



# **Programmer's Guide**

**Version 3.7 Users Guide**

*Revised 5/21/03*

Copyright 1996-2003 by Wavelink Corporation. All rights reserved. This manual may not be reproduced, in whole or in part, without prior written permission from Wavelink Corporation.

Wavelink<sup>®</sup> is a registered trademark of Wavelink Corporation.

Symbol<sup>®</sup>, Spectrum One<sup>®</sup>, and Spectrum24<sup>®</sup> are registered trademarks of Symbol Technologies, Inc.

Microsoft, MS<sup>®</sup>, MS-DOS<sup>®</sup>, Windows<sup>®</sup>, Windows NT<sup>®</sup>, Visual C++<sup>®</sup>, Visual Basic<sup>®</sup>, are either registered trademarks or trademarks of Microsoft Corporation in the USA and other countries.

Winzip, Winzip Self Extractor are either registered trademarks or trademarks of Niko Mak Computing, Inc. in the USA and other countries.

Wavelink Studio Users Manual

Revision 5/21/03

# Table of Contents

<b>Chapter 1: Introduction</b>	<b>1</b>
About this Document .....	1
Assumptions .....	1
Document Conventions .....	2
Additional Information .....	3
About Wavelink Studio COM .....	3
Wavelink Server .....	3
Wavelink Client .....	4
Supported Devices .....	4
Wavelink Development Library .....	5
Widget Objects .....	6
 <b>Chapter 2: Application Frameworks</b>	 <b>9</b>
Referencing the Wavelink COM Development Library .....	9
Including WaveLink Objects in Visual Basic 6.0 Projects .....	9
Including Wavelink Objects in other COM Languages .....	10
 <b>Chapter 3: Error Handling</b>	 <b>11</b>
 <b>Chapter 4: I/O Techniques</b>	 <b>13</b>
Handling Out-of-range Devices .....	13
Optimizing RF Traffic .....	14
Displaying Data on the Device Screen .....	15
Using RFInput .....	16
Invoking RFInput .....	16
Waiting for User Input .....	18
Processing Returned Input .....	19
Input Modes .....	20
Input Timeouts .....	22
Using High Speed Display .....	23
Storing Screens .....	24
Storing Screen Templates .....	25
Automating the Workflow .....	27
Designing the Application .....	27
Adding Bar Code Symbolologies to your Applications .....	29
Using Bar Code Symbolologies .....	31
Adding Tones to Your Applications .....	36
Navigating the Application .....	38
Using Function Keys .....	38
Using Menus .....	40
Using Message Boxes .....	42

<b>Chapter 5: Widgets</b>	<b>45</b>
Using Widgets	45
Handling Events	48
Using Widget-based Dialog Boxes	49
Using Menu-based Widgets	50
Widget Transactions	53
Positioning Widgets	53
Hiding and Disabling Widgets	54
Setting the Focus	55
 <b>Chapter 6: Writing Applications for Multiple UIs</b>	 <b>57</b>
General Techniques	57
Returning the Screen Dimensions	57
Returning the Device Type	58
Object-Oriented Techniques	59
Program Overview	60
Model-View-Controller Paradigm	60
Finite State Machines	62
Program Source Files	63
Initializing the Application & Starting the Finite State Machine	64
Finite State Machine Classes	67
Populating the State List and View List	67
Implementing the Finite State Machine	70
Presentation View Classes	73
State Classes	74
C++ Source Files	75
WaveLinkPM.cpp	75
CWaveLinkPMFSM.cpp	76
CFSMBase.cpp	78
CStateList.cpp	79
CPresentationList.cpp	80
CPresentationBase.cpp	80
CStateBase.cpp	80

# Chapter 1: Introduction

This document presents an introduction to developing applications using the Wavelink Studio COM Development Library.

The Wavelink Studio COM Development Library provides development platforms such as Visual Basic, C++, and others with the ability to communicate with devices on a wireless network. By including the Wavelink Studio COM Development Library in your development platform, you can build highly-efficient wireless applications with a minimum of effort.

## About this Document

We are very interested in improving the Wavelink Development Library documentation and welcome all criticisms and suggestions for improvement. Please direct them to:

Wavelink Corporation, Development Library Documentation  
11332 NE 122nd Way  
Suite 300  
Kirkland, WA 98034-6936  
Phone: 425.823.0111  
Sales: 1.888.697.WAVE (Sales Only)  
Support: 1.888.699.WAVE (Support Only)  
Fax: 425.823.0143  
Email: [documentation@wavelink.com](mailto:documentation@wavelink.com)

## Assumptions

It is assumed that readers of this document are experienced in application design for one or more programming languages and that they have some familiarity with Visual Basic. It is also assumed that readers know how to interact with their data source and understand the RF technology they are using.

---

**NOTE** The sample code is primarily in Visual Basic. The programming techniques, however, apply to programming languages supported by the Wavelink Development Libraries, including C++.

---

## Document Conventions

This document uses the following typographical conventions:

- **Courier New.** Any time you interact with a Wavelink Studio COM option, such as a button, or type specific information into a text box, such as a file pathname, that option appears in the `Courier New` text style. This text style is also used for keyboard commands that you press.

Examples:

Click `Next` to continue.

Press `CTRL+ALT+DELETE`.

- **Bold.** Any time this document refers to an option, such as descriptions of the choices in a dialog box, that option appears in the **Bold** text style.

Examples:

Click `Open` from the **File** menu.

Click `Download` from the **HexFiles** menu.

- **Italics.** Any time this document refers to another section within the manual, that section appears in the *Italic* text style.

Example:

See the *Troubleshooting* section for possible causes of this problem.

- **Angle Brackets.** Any time this document uses a placeholder, representing an actual value such as a directory or class name, that section is enclosed by angle brackets.

Example:

`<classname>.<methodname>(TOKEN)`

- **Square Brackets.** Any time this document references an optional element, that section is enclosed by square brackets.

## Additional Information

This document is not an exhaustive guide to developing wireless applications with Wavelink. For more information about using Wavelink Studio COM, see the following documentation:

- The Wavelink Studio COM Development Library
- The Wavelink Studio COM Server
- Wavelink Studio Client documentation (device-specific)

## About Wavelink Studio COM

The Wavelink Development Library is one component in Wavelink Studio COM. Wavelink Studio COM includes the following primary components:

- *Wavelink Server*
- *Wavelink Client*
- *Wavelink Development Library*

### Wavelink Server

The Wavelink Server is a software application that provides direct access by end users to wireless applications, and total control over application and connection management in a wireless network.

The Wavelink Server creates an application instance for each new client that logs onto the network. When a client logs on, the server and client negotiate a connection using an Auto Discovery process that links the client to specific *Wavelink Port Monitors*.

A Port Monitor is responsible for accepting connections from mobile devices. As the name implies, each Port Monitor is assigned to a specific port on its host system. When a mobile device attempts to connect to a wireless application, the mobile device sends a connection request. This request is routed through a Discovery Manager, which contains a list of all available Port Monitors on a specific network subnet. If the Discovery Manager finds a Port Monitor that matches the connection request, it sends the location of the Port Monitor to the mobile device, allowing the device to connect to the desired application.

## Wavelink Client

The Wavelink Client is a thin-client that resides on and interacts with the mobile device and enables communication with the Wavelink Server. The Wavelink Client installs directly into the Non-Volatile Memory (NVM) of the mobile device.

When the clients boots up, it automatically connects to a server using the Auto-Discovery mechanism. All applications that a user can access automatically appear on a dynamic menu following sign-on.

The Wavelink thin-client software caches frequently-used configuration parameters and menus, intelligently sharing the burden of a wireless application with the application host.

The Wavelink Client also includes the following built-in features:

- **Scanning control.** Smart processing of barcode data at the handheld depends on type of barcode scanned; this saves needless RF communication back and forth between the handheld and host server.
- **Error control.** Keypad and scanner options limit user errors by controlling the acceptable format for input, including barcode types.
- **Tone control.** Customized handheld “tones” direct users through their work, improving worker productivity by eliminating the need to look at the display.

## Supported Devices

WaveLink supports the following device types, based on OS:

- DOS and Embedded
- Palm OS
- Windows CE/Pocket PC

---

**NOTE** For more information about specific devices supported, go to <http://www.wavelink.com/downloads/>.

---



## Wavelink Development Library

The Wavelink Development Library is the application programming interface (API) that allows custom application development for a wireless network.

All Wavelink-based applications incorporate a multi-threaded, multi-instance architecture, making application design more efficient while maximizing run-time performance. The Wavelink Development Library allows you to build customized wireless applications. The Wavelink Development Library permits a wide range of application developers to migrate their applications to wireless networks.

Developing applications using the Wavelink Development Library provides the following benefits:

- **Simplified coding.** The Wavelink API handles all the wireless network technical issues.
- **Built-in hardware support.** Custom applications automatically run on all 802.11 based LAN or GPRS WAN wireless devices.
- **Hardware upgrade support.** Heterogeneous hardware devices and operating systems with various UI's (stylus, key, or touch-based) can be deployed.
- **Device Control.** Wavelink applications allow you to take full control of your mobile devices through the built-in features of the Wavelink Client.

The Wavelink Development Library includes numerous objects tailored for wireless application development, including the following:

### RFBBarcode Object

The RFBBarcode object allows you to create barcode configurations that define valid and invalid barcode symbologies to associate with application input calls.

### RFFile Object

The RFFile object provides basic file input/output capabilities for remote mobile devices.

### RFIO Object

The RFIO object offers basic input and output methods that function over a Wavelink wireless network.

**RfMenu Object**

The RfMenu object creates menus on a mobile device and returns the user selected option.

**RfError Object**

The RfError object defines and displays simple message boxes within your application.

**RfTerminal Object**

The RfTerminal object provides current terminal state information and alters certain terminal options.

**RfTone Object**

The RfTone object utilizes the audio capabilities of a remote mobile device from your applications.

**Widget Objects**

The Wavelink Development Library includes the following objects that are supported on GUI-based devices:

**WaveLinkFactory Object**

The WaveLinkFactory object contains the methods necessary to build all types of widget objects.

**WaveLinkMenubarInfo Object**

The WaveLinkMenubarInfo object contains the methods necessary to create and manipulate a list of menus that a menubar widget can contain.

**WaveLinkScribblePad Object**

The WaveLinkScribblePad object displays a drawing pad and saves the newly created drawing to a specific (.JPG or .BMP) file.

**WaveLinkSignon Object**

The WaveLinkSignon object displays a screen that prompts the user for a name and password and uses that information to verify or deny access to an application.

**WaveLinkWidget Object**

The WaveLinkWidget object contains the methods necessary to build and modify all types of widget objects.

**WaveLinkWidgetCollection Object**

The WaveLinkWidgetCollection object logically groups widget objects together.



## Chapter 2: Application Frameworks

This section includes information for including the library in each respective development environment. For additional information and programming considerations, see the Wavelink Development Library documentation.

---

**NOTE** For information about running Wavelink applications from the server, see the Wavelink Server documentation.

---

### Referencing the Wavelink COM Development Library

Before developing wireless applications, you must include the Wavelink COM Development Library in your specific development environment. This provides access to the custom objects and methods necessary for wireless application development.

#### Including WaveLink Objects in Visual Basic 6.0 Projects

To access the Wavelink Development Library in Visual Basic, you reference it in your project.

##### To include the Wavelink Development Library in a Visual Basic Project:

- 1 Open the Visual Basic project to which you want to add the Wavelink Development Library.
- 2 Remove the default form.

In the Visual Basic Project window, right-click the default form and select `Remove Form1`.

- 3 Add a module to the project.

In the Visual Basic Project window, right-click the project and select `Add > Module`.

- 4 Reference the Wavelink Development Libraries

From the **Project** menu, select `References`.

Enable the **WaveLink** and **WavelinkOLE** checkbox and click `OK`.

- 5 In your project, declare any objects you plan to use.

Example: `Public wlio As New RFIO` creates a public instance of the RFIO Object.

### **Including Wavelink Objects in other COM Languages**

You can access the Wavelink Development Library in any COM-based language such as C++. The libraries are located in the following directory:

`<installpath>\Include\Libs`

In your programming environment, reference the Wavelink and WavelinkOLE type libraries and dynamic link libraries (.dll file extension) as instructed in the documentation for your specific programming environment.

## Chapter 3: Error Handling

You can check for returned error values when calling a WaveLink Development Library method within your application. It is of *absolute* importance to check for returned error values after making an input call such as RFInput or GetEvent. The reason for this is that proper functioning of your wireless applications is dependent upon certain returned error values.

For example, on a disconnect command from the WaveLink Administrator a specific error message is returned to your application. This error message is returned to the first WaveLink function that communicates with the mobile device (for example, an RFIO method) before the device is physically disconnected from the network. Every subsequent WaveLink function following disconnect will then return the error message. Upon receiving this error message, you must instruct the application to exit. If this error message is not trapped at this time, the application can get caught in a loop and consume remaining system resources while waiting for further user input from a disconnected device.

The Wavelink COM Development Library uses the RFGetLastError method to return error values. The following code snippet shows an example of using this method with an RFInput method call.

```
' VB Sample Code
wlio As New RFIO
nError As Integer
.
.
.
' In this example, we use an RFInput call
pszWelcomeIn = wlio.RFInput(pszWelcomeInDefault, 1, 1, 0
                        33, "BarCode", WLCAPSLOCK, WLNO_RETURN_BKSP)
' It is critical to check for errors after communication
' occurs between the app and the RF device.
nError = wlio.RFGetLastError
If nError <> WLNOERROR Then
    GoTo ExitApp
Else
    ' process input
```

The RFGetLastError method returns WLNOERROR when no error occurred during the method invocation. The method returns a variety of constants that indicate specific errors. You can use the method as needed to check for specific errors (for example, when you send data to a port), for debugging

purposes, or in order to execute conditional actions. Most of the objects in the Wavelink Development Library contain an `RFGetLastError` member function. See the `RFGetLastError` method in the COM Development Library documentation for more information.

---

**NOTE** The Wavelink COM Development Library also returns a function status on selected methods. See the reference information on specific methods in the COM Development Library for more information.

---



## Chapter 4: I/O Techniques

In addition to handling the technical issues associated with wireless communication, Wavelink incorporates numerous built-in features that directly support the specialized I/O requirements of wireless applications. These specialized requirements include the need to handle out-of-range mobile devices and to limit radio traffic while increasing an application's efficiency.

In the following sections, you will learn about automated features and programming techniques specific to Wavelink I/O:

- How Wavelink handles [out-of-range devices](#)
- How Wavelink [optimizes the datastream](#)
- How to [print data](#) on the mobile device screen
- How to [return input](#) from the device, using built-in Wavelink options for maximum control.
- How to take advantage of the [high speed display](#) features included with Wavelink.
- How to [automate workflow](#) for end users by building intuitive applications that incorporate barcode symbologies and audio tones.
- How to incorporate [navigational aids](#) such as function keys and menus.
- How to use [message boxes](#) to display information to the user.

### Handling Out-of-range Devices

Wavelink Studio COM tailors all aspects of I/O specifically for the wireless environment and takes advantage of the Wavelink thin-client/server architecture. Because users can roam in and out of range in wireless environments, Wavelink incorporates features to automatically handle these events within its architecture.

Wavelink Studio COM follows a continuously-connected, server-side architecture that incorporates an intelligent thin client on the mobile device. In this model, data travels from the mobile device through the server to the

application and vice-versa. Messaging from the application to the server takes place from the user application through the API.

In the Wavelink architecture, the duty cycle for the application/server is between 1% and 10%, and the processing time is on the order of milliseconds. The application/server wait time, however, can be several seconds or longer, while the application waits for the user to respond to an input prompt.

No unsolicited data transmits during the application/server wait time; the server blocks the transmission of data, thus maintaining the IP stack, while waiting for returned input. This provides considerable control over the wireless network, as the mobile device can roam out of range without dropping the connection. Before responding to the server with user input, the client buffers the data and verifies its "range" status; if the client is out of range, it displays a message directing the user to move within range. When the user complies, the client automatically completes the input call by sending the data.

## Optimizing RF Traffic

In the Wavelink architecture, the API stores data destined for the mobile device in an output queue, sending the data to the device when the RF packet reaches 100 bytes in size (or until the application makes an input call). By incorporating this design, the API combines multiple method calls into single RF packets, greatly reducing wireless network traffic.

In addition, some Wavelink objects use built-in Wavelink Client features to speed the application by first storing and then using files on the mobile device. This methodology further reduces network traffic. The Wavelink objects and the file extensions for the files they store are as follows:

<b>WaveLinkAuxPort object</b>	.CFG
<b>RFIO object</b>	.SCR
<b>RFMenu object</b>	.MNU
<b>RFTone object</b>	.TON
<b>RFBarcode object</b>	.BAR

When storing and accessing files using these objects, it is not necessary to include the extensions. However, if you interact with these files using the

RFFile object, it is necessary to include the appropriate file extension (or ".\*" for all file extensions). All filenames can be up to eight characters in length.

## Displaying Data on the Device Screen

The RFPrint method of the RFIO object allows you to print static or dynamic data on the mobile device screen. The following code shows an example of using RFPrint. In this example, you position the display text on the screen and instruct RFPrint to clear the screen before displaying text.

```
Public wlio As New RFIO
.
.
.
wlio.RFPrint 1, 0, "      Enter Product      ", _
            WLCLEAR + WLREVERSE
```

The arguments set within the preceding RFPrint call have the following effects:

- |         |   |
|---------|---|
| 1       | Sets the starting left coordinate for the input prompt, in cell coordinates (columns). Columns are numbered from the left, starting with column 0.  |
| 0       | Sets the starting top coordinate for the input prompt, in cell coordinates (rows). Rows are numbered from the top, starting with row 0.   |
| WLCLEAR | Sets an output mode for the RFPrint call. Output modes can: clear all or part of the screen before displaying output, format the output, or extend the functionality of the RFPrint method. You can use multiple output modes within the RFPrint call. For example, by passing WLCLEAR + WLREVERSE, as shown in this argument, RFPrint clears the screen before displaying the text and displays the text in reverse color. |

New data overwrites old data on the device screen on a cell-by-cell basis. However, to clear all or part of a screen that you do not specifically overwrite, you must use one of the output modes included with RFPrint, such as WLCLEAR or WLCLREOLN.

The API sends data to the device when the output buffer reaches 100 bytes or when the application makes an input call. The next section shows how to add an input call to your application using RFInput.

---

**NOTE** For more information about RFPrint, see the *Wavelink Studio COM Development Library* documentation.

---

## Using RFInput

The RFIO object handles the majority of I/O requirements in a Wavelink application, and the RFInput method is the principle means by which it returns user input. Like all Wavelink input methods, RFInput automatically incorporates blocking to maintain the wireless connection.

RFInput displays an input prompt on the screen of the mobile device, at coordinates specified within the RFInput call.

Using the RFInput method involves several aspects:

- 1 [Invoking RFInput to display output.](#)
- 2 [Waiting for user input.](#)
- 3 [Processing returned input.](#)

### Invoking RFInput

Before invoking RFInput, you typically instruct the user to take action. In this example, RFPrint calls instruct the user to enter the item quantity.

```
Public wlio As New RFIO
Dim inQty As String
.
.
.

wlio.RFPrint 0, 0, "      Enter Item      ", _
      WLCLEAR + WLREVERSE
wlio.RFPrint 0, 1, "      Quantity      ", WLREVERSE
```

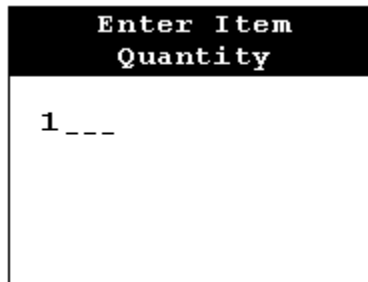
Next, use RFInput to display an input prompt. The inQty variable contains the default value. In most cases, you simply pass an empty string ("") for the

default value, but here you pass a variable to set the default value to "1" (string value).

```
inQty = "1"
inQty = wlio.RFInput(inQty, 4, 2, 4, "", NORMALKEYS, _
    WLNO_RETURN_BKSP + WLNUMERIC_ONLY)
' Add error handling code.
```

When the application invokes the RFInput method, the API automatically clears the output buffer, sending data to the mobile device for immediate display.

Figure 4-1 shows how the preceding code will appear on the mobile device.



**Figure 4-1.** Example of a Screen Using RFInput

The arguments set within the previous RFInput call have the following effects:

- |       |  |
|-------|--|
| inQty | Contains the display value to appear in the input prompt. When the user presses Enter without keying or scanning data, RFInput returns a Chr\$(13) to the application. The default value itself is NOT returned to the application. Consequently, to process the user's response based on a default value, you must first process input based on the Enter key, Chr\$(13). |
| 4     | Sets the maximum input length to 4 characters. RFInput automatically returns input when the input reaches the maximum length. This argument also defines the number of fill characters that appear in the input prompt—underscores by default. (Note: You can change the default fill character using the SetFillCharacter method.)  |

2	Sets the starting left coordinate for the input prompt, in cell coordinates (columns). Columns are numbered from the left, starting with column 0.
4	Sets the starting top coordinate for the input prompt, in cell coordinates (rows). Rows are numbered from the top, starting with row 0.
""	Defines the barcode configuration for the RFInput call. The empty string ("" ) instructs RFInput to use the default barcode configuration. See <a href="#">Automating the Workflow</a> on page 27 for more information about working with barcodes. (Note: If you want to prevent scanned input, pass WLDISABLE_SCAN as the input mode argument).
NORMALKEYS	Defines the keyboard shift state, either normal (NORMALKEYS) or all caps (CAPSLOCK).
WLNO_RETURN_BKSP	Defines the <i>input mode</i> for the input prompt. Input modes can control or alter the actual input, or extend the functionality of the RFInput method. Examples of using different <a href="#">input modes</a> are provided later in this section. You can use multiple input modes within the RFInput call, for example, by passing WLNO_RETURN_BKSP + WLNUMERIC_ONLY in this argument.

See the following sections for more information about RFInput:

- [Waiting for User Input](#) on page 18
- [Processing Returned Input](#) on page 19

## Waiting for User Input

By default, RFInput returns input from the user when one of the following conditions occur:

- The user presses the Enter key or a function key combination (for example, Ctrl+X, an arrow key, F1 to F9).
- The user scans a barcode.

- Input reaches the maximum input length, as defined within the RInput call.
- The user presses the Backspace key when the "cursor" is in the first position of the input prompt.

Using input modes such as WLNO\_RETURN\_BKSP, you can change the default conditions in which RInput returns input. See [Changing the Conditions for Returning Input](#) on page 21 for more information.

## Processing Returned Input

In the input call shown previously (see [Invoking RInput](#) on page 16),

```
inQty = wlio.RInput(inQty, 16, 2, 4, "", NORMALKEYS, _
                  WLNO_RETURN_BKSP + WLNUMERIC_ONLY)
```

inQty will contain user input. In addition to returning the user input, RInput returns an *input type*. You normally use the input type when processing returned data from the RInput call. RInput returns one of seven possible input types, including the following:

WLKEYTYPE	- Keypad/keyboard input
WLSCANTYPE	- Scanned input
WLCOMMANDTYPE	- Function key input

When processing input, use the LastInputType method to return the input type. The following example shows how to process input from RInput using a Case statement and the input type. In this case, the code processes the default quantity of "1" by re-assigning the default quantity to inQty.

```
Dim inType As Integer
.
.
.
inType = wlio.LastInputType()
Select Case inType
Case WLKEYTYPE           ' keyboard
    If inQty = Chr$(13) Then ' [ENTER] key
        inQty = "1"
        ProcessQtyIn       ' input handling procedure
    Else
        ProcessQtyIn
```

```

        End If
    Case WLCOMMANDTYPE          ' command key
        .
        .
        .
    Case Else
End Select

```

## Input Modes

You can easily extend the efficiency of I/O in your application by using input modes to control the input peripherals on the mobile device (such as the keyboard and scanner). In addition, input modes for the feature-rich `RFInput` method allow you to: control the content of input, define how characters echo in the input prompt, change the conditions in which `RFInput` returns input to the application, and carry out other miscellaneous actions.

### Controlling the Input Type

To prevent users from passing bad data to your application, you can disable input peripherals on the device. For example, at a prompt where you want the user to scan an item, you can easily disable the keypad and the function keys, forcing the user to use the scanner. To limit the type of input from the user, pass one or more of the following arguments within the `RFInput` call:

- `WLDISABLE_SCAN`. Disables the scanner on the mobile device.
- `WLDISABLE_KEY`. Disables the keypad on the mobile device.
- `WLDISABLE_FKEYS`. Disables the function keys on the mobile device.

The following line of code shows an example of using these input modes:

```

inQty = wlio.RFInput(inQty, 16, 2, 4, "", NORMALKEYS, _
    WLNO_RETURN_BKSP + WLDISABLE_SCAN + _
    WLDISABLE_FKEYS)

```

### Controlling the Content of Input

You can use input modes that limit the content of input. For example, if you want the user to enter a quantity, you can easily set `RFInput` to disable all alpha characters on the keyboard, forcing the user to enter a number.

To limit the content of input, pass one of the following input modes:



- `WLALPHA_ONLY`. Disables numeric characters on the keypad.
- `WLNUMERIC_ONLY`. Disables the alphabetic characters on the keypad.

### Changing the Conditions for Returning Input

You can also use input modes that change the conditions in which `RFInput` returns data.

- `WLNO_RETURN_FILL`. Input returns only when the user presses the Enter key. Input does *not* return when user input reaches the maximum length. Use this input mode if your input length requirements exceed the available space on the screen of the mobile device.
- `WLFORCE_ENTRY`. Input returns only when the user enters an actual value at the prompt. Input does *not* return when the user presses Enter, unless an actual value has been previously keyed in.
- `WLNO_RETURN_BKSP`. Input does *not* return when the user presses the Backspace key while the "cursor" is in the first position of the input prompt. Use this input mode to prevent the accidental return of input by users who are manually correcting their input. For example, if a user keys in four characters they may accidentally key in five Backspace characters to fix a mistake. `RFInput` would normally return input on the fifth occurrence of the Backspace key. `WLNO_RETURN_BKSP` is also useful for users who habitually press the Backspace key because they expect to move to a previous field on the screen.

---

**NOTE** Input modes that disable options on the device (for example, `WLDISABLE_FKEYS`) also effectively change the conditions in which `RFInput` returns input.

---

### Formatting Echoed Characters in the Input Prompt

When the user types in text at the input prompt, you can define the characters echoed on the screen of the mobile device.

- `WLECHO_ASTERISK`. Echoes (i.e., displays) all characters on the device using asterisks. This input mode is useful for hiding passwords from casual observers.

- **WLSUPPRESS\_ECHO.** Disables the echoing of characters on the device. You might want to use this mode for passwords or situations where the user can press any key.

### Other Options

- **WLCLR\_INPUT\_BUFFER.** Clears the input buffer of pending data. Use this RFINput mode following the use of timed messages created with the RFError object. This prevents the user from entering unwanted data into the input buffer after the message expires.
- **WLBACKLIGHT.** Turns on the display backlight on the mobile device when the application invokes RFINput.
- **WLIGNORE\_CRLF.** Includes carriage return (CR) or line feed (LF) commands within a single input packet. By default, when input contains ASCII character codes for a carriage return or line feed, RFINput breaks the input into separate input packets. If a barcode contains internal carriage returns or line feeds, you must use WLIGNORE\_CRLF.

See the Wavelink Studio COM Development Library documentation for additional input mode options.

### Input Timeouts

By default, [RFINput prompts](#) do not expire. However, you might want to set an expiration time for the prompt so that you can continue the application. Use the [SetInputTimeout method](#) to set the default, passing in the number of seconds that must elapse before the prompt expires.

```
Public wlio As New RFIO
Dim inData As String
.
.
.
wlio.SetInputTimeout 5
inData = wlio.RFINput(...)
' Add error handling code.
```

When an RFINput prompt expires, it returns an input type of WLTIMEDOUT to the application. The following code fragment shows how you can incorporate this into your Case statement.

```
Dim inType As Integer
.
```

```
.  
.inType = wlio.LastInputType()  
  Select Case inType  
    Case WLTIMEDOUT           ' prompt expired  
      ContinueProcessData
```

## Using High Speed Display

Wavelink provides high-speed display through the RFIO object. This methodology allows you to store screen files on the mobile device and—when they are needed—retrieve them for immediate display. Whenever you can retrieve stored information from the device memory, you reduce RF traffic and speed up your application, allowing the greatest number of users rapid access to your applications.

The PushScreen, PullScreen, and RestoreScreen methods of the RFIO object provide the high-speed display functionality. The PushScreen method is the starting point; use this method to store the current screen as a file on the device with a .SCR extension.

---

**NOTE** It is recommended that you use the PushScreen method whenever you might re-use a screen. For screens that contain a significant amount of text, such as a Help screen, it is especially important to incorporate the Push/Pull/RestoreScreen methodology.

---

When you need to display the stored screen, you can use either the PullScreen or RestoreScreen method. PullScreen displays the specified screen while deleting it from the mobile device. RestoreScreen also displays the specified screen, but does not delete it from the device. For this reason, RestoreScreen is the optimal choice when you design the application for multiple re-displays.

The following sections contain code samples that demonstrate how to:

- [Store a screen](#)
- [Store a screen template](#)

## Storing Screens

The following code shows how you can use high-speed techniques for a help screen in your application. In this example, you save the current screen as a .SCR file on the device (CurntScr), then check to see if the application has previously displayed the help screen. If it has, you can immediately restore the help screen with RestoreScreen. Typically, you nest the following code within the Case statement (not shown) after the user presses the F1 function key.

---

**NOTE** The following code uses the RFTerminal object to return the total number of display rows on the device. See the *WaveLink Development Library* documentation for more information about this object.

---

```
Public wlio As New RFIO
Public wlterm As New RFTerminal
Dim firstItemHelp As Boolean
.
.
.
wlio.PushScreen "CurntScr"
If helpPushed = False Then
    wlio.RFPrint 0, 0, " Enter the Item ", WLCLEAR + WLNORMAL
    wlio.RFPrint 0, 1, " to Price Check ", WLNORMAL
    wlio.RFPrint 0, (wlterm.TerminalHeight - 1), _
        "      Press Any Key      ", WLREVERSE + WFLUSHOUTPUT
    wlio.PushScreen "helpscr", WL_IGNOREWIDGETS
    helpPushed = True
Else
    wlio.RestoreScreen "helpscr"
End If
wlio.GetEvent
' Add error handling code.
wlio.PullScreen "CurntScr"
```

Because the Wavelink Server does not send data to the mobile device until the RF packet reaches 100 bytes in size (or until the application makes an input call), it is important to clear the output queue before using PushScreen. This guarantees that the screen you store on the device will be complete. As shown in the preceding example, use the WFLUSHOUTPUT output mode within an RFPrint call to clear the output queue.

As an alternative to using `WLFLUSHOUTPUT`, you can use the `RFFlushoutput` method to clear the output queue. See the *Wavelink Studio COM Development Library* documentation for more information.

In the preceding code, the `GetEvent` method effectively pauses the application before restoring the original screen (`CurntScr`). `GetEvent` returns a single input event from the user (single key, function key combination, scan- or widget-based event). In this case, your intention is to pause the application while the user reads the help screen.

```
wlio.GetEvent
```

After the user presses a random key, you restore the original screen. You can use either `RestoreScreen` or `PullScreen` to restore the original screen. In this case, because the application stores the current screen each time the user calls for Help, you don't need to keep the screen file, so you delete it with `PullScreen`.

```
wlio.PullScreen "CurntScr"
```

As part of your cleanup operation for the application, delete the help screen file from the device using the `RFFile` object:

```
Public wlFile As New RFFile
.
.
.
wlFile.RFDeleteFile("ItemHelp.scr")
```

## Storing Screen Templates

In addition to storing complete screens with `PushScreen`, you can use it to store screen “templates”—partial screens which can be filled with dynamic data on the fly.

The following code shows an example of storing a screen template for an “item lookup” screen in an application:

```
wlio As New RFIO
.
.
.
wlio.RFPrint 0, 0, "          Wavelink          ", WLCLEAR
      + WLREVERSE
```

```

wlio.RFPrint 0, 1, " Price Check Demo ", WLNORMAL
wlio.RFPrint 0, 3, "Item: ", WLNORMAL
wlio.RFPrint 0, 5, "Desc: ", WLNORMAL
wlio.RFPrint 0, 8, "Cost: ", WLNORMAL
wlio.RFPrint 0, 10, "Qty : ", WLNORMAL
wlio.RFPrint 0, 12, "On Order: ", WLNORMAL
wlio.RFFlushoutput
wlio.PushScreen "LkupScrn"

```

The application stores the screen as a .SCR screen file on the device named LookupScrn.

When you show the LookupScrn, you can display dynamic data from your database using RFPrint calls, and restore the screen file template you previously created:

```

Public wlio As New RFIO
Dim itemID As Integer
Dim itemDesc As Integer
Dim itemCost As Integer
Dim itemQty As Integer
Dim itemOnOrder As Integer
.
.
.
' Use RFInput to obtain an Item code.
' Obtain the required information from your database
' and store:
' the item ID in itemID,
' the item description in itemDesc,
' the price in itemCost,
' the quantity in itemQty,
' the number on order in itemOnOrder.
.
.
.
wlio.RestoreScreen "LkupScrn"
wlio.RFPrint 6, 5, itemID, WLNORMAL
wlio.RFPrint 6, 3, itemDesc, WLNORMAL
wlio.RFPrint 0, 8, itemCost, WLNORMAL
wlio.RFPrint 0, 10, itemQty, WLNORMAL
wlio.RFPrint 0, 12, itemOnOrder, WLNORMAL

```

```
wlio.GetEvent
```

## Automating the Workflow

Because many wireless applications must support barcode scan events, Wavelink includes support for high-efficiency scan-based applications. However, you must consider several issues when designing these applications.

First, scan-based applications typically involve interaction with a database. It is vital to eliminate errors in such cases to avoid corrupting your database. For example, end users might scan the wrong item or scan the wrong barcode on the correct item. Product labels increasingly contain multiple barcodes, so this concern is critical. You can use Wavelink's built-in barcode features to easily verify that the correct barcode has been scanned, actually disabling all barcodes that do not meet a specific barcode definition.

The second issue you must consider when designing scan-based applications is that many of these applications must also support high-volume transactions. In these cases, it is vital to build screens that process scanned information with maximum efficiency. The examples in this section are excerpts from an event-driven cycle count application. The goal is to build a highly intuitive application that allows the user to quickly and efficiently scan locations, items, and quantities from a single RFInput prompt.

The following sections contain sample code related to four development tasks:

- [\*Designing the Application\*](#)
- [\*Adding Bar Code Symbologies to your Applications\*](#)
- [\*Using Bar Code Symbologies\*](#)
- [\*Adding Tones to Your Applications\*](#)

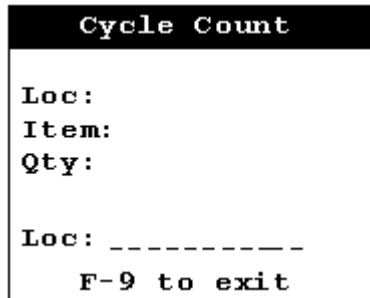
### Designing the Application

This section provides a brief overview of the cycle count application, showing how it appears to the end user, and describing what can happen at each stage. Figure 4-2 shows how the application initially appears on the mobile device.

---

**NOTE** See [Automating the Workflow](#) on page 27 for an overview of scan-based applications.

---



The screenshot shows a terminal window titled "Cycle Count". Inside, the text is as follows:

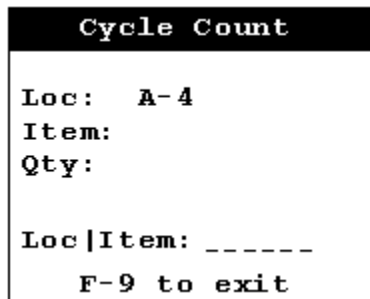
```
Loc:
Item:
Qty:

Loc: -----
      F-9 to exit
```

**Figure 4-2.** *The Cycle Count Application Screen, Initial State*

In this application, the user is forced to either scan a location or hit F-9 to exit when the screen initially appears. The application does not accept any other input.

When the user enters a location, the application attempts to validate the location. If the user enters an invalid location, the application displays an error and loops without changing anything. However, if the user enters a valid location, the valid location appears following the `Loc:` output field. Figure 4-3 shows how the screen appears at this point.



The screenshot shows the same terminal window titled "Cycle Count". The text is now:

```
Loc:  A-4
Item:
Qty:

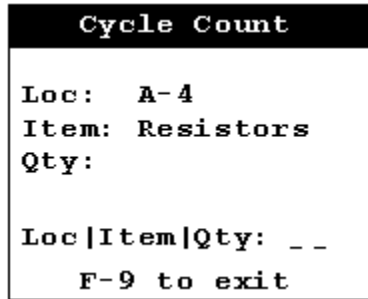
Loc|Item: -----
      F-9 to exit
```

**Figure 4-3.** *The Cycle Count Screen with Valid Location*

After looping, the application resets the barcode configuration to allow both location and item barcode scans. If the user scans another location at this point, the application processes the location as it did previously. If, on the other hand, the user scans an item, the application attempts to validate the



item. If the item is invalid, the application displays an error and loops without changing the location. If the item is valid, the application displays the item description after the `Item:` output field and increments the item quantity by 1 in a table or recordset. Figure 4-4 shows how the screen might appear at this point.



**Figure 4-4.** *The Cycle Count Screen with Valid Item*

Once the user scans a valid item, the application allows any of the following input: a new location, a new item, the same item (in which case the application increments the quantity by 1), or a keyed-in quantity.

Whenever the user enters a new location, the application stores the current item quantity in the database.

The following sections contain the code excerpts that make this application work:

- [Adding Bar Code Symbolologies to your Applications](#)
- [Using Bar Code Symbolologies](#)

## Adding Bar Code Symbolologies to your Applications

Before you can write code to build the [cycle count](#) screen, you must create one or more *barcode configurations*. A barcode configuration is a file that defines the decode state, expand state, minimum length, and maximum length for one or more specific barcode symbolologies. The features specific to each barcode symbology are as follows:

- **Decode State.** Determines whether the scanner decodes a scanned barcode of the specified symbology.

- **Expand State.** Determines whether the scanner expands a scanned barcode of the specified symbology. The expand state is relevant for UPC-E0 barcode types and should be set to False in all other barcode symbologies.
- **Minimum Length.** Defines the minimum length of a scanned barcode of the specified symbology.
- **Maximum Length.** Defines the maximum length of a scanned barcode of the specified symbology.

The first step in your program is to "push" the default barcode configuration. Mobile devices enabled for barcode use typically contain a generic, default barcode configuration. Unlike pushing screens, the reason you push a barcode configuration is not to speed the application, but to replace it with one or more custom barcode configurations—ones that are ideal for your application needs.

```
Public wlbar As New RFBBarcode
.
.
.
wlbar.PushBarcode "Defltbar"
```

After pushing the default barcode configuration, create the first barcode configuration you will need—in this example a barcode configuration named Code39. Code\_39 barcodes represent warehouse locations. Before adding the barcode, clear the current barcode object of any barcode symbology definitions.

```
wlbar.ClearBarcodes
wlbar.AddBarcode CODE_39, False, DECODEON, 12, 12
```

Bar code configurations are normally attached to RFInput calls. Before you can use a barcode configuration within an RFInput call, however, you must store it. When you call the StoreBarcode method, you store the specified barcode configuration as a .BAR file on the mobile device. The second argument passed in StoreBarcode (BCDENABLED) defines the default decode state for all other barcodes NOT explicitly allowed. In the following example, BCDENABLED enables all barcodes except Code\_39 barcodes of length 12.

```
wlbar.StoreBarcode "Code39", BCDISABLED
```

Next, add a second barcode symbology definition to the current barcode object and store it as a new barcode configuration. The new barcode configuration, `Codeboth`, accepts either a `Code_39` barcode of length 12 or a `UPC_A` barcode of any length. If you want to remove the minimum and maximum length restriction from the symbology, pass arguments of 0 for the minimum and maximum length.

```
wlbar.AddBarcode UPC-A, False, DECODEON, 0, 0
wlbar.StoreBarcode "CodeBoth", BCDISABLED
```

In this application, `UPC_A` barcode symbologies represent item descriptions. As you will see later, by creating a barcode configuration that accepts both `Code_39` and `UPC_A` barcodes, you can scan either a warehouse location or an item from a single prompt, and process each accordingly.

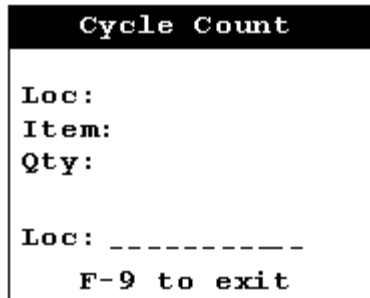
To use the barcode symbologies you have created, see [Using Bar Code Symbologies](#).

## Using Bar Code Symbologies

To invoke the main screen for the `cycle count` application, you add the following code:

```
Public wlio As New RFIO
Public wlterm As New RFTerminal
.
.
.
wlio.RFPrint 0, 0, "    Cycle Count    ", WLREVERSE
wlio.RFPrint 0, 2, "Loc : ", WLNORMAL
wlio.RFPrint 0, 3, "Item: ", WLNORMAL
wlio.RFPrint 0, 4, "Qty : ", WLNORMAL
wlio.RFPrint 0, (wlterm.TerminalHeight - 1), _
    "    F-9 to exit    ", WLREVERSE
```

When the application displays the cycle count screen, the static output fields remain on screen until you clear them or overwrite them. Figure 4-5 shows how this screen initially appears on the mobile device.



**Figure 4-5.** *The Cycle Count Application Screen, Initial State*

Your next step is to build a loop structure that contains three subroutines. The three subroutines handle the following tasks: preparing the RFInput call, the invoking RFInput and returning input, and processing the returned input.

```
Dim AppFinish As Boolean
AppFinish = False
.
.
.
Sub InputLoop()

    While AppFinish = False
        PrepareInput
        ReadData
        ProcessInput
    Wend

End Sub
```

The first subroutine, `PrepareInput()`, determines whether the user has previously scanned a valid location or item, and it sets the barcode type and the input mode to be passed into the `RFInput` call accordingly. When the application initially processes this code, the current location is an empty string (""). Therefore, to force the user to scan a `Code_39` location, you set the barcode configuration to `Code39` and disable the keypad.

On subsequent iterations of `PrepareInput()`, the current location is set, but the current item might be an empty string. If the current item is an empty string, you set the barcode configuration to `CodeBoth`. This allows the user to scan either an item or a new location. Finally, if a valid item was previously set, you want the user to scan an item, key in a quantity, or scan a new location. In

this case, you set the barcode configuration to CodeBoth and do NOT disable the keypad.

Here is the code for the PrepareInput() subroutine:

```
Dim barCfg As String
Dim inMode As Int
Dim displVal As String
Dim currentLoc As String
Dim currentItem As String
currentLoc = ""
currentItem = ""
.
.
.
Sub PrepareInput()
    If currentLoc = "" Then
        barCfg = "Code39"
        inMode = WLDISABLE_KEY + WLNO_RETURN_BKSP
        displVal = "Loc:"
    ElseIf currentItem = "" Then
        barCfg = "CodeBoth"
        inMode = WLDISABLE_KEY + WLNO_RETURN_BKSP
        displVal = "Loc|Item:"
    Else
        barCfg = "CodeBoth"
        inMode = WLNO_RETURN_BKSP
        displVal = "Loc|Item|Qty:"
    End If
End Sub
```

The next subroutine, ReadData(), invokes RFInput and stores the returned data. This subroutine uses the values previously set for the current barcode configuration and input mode. You store the returned data, the returned input type, and the returned barcode type in three variables. To return the barcode type, use the LastBarcodeType method.

```
wlio As New RFIO
Dim inData As String
Dim inType As Integer
Dim barType As Integer
.
.
.
Sub ReadData()
```

```

        inData = wlio.RFInput(displVal, 15, 0, 6, NORMALKEYS, barCfg,
-
        inMode)
        ' Add error handling code.
        inType = wlio.LastInputType()
        barType = wlio.LastBarcodeType()

End Sub

```

To process input returned from the RFInput call, create the ProcessInput() subroutine. In this subroutine, if the user presses F9, you set the variable, AppFinish, to True and exit the subroutine. If the user scans a Code\_39, you must validate the location. A valid location returns True, in which case you update the screen with the current location and store the current table or recordset in your database. Invalid locations return False, in which case you exit the subroutine.

If, on the other hand, the user scans something other than Code\_39, it must be an item (UPC\_A). In this case, you validate the item and update the screen with the current item. Then you update the item in the table with a quantity of 1.

Finally, if the user keys in a value, you update the quantity in the table with the keyed in value. Because you already automatically incremented the quantity by 1 when the user first scanned the item, you can subtract 1 from the keyed-in quantity.

The code for the ProcessInput() subroutine is shown here:

```

Dim inQty As Integer
.
.
.
Sub ProcessInput()
    If inType = WLCOMMANDTYPE Then
        If data = 9 Then
            AppFinish = True
            StoreItemQty
        End If
        Exit Sub
    ElseIf inType = WLSCANTYPE Then
        If barType = Code_39
            If ValidateLocation() Then
                wlio.RFPrint 6, 2, currentLoc, WLNORMAL + WLCLREOLN
                wlio.RFPrint 6, 3, "", WLNORMAL + WLCLREOLN + _

```

```

        WLFLUSHOUTPUT
        StoreItemQty          ' Store item/quantity info
                              ' from previous location in
                              ' your database.

        currentItem = ""
    End If
    Exit Sub
Else
    If ValidateItem() Then
        wlio.RFPrint 6, 3, currentItem, WLNORMAL + _
            WLCLREOLN + WLFLUSHOUTPUT
        inQty = 1
        UpdateQty inQty      ' Update item quantity in your
                              ' table by 1.

    End If
    End If
Else
    inQty = CInt(inData)      ' Store keyed input as a string
    UpdateQty inQty-1        ' Update item quantity in your
                              ' table by keyed in value -1.

    End If
End Sub

```

The RFPrint call from the preceding code, shown again here,

```
wlio.RFPrint 6, 3, currentLoc, WLNORMAL + WLCLREOLN
```

uses WLCLREOLN as an output mode; WLCLREOLN clears the output to the end of the line, row 3. Because the user might have previously scanned another location, it is important to clear the old location before displaying the new one. Use the same methodology to replace the old item with a new one.

---

**NOTE** Before you can display the valid location and item with RFPrint, you must store the valid location and item in the relevant variables, currentLoc and currentItem, when you create the ValidateLocation() subroutine (not shown), and the ValidateItem() subroutine (not shown).

---

At the conclusion of your application, use the PullBarcode method to restore the default barcode configuration, and the DeleteBarcodeFile method for cleanup:

```

wlbar.PullBarcode "Defltbar"
wlbar.DeleteBarcodeFile "Code39"

```

```
wlbar.DeleteBarcodeFile "CodeBoth"
```

## Adding Tones to Your Applications

You can greatly speed up scanning tasks for end users by incorporating aural tones in your applications. The advantages of using tones increases in proportion to the volume of transactional events you must support.

In a receiving application, for example, the end user might be required to scan a palette of 50 boxes. If the design of the application forces the user to look at a screen after every scan, scanning will be slow, and the job will be relatively difficult, increasing the likelihood of user error.

However, by using tones in combination with well-designed applications, the user can scan through each item in rapid succession, without ever checking the screen to verify the output. With tones, you can accomplish this by informing users aurally that they scanned the correct item. When users scan the wrong item or barcode, you can alert them with a different tone. In this scenario, you can handle all error checking behind the scenes using some of the techniques mentioned previously.

A *tone configuration* is a file that defines the frequency and duration of one or more tones. Tone configuration files are stored in the mobile device memory with a .TON extension.

In the following example, you create a tone configuration and store it on the mobile device. When creating a tone configuration, use the AddTone method to pass the frequency for each tone, in hertz, as the first argument, and the duration of the tone, in milliseconds, as the second argument.

```
wlTone As New RFTone
.
.
.
wlTone.ClearTones
wlTone.AddTone 880, 100
wlTone.AddTone 440, 200
```



```
wlTone.StoreTones "ErrBeep"
```

Before you can play any tone configuration, you must store it on the mobile device. The StoreTones method call from the preceding code, shown here:

```
wlTone.StoreTones "ErrBeep"
```

stores the current tone object as a tone configuration named "ErrorBeep." You will use this tone configuration to inform the user that an error has occurred during a scan operation.

---

**NOTE** The default tone plays automatically following successful scans. For maximum efficiency, avoid attaching an extra tone to the default tone.

---

Using the tone you've created within the [cycle count](#) application, the code for processing input now invokes the PlayTone method, and looks like this:

```
Public wlTone As New RFTone
Dim inQty As Integer
.
.
.
Sub ProcessInput()
    If inType = WLCOMMANDTYPE Then
        If data = 9 Then
            AppFinish = True
            StoreItemQty
        End If
    Exit Sub
    ElseIf inType = WLSCANTYPE Then
        If barType = Code_39
            If ValidateLocation() Then
                wlio.RFPrint 6, 2, currentLoc, WLNORMAL + WLCLREOLN
                wlio.RFPrint 6, 3, "", WLNORMAL + WLCLREOLN + _
                    WLFUSHOUTPUT
                StoreItemQty           ' Store item/quantity info
                                     ' from previous location in
                                     ' your database.
                currentItem = ""
            Else
                wlTone.PlayTone "ErrBeep"
            End If
        Exit Sub
    Else
```

```

    If ValidateItem() Then
        wlio.RFPrint 6, 3, currentItem, WLNORMAL + _
            WLCLEOLN + WLFLUSHOUTPUT
        inQty = 1
        UpdateQty inQty          ' Update item quantity in your
                                ' table by 1.
    Else
        wlTone.PlayTone "ErrBeep"
    End If
End If
Else
    inQty = CInt(inData)        ' Store keyed input as a string
    UpdateQty inQty-1          ' Update item quantity in your
                                ' table by keyed in value -1.
End If
End Sub

```

To clean up at the conclusion of your application, add the following line of code:

```
wlTone.DeleteToneFile "ErrBeep"
```

## Navigating the Application

Wavelink incorporates the use of [function keys](#) and [menus](#) as a means to provide users with some control over application navigation, within the constraints defined by the application developer.

### Using Function Keys

Function keys on mobile devices include Ctrl+X, arrow keys, and Function 1 through Function 9. As mentioned previously, whenever the application returns input from the device, it also returns an input type. For function keys, the input type is WLCOMMANDTYPE. You can check for the input type using the LastInputType method following an RFInput call.

In most cases, you want to inform the user that pressing a certain key will invoke a Help screen, cause your application to exit, or do some other task. You can use an RFPrint call for this purpose.

```

wlio As New RFIO
Dim inDefault As String
Dim inData As String
.

```

```

.
.
wlio.RFPrint 0, 0, "          Wavelink          ", WLCLEAR
      + WLREVERSE
wlio.RFPrint 0, 1, "  Price Check Demo  ", WLNORMAL
wlio.RFPrint 0, 3, "Item: ", WLNORMAL
wlio.RFPrint 0, 5, "Desc: ", WLNORMAL
wlio.RFPrint 0, 8, "Cost: ", WLNORMAL
wlio.RFPrint 0, 10, "Qty : ", WLNORMAL
wlio.RFPrint 0, 12, "On Order: ", WLNORMAL
wlio.RFPrint 0, 16, "  F-1 Help   ", WLREVERSE

```

The last RFPrint call tells the user how to open the Help screen.

Store the input in inData using a call to RFInput.

```

inDefault = ""
inData = wlio.RFInput(inDefault, 12, 6, 3, ""
' Add error handling code.

```

Process the input using the LastInputType method. If the input type is WLCOMMANDTYPE, either exit or display the relevant help screen, depending on what the user pressed.

```

Dim inType As Integer
.
.
.
inType = wlio.LastInputType()
Select Case inType
Case WLCOMMANDTYPE          ' command key
  Select Case inData
    Case Chr$(24)          ' Control X - exit
      Finish
    Case "1"                ' F1 key - help
      ' Display the relevant Help screen
    End Select
Case WLKEYTYPE              ' keyboard input
  ' process keyed information
Case Else

```

```
End Select
```

---

**NOTE** Wavelink allows you to create your own custom hotkeys using the AddHotkey method of the RFIO object. See the *Wavelink Studio COM Development Library* documentation for more information.

---

## Using Menus

Wavelink incorporates special menu functionality through the RFMenu object. Although it is possible to create menus using a series of RFPrint calls and then checking the input, you can streamline the process of creating, displaying, and returning input from menus by using the RFMenu object. The advantages of using this methodology include:

- **Reduced RF traffic.** You can store the menu on the device, reducing potential RF traffic generated by repeated calls to RFPrint.
- **Simplified Coding.** The code required to process a menu selection is minimal.
- **Reduced Errors.** Wavelink menus limit user input to Ctrl+X, Clear, and the options provided in the menu. No other input is accepted.

A *menu configuration* is a file containing a single menu that consists of one or more title lines and one or more options that a user may select.

Before you create the menu configuration, reset the menu object; this will clear the RFMenu object of any titles or options currently set. This allows you to re-use the menu object as needed.

```
wlmenu As New RFMenu
.
.
.
wlmenu.ResetMenu
```

The following example creates a menu configuration named "MainMenu" and stores it on the mobile device. To add descriptive titles on the menu screen, use the AddTitleLine method. Then use the AddOption method to add five menu options to the menu configuration.

```
wlmenu.AddTitleLine "Wavelink Menu"
wlmenu.AddTitleLine " "
```

```

wlmenu.AddOption "Student Info"
wlmenu.AddOption "Scan Test"
wlmenu.AddOption "Cycle Count"
wlmenu.AddOption "Device Info"
wlmenu.AddOption "Exit"
wlmenu.StoreMenu "MainMenu"

```

Before you can use the menu configuration, you must store it on the mobile device. The StoreMenu method call from the preceding code, shown here:

```
wlMenu.StoreMenu "MainMenu"
```

stores the current menu object as a menu configuration named "MainMenu." This file acquires a .MNU extension.

After storing the menu configuration, you can use it later by calling DoMenu.

```

Public wlio As New RFIO
Dim menuRslt As Integer
.
.
.
wlio.RFPrint 0, 0, "", WLCLEAR + WLFLUSHOUTPUT
menuRslt = wlmenu.DoMenu("MainMenu")

```

DoMenu displays the named menu configuration on the screen of the mobile device and will return the numeric value of the option selected by the user. MainMenu contains five option lines; if the user chooses the second menu option ("Scan Test"), DoMenu returns a 2 to the application, storing the value in menuRslt.

If the specified menu configuration cannot be found, or if any other error occurs, DoMenu automatically returns a -2. If the user presses Ctrl+X or Clear, DoMenu automatically returns a -1. The following code checks for the numeric value of the selected option, and processes the option accordingly.

```

Select Case menuRslt
Case -2          ' an error has occurred
    ' Display an error message
    .
    .
    .
Case -1          ' Ctr or Ctrl X user exit
    ExitApp
Case Else        ' valid menu selection
    Select Case menuRslt

```

```

        Case 1      ' Student Information
            StudentScreen
        Case 2      ' Scan Test
            ScanTest
        Case 3      ' Cycle Count
            CycleCount
        Case 4      ' Device Information
            Version
        Case 5      ' Exit
            ExitApp
        Case Else   ' should not get here
            ' Display an error message
    End Select
End Select

```

At the conclusion of your application, include the following code for cleanup purposes:

```
wlmenu.DeleteMenu "MainMenu"
```

## Using Message Boxes

In writing a Wavelink application, you can choose to display text messages using either RFPrint calls or the RFError object.

The best choice is based on radio traffic considerations. RFPrint initially generates more radio traffic. However, by using PushScreen/PullScreen technology, you greatly reduce the radio traffic whenever you need to re-display a screen. You cannot use the PushScreen/PullScreen methods with the RFError object. For this reason, RFPrint is the better choice except in cases where it's unlikely the application will re-display a message.

The RFError object allows you to quickly create message boxes and display them to the user. The following example creates a message and displays it on row 0 and 1 of the mobile device. The SetErrorLine method creates the message text and sets the row number for the text. The Display method displays the message for a specified number of seconds. To display the message until the user presses any key, pass a value of 0.

```

Public wlmsg As New RFError
.
.
.

```

```
wlmsg.ClearError
wlmsg.SetErrorLine "    Invalid    ", 0
wlmsg.SetErrorLine "    Quantity   ", 1
wlmsg.Display 0
```

The user can automatically close the message box by pressing the CLR key. Because `RLError` objects might expire before the user hits a key, it is recommended that you use the input mode, `WLCLR_INPUT_BUFFER`, with the `RFin` call that follows the message box.





## Chapter 5: Widgets

Wavelink supports the creation and manipulation of widgets on mobile devices that run Palm OS or Windows CE. The Wavelink Development Library includes support for widget buttons, bitmaps, checkboxes, fields, hotspots, labels, repeater buttons, and selector triggers. The library also supports menu-based widgets such as menubars, popup triggers, and push buttons, as well as pre-defined dialog boxes for signature capture and signon.

### Using Widgets

The primary Wavelink objects you will use to create and manipulate widgets on the mobile device are:

- **WaveLinkFactory.** Use this object to build different types of widget objects.
- **WaveLinkWidget.** Use this object to modify, manipulate, and identify a widget object.
- **WaveLinkWidgetCollection.** Use this object to store, display, and manipulate a group of widget objects.

The examples in this section describe the first screen in a signature capture operation. The initial screen prompts the user for a name, a file name for the captured signature, and then requires the user to press the OK button before continuing the operation.

When you initialize your widget-based objects, you can also create a variable of type `WaveLinkWidget` for widget objects with which the user will interact. In this case, you create variables for the "field" widgets so you can process these widgets later based on their unique identifiers.

```
Public wlwidcoll As New WaveLinkWidgetCollection
Public wlfo As New RFIO
Public wlfactory As New WaveLinkFactory
Dim mynameFld As WaveLinkWidget
Dim myfilenameFld As WaveLinkWidget
```

Before creating the widgets for this screen, clear widgets from the device and the collection. This step is mandatory when re-using the collection.

```

.
.
.
wlwidcoll.DeleteAllWidgets
wlwidcoll.DeleteWidgets

```

Next, create the first widget. In this example, use the `CreateButton` method of the `WaveLinkFactory` object to create a button widget. The arguments you pass in the `CreateButton` method define the following elements of the widget: the starting left coordinate, the starting top coordinate, the width, the height, the button label, and the name of the collection that will contain the widget.

```
wlfactory.CreateButton 3, 10, 0, 0, " Ok ", wlwidcoll
```

The first two arguments in the `CreateButton` method (and in most other widget creation methods) represent the horizontal and vertical coordinates of the widget on the mobile device screen.

---

**NOTE** You can use the `DefaultCoordinateType` property of the `WaveLinkFactory` object to change how the application interprets values passed for the vertical and horizontal coordinates, and for the height and width.

---

The third and fourth argument in the `CreateButton` method determine the width and the height of the widget. Many widgets allow you to pass values of zero (0) for the width and height to automatically size, or *autosize*, the widget. The specific widget type determines how the application handles autosizing values. For example, when you autosize a button widget, the application sizes the widget according to the size of the text that it contains. See the *Wavelink Development Library* documentation for information specific to each widget type.

Create a second button widget. Add this widget to the same collection in order to manipulate the widgets as a group.

```
wlfactory.CreateButton 9, 10, 0, 0, " Cancel ", wlwidcoll
```

Next, create a field widget to obtain the user's name. In this example, use the `RFPrint` method to tell the user what to do.

```

wllo.RFPrint 3, 2, "Enter your Name:", WLCLEAR
Set mynameFld = myfactory.CreateField(3, 3, 12, 1, "", _

```

```
wlwidcoll)
```

Then create a second field widget to obtain the desired file name for the signature.

```
wlio.RFPrint 3, 6, "Save file as:", WLNORMAL  
Set myfilenameFld = myfactory.CreateField(3, 7, 12, 1, "", _  
    wlwidcoll)
```

As mentioned previously, you will use the WaveLinkWidgetCollection object to store and display a group of widgets. You can assign any widget to a collection, but typically you will put widgets in the same group to cause them to appear simultaneously on the mobile device screen. To store and display the widgets in the current collection, use the StoreWidgets method of the WaveLinkWidgetCollection object.

```
myCollection.StoreWidgets
```

The API sends data to the device when the output buffer reaches 100 bytes or when the application makes an input call. See [Handling Events](#) on page 48 for more information about making the input call.

Figure 5-1 shows how this screen appears on a mobile device running Palm OS.



**Figure 5-1.** *Widget Example Screen*

When you are finished with a group of widgets, you can clear them off the screen of the mobile device with the WaveLinkWidgetCollection object.

```
wlwidcoll.Clear
```

The methodology for creating widgets varies somewhat depending on the specific widget type, as follows:

- For buttons, bitmaps, checkboxes, fields, hotspots, labels, repeater buttons, and selector triggers, the preceding example shows the correct methodology. See the *Wavelink Development Library* documentation for more information about creating specific widget types.
- For widgets based on dialog boxes, such as the scribblepad and signon widgets, see [Using Widget-based Dialog Boxes](#) on page 49 for additional information.
- For widgets based on menus, such as menubars, popup triggers, and push buttons, see [Using Menu-based Widgets](#) on page 50 for additional information.

## Handling Events

You can use either the `GetEvent` or `RFInput` method to return widget-based input from the mobile device. `GetEvent` returns input from the application after a single input event.

The [example](#) in the previous section used a button widget and a field widget. For a button widget, an input event occurs when the user clicks the button. For a field widget, an input event occurs when the user exits the field. Because you want to process input from multiple widgets (i.e., the widget fields and one of the two buttons), you must create a loop structure containing the `GetEvent` call.

When returning input from a button widget, `GetEvent` returns the button label text. This allows you to easily process input from a button widget using its label text, in this case either "Ok" or "Cancel."

In the case of field widgets, `GetEvent` returns the contents of the field.

In addition to returning information specific to a widget (such as the field contents), `RFInput` and `GetEvent` also return a unique widget identifier (ID), which you can access using the `LastExtendedType` method. You can compare the widget ID returned by `LastExtendedType` to the ID of specific widgets, and process accordingly if the IDs match up.

---

**NOTE** The API creates the widget ID automatically when you first create the widget. You can access the ID of the widget you created using the `WidgetID` property.

---

```
Public wlio As New RFIO
Dim fileName, userName As String           ' User input
variables
Dim result As String                       ' User input variables
Dim done As Boolean                        ' Loop variable
.
.
.

' Initialize loop variable
done = False

While done = False
    result = wlio.GetEvent
    ' Add error handling code.
    If mynameFld.WidgetID = wlio.LastExtendedType Then
        userName = result
    ElseIf myfilenameFld.WidgetID = wlio.LastExtendedType Then
        fileName = result
    ElseIf result = "Ok" Then
        done = True
    ElseIf result = "Cancel" Then
        Goto ExitApp
    Else
        Goto ErrorHandler
    EndIf
Wend
```

## Using Widget-based Dialog Boxes

Wavelink supports a pre-defined dialog box for signature capture. This section continues the [example](#) in the preceding section, and shows the portion of the application that displays the signature capture dialog box. The `WaveLinkScribblePad` object defines this dialog box.

```

Dim wlscribble As New WaveLinkScribblePad
Dim myWidget As WaveLinkWidget
.
.
.
' Obtain the user name and store it in userName.
' Obtain the new file name in store it in fileName.

```

In this example, you set the title for the scribble pad dialog box to the name of the current user using the `DisplayText` property of the `WaveLinkWidget` object.

```

Set myWidget = wlscribble.Title
myWidget.DisplayText = userName

```

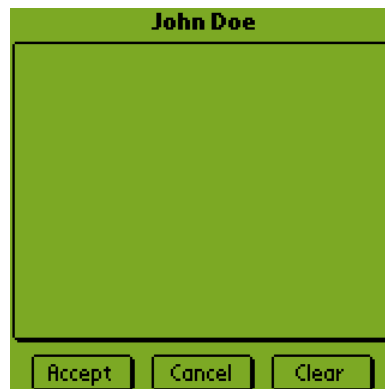
Next, use the `DisplayDialog` method of the `WaveLinkScribblePad` object to display the dialog box and return the signature or image. Here, you pass the name of the file as an argument to save the signature as `"C:\filename.jpg"`.

```

wlscribble.DisplayDialog "C:\" + fileName + ".jpg"

```

Figure 5-2 shows how this dialog box will appear on the mobile device screen.



**Figure 5-2.** *The Scribble Pad Dialog Box*

See the *WaveLink Development Library* documentation for more information about using this object.

## Using Menu-based Widgets

Menu-based widgets include popup triggers, push buttons, and menubars. These widgets use the `RFMenu` object to populate the widget with options,

which alters the [widget creation process](#) for these widgets. The following example shows how to create a popup trigger widget.

---

**NOTE** For more information about creating menu configurations, see [Using Menus](#) on page 40.

---

Before you create the popup trigger, you must build the menu configuration you intend to use.

```
Dim wlfactory As New WaveLinkFactory
Dim wlwidcoll As New WaveLinkWidgetCollection
Dim wlmenu As New RFMenu
Dim welcomeMenu As WaveLinkWidget
.
.
.
wlmenu.ResetMenu
wlmenu.SetMenuWidth 18
wlmenu.AddOption "Basic Car Wash"
wlmenu.AddOption "Basic Wash/Wax"
wlmenu.AddOption "Carnuba Wax"
wlmenu.AddOption "Interior Clean"
wlmenu.AddOption "Full Detail"
wlmenu.AddOption "Special Detail"
wlmenu.AddOption "Device Version"
wlmenu.AddOption "Exit"
wlmenu.StoreMenu "MainMenu"
.
.
.
```

Next, use the `CreatePopupTrigger` method to create the widget, passing in the name of the associated menu configuration. Store the widget as usual.

```
Set welcomeMenu = wlfactory.CreatePopupTrigger(3, 5, 0, 0, _
    "MainMenu", 1, wlwidcoll)

wlwidcoll.StoreWidgets
```

The coding required to create a popup trigger and a push button widget involve the same basic steps. See the *Wavelink Development Library* documentation for more information.

Menubar widgets, unlike popup triggers and push buttons, can contain multiple menus. To handle this additional functionality, the menubar widget

uses the WaveLinkMenubarInfo object. The WaveLinkMenubarInfo object contains the methods necessary to create and manipulate a set of menus. The following example shows how to create a menubar widget.

```
Public wlmenuinfo As New WaveLinkMenubarInfo
Public wlmenu As New RFMenu
Public wlwidcoll As New WaveLinkWidgetCollection
Public wlfactory As New WaveLinkFactory
Dim myWidget As WaveLinkWidget

.
.
.
    ' Create the menus that will appear in the main menu bar
    wlmenu.ResetMenu
    wlmenu.AddTitleLine "Widget Demo's"
    wlmenu.AddOption "Buttons"
    wlmenu.AddOption "Pen Capture"
    wlmenu.StoreMenu "MenOne"

    wlmenu.ResetMenu
    wlmenu.AddTitleLine "Exit"
    wlmenu.AddOption "Quit"
    wlmenu.StoreMenu "MenTwo"
```

Insert the two menus into the WaveLinkMenubarInfo object using the Insert method. The Insert method adds a menu configuration to the WaveLinkMenubarInfo object at a specified index. By passing an argument of -1 as the index value, you automatically tack the named menu to the end of the list. In this example, MenOne will appear as the first menu in the menubar.

```
    wlmenuinfo.Insert -1, "MenOne"
    wlmenuinfo.Insert -1, "MenTwo"

.
.
.
```

Then use the CreateMenubar method to create the menubar. The menubar always displays at the top of the mobile device screen, so the only arguments you need to pass into this method are the name of the WaveLinkMenubarInfo object you want to associate with the menubar, and the name of the widget collection.

```
    ' Create the menubar
    Set myWidget = myFactory.CreateMenubar myMenInfo,
myCollection
```



```
wlwidcoll.StoreWidgets
```

You can process input from menu-based widgets as you would a standard RFMenu object. See [Using Menus](#) on page 40 for more information. Because menubar widgets contain multiple menus, you must also first determine the menu from which the user selected the option before you can process the selected option itself. For example, in the previous code, you inserted the "MenOne" menu into the WaveLinkMenubarInfo object at the first index position (0 index). You can use the LastExtendedType method to return the index value of the menubar.

## Widget Transactions

The WaveLinkWidget object provides numerous functions for manipulating widgets before and after you display them on the screen of the mobile device. This section describes a few of these functions:

- [Positioning Widgets](#)
- [Hiding and Disabling Widgets](#)
- [Setting the Focus](#)

### Positioning Widgets

The first two arguments you pass in most of the widget creation methods represent the horizontal and vertical coordinates of the widget on the mobile device screen. The values you pass for these coordinates must be one of three coordinate types:

BYCELL	Position the widget using cell coordinates (rows and columns), measured from the left edge (column 0) or top edge (row 0) of the screen.
BYPIXEL	Position the widget using pixel coordinates, measured from the left or top edge of the screen.
BYPERCENTAGE	Position the widget using percentage coordinates, measured from the top or left edge of the screen.

You can use the `DefaultCoordinateType` property of the `WaveLinkFactory` object to change how the application interprets these values. The default coordinate type for all widgets is cell coordinates.

For example, in the following code:

```
Dim myCollection As New WaveLinkWidgetCollection
Dim myFactory As New WaveLinkFactory
.
.
.
myFactory.DefaultCoordinateType = BYPIXEL
myFactory.CreateBitmap 20, 20, 35, 25, "logo.bmp", myCollection

myCollection.StoreWidgets
```

you set the default coordinate type to pixel dimensions, and pass values for the horizontal and vertical coordinates in pixels. In this case, you position the bitmap 20 pixels from the top left edge of the screen.

## Hiding and Disabling Widgets

The `StoreWidgets` method of the `WaveLinkWidgetCollection` object stores all widgets in the current collection. By default, widgets display immediately when you use the `StoreWidgets` method. However, you can use the `InitialFlags` property of the `WaveLinkWidget` object to change this. For example, in the following code,

```
Public wlfactory As New WaveLinkFactory
Public wlwidcoll As New WaveLinkWidgetCollection
Dim myButton As WaveLinkWidget
.
.
.
Set myButton = wlfactory.CreateButton (3, 10, 0, 0, " Ok ", _
    wlwidcoll)
myButton.InitialFlags = INITHIDDEN
```

you set the initial state of the button widget to hidden. Hidden widgets do not display when you call the `StoreWidgets` method, but instead appear when you use the `Show` method of the `WaveLinkWidget` object or the `ShowWidgets` method of the `WaveLinkWidgetCollection` object.

```
myButton.Show True
```

You can hide or display any widget using the Show method or any group of widgets using the ShowWidgets method.

In addition to hiding and showing widgets, you can also disable and enable widgets. Disabled widgets appear on the screen of the mobile device, but do not return when you tap them. Typically, disabled widgets appear in gray color or with a crosshatch pattern in place of the label text. You can disable a widget before displaying it on the mobile device using, again, the InitialFlags property.

```
Public wlfactory As New WaveLinkFactory
Public wlwidcoll As New WaveLinkWidgetCollection
Dim myButton As WaveLinkWidget
.
.
.
Set myButton = wlfactory.CreateButton (3, 10, 0, 0, " Ok ", _
                                     wlwidcoll)
myButton.InitialFlags = INITDISABLED
```

To enable the widget, use the Enable method of the WaveLinkWidget object or the EnableWidgets method of the WaveLinkWidgetCollection object.

```
myButton.Enable True
```

## Setting the Focus

When working with field widgets, you can change the input focus from one widget to another. The Focus method of the WaveLinkWidget object provides this functionality. The following example shows how to use the Focus method to process input from the initial screen of the signature capture operation from the earlier example ( See [Handling Events](#) on page 48). The revised code for processing input now looks like this:

```
Public wlio As New RFIO
Dim fileName, userName As String           ' User input
variables
Dim result As String                       ' User input variables
Dim done As Boolean                        ' Loop variable
.
.
.
```

```
' Initialize loop variable
done = False

While done = False
    result = wlio.GetEvent
    ' Add error handling code.
    Select Case result
        Case "Ok"
            done = True
        Case "Cancel"
            GoTo ExitApp
        Case Else
            If mynameFld.WidgetID = wlio.LastExtendedType Then
                userName = result
                myfilenameFld.Focus
            End If
            If myfilenameFld.WidgetID = wlio.LastExtendedType
Then
                fileName = result
            End If
        End Select
    Wend
```

In this revised code, when the user exits the field that prompts the user for a name, the input focus switches automatically to the other field on the screen.

## Chapter 6: Writing Applications for Multiple UIs

With the rise of ubiquitous computing and the multiplication of wireless LANs and WANSs, the coupling of an application's logic and its presentation become an increasing hindrance. The fact that applications often require more than one interface increases the need to separate these two application components. This section of this document presents different options for handling these application types:

- In the first section, [general techniques](#) based on Wavelink objects demonstrate how to obtain critical device information at run-time, allowing you to base the presentation on the current device. These techniques can be used in traditional or object-oriented application designs.
- In the second section, [object-oriented techniques](#) demonstrate how to write applications that de-couple the presentation layer from the logic layer. These techniques also incorporate Wavelink objects to obtain device information and build the actual device-specific UI.

### General Techniques

To write applications for use on mobile devices with varied screen sizes and potentially different operating systems, Wavelink includes methods to return critical information from the currently connected mobile device. The `RFTerminal` object provides this functionality. Several `RFTerminal` functions provide the following information:

- [Information about the display dimensions of the mobile device.](#)
- [Information about the mobile device type.](#)

### Returning the Screen Dimensions

The following `RFTerminal` functions provide information about the display size of the current mobile device:

- **TerminalHeight.** This method returns the height (in characters) of the device display.

- **TerminalWidth.** This method returns the width (in characters) of the device display.

In the following code, the `TerminalHeight` method places output text in the middle of the screen and on the bottom row of the mobile device. This methodology is especially useful when creating applications for devices with different or unknown screen dimensions. In this example, the output text, `Name:`, will appear in the vertical center of the device screen, irrespective of the screen dimensions. The output text, `Ctrl+X = Log Off`, will appear on the final row of the device screen.

---

**NOTE** Because the the first row of the mobile device is 0 rather than 1, you must subtract 1 from the returned height to obtain the last row.

---

```
wlterm As New RFTerminal
wlio As New RFIO
lastLine As Integer
signonLine As Integer
.
.
.
lastLine = termIface.TerminalHeight () - 1
signonLine = lastLine / 2
.
.
.
wlio.RFPrint (0, 0, "    Sign On Screen    ", _
              WLCLEAR | WLREVERSE);
wlio.RFPrint (0, signonLine, " Name:", WLNORMAL);
wlio.RFPrint (1, lastLine, " Ctrl+X = Log Off ", WLREVERSE);
```

## Returning the Device Type

The `RawTerminalType` method returns the device type. Using this method, you can incorporate code into your applications that allows you to handle multiple device types, for example, a Palm OS and a DOS device.

To simplify the coding later, you can assign a set of device types to a variable. In this case, you distinguish only between several GUI-based devices and all other devices. By making this distinction, you can build screens based on whether or not the device supports widgets.

```

wlterm As New RFTerminal
nRFTerminalType As Integer
guiDevice As Boolean
.
.
.
nRFTerminalType = wlterm.RawTerminalType
Select Case nRFTerminalType
    Case PDT1740                                ' a Palm OS device
        guiDevice = true
    Case PDT2740                                ' a Windows CE device
        guiDevice = true
    Case Else                                    ' DOS devices
        guiDevice = false
End Select

```

In this example, you create a portion of the screen based on whether or not the device supports widgets. In this case, if the device supports widgets, create the widgets in the `WidgetAppMenu()` subroutine (not shown). If the device does not support widgets, you build a menu using the `RFMenu` object.

```

wlmenu As New RFMenu
' Add a menu configuration (not shown)
.
.
.
wlio.RFPrint 0, 0, "Wavelink Corporation ", WLCLEAR + WLREVERSE
wlio.RFPrint 0, 2, " Process Options  ", WLNORMAL +
WLFLUSHOUTPUT
If guiDevice = true Then
    WidgetAppMenu
Else
    nProcessRslt = wlmenu.DoMenu("MainMenu")
End If

```

Using this technique, you can create and process screens for any device type as needed. See [Using Widgets](#) on page 45 for more information about designing widget-based screens.

## Object-Oriented Techniques

This section shows how to write applications for multiple UIs using Wavelink objects within an object-oriented framework. The goal of this approach is to create applications that efficiently handle multiple UIs and are easy to debug,

modify, extend, and re-use. To accomplish these goals, the application design is based on the Model-View-Controller paradigm and incorporates a Finite State Machine (FSM).

---

**NOTE** Although the sample code in this section is in C++, the programming techniques also apply to Visual Basic.

---

The sections that follow provide detailed information about how the sample application, Wavelink Program Manager, works. These sections are arranged by topic, as follows:

- *Program Overview*
- *Initializing the Application & Starting the Finite State Machine*
- *Finite State Machine Classes*
- *Presentation View Classes*
- *State Classes*

## **Program Overview**

This Program Manager application allows a user to run a selected application from a list of applications. The program initially prompts the user for a name and password. When the user enters this information, the program validates the input, and if the input is valid, it presents an application menu. When the user selects an application menu option, the Program Manager launches the corresponding application.

This application is based on the following design patterns:

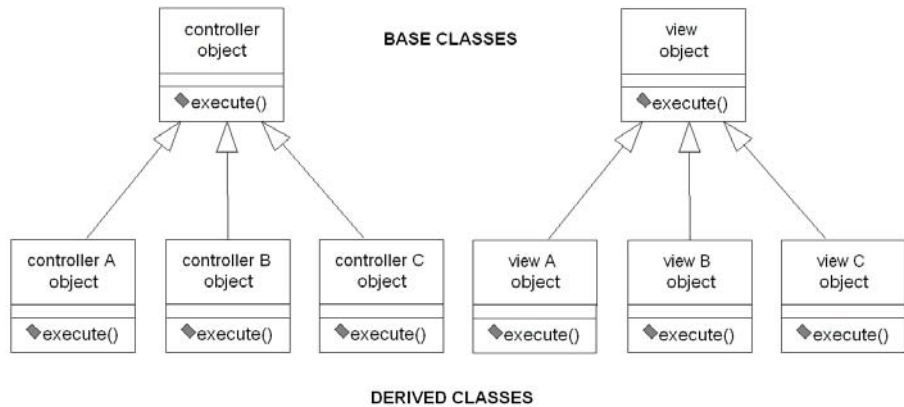
- *Model-View-Controller Paradigm*
- *Finite State Machines*

### **Model-View-Controller Paradigm**

The Program Manager application is based on the Model-View-Controller paradigm. In this paradigm, the following application elements are separated into different specialized objects: the output (or view), the input and program flow (the controller), and the data and data processing (the model).



The distinct roles of the view and controller elements, represented by two distinct classes, are especially critical in the Program Manager application. Because the behavior of these classes (i.e., output behavior and input/program flow behavior) is easy to stylize in most applications, two generic base classes and a set of associated derived classes are used. Figure 6-1 shows the class structure for the view and controller elements.



**Figure 6-1.** *The View and Controller Classes*

The view and controller classes and their derived classes use polymorphism. In polymorphism, the base class provides generic functions (a generic interface) that must be common to all derived classes. In this way, any of the derived classes can be invoked at run-time, because each such class contains all the essential methods.

For example, in the Program Manager, several derived view classes represent pen-based devices and several others represent all other devices. If the application initially detects a pen-based device at run-time, it loads all the pen-based views. Consequently, when the application invokes a generic "execute view" method, the execute method of the corresponding pen-based view will be invoked.

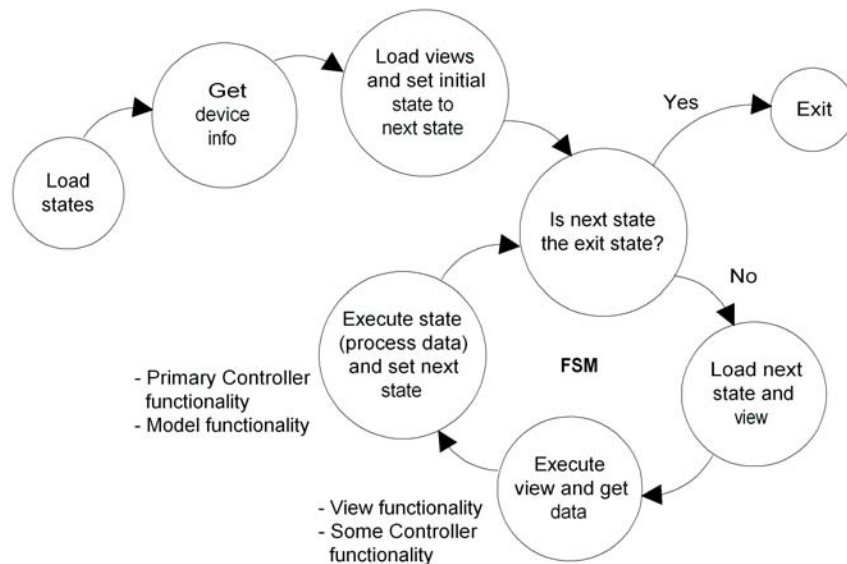
Using the Model-View-Controller paradigm, the controller classes contain application logic while the view classes contain the UI. This de-coupling of the logic layer from the presentation layer reduces the debug and coding cycles and promotes re-use. For example, you can easily add or subtract views for different device types without affecting the application logic. Furthermore, you can easily extend the application by adding new controller sub-classes along with corresponding views.

## Finite State Machines

To implement the application's [Model-View-Controller paradigm](#), the Program Manager incorporates a Finite State Machine (FSM). A Finite State Machine consists of a machine object containing a list of states (state objects) that perform the steps of the application. The states contain most of the controller functionality in the Model-View-Controller paradigm.

A machine object starts by executing the first state. The state processes input and returns the identifier of the next state to be executed. The application exits when the exit state identifier is processed.

In the Program Manager, the machine object also executes a presentation view (based on the state) before executing the state itself. Each presentation view represents a specific UI for specific device types (in this case, pen-based devices or all other devices). The diagram shown in Figure 6-2 shows this process in detail.



**Figure 6-2.** *Finite State Machine*

For the Program Manager, a separate state is required to perform each of the following functions: 1) process the user name and password, 2) process the user's menu selection and launch the appropriate application, and 3) perform error handling and exit the application. Three presentation views (one for

each state) are also required for pen-based devices, and a second set of presentation views are needed for other device types.

The advantages of the Finite State Machine are summarized here:

- The generic Finite State Machine functionality can be re-used in multiple applications. The Finite State Machine does not need to know anything about the user interface or the specific states of the application.
- Modification to the application is simplified through adding or deleting state classes.
- The Finite State Machine functionality can exist at multiple levels (for example, a menu machine can execute states that contain their own Finite State Machines).

### Program Source Files

In addition to header files and a C++ file containing the mandatory "main" function, the Wavelink Program Manager includes several source files for the different classes associated with the application. The source files for this application are as follows.

<code>WaveLinkPM.cpp</code>	Initializes COM and includes the C++ "main" function. In this sample program, the code following the "main" function runs the Finite State Machine for the application.
<code>CFSMBase.cpp</code>	This class implements the Finite State Machine. This machine object is generic to allow for reusability.
<code>CWaveLinkPMFSM.cpp</code>	This class, derived from <code>CFSMBase</code> , implements application-specific initialization methods to populate the objects that contain the list of possible states (state objects) and the list of possible presentation views (view objects).
<code>CPresentationList.cpp</code> <code>CStateList.cpp</code>	These container classes hold the presentation views and the states, respectively.

<code>CPresentationBase.cpp</code>	This class is the base class for the presentation view classes.
<code>CSignonDefaultView.cpp</code> <code>CSignonPenView.cpp</code> <code>CMenuDefaultView.cpp</code> <code>CMenuPenView.cpp</code> <code>CErrorDefaultView.cpp</code>	These classes, derived from <code>CPresentationBase</code> , instantiate specific presentation views based on specific device types, in this case, pen-based devices vs. all other devices.
<code>CStateBase.cpp</code>	This class is the base class for the state classes.
<code>CSignonSt8.cpp</code> <code>CMenuSt8.cpp</code> <code>CErrorSt8.cpp</code>	These classes, derived from <code>CStateBase</code> , instantiate the states (state objects) that process user input, and either advance the program to the next state in the Finite State Machine, or loop back without changing the state. For the <code>CErrorSt8</code> class, this class sets a value to terminate the Finite State Machine.

---

**NOTE** This list does not include the header files associated with each source file. See [C++ Source Files](#) on page 75 for more information.

---

## Initializing the Application & Starting the Finite State Machine

The `WaveLinkPM.cpp` source file initializes COM and contains the `main` function (required in C++) where processing begins. In this application, the code associated with the `main` function, shown here, runs the [Finite State Machine](#).

---

**NOTE** See [WaveLinkPM.cpp](#) on page 75 for the complete code in `WaveLinkPM.cpp`.

---

```
// import libraries and include header files
.
.
.
int TerminalType();
```

```
int main(int argc, char* argv[])
{

    if (SUCCEEDED (CoInitialize (NULL))) {
    try {

        CWaveLinkPMFSM currentMachine;
        int termType;

        termType = TerminalType();

        currentMachine.InitializeStateList ();
        currentMachine.InitializeViewList (termType);
        currentMachine.ExecuteMachine (SIGNON);

    }// try
    catch (const _com_error& ce) {
    }// catch (const _com_error& ce)

    CoUninitialize ();
    }// if (SUCCEEDED (CoInitialize (NULL)))

    return 0;
}
```

In the preceding code, the line:

```
CWaveLinkPMFSM currentMachine;
```

instantiates the Finite State Machine object (or machine object), `currentMachine`, as an object of type `CWaveLinkPMFSM`. `CWaveLinkPMFSM` is derived from the Finite State Machine base class, `CFSMBase`, and contains the initialization methods specific to the Program Manager.

You then use the machine object to invoke the `InitializeStateList` method. `InitializeStateList` loads three states (a signon, menu, and error state object) into a container object.

```
currentMachine.InitializeStateList ();
```

---

**NOTE** For detailed information about the `InitializeStateList` method, see [Populating the State List and View List](#) on page 67.

---

You must also load the required views into a second container object. However, before you load the views, you must obtain the current device type. The following code excerpt from `WaveLinkPM.cpp` shows how you can obtain the device type.

First, invoke the `TerminalType` method of `WaveLinkPM.cpp` and store the result in a variable.

```
.
.
.
int TerminalType();
.
.
.
termType = TerminalType();
```

In the `TerminalType()` method, the `ReadTerminalInfo` method of the `RFTerminal` object makes a call to the mobile device to obtain current device information. After calling `ReadTerminalInfo`, invoke the `RawTerminalType` method to return the current device type (for example, `PALM_PILOT` or `CE2740`).

```
// Note: The TerminalType() method is inserted at the end of the
// WaveLinkPM.cpp source file.
.
.
.
//Returns the raw terminal type of the device
int TerminalType()
{
IRFTerminalPtr termIface(__uuidof(RFTERMINAL));

termIface->ReadTerminalInfo ();

return termIface->RawTerminalType();
}
```

After this code executes, the variable `termType` contains the current device type. Based on this information, the machine object calls the `InitializeViewList` method to load three presentation views into a container object. These views correspond to the three states (i.e., a signon view, a menu view, and an error view).

```
currentMachine.InitializeViewList (termType);
```

---

**NOTE** For detailed information about the `InitializeViewList` method, see [Populating the State List and View List](#) on page 67.

---

After initializing the states and presentation views, the following line of code executes the main routine of the Finite State Machine. You pass the initial state identifier, `SIGNON`, as an argument.

```
currentMachine.ExecuteMachine (SIGNON);
```

The implementation of the `ExecuteMachine` method is contained in the base class for the Finite State Machine. See [Finite State Machine Classes](#) for more information.

## Finite State Machine Classes

The Program Manager includes two Finite State Machine classes, `CFSMBase` and `CWaveLinkPMFSM`.

`CFSMBase` contains the main execution routine of the Finite State Machine. As a generic type of Finite State Machine, `CFSMBase` adds re-usability to the application. Re-use can be implemented in other applications or at different levels within a single application (for example, a menu machine can execute states that contain their own Finite State Machines).

Finite State Machine functionality specific to the WaveLink Program Manager application is contained in the derived class, `CWaveLinkPMFSM`. This class contains the application-specific initialization methods; it [populates the state list and presentation view list](#).

---

**NOTE** See [Implementing the Finite State Machine](#) on page 70 for more information about the base class for the Finite State Machine, `CFSMBase`.

---

## Populating the State List and View List

To clarify the program execution during the [initialization phase](#), this section shows what occurs when the Finite State Machine creates the states and presentation views that it requires at run-time.

For the Program Manager, a separate state is needed to perform each of the following functions: 1) process the user name and password, 2) process the user's menu selection and launch the appropriate application, and 3) perform error handling and exit the application. The state classes, CSignonSt8, CMenuSt8, and CErrorSt8 handle these tasks, respectively. At the conclusion of each state's execution, it returns the state identifier corresponding to the next state in the Finite State Machine (for example, CSignonSt8 returns the state identifier, MENU).

The following code fragment shows how CWaveLinkPMFSM.cpp stores the states for the application. A container class, CStateList, is required to hold the states for the application. As shown here, the push\_back method stores the different states in stateList (a container object of type CStateList).

---

**NOTE** CStateList is based on the vector data type. In C++, a vector functions much like an array. For the source code for the CStateList class, see [CStateList.cpp](#) on page 79. For the source code to the CWaveLinkPMFSM class, see [CWaveLinkPMFSM.cpp](#) on page 76.

---

```
// include header files
.
.
.
void CWaveLinkPMFSM::InitializeStateList ()
{
    stateList.push_back (new CSignonSt8(this));
    stateList.push_back (new CErrorSt8());
    stateList.push_back (new CMenuSt8(this));
}
```

In addition to the states, the application also requires presentation views to build the screens on the mobile device depending on the current state. In the Program Manager, the InitializeViewList method stores a set of presentation views in viewList (a container object of type CPresentationList) based on whether or not the device is pen-based.

The InitializeViewList method takes an argument containing the current device type.



```
void CWaveLinkPMFSM::InitializeViewList (int uiType)
```

Using a switch statement, the `InitializeViewList` method compares the device type to a set of constants representing pen-based devices. If the device type matches one of the pen-based constants, the container object stores the three presentation views that are specific to pen-based devices. If no match is found, the container object stores the three generic presentation views instead.

```
void CWaveLinkPMFSM::InitializeViewList (int uiType)
{
    switch(uiType)
    {
        case PALM_PILOT:
        case CE2740:
        case CE7200:
        case CE7540:
        case INTERMEC:
        case WINDOWS:
        case PPC:
        case HPC:
        case HPCPRO:
            viewList.push_back (new CSignonPenView());
            viewList.push_back (new CErrorDefaultView());
            viewList.push_back (new CMenuPenView());
            break;
        default:
            viewList.push_back (new CSignonDefaultView());
            viewList.push_back (new CErrorDefaultView());
            viewList.push_back (new CMenuDefaultView());
            break;
    }
}
```

The generic presentation views will use the RFIO object to build text-based screens on the mobile device, whereas the pen-based screens use Wavelink widgets.

For example, if the user is using a Palm Pilot, the device type, `PALM_PILOT`, will be passed to the `InitializeViewList` method at run-time. As a result, the container object stores the pen-based presentation views corresponding to the different states in the Finite State Machine: `CSignonPenView`, `CMenuPenView`, and `CErrorDefaultView`.

---

**NOTE** Use the `RFTerminal` object to obtain current device information such as the device type. See [Returning the Device Type](#) on page 58 for more information.

---

### Implementing the Finite State Machine

`CFSMBase` is the base class for the Finite State Machine. The `ExecuteMachine` method in this class is the main program loop for the application. `ExecuteMachine` finds and executes the views and states.

---

**NOTE** The Program Manager's `main` function invokes the `ExecuteMachine` method. See [Initializing the Application & Starting the Finite State Machine](#) on page 64 for more information.

---

In the following code fragment,

```
void CFSMBase::ExecuteMachine(int initialState)
{
    int nextState = initialState;
    CStateBase* currentState;
    CPresentationBase* currentView;
    .
    .
    .
    // Add the main application loop here (not yet shown).
    .
    .
    .
}
```

the `ExecuteMachine` method initializes the `nextState` variable with the `initialState` parameter, `SIGNON`, and creates a generic state object, `currentState`:

```
CStateBase* currentState;
```

and a generic presentation view object, `currentView`:

```
CPresentationBase* currentView;
```

---

**NOTE** The Program Manager's `main` function passes the initial state, `SIGNON`, when it invokes the `ExecuteMachine` method.

---

The `ExecuteMachine` method then executes the views and states, which is the main application loop for the program. The application remains in the loop until the next state value equals the exit state value (-1), at which point the application exits. The main application loop obtains a reference to a presentation view based on the current state ID. To obtain the correct reference, it invokes the `ParseViewList` method. Next, it obtains a reference to a state based on the current state ID. To obtain the correct reference, it invokes the `ParseStateList` method.

```
.
.
.
while(nextState != -1)
{
    currentView = ParseViewList (nextState);
    currentState = ParseStateList (nextState);

    // include error checking here (not shown)

    currentView->ExecuteView (currentState);
    nextState = currentState->ExecuteState ();
} //end while(nextState != -1)
```

The `ParseViewList` and the `ParseStateList` methods function in a very similar way. When the machine object calls the `ParseViewList` method, passing it the current state ID, the code iterates through the list of [presentation views](#) and compares their associated state IDs to the current state ID. When it finds a match, it returns a reference to the matching presentation view. Here is the code for the `ParseViewList` method.

```
CPresentationBase* CFSMBase::ParseViewList(int stateID)
{
    CPresentationBase* tempView;

    for(int lcv = 0; lcv < viewList.size (); lcv++)
    {
        tempView = viewList[lcv];
```

```
if(tempView->StateID() == stateID)
return tempView;
}

return NULL;

}
```

For example, when the initial state ID, SIGNON, is passed to the ParseViewList method, the method iterates through the list of presentation views and returns either a reference to the CSignonPenView object (for a pen-based device) or the CSignonDefaultView object (for other devices). Only these two presentation views will return a value of SIGNON when their respective StateID methods are invoked.

---

**NOTE** The ExecuteMachine method and the parsing methods shown here mimic the observer-observable design pattern. For Java applications or complex C++ applications, it is recommended that you incorporate a true observer-observable relationship.

---

In similar fashion, the ParseStateList method obtains a reference to the current state. Initially, this returns a reference to the CSignonSt8 object that will process the user name and password.

The next portion of the main application loop that executes is:

```
currentView->ExecuteView (currentState);
```

In the preceding line of code, `currentView` now contains a reference to the correct presentation view object, and invokes the `ExecuteView` method of that object. The presentation view objects are derived from [CPresentationBase](#) and are based on polymorphism. All the classes derived from `CPresentationBase` implement the `ExecuteView` method, but they implement the method specific to a device type and a state (for example, the signon state). Initially, the `ExecuteView` method for each presentation view displays the signon screen for either a pen-based device (`CSignonPenView`) or all other devices (`CSignonDefaultView`).

During the first iteration of the main application loop, the presentation view prompts the user for a name and password and stores this information, but does not validate the input. The validation functionality belongs to the logic

layer of the application and therefore resides in one of the states (in this case, CSignonSt8). The following line of code invokes the ExecuteState method of the current state object.

```
nextState = currentState->ExecuteState ();
```

Like the presentation views, the states also incorporate polymorphism. CStateBase provides the StateID and ExecuteState methods as the generic interface for the states. The ExecuteState method processes input for each state (for example, it validates the user name and password in CSignonSt8) and returns the state ID of the next state in the Finite State Machine. For example, the ExecuteState method in CSignonSt8 returns the state identifier, MENU, and stores this value in the nextState variable.

## Presentation View Classes

The following code shows the header file for the CPresentationBase, the base class for the presentation view classes. This code shows the required methods, StateID and ExecuteView, that must be included with all derived classes.

---

**NOTE** See [CPresentationBase.cpp](#) on page 80 for the code included in CPresentationBase.cpp.

---

```

////////////////////////////////////
//
//CPresentationBase

#ifndef __CPRESENTATIONBASE_H__
#define __CPRESENTATIONBASE_H__

#include "CStateBase.h"

class CPresentationBase {
public:
    //Default CTOR
    CPresentationBase();

    //Not implemented but required for derived classes
    virtual int StateID() = 0;
    virtual void ExecuteView(CStateBase* currentState) = 0;

    //Member value used to determine whether view should be pushed

```

```

    or restored
    bool stored;
};

#endif

```

The StateID method returns the state identifier for each derived class. The machine object uses this method when parsing the view list to obtain the correct view. The following code shows this code fragment from CSignonPenView.

```

//Return SignonSt8 id
CSignonDefaultView::StateID ()
{
    return SIGNON;
}

```

The ExecuteView method presents the UI associated with each state and returns the required input.

---

**NOTE** The ExecuteView methods use Wavelink objects to build the screen on the device. See [Chapter 4: I/O Techniques](#) on page 13 and [Using Widgets](#) on page 45 for more information about building screens. For information about positioning text on the screen based on the screen dimensions of the current device, see [Returning the Screen Dimensions](#) on page 57.

---

The sample code for the ExecuteView methods in the derived presentation view classes do not include new programming concepts or models. See [C++ Source Files](#) on page 75 for information about the source code for the derived presentation view classes.

## State Classes

The state classes derive from CStateBase and include CSignonSt8, CMenuSt8, and CErrorSt8. Like the presentation view classes, these classes include the StateID method that returns the state identifier. However, the primary method in each state class is ExecuteState; this method processes application data based on the current state. In addition, it returns the identifier of the next state in the Finite State Machine.

For example, the ExecuteState method in CSignonSt8 validates the user name and password, then returns the identifier for the next state, MENU.

As the final state in the Finite State Machine, the CErrorSt8 returns a value that will terminate the program.

The sample code for the classes derived from CStateBase does not include new programming concepts or models. See [C++ Source Files](#) for information about the source code for the derived state classes.

## C++ Source Files

This section includes sample code for the following Program Manager source files:

*WaveLinkPM.cpp*  
*CStateList.cpp*  
*CPresentationList.cpp*  
*CWaveLinkPMFSM.cpp*  
*CFSMBase.cpp*  
*CPresentationBase.cpp*  
*CStateBase.cpp*

---

**NOTE** The sample code included here is updated for the Wavelink Program Manager, shipped with Wavelink Studio COM, which is also compatible with Wavelink Studio 3.6.

---

### WaveLinkPM.cpp

```

////////////////////////////////////
// WaveLinkPM.cpp - Initializes COM and runs the WaveLinkPM
//finite state machine.
//

#import "WaveLink.tlb" no_namespace named_guids
#import "WaveLink.tlb" named_guids
#import "PMAFunctions.tlb" no_namespace named_guids
#include "CWaveLinkPMFSM.h"

int TerminalType();

int main(int argc, char* argv[])
{

if (SUCCEEDED (CoInitialize (NULL))) {
try {

```

```

CWaveLinkPMFSM currentMachine;
int termType;

termType = TerminalType();

currentMachine.InitializeStateList ();
currentMachine.InitializeViewList (termType);
currentMachine.ExecuteMachine (SIGNON);

} // try
catch (const _com_error& ce) {
} // catch (const _com_error& ce)

CoUninitialize ();
} // if (SUCCEEDED (CoInitialize (NULL))

return 0;
}

//Returns the raw terminal type of the device
int TerminalType()
{
IRFTerminalPtr termIface(__uuidof(RFTERMINAL));

termIface->ReadTerminalInfo ();

return termIface->RawTerminalType();

}CWaveLinkPMFSM.cpp

```

### **CWaveLinkPMFSM.cpp**

```

////////////////////////////////////
//
//CWaveLinkPMFSM.cpp - Derived off CFSMBase, this adds methods
that
//populate the stateList and viewList.
//

#include "CSignonSt8.h"
#include "CErrorSt8.h"
#include "CMenuSt8.h"
#include "CSignonDefaultView.h"
#include "CErrorDefaultView.h"
#include "CMenuDefaultView.h"

```



```
#include "../Debug/WaveLink.tlh"
#include "CMenuPenView.h"
#include "CSignonPenView.h"
#include "CWaveLinkPMFSM.h"

//Add the three states to the stateList object.
void CWaveLinkPMFSM::InitializeStateList ()
{
    stateList.push_back (new CSignonSt8(this));
    stateList.push_back (new CErrorSt8());
    stateList.push_back (new CMenuSt8(this));
}

//Add state views based on ui type to the viewList object.
void CWaveLinkPMFSM::InitializeViewList (int uiType)
{
    switch(uiType)
    {
        case PALM_PILOT:
        case CE2740:
        case CE7200:
        case CE7540:
        case INTERMEC:
        case WINDOWS:
        case PPC:
        case HPC:
        case HPCPRO:
            viewList.push_back (new CSignonPenView());
            viewList.push_back (new CErrorDefaultView());
            viewList.push_back (new CMenuPenView());
            break;
        default:
            viewList.push_back (new CSignonDefaultView());
            viewList.push_back (new CErrorDefaultView());
            viewList.push_back (new CMenuDefaultView());
            break;
    }
}

//Cleanup
CWaveLinkPMFSM::~CWaveLinkPMFSM ()
{
    for(int i=0; i < 3; i++)
    {
        delete viewList[i];
        delete stateList[i];
    }
}
```

```

}
}

```

### **CFSMBase.cpp**

```

////////////////////////////////////
//
//CFSMBase.cpp - Implementation of the finite state machine
//base class. ExecuteMachine finds and executes the views and
//states.
//ParseStateList & View returns the correct state ref based on
it's
//ID.
//

#include "CFSMBase.h"

//Empty CTOR
CFSMBase::CFSMBase()
{
}

void CFSMBase::ExecuteMachine(int initialState)
{
    int nextState = initialState;
    CStateBase* currentState;
    CPresentationBase* currentView;

    while(nextState != -1)
    {
        currentView = ParseViewList (nextState);
        currentState = ParseStateList (nextState);

        currentView->ExecuteView (currentState);
        nextState = currentState->ExecuteState ();
    } //end while(nextState != -1)

}

//Return reference to state indicated by id
CStateBase* CFSMBase::ParseStateList (int stateID)
{
    CStateBase* tempState;

    for(int lcv = 0; lcv < stateList.size (); lcv++)

```

```

{
tempState = stateList[lcv];

if(tempState->StateID() == stateID)
return tempState;
}

return NULL;
}

//Return reference to view indicated by id
CPresentationBase* CFSMBase::ParseViewList(int stateID)
{
CPresentationBase* tempView;

for(int lcv = 0; lcv < viewList.size (); lcv++)
{
tempView = viewList[lcv];

if(tempView->StateID() == stateID)
return tempView;
}

return NULL;
}

```

### **CStateList.cpp**

```

////////////////////////////////////
//
//CStateList.cpp - Derived from vector, holds CStateBase refs
//

#include "CStateList.h"

//Default CTOR
CStateList::CStateList()

```

```
{
}
```

### **CPresentationList.cpp**

```
////////////////////////////////////
//
//CPresentationList.cpp - Class is derived from vector and holds
//presentation views.
//
```

```
#include "CPresentationList.h"
```

```
//Default CTOR
CPresentationList::CPresentationList()
{
}
```

### **CPresentationBase.cpp**

```
////////////////////////////////////
/
//CPresentationBase - Base class for presentation views
//
```

```
#include "CPresentationBase.h"
```

```
//CTOR initializes stored variable to false. This variable is
used //to determine whether a screen needs
//to be pushed or restored.
CPresentationBase::CPresentationBase ()
{
stored = false;
}
```

### **CStateBase.cpp**

```
////////////////////////////////////
/
//CStateBase.cpp - Base class for state objects, no
implementation
//
```

```
#include "CStateBase.h"
```

```
//Default CTOR
CStateBase::CStateBase()
```

---

```
{  
}
```



# Index

## A

- AddBarcode method 30
- AddOption method 40
- AddTitleLine method 40
- AddTone method 36
- application navigation 38
- automating the workflow 27
- autosize 46

## B

- bar code configurations 18, 29
  - default 30
- bar codes 27
  - using 31
- BCDISABLED 30
- bitmap widgets 48
- button widget 46
- BYCELL 53
- BYPERCENTAGE 53
- BYPIXEL 53

## C

- checkbox widgets 48
- classes
  - presentation views 73
  - states 74
- Clear method 48
- ClearBarcode method 30
- ClearTones method 36
- COM Development Library
  - see Wavelink Development Library
- COM-based exceptions 11
- conditions for returning input 18
- conventions 2
- CreateButton method 46
- CreateField method 46
- CreateMenubar method 52
- CreatePopupTrigger method 51

- cycle count application 27
  - adding bar code symbologies 29
  - adding tones 36
  - designing 27
  - using bar code symbologies 31

## D

- DefaultCoordinateType property 54
- DeleteAllWidgets method 46
- DeleteBarcodeFile 35
- DeleteMenu method 42
- DeleteWidgets method 46
- dialog boxes 49
- disabled widgets 54
- Display method 42
- display sizes 57
- DisplayDialog method 50
- displaying data 15
- DisplayText property 50
- document conventions 2
- DoMenu method 41

## E

- Enable method 55
- EnableWidgets method 55
- error handling 11
- exceptions 11

## F

- field widget 46
- file extensions 14
- Finite State Machine 62
  - classes 67
  - implementing 70
  - source files 76, 78
- flushing output 24
- Focus method 55
- function keys 38

**G**

GetEvent 25  
GetEvent method 48  
GUI devices 57

**H**

handling application errors 11  
hidden widgets 54  
high speed display 23  
    storing a screen 24  
    storing screen templates 25  
hotspot widgets 48

**I**

I/O techniques 13  
    handling out-of-range devices 13  
    optimizing RF traffic 14  
InitialFlags property 54  
input  
    changing return conditions 21  
    controlling content 20  
    controlling input type 20  
    formatting 21  
    setting a timeout 22  
    types of 19  
input modes 20  
input type 19  
Insert method 52

**L**

label widgets 48  
LastBarcodeType method 33  
LastExtendedType method 48, 53  
LastInputType method 19

**M**

menu configurations 40  
menubar widgets 51  
menu-based widgets 50  
message boxes 42

Model-View-Controller paradigm 60  
multiple UIs 57  
    device types 58  
    Finite State Machine 62  
    general programming techniques 57  
    Model-View-Controller paradigm 60  
    object-oriented programming  
        techniques 59  
    screen dimensions 57

**N**

navigation 38

**O**

object-oriented programming 59  
out-of-range devices 13  
output buffer 24  
output modes 15

**P**

PlayTone method 37  
popup trigger widgets 50  
Port Monitors 3  
presentation views 67, 73  
processing returned input 19  
Program Manager 60  
    C++ source files 75  
    container classes 67, 79, 80  
    Finite State Machine classes 67  
    initializing the application 64  
    main program loop 70  
    presentation view classes 73  
    source files 63  
    starting the Finite State Machine 64  
    state classes 74  
PullBarcode 35  
PullScreen 23  
PushScreen 23



**R**

- RawTerminalType method 58
- repeater button widgets 48
- RestoreScreen 23
- RF packets 14
- RF traffic 14
- RFBarcode object 5, 29
- RFDeleteFile method 25
- RFEError object 6, 42
- RFFile object 5, 25
- RFInput method 16
  - input modes 20
  - input timeout 22
  - invoking 16
  - processing input 19
  - returning input 18
- RFIO object 5, 15, 16
- RFMenu object 6, 40, 50
- RFPrint method 15
- RFTerminal object 6, 57, 64
- RFTone object 6, 36

**S**

- scan-based applications 27
- screen dimensions 57
- screen templates 25
- scribble pad 49
- selector trigger widgets 48
- SetInputTimeout method 22
- SetMessageLine method 42
- Show method 54
- ShowWidgets method 54
- signature capture 49
- source files 75
- state list 67
- states 67, 74
- StoreBarcode method 30
- StoreMenu method 41
- StoreTones method 37

- StoreWidgets method 47

- storing a screen 24

- storing files 14

**T**

- Terminal Height method 57

- TerminalWidth method 58

- tone configurations 36

**V**

- view list 67

- views

- see presentation views

**W**

- Wavelink Client 4

- Wavelink Development Library 5

- referencing the COM library 9

- Wavelink Program Manager

- see Program Manager

- Wavelink Server 3

- Wavelink Studio 3

- WaveLinkFactory object 6, 45

- WaveLinkMenubarInfo object 6, 52

- WaveLinkScribblePad object 6, 49

- WaveLinkSignon object 6

- WaveLinkWidget object 6, 45, 53, 54, 55

- WaveLinkWidgetCollection object 7, 45

- widget objects 6

- WidgetID method 49

- widgets 45

- dialog-boxes 49

- handling events 48

- hiding and disabling 54

- menu-based 50

- positioning 53

- setting the focus 55

- transactions 53

- unique identifiers 49

- using 45

WLALPHA\_ONLY 21  
WLBACKLIGHT 22  
WLCLEAR 15  
WLCLR\_INPUT\_BUFFER 22, 43  
WLCLREOLN 35  
WLCOMMANDTYPE 19, 38  
WLDISABLE\_FKEYS 20  
WLDISABLE\_KEY 20  
WLDISABLE\_SCAN 20  
WLECHO\_ASTERISK 21  
WLFLUSHOUTPUT 24  
WLFORCE\_ENTRY 21  
WLIGNORE\_CRLF 22  
WLKEYTYPE 19  
WLNO\_RETURN\_BKSP 21  
WLNO\_RETURN\_FILL 21  
WLNUMERIC\_ONLY 21  
WLREVERSE 15  
WLSCANTYPE 19  
WLSUPPRESS\_ECHO 22  
WLTIMEDOUT 22