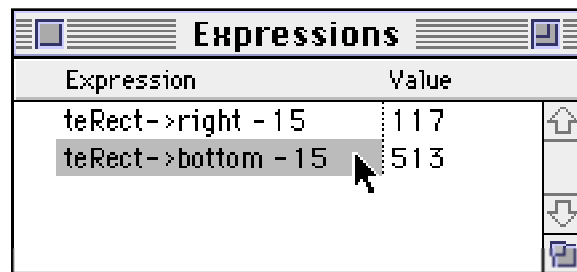
The expression window treats all expressions as arithmetic: the debugger does not interpret the expression's result as a logical value, as it does in the breakpoint window.

See also <u>"Expression Window" on page 44.</u>

Expressions 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 <u>"Setting Breakpoints" on page 80.</u> 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 (<u>Figure 5.2</u>). 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**
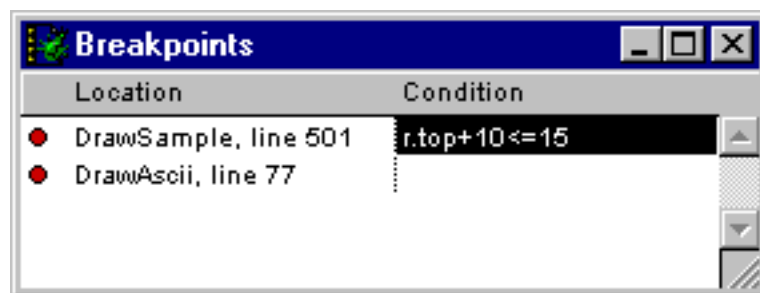
A conditional breakpoint stops the program if the expression yields a true (nonzero) result when execution reaches the breakpoint. If the expression produces a false (zero) result, execution continues without stopping.

See also "Breakpoint Window" on page 45 and "Conditional Break-points" on page 82.

Expressions in the 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 50.

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 be-

tween 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.

- (Mac OS) To refer to register values, use the ® symbol and a register name. (Type Option-r to get the ® symbol.)

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 can-
  not 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
    character-constant
    floating-constant
    string-literal

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

<register-name>
    ®Rnumber
    ®FPRnumber
    ®RTOC

<register-name>
    $PC
    $SP
    $RTOC
    $Anumber
```

NOTE: Registers not targeted by the processor will not display random values for unknown register expressions.

NOTE: For specifying a register, the range for number depends on the number of registers available on the target processor.

```
<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>

<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 preferences in the MW Debug application. 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. The specific panels that appear are in the following two catagories:

- MW Debug Preference Panels
- Integrated Debugger Target Panels

MW Debug Preference Panels

There are several common debugger panels that are available for all targets. Additional panels may appear for particular targets. See the appropriate targeting manual for complete information on target-specific debugger preferences. The common panels that appear in MW Debug are:

- Settings
- Display
- Symbolics
- Program Control
- Win32 Settings
- Windows Java Settings
- Windows Runtime Settings

Settings

The Settings panel is shown in <u>Figure 6.1</u>. The Settings panel includes options related to the saving of debugger settings between sessions

Figure 6.1   Settings preferences
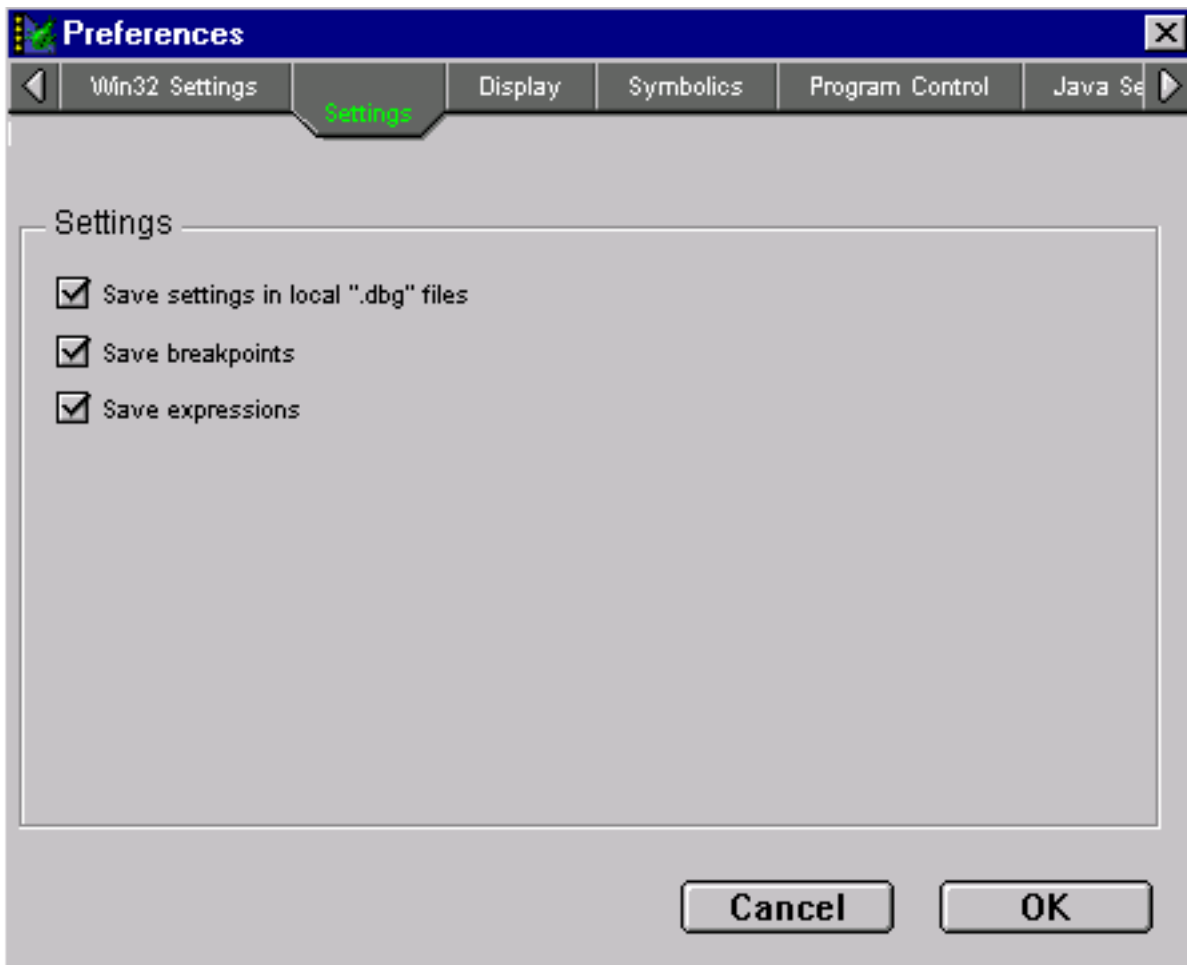
Save settings in local ".dbg" files

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 symbolics

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 symbolics 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 symbolics 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 44.](#)

Display

The Display panel is shown in [Figure 6.2](#). The Display panel includes options related to the saving of display of various items in the debugger's windows.

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.

Figure 6.2   Display preferences
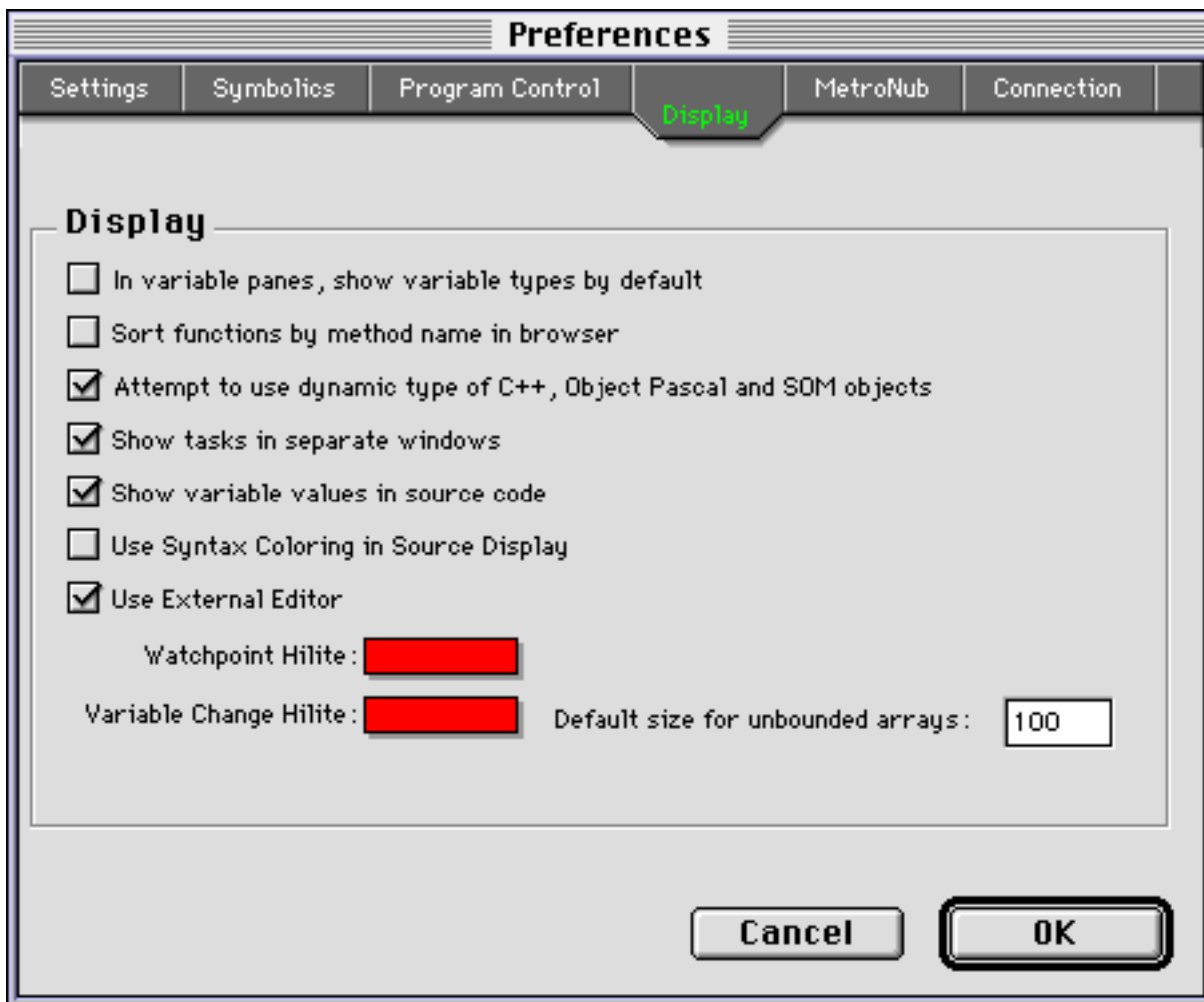
Attempt to use dynamic type of C++, Object Pascal objects and SOM objects

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.

Show tasks in separate windows

Allows you to toggle how to display tasks. If this preference is turned on, double-clicking on a task in the Process window will bring up a separate stack crawl window to display the code. If this option is turned off, the task popup menu will appear at the bottom of the stack crawl window. Use this menu to choose a task to display in the same stack crawl window.

NOTE: The effect of this option does not occur immediately. The setting that is active for the start of a debugging session stays active for the duration of that session. If you change this setting in the middle of a debugging session, you must stop debugging and then restart debugging to make the new preference take effect.

Use Syntax Coloring in Source Display

Allows you to toggle how to display your source code with respect to syntax coloring. If this preference is turned on, your source code text will have different colors with respect to function. (comments will appear green for example). If this option is turned off, the source code will appear as the default text color for the specified target.

Use External Editor

Allows you to toggle if an external editor is to be used to edit your source code.

Watchpoint Hilite

Allows you to set the color that the debugger uses to identify a watchpoint. Clicking the color swatch displays the standard dialog for picking a color. The default color is red.

Variable Change Hilite

Allows you to set the color that the debugger uses to identify a changed variable. Clicking the color swatch displays the standard dialog for picking a color. The default color is red.

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 6.3. The Symbolics panel includes options related to opening, closing, and management of symbolics files and Java class and zip files.

Use temporary memory for SYM data

Uses temporary memory to store the symbolics 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.

At startup, prompt for SYM file if none specified

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

Figure 6.3   Symbolics preferences
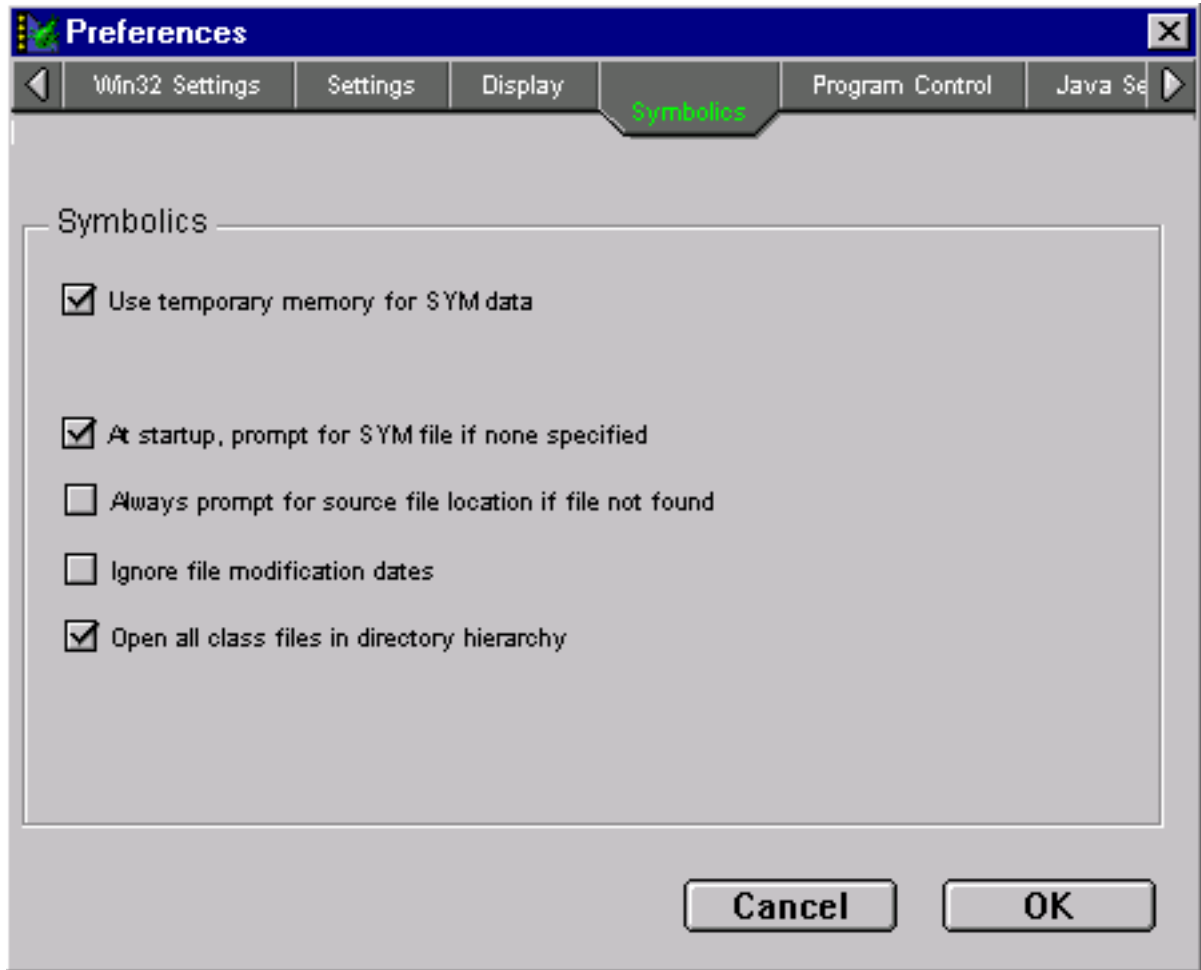
Always prompt for source file location if file not found

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 a Targeting manual for information on target-specific preferences.

Program Control

The Program Control panel is shown in <u>Figure 6.4</u>. The Program Control panel includes options related to the launching, termination, and control of the program being debugged.

Automatically launch applications when SYM file opened

Automatically launches a target program when its symbolics file is opened, setting an implicit breakpoint at the program's main entry point. Deselecting this preference allows you to open a symbolics 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/Option key when opening a symbolics file.

Confirm "Kill Process" when closing or quitting

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

Figure 6.4   Program Control preference
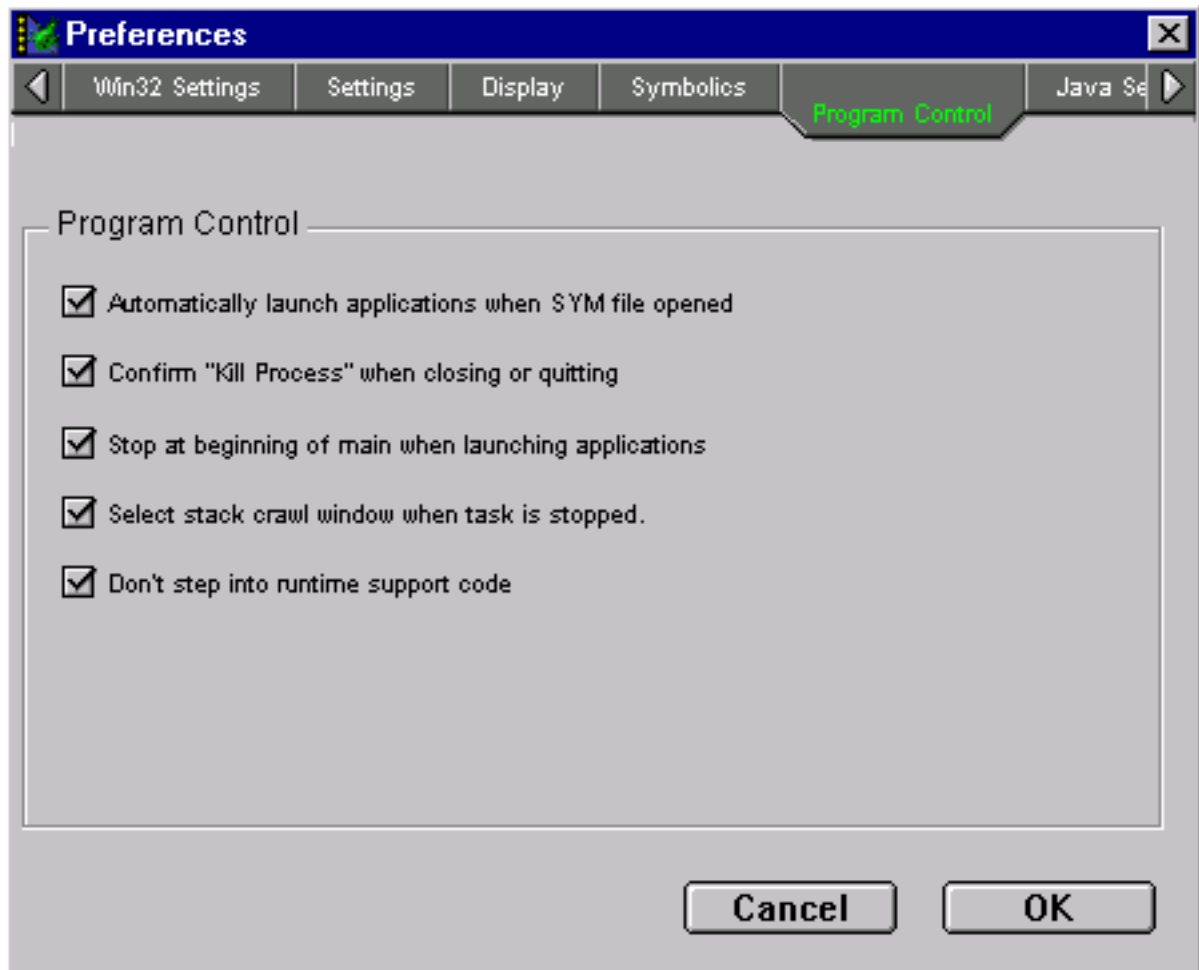
Stop at program main when launching applications

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.

Select stack crawl window when task is stopped

Automatically brings Stack Crawl window to the front when a task is stopped. If this option is turned off, the Stack Crawl window will remain in its previous position. This is useful if you have variable windows open and want to see the variables change as you step through your code. You can control the Stack Crawl window even if it's not the currently active window.

Figure 6.5   Program Control preference (Mac OS)

Don't step into runtime support code

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

QC-aware (Mac OS)

Makes the debugger aware of Onyx Technology's QC system extension. When QC reports an error, the debugger stops the target program at the point of the error and displays an alert. After reporting a QC error, the debugger deactivates QC. Use the QC hot-key combination to reactivate QC before beginning debugging and after each QC error report.

Deselecting this preference makes the debugger ignore QC error reports and prevents it from deactivating QC.

Java Runtime (Mac OS)

This option allows you to choose between using Metrowerks Java runtime or the Apple MRJ when debugging Java Applets. See *Targeting the Java VM* for more details on using this preference.

Win32 Settings

The Win32 Settings panel is shown in <u>Figure 6.6</u>. 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.

Figure 6.6   Win32 Settings
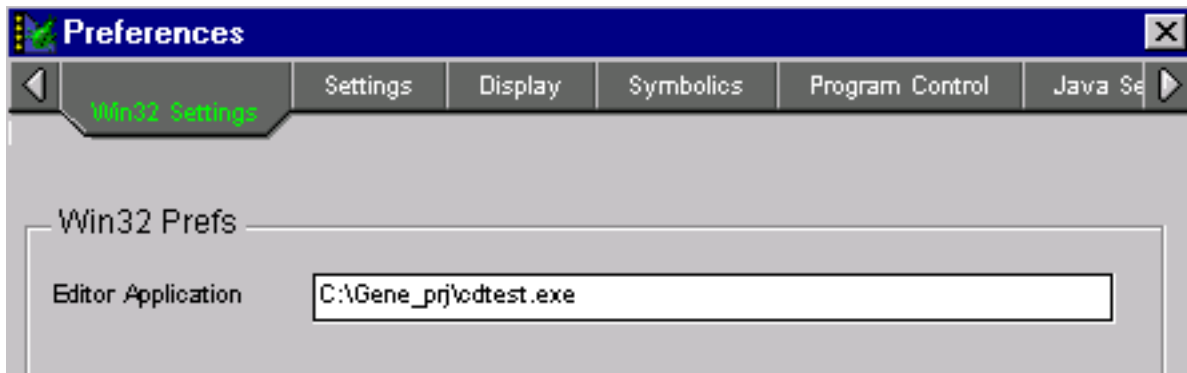
Windows Java Settings

The Java Settings panel is shown in <u>Figure 6.7</u> includes options specific to debugging Java programs and applets. Edit the fields in this panel when debugging Java programs and applets.

Figure 6.7   Java Settings
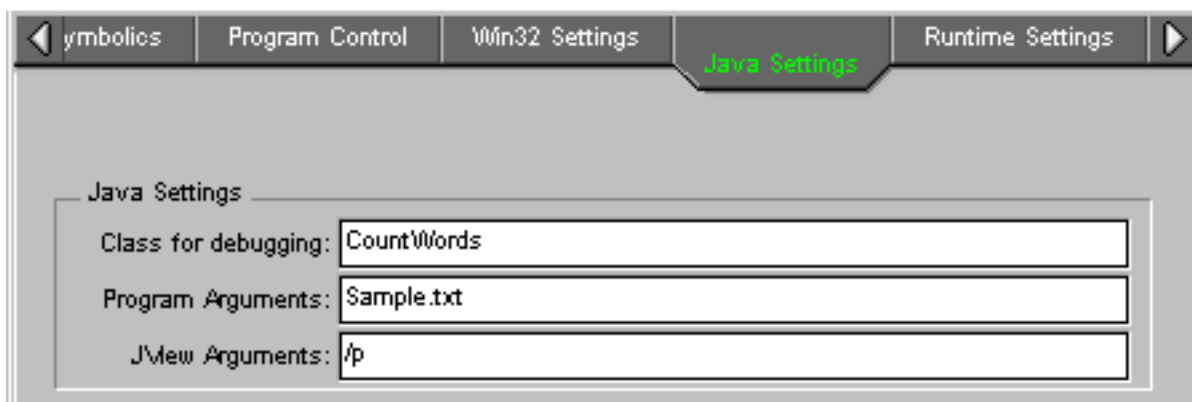
Class for debugging: Edit this field to specify the class file you want to debug.

Program Arguments: Edit this field to specify command line arguments to be used by your project when a java application is debugged.

JView Arguments: Edit this field to specify any arguments jview may require while debugging your project.

See Also   *Targeting Java* for more information on debugging Java programs and applets.

Windows Runtime Settings

The Runtime Settings panel is shown in Figure 6.8 includes options specific to the configuration of the Windows environment. This panel consists of two main areas: Environment Settings and General Settings.

Figure 6.8   Runtime Settings
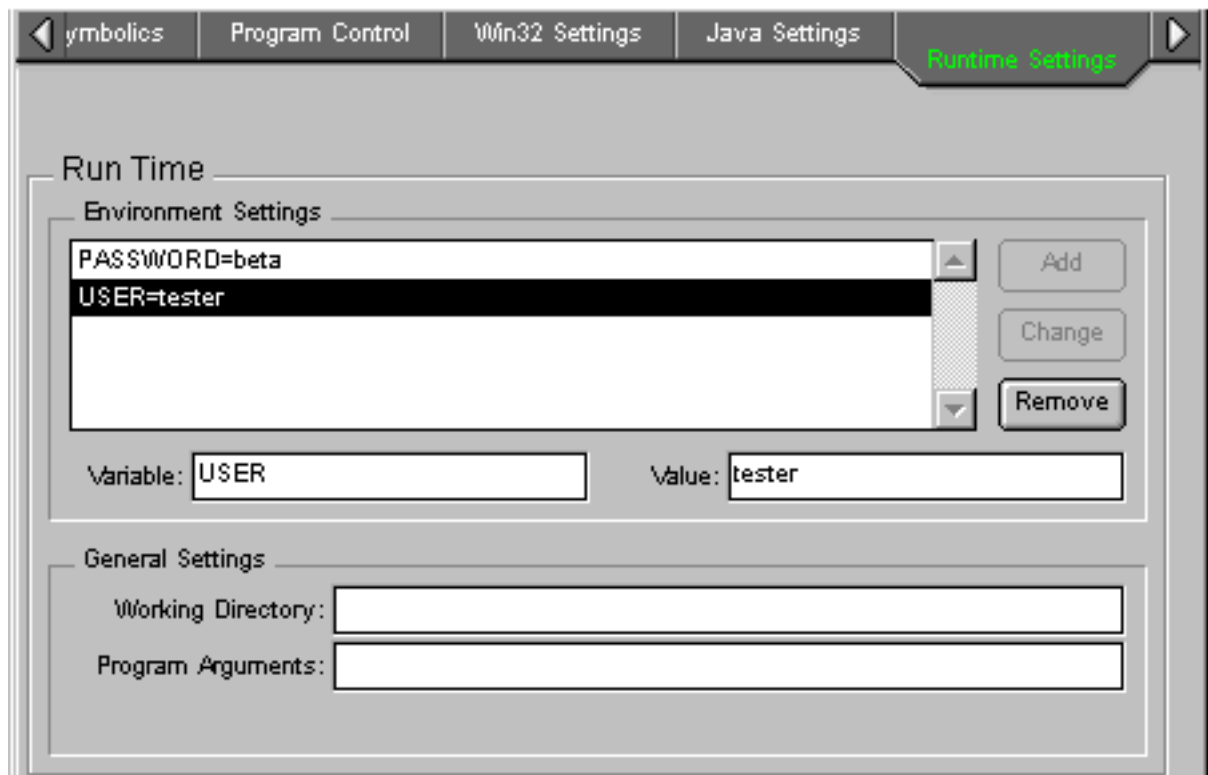
The Environment Settings area allows you to specify environment variables that are set and passed to your program as part of the `envp` parameter in the `main()` or available from the `getenv()` call and are only available to the target program. When the your pro-

gram 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.

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.

Integrated Debugger Target Panels

The following preference panels for targets appear in the debugger integrated into the CodeWarrior IDE. These panels include:

- Target Settings
- x86 Exceptions (Windows)

Target Settings

The Target Settings panel is shown in Figure 6.9 includes options to enable log activities and to specify the application where a shared library, DLL, or code resource are debugged.

Figure 6.9   Target Settings panel
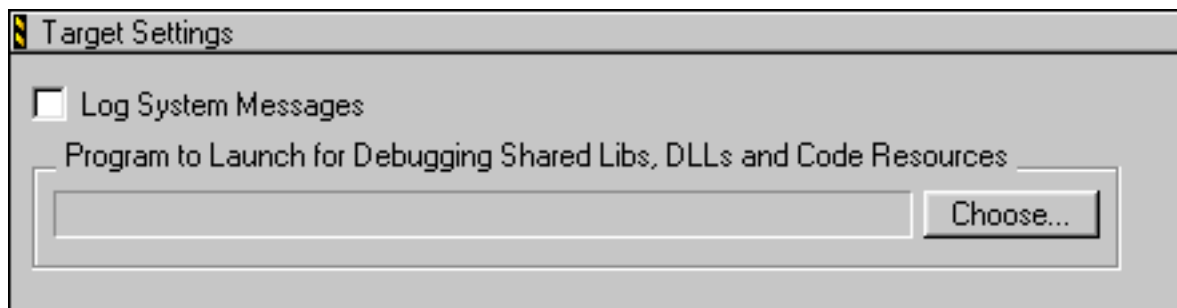
Log System Messages

Enable this option to log all system messages to a file. When disabled, no log file is created.

Program to Launch for Debugging Shared Libs, DLLs and Code Resources

Click **Choose** to select the name of the application to launch when debugging shared libraries, DLLs, or code resources.

This is not a debugger application but the application for which the shared library, DLL, or code resource was written to interact with. For example, if you're writing a Photoshop plug-in, you would use the **Choose** control to select Photoshop as the target application. When **Debug** is chosen, the IDE builds the plug-in, loads the symbolic information for the plug-in, then launches Photoshop to enable you to debug the plug-in.

x86 Exceptions (Windows)

Use the x86 Exceptions panel shown in to specify which exceptions the integrated debugger should catch.

Figure 6.10   x86 Exceptions panel

7

Debugger Menus

This reference chapter describes each menu item in MW Debug.

Debugger Menus Overview

There are six menus:

- **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 (Windows)**—learn about MW Debug
- **Apple Menu (Mac OS)**—learn about MW Debug

NOTE: The actual location of debugger menu commands may be different in the IDE's integrated debugger. See the *IDE User Guide* for more information.

File Menu

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

Open

Opens an existing symbolics file to debug. A Standard File dialog box appears, prompting you to select a symbolics file. The symbolics 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 the Java VM* for more information.

See also "Preparing for Debugging" on page 19 for information on generating symbolic information for your target.

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 re-executes 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 124.

Edit *filename*

Opens the designated source-code file in the CodeWarrior IDE Editor. The CodeWarrior environment must already be running; the debugger will not launch it automatically.

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 127.

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 A Copy As

Displays a Standard File dialog box for saving the contents of the log window under a different name. The old name remains associated with the log window; later **Save** commands will continue saving to the old file name rather than switching to the new 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 <u>"Settings" on page 118.</u>

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 <u>"Confirm "Kill Process" when closing or quitting" on page 124.</u>

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.

S**ee also** <u>"Edit filename" on page 135</u> 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 <u>"Using the Find Dialog" on page 77.</u>

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, Shift-Ctrl/Shift-Cmd G (find previous) or Shift-Ctrl/Shift-Cmd H (find previous selection).

See Also "Using the Find Dialog" on page 77.

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 117.

Control Menu

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

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 64.

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.

NOTE: The Stop command is dependent on operating system services and does not work for all targets. See the appropriate Targeting manual for more information.

See also "Stopping Execution" on page 68.

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 70.

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 65.

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 <u>"Stepping Into Routines" on page 66.</u>

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 <u>"Stepping Out of Routines" on page 67.</u>

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 appropriate Targeting manual for more information on debugging C++ exceptions.

Switch to Monitor (Mac OS)

Gives control to the Macintosh ROM Monitor program or any low-level debugger (such as MacsBug) that you may have installed on your computer.

NOTE: The `MacsBug macros` file supplied with MW Debug contains a pair of MacsBug macros for switching in the opposite direction. If you install these macros in your `Debugger Prefs` file (us-

---

ing a resource-management tool such as ResEdit), you can enter the CodeWarrior debugger from MacsBug by typing `cw` from 68K code or `cwp` from PowerPC code.

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 44.

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 48.

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 48.

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 44.

Set/Clear Watchpoint

Sets or clears a watchpoint for the selected variable or range of memory. You may select a variable or range of memory in the memory window, or you may select a variable from any variable window. If a watchpoint already exists, this command changes to **Clear Watchpoint**.

See also "Setting Breakpoints" on page 80 and "Clearing Watchpoints" on page 88.

Clear Current Watchpoint

Will clear the watchpoint your program has just hit and stopped at.

See also "Clearing Watchpoints" on page 88.

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.

Figure 7.1 Using View As

Selecting a data type enters it in the **New Type** field.

To make it a pointer, add an asterisk (*) after the name of the data type. Use two asterisks for handles.
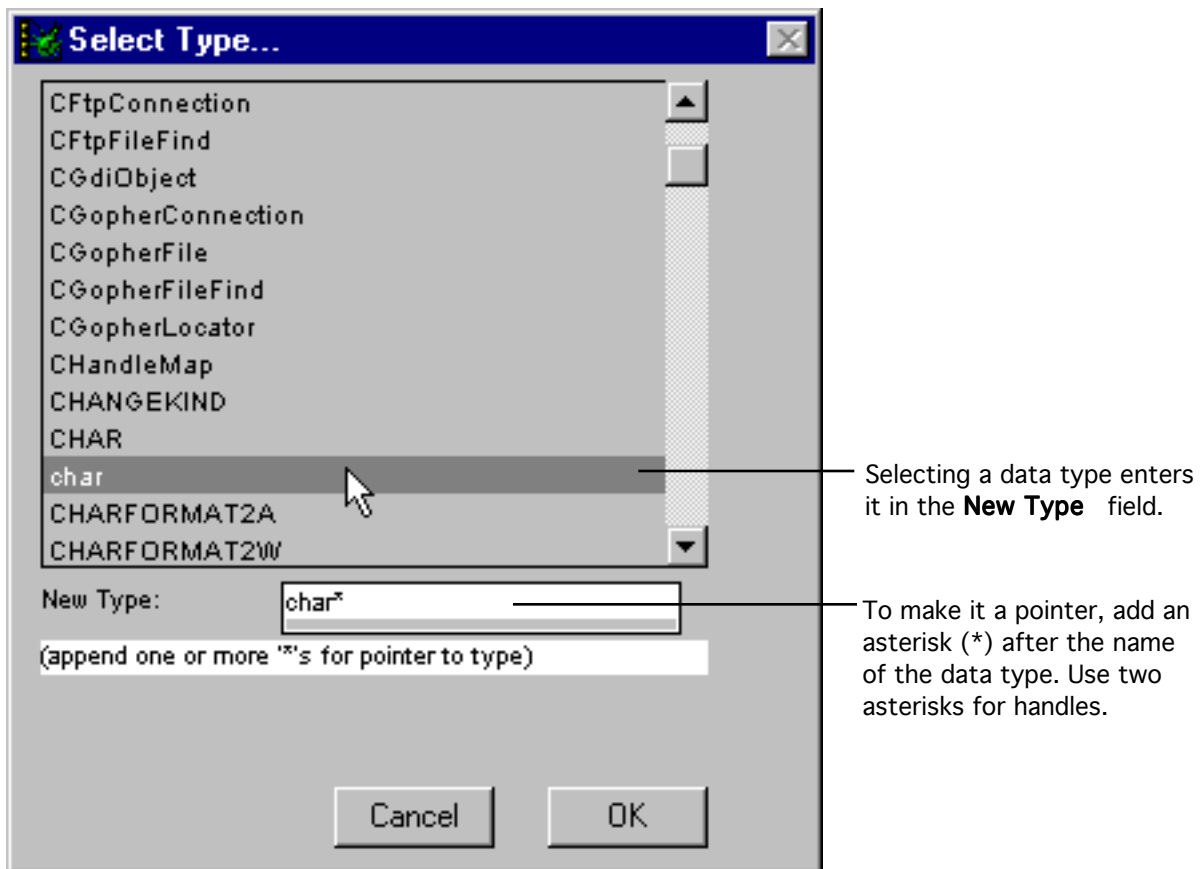
See also "Viewing Data as Different Types" on page 94, "Viewing Raw Memory" on page 98, and "Viewing Memory at an Address" on page 98.

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 48 and "Viewing Memory at an Address" on page 98.

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 50.

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 93.

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.

See also ["Character" on page 145](#) 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 reformat 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 Toolbar (Mac OS)

Displays or hides the mini toolbar. This command toggles between **Show Toolbar** and **Hide Toolbar**, depending on whether the toolbar is currently visible 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

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 44.

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 45.

Show/Hide Watchpoints

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

See also "Watchpoint Window" on page 46.

Close All Variable Windows

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

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 Window" on page 52.

Show/Hide FPU Registers

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

screen. (Some targets do not have an FPU, and the FPU register window is not available for them.)

See also <u>"Register Window" on page 52.</u>

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

Help menu (Windows)

The Help menu contains 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.

Apple Menu (Mac OS)

The Apple menu contains one command for the debugger, **About Metrowerks Debugger.** This command displays copyright and author information about the application, as well as credits.

8

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

If you find that the debugger is causing problems, the first thing you should try is delete the debugger preferences file. This file is called MWDebug.prf and is located in the Metrowerks directory in your Windows directory.

This chapter discusses various problems people have encountered while debugging their programs. There are suggested solutions for each problem. If your problem is not in this chapter, please contact Metrowerks Technical Support for assistance.

The general topics covered include:

- General Problems
- Problems Launching the Debugger
- Problems Running/Crashing the Debugger
- Problems with Breakpoints
- Problems with Variables
- Problems with Source Files

General Problems

There may come a time when one of the solutions in this section doesn't seem to work, or your specific problem isn't in here. Before

sending a note to Metrowerks Technical Support, try one or more of the following:

- Remove easily regenerated files and data, including .(x)symbolics, .dbg file, preferences, binaries in your project (choose **Remove Binaries and Compact** in the CodeWarrior IDE. See the *IDE Users Guide* for more information).

- Copy new versions of the IDE and Debugger to your hard drive.

- Check for extension conflicts.

- Try a few sample sessions with all possible extensions off.

Problems Launching the Debugger

This section lists questions and problems with launching the debugger.

The debugger won't launch

Problem

Even if **Enable Debugger** is selected, when I run my application the debugger doesn't launch.

The **Run** or **Debug** command in the CodeWarrior **Project** menu is dimmed.

Background

You can launch the debugger automatically from the CodeWarrior IDE only if the project is an application project, and the debugger is in the same folder as the CodeWarrior IDE.

Solutions

- Make sure your project generates an application. The **Run** command is only available when creating an application.

- Make sure the CodeWarrior debugger application is in the same folder as the CodeWarrior IDE application.

- Launch the debugger directly by double-clicking its icon.

See also "Launching the Debugger Directly" on page 24.

Debug does nothing

Problem

The **Run** command does nothing.

Background

You must have everything set up properly for the debugger to work correctly. In addition, make sure your code actually does something!

Solutions

- Make sure debugging is enabled.
- Consult the debugger release notes for the latest information on incompatibilities with third-party software.

See also "Setting Up a Target for Debugging" on page 19.

Errors reported on launch (Mac OS)

Problem

I get error -27 when I start debugging and -619 when I quit.

Background

You may be using an older version of RamDoubler that is not compatible with the debugger.

Solution

- Upgrade RamDoubler to version 1.5.2 or better.

Slow launching (Mac OS)

Problem

I have a big project. When I run with the debugger, it takes a long time (minutes) before the debugger is up, and I can hear the hard drive thrashing around like crazy.

Background

The debugger uses temporary memory to store symbolics information. If you are using virtual memory, it is better to disable the debugger's use of temporary memory.

Solution

- In the debugger's Preferences dialog, turn off the **Use temporary memory for SYM data** preference. Increase the size of the debugger partition by about the size of your symbolics file.

Problems Running/Crashing the Debugger

This section covers problems that occur while you're running the debugger.

Project works in the debugger, crashes without

Problem

My project works fine when running under debugger control. When I run without the debugger, my program crashes.

Background

Running under the debugger changes the operating environment in which your program runs. This can have the strange effect of making an otherwise buggy program work correctly. It is hard to tell precisely what your particular problem may be, but we can suggest a couple of factors that may causes this kind of odd behavior.

For one, the debugger slows things down. If you have code that is time-sensitive, things may happen too fast when the debugger is not present. Keep this in mind when tracking down the problem.

The presence of the debugger can also modify how memory is managed in your project. A block of memory may not move if the debugger is running, for example. With the debugger absent, the block moves and a memory-related bug strikes.

Solutions

- Look for time-sensitive problems, race conditions, and so forth.
- Look for memory-related problems, such as accessing null pointers or handles, improperly disposing of resource handles, or disposing of handles more than once.
- Develop your low-level debugging skills.

Problems with Breakpoints

This section covers problems related to setting and clearing breakpoints.

Statements don't have breakpoints

Problem

Some statements don't have dashes in the breakpoint column, making it impossible to set them.

Background

The CodeWarrior linkers are very smart. They do not generate symbolics information for source code that is not linked into the final product. If a statement is never actually used, the linker does not include it in the final object code. You cannot set a breakpoint on such a statement, because the object code does not exist.

Code optimization may also reorganize the object code extensively, affecting the correspondence between object code and source code and making it difficult or impossible to set breakpoints accurately.

Solution

- Make sure all your code is used. Change the source code if necessary.

- Check your source code to see if statements were ignored by the compiler because of compiler directives.

- Turn off all compiler optimizations, set **Instruction Scheduling** off, and set **Don't Inline** on, and rebuild your project. Optimization may be making changes in object code that do not correspond to your source code. You should get a breakpoint marker at every "meaty" statement.

- Use a coding style wherein you put only one statement on a source line—the compiler will output breakpoint information for multiple statements on a line, but the debugger only shows which source line you're on, so you may end up stepping multiple times on the same source line.

See also "Impact of Optimizing Code on Breakpoints" on page 83.

Breakpoints don't respond

Problem

I set a breakpoint, but it doesn't work.

Background

Breakpoints stop execution only if the breakpoint is reached, it is active, and its condition (if it has one) is true.

Solutions

- Step through your code to verify whether you're reaching the statement at which you placed the breakpoint.

- Look in the breakpoint window to see if the breakpoint is inactive.

- If the breakpoint has a condition, make sure it tests true. The debugger ignores breakpoints with false conditions.

See also "Setting Breakpoints" on page 80 and "Conditional Break-points" on page 82.

Problems with Variables

This section covers problems related to variables.

A variable doesn't change

Problem

I have a variable and I assign it a value, but the value doesn't change in the debugger.

Background

You aren't using the variable for anything later on in the code. As a result, the compiler has optimized it away.

Solutions

- Remove the unused variable from your code.
- Modify your code to use the variable.

Variables are assigned incorrect values

Problem

I notice that values seem to be changing incorrectly. I have encountered one of these two problems:

- Two or more variables are being set to the same value simultaneously.
- One variable is receiving a value that is supposed to be assigned to another.

Background

The compiler has recognized that the variables are not used concurrently, and has given the variables the same storage location. What you are seeing is a kind of automatic compiler optimization called "register coloring." Register coloring checks to see how variables are used in a routine. If two or more variables are in the same scope but are not used at the same time, the compiler may use the same processor register for both variables. Using registers instead of memory to store and manipulate variables improves a program's performance.

Listing 8.1 is a good example of the kind of code that results in register coloring. Because four different variables are set but never used simultaneously, the compiler has arranged for all four to use the same register. The debugger, however, has no way of knowing that all four variables share the same register, so it shows all four variables changing with each assignment. In fact, the code shown in Listing 8.1 does nothing at all; serious optimization might eliminate it entirely.

Listing 8.1   Variables changing with register coloring

```
void main(void)
{
  long a = 0, b = 0, c = 0, d = 0;

  a = 1; /* a is set to 1 */
  b = 2; /* a is set to 2, b remains unchanged */
  c = 3; /* a is set to 3, c remains unchanged */
  d = 4; /* a is set to 4, d remains unchanged */
}
```

Solutions

- Do nothing. Register coloring is not a problem.
- To prevent register coloring in C/C++, declare your variables with the `volatile` keyword. Do this with a preprocessor directive so that you can easily remove the `volatile` storage class specifiers after debugging.

See also the *C, C++, and Assembly Language Manual* for more information.

Strange variables

Problem

The debugger shows variables in the local and global variable panes that are not declared in the source code.

Background

The compiler often creates its own temporary variables in the object code as it translates source code. These temporary variables appear in the debugger with a dollar sign ($) in their names. The debugger also displays C++ virtual base class types with a $ prefix.

The compiler and linker often add variables from libraries and run-time routines that help initialize and terminate your program.

Solution

- None. This is not a problem that needs correction.

Strange data types

Problem

When **Show Types** is selected in the **Data** menu, some enumerated values are displayed as having type "?anon*x*," where *x* is an arbitrary number.

Background

The debugger cannot display the names of enumerated types if the names are not defined in the source code. At compile time, the compiler assigns a generic type name to such enumerated types. It is this generic name that the debugger displays.

For example, in [Listing 8.2](#), with **Show Types** selected, variable `my-Marx` will be displayed as having the anonymous type `?anonx`, because its enumerated type has no name. On the other hand, variable

`myBeatle` will be shown with type `Beatle`, because its enumerated type is defined with that name.

Listing 8.2   Unnamed enumerated types (C/C++)

```
// Debugger displays as anonymous type
enum {Groucho,
  Harpo,
  Chico,
  Zeppo } myMarx = Harpo;

// Debugger displays as type Beatle
typedef enum Beatle {John,
  Paul,
  George,
  Ringo} myBeatle = John;
```

Solution

- None. This is not a problem that needs correction.

Unrecognized data types

Problem

I've declared my own data type. Why can't I view a variable as that type?

Background

The symbolics file includes information only about types that are used in the program. Types defined in `typedefs` are not stored in the symbolics file, so you need to view the variable as the type it is derived from. For example, if you have declared a type `MyLong` based on the `long` data type, you can view it as a `long`, but not as a `MyLong`.

See also <u>"Viewing Data as Different Types" on page 94.</u>

Solution

- Use the base data type.
- Choose the **Show Types** item from the **Data** menu to see what the debugger thinks the type is.

"undefined identifier" in the expression window

Problem

Using a user-defined type in an expression in the expression window gives an "• undefined identifier •" value.

Background

The debugger does not recognize data types that are simply aliases of another type, because such alias types are not included in the symbolics file.

For example, given the Pascal type declaration

```
TYPE
  MYBIGINT = LONGINT;
```

the expression

```
MYBIGINT(thePtr)
```

in the debugger's expression window will display its value as "• undefined identifier •." To get the correct result, use this expression instead:

```
LONGINT(thePtr)
```

Solution

- Use the original data type instead of the defined data type.

See also <u>"Expression Limitations" on page 107.</u>

Problems with Source Files

This section covers problems related to source-code files.

No source-code view

Problem

All I see in the source pane is assembly-language code. The source popup menu won't let me show source code.

Background

There is no symbolics information available for that code. You may not have turned on debugging for a file, or you may be stepping through some ancillary code added by the linker that has no corresponding source code (for example, glue code). Without symbolics information, the debugger can only show the code in assembly language.

Solutions

- If the code is from your own source file, make sure the CodeWarrior IDE generates symbolics information for the file.
- If the code is from some other source (such as a compiled library), step out of the function to return to the caller. There is no source code to view.

See also <u>"Setting Up a File for Debugging" on page 21.</u>

Outdated source files

Problem

When I run my project, I get a warning that says the modification dates don't match. What's going on?

Background

The symbolics file keeps track of when the source file on which it is based was last changed. If the date and time stored in the symbolics file do not match those of the original file, the debugger warns you that the symbolics information may no longer be current.

Solution

- Touch the source file (or make a do-nothing change and save it), then rebuild your project or bring it up to date.

Sharing source code between projects

Problem

The debugger displays an alert when attempting to view the same source-code file from two different browser windows.

Background

The debugger cannot open the same file from different browser windows.

Solution

- Create copies of the file so that each project has its own version.

Spurious ANSI C code in Pascal projects

Problem

I'm working in Pascal, and when I'm stepping through code I find ANSI C routines! I haven't included any ANSI C libraries. What's going on?

Background

The Pascal runtime library was written in C and uses ANSI C routines. These are the routines that show up when debugging a Pascal program.

Solution

- None. This is not a problem that needs correction.

Debugger Error Messages

Following is a list of error messages that you may receive from the debugger, with some hints about the possible causes or circumstances of the error. Messages listed without comment are self-explanatory.

An unknown error occurred while trying to target an existing process.

Bad type code

Internal error.

Bus Error

Attempt to read or write to an invalid address.

can't display value -- type information not supported

The symbolics file contains a data type that MW Debug does not support.

Can't use this source file, it was not saved before running, or was edited after linking.

The debugger doesn't have access to the same text the compiler saw. The debugger will just issue a warning unless the debug information references nonexistent text, in which case it gives you this error message.

class name expected

Unexpectedly encountered something other than a class name while evaluating an expression.

Could not complete your request because the process is not

suspended.

The command you issued cannot be performed while the program is running.

Could not set a watch point because the page containing that memory location overlaps low memory or the system heap.

Watchpoints cannot be set in low memory or in the system heap.

Could not set a watch point because the page containing that memory location overlaps the stack.

Watchpoints are implemented via the memory write-protection mechanism, which operates at the page level. You cannot write-protect a page of memory containing part of the stack.

Couldn't locate the program entry point, program will not stop on launch.

When launching a program, the debugger normally sets an implicit breakpoint at the beginning of the function named main() (in C/C++) or the main program (in Pascal). If it can't find such a routine, it just launches the program and lets it run.

identified or qualified name expected

Unexpectedly encountered something other than an indentifier or qualified name while evaluating an expression.

illegal character constant

Invalid character constant encountered while evaluating an expression.

illegal string constant

Invalid string constant encountered while evaluating an expression.

illegal token

Invalid token encountered while evaluating an expression.

Invalid C or Pascal string.

An ill-formed string was encountered in evaluating an expression.

Invalid character constant.

An invalid character constant was encountered in evaluating an expression.

Invalid escape sequence inside string or character constant.

C/C++ escape sequence in a string wasn't valid syntax.

invalid pointer or reference expression

Invalid pointer or reference expression encountered while evaluating an expression.

invalid type declaration

Invalid type encountered while evaluating an expression.

invalid type information in SYM file

MW Debug is unable to display a variable because of bad data in the symbolics file.

New variable value is too large for the destination variable.

For example, you have attempted to assign a 20-byte string to a 10-byte string variable.

No type with that name exists.

MW Debug doesn't recognize a type name you have entered in the **View As** dialog.

Register not available

The debugger is unable to get a valid register value to display a register variable. For example, when looking at routines up the stack from the current routine, the debugger can't dig out the saved regis-

ter values unless all routines below it on the stack have debug information.

string too long

String exceeds maximum permissible length.

The new variable value is the wrong type for the destination variable.

You have attempted to assign a value of the wrong type to a variable, such as a string to an integer variable.

typedef name expected

Unexpectedly encountered something other than a typedef name while evaluating an expression.

Unable to step from here.

The debugger cannot step execution from this point.

Unable to step out from here.

The debugger cannot step out from this point.

undefined identifier

Undefined identifier encountered while evaluating an expression.

unexpected token

Unexpected token encountered while evaluating an expression.

unknown error "^0"

An internal error that was not expected to reach the user.

unterminated comment

A closing comment bracket is missing.

Variable or expression cannot be used as an address.

For example, if `r` is a `Rect`, `*(char*)r` is invalid.

Warning - this SYM file has some invalid or inconsistent data. The debugger may show incorrect information.

Your symbolics file may have been corrupted.

'*' or '&' expected

Unexpectedly encountered something other than a pointer or reference operator while evaluating an expression.

Index

Symbols

$ in variable name 157
? in variable name 157

A

active pane 28
ANSI
    C code in Pascal project 161
    escape sequence 145
Apple menu (debugger) 148
array window 48, 92
    setting base address 48
arrays
    setting size 122
assembly
    memory display 31, 35
    register display 31, 35
    viewing 25, 35, 42, 160

B

Break on C++ Exception command (debugger) 140
breakpoint
    clearing 45, 81
    clearing all 140
    conditional 45, 82
    conditional expression 105
    conditional, and loops 83
    conditional, creating 83
    defined 79
    effect of temporary breakpoint on 81
    missing 153
    setting 80, 154
    setting in breakpoint window 45
    setting in browser window 42
    temporary 68, 81
    viewing 81
breakpoint window 45, 82
browser source pane 42
browser window
    compared to program window 37
    navigating code in 72
    setting breakpoint in 42

C

C language
    entering escape sequences 145
    viewing character strings 145
C string
    entering data as 97
    viewing data as 93
C String command (debugger) 145
C++
    debugging 121, 124
    ignoring object constructors 127
    methods, alphabetizing 40, 120
call-chain navigation 71
changing
    font and color in debugger 79
    memory 51
    memory, dangers of 51
    registers 53
    variable values 96
Character command (debugger) 145
Clear All Breakpoints command (debugger) 81,
  140
Clear command 88
Clear command (debugger) 44, 46, 137
Clear Current Watchpoint command 88, 142
Clear Watchpoint command 46, 142
clearing breakpoint 45, 81
Clipboard
    while in the debugger 137
Close All Variable Windows command
  (debugger) 41, 147
Close command (debugger) 134
Collapse All command 90, 141
conditional breakpoint 45, 82
    and loops 83
    creating 83
    expressions and 105
control buttons 31
Control menu (debugger) 138
conventions 10
    figures 12
    host terminology 11
    keyboard shortcuts 12
Copy command (debugger) 47, 137

Index

CodeWarrior

Debugger User Guide

Credits

writing lead: L. Frank Turovich

other writers: Gene Backlin, Carl Constantine, Marc Paquette, Jim Trudeau

engineering: Mike Lockwood, 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

metrowerks ®

Guide to CodeWarrior Documentation

CodeWarrior documentation is modular, like the underlying tools. There are manuals for the core tools, languages, libraries, and targets. The exact documentation provided with any CodeWarrior product is tailored to the tools included with the product. Your product will not have every manual listed here. However, you will probably have additional manuals (not listed here) for utilities or other software specific to your product.

Core Documentation	
IDE User Guide	How to use the CodeWarrior IDE
Debugger User Guide	How to use the CodeWarrior debugger
CodeWarrior Core Tutorials	Step-by-step introduction to IDE components
Language/Compiler Documentation	
C Compilers Reference	Information on the C/C++ front-end compiler
Pascal Compilers Reference	Information on the Pascal front-end compiler
Error Reference	Comprehensive list of compiler/linker error messages, with many fixes
Pascal Language Reference	The Metrowerks implementation of ANS Pascal
Assembler Guide	Stand-alone assembler syntax
Command-Line Tools Reference	Command-line options for Mac OS and Be compilers
Plugin API Manual	The CodeWarrior plugin compiler/linker API
Library Documentation	
MSL C Reference	Function reference for the Metrowerks ANSI standard C library
MSL C++ Reference	Function reference for the Metrowerks ANSI standard C++ library
Pascal Library Reference	Function reference for the Metrowerks ANS Pascal library
MFC Reference	Reference for the Microsoft Foundation Classes for Win32
Win32 SDK Reference	Microsoft's Reference for the Win32 API
The PowerPlant Book	Introductory guide to the Metrowerks application framework for Mac OS
PowerPlant Advanced Topics	Advanced topics in PowerPlant programming for Mac OS
Targeting Manuals	
Targeting BeOS	How to use CodeWarrior to program for BeOS
Targeting Java VM	How to use CodeWarrior to program for the Java Virtual Machine
Targeting Mac OS	How to use CodeWarrior to program for Mac OS
Targeting MIPS	How to use CodeWarrior to program for MIPS embedded processors
Targeting NEC V810/830	How to use CodeWarrior to program for NEC V810/830 processors
Targeting Net Yaroze	How to use CodeWarrior to program for Net Yaroze game console
Targeting Nucleus	How to use CodeWarrior to program for the Nucleus RTOS
Targeting OS-9	How to use CodeWarrior to program for the OS-9 RTOS
Targeting Palm OS	How to use CodeWarrior to program for PalmPilot
Targeting PlayStation OS	How to use CodeWarrior to program for the PlayStation game console
Targeting PowerPC Embedded Systems	How to use CodeWarrior to program for PPC embedded processors
Targeting VxWorks	How to use CodeWarrior to program for the VxWorks RTOS
Targeting Win32	How to use CodeWarrior to program for Windows