AN1246

How to Create Widgets in Microchip Graphics Library

Authors: *Paolo Tamayo*
Harold Serrano
Microchip Technology Inc.

INTRODUCTION

The Microchip Graphics Library consists of various, readily usable Widgets which include the functions required in various applications. The Widgets can be customized for size, colors and the type of fonts used to adapt to the overall appearance and functions of the application.

In some cases, the standard Widgets might not function in a manner as desired by the application designer. Customized attributes (or even new Widgets) may be needed for the application designer to achieve the optimal GUI design. For example, a designer could use a slider to indicate fluid levels; however, the user may not fully understand what the slider is for unless an icon or label is added to the screen.

In another example, one could implement a security keypad console using the library's standard Widgets. This is done by using a set of buttons along with an edit box for text entry. If this same application requires the keypad configuration to change on-the-fly, using the standard Widgets might be more difficult. In such situations, it becomes necessary to design specific Widgets which can perform functions to make the application more efficient.

This application note serves as a useful guide in creating customized Widgets. The essential components of a Widget are enumerated and described in this document. This application note also outlines the process of integrating the new Widget into the Graphics Library in order to utilize the already implemented routines for processing messages and rendering Widgets.

This application note is an advanced topic for the users of the Microchip Graphics Library. Most of the applications can be created using the standard Widgets that come along with the installation of the library.

Note: The users of standard Widgets can skip reading this document.

For users who intend to create their own Widgets, it is assumed that they are familiar with the structure and operation of the Microchip Graphics Library and C programming syntax. For more information on the Microchip Graphics Library, refer to the `Graphics Library Help File` that is included in the most current version of the library and the other relevant application notes on the library. The library can be downloaded from www.microchip.com/graphics.

AN EXAMPLE WIDGET

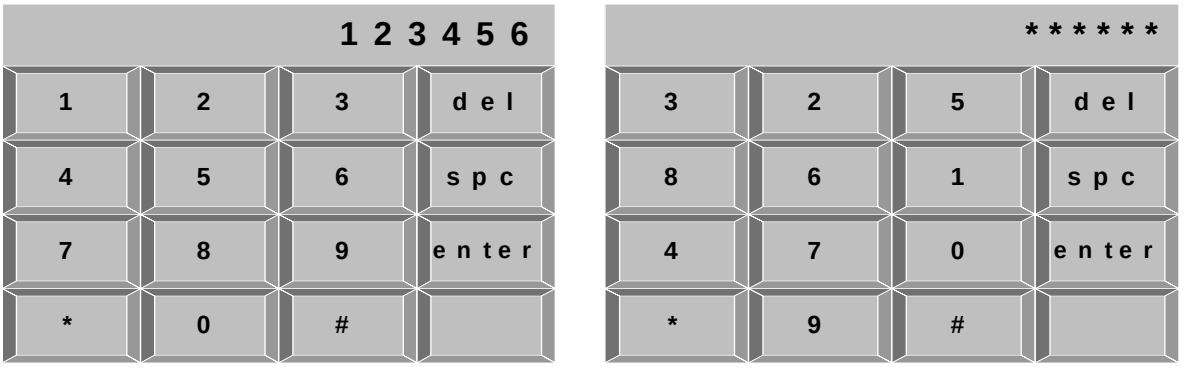
To aid in the discussion of the Widgets, consider the example of a key entry user interface commonly used in security systems. The function of this interface is to receive inputs from the user using the available alphanumeric keys. Since this is used for security systems, the important application specifications are as follows:

- Provide a versatile interface where the number of keys and the characters assigned to each key can be dynamically changed.
- Provide an option to display the * character in the display screen instead of the actual keyed in characters.

One mode is shown on the left keypad of Figure 1 and the other is shown on the right keypad, where the characters entered are replaced with the * character and the locations of the numbers are randomized.

The user interface can be created with all the important features using the standard Button and Edit Box Widgets of the Graphics Library. The drawback of using the standard Widgets is slower rendering, larger code and more RAM space usage. On the other hand, the newly implemented Widget can take advantage of the already existing rendering and messaging infrastructure in the Graphics Library, resulting in a faster rendering with less code.

FIGURE 1: SECURITY KEYPAD APPLICATION



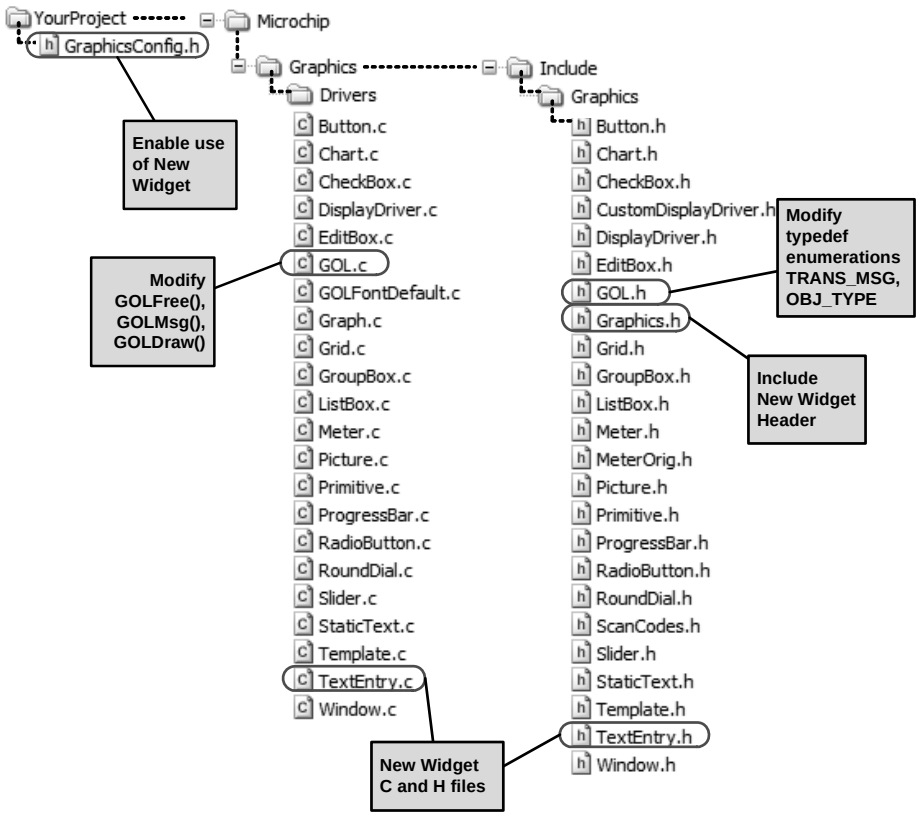
Before examining the details of the requirements, download the latest version of the Microchip Graphics Library from www.microchip.com/graphics. Installing the library with the default installation directory will place the library files and demos under the "Microchip Solutions" directory. The files, `Graphics.h`, `GOL.c`, `GOL.h`, `TextEntry.c` and `TextEntry.h`, found in the directory, are the code files referred to in this document. Refer to these files and use them as a guide while reading the steps described in this application note. The application's directory, named AN1246, contains the application-specific code for implementing the Widget described in the example.

The remaining sections of the document describe:

- Components of a Widget
- Files that need to be edited in the Graphics Library
- Application Programming Interfaces (APIs) that are implemented to fully integrate the Widget into the Graphics Library

Figure 2 displays the files that are to be created and/or modified if you wish to integrate the new Widget into the Graphics Library.

FIGURE 2: FILES TO BE CREATED AND MODIFIED



COMPONENTS OF A WIDGET

To create a new Widget for use in the Microchip Graphics Library, the following steps are performed:

- A new C file and a header file are created for the Widget.
- Modifications are made to the GOL.c, GOL.h and Graphics.h files of the library.

The changes to the Graphics Object Layer (GOL) files are discussed in detail in the subsequent sections of this document. Similar to the standard Widgets, the new Widget must be enabled in the application implemented in the GraphicsConfig.h file. Table 1 lists and describes the different components of a Widget. It also serves as a summary of needed items during the implementation of a new Widget.

TABLE 1: COMPONENTS OF A WIDGET

Name	Syntax	Description
Widget Structure	For example, BUTTON, SLIDER	The main structure of the Widget that holds the user-assigned unique ID, dimension, state, style scheme details and Widget specific parameters.
Create Function	ObjCreate()(1)	This function is used to create the Widget in memory.
Drawing Function	ObjDraw()	This function is called by GOLDraw() to render the Widget to the screen.
Translate Message Function	ObjTranslateMsg()(3)	This function is called by GOLMsg() to evaluate if the Widget is affected by the user action.
Message Default Function	ObjMsgDefault()(3)	An optional function, called by GOLMsg(), to perform a default action on the Widget. A Widget can also be created without any default action. If implemented, applications can enable or disable the call to this function by modifying the return value of the GOLMsgCallback() function.
Type Definition	OBJ_STRNAME(2)	The type definition of the Widget that should be added in GOL_OBJ_TYPE enumeration in the GOL.h file.
Use Object Definition	USE_STRNAME(2)	The compile switch definition to enable the use of the Widget. It is placed in the GraphicsConfig.h application file. If the definition is not present, the Widget and its corresponding routines in the library will be ignored in the compilation of the application.
Header File Inclusion	#ifdef USE_STRNAME #include "*.h" #endif	The Graphics.h header file is used to include all GOL files in the build of the application. The header file of the new Widget must be added in the Graphics.h file for the project build inclusion. The syntax indicates that the inclusion is enabled only when the Use Object definition is added in the application level GraphicConfig.h file.
Widget Specific APIs	ObjFunctionName()	These are the functions implemented for easy usage of the Widget. FunctionName is user-defined and should correspond to the operation of the API. For example, use BtnSetText() to set the string shown on the Button Widget.

Note 1: Obj represents the prefix assigned to a particular Widget. For example, Btn is used for the Button Widget.
2: STRNAME represents the name of the Widget structure; for example, BUTTON for a Button Widget (i.e., OBJ_BUTTON or USE_BUTTON).
3: Direct application code calls to ObjTranslateMsg() and ObjMsgDefault() functions are discouraged. Use of GOLMsg() is recommended to simplify the use of messaging in the Graphics Library.

This document explains how a Widget is implemented to support the text entry interface application mentioned earlier. Since the functionality of the Widget expects inputs from the user through the displayed keys on the screen, the Widget is named Text Entry Widget. The 'Te' is used as the abbreviation to prefix the standard function names, and TEXTENTRY is used for macros that need the name spelled out.

Note:	The completed version of the Widget is already included in the installation of the Graphics Library with version number 1.60 or later. The code shown here is for your reference and guidance.

For your own purpose, you can name your new Widget whatever is the most applicable for your application.

The summary of the generalized prefixes and names for the Text Entry Widget components is shown in Table 2.

TABLE 2: TEXT ENTRY WIDGET COMPONENTS

Name	Syntax
Widget Structure	TEXTENTRY
Create Function	TeCreate()
Drawing Function	TeDraw()
Translate Message Function	TeTranslateMsg()
Message Default Function	TeMsgDefault()
Type Definition	OBJ_TEXTENTRY
Use Object Definition	USE_TEXTENTRY
Widget Specific APIs	TeFunctionName()

WIDGET STRUCTURE

The TEXTENTRY is the structure name we have chosen for the new Text Entry Widget. For use within the Graphics Library, the new Widget's structure must be defined in the TextEntry. h file.

EXAMPLE 1: TEXTENTRY STRUCTURE

```
typedef struct
{
        OBJ_HEADER hdr;           // Generic header for all Objects.
        WORD UsrDefMember1;       // Optional User defined Widget parameter.
        WORD UsrDefMember2;       // Optional User defined Widget parameter.
        …
        SHORT UsrDefMemberN;      // Optional User defined Widget parameter.
} TEXTENTRY;
```

The number of user-defined Widget parameters depends on the desired functionality of the new widget. The parameters are selected to make the operation of the Widget optimized and efficient. The selection of parameters can be done by identifying the support APIs that will be used with the Widget.

For example, the Text Entry Widget expects a versatile interface where the number of keys and the characters assigned to each key can be dynamically changed. We could assign a parameter to define the number of keys. However, since the arrangement of the keys can be dynamically changed, we can further simplify the code if we assign the number of horizontal and vertical keys. Thus, in the example, we will see the verticalkeys and horizontalkeys declared in the structure.

The data types for these user-defined parameters can be of any type and depend on their usefulness to the Widget. For example, one parameter can be a pointer, while the rest can be characters, or signed or unsigned integers.

The OBJ_HEADER structure is already defined in the library. This structure, which is the first member of the Widget structure, defines:

- Unique ID set by the user
- State variable that helps in rendering the Widget
- Dimensions (left, right, top and bottom) that define the position and size of the Widget
- Pointer to the next Widget
- Style scheme used in the Widget

EXAMPLE 2: OBJECT_HEADER STRUCTURE

```
typedef struct
{
        WORD ID;                  // Unique id assigned for referencing.
        void *pNxtObj;            // Pointer to the next object.
        GOL_OBJ_TYPE type;        // Identifies the type of GOL object.
        WORD state;               // State of object.
        SHORT left;               // Left position of the Object.
        SHORT top;                // Top position of the Object.
        SHORT right;              // Right position of the Object.
        SHORT bottom;             // Bottom position of the Object.
        GOL_SCHEME *pGolScheme;   // Pointer to the scheme used.
} OBJ_HEADER;
```

The type parameter must be added to the GOL_OBJ_TYPE enumeration in the GOL.h file, as shown in Example 3. The format used is OBJ_WIDGETNAME; therefore, for the Text Entry Widget, the parameter is OBJ_TEXTENTRY.

EXAMPLE 3: OBJECT TYPE ENUMERATION

```
typedef enum
{
        OBJ_BUTTON,             // Type defined for Button Object.
        OBJ_WINDOW,             // Type defined for Window Object.
        OBJ_CHECKBOX,           // Type defined for Check Box Object.
        OBJ_RADIOBUTTON,        // Type defined for Radio Button Object.
        OBJ_EDITBOX,            // Type defined for Edit Box Object.
        …
        OBJ_GRID,               // Type defined for Grid Object.
        OBJ_CHART,              // Type defined for Chart Object.
        OBJ_TEXTENTRY,          // ADD your widget object in the list of object types
        OBJ_UNKNOWN
} GOL_OBJ_TYPE;
```

Example 4 defines the Text Entry Widget structure. As explained earlier, the OBJ_HEADER is required and is used to define the dimensions, states and style of the Widget. Additional structure members are added to support the different features of the Widget. The horizontal and vertical keys are added to define the number of keys. Having these two key parameters allows the Widget to have variable vertical and horizontal key counts. The string displayed on the Widget will be defined with the maximum characters that it can hold and an option to change the font used to display the inputs. The limit on the string length simplifies the management of the string buffer pointer, pTeOutput, and the addition of a separate font for the display allows the versatility of having different character sizes for the key characters and the display.

EXAMPLE 4: TEXT ENTRY WIDGET STRUCTURE

```
typedef struct
{
        OBJ_HEADER hdr;           // Generic header for all objects
        SHORT horizontalkeys;     // Number of Horizontal keys
        SHORT verticalkeys;       // Number of Vertical Keys
        XCHAR *pTeOutput;         // pointer to the buffer assigned by the user
                                  // which holds the text shown in the editbox
        WORD length;              // length of array, keeps track if length<MaxSize
        WORD teOutputLen;         // Maximum length of the buffer pTeOutput
        KEYMEMBER *pActiveKey;    // pointer to the currently active key in the list
        KEYMEMBER *pHeadOfList;   // pointer to the list of KEYMEMBER list
        void *pEditBoxFont;       // pointer to the font to be display on the editbox
}TEXTENTRY;
```

To dynamically assign the characters or strings to a key, an additional structure, KEYMEMBER, is created to hold the information of each key. The pKeylist Pointer is used to refer to the list of KEYMEMBERs. This enables the application to dynamically change the characters assigned to the keys, and thus, supports the feature to change the keys or randomize the key positions.

In this type of implementation, each key has a corresponding structure to describe:

- Position on the screen
- Associated characters or strings
- Current state (pressed or released state)

The application needs to change the linked list to change the characters or strings associated with each key. Implementing the key attributes in a linked list also facilitates the easy processing of the data.

The decision to add additional structures is dependent upon the desired functionality of the Widget.

EXAMPLE 5: KEYMEMBER WIDGET STRUCTURE

```
typedef struct
{
        SHORT left;               //left position of the key
        SHORT top;                //top position of the key
        SHORT right;              //right position of the key
        SHORT bottom;             //bottom position of the key
        SHORT index;              //index of the key in the list
        WORD state;               //State of the key. Either Pressed or Released.
        WORD update;              //flag to indicate key needs to be redrawn
        WORD command;             //command of the key
        XCHAR *pKeyName;          //text assigned to the key
        SHORT textWidth;          //computed text width, done at creation.
        SHORT textHeight;         //computed text height, done at creation.
        void *pNextKey;           //pointer to the next text
}KEYMEMBER;
```

ASSIGNING STATE BITS

The Widgets in the Graphics Library are rendered and controlled directly by the state variable. Each bit of the state variable is interpreted as state bits, where each bit can be assigned to a specific state. Apart from the required state bits, you will also need to define state bits specific to your Widget. The state variable is composed of two main components:

- Drawing State Bits – Indicates if the object needs to be hidden, partially redrawn or fully redrawn in the display.
- Property State Bits – Defines action and appearance of the objects.

The six Most Significant bits (MSbs) are allocated for drawing states and the remaining bits are allocated for the property states. Some common property and drawing state bits are given in Table 3. Apart from the optional focus feature controlled by `OBJ_FOCUS` and `OBJ_DRAW_FOCUS`, the rest of the bits must be implemented. Focus is an optional feature of a Widget where a dashed rectangle is drawn over the face of the Widget to indicate that the Widget is currently focused. This feature provides feedback to users in systems where touch screen is not supported and Widget selection is done through other means, such as a physical button or a switch. The bit mask locations of the state bits are also reserved to these states. The new Widgets must not use these mask bits for other purposes aside from the indicated usage given in Table 3.

TABLE 3: COMMON WIDGET STATE BITS

State Name	Type	Bit Mask Location	Description
`OBJ_FOCUSED`	Property	0x0001	The Widget is in the focused state. This is usually used to show the selection of the object. Not all objects have this feature.
`OBJ_DISABLED`(1)	Property	0x0002	The Widget is disabled and will ignore all messages. The Widget is redrawn using the disabled colors defined in the style scheme of the Widget.
`OBJ_DRAW_FOCUS`	Drawing	0x2000	The focus for the object is redrawn.
`OBJ_DRAW`(1)	Drawing	0x4000	The Widget is redrawn completely.
`OBJ_HIDE`(1)	Drawing	0x8000	The Widget is hidden by filling the area occupied by the Widget with the common background color. This has the highest priority over all drawing states. When a Widget is set to be hidden, all other drawing states are overridden.

Note 1: These state bits are required for all of the Widgets.

Drawing State Bits

Apart from the common drawing state bits, three more drawing state bits are available for use. They are bit 10, bit 11 and bit 12 of the state variable (0x0400, 0x0800 and 0x1000). These bits can be used depending on the characteristics of the new Widget.

The required drawing state bits for the Text Entry Widget are defined in Table 4.

TABLE 4: DRAWING STATE BITS FOR TEXT ENTRY WIDGET

State Name	Bit Mask Location
`TE_HIDE`	0x8000
`TE_DRAW`	0x4000
`TE_UPDATE_KEY`	0x2000
`TE_UPDATE_TEXT`	0x1000

The drawing state bits are used by the drawing function of the Widget to render or hide the Widget in the display. For our example Widget, TeDraw( ) will be the drawing function.

Applications can use the Widget specific draw function (i.e., TeDraw( )) to render or hide the widget; however, it is recommended that the more generic GOLDraw( ) function be used instead. This function is implemented in the GOL.c file. Application can set or reset the drawing state bits of the Widgets by calling the SetState( ) and ClrState( ) functions, and then calling GOLDraw( ) to automatically render or hide Widgets from the screen. GOLDraw( ) checks each drawing state of the Widget and performs the rendering or hiding of the Widget from the screen by calling its respective drawing functions.

To integrate the new Widget into the processing of GOLDraw( ), the function must be modified to include the processing of the drawing states of the new Widget. To achieve this, the code shown in Example 6 must be added to the switch statement of the GOLDraw( ) function.

EXAMPLE 6: GOLDRAW( ) MODIFICATION TO INCLUDE THE TEXT ENTRY WIDGET

```
….
while(pCurrentObj != NULL){
        if(IsObjUpdated(pCurrentObj)){
            switch(pCurrentObj->type){
                #ifdef USE_BUTTON
                    case OBJ_BUTTON:
                        done = BtnDraw((BUTTON*)pCurrentObj);
                        break;
                #endif
                …
                #ifdef USE_CHART
                    case OBJ_CHART:
                        done = ChDraw( (CHART *)pCurrentObj );
                        break;
                #endif
                #ifdef USE_TEXTENTRY
                    case OBJ_TEXTENTRY:
                        done = TeDraw( (TEXTENTRY *)pCurrentObj );
                        break;
                #endif
                    default:
                        break;
}
if(done){
            GOLDrawComplete(pCurrentObj);
}else{
            return 0; // drawing is not done
}
```

In Example 6, note the use of OBJ_TEXTENTRY which was added to the typedef enumeration for GOL_OBJ_TYPE in Example 3. The USE_TEXTENTRY compile switch is added to include or remove the Widget in the compilation of the application. The inclusion or exclusion of the Widget is done by defining USE_TEXTENTRY in the GraphicsConfig.h file. GOLDraw( ) automatically resets the drawing state bits once the TeDraw( ) function of the Widget, which will be implemented in TextEntry.c, returns a '1'. The Reset of the drawing state bits is performed by the GOLDrawComplete( ) function of the Widget.

Property State Bits

The property state bits are used to define the appearance and action of the Widget. For example, in Slider Widget, the SLD_VERTICAL state bit defines how the slider is drawn. If the bit is set, the slider is drawn vertically, and if not set, the slider is drawn horizontally. For the Button Widget, the BTN_PRESSED state bit indicates that the Button will be drawn with the light and dark emboss colors interchanged, emulating the pressed effect. When the bit is reset, it returns to the default assignment of the light and dark emboss colors, emulating the unpressed effect. Refer to the Slider.h and Button.h files for details and examples on property state bits.

As mentioned earlier, state bits can be set and reset using the corresponding APIs. For setting the state, use the SetState() API, and for resetting or clearing the state bits, use the ClrState() API. State bit status can be queried using the GetState() API. For consistency, these APIs are used to set, reset and query a state bit when implementing the drawing function of the Widget.

In the Text Entry Widget, the property state bits (lower 10 bits) are appended to the already identified drawing state bits (upper 6 bits) and are listed and described in Table 5.

The choice of the state bits is based primarily on the optimized operation of the Widget. For example, the drawing state bit, TE_UPDATE_KEY, will indicate that a key will be drawn as pressed or released. Similarly, the drawing state bit, TE_UPDATE_TEXT, will require an update on the text area only. If both of these state bits are set, then the key and the text area will be updated.

The property state bit, TE_KEY_PRESSED, on the other hand, will indicate that at least one of the keys is currently pressed. This will serve as a flag in the drawing function to check for the current status of the keys.

TABLE 5: STATE BITS OF TEXT ENTRY WIDGET (1)

State Name	Type	Bit Mask Location	Description
TE_DISABLED	Property	0x0002	The Widget is disabled and will ignore all messages. The Widget is redrawn using the disabled colors defined in the style scheme of the Widget.
TE_KEY_PRESSED	Property	0x0004	A key is pressed; this can refer to any key.
TE_ECHO_HIDE	Property	0x0008	Text entries echoed into the edit box are replaced by the * character.
TE_HIDE	Drawing	0x8000	The Widget is hidden by drawing the common background color over the object.
TE_DRAW	Drawing	0x4000	The Widget is redrawn completely.
TE_UPDATE_KEY	Drawing	0x2000	The Widget is partially redrawn. Only the key that is active is redrawn.
TE_UPDATE_TEXT	Drawing	0x1000	The Widget is partially redrawn. Only the text area is redrawn.

Note 1: The definitions of the state bits of the Text Entry Widget should be placed in the TextEntry.h file

STYLE SCHEME

The last member in the OBJ_HEADER structure is the Style Scheme Pointer. It is a pointer to the GOL_SCHEME structure which defines all the colors and fonts used in the Widget.

EXAMPLE 7: STYLE SCHEME STRUCTURE

```
typedef struct
{
        WORD  EmbossDkColor;          // Emboss dark color used for 3d effect.
        WORD  EmbossLtColor;          // Emboss light color used for 3d effect.
        WORD  TextColor0;             // Character color 0 used for objects that
                                      // supports text.
        WORD  TextColor1;             // Character color 1 used for objects that
                                      // supports text.
        WORD  TextColorDisabled;      // Character color used when object is in a
                                      // disabled state.
        WORD  Color0;                 // Color 0 usually assigned to an Object state.
        WORD  Color1;                 // Color 1 usually assigned to an Object state.
        WORD  ColorDisabled;          // Color used when an Object is in a disabled
                                      // state.
        WORD  CommonBkColor;          // Background color used to hide Objects.
        void  *pFont;                 // Font selected for the scheme.
} GOL_SCHEME;
```

The style scheme structure consists of nine colors and a Font Pointer. The ColorDisabled and TextColorDisabled are specifically used to assign colors when the Widget is in the disabled state. The CommonBkColor is used to assign the screen background color to hide the Widget on the screen. This is achieved by overlaying the Widget with the color assigned to the CommonBkColor. The size of the overlay is equal to the dimension of the Widget. EmbossDkColor and EmbossLtColor are specifically used to create the 3-D effect for the Widget.

The size of the emboss is a global setting and is defined by the GOL_EMBOSS_SIZE set in the GraphicsConfig.h file.

Additional color variables and fonts can be added to the Widget structure if the colors defined in the GOL_SCHEME do not satisfy the requirement of the Widget. If your Widget needs additional color variables or fonts, an API should be added to easily change these colors and fonts. For example, ChSetTitleFont() is used for the Chart Widget to set the font of the chart title.

CODING THE WIDGET

Now that the Widget structure components have been identified, it is time to code the different functions that will create, manage and delete the Widget.

Create Function

The “create function” allocates memory and initializes the structure members of the Widget. Allocation is the first task in the “create function”. This is done by using the `malloc` function.

If, for some reason, the system cannot allocate memory for the Widget, the “create function” must return a NULL. This gives the application layer a chance to perform a recovery operation if the Widget creation fails.

The next task in the “create function” is the initialization of the first nine and other user-defined parameters of the Widget structure.

EXAMPLE 8: INITIALIZING WIDGET MEMORY

```
TEXTENTRY *TeCreate(…) {
    TEXTENTRY *pTe = NULL;
        pTe = (TEXTENTRY*)malloc(sizeof(TEXTENTRY));
        if (pTe = NULL)
            return NULL;
…
}
```

EXAMPLE 9: INITIALIZING PARAMETERS

```
TEXTENTRY *TeCreate
(                               WORD ID,
                                SHORT left, top, right, bottom,
                                SHORT horizontalKeys,
                                SHORT verticalKeys,
                                XCHAR *pText,
                                WORD state,
                                void *pEditBoxFont,
                                GOL_SCHEME *pScheme)
{
    pTe->hdr.ID                 = ID;               // unique id for referencing
    pTe ->hdr.pNxtObj           = NULL;             // initialize pointer to NULL
    pTe ->hdr.type              = OBJ_TEXTENTRY;    // set object type
    pTe ->hdr.left              = left;             // left position
    pTe ->hdr.top               = top;              // top position
    pTe ->hdr.right             = right;            // right position
    pTe ->hdr.bottom            = bottom;           // bottom position
    pTe ->hdr.state             = state;            // state
    // Set the color scheme to be used
    if (pScheme = NULL)
        pMy ->hdr.pGolScheme = _pDefaultGolScheme;
    else
        pMy ->hdr.pGolScheme = (GOL_SCHEME *)pScheme;
    // add other parameter initializations code here
    …
}
```

Note that the Style Scheme Pointer, `pGolScheme`, is assigned to `_pDefaultGolScheme`. This is the default style scheme assigned to any Widget if no style scheme is assigned to the Widget at creation (i.e., `pScheme` = NULL).

Aside from the standard Widget structure members, the parameters of the “create function” are expanded to initialize the additional structure members that have been added earlier to implement the Widget’s specifications. Example 9 shows the additional parameters that are passed to the function. The sequence of parameters is not mandatory, but the sequence in the example is a typical sequence of parameters in the already implemented Widgets of the library. For details on usage of the parameters, refer to the `TextEntry.c` file downloaded with the Graphics Library.

After initializing the Widget parameters, the Widget must be added to the global active list of Widgets. This is done by calling the GOLAddObject() API.

EXAMPLE 10: ADDING THE TEXT ENTRY WIDGET TO THE ACTIVE LIST OF WIDGETS

```
TEXTENTRY *TeCreate(…){
        …
        GOLAddObject((OBJ_HEADER*) pTe);
        return pTe;
}
```

Note that the Object Pointer is casted to OBJ_HEADER. This is because all Widget management APIs operate on OBJ_HEADER pointers. Finally, the “create function” must return a pointer to the newly created Widget. If the pointer is not NULL, the Widget was successfully created.

Draw Function

Another function that needs to be created is the drawing function of the Widget. This drawing function should perform state-based rendering, which allows the Widget to support both blocking and non-blocking configurations. State-based rendering also helps with dividing the drawing function into the following tasks:

- Full and partial redraw of the Widget
- Drawing of focus if supported
- Hiding the Widget

Blocking and non-blocking configurations provide the Graphics Library with the ability to take advantage of hardware implemented rendering primitives. In blocking configuration, the drawing function will not exit until the Widget is drawn. It can be a complete rendering of the Widget or an update of a portion of the Widget. In a non-blocking configuration, the drawing function of the Widget can check the status of the primitive rendering functions implemented in the hardware, such as Line(), Bar() and Rectangle(). If the hardware is still busy executing the last called primitive rendering function, the drawing function exits and returns the processor control to the application. The drawing functions of the Widgets retain the state of the drawing flow. The next call to the draw function returns to the last primitive command to continue the rendering of the Widget. Non-blocking configuration will be used when USE_NONBLOCKING_CONFIG is defined in the GraphicsConfig.h file. If it is not defined, the library defaults to the blocking configuration.

In the non-blocking configuration, some of the primitive functions are now located in the driver layer as opposed to the primitive layer. More specifically, some of the primitive functions are implemented in the hardware of the display controller. Depending on the display controller, the hardware will perform one, two or all of the primitive rendering functions. Since the library can be used with any display controller, the drawing function of the Widget must check to see if the hardware is busy after every primitive rendering function call. This is done by calling the IsDeviceBusy() API. A TRUE is returned if the display controller hardware is busy. This API is actually a macro that may be modified to suit the hardware being used. It can also be modified to add system level control on the drawing functions of the Widgets.

For example, in applications using scheduled tasks, the scheduler can force the hardware to be busy to make sure that other tasks in the system get the needed processor time. Note that this is only possible in display controllers which have primitive rendering functions implemented in the hardware. The use of the IsDeviceBusy() API is shown in Example 11.

EXAMPLE 11: HARDWARE BUSY CHECK IN DRAWING FUNCTIONS OF THE WIDGETS

```
// this example code is just for illustration purposes only
WORD ExDraw( ){
        …
        case DRAW_STATE1:
                if(IsDeviceBusy())
                        return 0;
                SetColor(BLACK);
            Bar( left, top,right,bottom);
            state = DRAW_STATE2;
            break;
        case DRAW_STATE2:
                if(IsDeviceBusy())
                        return 0;
            // change left, top, right, bottom variables here
            Bar( left, top, right, bottom);
            state = DRAW_STATE3;
            break;
        …
}
```

In the DRAW_STATE1 case, IsDeviceBusy() returns a TRUE or '1' if the hardware is still busy with the last primitive rendering function call. In the case of DRAW_STATE2, the same hardware test is performed before the call to the next Bar() function. This can be any primitive function implemented in the hardware. If the hardware is indeed busy, the Widget drawing function should exit with a return value of '0'. If IsDeviceBusy() returns a FALSE or a '0', then the Draw() function of the widget can proceed in executing the next primitive function.

If the drawing function exits with '0', how can it recover and go back to the last unsuccessful primitive call? As mentioned earlier, the drawing function of the Widget must be implemented in a state-based manner. Example 12 shows one possible implementation. Each primitive rendering function can be assigned a state. A static rendering variable, "state", is used to keep track of the current state of the drawing flow. When the function exits, the variable maintains that state and when the drawing function of the Widget is called again, it can return to the last state and execute the next primitive function. Take note that the "state" variable mentioned here is referring to the static rendering state variable that controls the rendering flow of the Widget. This is not to be confused with the OBJ_STATE structure member "state" that refers to the state bits of the Widget.

EXAMPLE 12: STATE-BASED RENDERING CODE STRUCTURE

```
// this example code is just for illustration purposes only
WORD ExDraw( ){
typedef enum {
        DRAW_STATE0,
        DRAW_STATE1,
        DRAW_STATE2,
        DRAW_STATE3,
} MYW_DRAW_STATES;
        static MYW_DRAW_STATES state = DRAW_STATE0;
        static SHORT left, top, right, bottom;

        switch(state){
            case DRAW_STATE0:
                if(IsDeviceBusy())
                        return 0;
            ….
                state = DRAW_STATE1;
        case DRAW_STATE1:
                if(IsDeviceBusy())
                        return 0;
                SetColor(BLACK);
                Bar(left, top,right,bottom);
                // change left, top, right, bottom variables here
                state = DRAW_STATE2;
                break;
        case DRAW_STATE2:
                if(IsDeviceBusy())
                    return 0;
                Bar(left, top, right, bottom);
                state = DRAW_STATE3;
                break;
            …
}
```

Example 12 also shows where left, top, right and bottom variables, that define the area where the Bar( ) is drawn, can be modified. If the calculation is done in DRAW_STATE2 and the rendering state variable is at DRAW_STATE2, each time the Draw() function is called, more time is consumed if the hardware is always busy in that state. Having the location in DRAW_STATE1 (before the state is changed to DRAW_STATE2) optimizes the rendering flow. The four variables will maintain their values since they are declared as static variables inside the Widget's draw function.

The full redraw, partial redraw, drawing of the focus (if the Widget supports focus) and hiding of the Widget are all decided based on the drawing state bits of the Widget. Along with the property state bits, the overall look of the Widget is decided and drawn. To accommodate the additional rendering requirements of the Widgets, additional states must be implemented. An example of such requirements is the redrawing of the current pressed key to its unpressed state, and at the same time, redrawing of another unpressed key to its pressed state. This is a very common scenario when the user drags the touch across the keys without releasing the press on the screen. The additional drawing sequence can be just one state, as in the case of hiding the Widget, or a series of states to fully implement a multiple key press/release redrawing sequence. The final number of needed states in the rendering of the Widget will depend on the features and behavior of the Widget being implemented.

EXAMPLE 13: ADDITION OF HIDE AND FOCUS DRAWING STATES

```
// this example code is just for illustration purposes only
WORD ExDraw ( )
{
typedef enum {
        DRAW_HIDE,
        DRAW_STATE1,
        DRAW_STATE2,
        DRAW_STATE3,
        …
        DRAW_FOCUS,
} MYW_DRAW_STATES;
        static MYW_DRAW_STATES state = DRAW_HIDE;
        static SHORT left, top, right, bottom;

        switch(state){
            case DRAW_HIDE:
            if(IsDeviceBusy())
                return 0;
            if (GetState(pB, HIDE)) {
                // Hide the Widget
                SetColor(pB->hdr.pGolScheme->CommonBkColor);
                Bar(pB->hdr.left, pB->hdr.top,
                    pB->hdr.right, pB->hdr.bottom);
                return 1;
            }
            state = DRAW_STATE1;
        case DRAW_STATE1:
        …
            state = DRAW_FOCUS;
        case DRAW_FOCUS:
            if(IsDeviceBusy())
                return 0;
            if(GetState(pB, DRAW_FOCUS)){
                    SetLineType(FOCUS_LINE);
                    // set left, top, right, bottom variables to draw the focus
                    Rectangle(left, top, right, bottom);
            state = DRAW_HIDE;
            break;
        …
}
```

In Example 12, rendering of the focus rectangle and the hiding of the Widget are added. The rendering state, DRAW_STATE0, is now changed to DRAW_HIDE. The Widget is hidden by always drawing a bar on top of the Widget using the common background color, as defined in the Widget's current style scheme. Note that when the hide drawing bit state is set, there is no need to change the state; instead, the drawing must exit with a '1' to signify that the Widget drawing is complete.

Do not clear any drawing state bits in the Widget drawing function. The drawing state bits are cleared automatically by the GOLDraw() function that calls the Widget draw functions. For consistency, always use the GetState() API to check a particular bit state of the Widget and use FOCUS_LINE when drawing the focus line.

Rendering a Widget is made possible by calling primitive drawing functions in a predefined sequence that will render the Widget to the specified form determined by the designer of the Widget. For example, if the design of the Widget specifies that the Widget will have a red filled circle inscribed in a blue filled square, then the rendering of the blue filled square must first be performed before the filled circle is drawn. If the requirement also states that the square edges must be drawn with a dashed line, then the line type must be set before the square is drawn and changed before the circle is drawn. Therefore, the sequencing of primitives also means that the primitive drawing settings, such as line type, line thickness, font and color must also be sequenced properly. These settings can be set by the following APIs:

- SetLineType()
- SetLineThickness()
- SetFont()
- SetColor()

In some cases, the Widget may enable the clipping region using the SetClip() and SetClipRgn() APIs to set the boundaries of the clipping region. All of these primitive drawing settings and primitive rendering functions are sequenced in the Widget's draw function using the rendering states. Since GOLDraw() renders the Widgets in sequence, there will be no changes to the primitive drawing settings by another Widget's draw function until the current Widget drawing is done. The way the library is designed assures that the drawing sequences will be consistent.

The application code only needs to set the drawing state bits of a Widget and the next call to GOLDraw() will automatically render the Widget. Within GOLDraw(), a call to the GOLDrawCallback() function is provided for application-specific customized drawing. This callback function is performed only after all the Widgets have been rendered. Any modification to the primitive drawing settings by the application-specific customized drawing will not affect the Widget drawing functions.

The Text Entry Widget gives an opportunity to show how a loop that draws the same shape can be coded and adapted to the state-based rendering. Since there are n-number of keys in the Widget at any given time, the loop that draws the keys can be divided into states. The static variables can keep track of the keys that are drawn and not drawn.

Looking back at the structure of the Widget, the user-defined vertical and horizontal keys determine the number of keys drawn on the Widget. To draw the keys (drawn by the panel function from GOL.c), a loop can be implemented to draw the panels one by one. Each panel will represent one key. Since the drawing function must be able to recover the last unexecuted primitive and restart from there, the implementation of the loop becomes different. Example 14 describes the loop in a state-based rendering.

EXAMPLE 14: IMPLEMENTING LOOPS IN STATE-BASED RENDERING

```
WORD TeDraw(TEXTENTRY *pTe)
{
SHORT NumberOfKeys;
…
typedef enum {
        TE_REMOVE,
        TE_DRAW_PANEL,
        TE_INIT_DRAW_EDITBOX,
        TE_DRAW_EDITBOX,
        TE_DRAW_KEY_INIT,
        TE_DRAW_KEY_SET_PANEL,
        TE_DRAW_KEY_DRAW_PANEL,
        TE_DRAW_KEY_TEXT,
        TE_DRAW_KEY_UPDATE,
        TE_UPDATE_STRING_INIT,
        TE_UPDATE_STRING,
                } TE_DRAW_STATES;
static TE_DRAW_STATES state = TE_REMOVE;

…
switch(state) {
    case TE_REMOVE:
    …
/* ************************************************************ */
/*          Update the keys
/* ************************************************************ */
    case TE_DRAW_KEY_INIT:
te_draw_key_init_st:
            embossLtClr=pTe->hdr.pGolScheme->EmbossLtColor;
            embossDkClr=pTe->hdr.pGolScheme->EmbossDkColor;
            faceClr=pTe->hdr.pGolScheme->Color0;
            // if the active key update flag is set, only one needs to be redrawn
            if ((GetState(pTe, TE_DRAW) != TE_DRAW) &&
            (pTe->pActiveKey->update == TRUE))
            {
            CountOfKeys = (pTe->horizontalKeys*pTe->verticalKeys)-1;
            pKeyTemp = pTe->pActiveKey;
            } else {
                CountOfKeys = 0;
                pKeyTemp = pTe->pHeadOfList;
            }
            state = TE_DRAW_KEY_SET_PANEL;
    case TE_DRAW_KEY_SET_PANEL:
te_draw_key_set_panel_st:
    if (CountOfKeys < (pTe->horizontalKeys*pTe->verticalKeys)){
        // check if we need to draw the panel
        if (GetState(pTe, TE_DRAW) != TE_DRAW) {
            if (pKeyTemp->update == TRUE) {
                // set the colors needed
                if (GetState(pTe, TE_KEY_PRESSED)) {
                    embossLtClr=pTe->hdr.pGolScheme->EmbossDkColor;
                    embossDkClr=pTe->hdr.pGolScheme->EmbossLtColor;
                    faceClr=pTe->hdr.pGolScheme->Color1;
            } else {
                    embossLtClr=pTe->hdr.pGolScheme->EmbossLtColor;
                    embossDkClr=pTe->hdr.pGolScheme->EmbossDkColor;
                    faceClr=pTe->hdr.pGolScheme->Color0;
            }
        } else {
                 state = TE_DRAW_KEY_UPDATE;
                 goto te_draw_key_update_st;
             }
        }
```

EXAMPLE 14: IMPLEMENTING LOOPS IN STATE-BASED RENDERING (CONTINUED)

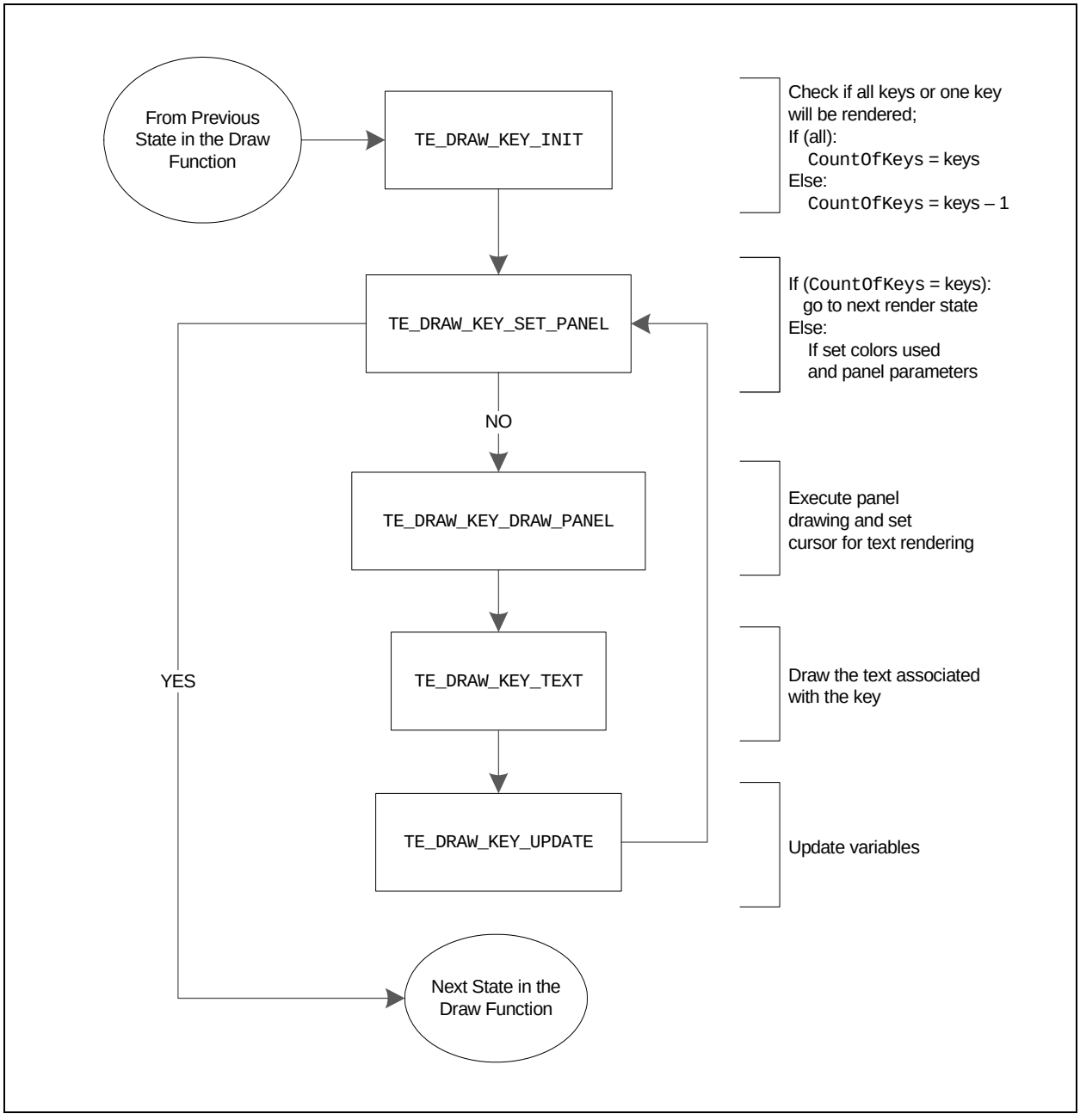
```
        // set up the panel
        GOLPanelDraw(pKeyTemp->left,pKeyTemp->top,pKeyTemp->right,pKeyTemp->bottom,0, faceClr,
embossLtClr, embossDkClr, NULL, GOL_EMBOSS_SIZE);
        state = TE_DRAW_KEY_DRAW_PANEL;
    } else {
        state = TE_UPDATE_STRING_INIT;
        goto te_update_string_init_st;
    }
    case TE_DRAW_KEY_DRAW_PANEL:
        if (!GOLPanelDrawTsk())
            return 0;
        // reset the update flag since the key panel is already redrawn
        pKeyTemp->update = FALSE;
        //set the text coordinates of the drawn key
        xText = ((pKeyTemp->left)+(pKeyTemp->right)-(pKeyTemp->textWidth))>>1;
        yText = ((pKeyTemp->bottom)+(pKeyTemp->top)-(pKeyTemp->textHeight))>>1;
        //set color of text
        // if the object is disabled, draw the disabled colors
        if (GetState(pTe, TE_DISABLED) == TE_DISABLED) {
            SetColor(pTe->hdr.pGolScheme->TextColorDisabled);
        } else {
            if ((GetState(pTe, TE_DRAW) != TE_DRAW) &&
                (GetState(pTe, TE_KEY_PRESSED)) == TE_KEY_PRESSED) {
                SetColor(pTe->hdr.pGolScheme->TextColor1);
                } else {
                    SetColor(pTe->hdr.pGolScheme->TextColor0);
                }
            }
            //output the text
            MoveTo(xText, yText);
            // set the font to be used
            SetFont(pTe->hdr.pGolScheme->pFont);
            state = TE_DRAW_KEY_TEXT;
    case TE_DRAW_KEY_TEXT:
        if(!OutText(pKeyTemp->pKeyName))
            return 0;
        state = TE_DRAW_KEY_UPDATE;
    case TE_DRAW_KEY_UPDATE:
te_draw_key_update_st:
        // update loop variables
        CountOfKeys++;
        pKeyTemp=pKeyTemp->pNextKey;
        state = TE_DRAW_KEY_SET_PANEL;
        goto te_draw_key_set_panel_st;
            …
    }
}
```

The loop is controlled by the static variable, `CountOfKeys`, and the pointer to a member of the list of keys, `pKeyTemp`. Rendering a single panel or key is decided if the active key parameter update is enabled. Only one key can be processed by the Widget at a time. When the `TE_DRAW` state bit of the Widget is set, the Widget is redrawn fully. The entire rendering process is controlled by using states. The flow of the states is shown in Figure 3.

While testing the message processing of the Widget, pay attention to the combination of states and messages that the Widget might be receiving. A lot of these cases are best checked by creating a short application that will test scenarios of the different combination of messages and states.

For example, moving your touch from a key press should cancel the 'Press'.

FIGURE 3: STATE-BASED LOOP



Disabling the Widget

One of the important features of a Widget is its ability to enter a disabled state. In this state, the Widget will not accept any message. Implementation of the disabled state is enforced in the translate message function, which is described in detail in the next section.

Translate Message Function

The messages from the user are processed by the GOLMsg() function. This function is responsible for determining which of the created Widgets in an application was affected by the message. It determines the affected Widget by calling each of the translate message functions of the Widgets. The Widget, which replies with a translated message not equal to OBJ_MSG_INVALID, is the affected Widget. A disabled Widget will also reply with OBJ_MSG_INVALID. It is not mandatory for all Widgets to process messages. These types of Widgets will implement their translate message function that automatically returns OBJ_MSG_INVALID when called.

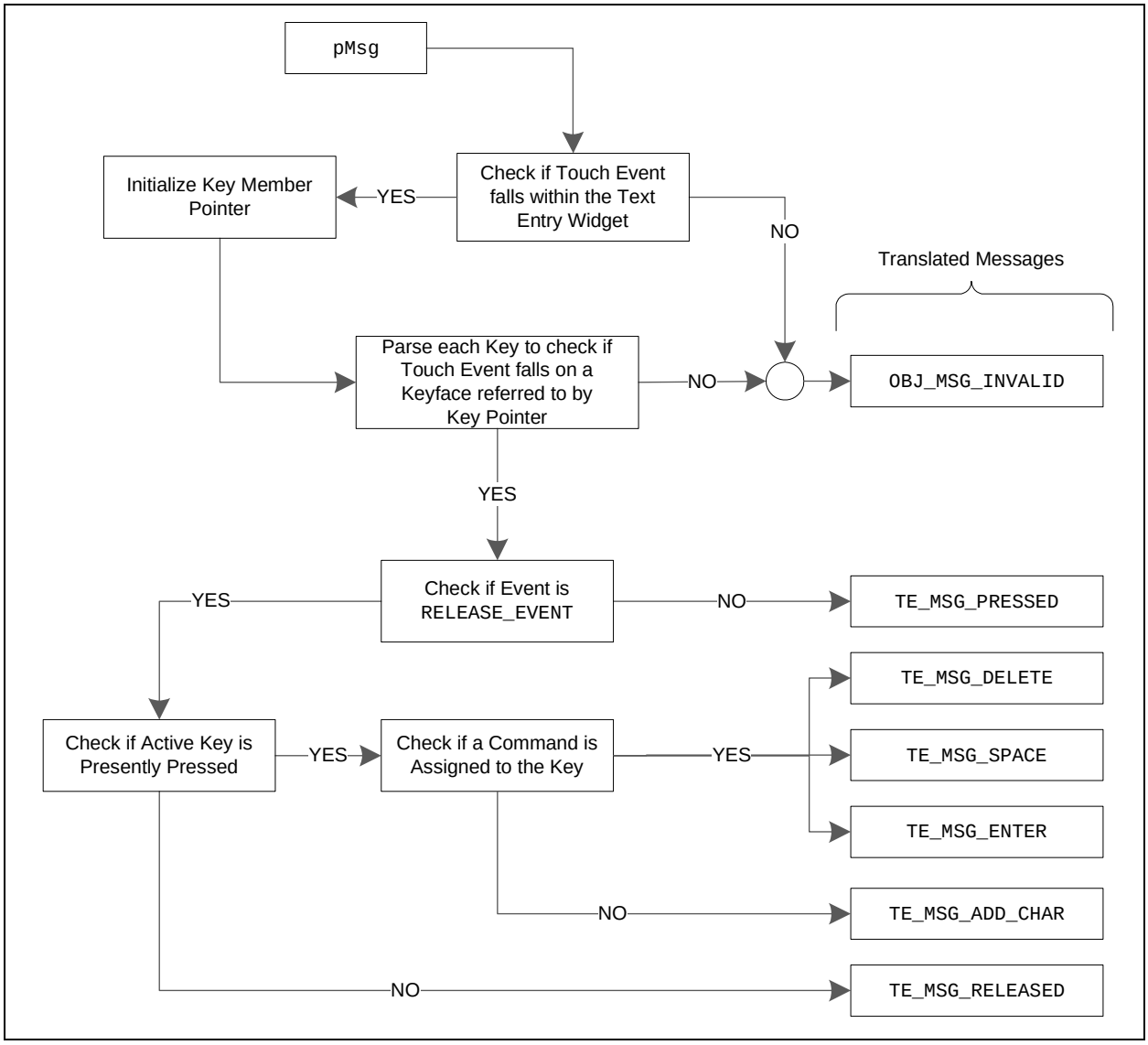
Translated messages for the Widget are defined based on the characteristics of the Widget. The definitions are usually specific to the Widgets. The definition must be added to the TRANS_MSG type enumeration found in the GOL.h file (see Example 13).

The translate message function implementation should cover all the defined translated messages that are added in the TRANS_MSG enumeration specific for the new Widget. For the Text Entry Widget, one possible implementation is shown in Figure 4 as a flowchart.

EXAMPLE 15: TRANSLATED MESSAGES ENUMERATION

```
typedef enum
{
        OBJ_MSG_INVALID = 0,      // Invalid message response.
        CB_MSG_CHECKED,           // Check Box check action ID.
        CB_MSG_UNCHECKED,         // Check Box un-check action ID.
        …
        BTN_MSG_PRESSED,          // Button pressed action ID.
        BTN_MSG_RELEASED,         // Button released action ID.
        …
        TE_MSG_RELEASED,          // Text Entry key released ID.
        TE_MSG_PRESSED,           // Text Entry key pressed ID.
        TE_MSG_ADD_CHAR,          // Text Entry Add character ID.
        TE_MSG_DELETE,            // Text Entry Delete character ID.
        TE_MSG_SPACE,             // Text Entry Insert Space character ID.
        TE_MSG_ENTER,             // Text Entry Enter Action ID.
} TRANS_MSG;
```

FIGURE 4: TRANSLATE MESSAGE FLOW FOR TOUCH SCREEN MESSAGES



Each key is parsed if the touch event falls on one of the keys. If none of the keys are affected, the translated message returned is `OBJ_MSG_INVALID`. If any one of the keys is affected, it checks if the event was a Press. If it was a 'Press' event, then it returns `TE_MSG_PRESSED`. The affected key is set as the active key. The drawing function is then able to determine which of the keys will be redrawn in the pressed state.

If it was a 'Release' event, it checks if the currently active key is in the pressed state. If the key is in the pressed state, it checks if the key has a corresponding command assigned to it. An example for a corresponding command is the delete character command. If no command is assigned to the key, it replies with `TE_MSG_ADD_CHAR`. If the key has an associated command, it returns with the corresponding translated message for the command. If the active key is not in the pressed state, the response message is `TE_MSG_RELEASED`.

The translated message function does not change the Widget's state. It just replies with the action that the Widget will perform based on the message that arrived from the user. This reply gives the application a chance to implement specific action based on the message that affected the Widget or use the default action of the Widget. This is made possible by the `GOLMsgCallback()` function, also called inside `GOLMsg()`. If the application decides to use the default action, `GOLMsgCallback()` must return a '1'. In this case, the application calls the message default function of the Widget.

Refer to the `TextEntry.c` file for details on implementation of the translate message function for the Text Entry Widget.

Message Default Function

The message default function performs the default action of the Widget based on the translated message. This is where the property and draw state bits are modified to perform the default action. Example 16 shows the message default function for the Text Entry Widget.

The code shows that each of the translated messages is processed to perform the action on the Widget. States are cleared or set, and the Widget is partially redrawn to save the time in rendering. Since all the Widget components do not change for each of the translated messages, a partial redraw is performed. Redrawing will be performed by the Widget's `GOLDraw()` drawing function.

EXAMPLE 16: MESSAGE DEFAULT FUNCTION

```
void TeMsgDefault(WORD translatedMsg,
    switch(translatedMsg) {
        case TE_MSG_DELETE:
            SetState(pTe,TE_UPDATE_KEY|TE_UPDATE_TEXT);
            break;
        case TE_MSG_SPACE:
            TeSpaceChar(pTe);
            SetState(pTe,TE_UPDATE_KEY|TE_UPDATE_TEXT);
            break;
        case TE_MSG_ENTER:
            SetState(pTe,TE_UPDATE_KEY);
            break;
        case TE_MSG_ADD_CHAR:
            TeAddChar(pTe);
            SetState(pTe,TE_UPDATE_KEY|TE_UPDATE_TEXT);
            break;
        case TE_MSG_PRESSED:
            (pTe->pActiveKey)->state = TE_KEY_PRESSED;
            SetState(pTe,TE_KEY_PRESSED|TE_UPDATE_KEY);
            return;
        case TE_MSG_RELEASED:
            (pTe->pActiveKey)->state=0;
            ClrState(pTe, TE_KEY_PRESSED);
            SetState(pTe, TE_UPDATE_KEY);
            return;
    }
    (pTe->pActiveKey)->state = 0;
    ClrState(pTe, TE_KEY_PRESSED);
}
```

Widget Run-Time Deallocation

The Widgets are removed from memory using the GOLFree() function found in the GOL.c file. This function deletes all the Widgets in the active list. The active list of Widgets is the list that is parsed by GOLDraw() and GOLMsg() to render and process the messages, respectively. If the Widget itself creates and references another object in the memory, then a corresponding function must be created for the Widget to remove these additional objects. Failure to do this will result in a memory leak. If the application continuously creates and frees the Widget, the memory allocated by the Widget for these additional objects will eventually consume all the memory space allocated for dynamically created objects.

The Text Entry Widget requires an additional list of structures that is also created dynamically when the Widget is used and allocated space in memory. This list defines the characters and commands associated with the keys. Since this is a dynamically created space, GOLFree() must free the memory used by the list and the Widget. Example 17 shows the changes made to the GOLFree() function found in the GOL.c file to include the call to the TeDelKeyMembers() API to remove the list. Please refer to the TextEntry.c file for details of the TeDelKeyMembers() implementation.

EXAMPLE 17: GOLFREE() IMPLEMENTATION WITH TEXT ENTRY WIDGET ADDED

```
void GOLFree( ) {
OBJ_HEADER * pNextObj;
OBJ_HEADER * pCurrentObj;

    pCurrentObj = _pGolObjects;
    while(pCurrentObj != NULL){
        pNextObj = pCurrentObj->pNxtObj;
            #ifdef USE_LISTBOX
            if(pCurrentObj->type == OBJ_LISTBOX)
                    LbDelItemsList((LISTBOX*)pCurrentObj);
            #endif

            #ifdef USE_GRID
            if(pCurrentObj->type == OBJ_GRID)
                    GridFreeItems((GRID*)pCurrentObj);
            #endif

            #ifdef USE_CHART
            if(pCurrentObj->type == OBJ_CHART)
                    ChRemoveDataSeries((CHART*)pCurrentObj, 0);
            #endif

            #ifdef USE_TEXTENTRY
            if(pCurrentObj->type == OBJ_TEXTENTRY)
                    TeDelKeyMembers ((TEXTENTRY*)pCurrentObj);
            #endif

            free(pCurrentObj);
            pCurrentObj = pNextObj;
    }
    GOLNewList();
}
```

Supporting Functions

Now that the Widget is fully integrated into the Graphics Library, the supporting functions can be added for easy usage of the Widget. Such supporting functions are listed and described in Table 6.

For a complete list of functions, refer to the `TextEntry.c` and `TextEntry.h` files.

TABLE 6: SUPPORTING FUNCTIONS(1)

Function	Description
`TeSetBuffer(pTe, pText)`	Function to set the string on the edit box of the Text Entry Widget. `pText` is the buffer address.
`TeGetBuffer(pTe)`	Function to get the buffer address of the string displayed in the edit box.
`TeClearBuffer(pTe)`	Function to clear the contents of the edit box.
`TeIsKeyPressed(pTe, index)`	Function to test if the particular key referenced by the index is currently pressed.
`TeSetKeyCommand(pTe, index, command)`	This function assigns a command to a particular key referenced by the index. On assigning the command, the key will not echo the assigned string or character to the key, but rather perform the command.
`TeGetKeyCommand(pTe, index, command)`	This function returns the current command assigned to a particular key referenced by the index.
`TeCreateKeyMembers(pTe, pText[])`	This function creates the list of keys and assigns each entry in the `pText` array to each created key.
`TeDelKeyMembers(pTe)`	This function removes the list of keys associated with the Key Entry Widget. It deletes the memory space used by the list of keys. This is the same function called in `GOLFree()` when the Widget is being deleted from memory.

Note 1: `pTe` is the pointer to the Text Entry Widget.

Check List for Creating a New Widget

Table 7 lists the items that need to be checked to make sure that the new Widget is integrated properly into the Graphics Library.

TABLE 7: CHECK LIST FOR NEW WIDGET

Item	Guide Questions
`USE_WIDGETNAME`	In users' Widget C code, is the compile switch that is specific for the Widget and defined in the `GraphicsConfig.h` file implemented? Compiler includes the Widget code in the build only if it is defined in the `GraphicsConfig.h` file. (Ex: `USE_TEXTENTRY` for Text Entry Widget.)
Widget State Bits	Double-check if all the state bits defined are using the standard drawing and property state bits for hiding, drawing and enabling. Check that all additional state bits are not using the standard states, including optional states of standard Widgets. (Optional state example is for focus.)
`GOL_OBJ_TYPE`	Is the new Widget type included in the `GOL_OBJ_TYPE` enumeration found in `GOL.h`? (Ex: Add `GOL_TEXTENTRY` for type Text Entry Widget.)
`TRANS_MSG`	Are the new translated messages specific to users' Widget included in the `TRANS_MSG` enumeration found in the `GOL.h` file? Users may reuse the existing translated messages.
`OBJ_HEADER`	Do the first nine members of the Widget structure have the same data type or size as the `OBJ_HEADER`? If the user has used `OBJ_HEADER` as the first structure member, then there is no need for a check.

TABLE 7: CHECK LIST FOR NEW WIDGET (CONTINUED)

Item	Guide Questions
Create Function	In users' "create function", check for the following items: • Is the Widget dynamically created in the memory? • Is the function returning NULL when dynamic memory allocation fails? • Are all Widget parameters properly initialized in this function? • Is the style scheme initialized to the user-defined style scheme? If not, is it initialized to the default style scheme? • If the Widget uses a different font, other than the font defined in the style scheme, is this font initialized to `GOLFontDefault` if the font is specified as NULL in the parameters? • Is the new Widget added to the global list of active Widgets (use `GOLAddObject()` to perform this task)? • Is the function returning the address of the new Widget if created and initialized properly?
Draw Function	In users' draw function, check for the following items: • Is the draw function rendering the Widget using the states? • Are the states defined to represent non-blocking rendering functions? This means that user function can recover and perform the next rendering function if it exited due to `IsDeviceBusy()` returning a '1'. • Is the function returning a '1' if `IsDeviceBusy()` returns a '1' and '0' if the rendering of the Widget is done?
Translate Message Function	In users translate message function, check for the following items: • Is the function returning `OBJ_MSG_INVALID` if it is not affected by the message? • Are all the defined translated messages covered by the function? • Does the Widget support touch screen? If so, can the code that processes the touch screen messages be removed by a compile switch if the touch screen is not used by the application (`#ifdef USE_TOUCHSCREEN`)? • Is the Widget supporting a keyboard? If so, can the code that processes the keyboard messages be removed by a compile switch if the keyboard is not used by the application (`#ifdef USE_KEYBOARD`)?
Message Default Function	In users' default message function, check for the following items: • Are all translated messages covered? • Check if states are set or cleared properly.
`GOLDraw()`	Is the `GOLDraw()` function modified to include the new Widget? This is a call to the draw function of the Widget.
`GOLMsg()`	Is the `GOLMsg()` function modified to include the new Widget? This is the call to the translate message function and message default function of the Widget.
`GOLFree()`	Is the new Widget also allocating dynamic memory apart from the "create function"? If so, the additional allocated memory must be freed in the `GOLFree()` function.
Header File Inclusion	Is the new Widget header file included in the build when `USE_WIDGETNAME` is defined in the `GraphicsConfig.h` file? Add the `#include USE_WIDGETNAME` in the `Graphics.h` file.

CONCLUSION

In certain applications, creating new Widgets from scratch is necessary to save code and to simplify the usage of the Widgets. The different functions and files in the Graphics Library, that require modification to implement the new Widget, have been discussed in this document. Code examples are provided to show the possible ways of implementing the required functions and macros. A checklist is also provided to check the items that will be implemented and modified to facilitate easy integration of the new Widget into the Graphics Library.

For details on the implementation of the Graphics Library, refer to the source code included in the installation of the library. The installer can be downloaded from www.microchip.com/graphics. The code for the Text Entry Widget example comes with the installation of the library.

REFERENCES

- Microchip Application Note *AN1136,* "*How to Use Widgets in Microchip Graphics Library*" (DS01136), Microchip Technology Incorporated.
- Microchip Application Note *AN1182,* "*Fonts in the Microchip Graphics Library*" (DS01182), Microchip Technology Incorporated.
- Microchip Graphics Library, `Microchip Graphics Library Help.chm`, Microchip Technology Incorporated (www.microchip.com/graphics).
- HIF 2131 – Designing with Microchip Graphics Library, Microchip Regional Training Center web site (`www.microchip.com/rtc`).

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-3090-4444
Fax: 91-80-3090-4080

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

01/16/09