

## GUI & Graphics Series — Menus and Messaging with the Propeller Window Manager Framework

*This GUI & Graphics Series tutorial illustrates how to create complex, data-driven GUI applications based on the Propeller Window Manager Framework (WMF). The demos provided illustrate the use of menus, buttons, and event handler development to implement entire GUI applications that control real world devices.*

*This tutorial builds upon the foundations built in the GUI & Graphics Series AN004: Getting Started with VGA and Terminal Output<sup>[1]</sup>. Also in this series, AN005: Simple VGA Menus<sup>[2]</sup> discusses this material in a more introductory manner.*

### Introduction

GUI programming is a large subject that by definition is a “design” process. Unlike fixed algorithms that perform calculations, move servos, or monitor sensors, GUI applications interface with users—humans—and are dynamic, artistic, and creative. That said, GUIs are so complex that massive libraries are developed to help programmers create these very demanding applications in a uniform way.

Event-driven graphical operating systems such as Windows®, OS X™, Android™, and others have tens to hundreds of thousands of lines of GUI code in them. This code draws GUI elements, manages user input, support mice, and or touch screen gestures, supports complex messaging and real-time structures, so user interfaces can run in parallel with application code. These GUI APIs can have hundreds of functions themselves, so learning just to use a GUI API is a steep curve.

Therefore, developing GUI applications on microcontrollers, with limited memory and performance compared to microprocessors, is challenging to say the least. There are lots of planning and design tradeoffs to consider before starting any programming. For example, will the GUI be event driven, hard coded? What data formats will controls be represented with?

Additionally, once the foundation for the GUI controls is written then there is still the matter of “designing” the look and feel of the GUI application itself. This skill can only be obtained from experience, reviewing other GUIs, understanding the target users, trial and error, and common sense.

Taking that into consideration, this application note implements a GUI with an event-driven model much like Windows®, OS X™, and Android™. This is very complex to develop, but gives a huge return on investment from the application programmer’s point of view since his (your) workload is reduced markedly. Additionally, as the Propeller Window Manager framework grows, new controls and functionality will fit into this model very easily since the definitions of each GUI control are data driven and not programmatic.

This application note relies on the pre-selected VGA driver `VGA_HiRes_Text_010.spin`. Using this driver saves a lot of time and coding. Again, refer to the application note [AN004<sup>\[1\]</sup>](#) for discussion of this driver.

## Architectural Concepts for the Propeller Window Manager Framework

The Propeller Window Manager framework, referred to as the WMF, is a single Spin object file: `WMF_Framework_010.spin`. The file contains a large API that consists of console terminal text methods, direct rendering methods, drawing methods, keyboard and mouse interfaces, along with a set of controls (menus and buttons). The idea of the framework is to completely insulate the application programmer from the details of GUI programming, and let the user application focus on the design of the GUI and the processing of events as they are generated from the user navigating the GUI itself. The following paragraphs will briefly discuss the architecture of the WMF itself as well as thresh out some conceptual fundamentals relating to the various elements of the system.

WMF isn't a bitmapped GUI, but rather a text-based GUI which can look just as good as long as there are characters available to draw lines, boxes, borders, shadows and so forth. Furthermore, WMF sets the VGA resolution to 800x600 (this can be changed in the VGA driver `VGA_HiRes_Text_010.spin` to a higher or lower setting). This resolution is a good middle ground that works on all VGA monitors and doesn't use too much memory. In this text mode, each character is 8x12 pixels, resulting in a screen that is 100x50 characters.

**Figure 1: Details of a 800x600 Pixel VGA Screen with 8x12 Character Font**

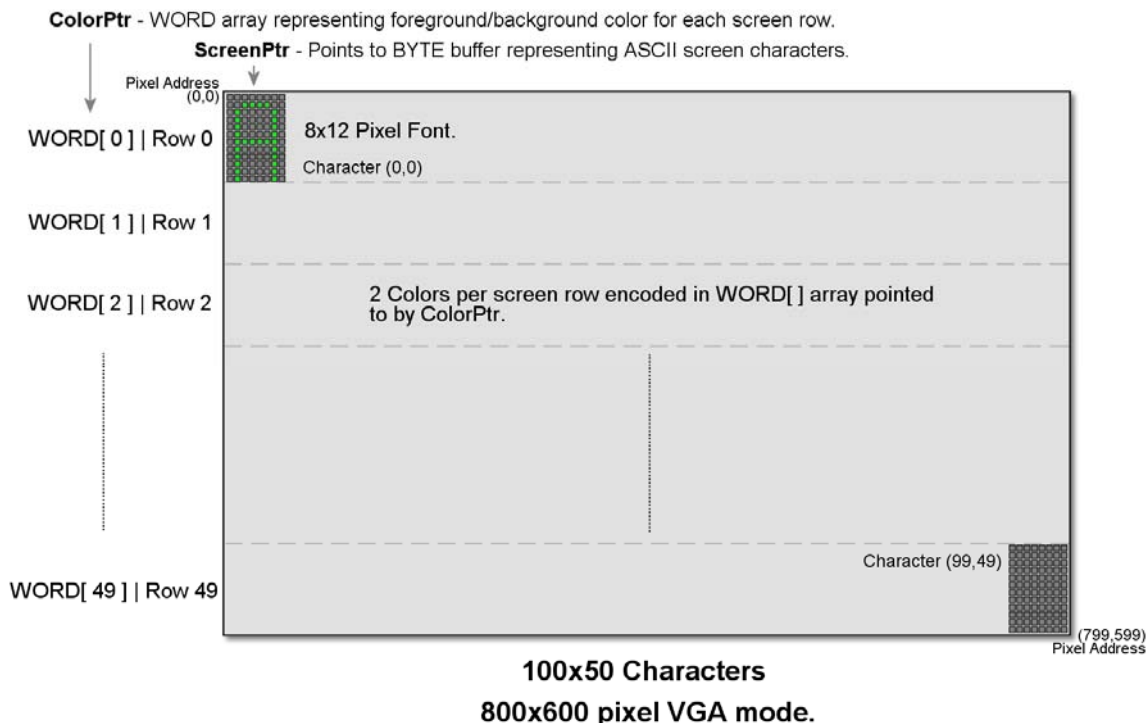


Figure 1 shows the layout of the VGA text screen. Luckily, the WMF insulates the user application from having to deal with the VGA screen mechanics. However, if you're interested please review application note [AN004<sup>\[1\]</sup>](#).

**Figure 2: Software Object Model Relationship**

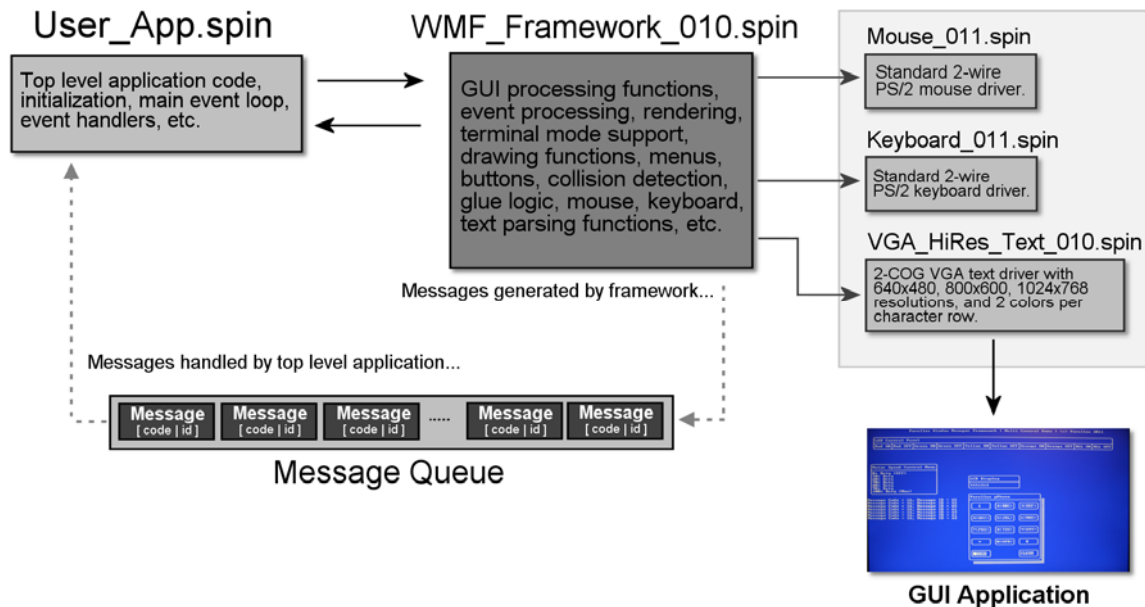


Figure 2 identifies the software objects needed for a complete application. The top-level application is up to you; in this case User\_App.spin is the place holder. The user application must include the WMF framework object WMF\_Framework\_010.spin, which in turn includes the VGA driver VGA\_HiRes\_Text\_010.spin along with the mouse and keyboard drivers Mouse\_011.spin and Keyboard\_011.spin respectively.

Thus, aside from the VGA driver, mouse, and keyboard, everything is encapsulated by the WMF\_Framework\_010.spin object itself—no need to hunt around to multiple modules for API methods, headers, and data structures. As the framework grows it might be prudent to break it up into more objects, but for now a single object is easier to work with.

## Understanding Controls and Events

Now, let's discuss how things work at a high level and drill down to the details. The top-level user application instantiates the WMF object, which in turn instantiates all the other objects, then the system starts running. It's the user application's responsibility to do these two things:

- Create the initial GUI "controls" with local data statements and attach them to the GUI
- Process messages from the WMF's event queue and take the appropriate actions by calling handler methods for various messages

When discussing GUIs and graphical operating systems, the term "control" simply means something on the GUI that the user interacts with. Examples of controls are menus, buttons, windows, text edit boxes, dialog boxes, and so forth. The WMF currently only supports two controls—menus and buttons—but that's enough to create quite a few GUI applications. Of course, static GUI elements like labels, boxes, frames, and so on are also supported with simple API calls to draw them, but those elements are not interactive.

Thus, the controls in a GUI application provide the means through which the user interfaces with the application. These controls have to be created by the user application, and then the windowing system (WMF in this case) renders the controls and processes them as the user interacts with them (via clicking, dragging, hovering, etc.) As the user interacts with

the controls, these interactions are referred to as “events,” hence the term “event-driven” model or programming. The user application sits in a loop and waits for events to occur, which is another way of saying the user application’s job is to wait for events and then “handle” them. This handling is where the rubber meets the concrete.

The “event handlers” in the user application actually do something, but have nothing to do with the GUI code. At the end of the day all the GUI code and WMF do is draw images on the screen, allow the user to interact with these images, and then finally send messages to the user application that indicate the type of event(s) that have occurred. The user application can then take action finally, via the event handlers. Examples might be to turn on a motor, play a sound, switch on a relay, and so forth. The irony is that the final application code might be as simple as turning on/off a set of 8 LEDs. This code might look like:

```
OUTA[ port_bit ] := 1
```

However, the GUI code that allows the user to interact with graphics on the screen by clicking a button or menu to turn on/off the LED might be hundreds or even thousands of lines of code!

**Figure 3: Data Flow Model of WMF’s Event-driven System**

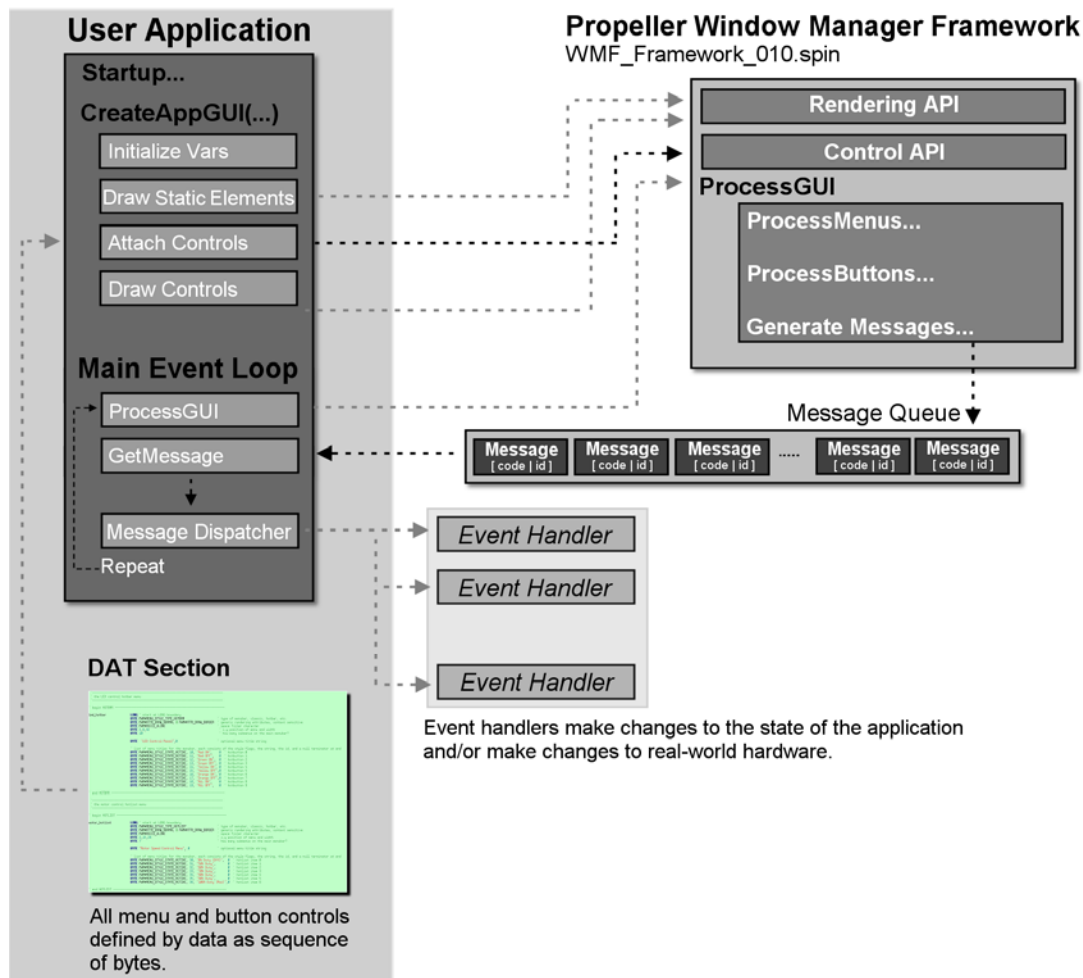


Figure 3 illustrates a more detailed data flow model of what we have been discussing. The user application creates a number of controls by declaring data structures for menus, buttons, and so forth. This is where the “data driven” part of the model comes from. Rather than using method calls to create controls, the user application uses data statements (text, strings, XML, etc.) to define controls. In the case of WMF, simple DAT statements are used. As an example, the listing below illustrates the declaration for one of the menus supported by WMF, the hotlist:

```
DAT
motor_hotlist LONG ' start at LONG boundary
                BYTE WMF#MENU_STYLE_TYPE_HOTLIST          ' type of menubar, classic, hotbar, etc.
                BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' generic rendering attributes, context
                                                                ' sensitive
                BYTE WMF#ASCII_VLINE                        ' space filler character
                BYTE 2,15,26                                ' x,y position of menu and width
                BYTE 7                                       ' how many submenus on the main menubar?

                BYTE "Motor Speed Control Menu", 0          ' optional menu title string

                ' list of menu items for the menubar, each consists of the style flags,
                ' the string, the id, and a null terminator at end
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 30, "0% Duty (Off)", 0 ' hotlist item 0
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 31, "50% Duty",      0 ' hotlist item 1
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 32, "60% Duty",      0 ' hotlist item 2
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 33, "70% Duty",      0 ' hotlist item 3
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 34, "80% Duty",      0 ' hotlist item 4
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 35, "90% Duty",      0 ' hotlist item 5
                BYTE WMF#MENU_STYLE_STATE_ACTIVE, 36, "100% Duty (Max)",0 ' hotlist item 6
```

These data structures are passed to the WMF which “attaches” the controls to the screen surface and begins drawing them and processing events. The WMF processes events simply by tracking the mouse state and determining if the mouse is over a control (by using collision detection algorithms) and if the user is clicking on the control (by using a state machine that tracks the mouse activity). As the user interacts with the controls, events are generated that are translated into “messages” that in turn get inserted into an event queue (a simple array of longs as shown in the listing below).

```
long gMsgQueue[ NUM_MESSAGES ] ' holds messages
long gMsgQueueHead, gMsgQueueTail ' head and tail index
long gMsgQueueNum               ' number of messages in queue,
                                ' helps to differentiate empty and full cases
```

This message queue is the common link between the user application and WMF framework. The WMF framework is the producer of messages and the user application is the consumer.

The user application’s entire view of the GUI is seen only through the messages it receives in the event queue. These events are the “eyes” and “ears” of the user’s application. If a message arrives indicating a button has been pressed, the user application parses the message to determine which button was pressed based on the control ID, and then calls the proper message/event handler.

This concept is very important, so let’s reiterate:

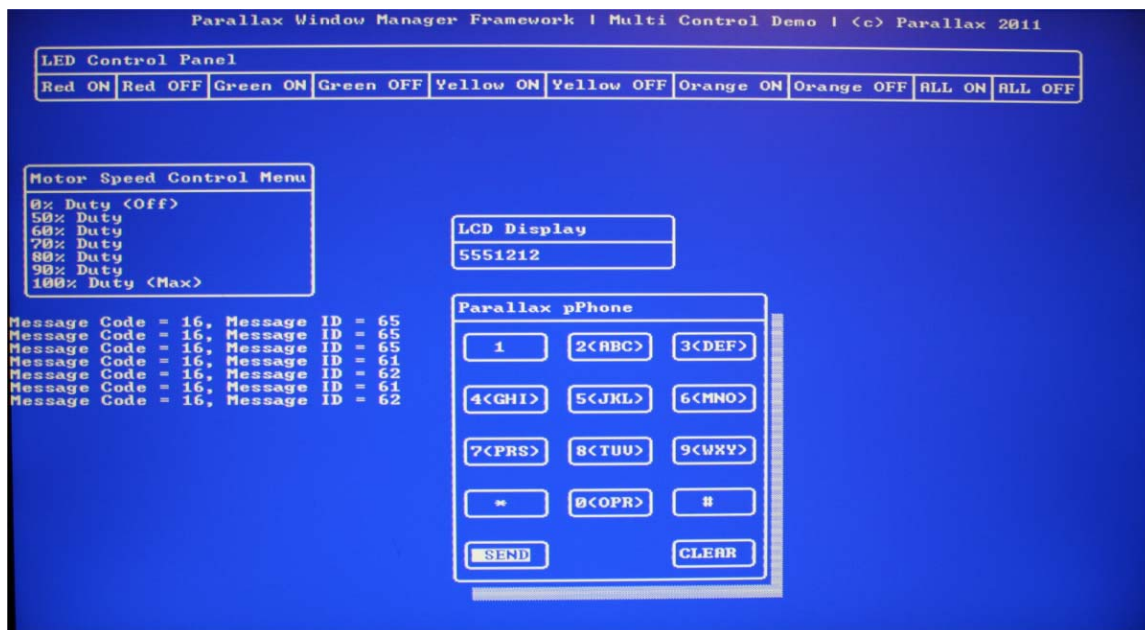
***The user application interacts with the GUI by means of interrogating and processing events/messages in the message queue generated by the GUI code and then takes actions in the real world.***

This process is repeated over and over every cycle. Thus, all the user application does is sit in a loop, call the WMF's main GUI processing method **ProcessGui** each cycle, and then check for messages.

The glue that connects the controls to the GUI are the identifiers (IDs) for each control element. Each control the user application initially creates is assigned an identification number. With this ID, the user code can figure out what control received the interaction, and the WMF tells the user what kind of interaction. With those two pieces of information, the user application can do what it needs to do. Considering all the steps, we have the updated model:

- The user application creates a number of controls (details on this in a moment).
- The WMF attaches the controls to the GUI (VGA screen).
- The user application sits in a tight loop making a call to the WMF's main entry point **ProcessGui**, this gives the GUI thread a cycle to operate, then it returns.
- The WMF processes mouse (and keyboard) actions to determine if the mouse is interacting with a control, and if so inserts messages into the message queue (which can overflow if the user application doesn't process messages often enough).
- The user application checks the message queue each cycle for new messages and if a message of interest is in the queue (a button was clicked, menu item selected, etc.) takes the appropriate action.

**Figure 4: Multi-control Demo illustrates Various Types of Controls and Static GUI Elements**



## Creating Controls

All controls in WMF are data driven rather than programmatically generated. This provides a very natural way to develop a control object with human-readable strings and engineer a GUI element iteratively. In contrast, making method calls to generate controls would be a long and tedious process. Data modeling also makes them much easier to modify since a single pointer can pass a control object to the Propeller Window Manager. Programmers

familiar with XML (eXtensible Markup Language) or with using external resource files will feel at home with this methodology.

There are two classes of GUI elements that the WMF supports: active and static. Static GUI elements are nothing more than window dressing without functionality. The user application simply makes API calls to WMF to draw them on the VGA screen (or intrepid coders can write directly to the VGA buffer).

Referring to Figure 4, static GUI elements are things like the text at the top that displays the demo's name. The LCD display for example uses a static "frame." Frames can be used to contain controls, draw nice boxes, etc. Both of these GUI elements were drawn at the start of the demo itself with these calls (:

```
WMF.PrintString(string("Parallax Window Manager Framework | Multi Control Demo | (c) ←
                                     Parallax 2011"), 16, 1, 0)
.
.
.

' draw frame for fake LCD
WMF.DrawFrame( PX-2, PY-10, 20, 5, string("LCD Display"), 0, gVideoBufferPtr, VGACOLS)
```

Typically, the user application will draw a number of static GUI elements to indicate to the user various areas of interest and display information in a clean manner. Additionally, as the user interacts with the GUI itself, it might update these static elements by erasing them and re-drawing them. For example, an application might need to dynamically display readouts for temperature, pressure, voltage, current, and/or whatever the particular application is controlling and/or monitoring. All these would be drawn and re-drawn with static GUI elements.

The complete listing of the following API methods is in WMF\_Framework\_010.spin. These two methods, **PrintString** and **DrawFrame**, are going to do 90% of the static rendering for you. Here are their complete prototypes:

```
PUB PrintString( pStrPtr, pCol, pRow, pInvFlag )
{{
DESCRIPTION: This method draws a string directly to the frame buffer avoiding the
terminal system.

PARMS:  pStrPtr - Pointer to string to print, null terminated.
        pCol   - Column(x) position to print (0,0) upper left.
        pRow   - Row(y) position to print.
        pInvFlag - Renders character with inverse video colors; i.e background swapped
                  with foreground color.

RETURNS: Nothing.
}}
```

**PrintString** draws directly to the VGA buffer, so be careful not to print out of bounds and step on memory. You can of course print to the VGA screen in "terminal" mode as well, which is much more restrictive. The powerful **DrawFrame** method can be used to create frames, containers, and very elegant displays; its prototype is shown below:

```
PUB DrawFrame( pCol, pRow, pWidth, pHeight, pTitlePtr, pAttr, pVgaPtr, pVgaWidth )
{{
DESCRIPTION: This method draws a rectangular "frame" at pCol, pRow directly to the
```



graphics buffer with size pWidth x pHeight. Also, if pTitlePtr is not null then a title is drawn above the frame in a smaller frame, so it looks nice and clean.

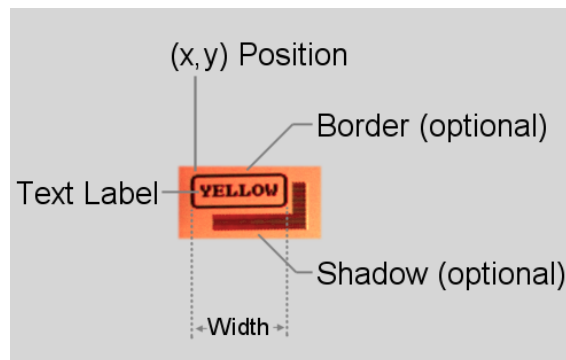
#### PARMS:

pCol - The column to draw the frame at.  
 pRow - The row to draw the frame at.  
 pWidth - The overall width of frame.  
 pHeight - The height of the frame.  
 pTitlePtr - ASCII String to print as title or null for no title.  
 pAttr - Rendering attributes such as shadow, etc. see CON section at top of program for all ATTR\_\* flags.  
 Currently only ATTR\_DRAW\_SHADOW is implemented.  
 pVgaPtr - Pointer to VGA character graphics buffer.  
 pVgaWidth - Width of VGA screen in bytes, same as number of columns in screen; 40, 64, 80, 100, etc.

RETURNS: Nothing.  
 }}

Active GUI elements that the user interacts with, such as menus and buttons, must be declared formally and attached to the GUI itself. It's a two-step process: first declare the control, and then secondly attach it to the GUI so the WMF can start rendering and processing it. Let's take a look at the data declaration formats of the various controls. WMF supports both menus and buttons. Let's begin with the simpler of the two: buttons.

**Figure 5: Features of a Button**



## Creating Button Controls

Buttons are the simplest GUI elements supported in the WMF. They are individual controls not attached to anything else (like menus, which are aggregate controls) and exist separately from other controls. Referring to Figure 5, a button consists of the following 6 fields:

- Field 1:** The type of button (currently there is only a single type).
- Field 2:** The rendering attributes of the button such as borders, shadows, etc.
- Field 3:** The (x, y) location of the button where (0,0) is the upper left hand character position of the screen.
- Field 4:** The state of the button. Usually all buttons are active, so this is a default field.
- Field 5:** A numeric identifier for the button that the WMF uses to refer to the button; you supply this.
- Field 6:** The text label for the button in ASCII format with NULL terminator.



The binary format of the single button type is shown below:

DAT

button\_name BYTE

```

BYTE BUTTON_STYLE_TYPE_STANDARD ' type of button, "STANDARD" supported currently.
BYTE {rendering attribute flags} ' generic rendering attributes, shadows, borders...
BYTE x, y                       ' x,y position of button (width based on string).

' flags, id, and string for the button itself followed by NULL terminator
BYTE {button style flags}, {button id}, "button title string",0

```

The first byte in any button declaration is the type of button. Currently only one type is supported: **BUTTON\_STYLE\_TYPE\_STANDARD** (find this constant and others in the **CON** section of **WMF\_Framework\_010.spin**).

The next byte controls the rendering attributes. This is a bit-encoded, context-sensitive field that "hints" to WMF what you would like visually. WMF then tries to do its best to give you what you want. The rendering attributes are shown below:

```

' general graphical rendering attributes for controls and windows
ATTR_DRAW_SHADOW      = $01
ATTR_DRAW_SOLID       = $02 ' (not implemented)
ATTR_DRAW_DASH        = $04 ' (not implemented)
ATTR_DRAW_BORDER      = $08
ATTR_DRAW_INVERSE     = $10
ATTR_DRAW_NORMAL      = $00

```

For a normal button with no borders or any ornamentation use **ATTR\_DRAW\_NORMAL**. To add a border use **ATTR\_DRAW\_BORDER**. To add a shadow to a border simply logically OR them together like this:

```
ATTR_DRAW_BORDER | ATTR_DRAW_SHADOW
```

The next pair of bytes is the (x,y) position of the control itself. Normally, a byte wouldn't be enough bits to represent a bitmapped control's position, but since this is a text-based GUI the screen is much smaller (100x50 characters at 800x600 pixels is the default).

The next byte in the declaration is called the button style flags. This is a bit-encoded byte that indicates the style and state of the control. The possible bit values are shown below:

```

BUTTON_STYLE_STATE_ACTIVE  = $10 ' if this bit is HIGH then button object is active
BUTTON_STYLE_STATE_GRAYED  = $00 ' if the above bit LOW then button object is inactive or grayed out
BUTTON_STYLE_STATE_DISABLED = $00 ' if the above bit LOW then button object is inactive or grayed out
BUTTON_STYLE_STATE_INACTIVE = $00 ' if the above bit LOW then button object is inactive or grayed out

```

There are only two unique values (\$10 and \$00) right now; *active* and *disabled*. Even though, there are two more constants defined (*grayed* and *inactive*) they use the same value as *disabled*. In the future, these constants may be needed to indicate slightly different functionality, but they are placeholders for now. For now, set all button style flags to **BUTTON\_STYLE\_STATE\_ACTIVE**.

The next byte in the declaration is the identifier of the control, which is very important. Whenever the user interacts with the GUI by clicking the button, focusing on it, losing focus, etc. a message is generated. Part of that message contains this ID number to identify to the user application which control generated the event. Since the user application (written by you) originally assigned this ID number to the button control, both sides know which button is doing what.

Therefore, these IDs typically are unique and create a 1:1 relationship with the control and the ID—but not necessarily. An interface might require two buttons that send the same message, but maybe these buttons are at different locations on the screen. This is perfectly legitimate and using the same ID on two buttons would accomplish this.

It's a good convention to start groups of ID numbers on a round number, in blocks, so it is easy to determine at a glance which set of controls the buttons are related to. One tactic is to start each set of IDs for related controls base 8 or base 10. So, one set of controls might have IDs (8,9,10,11,12...15) then another set (16,17,18...23) and so forth. Or, for base 10 it might have IDs (10,11,12...19) for one set and (50,51,52,...59) for another set. The point is, don't just use random numbers for IDs; use a convention and stick with it. However, remember there are only 255 values available (leaving 0 as a NULL ID—a good idea).

Finally, the last bytes of the declaration are the string itself that should be drawn on the VGA screen to label the button. This can be any size, but keeping them under 16 characters, or 8 preferably, will give the buttons concise labels. The string must be NULL terminated with 0.

As an example of a variety of button declarations, below are three buttons declared from one of the upcoming demos later in the app note (stripped of all comments) to illustrate just how simple the declarations are. This is the power of data-driven modeling. All the work is in the API, so the user application can write these trivial data statements and create controls.

```
button1  LONG
        BYTE WMF#BUTTON_STYLE_TYPE_STANDARD
        BYTE WMF#ATTR_DRAW_NORMAL
        BYTE 2, 6
        BYTE WMF#BUTTON_STYLE_STATE_ACTIVE, 50, "RED",0

button2  LONG
        BYTE WMF#BUTTON_STYLE_TYPE_STANDARD
        BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER
        BYTE 2, 9
        BYTE WMF#BUTTON_STYLE_STATE_ACTIVE, 51, "GREEN",0

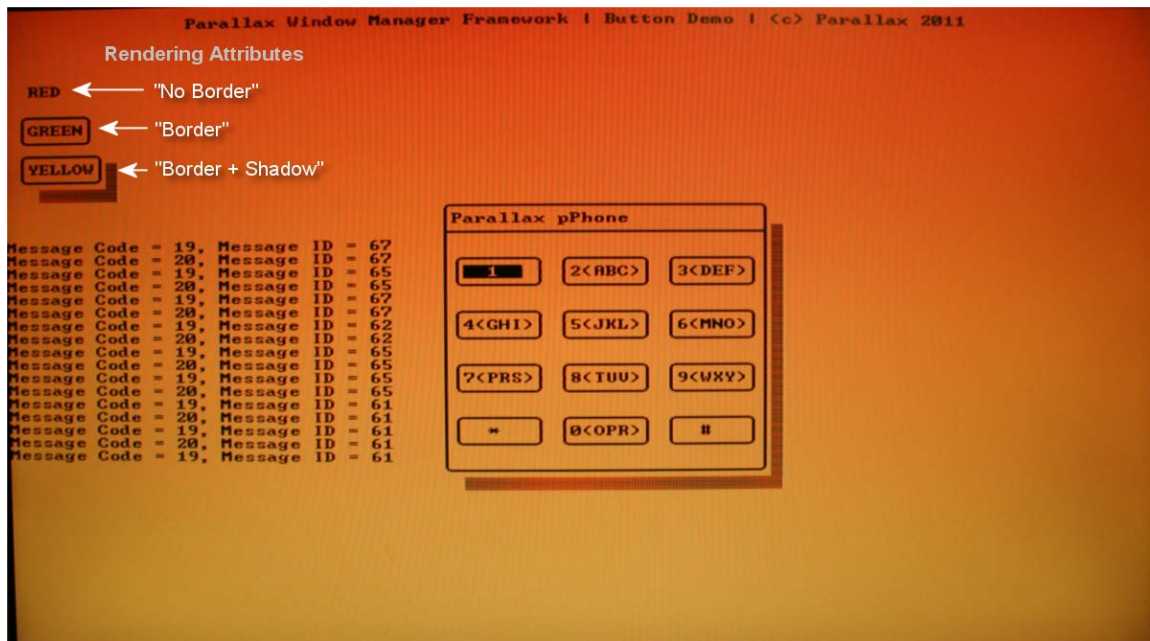
button3  LONG
        BYTE WMF#BUTTON_STYLE_TYPE_STANDARD
        BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER | WMF#ATTR_DRAW_SHADOW
        BYTE 2, 12
        BYTE WMF#BUTTON_STYLE_STATE_ACTIVE, 52, "YELLOW",0
```

Referring to Figure 6 below, these three declarations create a set of buttons with labels "RED", "GREEN" and "YELLOW" along with all useful permutations of attributes; no border, border, and finally border with shadow.

Notice that at the top of each declaration there is a dummy **LONG**. All this does is place the control on a long boundary which might come in handy latter down the line if some of the

fields are increased in size from bytes to words or longs, it will help with pointer arithmetic. At worst it wastes 3 bytes, but if saving memory is critical then omit this **LONG** syntax.

**Figure 6: Results of the Button Data Declarations for button1, 2, 3**



In most cases you will create buttons with borders. However, you might want to create a "stacked" control with the labels right next to each other, with no borders. For example, a number line control that looks like:

012345

...would have 6 buttons, each with no border, placed right next to each other. As the mouse hovers over each number (button) you could click it.

After declaring all the buttons, then the next step is to attach the buttons to the WMF, so it's aware of the controls and can start processing them and sending messages. This is accomplished with a call to the API method **AttachButton** as shown below:

```
' attach buttons
button_id1 := WMF.AttachButton( 0, @button1 )
button_id2 := WMF.AttachButton( 0, @button2 )
button_id3 := WMF.AttachButton( 0, @button3 )
```

We will discuss this step in detail in the section **Attaching Controls to WMF** (page 21). In brief, the API method **AttachButton** is called for each button, and the method returns a resource identifier (not to be confused with the ID of the button). This identifier is more like an index or handle into the global control tables needed when referencing the control—more on this later.

## Creating Menu Controls

Menus are actually very complex controls. Think about all the different possible GUI interactions with a menu—there are a lot of states to consider and a lot of graphics that have to be updated. WMF doesn't try to mimic menus as complex as those Windows or OS

X supports with menu bars, sub-menus, and so forth. WMF's simplified model captures the essence of what most embedded developers need in a menu: simply to list a number of options and then let the user hover over them, select one, and that's that.

With that in mind, refer back to Figure 4 on page 6 for reference. There are two kinds of menus that WMF supports: the "hotbar" and "hotlist." They both send the same kinds of messages and do the same thing, but they just look a bit different. Hotbars are horizontal menus that might go across the top or bottom of an application in a horizontal dominant design. Hotbars can have borders, titles, and shadows. Hotlists on the other hand are more like conventional menus that have a number of menu items listed in a vertical orientation. Hotlists can also have a title, border, and shadow.

Menu controls are a more complex than buttons since they have a number of menu items (or "hot buttons") each which have to be drawn and tracked by the WMF event system. Also, as the mouse hovers over the menu items they need to be highlighted and so forth, so there is a lot more for WMF to do as well, so no one gets off easy with menus.

Like buttons, menus have a style, an (x,y) position, and rendering attributes, as well as a definition for each menu item which is similar to a button in definition. The two menu types are different enough that they warrant separate discussions, so let's start with the simpler hotbar menu.

**Figure 7: Details of a HotBar**



## Declaring HotBar Menus

Referring to Figure 7, a hotbar consists of a number of horizontally organized menu items (or hot buttons) each with its own ID. These menu items may be separated by any ASCII character, in most cases a vertical line or space. Additionally, hotbars have may an optional border and title string. Other than having multiple menu items, hotbars aren't declared much differently from buttons. Here is the format of a hotbar:

```
DAT
hotbar_menu_format  BYTE MENU_STYLE_TYPE_HOTBAR      ' type of menubar, classic, hotbar, etc.
                    BYTE {rendering attribute flags} ' generic rendering attributes, shadows, borders etc.
                    BYTE {ascii code}               ' space filler character
                    BYTE x, y, width                ' x,y position of menu and width
                    BYTE {number of hot buttons}     ' how many hot buttons on the main menubar?
                    BYTE "Menu title string",0      ' optional menu title string

                    ' list of menu hot buttons for the menubar, each consists of the style flags,
                    ' id, the string, and null terminator
                    BYTE {hot button 0 style flags}, {hot button 0 id}, "hot button 0 title string",0
                    BYTE {hot button 1 style flags}, {hot button 1 id}, "hot button 1 title string",0
                    .
                    .
                    .
                    BYTE {hot button n style flags}, {hot button n id}, "hot button n title string",0
```

The header portion of a hotbar consists of 7 bytes. These bytes define the type of hotbar, rendering attributes, the item separator character, the (x,y) position, and overall width of the hotbar. Finally a single byte indicates how many menu items are on the hotbar. This is followed by an optional title string for the menu and the list of menu items themselves. Each menu item itself consists of a style flag byte, ID number, text label string, and NULL terminator. There is no limit to how many menu items a hotbar can have other than what will fit on the screen.

Let's begin with the first header byte which indicates the type or style of menu. This should be set to **MENU\_STYLE\_TYPE\_HOTBAR**. (The other type of menu **MENU\_STYLE\_TYPE\_HOTLIST** is discussed in the following section). The next field is the standard rendering attributes which is a bitwise logical OR of the same flags available to buttons:

```
' general graphical rendering attributes for controls and windows
ATTR_DRAW_SHADOW      = $01
ATTR_DRAW_SOLID       = $02 ' (not implemented)
ATTR_DRAW_DASH        = $04 ' (not implemented)
ATTR_DRAW_BORDER      = $08
ATTR_DRAW_INVERSE     = $10
ATTR_DRAW_NORMAL      = $00
```

Like the buttons, the same style options are available for hotbars. For a hotbar without borders or any ornamentation at all, use **ATTR\_DRAW\_NORMAL**. For a border alone use **ATTR\_DRAW\_BORDER**. For a border and shadow simply logically OR them together like this:

```
ATTR_DRAW_BORDER | ATTR_DRAW_SHADOW
```

Next is the (x,y) position on the VGA screen, nothing new there. However, the next byte *width* needs a little explanation. This data field provides more control of the overall look of the menu bar, to make it larger than it needs to be, for example. In most cases, set this to the size of the sum of the lengths of the menu item strings, taking into consideration a single space between each menu item and a final space at the end.

The formula to compute *width* is:

$$\text{width} = \text{Sum of the length of each menu item string} + \text{number of menu items} + 1$$

For example, given two menu items "On" and "Off" *width* would be calculated as:

$$\text{width} = (2 + 3) + 2 + 1 = 8$$

Obviously, the WMF could calculate this, and maybe it should. However, if you make the width larger than it needs to be, or smaller, you can create custom controls that don't look like normal hotbars and this might be your intent or need.

After this width byte is where the action starts, and the list of menu items starts. The first byte is the number of menu items in the list. This is followed by an optional NULL-terminated menu title string. Then, each menu item consists of a style flags byte, ID, string, and NULL terminator as shown below:

```
BYTE {hot button 1 style flags}, {hot button 1 id}, "hot button 1 title string",0
```

The potential style flags are the exact same as those for buttons, but with different names:

```
' this style is either, so only 1 bit needed to encode, all objects are active OR they are
' grayed/disabled/inactive
MENU_STYLE_STATE_ACTIVE   = $10 ' if this bit is HIGH then the menu object is active
MENU_STYLE_STATE_GRAYED   = $00 ' if the above bit LOW then the menu object is inactive or grayed out
MENU_STYLE_STATE_DISABLED = $00 ' if the above bit LOW then the menu object is inactive or grayed out
MENU_STYLE_STATE_INACTIVE = $00 ' if the above bit LOW then the menu object is inactive or grayed out
```

As with button controls, always set the style flag to **MENU\_STYLE\_STATE\_ACTIVE**. Later versions of the WMF will support these flags more actively and draw menu items grayed out etc. But for now, due to graphic limitations, always set this to active.

The next byte in the declaration of a menu item button is the ID number [0..255] that the WMF sends as part of the event message when this control is clicked, so the user application can figure out which control received the interaction. Once again, limit IDs to [1..255] for values and use ID numbers that are in groups, especially on the same hotbar.

Finally, there is the ASCII string itself that is drawn as the label, and a NULL terminator.

That's all there is to defining each hotbar menu item. However, since hotbars are horizontal entities with a collection of two or more menu item buttons drawn adjacent to each other, they require a single character between them. It's up to you which space-filler character to use.

If you enable the border attribute, set the third header byte (space filler character) to a vertical line for a structured menu bar. On the other hand, for a more free-floating menu bar with no border or ornamentation, an ASCII space character might be a better option.

WMF comes complete with a number of ASCII constants such as **ASCII\_VLINE** to import into the user app with the Spin syntax: *object\_name#constant\_identifier*. For example, call the WMF object "WMF" then import any constant from WMF with syntax similar to:

```
WMF#constant_identifier
```

Thus, in this case **WMF#ASCII\_VLINE** would do the trick. WMF has a number of useful ASCII constants in the **CON** section as shown below:

```
' box drawing characters
ASCII_HLINE = 14 ' horizontal line character
ASCII_VLINE = 15 ' vertical line character

ASCII_TOPLT = 10 ' top left corner character
ASCII_TOPRT = 11 ' top right corner character

ASCII_TOPT  = 16 ' top "t" character
ASCII_BOTT  = 17 ' bottom "t" character

ASCII_LTT   = 18 ' left "t" character
ASCII_RTT   = 19 ' right "t" character

ASCII_BOTLT = 12 ' bottom left character
ASCII_BOTRT = 13 ' bottom right character

ASCII_DITHER = 24 ' dithered pattern for shadows

NULL          = 0 ' NULL pointer
```

Additionally, a number of keyboard characters and punctuation are declared, so take a look in the **CON** section of `WMF_Framework_010.spin` to save remembering ASCII constants in the user application.

Below is an excerpt from one of the upcoming hotbar demos that shows the hotbar declarations in the **DAT** section of the demo (with some comments removed):

```
' begin HOTBAR -----
demo_hotbar_menu0 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTBAR           ' type of menubar
                    BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes
                    BYTE WMF#ASCII_VLINE                       ' space filler character
                    BYTE 4,7,60                                 ' x,y position of menu and width
                    BYTE 5                                      ' number of menu items on hotbar?
                    BYTE "User Profile Tabs",0                 ' optional menu title string

                    ' list of menu titles for the menubar, each consists of the style flags, id, the
                    ' string, and a null terminator at end
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 10, "My Profile", 0 ' hotbutton 0
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 11, "Update Email", 0 ' hotbutton 1
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 12, "Update Address", 0 ' hotbutton 2
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 13, "Reset Password", 0 ' hotbutton 3
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 14, "Mail", 0 ' hotbutton 4

' end HOTBAR -----

' begin HOTBAR -----
demo_hotbar_menu1 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTBAR           ' type of menubar
                    BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER | WMF#ATTR_DRAW_SHADOW ' attributes
                    BYTE WMF#ASCII_VLINE                       ' space filler character
                    BYTE 74,47,24                               ' x,y position of menu and width
                    BYTE 3                                      ' number of menu items hotbar?
                    BYTE "Language Setting", 0                  ' optional menu title string

                    ' list of menu titles for the menubar, each consists of the style flags, id, the
                    ' string, and a null terminator at end
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 20, "English", 0 ' hotbutton 0
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 21, "French", 0 ' hotbutton 1
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 22, "Russian", 0 ' hotbutton 2

' end HOTBAR -----

' begin HOTBAR -----
demo_hotbar_menu2 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTBAR           ' type of menubar, classic, hotbar, etc.
                    BYTE WMF#ATTR_DRAW_NORMAL                 ' rendering attributes
                    BYTE WMF#ASCII_SPACE                       ' space filler character
                    BYTE 87,2,10                                ' x,y position of menu and width
                    BYTE 2                                      ' number of menu items hotbar?
                    BYTE 0                                      ' optional menu title string

                    ' list of menu titles for the menubar, each consists of the style flags, id, the
                    ' string, and a null terminator at end
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 31, "HTML", 0 ' hotbutton 0
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 32, "Flash", 0 ' hotbutton 1

' end HOTBAR -----

' begin HOTBAR -----
demo_hotbar_menu3 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTBAR           ' type of menubar
                    BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes
                    BYTE WMF#ASCII_VLINE                       ' space filler character
                    BYTE 0,48,22                                ' x,y position of menu and width
```



```

        BYTE 3                                     ' number of menu items hotbar?
        BYTE 0                                     ' optional menu title string

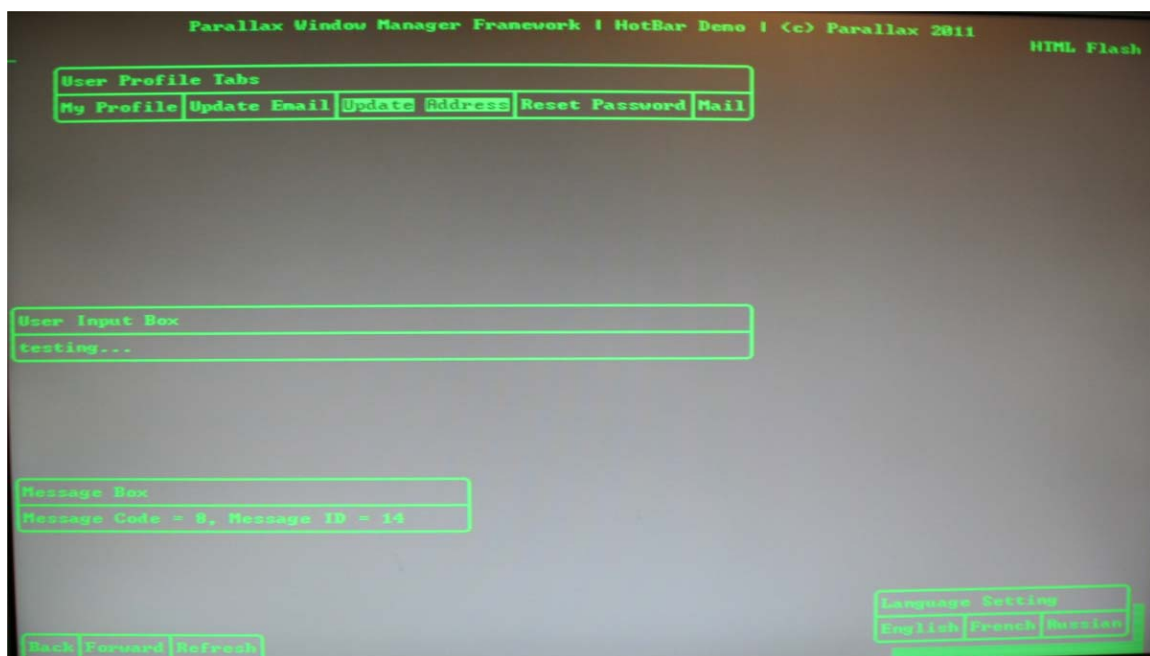
        ' list of menu titles for the menubar, each consists of the style flags, id, the
        ' string, and a null terminator at end
        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 40, "Back",    0 ' hotbutton 0
        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 41, "Forward", 0 ' hotbutton 1
        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 42, "Refresh", 0 ' hotbutton 2

' end HOTBAR -----

```

Although these menu declarations look lengthy, most of it is just copied and pasted with a few changes for the menu items themselves. Figure 8 below is a screen shot of the demo WMF\_HotBarMenuDemo\_010.spin which uses these declarations to create the hotbars.

**Figure 8: Demo Screen Shot of the Results of the Demo HotBar Menu Declarations**



Referring to the figure, the various hotbar declarations show the real power and flexibility of the hotbars. Near the top left of the figure is the "User Profile Tabs." This hotbar menu is more or less the standard way hotbars are intended to be used with borders and a title. The bottom right of the figure, is yet another variant, this time with a shadow, but still with borders and a title bar. Now, where things get interesting is at the bottom left of the figure—the hotbar with "Back," "Forward," and "Refresh." This shows what the hotbar looks like without a title bar. Finally, the top right of the figure illustrates one of the most creative uses of the hotbar and that's to create two free-floating buttons without and borders, title, etc. Here we see the "HTML" and "Flash" menu. Of course, this could have been achieved with two buttons as well, but using a single hotbar menu is more concise. Finally, ignore the "User Input Box" and "Message Box" as these are not menu controls, just static objects drawn for the demo.

After declaring all the hotbars, the next step is to attach them to the WMF so it's aware of the controls and can start processing them and sending messages. This is accomplished much the same way buttons are attached, but with a slightly different API call to **AttachMenu** as shown below:

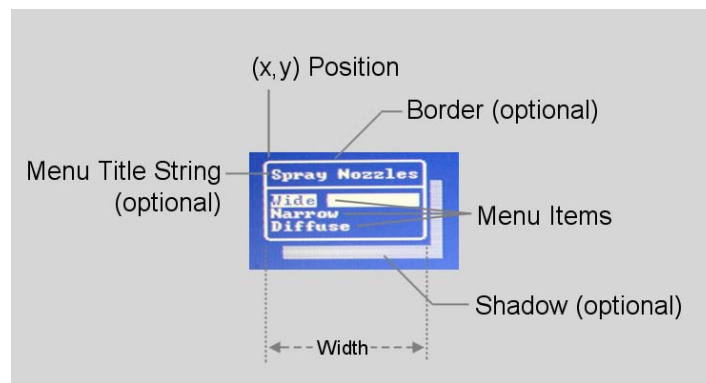
```

' attach menus
demo_hotbar_id0 := WMF.AttachMenu( 0, @demo_hotbar_menu0 )
demo_hotbar_id1 := WMF.AttachMenu( 0, @demo_hotbar_menu1 )
demo_hotbar_id2 := WMF.AttachMenu( 0, @demo_hotbar_menu2 )
demo_hotbar_id3 := WMF.AttachMenu( 0, @demo_hotbar_menu3 )

```

This step is discussed at length in the section below titled Attaching Controls to WMF on page 21. But, in brief, the API method is called for each hotbar, and the method returns a resource identifier (not to be confused with the ID of the button). This identifier is more like an index or handle into the global control tables needed when referencing the hotbar. But more on this later; let's finish up with the last control supported in the WMF—the hotlist.

**Figure 9: Details of a HotList**



## Declaring HotList Menus

Referring to Figure 9, a hotlist consists of a number of vertically organized menu items (or hot buttons), each with its own ID, and each padded to the width of the control itself. Additionally, hotlists have an optional border and title string. Hotlists are nearly identical to hotbars from a definition perspective, but they are simply vertically oriented and more traditional when rendered. In fact, hotlists are simpler than hotbars since there is no need to compute the width, simply set the width equal to or greater than the max of the largest menu item string or title string along with room for the borders (if you select them). Here is the format of a hotlist:

```

DAT
hotlist_menu_format BYTE MENU_STYLE_TYPE_HOTLIST ' type of menubar, hotlist, hotbar, etc.
                    BYTE {rendering attribute flags} ' rendering attributes, shadow, borders, etc.
                    BYTE {ascii code} ' space filler character/unused currently
                    BYTE x, y, width ' x,y position of menu and width

                    BYTE {number of hot buttons} ' how many menu items/hot buttons on the menubar?
                    BYTE "Menu title string",0 ' optional menu title string

                    ' list of menu items for the menubar, each consists of the style flags, id, the
string, ' and the null terminator
                    BYTE {hot button 0 style flags}, {hot button 0 id}, "hot button 0 title string",0
                    BYTE {hot button 1 style flags}, {hot button 1 id}, "hot button 1 title string",0
                    .
                    .
                    .
                    BYTE {hot button n style flags}, {hot button n id}, "hot button n title string",0

```

The header portion of a hotlist consists of 7 bytes and is identical to the hotbar's format, but slightly different in functionality. The first byte is once again the type of menu, in this case **MENU\_STYLE\_TYPE\_HOTLIST**. The next byte defines the hotlist rendering attributes such as borders and shadow. Next is the item separator/space filler character, which is unused currently in hotlists, but maintained to keep binary compatibility with the hotbar. In the future it might be used to fill the white space in the hotlist with something other than the ASCII space " " character. So, for now, make it anything you wish; it is ignored.

To use a hotlist binary declaration and hotbar interchangeably (so the user can click a button and change the layout from horizontal to vertical orientation), use a space filler character like the vertical line, so a menu converted from hotlist to hotbar will still look reasonable with a border.

Moving on, the next data fields are the x,y position and the overall width of the hotlist. Finally, a single byte indicates how many menu items are on the hotlist. This is followed by an optional title string for the menu and the list of menu items themselves. Each menu item itself consists of a style flag byte, ID number, text label string, and NULL terminator. There is no limit to how many menu items a hotlist can have other than what will fit on the screen.

Since hotbars and hotlists are nearly identical, the setup is nearly the same, other than the type byte, and the computation of the width. The hotlist is a vertical version of the hotbar, and vice versa.

With that in mind, the first byte should be set to **MENU\_STYLE\_TYPE\_HOTLIST**. The next field is the standard rendering attributes which is a bitwise logical OR of the same flags available to buttons and hotbars:

```
' general graphical rendering attributes for controls and windows
ATTR_DRAW_SHADOW      = $01
ATTR_DRAW_SOLID       = $02 ' (not implemented)
ATTR_DRAW_DASH        = $04 ' (not implemented)
ATTR_DRAW_BORDER      = $08
ATTR_DRAW_INVERSE     = $10
ATTR_DRAW_NORMAL      = $00
```

Just like the hotbar, the same options are available. For a hotlist without any borders at all or any ornamentation then use **ATTR\_DRAW\_NORMAL**. For a border use **ATTR\_DRAW\_BORDER**. For a border and shadow simply logically OR them together.

Next is the (x,y) position on the VGA screen, nothing new there. The next byte *width* is simplified now from the hotbar. Simply set the width equal to this method:

$$width = \text{MAX}(\text{title string, largest menu item string}) + \text{padding for borders}$$

So, if a title string is 20 characters, but the largest menu item is 5 characters, the width still needs to be at least 20 characters plus space for the borders. Therefore, if you have borders on the control then add a "+2" to the width and that's it. Of course, you can make the menu wider if you like as well.

After the *width* byte is where the list of menu items starts in the exact same format as hotbars. The first byte is the number of menu items in the list. Then each menu item consists of a style flags byte, ID, string, and NULL terminator as shown below:

```
BYTE {hot button 1 style flags}, {hot button 1 id}, "hot button 1 title string",0
```

The potential style flags are the exact same as those for buttons, but with different names as shown below:

```
' this style is either, so only 1 bit needed to encode, all objects are active OR they are
' grayed/disabled/inactive
MENU_STYLE_STATE_ACTIVE   = $10 ' if this bit is HIGH then the menu object is active
MENU_STYLE_STATE_GRAYED   = $00 ' if the above bit LOW then the menu object is inactive or grayed out
MENU_STYLE_STATE_DISABLED = $00 ' if the above bit LOW then the menu object is inactive or grayed out
MENU_STYLE_STATE_INACTIVE = $00 ' if the above bit LOW then the menu object is inactive or grayed out
```

As with hotbar controls, always set the style flag to **MENU\_STYLE\_STATE\_ACTIVE**. Later versions of the WMF will support these flags more actively and draw menu items grayed out etc. But for now, due to graphic limitations, always set this to active.

The next byte in the declaration of a menu item button is the ID number. Again, simply a number [0..255] that the WMF will send as part of the event message when this control is clicked, so the user application can figure out which control was interacted with. Once again, limit ID values to [1..255] and use ID numbers that are in groups, especially on the same hotbar.

Finally, is the ASCII string itself that is drawn as the label and a NULL terminator.

Below is an excerpt from one of the upcoming hotlist demos that shows the hotlist declarations in the **DAT** section (with some comments removed):

```
' begin HOTLIST -----
demo_hotlist_menu0 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTLIST      ' type of menubar.
                    BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes
                    BYTE WMF#ASCII_VLINE                  ' space filler character
                    BYTE 4,5,12                            ' x,y position of menu and width
                    BYTE 2                                  ' number of menu items/hot buttons
                    BYTE 0                                  ' optional menu title string

                    ' list of menu titles for the menubar, each consists of the style flags, id, the
                    ' string, and a null terminator at end
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 10, "Power ON", 0 ' hotbutton 0
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 11, "Power OFF", 0 ' hotbutton 1

' end HOTLIST -----

' begin HOTLIST -----
demo_hotlist_menu1 LONG ' start at LONG boundary
                    BYTE WMF#MENU_STYLE_TYPE_HOTLIST      ' type of menubar
                    BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes
                    BYTE WMF#ASCII_VLINE                  ' space filler character
                    BYTE 20,7,18                           ' x,y position of menu and width
                    BYTE 4                                  ' number of menu items/hot buttons
                    BYTE "Paint Colors", 0                 ' optional menu title string

                    ' list of menu titles for the menubar, each consists of the style flags, id, the
                    ' string, and a null terminator at end
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 20, "Forrest Green", 0 ' hotbutton 0
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 21, "Yellow Gumdrop", 0 ' hotbutton 1
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 22, "Blueberry Chill", 0 ' hotbutton 2
                    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 23, "Titanium White", 0 ' hotbutton 3

' end HOTLIST -----
```

```

' begin HOTLIST -----
demo_hotlist_menu2 LONG ' start at LONG boundary
                        BYTE WMF#MENU_STYLE_TYPE_HOTLIST           ' type of menubar
                        BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER | WMF#ATTR_DRAW_SHADOW ' attributes
                        BYTE WMF#ASCII_VLINE                        ' space filler character
                        BYTE 20,17,15                               ' x,y position of menu and width
                        BYTE 3                                       ' number of menu items/hot buttons
                        BYTE "Spray Nozzles", 0                     ' optional menu title string

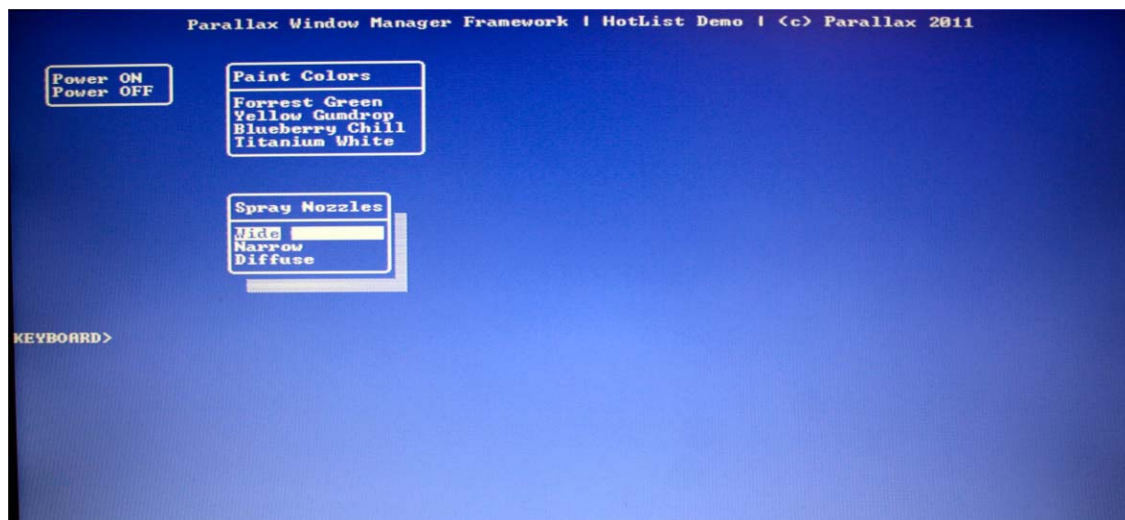
                        ' list of menu titles for the menubar, each consists of the style flags, id, the
                        ' string, and a null terminator at end
                        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 30, "Wide", 0 ' hotbutton 0
                        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 31, "Narrow", 0 ' hotbutton 1
                        BYTE WMF#MENU_STYLE_STATE_ACTIVE, 32, "Diffuse", 0 ' hotbutton 2

' end HOTLIST -----

```

Figure 10 below is a screen shot of WMF\_HotListMenuDemo\_010.spin which uses these declarations to create the hotlists.

**Figure 10: Demo Screen Shot of the Results of the Demo HotList Menu Declarations**



Referring to the figure, the various hotlist declarations show the power and flexibility of the hotlists. From top left moving clockwise there are examples of borders with no title, borders with title, and finally borders, title, and shadow.

As usual, after declaring all the hotlist controls, the next step is to attach them to the WMF so it's aware of the controls and can start processing them and sending messages. This is accomplished with the same **AttachMenu** method call used to attach hotbars, as shown below:

```

' attach menus
demo_hotlist_id0 := WMF.AttachMenu( 0, @demo_hotlist_menu0 )
demo_hotlist_id1 := WMF.AttachMenu( 0, @demo_hotlist_menu1 )
demo_hotlist_id2 := WMF.AttachMenu( 0, @demo_hotlist_menu2 )

```

Since both hotbars and hotlists share the same root type "menu," the WMF framework can share a lot of software. In fact, the menus are processed by the exact same methods. All the methods need to do is change how the menus are drawn and how collision detection is done. O, but other than that a lot of software is re-used, which is key on a microcontroller.

## Attaching Controls to WMF

After declaring all the controls in **DAT** statements then its time to attach them to WMF so it can start processing them and sending messages. WMF currently supports up to 16 buttons and 8 menu bars. These maximum values are set in WMF\_Framework\_010.spin in the constants section and declared in the **VAR** section at the top of the program. If you need more controls then simply locate and change these two constants:

```
' windowing "controls" constants
NUM_MENUBARS = 8 ' up to 8 total menubars, change this if you need more
                  ' in most cases, you will only have a single classic menu bar
                  ' and / or multiple hotbar menus since they are like buttons

NUM_BUTTONS = 16 ' up to 16 total buttons, change this if you need more

NUM_MESSAGES = 16 ' size of message queue, holds up to 8 messages
```

Also shown is the constant for the number of messages that the message queue can hold. This only needs to be increased if the user application won't be interrogating the message queue each cycle. In that case, the messages can pile up quickly (especially for buttons) and overflow the queue.

The buttons and menu controls are nothing more than **LONG** arrays with some attribute information in the upper **WORD** and a pointer to the data declaration of the control. These data structures are shown below for completeness:

```
' menubar data structures
' each menubar control consists of a LONG record that refers to a single menubar
' the format is:
' : [ 16-bit generic flags for future use | 16-bit pointer to menu data structure]
' : [ menuFlags:16 | menuPtr:16]
long gMenuBar[ NUM_MENUBARS ]

' button data structures
' each button control consists of a LONG record that refers to a single button
' the format is
' : [ 16-bit generic flags for future use | 16-bit pointer to button data structure]
' : [ buttonFlags:16 | buttonPtr:16]
long gButton[ NUM_BUTTONS ]
```

There are also a number of ancillary data structures for these controls, but it isn't important to use them. The important thing to remember is if you need more control slots, then you have to make adjustments in the WMF\_Framework\_010.spin file and change the constants.

## Attaching Buttons

There are two sets of methods that attach and detach buttons and menus. For buttons, the attaching method is:

**PUB AttachButton( pButtonFlags, pButtonPtr )**

**pButtonFlags** is for future expansion, set to 0 for now.

**pButtonPtr** is a pointer to the first byte of the button declaration.

The method searches through the button list, locates an available slot, and returns an integer resource identifier or "handle" to that slot. This ID (not to be confused with the IDs

you set for your buttons) is needed to reference the control in all API methods. The ID is actually the array index in the global `gButton[]` array that the control ends up in, but that could change in the future.

If a button doesn't need to be processed anymore, it can be detached from the WMF with a call to `DetachButton` shown below:

```
PUB DetachButton( id )
```

`id` is simply the return value from the original call to `AttachButton`.

After attaching the buttons to the WMF, make an initial call to draw the control. This is required to get it on the screen initially, then from there WMF will handle the refresh of the control. However, if you damage the VGA surface with your own calls and disturb the controls, it's a good idea to call the draw methods to make sure the GUI is intact.

This is a common problem with GUIs and you have probably seen this with Windows, Linux, and OS X. As you do work, the screen image gets "messed up" and you see copies of things under windows drawn incorrectly. Sooner or later the OS realizes this and performs a refresh on the screen and all the controls to clean the GUI up. Same idea here, if you do a lot of direct printing to the VGA screen from time to time, you might need to call the draw methods for the buttons (and menus) to clean things up.

With that in mind, here are the actual calls to attach the "RED," "GREEN," and "YELLOW" buttons and draw them initially (an excerpt from `WMF_ButtonDemo_010.spin`):

```
' attach buttons
button_id1 := WMF.AttachButton( 0, @button1 )
button_id2 := WMF.AttachButton( 0, @button2 )
button_id3 := WMF.AttachButton( 0, @button3 )

' draw buttons
WMF.DrawButton ( button_id1, -1, WMF#BUTTON_DRAW_CMD_STATIC )
WMF.DrawButton ( button_id2, -1, WMF#BUTTON_DRAW_CMD_STATIC )
WMF.DrawButton ( button_id3, -1, WMF#BUTTON_DRAW_CMD_STATIC )
```

Using the WMF is trivial. But, understanding everything that's going on takes a little work! The prototype for the WMF method `DrawButton` is shown below:

```
PUB DrawButton ( pButtonId, pSelected, pCommand )
{{
```

DESCRIPTION: This method draws a button. The method is rather complex and needs to determine a number of factors about the button and its control flags since user mousing and button variants (standard and momentary) have different behaviors.

PARMS:

`pButtonId` - The global id of the menu to render. This 0 based index is used to access the global menu array and retrieve the menu flags and pointer.

`pSelected` - Determines if the button should be rendered as "selected", 1 to draw as selected, 0 unselected

`pCommand` - The rendering command to save compute cycles. The method can render portions of the menu such as the static elements (border and shadow) or just the button, or all of it. These commands are supported:



```

' generic button rendering commands, applies to all buttons
BUTTON_DRAW_CMD_STATIC = $04 ' draw the button control static elements that don't
                              ' need to be re-rendered dynamically
BUTTON_DRAW_CMD_DYNAMIC = $08 ' draw all the dynamic button controls, but not the
                              ' static same effect as DSEL | SEL

RETURNS: The same pButtonId sent in.
}}
```

**DrawButton** is used internally by WMF as well as exposed to user applications, so it has a lot of functionality. To draw a button, send it the ID of the button, a Boolean that indicates the control is to be drawn in a “selected” inverse video mode, and finally a command code that describes what exactly to draw. This parameter is the most interesting.

GUIs eat up a lot of cycles—any work that we can avoid, we will try to do so. The user application could potentially have a lot of buttons and controls on the screen, and the act of redrawing the buttons could use a lot of compute cycles, especially, if the static elements of the buttons are re-drawn when they don’t need to be. For example, if a button has a border, this border never changes regardless of the button’s state. Therefore, it only needs to be drawn once when the application starts (or when the screen is really disturbed). Hence, the **pCommand** parameters allow a request to draw only the dynamic or only the static elements of the control (or both by logically OR’ing the values together). This way, we can minimize the amount of redraw and speed up the GUI.

## Attaching Menus

For both menu types, hotbars and hotlists, use the same attaching and detaching methods. To attach a menu, use the method shown below:

```
PUB AttachMenu( pMenuFlags, pMenuPtr )
```

**pMenuFlags** is for future expansion, set to 0 for now.

**pMenuPtr** is a pointer to the first byte of the button declaration.

The method searches through the menu list, locates an available slot, and returns an integer resource identifier or “handle” to that slot. This ID (not to be confused with the IDs you set for your buttons) is needed to reference the control in all API methods. The ID is actually the array index in the global **gMenuBar[]** array that the control ends up in, but that could change in the future.

Detach a no-longer-needed menu bar from the WMF with a call to **DetachMenu** shown below:

```
PUB DetachMenu( id )
```

**id** is simply the return value from the original call to **AttachMenu**.

After attaching the menu bar to the WMF, make an initial call to draw the control. As with the buttons, this is required to get it on the screen initially, then from there WMF will handle the refresh of the control. However, if you damage the VGA surface with your own calls and disturb the controls, it’s a good idea to call the redraw methods to make sure the GUI is intact.

The code below lists the actual calls to attach the hotlist menus in the demo **WMF\_HotListDemo\_010.spin**:

```

' attach menus
```

```

demo_hotlist_id0 := WMF.AttachMenu( 0, @demo_hotlist_menu0 )
demo_hotlist_id1 := WMF.AttachMenu( 0, @demo_hotlist_menu1 )
demo_hotlist_id2 := WMF.AttachMenu( 0, @demo_hotlist_menu2 )

' put gui elements up once
WMF.DrawHotMenu ( demo_hotlist_id0, -1, WMF#MENU_DRAW_CMD_MENUBAR_STATIC )
WMF.DrawHotMenu ( demo_hotlist_id1, -1, WMF#MENU_DRAW_CMD_MENUBAR_STATIC )
WMF.DrawHotMenu ( demo_hotlist_id2, -1, WMF#MENU_DRAW_CMD_MENUBAR_STATIC )

```

The menu drawing method **DrawHotMenu** is much more complex than the button drawing method, but its prototype is very similar. You will need to understand it to make calls to it from time to time as you disturb the GUI image on the VGA screen and need to clean it up—refresh it. The **DrawHotMenu** prototype is shown below:

```

PUB DrawHotMenu ( pMenuId, pSelHotButton, pCommand )
{{

```

DESCRIPTION: This method draws either a hotbar or hotlist type menu. The method is rather complex and needs to determine a number of factors about the menu and its control flags. Also, due to the slow speed of spin and the memory confines, this method merges rendering of both types of "hot button" menus "hotbar" and "hotlist" for better or for worst :)

PARMS:

```

pMenuId      - The global id of the menu to render. This 0 based index is used to
               access the global menu array and retrieve the menu flags and
               pointer.

pSelHotButton - Which menu item/button to render as "selected", 0...,menu item-1, or
               -1 to not select anything.

pCommand      - The rendering command to save compute cycles. The method can
               render portions of the menu such as the static elements or just the
               hotbuttons, or all of it. These commands are supported:

' generic menubar rendering commands, applies to all menubars; hotbars, hotlists,
  etc.

MENU_DRAW_CMD_MENUBAR_STATIC = $04 ' draw the menubar control static elements that
                                     don't need to be re-rendered dynamically

MENU_DRAW_CMD_MENUBAR_DYNAMIC = $08 ' draw all the dynamic menu bar controls, but
                                     not the static same effect as DSEL | SEL

```

```

RETURNS: The same pMenuId sent in.
}}

```

**DrawHotMenu** is used internally by WMF as well as exposed to user applications, so it has a lot of functionality. To draw a menu (either hotbar or hotlist), send it the ID of the menu, an integer indicating the "selected" menu item (which will be drawn in inverse video mode), and finally a command code that describes what exactly to draw. This parameter is the same as for buttons, and selects the amount of drawing that takes place to save cycles. Hence, the **pCommand** parameters allow for a request that only the dynamic or only the static elements of the control be drawn (or both by logically OR'ing the values together). This minimizes the amount of redraw and speeds up the GUI.

**Figure 11: GUI Template WMF\_TemplateDemo\_010.spin in Action**

## Building a Generic User Application Template

GUIs are quite complex; there is simply no way around it. A lot of work has to be done. However, the trick is to hide this work and abstract it away with layers of software, so from the user application's point of view (code you write) implementing a GUI application is trivial. Thus, as a starting point, let's take a look at a complete template of sorts that all of the demos will follow. This software pattern was the basis of all the demos in the next section. Additionally, it's nice to see everything at once—with the WMF framework and API, it literally takes just a few dozen lines of code to get up and running.

Figure 11 shows the absolute minimal GUI application: a single static string up top with a single button. Also, notice the message print out to the left of the demo. As you move the mouse over the button, click, and release, numerous messages are sent for your review. The source code for the demo is named WMF\_TemplateDemo\_010.spin. It requires, a PS/2 mouse, keyboard, and VGA based Propeller board. The CON section of the demo has I/O pinout details that you might need to change for your particular Propeller board. Below is a complete listing of the demo with spurious comments deleted. Please take a close look at the demo, and then let's discuss the major components of it. (The ← symbol indicates code should continue on the previous line.)

```
CON
' -----
' CONSTANTS, DEFINES, MACROS, ETC.
' -----

_clkmode = xtal1 + pll16x
_xinfreq = 5_000_000

' import some constants from the Propeller Window Manager
VGACOLS = WMF#VGACOLS
VGAROWS = WMF#VGAROWS

' set these constants based on the Propeller device you are using
VGA_BASE_PIN    = 16      'VGA pins 16-23

MOUSE_DATA_PIN  = 24      'MOUSE data pin
MOUSE_CLK_PIN   = 25      'MOUSE clock pin
```

```

KBD_DATA_PIN      = 26      'KEYBOARD data pin
KBD_CLK_PIN       = 27      'KEYBOARD clock pin

OBJ
'-----
' Propeller Windows GUI object(s)
'-----

WMF               : "WMF_Framework_010" ' starts Propeller Window Manager and GUI

VAR
'-----
' DECLARED VARIABLES, ARRAYS, ETC.
'-----

byte  gVgaRows, gVgaCols  ' track number of VGA rows and columns
byte  gStrBuff1[64]       ' working string buffer
byte  gStrBuff2[64]       ' working string buffer
long  gVideoBufferPtr     ' pointer to video buffer

CON
'-----
' MAIN ENTRY POINT
'-----

PUB Start | WMF_message, WMF_message_id, WMF_message_code, key, index

' first step create the GUI itself
CreateAppGUI

' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
repeat
' process the GUI, this call animates and processes the GUI
' (menus and buttons only at this point)
WMF.ProcessGUI

' any messages generated in the ProcessGUI call are now in the message queue,
' process them...
if ( WMF.GetNumMessages > 0 )

' get the next message from the queue
wmf_message      := WMF.GetMessage
wmf_message_code := wmf_message & $FF
wmf_message_id   := (wmf_message >> 8)

' only process messages we are interested in, right here is where you would test
' the message code and id and then call your "handler" to do work, right now
' just print to screen...
WMF.StringTerm( string("Message Code = ") )
WMF.DecTerm( wmf_message_code, 4 )

WMF.StringTerm( string(", Message ID = ") )
WMF.DecTerm( wmf_message_id, 4 )

' test for message list off screen, if so reset and clear
if ( WMF.GetRowTerm > 47 )
WMF.GotoXYTerm( 0, 17 )
repeat index from 17 to 47
WMF.NewlineTerm
WMF.StringTerm( string("                                ") )
WMF.GotoXYTerm( 0, 18 )
else
WMF.NewlineTerm

' end PUB -----

```

```

PUB CreateAppGUI | retVal, button_id1, button_id2, button_id3, buttonIndex
' This method creates the entire user interface for the application
'
'-----
'Initialize GUI, starts VGA, Mouse, and Keyboard Drivers
'
'YOU MUST DO THIS FIRST (saving screen geometry is optional)
'-----
retVal := WMF.Init(VGA_BASE_PIN, MOUSE_DATA_PIN, MOUSE_CLK_PIN, KBD_DATA_PIN, KBD_CLK_PIN, WMF#NULL )

' rows encoded in upper 8-bits. columns in lower 8-bits of return value, redundant code
' really since we pull it in with a constant in the first CON section, but up to you!
gVgaRows := ( retVal & $0000FF00 ) >> 8
gVgaCols := retVal & $000000FF

' VGA buffer encoded in upper 16-bits of return value
gVideoBufferPtr := retVal >> 16
'
'-----
'setup screen colors
'-----
WMF.ClearScreen( WMF#CTHEME_AUTUMN_FG, WMF#CTHEME_AUTUMN_BG )

' AREA51 for testing methods...

WMF.PrintString(string("Parallax Window Manager Framework | Template Demo | (c) Parallax +
2011"), 16, 1, 0)

' attach buttons
button_id1 := WMF.AttachButton( 0, @button1 )

' draw buttons
WMF.DrawButton ( button_id1, -1, WMF#BUTTON_DRAW_CMD_STATIC )

' set terminal cursor for message printing
WMF.GotoXYTerm( 0, 18)

' return to caller
return

' end PUB -----

DAT
'
'-----
' DATA STRUCTURES AND WORKING RESOURCES
'-----
'
'-----
' Example Button
'-----

button1 LONG
    BYTE WMF#BUTTON_STYLE_TYPE_STANDARD ' type of button, "standard" supported currently
    BYTE WMF#ATTR_DRAW_BORDER           ' rendering attributes, shadows, borders, etc.
    BYTE 45, 20                         ' x,y position of button (width based on string)
    BYTE WMF#BUTTON_STYLE_STATE_ACTIVE, 10, "Push Me", 0 ' flags, id, and string for the
                                                         button itself

```

The application starts off declaring constants that are used to pass values to **init**, the main initialization method in the WMF framework. The **init** method takes as parameters the pin group for the VGA, mouse, and keyboard. Next is the **OBJ** section which includes the single object WMF\_Framework\_010.spin. This is required for every application. But, the nice

thing is that it includes drivers for VGA, mouse, and keyboard—these are all taken care of for you in the framework. Of course, you lose a little control, but the point of a framework is to take control from you and do the work so you don't have to.

Next the **VAR** section defines some globals to hold strings, the video pointer returned from the **init** call, and a couple other tracking variables for the width and height of the VGA screen. Now comes the entry point for the template: the standard Spin **start** method. Here's where the software pattern starts that you should use for all your projects.

## The Initialization Method **CreateAppGui**

Upon entry, the one-time location initialization method **CreateAppGUI** is called. This is a local method that serves to initialize everything for the application and GUI. Its name is arbitrary; **CreateAppGUI** is used as a convention in all the demos. The purpose of this method is to initialize the WMF framework, set up the colors for the VGA screen, initialize any global data structures, draw any static GUI elements, attach all controls, and finally call the draw methods one time to get things up on the screen.

Take a look at the code in **CreateAppGUI**—it follows these steps closely. In fact, you can copy this code exactly for your template and the only changes/additions you will make are the controls that are attached and drawn. In this case, a single button is attached and drawn. Thus, the entire GUI initialization for active controls is two lines of code!

After **CreateAppGUI** is complete, it returns back to the main **start** method. This is where things get interesting and the pattern is important to follow.

## The Main Event Loop

The main event loop is the heart of an event-driven system. The idea is that the user-level application needs to call a method that is an entry point into the GUI processing system and give it a cycle to operate. In other words, relinquish control to the WMF framework once a cycle to let it respond to user input and generate messages. This is so important, let's re-list the code fragment below:

```
' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
repeat
' process the GUI, this call animates and processes the GUI
' (menus and buttons only at this point)
WMF.ProcessGUI

' any messages generated in the ProcessGUI call are now in the message queue,
' process them...
if ( WMF.GetNumMessages > 0 )

' get the next message from the queue
wmf_message      := WMF.GetMessage
wmf_message_code := wmf_message & $FF
wmf_message_id   := (wmf_message >> 8)
.
.
.
```

The infinite **repeat** loop is entered at the top of the fragment; this repeats forever or until you need to do something non-GUI related. With each iteration of the **repeat** loop a call is made to **ProcessGUI**. This call does everything; it literally runs through hundreds of lines of code or more in the framework API, processes, draws, updates, computes user actions, and finally inserts messages into the message queue. Recall that this queue is the conduit of

communication from the WMF framework to the user application. Thus, on return from the **ProcessGUI** method the GUI will magically update based on user actions with the mouse, and messages will potentially be inserted into the message queue. This is where the messaging API comes into play.

## The Messaging API

As messages are inserted into the message queue by the WMF framework, the user application's job is to interrogate the message queue, pull messages from the queue, process them, and call event handlers to do "work." This is accomplished with two methods: **GetNumMessages** and **GetMessage**.

**GetNumMessages** checks the message queue and returns the number of pending messages that are in the queue. If the number is greater than 0, a call to **GetMessage** retrieves the message as a **WORD** in the following format.

Message Format: WORD = [message\_id:8 | message\_code:8]

Thus, the upper 8 bits hold the ID of the control itself that generated the message. This is the same ID that the control was created with, thus it is programmer specific. The message code is something else. This code is control context sensitive and gives you an idea of what kind of action occurred. For example, with menu items there is a single message generated if and only if the user clicks on a menu item and **releases** on that same item in one operation. If the user moves the mouse off the menu item then releases, or clicks in open space and then releases on a menu item, no messages are generated. Thus, menu item events are very specific. Here is the message code for these events:

```
' hotbar/list menu messages
WMF_MSG_MENU_BUTTON_CLICKED = $08 ' a button was clicked
```

Therefore, if you ever see the value of **WMF\_MSG\_MENU\_BUTTON\_CLICKED** come through the event queue, you know that a menu item/hot button was clicked.

Now, buttons are a lot more interactive and generate a lot more messages. The idea behind buttons is to make them as flexible as possible and support all kinds of user application ideas. Therefore, the event system gives a lot more information about standard buttons. For example, even if you don't click on a button, if you hover over a button, a message will be generated for that button indicating you are hovering over it. In other words, you have "focused" on it. Similarly, when you move the mouse off a button, you get a "focus lost" message. In addition to those messages, you get a message when you click on a button as well as a message when you click and release on a button. Thus, you can really interrogate buttons to a high degree of granularity and figure out what the user is doing or not doing.

However, since buttons generate so many messages, make sure to pull them continually from the event queue so they don't overflow. To do this, make calls to **GetMessage** with each iteration of the main event loop. The following code lists all the messages buttons can generate:

```
' button messages, unlike the menu's which require a press/release event ON the
' control, buttons are different,
' at least in our implementation.

WMF_MSG_BUTTON_CLICKED      = $10 ' this is sent when a complete press/release cycle
                                ' is complete
```



```

WMF_MSG_BUTTON_PRESSED      = $11 ' this is sent for all buttons when the button is
                                ' pressed initially
WMF_MSG_BUTTON_RELEASED     = $12 ' this is sent for all buttons when the button is
                                ' released while still on the button
WMF_MSG_BUTTON_ONFOCUS      = $13 ' this is sent for all buttons when the button is
                                ' initially hovered over / selected
WMF_MSG_BUTTON_LOSTFOCUS    = $14 ' this is sent for all buttons when the button was
                                ' hovered over/selected and moved off of

```

Taking all this in, let's refine the operations the main event loop performs:

- Calls **ProcessGUI** to process the controls, update the screen, and generate messages.
- Tests if any messages are in the event queue with a call to **GetNumMessages**.
- If there are messages in the event queue they are retrieved with a call to **GetMessage** and broken down in to their constituent components; control ID and message code.
- Finally, based on the message code and ID, calls the appropriate message handler to perform work.

The last item in the bullet list we need to cover in detail, but this is best discussed after seeing some of the demos for context. Therefore, let's dig into the demos now and follow up with the final subject of event and message handling.

## WMF GUI Examples

The following examples should serve to solidify everything discussed so far on the theory side of the Propeller Window Manager Framework. In fact, these demos really show how easy it is to build GUI applications (most are literally a few dozen lines of code not including the control declarations) and you can use these demos themselves as a starting point for all of your applications.

### Demo Preparation

All the example demos require a Propeller development board that has a PS/2 mouse and keyboard, audio, and a VGA port. Also, the demos assume a 5 MHz crystal. Any other particular requirements are detailed in each specific demo.

The Propeller Demo Board Rev. C<sup>[3]</sup> was used to create these particular demos, but other boards will work with a few changes to relevant constants controlling the port I/O pins, etc. in the **CON** section of each demo's code:

```

' set these constants based on the Propeller device you are using
VGA_BASE_PIN      = 16      'VGA pins 16-23

MOUSE_DATA_PIN    = 24      'MOUSE data pin
MOUSE_CLK_PIN     = 25      'MOUSE clock pin

KBD_DATA_PIN      = 26      'KEYBOARD data pin
KBD_CLK_PIN       = 27      'KEYBOARD clock pin

```

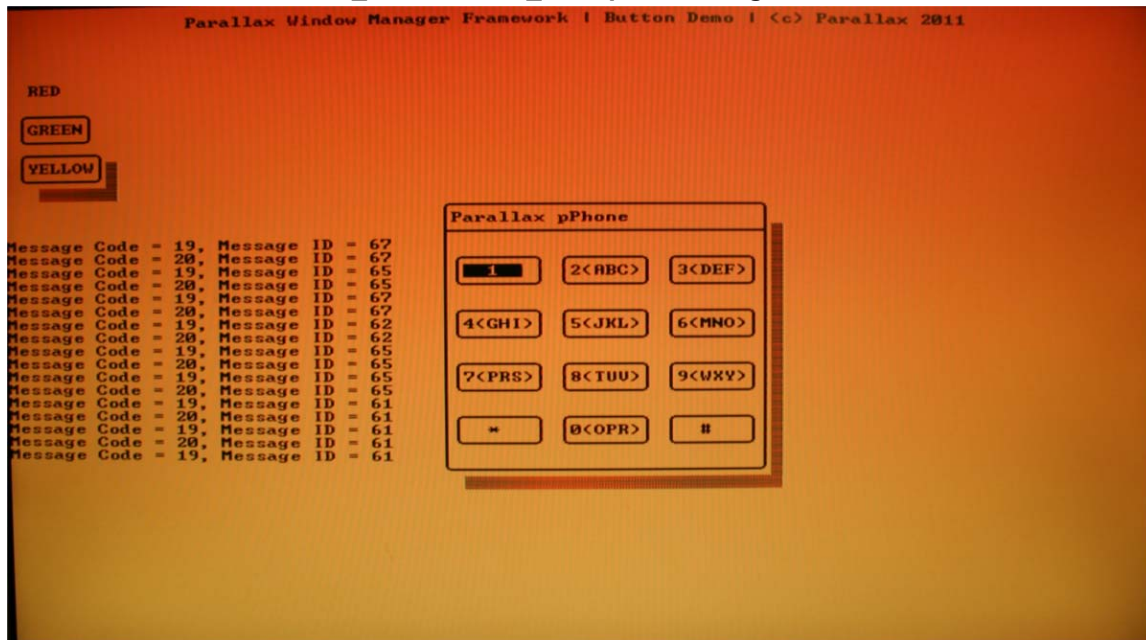
Simply make changes here to suit a particular board. Also, don't forget that the demos assume a 5 MHz crystal as well, so change as needed.

All the sample application top-level files are included in this application note's zip file (see Resources, page 49).

### Example 1: Button Demo

Load the sample application top-level file WMF\_ButtonDemo\_010.spin. Figure 12 below shows the demo running on a standard VGA-enabled LCD TV.

**Figure 12: Screen Shot of WMF\_ButtonDemo\_010.spin Running**



The button demo is based on the previous template demo as a starting point. As shown in Figure 12, the demo creates three buttons on the top left of the VGA screen as well as a phone-style keypad in the middle of the screen. The buttons on the top left illustrate some variants for buttons: plain, border, border plus shadows. The keypad matrix in the middle of the screen illustrates how static imagery can be used along with active controls to create a new-looking control. In this case, a frame was drawn with title "Parallax pPhone" and then a number of buttons were attached with positions on a 2D grid.

The demo doesn't do anything except allow you to move the mouse around and generate messages. All messages print to the screen on the left, and when the printing reaches the bottom of the screen it clears the messages and resets.

The complete source is there for review in the ZIP file, but it's nearly identical to the template demo's source. The only real changes are that there are more button control declarations in the DAT section, and the **CreateGUIApp** method attaches all the buttons and draws the frame for the phone keypad. The demo enters at the **start** method which makes a call to **CreateGUIApp**, listed below. (The + symbol indicates code should continue on the previous line.)

```
PUB CreateAppGUI | retVal, button_id1, button_id2, button_id3, buttonIndex
' This method creates the entire user interface for the application
'
'-----
' Initialize GUI, starts VGA, Mouse, and Keyboard Drivers
'
' YOU MUST DO THIS FIRST (saving screen geometry is optional)
'-----
```

```

retVal := WMF.Init(VGA_BASE_PIN, MOUSE_DATA_PIN, MOUSE_CLK_PIN, KBD_DATA_PIN, KBD_CLK_PIN, WMF#NULL )

' rows encoded in upper 8-bits. columns in lower 8-bits of return value, redundant
' code really
' since we pull it in with a constant in the first CON section, but up to you!
gVgaRows := ( retVal & $0000FF00 ) >> 8
gVgaCols := retVal & $000000FF

' VGA buffer encoded in upper 16-bits of return value
gVideoBufferPtr := retVal >> 16

' -----
' setup screen colors
' -----
WMF.ClearScreen( WMF#CTHEME_AUTUMN_FG, WMF#CTHEME_AUTUMN_BG )

' AREA51 for testing methods...

WMF.PrintString(string("Parallax Window Manager Framework | Button Demo | (c)Parallax 2011"),16, 1, 0)
' attach buttons
button_id1 := WMF.AttachButton( 0, @button1 )
button_id2 := WMF.AttachButton( 0, @button2 )
button_id3 := WMF.AttachButton( 0, @button3 )

' draw buttons
WMF.DrawButton ( button_id1, -1, WMF#BUTTON_DRAW_CMD_STATIC )
WMF.DrawButton ( button_id2, -1, WMF#BUTTON_DRAW_CMD_STATIC )
WMF.DrawButton ( button_id3, -1, WMF#BUTTON_DRAW_CMD_STATIC )

' draw frame around phone
WMF.DrawFrame( PX-2, PY-5, 28, 21, string("Parallax pPhone"), WMF#ATTR_DRAW_SHADOW, +
gVideoBufferPtr, VGACOLS )

' each button definition requires 13 bytes, so we can use a loop to insert
' each button rather than its specific starting address
repeat buttonIndex from 0 to 11

' attach buttons, not calculation of base address of each button from @keypad_button1
keypad_id[ buttonIndex ] := WMF.AttachButton( 0, @keypad_button1 + 13*buttonIndex )

' draw buttons
WMF.DrawButton ( keypad_id[ buttonIndex ], -1, WMF#BUTTON_DRAW_CMD_STATIC )

WMF.GotoXYTerm( 0, 18)

' return to caller
return

' end PUB -----

```

The header section initializes the WMF framework, then the actual code that does work and attaches the controls and draws the GUI is literally 12 lines of code! Take a look right under the "AREA 51" comment; that's where the custom code starts.

The method simply attaches the "RED," "GREEN," "YELLOW," buttons, draws them, then proceeds to draw the frame for the keypad and attach all the key buttons. This deserves a little explanation. A loop is used to locate the starting address of each button declaration in the DAT section. This is possible since each button uses the same size string for the button text. Thus, a constant offset can be used to index from button to button and a simple loop can be used rather than an explicit list of **AttachButton** calls with the starting address of each button. Keep this in mind: if at all possible, when creating a number of controls such as buttons, make them have the same string size (even if you have to pad with spaces) so accessing them is easier for pointer arithmetic.

Once the **CreateGUIApp** method completes, it returns to the **start** method. Execution continues with the standard infinite **repeat** loop that checks to see if the number of messages is greater than zero. If so, the next message is retrieved from the message queue and printed to the screen.

```
PUB Start | wmf_message, wmf_message_id, wmf_message_code, key, index

' first step create the GUI itself
CreateAppGUI

' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
repeat
' process the GUI, this call animates and processes the GUI (menus and buttons only at
' this point)
WMF.ProcessGUI

' any messages generated in the ProcessGUI call are now in the message queue,
' process them...
if ( WMF.GetNumMessages > 0 )

' get the next message from the queue
wmf_message      := WMF.GetMessage
wmf_message_code := wmf_message & $FF
wmf_message_id   := (wmf_message >> 8)

' only process messages we are interested in.

WMF.StringTerm( string("Message Code = ") )
WMF.DecTerm( wmf_message_code, 4 )

WMF.StringTerm( string(", Message ID = ") )
WMF.DecTerm( wmf_message_id, 4 )

' test for message list off screen, if so reset and clear
if ( WMF.GetRowTerm > 47 )
    WMF.GotoXYTerm( 0, 17 )
    repeat index from 17 to 47
        WMF.NewlineTerm
        WMF.StringTerm( string(" ") )
    WMF.GotoXYTerm( 0, 18 )
else
    WMF.NewlineTerm

' end PUB -----
```

The actual code that does all the work is only about 5 lines long. The rest is to pretty-print the messages out and clear the screen when they scroll off. Thus, this entire GUI application takes less than 20 lines of actual work code to get up and running. That's the power of a data-driven, event-handling GUI system.

The **DAT** section declarations of the all the buttons have been omitted; please review them in the source code itself. They are standard button declarations, mostly copied and pasted.

Experiment with this demo, and notice that it generates a lot of messages. Buttons generate messages when the user mouses over them, clicks them, releases them, etc. This gives the control needed to write very responsive GUI loops and give buttons various behaviors. However, what if not all of these messages are wanted or needed? There is no way to stop the event system from inserting these events into the message queue. However, simple conditional(s) in the main event loop can filter them so that it only processes specific types of messages.

For example, to only process button and menu click events, add this bit of code in the main loop right before processing the message. (The + symbol indicates code should continue on the previous line.)

```

    only process messages we are interested in
    if ( (wmf_message_code == WMF#WMF_MSG_MENU_BUTTON_CLICKED) OR +
                                                (wmf_message_code==WMF#WMF_MSG_BUTTON_CLICKED))
    .
    .
    .

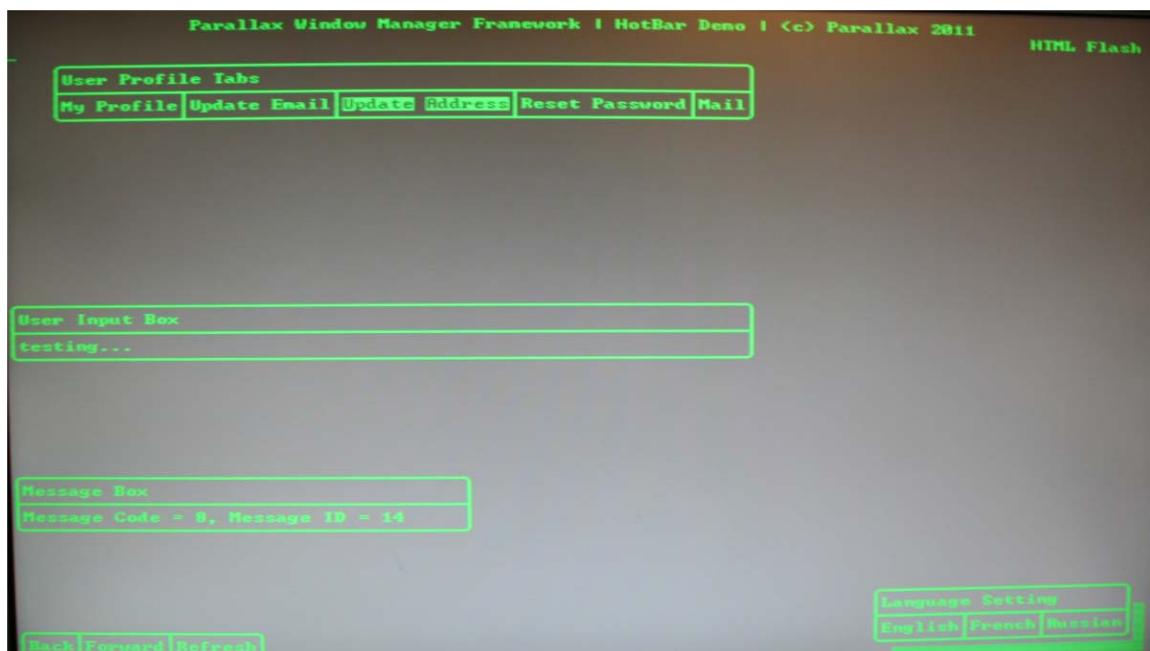
```

In this case, once a message has been retrieved with **GetMessage** it will be ignored if it is not one of the above message codes. As an example of putting this to use, take a look at **WMF\_ButtonDemo2\_010.spin**. It's identical to the demo discussed here except it adds this single filter to the message printed logic. Thus, it only prints a message out when it's a complete click cycle message and all the mouse on-focus/off-focus type events are ignored. Furthermore, it could filter for certain IDs, etc. It is up to you. But, remember, the user application always has to read all the messages out of the message queue so it doesn't overflow even if it doesn't process them.

## Example 2: HotBar Menu Demo

Load the sample application top-level file **WMF\_HotBarMenuDemo\_010.spin**. Figure 13 below shows the demo running on a standard VGA-enabled LCD TV.

**Figure 13: Screen Shot of WMF\_HotBarMenuDemo\_010.spin Running**



This demo illustrates the use of the horizontally oriented hotbar menus. It creates a total of four hotbar menus and places them roughly at the four corners of the screen. Each menu illustrates a variation of the hotbar (clockwise from top right): no borders or title, borders with title and shadow, borders without title, borders with title.

Try moving the mouse around and clicking on the various menu items in the hotbars, which will print messages in the "Message Box" frame. This is a static GUI element created in the **CreateGUIApp** method as part of the initialization. The code fragment is shown below:

```
' set the terminal cursor down screen a bit, and draw a little frame around it
WMF.GotoXYTerm( 1, 25 )
WMF.DrawFrame( 0, 22, 64, 5, string("User Input Box"), 0, gVideoBufferPtr, VGACOLS )

' message box
WMF.DrawFrame( 0, 35, 40, 5, string("Message Box"), 0, gVideoBufferPtr, VGACOLS )
```

Additionally, the "User Input Box" frame for a user keyboard entry is created and drawn. We will get to this in a moment.

The **DAT** declarations for the menus were shown in the previous section, Declaring HotBar Menus. This demo works exactly as the button demos do: declare the menus in the **DAT** section, attach and draw the menus in the **CreateGUIApp** method along with any static element rendering (the two frames in the listing above), and finally return to the main event where all the action occurs. Let's discuss this in context of the complete **start** method shown below:

```
PUB Start | wmf_message, wmf_message_id, wmf_message_code, key

' first step create the GUI itself
CreateAppGUI

' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
repeat
' process the GUI, this call animates and processes the GUI
' (menus and buttons only at this point)
WMF.ProcessGUI

' any messages generated in the ProcessGUI call are now in the message queue,
' process them...
if ( WMF.GetNumMessages > 0 )

' get the next message from the queue
wmf_message      := WMF.GetMessage
wmf_message_code := wmf_message & $FF ' low byte is the code
wmf_message_id   := (wmf_message >> 8) ' high byte is the control id

' print message code and id in "message box" frame
bytefill(@gStrBuff1, 0, 64 )
WMF.StrCpy( @gStrBuff1, string("Message Code = " ) )
WMF.itoa(wmf_message_code, 10, 2, @gStrBuff1[ strlen( @gStrBuff1 ) ])
WMF.StrCpy( @gStrBuff1[ strlen( @gStrBuff1 ) - 1 ], string(", Message ID = " ) )
WMF.itoa(wmf_message_id, 10, 2, @gStrBuff1[ strlen( @gStrBuff1 ) ])
WMF.PrintString( @gStrBuff1, 1, 38, 0 )

' process keyboard input for fun, type and see it in middle of screen
if ( (key := WMF.KeyboardKey) <> 0)
' do what you want with keyboard input...
WMF.PrintTerm( key )

' end PUB -----
```

This once again uses the standard software pattern: call **CreateGUIApp** at the top of the event loop a single time, call **ProcessGUI** each cycle, test the number of messages in the

queue, and then if any messages are in the message queue read them out with a call to **GetMessage** and process them.

However, this time the code decomposes the message and “pretty prints” it. Instead of printing the messages to the screen using terminal console calls, it parses the message and then generates and prints a single string within the confines of the static decoration frame “Message Box.” This illustrates the use of static elements to create new controls from the user’s point of view.

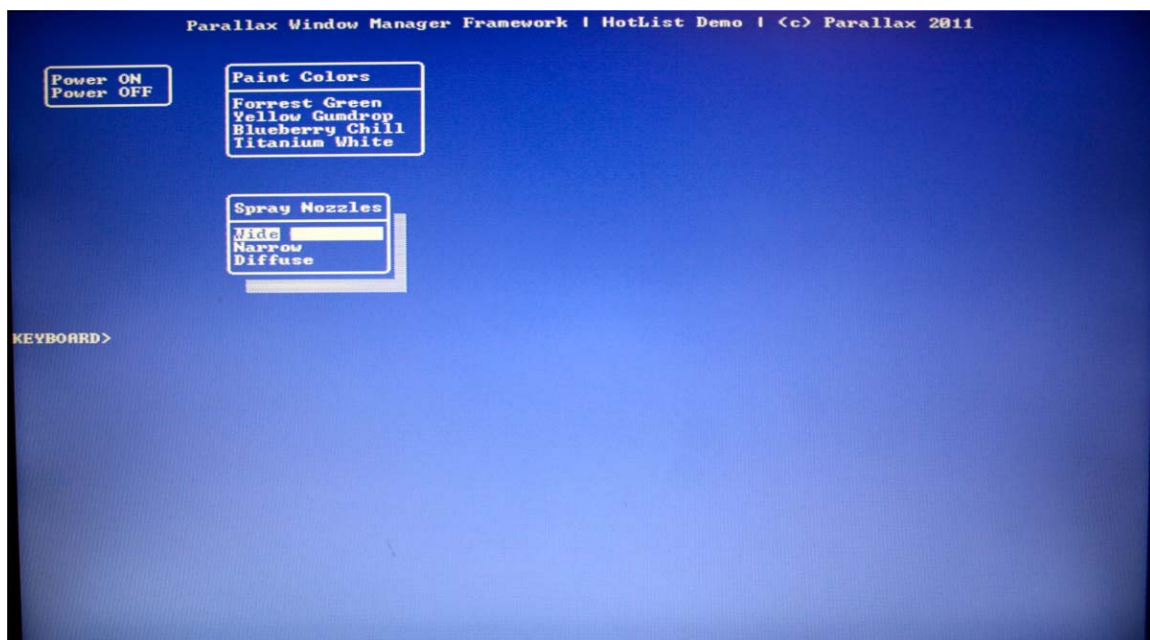
Also, a second feature of this demo which hasn’t been discussed is the integration of keyboard input. The WMF framework uses the mouse alone to click on controls and interact with them. However, there may be times where an application needs textual input. This is achieved by making calls to the WMF and tunneling through it to the keyboard driver. In fact, most of the keyboard methods are supported in WMF by pass-through methods. The end of the method above shows an example where the keyboard **WMF.KeyboardKey** is called which returns the last ASCII key pressed or 0 otherwise. This key value is then printed to the screen in the static frame labeled “User Input Box.” Go ahead and try typing at any time with the local keyboard to see the output printed in this box.

This illustrates the integration of both mouse and keyboard simultaneously without the user application having to worry about the details of managing either. To take this further, imagine having a GUI button or menu item that controls a text input box. For example, when the user clicks a specific button or menu item, a static frame could be drawn for the user to enter an address or other information. To learn more about the keyboard methods make sure to review the **WMF\_Framework\_010.spin** source and peruse the API listing at the end of this application note.

### Example 3: HotList Menu Demo

Load the sample application top-level file **WMF\_HotListMenuDemo\_010.spin**. Figure 14 below shows the demo running on a standard VGA-enabled LCD TV.

**Figure 14: Screen Shot of WMF\_HotListMenuDemo\_010.spin Running**





This demo illustrates the use of the vertically oriented HotList menus. This type of menu is more common with lists of information or drop-down menus. In fact, combining a hotbar with hotlists could create a complete Windows-like menu with a menu bar (hotbar) and sub-menus (hotlists). The demo itself is once again very similar to the previous hotbar demo; the most important difference is that hotlist menus rather than hotbar menus are defined in the **DAT** section. Other than that, the programs are nearly identical: enter the **start** method, call **CreateGUIApp**, initialize the application, attach menus, and then return to the main event loop to retrieve messages and print them to the screen.

Hopefully, the power of the WMF framework is becoming evident. In most cases, just change a few lines of code (and of course the menu definitions themselves) to instantly create a complete and functional GUI. Thus, the work of creating the GUI portion of the application is almost trivial. The real work is in the event handlers themselves. These handlers contain the actual code that performs the work intended by the GUI messages from the user's interaction with the GUI.

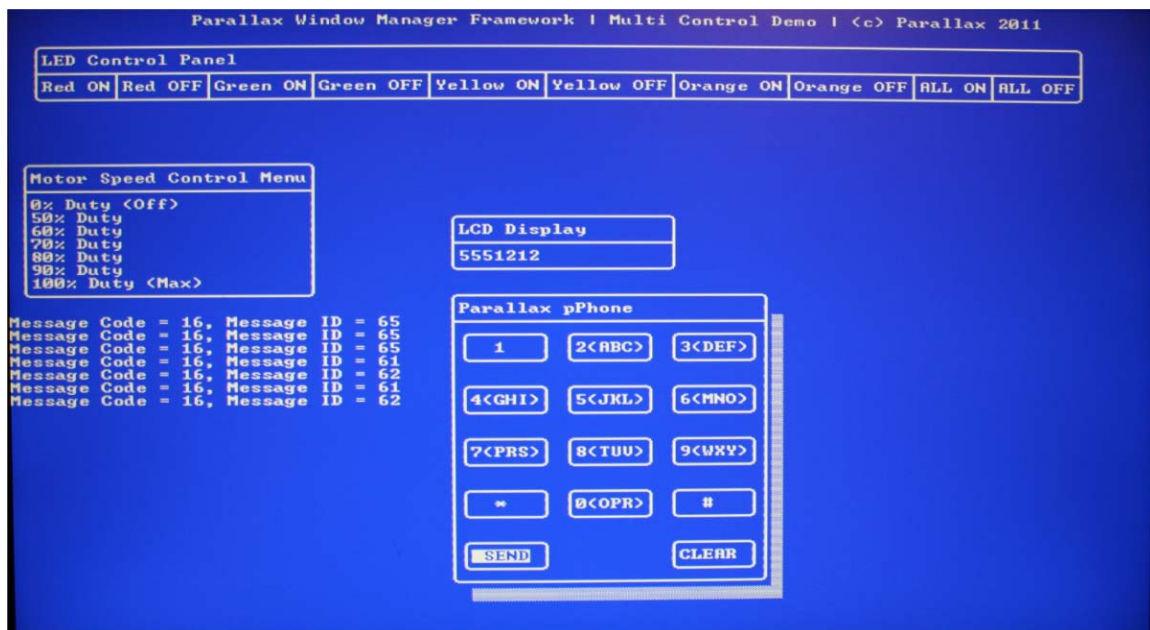
Of course, these handlers usually have nothing to do with GUI programming. They are servants of the GUI program and might control motors, play music, launch missiles, change the position of an antenna, etc. These handlers do the work that the user wants performed as he indicates these desires through his interaction with the GUI. With this in mind, the next demo integrates everything demonstrated so far and illustrates a complete multi-control GUI with event handlers that control LEDs, motors, and make sounds.

#### Example 4: Multi-Control Demo with the Real-World Controls and Event Handlers

This demo generates sound, turns a motor and flashes LEDs. Your Propeller development board will need a set of headphones or an audio amplifier such as a TV or stereo, 4 LEDs (different colors ideally), and a small, low-current DC motor (3 to 6 V).

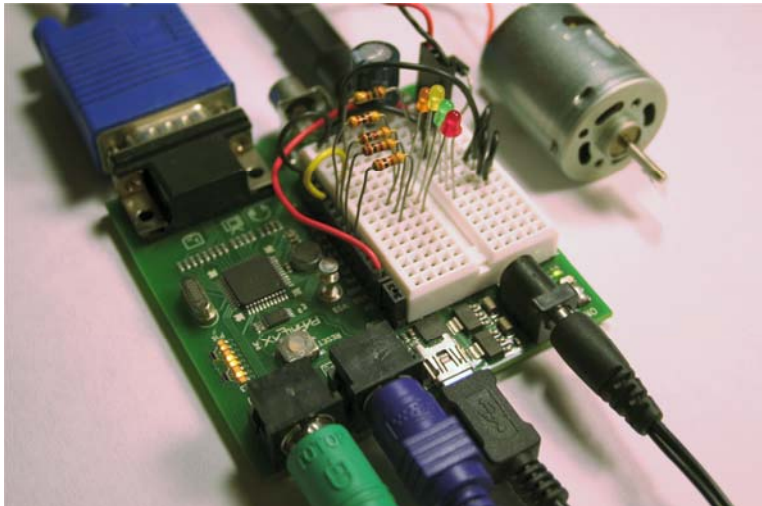
Load the sample application top-level file **WMF\_MultiControlDemo\_010.spin**. Figure 15 below shows the demo running on a standard VGA-enabled LCD TV.

**Figure 15: Screen Shot of WMF\_MultiControlDemo\_010.spin Running**



This demo is the culmination of everything covered thus far. Referring to Figure 15, there are three primary controls and two static GUI elements. The active controls are not only functional, but are connected to event handlers in the program itself that perform actual work based on the messages from each control. So, to get the most out of this demo you need to hook up some real-world hardware to your Propeller development board to see everything work. Figure 16 is a photograph of the Propeller Demo Board<sup>[3]</sup> used as the platform for this demo, along with some LEDs, a motor, and other circuitry on the white solderless prototyping board.

**Figure 16: The Propeller Demo Board Setup for the Multi-control Demo**



This demo not only shows how to use all the different types of controls at once in the same application, but how to connect the controls to real-world devices by interpreting the messages in the main event loop rather than just printing them to the screen. The listing below is an excerpt from the demo that shows the entire **start** method in the demo that contains the event loop and the calls to the event handlers. (The **←** symbol indicates code should continue on the previous line.)

```
PUB Start | wmf_message, wmf_message_id, wmf_message_code, key, index

' first step create the GUI itself
CreateAppGUI

' MAIN EVENT LOOP - this is where you put all your code in an infinite loop...
repeat
' process the GUI, this call animates and processes the GUI
' (menus and buttons only at this point)
WMF.ProcessGUI

' any messages generated in the ProcessGUI call are now in the message queue,
' process them...
if ( WMF.GetNumMessages > 0 )

    ' get the next message from the queue
    wmf_message      := WMF.GetMessage
    wmf_message_code := wmf_message & $FF
    wmf_message_id   := (wmf_message >> 8)

    ' only process messages we are interested in
    if ( (wmf_message_code == WMF#WMF_MSG_MENU_BUTTON_CLICKED) OR      ←
        (wmf_message_code==WMF#WMF_MSG_BUTTON_CLICKED))
        ' this is where you put all your message "handlers", these handlers inspect
```

```

' the message and then make calls to your internal logic, drivers, methods,
' whatever you need to do to update your application and the GUI itself

' process all LED hotbar messages, we know id are from 10 to 19 inclusive
if ( (wmf_message_id => 10) AND (wmf_message_id =< 19))
' call handler
  LED_MessageHandler ( wmf_message_code, wmf_message_id )

' process all motor hotlist messages, we know id are from 30 to 36 inclusive
if ( (wmf_message_id => 30) AND (wmf_message_id =< 36))
' call handler
  Motor_MessageHandler ( wmf_message_code, wmf_message_id )

' process all phone keypad button matrix messages,
' we know id are from 61 to 74 inclusive
if ( (wmf_message_id => 61) AND (wmf_message_id =< 74))
' call handler
  Phone_MessageHandler ( wmf_message_code, wmf_message_id )

' this code prints the messages to screen for reference
WMF.StringTerm( string("Message Code = ") )
WMF.DecTerm( wmf_message_code, 4 )

WMF.StringTerm( string(", Message ID = ") )
WMF.DecTerm( wmf_message_id, 4 )

' test for message list off screen, if so reset and clear
if ( WMF.GetRowTerm > 47 )
  WMF.GotoXYTerm( 0, 23 )
  repeat index from 23 to 47
    WMF.NewlineTerm
    WMF.StringTerm( string(" ") )
  WMF.GotoXYTerm( 0, 24 )
else
  WMF.NewlineTerm

' end PUB -----

```

Reviewing the code, it looks like the standard event loop seen throughout these demos. It makes a call to **GetNumMessages** and if there are messages in the queue, that method calls **GetMessage** and parses the message into its components. Here is where the similarities diverge.

In this event loop, a number of conditional filters ask the question “Which control sent this message?” Based on the answer, one of three different “event handlers” is called (highlighted above). These event handlers are nothing more than method calls, but the idea is that they handle the events and finally take some action.

The handlers can be quite complex and diverge from the subject of GUI programming. So, we aren’t going to look at all of them, but let’s look at the simplest which is the LED control handler. Reviewing the code above, if the message ID value is in the 10 to 19 range, then these messages must be from the LED control hotbar (because we assigned these IDs when we created the control). With this knowledge in hand, the method **LED\_MessageHandler** is called and it determines what to do based on the ID of the message. Here’s the method for inspection:

```

PUB LED_MessageHandler (pCode, pId)

' This method handles all messages from the LED hotbar and processed them
' by turning on/off the various LEDs connected to the demo unit.

```

```

' This method is representative of a "message handler" in an event based
' GUI system, you might have many of these in your complete application that
' process various messages from the GUI. Of course, we could optimize this method
' but that's not the point here, the point is to be quick and dirty and show how
' each user GUI actions translate into actions in your program and how to process
' them.

' what is the menu item id?
case pId
10:
    OUTA[ LED_RED_PIN ] := 1 ' turn red LED on
11:
    OUTA[ LED_RED_PIN ] := 0 ' turn red LED off
12:
    OUTA[ LED_GREEN_PIN ] := 1 ' turn green LED on
13:
    OUTA[ LED_GREEN_PIN ] := 0 ' turn green LED off
14:
    OUTA[ LED_YELLOW_PIN ] := 1 ' turn yellow LED on
15:
    OUTA[ LED_YELLOW_PIN ] := 0 ' turn yellow LED off
16:
    OUTA[ LED_ORANGE_PIN ] := 1 ' turn orange LED on
17:
    OUTA[ LED_ORANGE_PIN ] := 0 ' turn orange LED off
18:
    ' turn them all on
    OUTA[ LED_RED_PIN ] := 1
    OUTA[ LED_GREEN_PIN ] := 1
    OUTA[ LED_YELLOW_PIN ] := 1
    OUTA[ LED_ORANGE_PIN ] := 1

19:
    ' turn them all off
    OUTA[ LED_RED_PIN ] := 0
    OUTA[ LED_GREEN_PIN ] := 0
    OUTA[ LED_YELLOW_PIN ] := 0
    OUTA[ LED_ORANGE_PIN ] := 0

' return to caller
return

' end PUB -----

```

As an example, when the LED control hotbar was created, ID number 14 was assigned to "Yellow ON." Looking at the above code, ID 14 sets the I/O pin for the yellow LED to HIGH, turning it on. It's as simple as that!

Thus, all the event handlers simply interrogate the message IDs and perform whatever action the GUI visually indicates should be taken. Now, that the event handlers have been discussed, let's review each sub-system. We will return to event handling and message processing after this demo in the final section of the application note to solidify the concepts shown in this example.

### The Parallax pPhone DTMF keypad

The keypad in the middle of the demo GUI requires sound output. Many Propeller-based development boards have built-in sound hardware (usually an R/C integrator and filter for a PWM A/D converter). The keypad operates much like a cell phone: dial a series of numbers by pressing the numbers on the keypad, and the numbers will show up in the "LCD Display" static display element above the keypad. Then press the "Send" button and the demo iterates through the fictitious phone number and generates DTMF tones. All this work is

done in the main loop by interpreting the messages in the message queue and then calling an event handler for the keypad/sound generation (more on the event-handling aspect at the end of this demo).

The GUI aspect of the DTMF keypad is nothing more than a matrix of buttons, fourteen in total, with a static frame around them. Each button has the exact same label string size to make their memory footprint the same size for ease of indexing. For example, here's the declaration of the "[ 0 (OPR) ]" button in the **DAT** section of the demo:

```
keypad_button0  BYTE
                BYTE WMF#BUTTON_STYLE_TYPE_STANDARD      ' type of button
                BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering
                BYTE PX+9, PY+12                          ' x,y position of button
                BYTE WMF#BUTTON_STYLE_STATE_ACTIVE, 71, "0(OPR)",0 ' style, id, string, null
```

As buttons are pressed, a string buffer is filled with the alpha/numeric value of the keys, and the values are printed to the static LCD display frame for viewing as well. When the "Send" button is pressed, the number is "dialed" (again the event handler for the keypad does this work).

Of course, there is no phone equipment, so instead the program generates DTMF tones by using a sound driver (NS\_sound\_drv\_052\_11khz\_16bit.spin). It sends the sounds to a single pin which must connect to your Propeller board's audio amplifier, headset, or TV audio port. If needed, change the sound I/O pin in the **CON** section of the demo program to match your particular hardware.

If you don't have sound hardware, then you can build some with a few parts. The code listing below shows the audio I/O pin for sound output and a schematic of the circuit needed for a hardware platform that doesn't have sound built in.

```
' Audio I/O pin (mono), this is where sound will be generated in WMF format

AUDIO_PIN = 10 ' pin 10 us built in sound amp in demo board
                ' C3 uses pin 24, HYDRA uses pin 7 -- all have dedicated WMF filters
                ' if you don't have a dedicated WMF filter then simply use whatever
                ' I/O pin you want and use the circuit below to hook up the sound pin

{{
Audio WMF filter circuit with low pass filter R1|C1
cutoff at f3db = 1/2*PI*R1*C1 = ~21.2Khz

    Audio Output Pin --> R1 = 1K --> C2 = 10-22uF --> To 4-8 ohm small speaker or audio amp
                        |
                        C1 = 7.5nF
                        |
                        GND

}}
```

### The LED Control Panel

The LED control panel at the top of the GUI is a hotbar menu with 10 menu items on it. Here's the declaration of the hotbar in the **DAT** section of the demo:

```
' begin HOTBAR -----
led_hotbar LONG ' start at LONG boundary
            BYTE WMF#MENU_STYLE_TYPE_HOTBAR      ' type of menubar
            BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes
```

```

    BYTE WMF#ASCII_VLINE      ' space filler character
    BYTE 2,6,92                ' x,y position of menu and width
    BYTE 10                    ' number of menu items/buttons
    BYTE "LED Control Panel",0 ' optional menu title string

    ' list of menu titles for the menubar, each consists of the style flags, id,
    ' the string, and a null terminator at end
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 10, "Red ON", 0 ' hotbutton 0
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 11, "Red OFF", 0 ' hotbutton 1
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 12, "Green ON", 0 ' hotbutton 2
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 13, "Green OFF", 0 ' hotbutton 3
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 14, "Yellow ON", 0 ' hotbutton 4
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 15, "Yellow OFF", 0 ' hotbutton 5
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 16, "Orange ON", 0 ' hotbutton 6
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 17, "Orange OFF", 0 ' hotbutton 7
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 18, "ALL ON", 0 ' hotbutton 8
    BYTE WMF#MENU_STYLE_STATE_ACTIVE, 19, "ALL OFF", 0 ' hotbutton 9

' end HOTBAR -----

```

The LED hotbar controls four LEDs (red, green, yellow, orange). Clicking the ON/OFF buttons on the hotbar with a mouse will control the state of the LEDs. Once again, the main loop reads the messages from the message queue and when it sees a message from the LED control menu it calls an event handler that knows how to turn LEDs on and off.

To test-drive this, hook up four LEDs to a set of I/Os on your Propeller development board. Ideally, they should be red, green, yellow, orange to match the GUI, but they can be any color. Hook up the LEDs to the four I/O pins using a simple resistor circuit as shown in the listing.

```

' LED I/O pins, hook an LED up to each I/O pin with the circuit shown below
' (ideally with the same colors!)
LED_RED_PIN    = 1
LED_GREEN_PIN  = 2
LED_YELLOW_PIN = 3
LED_ORANGE_PIN = 4 ' each LED can be directly driven by a Prop I/O pin,
                    ' but a 100-330 ohm resistor is desired to limit current
                    ' as shown in the circuit example below:

{{
    LED Output Pin  --- R1= 100-300 ohm ---> GND
}}

```

The I/O pins on the Propeller Demo Board header support P0...P7, so P1 to P4 were used for this demo. Simply change these to whatever your Propeller development board has available and wire up each LED as shown above.

### The Motor Speed Control Menu

The final GUI element in the demo is a DC motor controller. For this GUI control, a hotlist menu was selected for its vertical orientation since setting speeds seems to fit more with an increasing list from top to bottom. The menu declaration is shown below:

```

' begin HOTLIST -----
motor_hotlist LONG ' start at LONG boundary
               BYTE WMF#MENU_STYLE_TYPE_HOTLIST ' type of menubar
               BYTE WMF#ATTR_DRAW_NORMAL | WMF#ATTR_DRAW_BORDER ' rendering attributes

```

```

BYTE WMF#ASCII_VLINE          ' space filler character
BYTE 2,15,26                  ' x,y,width of menu
BYTE 7                          ' number of menu items
BYTE "Motor Speed Control Menu", 0 ' optional menu title string

' list of menu titles for the menubar, each consists of the style flags, id,
' the string, and a null terminator at end
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 30, "0% Duty (Off)", 0 ' hotlist item 0
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 31, "50% Duty", 0 ' hotlist item 1
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 32, "60% Duty", 0 ' hotlist item 2
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 33, "70% Duty", 0 ' hotlist item 3
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 34, "80% Duty", 0 ' hotlist item 4
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 35, "90% Duty", 0 ' hotlist item 5
BYTE WMF#MENU_STYLE_STATE_ACTIVE, 36, "100% Duty (Max)", 0 ' hotlist item 6

' end HOTLIST -----

```

The motor GUI element was trivial to implement; the real work was in the event handler that generates the DC motor signal along with the electronics to drive a motor. (The simple motor driver, based on the AN001 Counters application note<sup>[4]</sup>, came from a search of the Parallax Object Exchange). The idea here is that we will pulse the DC motor with a PWM stream of 3.3 V and 0 V and as the duty cycle of the PWM signal changes, the average voltage or total current into the DC motor can be controlled.

The simple driver circuit below takes the PWM signal from a Propeller I/O pin and allows it to drive a device in the 100–200 mA range, such as the small DC motor used here. Remember, the Propeller itself can't source or sink that much current, so some form of amplification/driver circuit is needed. The code listing below shows the circuit design as well as the default I/O pin used for the motor driver output:

```

' DC motor 3-6V, 100-200mA current at most, so regulator can handle it, WMF signal will be
' generated to control the motor

DC_MOTOR_PIN = 6 ' the demo drives the motor with WMF directly if you have a
                  ' small 3-6V motor that doesn't require more than 100-200mA
                  ' you can drive it directly with the circuit shown below:

{{
    5V <-- ideally 5-6V, most Propeller boards have a 5V line, use it.

    DC motor ( )
    R1=100ohm
    WMF Output Pin --- R1 ---> Q1
    Q1 = 2N2222/2N3904 NPN general purpose transistor
        (beta/Hfe ~150-300) used as motor load switch
    D1 = 1N4001 diode/rectifier (for inductive kickback)
    C1 = 220-1000uF energy storage cap (recommended)
    GND
    GND
    GND
}}

```

The circuit works as follows: when the PWM signal comes in at the base of Q1 through R1, it turns on transistor Q1 and Q1 conducts hard. As Q1 conducts, current flows from the 5 V supply, through the motor, then through the collector-emitter junction of Q1 to ground. When the PWM signal falls back to 0 V, Q1 stops conducting; however, the coil in the DC motor is still energized and that energy flows through diode D1 and doesn't damage Q1. Additionally, C1 stores charge, so as high-current peaks are needed by the motor it reduces the spikes on the DC power rails and supplies the current itself without disturbing the rest of the circuitry.

If you are so inclined, hook up the motor circuit and set the `DC_MOTOR_PIN` to a free I/O on your Propeller development board. As the demo runs, select different duty cycles with the GUI control menu, which in turn will change the duty cycle on the PWM driver, resulting in the motor changing speed. (It's actually fun and highly recommended.) Again, this is done in the main event loop by filtering messages from the DC motor control menu and then passing them to the DC motor event handler that knows how to drive the motor.

The main idea of this demo was not only to show how to use all the different GUI elements in the same application, but more importantly how to augment the main event loop with message filters that parse and send messages to the appropriate message handlers for each control. Let's now formalize the concept of message handler now that a concrete example has been reviewed.

## Processing Messages and Events

The final subject of discussion is message processing and event handling. We have waited until the end of the application note, so you had as much context as possible, plus this discussion uses the previous multi-control demo as the generic model, so we can re-use some content. With that in mind, let's jump right into it with Figure 17.

**Figure 17: The GUI and Event-handling Model for the Multi-control Demo of Example 4**

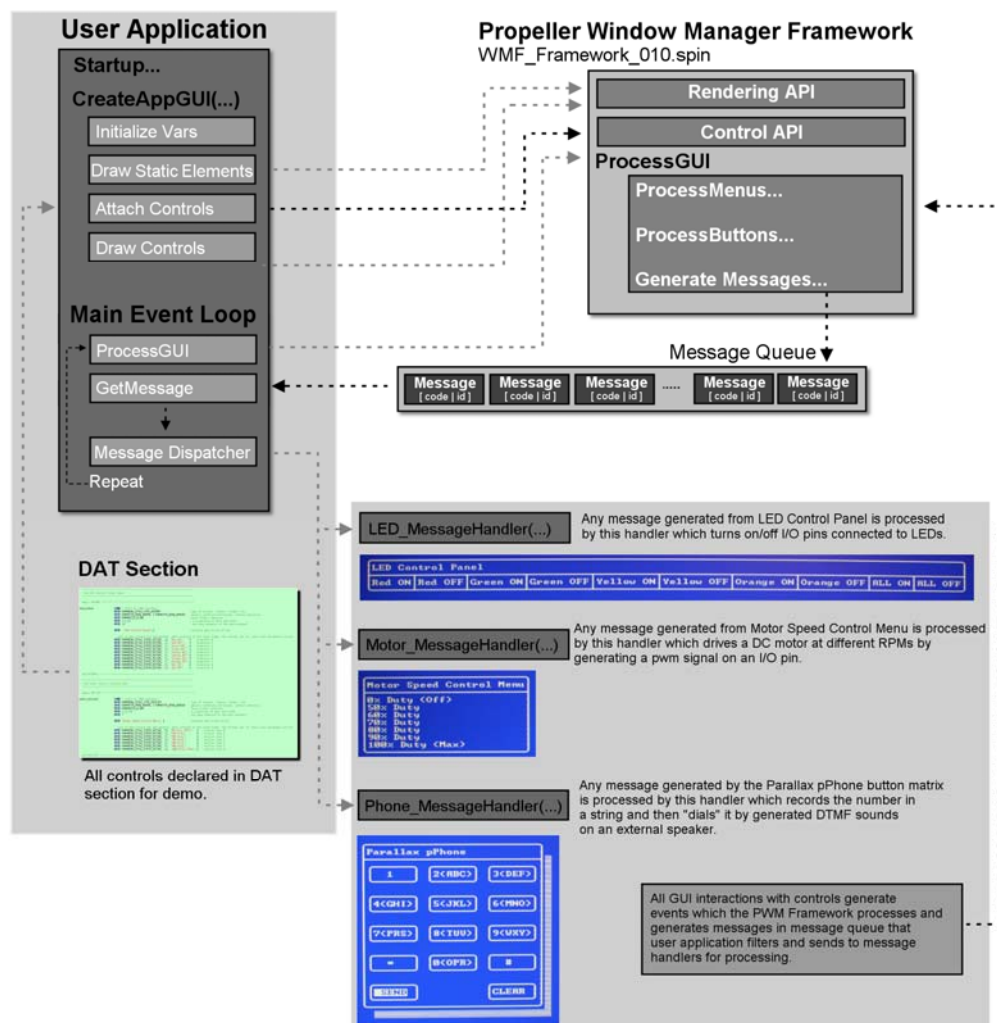




Figure 17 depicts the software model for the previous multi-control demo, but serves as a good general model for all event-handling programming. Referring to the figure, there are a number of GUI controls (hotbar, hotlist, and a matrix of buttons). These controls were declared in the **DAT** section of the program, attached to the WMF, and drawn once initially. Then once the main event loop starts and the loop makes calls to **ProcessGUI** the action begins.

As the WMF processes the GUI elements from its entry point in **ProcessGUI**, the user's interactions with the controls (pressing the buttons and selecting menu items) generates messages. These messages are then placed into the message queue for the top-level user application to process.

The user application retrieves the messages with calls to **GetNumMessages** and **GetMessage**. This is where the action takes place. Every single message represents an action performed by the user with some intent as he interacted with the GUI. Since we wrote the top-level application and created all the controls and assigned the IDs to each control, we know exactly what each control refers to based on its ID. Therefore, we can take action by determining what ID a message refers to and then sending that message to the appropriate message handler.

Referring back to Figure 17, there are three main controls in need of event handling:

- The LED Control Panel (control IDs [10...19] inclusive)
- The Motor Speed Control Menu (control IDs [30...36] inclusive)
- Parallax pPhone Keypad (control IDs [61...74] inclusive)

Handling events really means filtering these messages, so that message IDs for each control are routed to a single handler method that can process the messages. The code excerpt below shows how the routing is done in the demo:

```
' process all LED hotbar messages, we know ids are from 10 to 19 inclusive
if ( (wmf_message_id => 10) AND (wmf_message_id <= 19))
    call handler
    LED_MessageHandler ( wmf_message_code, wmf_message_id )

' process all motor hotlist messages, we know ids are from 30 to 36 inclusive
if ( (wmf_message_id => 30) AND (wmf_message_id <= 36))
    call handler
    Motor_MessageHandler ( wmf_message_code, wmf_message_id )

' process all phone keypad button matrix messages, we know ids are from 61 to 74
  inclusive
if ( (wmf_message_id => 61) AND (wmf_message_id <= 74))
    call handler
    Phone_MessageHandler ( wmf_message_code, wmf_message_id )
```

So, a simple **if** conditional tests the value of the ID, and then based on the results, passes the whole message intact to the appropriate event handler. This is the model to follow, though of course a **case** statement would also work. The point is that the main event loop in the **start** method does **not** process events, it simply retrieves them and then sends them to the appropriate handlers.

Once in the handlers, the message code and ID are used to perform the appropriate action and that's not our concern—it is no longer GUI code, but application code. This is a very

important delineation. Once the message is sent to the event handler, the user application code takes over and does whatever it needs to.

Of course, the event handlers are free to draw static items, update the GUI, and even delete GUI items. For example, an application might have multiple screens: one screen is the main menu, and selecting a main menu item opens up one of ten different sub-screens each with its own GUI. There is no need to create all ten GUIs at the same time, so the main application would only create the single main menu GUI, let the user interact, then once the user selects a sub-screen (and another GUI) detach the main menu GUI and attach the selected working GUI, and so forth. This way the WMF is processing the least amount of controls at once time, conserving memory.

## Developing Event Handlers

Although event handlers have nothing to do with GUI programming there are still some guidelines to follow so the GUI application runs smoothly:

- Entering an event handler starves the main event loop and the GUI processing calls to **ProcessGUI** since the WMF is currently a single-cog application. As a result, the user might try to interact with the GUI and nothing will happen. Therefore, try to do the processing quickly and return to the main event loop. If it needs to perform a lot of work, then start another cog and let it do the work and return immediately to the main event loop.
- Write robust event handlers that have a contingency plan for an unexpected message code or ID—the method should exit gracefully and not crash. In other words, event handlers should have a default catch-all at the end that processes any messages that weren't explicitly processed.
- If an event handler makes changes to the GUI itself (deleting controls, drawing on the screen etc.) be careful not to cause problems with any remaining event handlers that might be called in the current iteration of the main event loop. If a change is made to a control on the GUI surface and then another event handler down the line tries to do the same thing, it could create a problem, bug, or crash.
- Sometimes it is valid to ignore messages and just flush out the message queue. The WMF has a method to do this: **FlushMessages**. Calling this method clears out the message queue.
- In some cases, it's necessary to cheat a little and insert messages into the message queue to simulate an event—also perfectly valid in event-based systems. For example, to develop a word processor that supports macros, you might record all the messages a set of user interactions generates and then by use of a keyboard hotkey insert these messages manually into the message queue. Then the main event loop won't know the difference and the proper event handlers will be called as if the user generated the messages himself. Use **PostMessage** to do this.

## Summary

This application note has covered the fundamentals of the Propeller Window Manager (WMF) as a foundation to using the demos and examples as starting points for complex GUI applications. Additionally, the basic mechanics of the framework object itself have been discussed enough that hopefully you can go into the primary file `WMF_Framework_010.spin` and add new controls if you need them. However, buttons, menus, static text and frames

should be enough to create the majority of GUI applications embedded systems require, and what's been covered here should provide a good head start to creating more complex graphical applications.

## API Listing

The following is a complete API functional listing for WMF\_Framework\_010.spin. Look to the source code for more details and complete parameter descriptions for each of these public methods.

### Initialization Method

**Init**( pVGABasePin, pMouseDataPin, pMouseClkPin, pKeyboardDataPin, pKeyboardClkPin, pBackBufferPtr ) — Initializes the framework, VGA driver as well as basic terminal parameters.

### Message Queue Methods

**PostMessage**( pMessage ) — Inserts a message into the message queue.

**GetMessage** — Removes a message from the message queue.

**PeekMessage** — This method "peeks" the current message at the head of the message queue, but doesn't remove it.

**GetNumMessages** — Returns the number of messages in the message queue.

**FlushMessages** — Flushes out the message queue and resets everything.

### GUI and Window Management Methods

**ProcessGUI** — This is the workhorse of the GUI system that processes all the controls. It must be called every iteration of the main event loop.

**AttachButton**( pButtonFlags, pButtonPtr ) — Attaches a button to the GUI.

**DetachButton**( id ) — Detaches a button from the GUI.

**DrawButton**( pButtonId, pSelected, pCommand ) — Draw a button on the screen.

**ProcessButtons** — Processes all the buttons attached to the GUI.

**AttachMenu**( pMenuFlags, pMenuPtr ) — Attaches a menu to the GUI.

**DetachMenu**( id ) — Detaches a menu from the GUI.

**ProcessMenus** — Processes all the menus attached to the GUI.

**DrawHotMenu**( pMenuId, pSelHotButton, pCommand ) — Draws hotbar and hotlist menus to the screen.

### Direct Frame Buffer Methods for General Rendering for Console and Controls

**PrintString**( pStrPtr, pCol, pRow, pInvFlag ) — Draws a string directly to the frame buffer.

**PrintChar**( pChar, pCol, pRow, pInvFlag ) — Draws a character directly to the frame buffer avoiding the terminal system.

**ClearScreen**( pFGroundColor, pBGroundColor ) — Clear the screen at the memory buffer level, very fast.

**SetLineColor**( pRow, pFGroundColor, pBGroundColor ) — Sets the sent row to the given foreground and background color.

**DrawFrame**( pCol, pRow, pWidth, pHeight, pTitlePtr, pAttr, pVgaPtr, pVgaWidth ) — This method draws a rectangular "frame" with title and shadows.

## Mouse and Keyboard “Driver Pass-through” Methods

**GetMouseXYButtons** — Returns the mouse state to the caller.

**KeyboardPresent** — Checks if the keyboard is present.

**KeyboardKey** — Gets a key from keyboard, no waiting.

**KeyboardGetKey** — Gets a key from keyboards, waits for key press.

**KeyboardNewKey** — Clears keyboard buffer and gets a new key.

**KeyboardGotKey** — Checks if there is a key waiting in keyboard buffer.

**KeyboardClearKeys** — Clears the keyboard buffer.

**KeyboardKeystate** — Returns the state of a particular key.

## Text Console Terminal Methods

**StringTermLn( pStringPtr )** — Prints a string to the terminal and appends a newline.

**StringTerm( pStringPtr )** — Prints a string to the terminal.

**DecTerm( pValue, pDigits )** — Prints a decimal number to the screen.

**HexTerm( pValue, pDigits )** — Prints the sent number in hex format.

**BinTerm( pValue, pDigits )** — Prints the sent value in binary format with 0's and 1's.

**NewlineTerm** — Moves the terminal cursor home and outputs a carriage return.

**PrintTerm( pChar )** — Prints the sent character to the terminal console with scrolling.

**OutTerm( pChar )** — Outputs a character to the terminal. This is the primary interface from the client to the driver in terminal mode.

**SetColTerm( pCol )** — Sets terminal x column cursor position.

**SetRowTerm( pRow )** — Sets terminal y row cursor position

**GotoXYTerm( pCol, pRow )** — Sets the x column and y row position of terminal cursor.

**GetColTerm** — Retrieves x column cursor position.

**GetRowTerm** — Retrieves y row terminal cursor position.

## String and Numeric Conversion Methods

**StrCpy( pDestStrPtr, pSourceStrPtr )** — Copies the NULL terminated source string to the destination string and null terminates the copy.

**StrUpper( pStringPtr )** — Converts the sent string to all uppercase.

**ToUpper( pChar )** — Returns the uppercase of the sent character.

**IsInSet( pChar, pSetStringPtr )** — Tests if sent character is in sent string.

**IsSpace( pChar )** — Tests if sent character is white space, cr, lf, space, or tab.

**IsNull( pChar )** — Tests if sent character is NULL, 0.

**IsDigit( pChar )** — Tests if sent character is an ASCII number digit [0..9], returns integer [0..9]

**IsAlpha( pChar )** — Tests if sent character is in the set [a...zA...Z].

**IsPunc( pChar )** — Tests if sent character is a punctuation symbol !@#\$%^&\*()--+={ } [ ] \ ; : " , < . > / ?

**HexToDec( pChar )** — Converts ASCII hex digit to decimal.

**HexToASCII( pValue )** — Converts a number 0..15 to ASCII 0...9, A, B, C, D, E, F.

**itoa(pNumber, pBase, pDigits, pStringPtr)** — "C-like" method that converts pNumber to string; decimal, hex, or binary formats.

**atoi( pStringPtr, pLength )** — "C-like" method that tries to convert the string to a number, supports binary %, hex \$, decimal (default).

## Time Methods

**DelayMilliSec( pTime )** — Delays pTime milliseconds and returns.

**DelayMicroSec( pTime )** — Delays pTime microseconds and returns.

## Resources

A zip archive of the following demo program files is available from this application note's page at [www.parallaxsemiconductor.com/an013](http://www.parallaxsemiconductor.com/an013).

- WMF\_ButtonDemo\_010.spin — Button demo
- WMF\_ButtonDemo2\_010.spin — 2<sup>nd</sup> Button demo
- WMF\_HotBarDemo\_010.spin — Hotbar demo
- WMF\_HotListDemo\_010.spin — Hotlist demo
- WMF\_MultiControlDemo\_010.spin — All controls plus real world output demo
- WMF\_Framework\_010.spin — The WMF framework itself
- WMF\_TemplateDemo\_010. — Template
- VGA\_HiRes\_Text\_010.spin — VGA driver used by framework
- Mouse\_011.spin — Standard Parallax mouse driver
- Keyboard\_011.spin — Standard Parallax keyboard driver
- NS\_sound\_drv\_052\_11khz\_16bit.spin — HYDRA<sup>[5]</sup> sound driver
- pwmasm\_010.spin — Simple pulse width modulation object based on AN001<sup>[4]</sup>

## References

1. Parallax Application Note: AN004 Getting Started with VGA and Terminal Output; [www.parallaxsemiconductor.com/an004](http://www.parallaxsemiconductor.com/an004)
2. Parallax Application Note: AN005 Simple VGA Menus; [www.parallaxsemiconductor.com/an005](http://www.parallaxsemiconductor.com/an005)
3. Propeller Demo Board. Parallax part #32100; [www.parallax.com](http://www.parallax.com)
4. Parallax Application Note: AN001 Propeller Counters; [www.parallaxsemiconductor.com/an001](http://www.parallaxsemiconductor.com/an001)
5. Game Programming for the Propeller Powered HYDRA; Parallax Press. Parallax part #70360; [www.parallax.com](http://www.parallax.com)

## Revision History

Version 1.0: original document.

---

Parallax Inc., dba Parallax Semiconductor, makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Parallax Inc., dba Parallax Semiconductor, assume any liability arising out of the application or use of any product, and specifically disclaims any and all liability, including without limitation consequential or incidental damages even if Parallax Inc., dba Parallax Semiconductor, has been advised of the possibility of such damages. Reproduction of this document in whole or in part is prohibited without the prior written consent of Parallax Inc., dba Parallax Semiconductor.

Copyright © 2011 Parallax Inc. dba Parallax Semiconductor. All rights are reserved.  
Propeller and Parallax Semiconductor are trademarks of Parallax Inc. All other trademarks herein are the property of their respective owners.