

Metrowerks CodeWarrior

User's Guide

DR1 Sample Documentation

January, 1994

Metrowerks Inc.
1500 du College, suite 300
St. Laurent, Qc
H4L 5G6, Canada
voice: (514) 747-5999
fax: (514) 747-2822
applelink: saleswerks
internet: sales@metrowerks.ca

Metrowerks CodeWarrior © Copyright 1993 by Metrowerks Inc. and its Licensors

All rights reserved.

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

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

About DR1 Sample Documentation

This sample document contains two chapters taken from the Metrowerks CodeWarrior User's Guide included on-line with the Metrowerks CodeWarrior DR1 Bronze and Gold releases. The two chapters, from in order of appearance, are:

- Chapter 3 Tutorial One: The Basics (S1)
- Chapter 14 The Debugger (S2)

About Metrowerks CodeWarrior User's Guide

The Metrowerks CodeWarrior User's Guide is a language-independent User's Guide which details the Metrowerks CodeWarrior Development Environment, demonstrates how to use Metrowerks CodeWarrior through five tutorial chapters, describes how to use Metrowerks CodeWarrior Debuggers, and gives system notes on a variety of issues from calling MPW C functions to porting C code to the PowerPC based Macintosh.

This sample document includes the first tutorial and the chapter which describes how to use Metrowerks CodeWarrior Debuggers.

For More Information

If you have any questions, comments, or would like more information on Metrowerks CodeWarrior DR1 Bronze or Gold, we may be reached at the following location:

Metrowerks Inc.
Attention: Matt Vacaro
Suite 300, 1500 du College
St. Laurent, QC
H4L 5G6 Canada

Voice: (617) 246-4525
(514) 747-599, ext 301

Fax: (514) 747-2822

applelink: saleswerks
internet: sales@metrowerks.ca

Chapter 3 Tutorial One: The Basics

This tutorial shows you essential aspects of the Metrowerks environment. In it you will build a simple application. Later tutorials show more advanced features such as Metrowerks CodeWarrior Debugger and complex text searching techniques. Read through this tutorial before moving on to the other tutorials.

This tutorial assumes that you are familiar with Apple Macintosh programming and the C language.

Tip: C is a case-sensitive programming language. Enter filenames and source code exactly as shown; keep uppercase and lowercase letters consistent with this tutorial's text.

Things you will know

By the end of this tutorial, you will know how to:

- use keyboard shortcuts to Metrowerks commands
- use the Toolbar
- create a new project
- open a source code file
- use the editor window features
- use the Preferences dialog box
- use the Message Window
- compile, make, and run a project into an application

Chapter Outline

Building a small application: Silly Balls.....	3
About Silly Balls.....	3
Things you will do.....	3
Launching Metrowerks.....	3
The About Box.....	3
Keyboard Tips.....	4
The Toolbar.....	4
About Projects.....	5
Creating the Silly Balls Project.....	5
Changing Silly Balls.....	6
Navigating the Editor Window.....	7
The MyPaint() Function.....	8
Customizing the Environment: Preferences.....	11
Find and Replace.....	12
Saving Your Work.....	14

Creating Your Application.....	15
Adding MacOS.lib to the Project.....	15
The Resource File.....	16
Compiling Silly Balls.c.....	16
Egad! Compiler Error!.....	17
Make: Building Better Silly Balls.....	18
Egad! Et tu, Linker?.....	19
Run: (Finally) Seeing the Results of Your Work.....	20
Congratulations!.....	20
What Is Next.....	20

Building a small application: Silly Balls

About Silly Balls

MPW (Macintosh Programmer's Workshop) is Apple Computer's development environment for the Macintosh computer.

Silly Balls is a small program written in MPW C by Apple Macintosh Developer Technical Support. Silly Balls opens a window and draws randomly placed, multi-colored circles.

The source code you will use in this tutorial has already been modified to work with the Metrowerks C compiler.

Things you will do

In this tutorial you will go through all the steps necessary to build a stand-alone, double-clickable application named Silly Balls. The source code is provided but you will modify, compile, and link it.

Launching Metrowerks

This tutorial requires that you work with the Metrowerks C integrated development environment to build the Silly Balls application.

To launch the Metrowerks C environment:

1. Find the SillyBalls project folder and drag it from your CodeWarrior CD (compact disc) to your hard disk.
2. Find the Metrowerks application on your hard disk or CD-ROM.

To start the Metrowerks C application:

3. Double-click its icon.

The application is loaded and launched (Figure 3.1).

Figure 3.1 **Launching Metrowerks**



MW C/68K 1.0d12

The About Box

Before working on Silly Balls, take a look at the **About Metrowerks...** box:

1. Choose **About Metrowerks...** in the Apple () menu.

Watch as the incredible graphics, animation, and sound dazzle you with the names of the intelligent, creative people who work at Metrowerks.

2. Press the mouse button to close the **About Metrowerks...** box.

Keyboard Tips

To use some Metrowerks commands and features you need to use the keyboard with the mouse. Holding a specified key down as you click an object on the screen will give a different result than simply clicking the object. Using the keyboard with the mouse gives you access to useful shortcuts and alternate commands.

For example (Figure 3.2):

1. Click the **Search** menu and look for the **Find Next** Command.
2. Release the mouse button without selecting anything in the **Search** menu.
3. Hold down the Shift key and click the **Search** menu again.

Shift-clicking the Search menu replaces the **Find Next** command with the **Find Previous** command (Figure 3.2).

4. Release the mouse button and the Shift key without selecting anything in the **Search** menu.

Figure 3.2 Example of Shift-clicking



The Toolbar

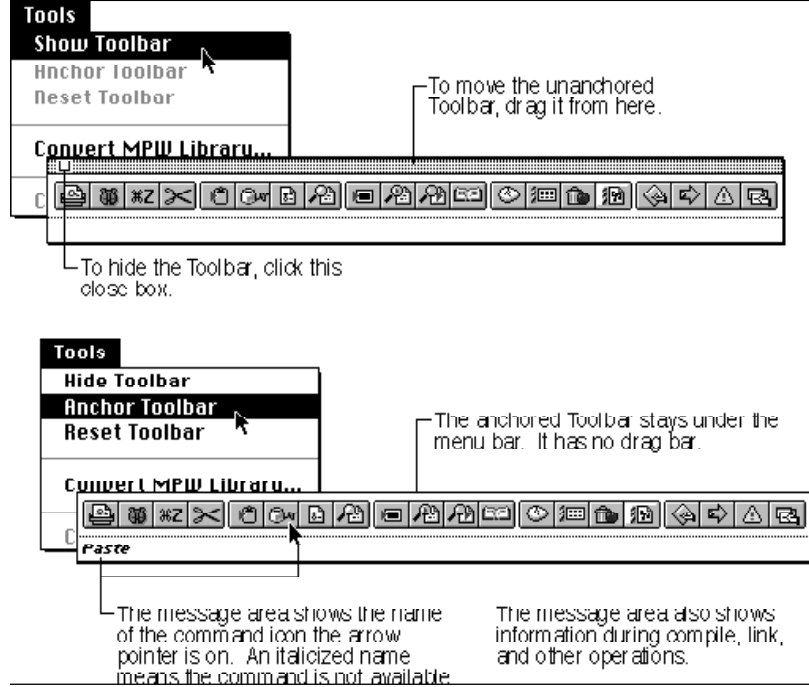
The Toolbar (Figure 3.3) contains a palette of command icons and a message area. You can remove commands from and add commands to the palette. The message area shows status information during compile, link, and search operations.

In these tutorials you can choose commands from the menu bar or from the Toolbar. Whenever you must use a command, its Toolbar icon appears in the margin.

To display the Toolbar:

1. If the Toolbar is not on the screen, choose **Show Toolbar** from the **Tools** menu.
- To anchor the Toolbar:
2. If the Toolbar is not anchored, choose **Anchor Toolbar** from the **Tools** menu.

Figure 3.3 The Toolbar



Tip: Just like menu commands, the keyboard is also used with the mouse to select different commands from the Toolbar. For example, Shift-clicking the Find Next command icon in the Toolbar shows Find Previous in the Toolbar message area.

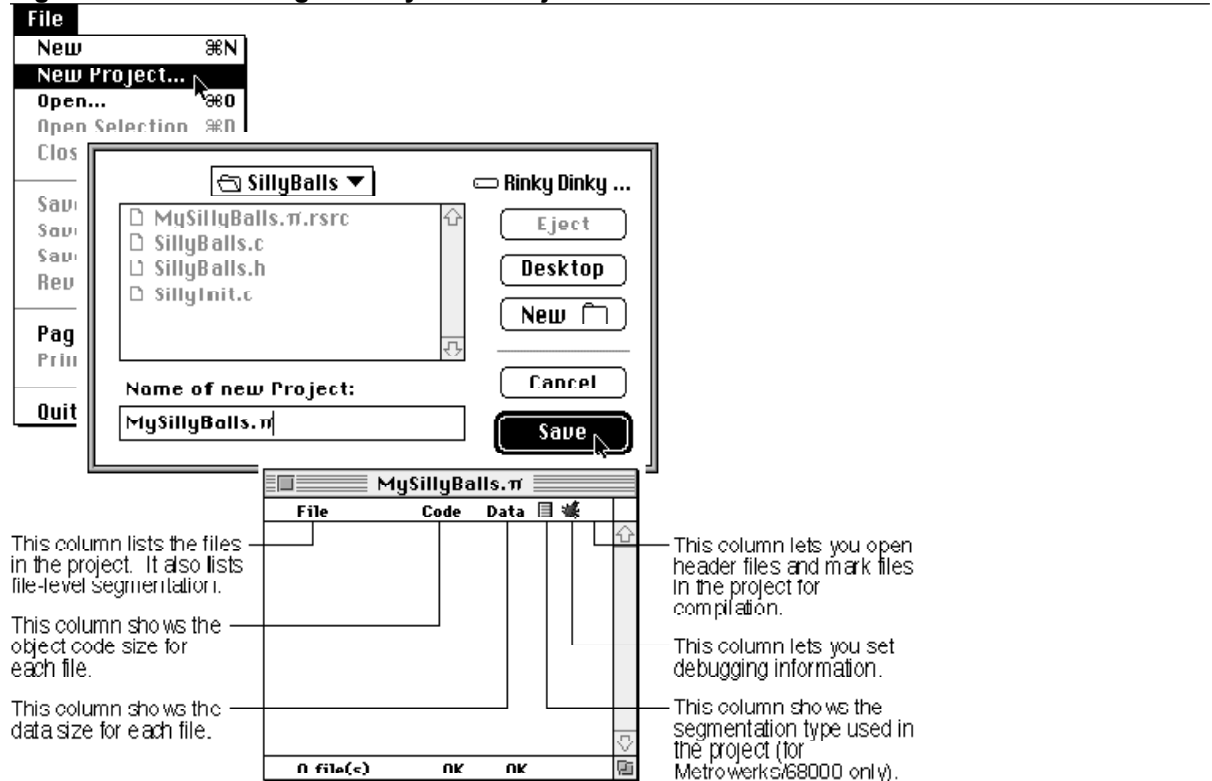
About Projects

A project holds information about the files you need to create your program. Among other things, it holds all of the source code and library file names used in your program as well as the object code compiled from those files.

Creating the Silly Balls Project

To create the Silly Balls application, you will first create a project.

Figure 3.4 Creating the Silly Balls Project



1. Choose **New Project** from the **File** menu.

A dialog box appears prompting you to enter a project name and location on your hard drive (Figure 3.4).

2. Use the dialog box controls to open the Silly Balls project folder.
3. Enter the name `MySillyBalls.n` in the entry field and click **Save**.

A project window appears (Figure 3.4) with the title `MySillyBalls.n`. Although they won't be used in this tutorial, there are controls in the project window to generate debugging information, to segment the program, and open header files.

Later, you will put the files in the project window for creating the Silly Balls application.

Changing Silly Balls

You are going to change Silly Balls to make it draw a more interesting pattern.

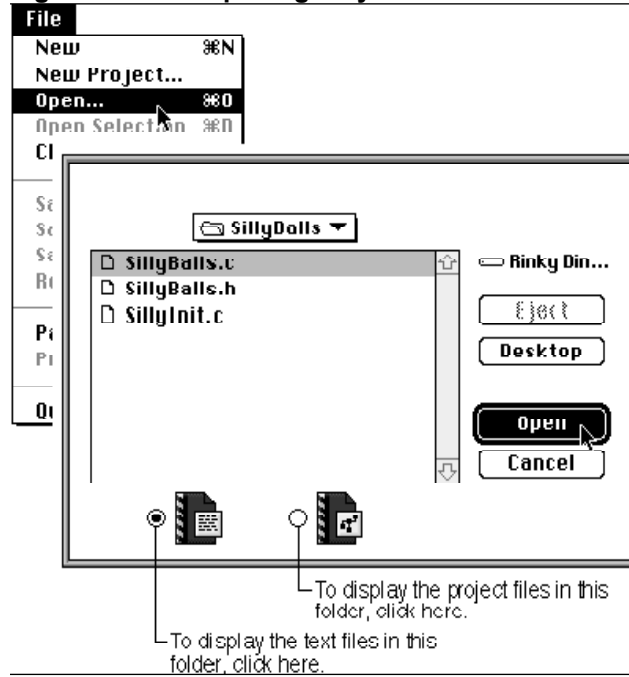


1. Choose **Open** from the **File** menu.

2. Select `SillyBalls.c` and click **Open** (Figure 3.5).

A window appears showing the `SillyBalls.c` source code.

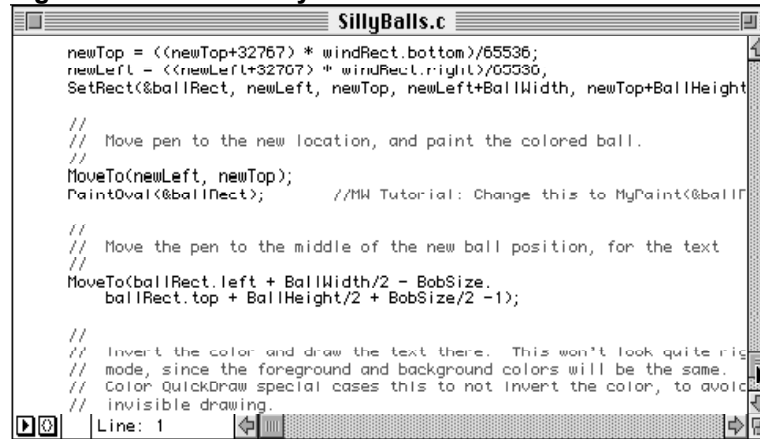
Figure 3.5 Opening SillyBalls.c



Navigating the Editor Window

The editor window lets you edit and view a source code file quickly and easily (Figure 3.6).

Figure 3.6 The SillyBalls.c editor window



To quickly look at all of SillyBalls.c:

1. Slowly drag the vertical scroll box down to the bottom of the vertical scroll bar (Figure 3.6).

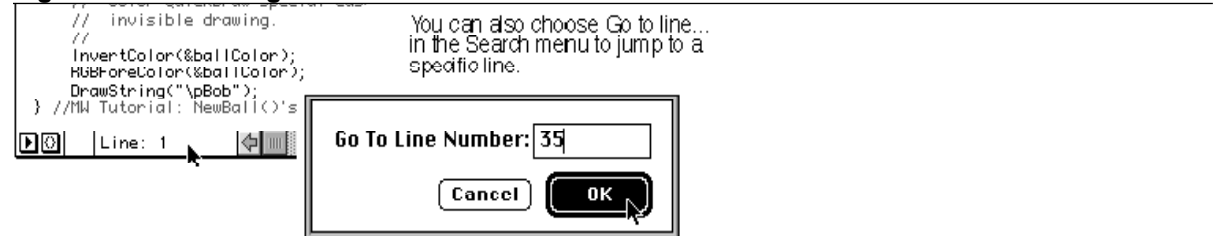
Notice how the text scrolls with the scroll bar as it is moved down. This is the **dynamic scrolling** feature in effect. Notice also that parts of the source code have different colors. This is the **Color Syntax** feature which will be explained later in this tutorial.

To read a description of `SillyBalls.c` (Figure 3.7):

2. Click the Line box at the bottom-left part of the window.
3. Enter 35 and click **OK**.

The window places the insertion point at line 35, showing comments describing what Silly Balls does.

Figure 3.7 Going to line 35

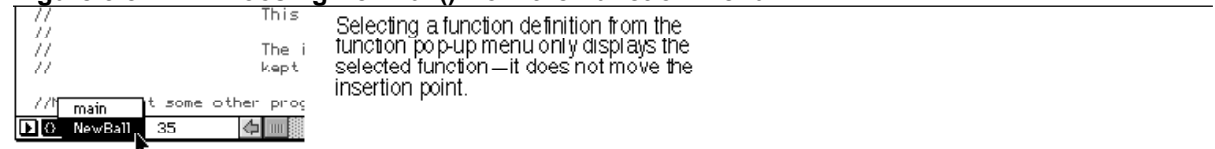


To look at the function you will be changing:

4. Choose `NewBall` from the Function pop-up menu (the `{}` icon in the lower left corner of the window).

The window shows the definition (Figure 3.8) of the function `NewBall()`.

Figure 3.8 Choosing `NewBall()` from the Function menu



The `MyPaint()` Function

The `NewBall()` function picks a random point in the window and draws a circle using the `QuickDraw PaintOval()` routine in a random color with the name "Bob" in it. You will make `NewBall()` more interesting by having it call a new function, `MyPaint()`, which you will enter.

1. Move the insertion point to the end of the last line in the source code file, at the end of the definition for `NewBall()`.
2. Press Return twice, then type this in exactly without pressing the Return key:

```
void MyPaint(Rect *myRect
```

Do you see how the word `void` is a different color (or shade of gray)? This is the **Color Syntax** feature in effect. The **Color Syntax** feature colors keywords differently from the rest of the source code.

Watch the left parenthesis in `MyPaint (Rect` to see the **Balance While Typing** feature in effect:

3. Type the right parenthesis, `)`.

Notice how the left parenthesis is momentarily highlighted as you entered its matching right parenthesis. **Balance While Typing** momentarily highlights a left brace, `{`, bracket `[`, or parenthesis `(` when you enter its matching right partner.

Enter the beginning of the function block:

4. Press Return, type an opening brace, `{`, and press Return again.
5. Press the Tab key, type the following variable declaration, then press Return:

```
    RGBColor myColor;
```

When you press Return, notice how the insertion point is on the next line, aligned with the indented text above it. This is the **Auto-indent** feature in action. **Auto-indent** automatically indents source code text at the indentation level of the last line entered. You can back up the insertion point by pressing the Delete key.

Now you can enter the rest of the function body:

6. Press Return and enter the rest of the source code for the function exactly as it appears here, terminating each line with a press of the Return key:

```
    // Get the foreground color
    GetForeColor (&myColor);

    // Invert the color
    InvertColor (&myColor);
    RGBForeColor (&myColor);

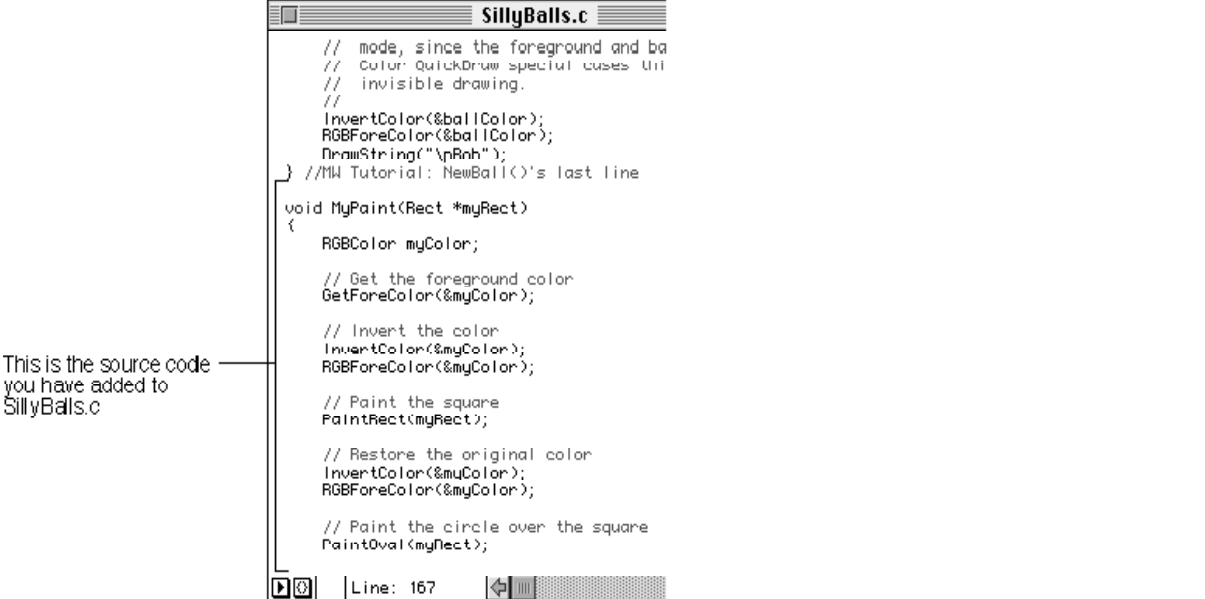
    // Paint the square
    PaintRect (myRect);

    // Restore the original color
    InvertColor (&myColor);
    RGBForeColor (&myColor);

    // Paint the circle over the square
    PaintOval (myRect);
```

Notice that comments (text preceded by `//`) are colored differently from the rest of the source code. Again, this is the **Color Syntax** feature in action.

Your new function should look like the source code entered in Figure 3.9.

Figure 3.9 Fresh MyPaint()

```

// mode, since the foreground and ba
// Color: QuickDraw special cases UI
// invisible drawing.
//
InvertColor(&ballColor);
RGBForeColor(&ballColor);
DrawString("\pRnh");
} //MM Tutorial: NewBall()'s last line

void MyPaint(Rect *myRect)
{
    RGBColor myColor;

    // Get the foreground color
    GetForeColor(&myColor);

    // Invert the color
    InvertColor(&myColor);
    RGBForeColor(&myColor);

    // Paint the square
    PaintRect(myRect);

    // Restore the original color
    InvertColor(&myColor);
    RGBForeColor(&myColor);

    // Paint the circle over the square
    PaintOval(myRect);
}

```

This is the source code you have added to SillyBalls.c

Now you must enter the function prototype at the beginning of the source code file so the compiler recognizes the data types passed to and returned by `MyPaint()`.

7. Place the insertion point after the prototype for `NewBall()`, near the beginning of the document.
8. Enter `void MyPaint(Rect *)`; and press Return (Figure 3.10).

Figure 3.10 Entering the MyPaint() prototype

```

//MW ** Metrowerks note **
// All changed code by Metrowerks i
// There is one type of modificatio
// • Added argument type and return
// In order to pass with extended
//
// 8/31/93 JA

//MW Tutorial ** Metrowerks Tutorial
// All changed code for the Metrower

// MW tutorial
#include <MacHeaders>
#include "SillyBalls.h"

/* Globals */
Rect windRect;

/* Prototypes */
extern void Initialize(void); //MW
void NewBall(void); //MW
void MyPaint(Rect *);

//
// Main body of program SillyBalls
//

//MW specified argument and return ty
void main(void)

```

This is the function
prototype you entered.

Line: 72

Customizing the Environment: Preferences

Not only can you customize the Toolbar, you can also change parts of the editor to suit the way you work. You can customize aspects of the editor and the document you are working on with the **Font Preferences** and the **Editor Preferences**.

To examine the current **Font Preferences**:

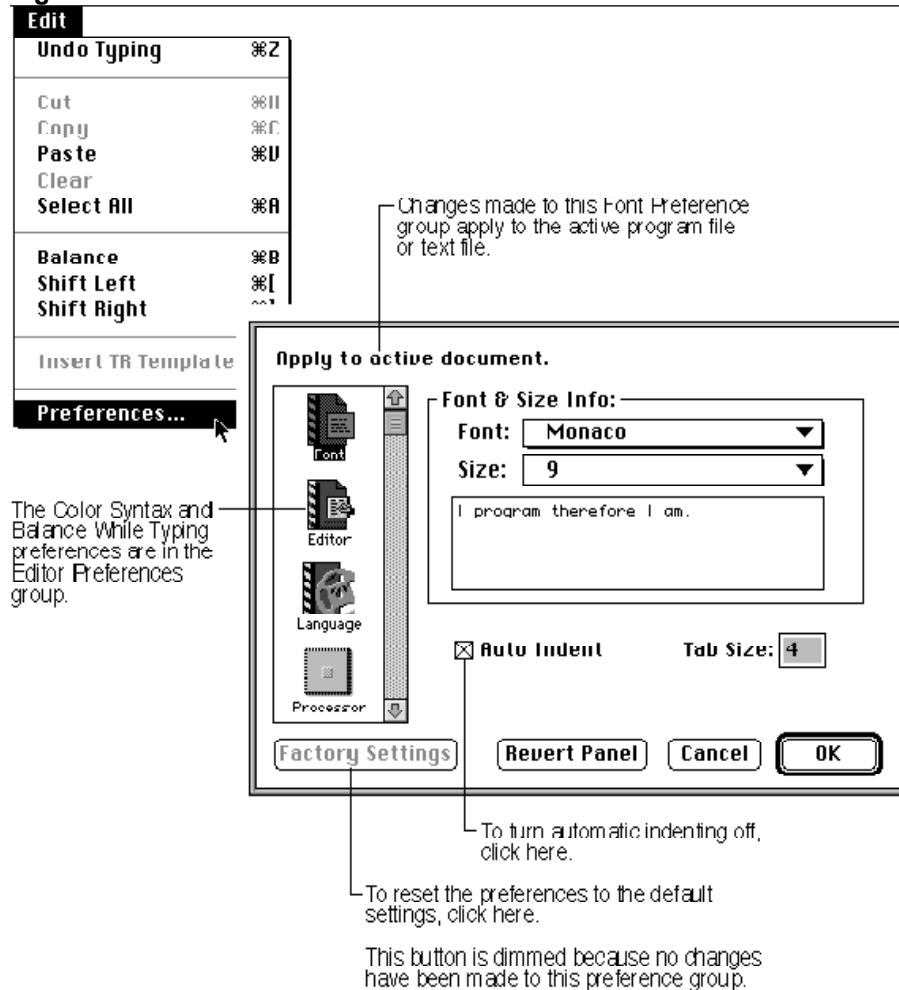
1. Choose **Preferences...** from the **Edit** menu.

The **Preferences** dialog box appears.

2. Choose the **Font** icon in the scroll window of the dialog box (Figure 3.11).

The **Font Preferences** group appears.

Figure 3.11 Font Preferences



To leave the **Preferences** dialog box without saving changes made to the preferences:

3. Click **Cancel**.

Other groups of preferences are examined in later tutorials.

Find and Replace

To use your new function, the `PaintOval()` call in the `NewBall()` function must be changed. In `NewBall()`, replace the call to `PaintOval()` with `MyPaint()`.



1. Choose **Find...** from the **Search** menu.

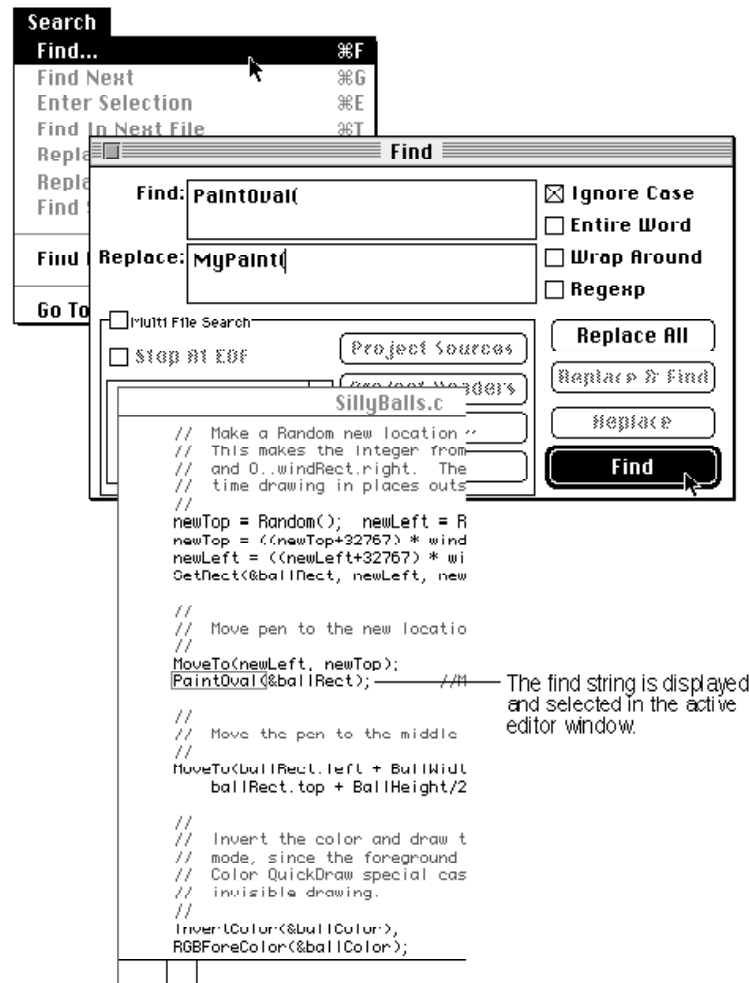
A dialog box appears prompting you to enter a search string and a replace string (Figure 3.12).

2. Enter `PaintOval(`

This is the string to search for.

3. Press `Tab` to move to the insertion point to the replacement entry field and enter `MyPaint(`
4. Click **Find** (Figure 3.12).

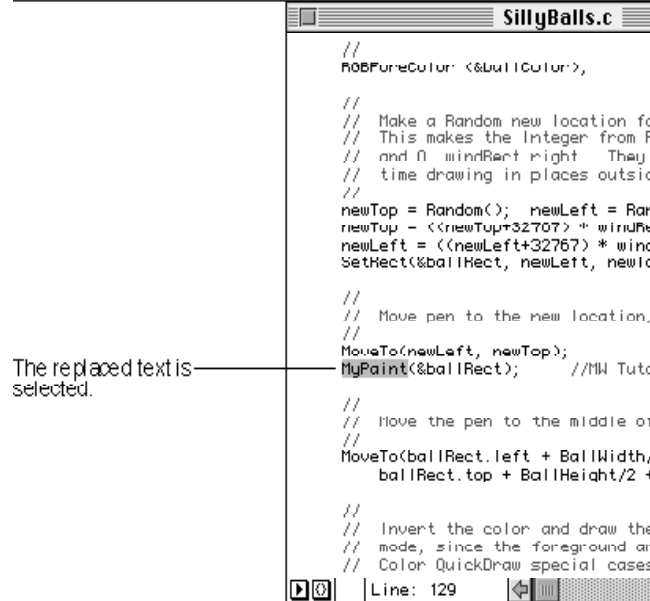
Figure 3.12 Finding and replacing



The search begins at the insertion point. Then, the first occurrence of `PaintOval(` is displayed and selected (Figure 3.12). You may have to drag the Find window to see the text the editor found.

5. Click **Replace** (Figure 3.13).

`PaintOval(` is replaced with `MyPaint(`, shown in Figure 3.13.

Figure 3.13 Replacing the call to PaintOval()

To close the Find dialog box:

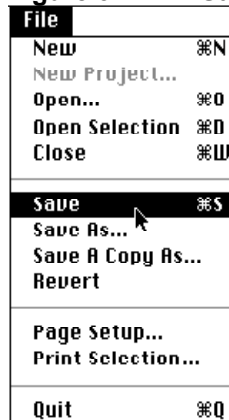
6. Click the close box of the Find window.

Saving Your Work

To save the changes you made to the source code:



1. Choose **Save** from the **File** menu (Figure 3.14).

Figure 3.14 Saving SillyBalls.c

Creating Your Application

Before it can be compiled and made into an application, `SillyBalls.c` must be added to the project.

1. Make sure `SillyBalls.c` is the active window.



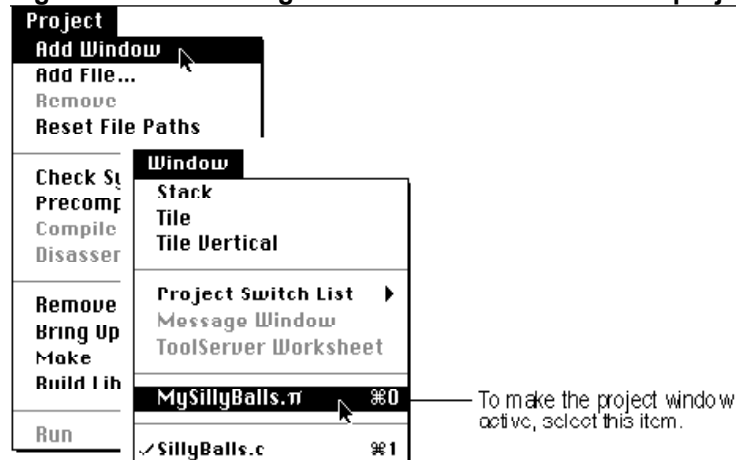
2. Choose **Add Window** in the **Project** menu.

Because the `SillyBalls.c` source code window is over the project window, you can't see that it has been added to the project.

To see the project window:

3. Choose `SillyBalls.p` from the **Window** menu (Figure 3.15).

Figure 3.15 Adding a source code window to the project



Adding MacOS.lib to the Project

You may have noticed that some function calls in `SillyBalls.c` are not defined in `SillyBalls.c`. These functions, like `PaintOval()`, and `GetForeColor()` for example, are Macintosh Toolbox routines. The linker needs to know how to get to these Toolbox routines. `MacOS.lib` is a library that does this: it contains object code that allows access to the Macintosh Toolbox.

To add `MacOS.lib` to the project:



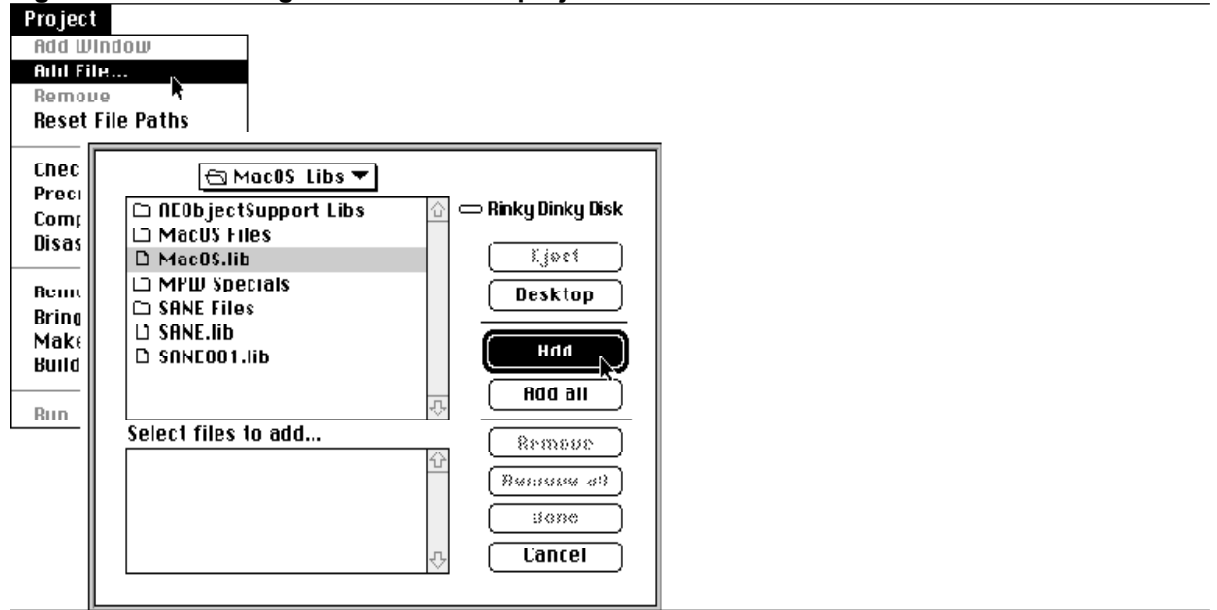
1. Choose **Add File...** from the **Project** menu.

A dialog box appears prompting you to choose files to add to the project (Figure 3.16).

2. Find and open the `Libraries f` folder in the `Metrowerks C/C++ 68K` folder.

3. Next, open the MacOS-Libs folder.
4. Select MacOS.lib and click **Add** (Figure 3.16),
5. Click **Done**.

Figure 3.16 Adding MacOS.lib to the project



The Resource File

If a resource file has the same name as the project with “.rsrc” added to it, it is automatically added to the project (My SillyBalls.p.rsrc, for example). Such a resource file must be located in the project’s folder.

The project window shows two files: SillyBalls.c, MacOS.lib (Figure 3.17).

Figure 3.17 The project window



Compiling Silly Balls.c

To compile SillyBalls.c:

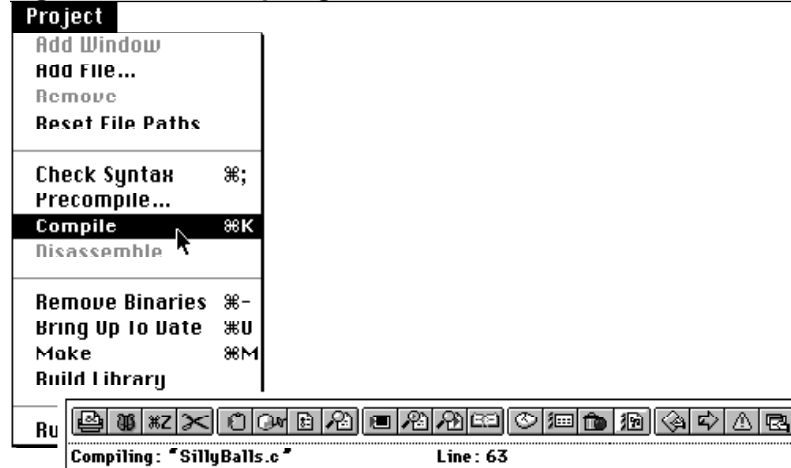
1. Make sure the SillyBalls.c window is active or SillyBalls.c is selected in the project window.



2. Choose **Compile** from the **Project** menu.

The message area in the Toolbar shows the compiler's progress (Figure 3.18).

Figure 3.18 Compiling a source code file

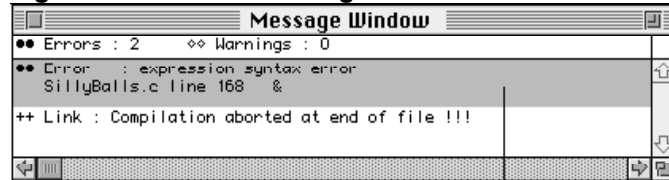


And then. . .

Egad! Compiler Error!

A window appears indicating that there has been a compiler error (Figure 3.19). This window is called the **Message Window**. In this case, the **Message Window** gives you access to the compiler errors that have occurred.

Figure 3.19 The Message Window



To open the source code file and select the source code that caused this error, double click or Option click this message.

In this case, the source code that caused the error is already selected because it is the only compiler error.

The **Message Window** is especially useful when many compiler errors occur in a project with many files. With the **Message Window** you can quickly jump to the source code statements that caused the errors.

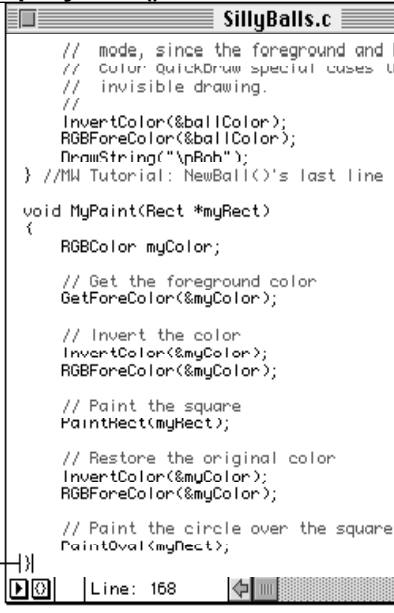
In this case, though, only one error occurred. To fix the error:

1. Click the `SillyBalls.c` window to make it active.

If you entered it exactly as shown earlier in Figure 3,9, the function `MyPaint()` has an intentionally missing closing brace that caused the compiler error.

2. Add the missing brace, `}`, as shown in Figure 3.20.

Figure 3.20 Touching up MyPaint()



```

// mode, since the foreground and b
// Color: QuickDraw special cases UI
// invisible drawing.
//
InvertColor(&ballColor);
RGBForeColor(&ballColor);
DrawString("\pAh");
} //M4 Tutorial: NewBall()'s last line

void MyPaint(Rect *myRect)
{
    RGBColor myColor;

    // Get the foreground color
    GetForeColor(&myColor);

    // Invert the color
    InvertColor(&myColor);
    RGBForeColor(&myColor);

    // Paint the square
    PaintRect(myRect);

    // Restore the original color
    InvertColor(&myColor);
    RGBForeColor(&myColor);

    // Paint the circle over the square
    PaintOval(myRect);
}

```

This is the right brace you entered after the last statement in `MyPaint()`.

To save the correction:

3. Choose **Save** from the **File** menu.

Make: Building Better Silly Balls

Now you are ready for the second attempt to create the Silly Balls application. Instead of using the **Compile** command, you will use the **Make** command. The **Make** command builds a project into an application, code resource, or shared library. It compiles any source code files that have changed or marked for compilation, and links all the libraries, resource files, and compiled object code together to make a program.

To attempt to build the project:

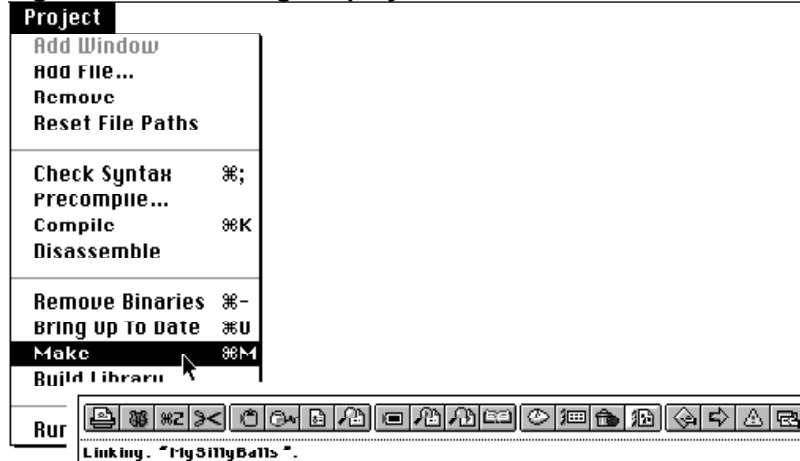


1. Choose **Make** from the **Project** menu.

The file begins compiling and the message area in the Toolbar shows the compiler's progress. Then the linker begins linking the functions in `SillyBalls` (Figure 3.21).

Note: If compiler errors occur, carefully check the `MyPaint()` function definition at the end of `SillyBalls.c` (Figure 3.9), check the `MyPaint()` prototype (Figure 3.10), and check the call to `MyPaint()` in `NewBall()` (Figure 3.13).

Figure 3.21 Making the project

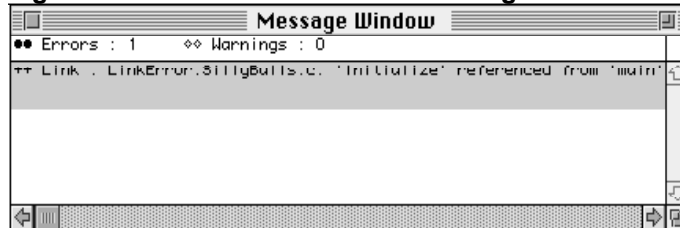


And then. . .

Egad! Et tu, Linker?

The Message Window appears again, this time indicating a linker error. There is one function call, `Initialize()`, that is not defined in `MySillyBalls.c`. The linker could not find the definition for `Initialize()`, so it signaled an error (Figure 3.22).

Figure 3.22 The return of the Message Window



Fortunately, this error is easily fixed:



1. Use the **Add File...** command to include the `SillyInit.c` file in the project (Figure 3.23).

`SillyInit.c`, in the `SillyBalls` folder, defines the `Initialize()` function.

Figure 3.23 The project window with SillyInit.c

File	Code	Data
1 SillyDalls.c	266	12
SillyInit.c	0	0
MacOS.lib	30962	0

3 file(s) 30K 0K

Run: (Finally) Seeing the Results of Your Work

Your project is now ready to be compiled and linked for the final time. If everything was entered correctly, the project will be built into an application.



1. Choose **Run** from the **Project** menu.

The **Run** command performs a **Make** command, then runs the application as if it were double clicked from the Finder desktop.

`SillyBalls` executes until you press the mouse button:

2. Press the mouse button.

Control returns to the Metrowerks environment.

Congratulations!

You have successfully created a project, modified a source code file, and built an application with the Metrowerks C environment.

What Is Next

The tutorials that follow Tutorial One will show you more advanced aspects of the Metrowerks environment. While this tutorial covered many aspects of the Metrowerks environment, the later tutorials focus on specific commands and features you can use to accomplish certain tasks.

Figure 3.24 lists more information on the commands and features covered in this tutorial.

Figure 3.24 For more information...

For more information Consult this part of the User's Guide on...

Keyboard shortcuts	“Things You Should Know” in Chapter 1
The Toolbar	“The Toolbar” in Chapter 12

Searching	“Replacing Text” and “Find” in Chapter 10
Preferences	“Preferences” in Chapter 9
Building a project	“Add File” and “Add Window” in Chapter 11
Editor commands	“More Text Editor Notes” in Chapter 9
Message Window	“Message Window” in Chapter 13
The editor window	“Opening a Program File” in Chapter 8

The Metrowerks CodeWarrior Debugger monitors the execution of your program. With it you can execute your program one statement at a time or you can execute it normally and stop at points you specify. When the debugger stops a program, you can examine variable values, the chain of function calls, and the source code files in your project.

Chapter Outline

Installing the Debugger Nubs.....	24
Preparing a Project for the Debugger.....	25
Metrowerks Environment Preferences.....	25
The SYM File.....	25
Generating SYM information.....	25
Starting the Debugger.....	26
Launching the Debugger.....	26
Debugger Windows.....	26
The Program Browser Window.....	26
The File Browser Window.....	28
The Function Icon.....	28
Breakpoints.....	29
What a Breakpoint Does.....	29
Setting and Clearing Breakpoints.....	29
The File Menu.....	30
Open.....	30
Close.....	30
Quit.....	30
The Edit Menu.....	30
Undo.....	30
Cut.....	31
Copy.....	31
Paste.....	31
Clear.....	31
Select All.....	31
The Control Menu.....	31
Run.....	31
Stop.....	32
Kill.....	32
Step Over.....	33
Step Into.....	33
Step Out.....	33
Set Breakpoint.....	34
Clear All Breakpoints.....	34
Switch to Monitor.....	34
The Data Menu.....	34
Show Types.....	34
Expand.....	34
Collapse All.....	34

Copy to Collection.....	34
Open Window.....	35
Pointer.....	35
Array.....	35
View as.....	35
Signed Decimal.....	36
Unsigned Decimal.....	36
Hexadecimal.....	36
Character.....	36
C String.....	37
Pascal String.....	37
Floating Point.....	37

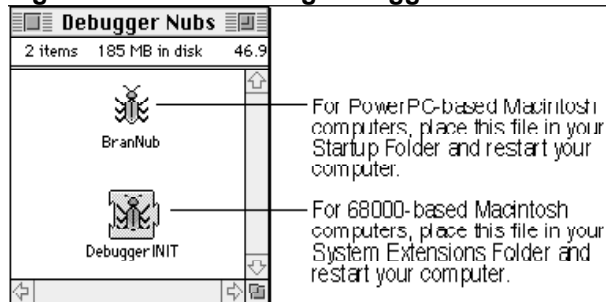
Installing the Debugger Nubs

The debugger for 68000-based Macintosh computers requires DebuggerINIT in the System Extensions Folder. The debugger for PowerPC-based Macintosh computers requires the BranNub application to be launched before launching the debugger.

These two files, DebuggerINIT and BranNub, are debugger nubs included with the CodeWarrior CD (Figure 14.1). A debugger nub that provides low-level services for the debugger.

Tip: To make sure the BranNub is always launched before the debugger, place BranNub in your Startup Folder. Restart your computer so BranNub is automatically launched at start-up time.

Figure 14.1 Installing debugger nubs



Developers Notes:

DebuggerINIT intercepts the `SysBreak()` and `SysBreakStr()` functions only. It does not currently intercept the `Debugger()` or `DebugStr()` functions.

The BranNub debugger nub intercepts the `Debugger()` and `DebugStr()` functions. It does not intercept the `SysBreak()` or `SysBreakStr()`.

Currently, the debugger only debugs applications (68000 and PowerPC) and shared libraries (PowerPC). Code resources (68000) are not supported yet.

This release of the debugger supports C, C++, and in-line assembly for the 68000 in C/C++. It does not yet support Pascal.

Preparing a Project for the Debugger

The debugger requires certain information about your program. Before using the debugger to monitor your program, you must prepare it from the C, C++, or Pascal environment. Follow the instructions in this section to generate the information the debugger needs.

Metrowerks Environment Preferences

In **Linker Preferences** for the 68000 environment, select the **Generate A6 Stack Frames**.

Consult Preferences... in Chapter 9 for more information on Linker and Project Preferences dialog box.

In the 68000 and PowerPC environment **Linker Preferences**, select **Generate SYM File** checkbox. Selecting the **Full Path in SYM Files** checkbox is recommended so the debugger can locate project files that are not in the project's folder. If you do not select **Full Path in SYM Files**, the debugger may prompt you to locate these files.

In **Project Preferences** for the 68000 and PowerPC environments, choose the **Can background** item in the **Size Flags** pop-up menu.

The SYM File

A SYM file contains symbolic information about the files in your project. As well as other pieces of information, the SYM file contains the names of functions and variables, their locations within the source code, and their locations within the application object code.

The debugger uses this symbolic information to present your program as C, C++, or Pascal statements and variables instead of assembly-language instructions and memory addresses.

The debugger expects the SYM file to have the same name as your source code file with ".SYM" appended to it. For example, if you want to debug the `Abstract Painter` application, the debugger requires a SYM file with the name `Abstract Painter.SYM`.

Generating SYM information

Consult Generating SYM Info in Chapter 11 for more information.

The linker optionally creates the SYM file the debugger requires. From the Metrowerks environment, select the **Generate SYM info** marker in the project window of each source file you want to use with the debugger and set the preferences discussed earlier.

<p>Note: If a source code file is not marked to generate SYM information from the project window, the debugger cannot display any information about the file while it is executing.</p>
--

Starting the Debugger

Launching the Debugger

There are three ways to launch the debugger:

Consult the debugger's Open command for information on opening a SYM file to debug a program.

- (a) Double-click a SYM file.
- (b) Double-click the debugger icon to launch the debugger application. When you launch the debugger this way, a dialog box appears, letting you open a SYM file.
- (c) Drop a SYM file onto the debugger icon. When launched this way, the debugger opens the SYM file.

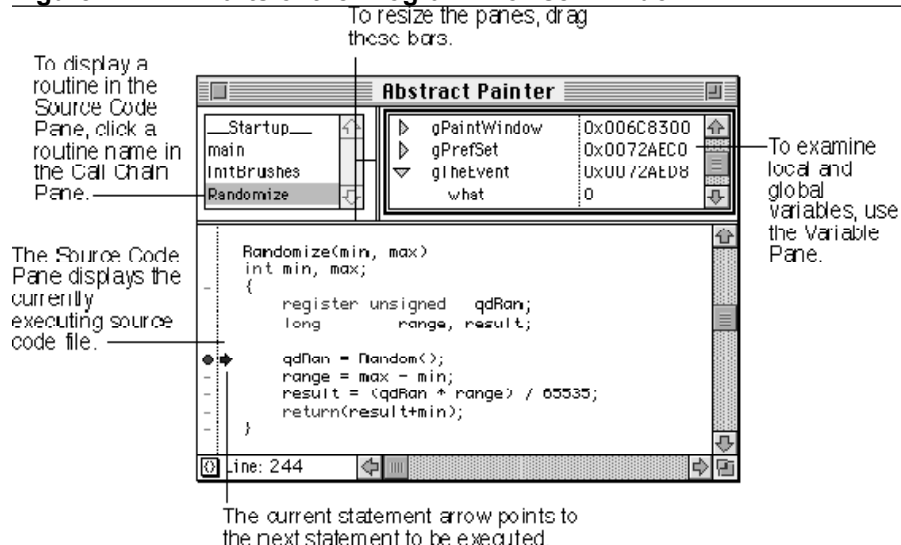
Debugger Windows

The debugger presents information about the target program (the program being debugged) using the Program Browser Window and the File Browser Window.

The Program Browser Window

The **Program Browser Window** displays debugging information about the source code file in your project containing the currently running function.(Figure 14.2).

Figure 14.2 Parts of the Program Browser Window



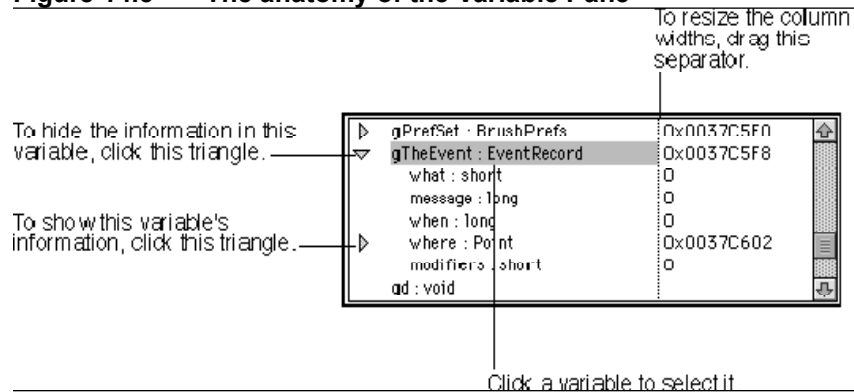
The **Source Code Pane** shows the currently executing source code file (Figure 14.2). The debugger takes the source code directly from the project's source code files, including any comments and white space. The Pane shows C,

C++, Pascal, and in-line assembly code exactly as it appears in your program's source code.

If more than one function call is on a line, each function 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.

The **Call Chain Pane** in the Program Browser Window shows the current subroutine calling chain (Figure 14.2). A function appears below the function that called it.

Figure 14.3 The anatomy of the Variable Pane



Consult debugger's Expand, Collapse, and Collapse all commands later in this chapter.

The **Variable Pane** shows a list of local and global variables and their values (Figure 14.3). The local variables listed in the Variable Pane belong to the currently executing function.

The Variable Pane lists the variables in outline form, like a list view in the Finder. Click the triangle next to an entry to show or hide the entries inside it. For example, in Figure 14.3, clicking the right-pointing arrow next to `gTheEvent` (which is inside the *Globals for Abstract Painter* entry) displays its members. Clicking on the arrow again hides the C struct members.

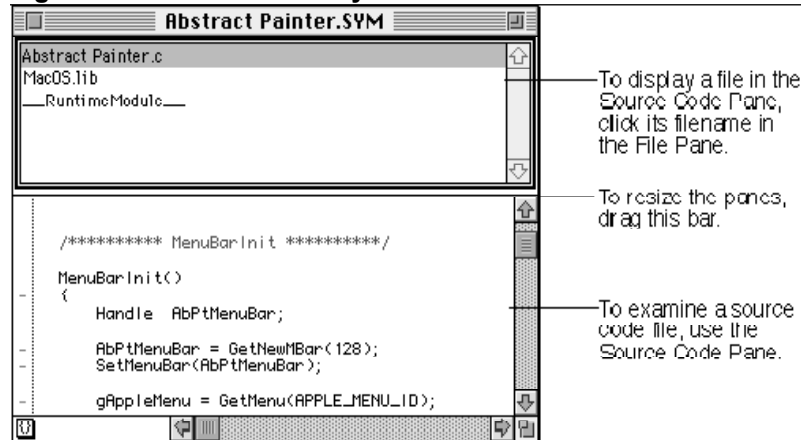
Note: There is no register or memory display in the Variable Pane when stepping through in-line assembly code,

Developers Note:

Currently, the debugger only displays static local and external global variables in the Variable Pane. The debugger does not display static global variables in this release.

The File Browser Window

Figure 14.4 The anatomy of the File Browser Window



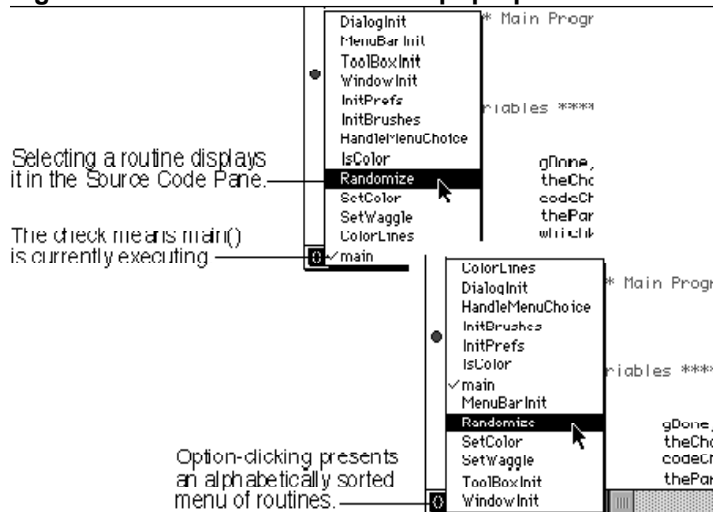
The **File Browser Window** lets you to set breakpoints and displays your project's source code files (Figure 14.4). You can only view source code files in the File Browser Window; library files are not displayed.

Note: Only source code files can be displayed in the Source Code Pane.

The **Source Code Pane** in the File Browser Window shows the source code file selected in the File Pane.

The Function Icon

Figure 14.5 The Function Icon pop-up menu



The **Function** pop-up menu, at the bottom-left corner of the Source Code Panes of the Program Browser and File Browser Windows, contains a list of the functions in the source file in the Source Code Pane (Figure 14.5). Selecting a function in the **Function** menu displays it in the Source Code Pane. Option-clicking the **Function** menu presents an alphabetically sorted menu.

Breakpoints

What a Breakpoint Does

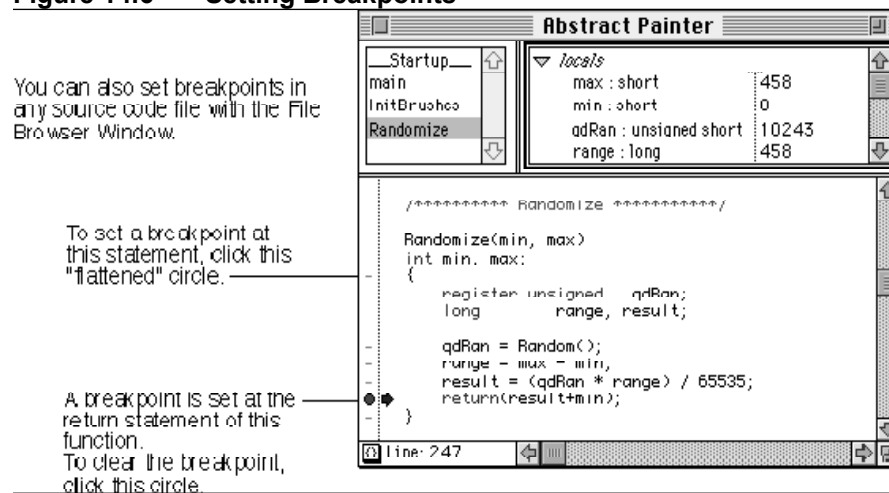
A **Breakpoint** marks a statement to suspend the target program's execution and return control to the debugger. When the debugger reaches a statement with a breakpoint it stops the program before the statement is about to execute.

Setting and Clearing Breakpoints

Consult the debugger's Set Breakpoint and Clear All Breakpoints commands later in this chapter.

From the Source Code Panes of the Program Browser and File Browser Windows, you can set a breakpoint for any statement in any source code file in the project (Figure 14.6).

Figure 14.6 Setting Breakpoints



Tip: Not only is your code easier to read if you put only one statement on a line of source code, it is easier to debug because the debugger allows one breakpoint per line of source code, no matter how many statements a line may have.

The File Menu

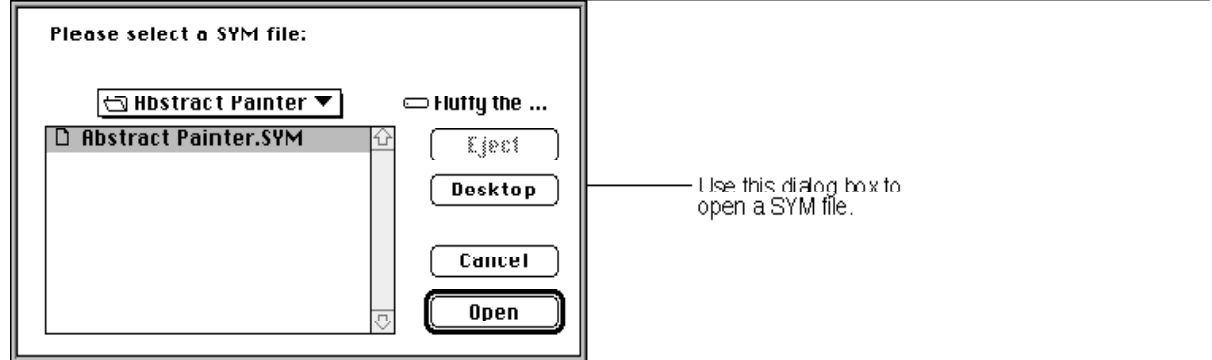
Open

Consult The SYM File earlier in this chapter for information on generating a SYM file for your project.

Opens an existing SYM file to debug a program. The dialog box in Figure 14.7 appears, prompting you to select a SYM file. The SYM file must be in the same folder as its target program (the program you want to debug).

Once you choose the SYM file, the debugger loads it into memory, loads the target program, places a breakpoint at the main entry point of the program, then launches the program. The debugger then pauses the program at the initial breakpoint, returning control to the debugger.

Figure 14.7 Opening a SYM file



Close

Stops the target program and closes the SYM file and the debugger windows.

Quit

Stops and closes the current target program and quits the debugger.

The Edit Menu

Undo

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

Cut

Deletes the selected text and puts it in the Clipboard. You cannot cut selected source code from a Source Code Pane.

Copy

Copies the selected text into the Clipboard.

Paste

Pastes text in the Clipboard into the active window. You cannot paste the Clipboard contents into a Source Code Pane.

Clear

Deletes the selected text without placing it in the Clipboard. You cannot clear selected source code from a Source Code Pane.

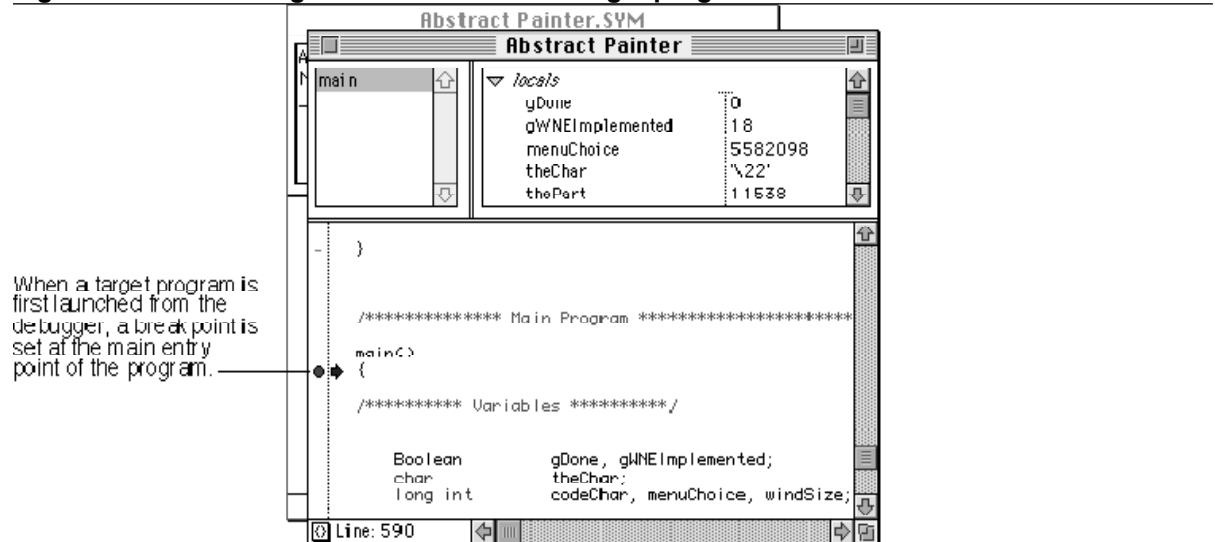
Select All

Selects all of the text in the active window.

The Control Menu**Run**

Executes the target program starting at the current statement arrow until a breakpoint is reached or you issue a **Stop** or **Kill**, or **Close** command. If the target program reaches a breakpoint or you issue a **Stop** command, the debugger regains control and the Program Browser window appears showing the current statement arrow and the current values of local and global variables.

Using the **Run** command after the **Kill** command executes the program from its beginning. The debugger places a breakpoint at the program's main entry point. It then starts running the program, pausing at the initial breakpoint (Figure 14.8).

Figure 14.8 Starting the execution of the target program

Stop

Suspends the execution of the target program and returns control to the debugger. The Program Browse window appears showing the current values of the local and global variables and the current statement arrow pointing to the next statement to execute. **Stop** is dimmed in the **Control** menu and a message appears in the Program Browser Window's Source Code Pane when a program is stopped.

To continue executing a stopped target program you can:

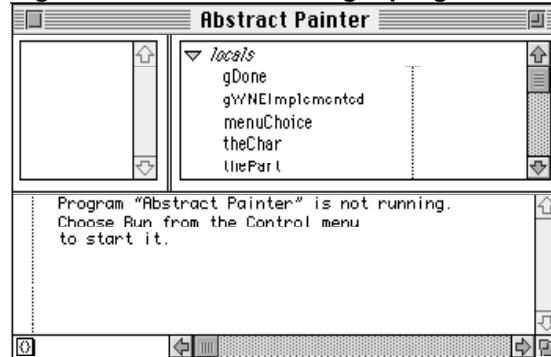
- (a) continue the normal execution of the target program with the **Run** command. Execution will continue beginning at the current statement arrow.
- (b) step through the target program one statement at a time with the **Step Over**, **Step Into**, and **Step Out** commands in the **Control** menu.

Note: If the target program is running in the foreground, **Stop** will not work. **Stop** works only if the target program regularly calls the `WaitNextEvent/GetNextEvent` Macintosh Toolbox routines.

Kill

Stops the execution of the target program and returns control to the debugger. The target program's execution is killed, not suspended (Figure 14.9). While a breakpoint and the **Stop** commands allow you to resume program execution from the point where it was stopped, **Kill** requires **Run** to restart a program. Using **Run** after using **Kill** starts the program from its main entry point.

Figure 14.9 A Killed target program



Step Over

Executes a single statement, stepping over function calls. The statement pointed to by the current statement arrow is executed and control returns to the debugger. When the debugger reaches a function call, it executes the function without displaying it in the Program Browser Window. In other words, the **Step Over** command does not go deeper into the call chain. However, **Step Over** does follow execution to a function's caller when a function terminates.

Step Into

Executes a single statement, stepping into function calls. The statement at the current statement arrow is executed and control returns to the debugger. The debugger follows function calls, showing the execution of the called function in the Program Browser Window. Unlike **Step Over**, the **Step Into** follows the program flow deeper into the call chain when it executes a function call.

Step Out

Executes the rest of the current function until it exits to its caller. Unlike the **Step Over** and **Step Into** commands, **Step Out** executes the program normally from the statement at the current statement arrow and returns control to the debugger when the function returns to its caller.

Tip: Functions with no debugging information, such as library functions, are not displayed in the Source Code Pane. Use **Step Out** to execute and exit functions that have no debugging information.

Set Breakpoint

Set a breakpoint at the selected source code statement. **Set Breakpoint** sets a breakpoint only if the selection begins and ends on the same statement.

Clear All Breakpoints

Clears all the breakpoints in all the source code files in the target programs being debugged.

Switch to Monitor

Gives control to the Macintosh ROM Monitor program and any low-level debugger you may have installed on your computer.

The Data Menu

Show Types

Shows the data types of the local and global variables in the Variable Pane of the Program Browser Window.

Expand

Displays the C member, C++ data member, or Pascal field variables inside the selected structured variable or dereferences the selected pointer in the Variable Pane of the Program Browser Window.

Collapse All

Hides all variables so that only the *locals* and *Globals* entries in the Variable Pane appear.

Copy to Collection

Copies the variable selected in the Variable Pane to the Collection Window. You can also drag and drop variables from the Variable Pane to the Collection Window

Developers Note:

This command is not yet implemented.

Open Window

Creates a separate window to display the selected variable. Open Window is useful to monitor the values of large structured variables (Pascal `records` or C/C++ `structs`).

Developers Note:

This command is not yet implemented.

Pointer

Views a variable as a pointer. This command is checkmarked when a pointer variable is selected in the Variable Pane of the Program Browser Window.

Array

Views an array in a separate window.

Developers Note:

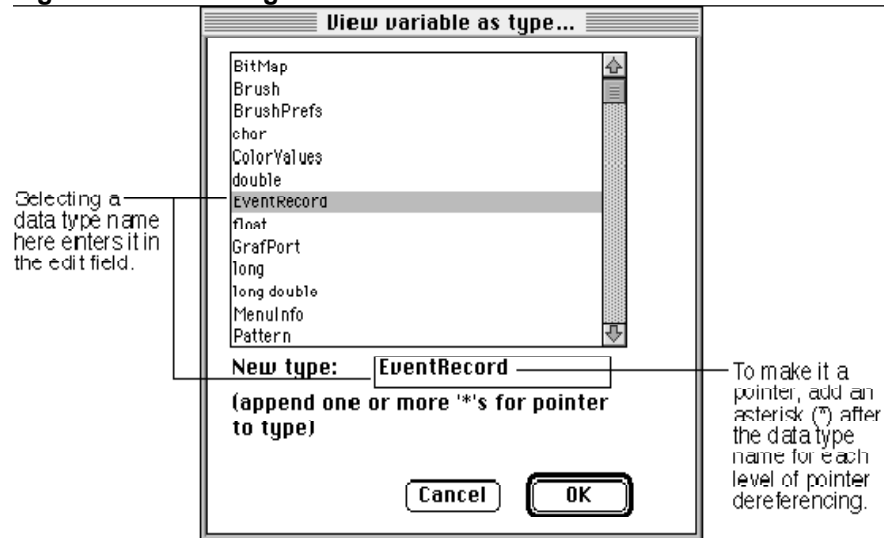
This command is not yet implemented.

View as...

Views the selected variable as a different type. This command interprets the selected variable in the Variable Pane as a different data type.

Memory variables can be viewed as any data type. The debugger ignores excess memory if the new data type is smaller than the variable's original type. The debugger reads more memory if the new type is larger than the original type. A register variable can only be viewed as a type that has the same size as the register.

To use **View as...**, first select a variable in the Variable Pane. Choose the **View as...** command in the **Data** menu. A dialog box appears showing a list of all the data type names defined in the project (Figure 14.10). Choosing a data type enters it in the edit field. If you want the selected variable to be interpreted as a pointer, append an asterisk (*) to the data type name. Click **OK** to see the variable as a variable of the type chosen.

Figure 14.10 Using View as...

Signed Decimal

Views the selected variable in the Variable Pane as a signed decimal value.

Unsigned Decimal

Views the selected variable in the Variable Pane as an unsigned decimal value.

Hexadecimal

Views the selected variable in the Variable Pane as a hexadecimal value.

Character

Views the selected variable in the Variable Pane as a character value.

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

C String

Views the selected variable in the Variable Pane as a C character string. A C character string is a sequence of ASCII characters terminated by a null character (`'\0'`). The string is displayed without the null character.

Consult the debugger's Character command for information on non-printable characters.

Pascal String

Views the selected variable in the Variable Pane as a Pascal character string. A Pascal character string consists of 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.

Floating Point

Views the selected variable in the Variable Pane as a floating point value.