Home

Using Hawk™ Macros

Version 2.5

RadiSys
THE POWER OF WE

www.radisys.com

Revision A • July 2006

Copyright and publication information

This manual reflects version 2.5 of Hawk.
Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Contents

1 Introduction

A macro may be defined as something simple that represents something bigger or more complex. Under this definition, macros include everything from keystroke recordings that are assigned to keys, the % macros Hawk™ uses in command lines, up to more sophisticated things written in a programming language.

A Macro Language provides a method of creating macro source code, which may contain control structures and variables, and is normally interpreted at runtime. Hawk™ has several macro languages from which to choose.

Choosing a Macro Language

Each of the different methods of extending Hawk™ has its strengths. The following descriptions will help you determine which is best for you.

DLL Extensions

You can write extensions for Hawk™ using any compiler that can produce DLLs. This capability is referred to as DLL Extensibility.

This method of extending Hawk™ has the following benefits:

- Familiar compiling, and debugging tools
- Increased speed
- Access to functions in other DLLs and libraries.

Most sophisticated extensions are written using DLL Extensibility.

DLL extensibility, however, is not as convenient as other methods of extending Hawk™ in some situations. For example, it is not as convenient for simple jobs, or single use programs. DLL extensibility can also provide little protection against user error, which could cause a system crash. Under these circumstances, one of the supplied macro languages may be a better choice for extending Hawk™. Each language has strengths and weaknesses, which are discussed in the following sections.

AppBasic

AppBasic is the Hawk™ macro language that is similar to Microsoft's Visual Basic for Applications. It has its own editor, debugger, and basic functions. You'll find it on a tab of the output window.

Using this macro language, you can access functions in the Microsoft Windows API, all of the Hawk™ API functions, except those designated as non-interactive, and functions in most any external DLL.

For those who are familiar with basic in any of its various forms, this is an attractive option for both simple and complex macros.

API (C-like) Macros

Hawk™ API Macros is the simplest of the available macro languages. Using C-like structure and syntax, you can quickly create functions suitable for assigning to keys and other simple uses. When writing an API Macro, you can use any function that is available from the API command line.

You can write API Macros in the dialog found on the Tools menu, or you can write them in a standard edit Window, then copy them into the API Macros dialog.

Perl

Perl is, along with JavaScript, perhaps the most popular language for writing extensions to web pages. The syntax is similar to C or AWK, but it has many features for the Internet built-in.

This language is least suitable for writing simple functions for assignment to keys, because of the time it takes for the interpreter to be loaded initially. It does, however, allow Perl programmers to program in a familiar language, using familiar extensions and libraries. The Hawk™ version of Perl is based on the Gnu released Perl interpreter.

2 AppBasic

This section explains how to use the AppBasic Macro Language in Hawk™ to create your own macro functions. It includes the following sections:

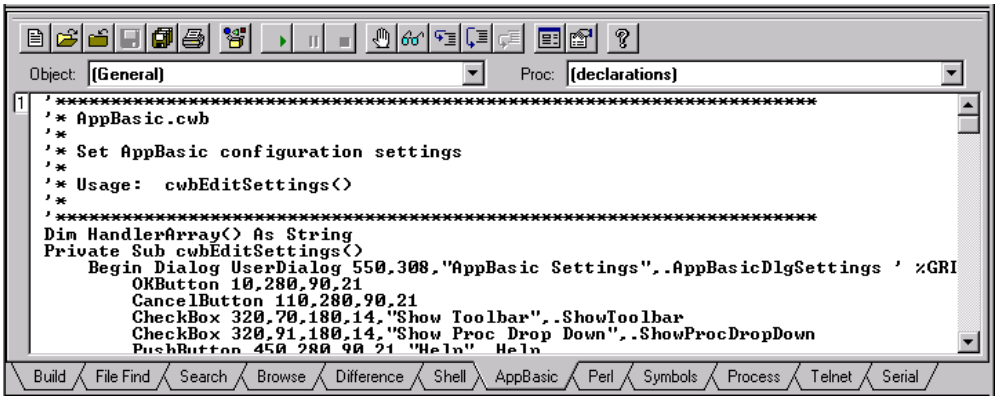
Overview

The AppBasic Macro Language is an interpreted language that is similar in definition and structure to Microsoft's Visual Basic. Through it, you have access to the Windows API, Hawk™ API, and functions in independent DLLs.

Example files are supplied along with Hawk™ to aid you in getting started. You will find them in the Hawk™ MACROS subdirectory. When you press on the File Open button on the AppBasic toolbar for the first time, you will see a list of these sample AppBasic Macros. Several of these macros are covered in this chapter, as well.

The advantage of an interpreted language is that no compiling is required. You can write your macros and use them immediately. By using the AppBasic window, a tab on the Hawk™ Output window, you can add break points and debug your macro, as well.

Figure 2-1. Output Window with AppBasic Tab Selected



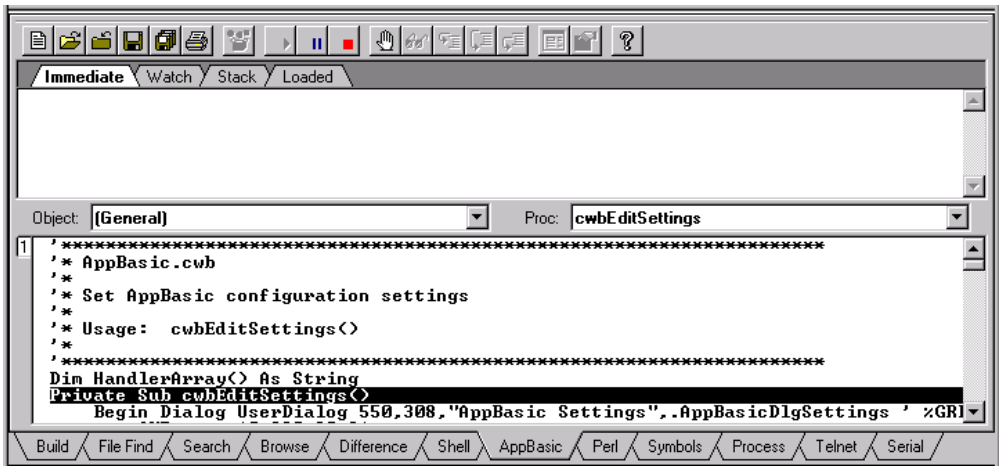
Getting Started

To enable the AppBasic Macro Language, select the Libraries option from the Tools menu. The Libraries dialog displays a list of optional packages or modules you can use with Hawk™. Select the check box for "AppBasic Macro Language". There is also a "Basic" module listed that provides template expansion and language coloring support for the Basic programming language in general. We suggest you enable this option as well.

AppBasic Environment

After loading the AppBasic package, you should be able to see an AppBasic tab on the Output window when it is visible. You can make the Output window visible by selecting it from the Window menu.

Figure 2-2. AppBasic Environment



Two Editors

You can select one of two editors to use to edit your AppBasic source code. You can either use Hawk™ or the Rich Text Editor as your editor.

If you use Hawk™, you get the benefits of standard keystrokes and advanced features. If you use the Rich Text Editor, you get a very close emulation of the Visual Basic editor, including automatic case correction.

You can select which editor appears in the AppBasic window by enabling or disabling the Rich Text Editor. You will find this setting on the View submenu, when you select `AppBasic` from the Tools menu.

Special Keybindings

The following keybindings are in effect in the Rich Text Editor and cannot be changed. When using the Hawk™ Editor in AppBasic, these keybindings are only valid when you have a `.cwb` file current. Otherwise, they revert to your original keybinding for your default keymap.

Table 2-1. AppBasic Keybinding

Keystroke	Operation
`Ctrl-A`	View Macro
`Ctrl-E`	View Immediate Tab Window
`Ctrl-W`	View Watch Tab Window
`Ctrl-T`	View Stack Tab Window

Table 2-1. AppBasic Keybinding (Continued)

Keystroke	Operation
Ctrl-L	View Loaded Tab Window
F5	Debug Run
Shift-F5	Debug Stop
Esc	Debug Pause
F8	Debug Step Into
Shift-F8	Debug Step Over
Ctrl-F8	Debug Step Out
F7	Debug Step To
Ctrl-Shift-F9	Debug Clear All Breakpoints
Ctrl-F9	Debug Add Watch
Shift-F9	Debug Quick Watch
F9	Debug Toggle Breakpoint
Ctrl-N	File New
Ctrl-O	File Open
Ctrl--	File Close
Ctrl-S	File Save
Ctrl-P	File Print

Two Toolbars

AppBasic has two nearly identical toolbars that you can use. One toolbar, depicted in Figure 2-3., has a fixed location at the top of the AppBasic Window. The other toolbar is a Dockable toolbar. You can position it anywhere on your screen, or dock it against any edge of the Hawk™ Client Area. We refer to this as the AppBasic Dockable toolbar, to avoid confusion. You can enable this toolbar via the Toolbars dialog on the Tools menu.

Figure 2-3. AppBasic Window Toolbar

Pop-up Menu

When you right-click in the AppBasic window, a menu pops up that has a number of additional options. If you don't find what you are looking for on the Toolbar or the Tools menu, check this menu.

Online Help

Online help is available for the AppBasic window, the AppBasic Language, and its built-in functions. For help on using the AppBasic window, or language syntax, press the Help button on the AppBasic window toolbar. For help on functions or subroutines, press the [F1] key anytime the cursor is in the AppBasic window. Hawk™ will attempt to bring up help for the word at the cursor. If there is no word at the cursor, you can browse a list of available functions and subroutines.

UserDialog Editor

AppBasic comes with its own dialog editor. A UserDialog is a dialog defined in a macro program. It is described within a `Begin Dialog...End Dialog` block. To create or edit a UserDialog graphically, place the cursor in a UserDialog block and press the `Edit UserDialog` button.

Object Browser

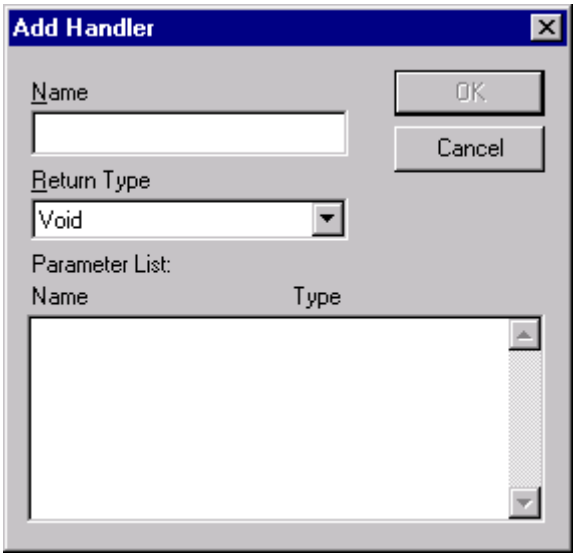
The Object Browser shows information about all the available special data types, particularly for OLE Automation. Select the label that you wish to look up, and click the `Browse Object` button.

If no label is selected, or the label is not found in the known libraries, the edit box at the top of the dialog will be empty. You may still browse the existing data types and methods.

Creating a Macro

Begin by selecting the AppBasic tab on the Output window. The next step is to right-click in the window and select `Properties`. In this dialog, you can provide the names of the AppBasic functions you want to call from `LibFunctionExec`, the Hawk™ API Command, or bind to a Hawk™ button.

Figure 2-4. Add Handler Window



The functions are referred to as Handlers by AppBasic. A Handler exports the function for use by defining its return value, along with parameters and types. For those who are familiar with writing DLL extensions for Hawk™, Handlers serve the same purpose as calling `LibExport`. Once made available in this way, Handlers can be assigned to keys, run from menus and the like.

You can define and create Functions or Subs in the Module Properties dialog before or after creating a Handler. The handler must be added at some point, however, for the function to be called from Hawk™.

When you have finished adding your functions, select [`OK`]. Then, on the Modules Properties dialog, select [`Add`].

Creating a Handler

1. In the Add Handlers dialog, name your function and declare its type..

Functions and Subroutines must start with a letter and can then be followed by an underscore or letter.

2. Type in any parameters you wish. .

Parameters cannot be added later; to do this you must delete the function and add it again.

3. For example enter "`DoThis`" as the name of the subroutine, use a void return type and empty parameter list. Press [`OK`] to return to the Module Properties dialog, then press [`OK`] again.

When you are back in the AppBasic tab, use the Proc list to select `DoThis`. The `DoThis` subroutine will be inserted as follows:

```
Private Sub DoThis()
End Sub
```

4. Now add the following line inside the "`DoThis`" Sub.

   ```
   MsgNotify "This is a test"
   ```

   `MsgNotify` is a Hawk™ API function that notifies the user with a message.

   The `DoThis` subroutine will now look as follows:

   ```
   Private Sub DoThis()
      MsgNotify "This is a test"
   End Sub
   ```

5. You should now save the macro to a file. By convention, the `.cwb` extension is used for AppBasic macros.
6. Now Run the Macro. You may do this in any one of the following ways:

   - Press the `Start/Resume` button on the AppBasic Window toolbar . 
   - Press the `Run Current Macro` button on the AppBasic Dockable toolbar. 
   - Click the right-mouse button when over the AppBasic Output window and select `Run`.

7. After loading using the AppBasic Load Macros dialog, you can go to the Hawk™ menu Tools -> API Command and type

   `DoThis`

8. Click `OK`.

When you run the `DoThis` subroutine in any of the ways above, a message box appears with the words `This is a test`.

Object and Proc Drop Down Lists

The object list shows all the objects for the current module. The object named "(general)" groups together all of the procedures that are not part of any specific object.

The proc list shows all the procedures for the current object. Selecting a procedure that is not bold inserts the proper procedure definition for that procedure.

Figure 2-5. The Object and Proc Drop-downs Located Below the Toolbar.

Handlers you added in the Properties dialog will appear in the Object and Proc list. As an example: You create a function named `Srch_Backwards()`, then the Object list

contains "`Srch`" and the Proc list contains "`Backwards`". The underscore divides the function name between the Object and Proc lists.

Private Sub Main

Any initialization can be done in a special subroutine named `Private Sub Main()`.

For example:

```
Private Sub Main()
   Dim firstTime As Boolean
   Dim i As Integer
    firstTime = True
   If (firstTime = True) Then
        ' no LibExports() needed here, use Modules Property Dialog
And add
        ' handler as a replacement.
   firstTime = False
   Set SaveEvent = EventRegister(EVENT_SAVE_BUFFER, EVENT_NORMAL,
"Buffer_Saved")
   End If
End Sub
```

Tips on Creating Macros

- You may only edit macros that are not currently loaded for execution
- To unload a macro, right-click on the AppBasic tab of the Output window to bring up the Show Loaded Modules dialog
- Removing the check from the check box in front of the filename will unload the macro. It can also be unloaded by calling `cwbUnloadFile(filename)` through the API Command prompt.
- When your cursor leaves a line of source code, it is automatically processed. You may note that capitalization has consequently changed, if you are using the Rich Text Editor.
- A dot in the left margin of the line indicates a break point. Break points may be toggled on/off, using the button on the AppBasic toolbar.

Creating a Modal User Dialog

To create a modal user dialog follow these steps:

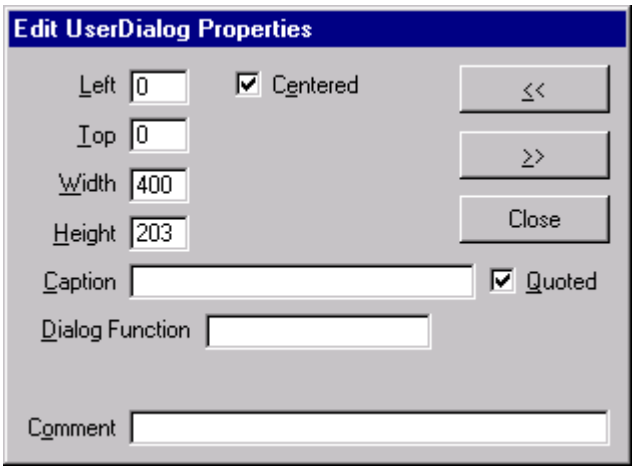
1. Place the cursor in the subroutine or function that you want to place the dialog code in.
2. Then either:
   - Select the AppBasic toolbar button Edit UserDialog.

   Or

   - Select the Hawk™ AppBasic toolbar button Insert/Edit User Dialog.

3. Create the dialog in the UserDialog Editor. Add an OK and/or a Cancel button to the dialog.

Figure 2-6. Edit UserDialog Window



A Cancel button is needed to activate the "X" system menu in the top right hand corner of the dialog.

4. Name your dialog in the Dialog Function box. To get to this dialog you can:
   - Click the right-mouse button on the dialog that you are creating

   Or

   - Select the Edit Item Properties button

   Or

   - Double-click on the titlebar on the dialog that you are creating.

   This is a very important step in the process. The question about creating the skeleton will not show up if you have not filled in the Dialog Function window.
5. Save and Exit the UserDialog Editor, by clicking on the Save and Exit button.
6. Answer `YES` to Create the skeleton dialog function?.

   Your code should now look similar to the following:

```
Private Sub DoThis()
   MsgNotify "This is a test"
   Begin Dialog UserDialog 400,91,"DoThis Test
   dialog",.DoThisTest
      Text 30,28,330,28,"This is a test dialog.",.Text1
      OKButton 40,63,90,21
      CancelButton 160,63,90,21
   End Dialog
   Dim dlg As UserDialog
   Dialog dlg
```

```
End Sub

Rem See DialogFunc help topic for more information.
Private Function DoThisTest(DlgItem$, Action%, SuppValue%) As
Boolean
   Select Case Action%
   Case 1 ' Dialog box initialization
   Case 2 ' Value changing or button pressed
   Rem DoThisTest = True ' Prevent button press from closing
   dialog
   Case 3 ' TextBox or ComboBox text changed
   Case 4 ' Focus changed
   Case 5 ' Idle
   Rem DoThisTest = True ' Continue getting idle actions
   End Select
End Function
```

7. Change the following line of code from:

   ```
   Dialog dlg
   ```

   To:

   ```
   bButtonPushed = Dialog(dlg)
   ```

See the topic Dialog Instruction/Function in the AppBasic online help for more information on this topic and the return values. You can access AppBasic Help by pressing the ? button on the AppBasic window toolbar, or by pressing [F1] anytime the cursor is in the AppBasic window...

Modeless Dialogs are not available in AppBasic.

The final sample is listed below:

```
Private Sub DoThis()
   MsgNotify "This is a test"
   Begin Dialog UserDialog 400,91,"DoThis Test dialog",.DoThisTest
      Text 30,28,330,28,"This is a test dialog.",.Text1
      OKButton 40,63,90,21
      CancelButton 160,63,90,21
   End Dialog
   Dim dlg As UserDialog
   bButtonPushed = Dialog(dlg)

End Sub

Rem See DialogFunc help topic for more information.
Private Function DoThisTest(DlgItem$, Action%, SuppValue%) As
Boolean
```

```
    Select Case Action%
    Case 1 ' Dialog box initialization
    Case 2 ' Value changing or button pressed
    Rem DoThisTest = True ' Prevent button press from closing dialog
    Case 3 ' TextBox or ComboBox text changed
    Case 4 ' Focus changed
    Case 5 ' Idle
    Rem DoThisTest = True ' Continue getting idle actions
    End Select
End Function
```

Running The Macro

Run the macro to see if you have any syntax errors. When you have no syntax errors, go to the Tools -> API Command and execute `DoThis`. You will see a message box and then a dialog box.

The code within this subroutine will be run when the module is put into Run mode. Once a module is debugged, it can be loaded without being in AppBasic. The Hawk™ API command `cwbLoadFile (FileName)` will load the module and put it in run mode without being shown in AppBasic. You can bind the command `cwbLoadFile (FileName)` to a key, or add it to the [Editor] section of your `MWHAWK.INI` configuration file, to load it during start up.

Creating an EventHandler in AppBasic

Events provide a method of interrupting program flow to allow a function or functions to have a timely effect. A number of events have been built into AppBasic using event handlers in order to add flexibility. One reason for this flexibility is that the number and names of the functions the event executes need not be known to the function that triggers the event.

The functions that are executed when a specified event occurs are called Event Handlers. Since most events originate with the Hawk™ API functions, writing your own event handlers gives you access to the Hawk™ core. You can change the way critical functions work without rewriting Hawk™.

Additional information on events can be found in the Hawk™ API (Functions) Help file under the section “Programming” and the topics “Events” and “Using Events”

You can use events by following the three steps listed below:

1. Select the event that represents the action in which you wish to intervene.

   An event handler list can be found in the Hawk™ API Help file under the topic “Using Events”. The sample event handler program ties itself to the event that occurs when a character is entered into a Hawk™ buffer.

2. Write an appropriate event handler function for that event.

   `EventHandler` definition must be Global or the event handler will become unregistered.

3. Register the function for execution at the event with EventRegister.

```
Set X=EventRegister(EVENT_CHAR_INSERTED, EVENT_NORMAL,
  "EventTestHandler").
```

Deregistration of the event is handled automatically when the macro file is unloaded in Hawk™. To stop an event handler before terminating the program (module). Set returned `EventHandler Object` to `Nothing` to remove it manually.

Sample EHTEST.CWB

```
' ****************************************************************
' * EhTest.cwb
' *
' * Sample Event handler EVENT_CHAR_INSERTED, Message box pops up
' * and displays the character that was pressed.
' *
' * Usage:
' *     Test()
' ****************************************************************

' Note!!!! EventHandler must be Global or event handler will become
' unregistered.

Dim X As EventHandler

' This function needs to be executed for the handler to take affect.
' You can put the EventRegister in a main() subroutine if you want it
' to be available everytime the macro is executed.

Private Sub Main()
   Set
X=EventRegister(EVENT_CHAR_INSERTED,EVENT_NORMAL,"EventTestHandler")
End Sub

Private Function EventTestHandler(ID As Long, Datap As Long) As
Integer
   MsgBox StringFromPointer(Datap)
   EventTestHandler = 0
End Function
```

Debugging Your AppBasic Macro

When you are ready to begin debugging an AppBasic macro, the following steps will help you get started:

1. Open the macro file for modification, if you have not already done so (Load it into the AppBasic Window).

2. Set a Break point, perhaps at the first line of the Sub or Function you are debugging, by pressing [F9] or clicking the Toggle Breakpoint button. (A dot should appear in the margin to the left of the line.)
3. Enter Run mode by clicking on the Run, Step Over or Step Into button. The Immediate, Watch, Stack and Loaded tabs will then appear above the Edit window. You can also press [F8] to enter this mode if you are using the RTF editor instead of the default Hawk™ edit window. (Tools Menu -> AppBasic Macros -> View -> Rich Text Editor)
4. Call the Sub or Function so that you reach the Breakpoint you have set. You can do this by selecting API Command from the Tools menu and entering the name of the Sub or Function.

Break Points

Toggle a break point on the current line.

When you are debugging the macro and have hit a break point then notifications will be disabled in the other tabs in the Output window. For example if you have hit a break point in the macro and then you do a multiple search, the double clicking in the Search Output window will be disabled until you have finished running the macro or stopped it.

Evaluate Expression and Add Watch

You may evaluate an expression, assign a value to a variable, or call a subroutine by typing commands in the Immediate window when AppBasic is running a macro.

Table 2-2. Evaluate Expression Commands

Command	Results (when you press [Enter])
`?<expr>`	Shows the value of "`expr`"
`<var> = <expr>`	Changes the value of "`var`"
`Set <var> = <expr>`	Changes the reference of "`var`"
`<subname> <args>`	Calls a subroutine or built-in instruction
`Trace`	Toggles trace mode. Trace mode prints each statement in the immediate window when a macro/module is running.

The Watch window displays the variables, functions and expressions that are calculated. Each time execution pauses the value of each line in the Watch window is updated.

In addition, the following actions are available to you:

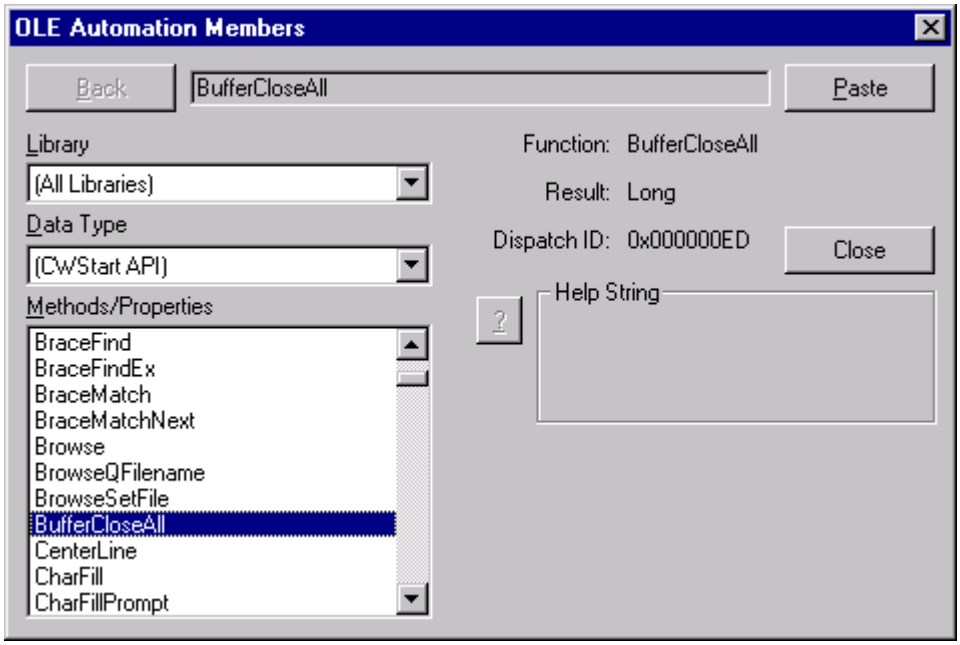
- The expression to the left of the "->" may be edited.

- Pressing `ENTER` updates all of the values displayed, to reflect any changes you have made.
- Pressing `CTRL+Y` deletes the line.

Object Browser

The Object Browser shows information about all the special data types that are available.

Figure 2-7. Object Browser



You can get to the Object Browser by pressing any of the following:

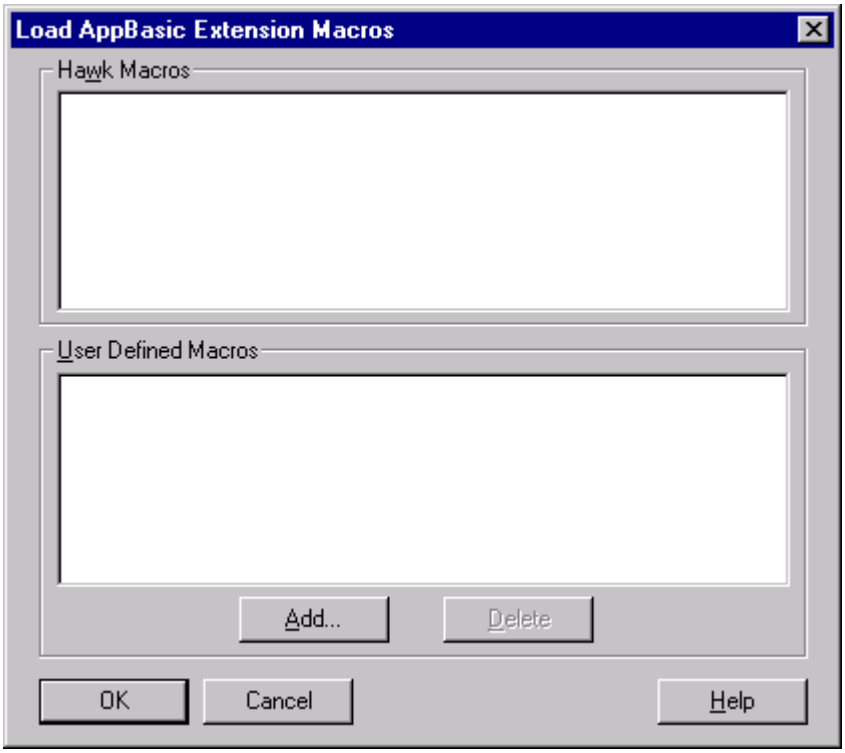
- The `Browse Object` button on the AppBasic window toolbar.
- The `Display Object Browser` button on the AppBasic Dockable toolbar.
- The right-mouse button, when over the AppBasic Output window. Select `Object Browser` from the popup menu.

Load Macros Dialog

To display the Load Macros window, select `Tools -> AppBasic Macros -> Load Macros` from the main Hawk™ menu.

When you choose the `Load Macros` menu item, you are presented with the following dialog.

Figure 2-8. Load Macros Dialog



This dialog loads the sample macros or user defined macros into memory.

AppBasic Samples

This section contains list of the `AppBasic` modules supplied with Hawk™ follows, along with use notes. Most of these samples are not intended to perform a useful task, but rather to show how features of Hawk™ are accessed through `AppBasic`.

Baskeywd.cwb

The function `AddBasKeywords` adds the keyword "`Integer`" to BASIC language ChromaCoding (syntax coloring).

Usage:

```
AddBasKeywords()
```

Colsum.cwb

`SumSelectedText` adds the selected column of numbers and outputs the total on the status line.

Usage:

```
SumSelectedText(1)
```

Diffdir.cwb

`DiffDirectory` differences the file in the reference directory against the files in the target directory. This uses a Modal dialog to display the files being differenced.

Usage:

```
DiffDirectory("c:\testdir1","c:\testdir2")
```

Dispname.cwb

`cwbDisplayFileName` displays the name of the current file on the status line. A "`*`" follows the filename when the file contains unsaved edits.

Usage:

```
cwbDisplayFileName()
```

EditNext.cwb

`cwbEditNextBuffer` makes the next buffer in the list of buffers current.

Usage:

```
cwbEditNextBuffer()
```

EditPrev.cwb

Edit the previous buffer in the list of buffers.

Usage:

```
cwbEditPrevBuffer()
```

EhTest.cwb

The function `Test` is a sample Event handler for the `EVENT_CHAR_INSERTED` event. It causes a message box to pop up and display the character that was pressed.

Usage:

```
Main    ()
```

GotoLine.cwb

`cwbGotoLine` prompts on the status bar for the number of a line to go to.

Usage:

```
cwbGotoLine ()
```

HashTest.cwb

The function `Test` creates a hash table.

Usage:

```
Main()
```

SaveHdl.cwb

When loaded, this module adds an `EVENT_SAVE_BUFFER` event handler, which pops up a dialog whenever a buffer is saved. This program used the main() subroutine to make the handler available when this program is running. No need to call a function to start it up. The Main() subroutine is the first function to be executed when any macro language file is loaded.

SnapVirt.cwb

Snap up to first character on end of line skipping virtual space.

Usage:

```
cwbSnapVirtual()
```

SrchRepl.cwb

This module uses `SearchTranslate()` to execute multiple search and replaces on a buffer, to test this macro use the "From:" text below and put it in a buffer for the replacement.

Translate the following text,

From:

```
West Siberian plains.<252>The European part Is covered
<SUBHEAD>Soviet Period</SUBHEAD>
/par /Tab Union of Soviet Socialist Republics<252><252><252>
```

To:

```
West Siberian plains.
/par /Tab The European part Is covered <SUBHEAD>Soviet
Period</SUBHEAD>
/par /Tab Union of Soviet Socialist Republics
```

Usage:

```
Search_Replace_Buffer()
```

Stripwt.cwb

Strips trailing white space (" " or \t) from the end of a range of lines in a buffer.

Usage:

```
cwbStripTrailingWhite(1, 20, 5)
```

Test.cwb

This module contains several functions to test the type of a return value. Use this on commands entered at the Tools -> API Command prompt, proceeded by a `?`.

For Example: the command `?TestBool` will give you:

```
Return:-1 (hex ffffffff), type:int
```

Usage: in Tools -> API Command box

```
?TestBool  ()
?TestInt   ()
?TestLong  ()
?TestString()
```

Wordcnt.cwb

`CountPhrase` counts the number of occurrences of a specified string that are found in the current document. A dialog prints out the results (Uses `Debug.Print`).

Usage:

```
CountPhrase ("the")
```

ZoomWin.cwb

`cwbZoomWindow` toggles the window between Maximized and Restored. This module uses `SendMessage()` windows call.

Usage:

```
cwbZoomWindow()
```

AppBasic-related API Commands

The following useful API commands are available from the Hawk™ API Command prompt (Tools -> API Command):

cwbLoadFile(<moduleName>)

Once an AppBasic module contains no syntax errors, it can be loaded for execution. It can then be run without the use of the AppBasic window.

cwbLoadFile "FileName"

Loads the module and makes its exported functions (handlers) available for running. You can also bind keys and buttons to the handlers in the loaded Macro file.

cwbUnloadFile (<moduleName>)

An AppBasic module cannot be modified while it is loaded for execution. If you have loaded the file using `cwbLoadFile()` from your `MWHAWK.INI` file or from a button and you want to now edit the file in AppBasic, you will need to unload the file using `cwbUnloadFile()`.

AppBasic Window Configuration

Most users will find the standard configuration of the AppBasic window satisfactory. If desired, however, you can turn off the AppBasic window toolbar and the AppBasic Object and Proc drop-down lists that appear at the top of the AppBasic window.

To change the appearance of the AppBasic window, go to the `Tools -> AppBasic Macros -> View Menu`. Here you can enable or disable the AppBasic window toolbar, the AppBasic Object and Proc drop-down list, or change from a Hawk™ Edit window to the WinWrap Rich Text Editor.

This configuration can also be set by right-clicking over the AppBasic window, and then selecting `View` from the pop-up menu.

If you wish to change the configuration of the window programmatically, the following Hawk™ API commands will assist you. These commands may be issued via the API Command dialog on the Tools menu. If you find them useful, you may make your settings more permanent by modifying similar commands in the `MWHAWK.INI` configuration file.

cwbShowToolbar(int nShow)

This function will Hide/Show the standard AppBasic toolbar.

For example:

```
cwbShowToolbar(0)
cwbShowToolbar(1)
```

The first example will cause the toolbar to be hidden. The second example will show the toolbar. Any positive integer value for the parameter will display the toolbar.

cwbShowProcDisplay(int nShow)

This function will Hide/Show the Object and Proc drop-down lists.

For example:

```
cwbShowProcDisplay(0)
cwbShowProcDisplay(1)
```

The first example will cause the Object and Proc drop-down lists to be hidden. The second example displays them. Any positive integer value for the parameter will display the Object and Proc drop-down lists.

cwbCreateCWWindow(int bCWWindow)

This function is used to change the Editor in AppBasic. There are two choices: one is to make it a Hawk™ Editor and the other is a Rich Text Editor.

For example to change to a Hawk™ Editor call:

`cwbCreateCWWindow(1)` or `cwbCreateCWWindow( TRUE )`

To change to Rich Text Editor call:

`cwbCreateCWWindow(0)` or `cwbCreateCWWindow( FALSE )`

Example Configuration File Settings

The AppBasic window's configuration is stored in the `MWHAWK.INI` file, in the `[AppBasic Setup]` section as shown below:

```
[AppBasic Setup]
cwbShowProcDisplay=2
cwbShowToolbar=1
```

Exported Functions in CWBASIC.DLL

cwbAddHandler(LPSTR lpszHandlerDef, LPSTR lpszModule)

Add a callable Sub/Function to AppBasic module—equivalent to `LibExport`

`lpdzHandlerDef`—Sub/Function Prototype. For example
`Private Sub CallMe(FName As String)`

`lpszmodule` - `cwb` file containing this Sub/Function (Full Path) for example: `c:\cw32\colsum.cwb.`

cwbToggleBreakPoint(void)

Toggle a break point in currently edited module at the current line.

cwbEditFile(LPSTR lpszFile)

Load a `cwb` file into the AppBasic editor.

cwbLoadFile(LPSTR lpszFile)

Load And Run AppBasic file.

cwbUnloadFile (LPSTR lpszFile)

Unload File from Engine (Stop Run).

cwbExecuteCommand(long cmd)

Execute an AppBasic Operational Command. Commands are shown in Table 2-3.:

Table 2-3. AppBasic Operational Commands

Command	Command	Command
`cmdFileNew = 0`	`CmdFileOpen = 1`	`cmdFileSave = 2`
`cmdFileSaveAs = 3`	`CmdFilePrint = 4`	`cmdFilePrintSetup = 5`
`cmdMacroRun = 6`	`CmdMacroPause = 7`	`cmdMacroEnd = 8`
`cmdDebugStepInto = 9`	`CmdDebugStepOver = 10`	`cmdDebugStepTo = 11`
`cmdDebugBreak = 12`	`CmdDebugQuickWatch = 13`	`cmdDebugAddWatch = 14`

Table 2-3. AppBasic Operational Commands (Continued)

Command	Command	Command
cmdDebugBrowse = 15	CmdDebugSetNext = 16	cmdDebugShowNext = 17
cmdHelpApp = 18	CmdHelpLanguage = 19	cmdHelpTopic = 20
cmdHelpAbout = 21	CmdEditUndo = 22	cmdEditCut = 23
cmdEditCopy = 24	CmdEditPaste = 25	cmdEditFind = 26
cmdEditReplace = 27	CmdEditAgain = 28	cmdEditFont = 29
cmdEditDelete = 30	CmdEditSelectAll = 31	cmdEditUserDialog = 32
cmdFileClose = 33	CmdFileSaveAll = 34	cmdDebugStepOut = 35
cmdSheetOpenUses = 36	CmdSheetCloseAll = 37	cmdSheet1 = 38
cmdSheet2 = 39	CmdSheet3 = 40	cmdSheet4 = 41
cmdSheet5 = 42	CmdSheet6 = 43	cmdSheet7 = 44
cmdSheet8 = 45	CmdSheet9 = 46	cmdFileNewCodeModule = 47
CmdFileNewObjectModule = 48	CmdFileNewClassModule = 49	cmdEditProperties = 50

cwbShowToolbar(int nShow)

Hide/Show Standard AppBasic Toolbar.

cwbShowProcDisplay(int nShow)

Hide/Show Object and Proc Dropdown Lists.

3 API (C-like) Macros

This chapter describes the Hawk™ API macros and how they are used interactively to write extensions to Hawk™. It includes the following sections:

Overview

Hawk™ API Commands are function calls that can be made interactively. This means that they may be assigned to keys, menu items, buttons, and issued from the API command line. Hawk™ API Macros build upon this capability to provide a quick and simple way to write extensions for Hawk™. The capabilities of this language are simplistic, and compare most closely to "Small C".

Further capabilities may be added in a future release. If you require more capabilities, please consider one of the other macro languages, or writing a DLL extension.

If you have previously used, or read about using API Commands interactively, a few differences should be noted:

- You can nest API Commands as parameters to other API Commands in macros,
- All functions must be followed by opening and closing parentheses around the parameter list, if any.
- Parameters must be separated by commas.

API Macros Defined

API Macros are one or more sequential Hawk™ API function calls that are given a name. They are stored in a file, `MWHAWK.MAC`, and may be recalled by name for execution within Hawk™.

Getting Started with API Macros

The API Command Macro dialog on the Tools menu lets you create, edit and test API Macros.

Creating a Macro

The first step in creating a macro is to give it a name in the Name edit box. Next, type in the API commands you wish to execute into the Edit box. Next, press the `Save` button. Pressing the `Run` button will send away the dialog. Any unsaved edits are automatically saved.

Editing a Macro

To edit a macro previously created, select its name from the Name list box. The text of the macro will appear in the Edit box and you can proceed to make your changes. Pressing either `Run` or `Save` will save your changes to disk.

You can also create API Macros in a standard Hawk™ edit window and paste them into this dialog. This gives you access to the Hawk™ advanced editing capabilities and ChromaCoding.

Language Definition

Comments

C++ style comments may be used in API Macros. When two forward slashes, '//', appear on a line, the remainder of the line is ignored.

Variables

Variables are not supported in Hawk™ API Macros.

Data Types

Two types are supported: strings (text) and long integers (numbers). Strings must start and end with the quote character (").

Using a function with the proper data return type is essential, when nesting function calls and for macro return values.

Expressions

Expressions are made up of numbers, operators, function or macro calls and parentheses. Numeric and logical expressions are allowed. Expressions can be used as function call parameters.

Empty (zero length) strings evaluate to 0 whereas non-empty strings evaluate to one. Logical expressions evaluate to a one or zero, `TRUE` or `FALSE`, respectively.

Number formats

Decimal, hexadecimal and octal number formats are supported. Hexadecimal numbers must start with '`0x`'. Octal numbers must start with '`0`'. All other numbers are considered decimal.

Operators

The numerical operators available are shown in the following table:

Table 3-1. Numerical Operators

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (remainder)

Table 3-1. Numerical Operators

Operator	Description
<<	Shift bits left
>>	Shift bits right
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
~	Bitwise Complement

Logical operators are shown in Table 3-2.:

Table 3-2. Logical Operators

Operator	Description
==	Equivalence
!=	Non-equivalence
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal
&&	AND
||	OR
!	Negation

Parentheses in Expressions

Left and right parentheses,' (' and ') ', may be used to group expression parts to set evaluation preference.

Function calls

Function calls, including those to Hawk™ API functions and macros take the form:

```
<macro_name> (<parameter1>,<parameter2>,...<parameterN>)
```

Spaces and tabs are allowed after the macro name and between parameters. Parameters must be separated by commas.

You may supply fewer parameters than a function is defined to use. The remaining parameters will be supplied for you as zeros or NULLs. Extra parameters are ignored.

Exiting and Macro return values

Use the return statement to exit a macro before the last line and optionally return a number (expression) or string (function call).

End the return statement with a semicolon.

Statements and Statement Blocks

Statements

Statements are simply a function or macro call on a single line, ended by a semicolon.

Statement blocks

Statement blocks are a series of statements. They are normally used in conjunction with control structures. Use curly braces '{' and '}', to begin and end statement blocks, respectively.

Braces do not need to be on a separate line.

Control Structures

The available control structures are conditional (`IF`) and iterative (`WHILE`). Both conditional and iterative statements can be used anywhere statements might be used.

Conditional statements

if statement

```
if (<expression>)
<statement>;
```

if-else statement

```
if (<expression>)
<statement>;
else
<statement>;
```

The `if (<expression>)` section must be all on one line.

Testing for several cases can be done by using the `if-else` statement in place of the statement following the `else`. The following does this and also uses a statement block:

```
if (<expression one>)
{
   <statement>;
   <statement>;
}
else if (<expression two>)
   <statement>;
else if (<expression three>)
   <statement>;
else
   <statement>;
```

Iterative statements

while statement

```
while (<expression>)
<statement>;
```

String Functions

Since library functions are not available in API Macros, a number of string functions are made available through the Hawk™ API for building and comparing strings.

`LPMSTR StringApnd( LPSTR str1, LPSTR str2 );`

Append `str2` onto `str1`. Neither string is freed.

`LPMSTR StringNApnd( LPSTR str1, LPSTR str2, int len2 );`

Append `len2` bytes of `str2` onto `str1`. Neither string is freed.

`int StringLength( LPSTR str );`

Return the length of str.

`int StringCompare( LPSTR str1, LPSTR str2 );`

Returns `< 0` if `str1` is alphabetically before `str2`. Returns `> 0` if `str1` is alphabetically after `str2`. Returns `0` if `str1` matches `str2`.

`int StringICompare( LPSTR str1, LPSTR str2 );`

Like `StringCompare` but ignores case.

`int StringNCompare( LPSTR str1, LPSTR str2, int len2 );`

Like `StringCompare` but only compares `len2` bytes.

`int StringNICompare( LPSTR str1, LPSTR str2, int len2 );`

Like `StringCompare` but ignores case and only compares `len2` bytes.

`LPMSTR StrAscii( int ch );`

Converts a character into a single character length string.

`LPMSTR StrItoA( long value, int radix );`

Converts a number into a string. Use a radix of `10` for decimal string.

```
LPMSTR StrLtrim( LPSTR string, LPSTR cset );
```

Trims characters in `cset` off the left of string.

```
LPMSTR StrTrim( LPSTR string, LPSTR cset );
```

Trims characters in `cset` off the right of string.

```
LPMSTR StrSubStr( LPSTR string, int start, int end );
```

Return a substring out of string.

```
LPMSTR StrFormatDate( long t, LPSTR fmtStr);
```

Formats a time/date value into a string.

```
LPMSTR TransformFilename( LPSTR filename, LPSTR spec);
```

Documented in the online help.

```
BOOL StrFileMatch( LPSTR fpattern, LPSTR fname, BOOL igcase );
```

Determine if a filename is matched by a pattern.

4 Perl

This chapter provides information about using Perl scrips in Hawk™. It includes the following sections:

Overview of Perl

There are two useful ways of looking at a Perl script. One is to look at Perl as an entity that may be executed in its entirety to perform the task specified by the operations of the script. Another way of looking at the script is as a collection of macros, each subroutine being a macro in its own right. Using Perl with Hawk™, you can make use of Hawk™ custom facilities that reinforce these views as you have need. These will be described below in "Loading and running scripts."

Getting Started with Perl

There are two loadable `.DLL`s necessary to use Perl with Hawk™: the Perl interpreter and the Perl language support DLL. To enable these modules, go to Tools -> Libraries and select the check box for “Perl Extension Language Interpreter”, and also "Perl Language".

If the Output window is not already showing, select it from the Window menu. You should then see a Perl tab on the Output window.

Even if you have Perl already installed on your system, you must check the “Perl Extension Language” check box to enable the Perl tab for the Output window.

The `cwp.dll` is a Perl extension module that provides access to Hawk™ API functions from within Perl scripts. In general, you can use any of the Hawk™ API functions listed in the online help, except for those whose data types are incompatible with Perl. For a list of available Hawk™ API functions, see the contents of the `cwp.pm` file..

If you have not already installed a command line version of Perl, we have included a copy of the ActiveWare 3.10 build of Perl 5.0 for Win32, `PW32I310.EXE` in the `PERLW32` directory on the Hawk™ CD.

Creating and Editing Perl Scripts

Unlike the AppBasic Macro Language, Perl scripts do not require a special edit window. You create and edit source files in standard Hawk™ edit windows. There are some sample Perl script source files in the Hawk™ MACROS subdirectory. Load a few of these and have a look.

Perl scripts are usually packaged in text files bearing a `.PL` extension. A Perl script consists of a 'main' section, and zero or more subroutine definitions. The main section is defined as any code not contained within subroutine definitions.

Perl Window

The Perl tab on the Output window acts as a virtual console for the Perl interpreter. That is, it replaces the stdin, stdout, and stderr devices. You can scroll through the output of Perl Scripts in this window. There is a limit on how many lines of output will be retained in this window. The default is 100 lines. You will find this setting in the Perl Properties dialog, on the Tools menu.

To help distinguish `stdin`, `stdout`, and `stderr` elements of the Output window, `CWPerl` uses three different colors. `stdout` is displayed using the output color. `stdin` is displayed using the color defined for line numbers. `stderr` is displayed using the color designated for comments. You can find the settings for these colors on the Colors tab of the Document Preferences dialog.

Pop-up Menu and Options

When you right-click on the `Perl` tab, or any portion of the Perl window part of the Output window, a menu pops up that has a number of additional options. If you don't find what you are looking for on the toolbar, or the Tools menu, check out this menu.

The Properties item on the pop-up menu brings up a dialog that lets you control how Perl interacts with Hawk™. This is where you specify options you would otherwise provide to Perl on its command line, when invoking it from the shell prompt.

There is a variety of choices from which to select Perl's input source and output destination including the Perl window, the current buffer, the clipboard, the current scrap buffer, and the current selection.

Different combinations of source and destination can produce interesting effects. For example, if `Current Document` is selected for both source and destination, a Perl script will see the entire content of the current document as its `stdin` and the buffer will be replaced with Perl's `stdout`. Similarly, you can accomplish a similar operation on the contents of a selection choosing `Selection` for both source and destination.

Input source and output destination can also be set from within a Perl Script. See the file `upcase.pl` for an example of how this is done.

Loading and Running Scripts

There are two different ways to run a Perl script in the Perl tab of the Output window. You can load the script directly on an as-needed basis, or you can load a Perl macro and run it any time you wish.

Running a Script Directly

From the Hawk™ API Command Line type the command

```
PerlExec <script file> <any parameters>
```

Where `<script file>` is the fully qualified path and filename of the script file, and `<any parameters>` is any parameters required by the script file. This will load the script, execute it, and unload the script automatically.

Loading a Perl macro

Loading a Perl macro is much the same as running a Perl script directly, with the added advantage of being able to invoke the subroutines of the script individually by simply typing them at the command line.

There are three ways to load a macro:

1. Go to the Tools menu and select `Perl Macros`, and then select `Load Macros`. Perl macros may be selected by checking the appropriate box in the top of the dialog in the area called Hawk™ Macros. User defined macros may be added from here by pressing the `Add` button.
2. Another way to load Perl macros is by right-clicking the `Perl` tab in the Output window and selecting Interpreters. This brings up the Hawk™ Perl interpreters dialog. Load the macro by pressing the `Load` key. All subroutines for the loaded macro (including `<main>`) are automatically displayed when you select a macro from the list. You may select an individual subroutine from the list to run.
3. Use the Hawk™ API command

   ```
   PerlLoad <filename>
   ```

Once the macro is loaded, you can invoke it by going to the Hawk™ API Command Line and entering the name of the subroutine. As previously mentioned, you may also run the subroutine by right clicking the `Perl` tab and selecting Interpreters. Select the macro from the list, and when the subroutine list is displayed, select the desired subroutine. Then press the `Run` button. You will be prompted for parameters; you may enter them if necessary, or simply hit the `RETURN` key to run without parameters.

You can load and execute a Perl script from the Hawk™ API Command Line by using the command:

```
PerlExec      filename.pl
```

If you only want to execute a specific subroutine in a script use:

```
PerlExecSub      filename.pl subname
```

In both of these cases, you may include additional command line parameters. All text following the syntactical elements shown in the two commands is broken down into parameters using normal command line parsing rules, with respect to spaces, quotes, escapes, and so forth.

If you just want to execute a simple Perl script that can be typed in one line use:

```
PerlExecStr      'perl-string'
```

Perl for Hawk™ is capable of hosting multiple simultaneous interpreters. The three commands create an interpreter, parse the input, execute the script and then dispose of the interpreter. You may alternately load one or more interpreters by using the command:

```
PerlLoad      filename.pl
```

This command also may be supplied command line arguments although they will only be utilized to the extent that they affect loading the interpreter. This command may be run several times to load different scripts. From time to time, a loaded interpreter may be utilized by the command:

```
PerlRun       filename.pl
```

If this command is issued as shown, the `main` of the script will be executed. Alternately, you can name a subroutine:

```
PerlRun filename.pl subname
```

As with previous commands, parameters may be supplied following either form, however, the first form will need an extra empty string (" ") to represent the `main` function.

Accessing Hawk™ Functions from Perl Scripts

Over eight hundred Hawk™ functions can be directly accessed from within Perl scripts from Hawk™. To do so, you'll need to place the following line near the top of your script:

```
use CWP;
```

This tells Perl to load `CWP.PM` which contains descriptions of functions contained in the compiled Perl extension module `CWP.DLL`. The `CWP.DLL` consists of small C functions that call the Hawk™ functions after first preparing the parameters and then make the return value, if any, available to Perl.

You'll need to use the `CWP::` prefix on Hawk™ functions to reference them, unless you take further steps (described later). This prefix tells Perl that the following function name is present in the `CWP` module. As an example, to find out where the cursor is on in the current document, you would use something like:

```
$line = CWP::BufQCurrentLine();
```

Constants that are held in the Hawk™ lookup table can be accessed using a special function. For example, to see if there is a column selection present you could use:

```
if (CWP::MarkQSelType() == CWP::CWConst("SELECTION_COLUMN")) {
}
```

A special function is also provided to execute any built-in or any function made available through `LibExport`. This function is similar to the Hawk™ API function `LibFunctionExec`). An example follows:

```
CWP::CWExec("ConfigFileRead", "[Editor]", 0);
```

This is equivalent to:

```
CWP::LibFunctionExec("ConfigFileRead [Editor] 0");
```

The primary difference is that in the latter example, the string will be parsed to separate it into parameters to be passed to the function. In the former, the parameters are explicitly segregated.

Importing Names into Perl's Namespace

If you have Hawk™ functions that you access frequently and their names don't conflict with standard names, you can save some typing by telling Perl that you want certain names imported into Perl's namespace. This is done with a modification of the `use` command, like this:

```
use CWP ("LibFunctionExec", "CWConst", "BufQCurrentLine");
```

or, equivalently:

```
use CWP qw(LibFunctionExec CWConst BufQCurrentLine);
```

The latter *quoted word* shorthand obviates the need to type so many quotes. After doing this, the named function may be invoked without the `CWP::` qualifier.

```
$line = BufQCurrentLine();
```

Unloading a Perl Macro

When you no longer need an interpreter it can be deleted with

```
PerlUnload      filename.pl
```

Using Perl's Debug Mode

There is a debugger supplied for Perl scripts. The debugger is itself a Perl script (`perl5db.pl`). When invoking Perl from a prompt, this script is automatically loaded if you use the `-d` flag. In Hawk™, you accomplish the same thing by checking the Debug Mode check box in the Perl Properties dialog (right-click the Perl tab).

When running a script in debug mode, there are three important requirements:

1. You should only use debug mode when invoking scripts with `PerlExec` on the API command line.
2. The script must not already have been loaded by Hawk™ prior to using the PerlExec command (meaning the script cannot be loaded from the Interpreters box or by the Tools -> Perl macros -> Load macros -> pop-up dialog. If it is loaded, unload it before attempting debug mode).
3. Ensure that you have completed executing the script in debug mode before attempting to run the script again.

Accessing Perl functions

You can also execute the subs of loaded Perl interpreters merely by typing their names on the Hawk™ API Command Line. Hawk™ has a mechanism to allow add-ons to respond to indicate their recognition of otherwise unknown (to Hawk™) function names. The AppBasic interpreter and the Perl interpreter alike respond to these queries.

This means that if you have a Perl script loaded that contains a sub called `MyPerlFunc`, you can execute that function simply by typing `MyPerlFunc` at the Hawk™ API Command Line. Perl will check all the loaded Perl scripts to see if at least one of them has the named subroutine.

Avoiding Ambiguity

The same subroutine name may appear in several Perl scripts. Hawk™, therefore, provides a mechanism to specify which Perl script you're referencing. Suppose that both `SCRIPT1.PL` and `SCRIPT2.PL` have a sub `MyPerlFunc`. You don't know which subroutine will be executed if you simply invoke `MyPerlFunc`.

One way to avoid this problem is to right-click the Perl tab and bring up the Interpreters dialog. Select the script you intend to call, and click its subroutine in the subroutine list.

Another way to specify that you wish MyPerlFunc in SCRIPT2.PL to be executed, is to invoke it as follows:

```
script2.pl!MyPerlFunc
```

Note that the extension .pl used in this example is optional. Without this specific invocation, the loading order usually determines which function will be executed if both an AppBasic script and a Perl script contain a given function name. Generally, the first loaded will be the one to execute.

Special API Functions for Perl

In addition to the 800 or more functions available from the Hawk™ API, there are three additional functions available when the Perl Extension Language (CWP) is loaded.

DWORD CWConst(LPSTR cw_expr);

This function provides access to Hawk™ constants (predefined values), and any "constants" you have defined with EvalAddStr, in Perl scripts. For example, you may want to begin a line selection. To do this you would use the constant SELECTION_LINE in a call to MarkBeginSel. A more specific example follows:

```
CWP::MarkBeginSel(CWP::CWConst("SELECTION_LINE"));
```

The argument string to CWConst can take any form that is acceptable to the Hawk™ function EvalExpression.

DWORD CWExec(LPSTR cw_funcname, ...);

This function provides an alternate form of executing a Hawk™ function that has been made available via LibExport(), such as all functions provided by DLLs, both standard and add-on. The advantage to using this form, rather than LibFunctionExec(), is that you needn't compile the arguments that you wish to pass into a string as is required by LibFunctionExec(). You could, for example, issue either of the following commands with the same effect:

```
CWP::LibFunctionExec("BufQOffsetEx", $line . " " . $column);
```

or

```
CWP::CWExec("BufQOffsetEx", $line, $column);
```

int CWPerlIO(int mode);

This function queries or sets the current Perl I/O mode. You may recall from the Perl Properties entry of the Perl pop-up menu that you can set the Perl input source and output destination using the radio buttons. This function serves the same purpose.

The mode argument has two components: output destination and input source. The latter occupies the least significant 8 bits of the mode value, while the former occupies the next most significant 8 bits. The semantics of the components are as follows:

```
// Perl I/O source/destinations
#define PERL_IN_CON0x0000      // input from 'console'
#define PERL_IN_BUF0x0001      // input from current buffer
```

```
#define PERL_IN_SEL0x0002      // input from selection
#define PERL_IN_CLIP0x0003     // input from clipboard
#define PERL_IN_SCRAP0x0004    // input from scrap buffer
#define PERL_OUT_CON0x0000     // output to 'console'
#define PERL_OUT_BUF0x0001     // output to current buffer
#define PERL_OUT_SEL0x0002     // output to selection
#define PERL_OUT_CLIP0x0003    // output to clipboard
#define PERL_OUT_SCRAP0x0004   // output to scrap buffer
```

To determine the current i/o mode, simply call `CWPerlIO` with a `mode` value of `-1`. In all cases, the return value is the prevailing 'mode' immediately prior to the call. This function is useful to make a Perl script that responds differently depending on certain prevailing conditions.

The call to `CWPerlIO` should be made before any output operations are performed, since changing the output destination causes all previous output to be discarded.

Files used by Perl for Hawk™

The files listed in Table 4-1. are found in the Hawk™ home directory after installation:

Table 4-1. Perl Files in Hawk™ Directory

Filename	Purpose
`CWPERL.DLL`	Provides Perl language support for Perl source scripts (`*.PL`).
`CWPERLI.DLL`	Adds the Perl tab in the Output window.
`CWPERLI.MNU`	Menu file that holds the menus for the right mouse click in Perl tab of Output window.

The files shown in Table 4-2. are found in the Hawk™ PerlLib subdirectory after installation:

Table 4-2. Perl Files in PerlLib Directory

Filename	Purpose
`CWP.DLL`	Allows access to Hawk™ API's in the Perl script.
`CWP.PM`	Exporter for Hawk™ APIs.
`CARP.PM`	Error routines.
`CONFIG.PM`	Win32 Perl configuration.

Table 4-2. Perl Files in PerlLib Directory (Continued)

Filename	Purpose
CWP.XS	Source Perl script used to rebuild cwp.dll.
DYNALOADER.PM	Dynamically loads C libraries into Perl code.
EXPORTER.PM	Default import method for modules.
PERL5DB.PL	Perl 5.0 Debugger.
TYPEMAP	Hawk™ variable types (for use with cwp.xs).
XSUBPP.PL	Converts Perl XS code into C code (for use with cwp.xs).
TERM\CAP.PM	Perl termcap interface.
TERM\COMPLETE.PM	Perl word completion module.
TERM\READLINE.PM	Perl interface to various C<readline> packages.

Other Perl Resources

Much useful information can be derived from your nearest CPAN (Comprehensive Perl Archive Network) web site, or some of the Perl books by O'Reilly Press, namely:

Programming Perl

Learning Perl for Win32 Systems.

```
There are many mirror sites for CPAN, one of which is:
ftp://ftp.cdrom.com/pub/perl/CPAN/
```

Other useful URLs include:

`http://www.perl.org` for general Perl information

`http://www.activestate.com` for Perl for Win32 systems

5 Function Definitions

This chapter defines where function definitions are stored in the Hawk™ directory structure.

Location of Functions

There is an opportunity for confusion as to exactly which function is being executed when a given name is typed. It could be a Hawk™ built-in function, a Hawk™ DLL function, an AppBasic macro, a Perl sub, a keystroke macro or an API macro. The following command will help you resolve the ambiguity.

`LibFunctionExistsWhere()` will return a string identifying the location of the function name that is supplied as its argument. Table 5-1. shows a few sample responses.

Table 5-1. LibFunctionExistsWhere Responses

Function	Response
`BufQCurrentLine`	`Built-Ins:BufQCurrentLine`
`DlgPrint`	`cwdialog:DlgPrint`
`MyPerlFunc`	`cwperli:myperl!MyPerlFunc`
`_jav_init`	`_jav_init->cwstart:_java_init`

The last example shows the response where a replacement function exists. The Perl example shows that both the Perl script filename and the sub name are given.

For non-LibExported functions, such as those connected by responding to `EVENT_LIB_EXISTS_FAILED` and `EVENT_LIB_EXEC_FAILED`, the `where` string is supplied by the responder to a new event `EVENT_LIB_EXISTS_WHERE`. The event handler supplies a `where` string (allocated) as the return value or zero if it doesn't claim the function.

Index

F

G

H

I

L

M

N

O

P