emWin

Window Manager
Widgets

Custom Widget Type
Creation Guide

Document: AN03002
Revision: 1
Date: April 22, 2016

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: April 22, 2016

Revision	Date	By	Description
1	160422	SC	Typos corrected.
0	130326	AS	Initial version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Change`	Changed sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table 1.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin

Graphics software and GUI

emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS

Real Time Operating System

embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP

TCP/IP stack

embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile

File system

emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and Compact-Flash cards, are available.

USB-Stack

USB device/host stack

A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher

Flash programmer

Flash Programming tool primarily for micro controllers.

J-Link

JTAG emulator for ARM cores

USB driven JTAG interface for ARM cores.

J-Trace

JTAG emulator with trace

USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software

Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Introduction

This document is intended to give a short and simple description on how to implement a custom emWin widget. This guide deals with an example implementation of a button. The used functions are explained by their meaning as it is necessary to understand the context.

A complete function reference can be found in the according chapter in the emWin user manual, which can be downloaded at www.segger.com.

1.1 Requirements

In order to be able to use this document it is required to have a licensed copy of emWin. This guide is written for emWin V5.20, but might be used for later versions as well. Knowledge of the used emWin functions is not required, since cross-references to the according function descriptions in the emWin user manual are included.

1.2 Overview

The actual task when implementing a custom widget is to provide the end user with the following:

- Create()-function which returns a handle to the created widget and allows setting callback function.
- Set()- and Get()-functions to request and set widget specific properties.
- Callback()-function to process messages which were sent to the custom widget.

In the following chapters the custom function are prefixed MYWIDGET_. These function are the ones which are meant to be used by the end user.

Chapter 2

Creating a custom widget type

This chapter describes how to implement a custom type of widget. This includes a function to create the widget, a function to handle messages and functions to set and get widget specific properties.

2.1 Requirements

Since widgets are actually windows with enhanced functionality, it is required to create a window with capabillities to store additional data. Depending on the actual type of widget implemented, it is a good practice to add further parameters to the prototype of the Create() function in order to force the user to set up the widget correctly from creation.

If a button widget has to be implemented and it is a requirement to define the text to be displayed on the button, a pointer to a string may be added to the prototype.

2.2 Creating the window

The first and basic step is creating a simple window. This can be done using the function `WM_CreateWindowAsChild()`. The prototype of this function can be copied and used for the custom create function `MYWIDGET_Create()`:

```
WM_HWIN MYWIDGET_Create(int x0, int y0, int xSize, int ySize, WM_HWIN hWinParent,
                         U32 Style, WM_CALLBACK * pfCallback, int NumExtraBytes) {
  WM_HWIN hWin;

  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent,
                                Style, pfCallback, NumExtraBytes);
  return hWin;
}
```

The parameters of the function `MYWIDGET_Create()` are passed to the function `WM_CreateWindowAsChild()` without any changes. A detailed description of the function can be found in chapter 15.7 "`The Window Manager`" -> "`WM API`" in the emWin user manual.

2.3 Widget handle

To allow the end user to easily distinguish between widget types, a specific type of handle can be defined:

```
typedef WM_HMEM MYWIDGET_Handle;
```

The prototype of the create function changes accordingly:

```
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style,
                                WM_CALLBACK * pfCallback, int NumExtraBytes);
```

2.4 Widget specific data

To be able to implement widget specific functionality, widget specific data has to be stored. A simple data structure should be sufficient as shown by the example below:

```
typedef struct {
  U8            Pressed;
  GUI_COLOR     aBkColor[3];
  GUI_COLOR     aTextColor[3];
  GUI_COLOR     FocusColor;
  const char *  pText;
  int           NumExtraBytes;
} MYWIDGET_Obj;
```

The background and text colors were declared as arrays, so the state of the widget can be distinguished by its visual appearance. In later sections, the following defines are used to access the color arrays according to the current state of the widget.

```
#define MYWIDGET_CI_UNPRESSED 0
#define MYWIDGET_CI_PRESSED   1
#define MYWIDGET_CI_DISABLED  2
```

The included variables of the `MYWIDGET_Obj`-structure can be defined as it is required by the actual type of widget to implement. The only variable which is required in every case is a variable to store the number of bytes which is defined by the end user at creation. In this case it is called `NumExtraBytes`. For the type of button widget, which is implemented in this guide the contained variables are used to store the state, the state specific colors and a pointer to a string.

This structure is going to be stored as user data. Nevertheless the opportunity for the end user to attach data to the widget has to be preserved. The function `MYWIDGET_Create()` changes as follows:

```
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style,
                                WM_CALLBACK * pfCallback, int NumExtraBytes) {
MYWIDGET_Handle hWin;

  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent, Style,
                                pfCallback, sizeof(MYWIDGET_Obj) + NumExtraBytes);
  return hWin;
}
```

Warning: The end user must not use the function WM_GetUserData() or WM_SetUserData() with a widget of a custom type as it is implemented using this guide, since the user would either overwrite widget specific data, or not retrieve the expected data.

Therefore custom functions for these purposes have to be implemented as shown in the following:

MYWIDGET_GetUserData()

Function description

Copies the given number of bytes of user data to the destination buffer.

Implementation

```
int MYWIDGET_GetUserData(MYWIDGET_Handle hWin, void * pDest, int SizeOfBuffer) {
  MYWIDGET_Obj   MyWidget;
  int            NumBytes;
  U8           * pExtraBytes;

  if (SizeOfBuffer <= 0) {
    return 0;
  }
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  pExtraBytes = (U8 *)malloc(sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
  if (pExtraBytes) {
    WM_GetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    if (SizeOfBuffer >= MyWidget.NumExtraBytes) {
      NumBytes = MyWidget.NumExtraBytes;
    } else {
      NumBytes = SizeOfBuffer;
    }
    GUI_MEMCPY(pDest, pExtraBytes + sizeof(MYWIDGET_Obj), NumBytes);
    free(pExtraBytes);
    return NumBytes;
  }
  return 0;
}
```

Parameter	Description
`hWin`	Handle to the Widget.
`pDest`	Pointer to the destination buffer.
`SizeOfBuffer`	Size of the destination buffer.

Table 2.1: MYWIDGET_GetUserData() parameter list

Return value

Number of copied bytes.

MYWIDGET_SetUserData()

Function description

Sets the given number of bytes of the source buffer as user data.

Implementation

```
int MYWIDGET_SetUserData(MYWIDGET_Handle hWin, void * pSrc, int SizeOfBuffer) {
  MYWIDGET_Obj   MyWidget;
  int            NumBytes;
  U8           * pExtraBytes;

  if (SizeOfBuffer <= 0) {
    return 1;
  }
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  pExtraBytes = (U8 *)malloc(sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
  if (pExtraBytes) {
    WM_GetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    if (SizeOfBuffer >= MyWidget.NumExtraBytes) {
      NumBytes = MyWidget.NumExtraBytes;
    } else {
      NumBytes = SizeOfBuffer;
    }
    GUI_MEMCPY(pExtraBytes + sizeof(MYWIDGET_Obj), pSrc, NumBytes);
    WM_SetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    free(pExtraBytes);
    return 0;
  }
  return 1;
}
```

Parameter	Description
`hWin`	Handle to the Widget.
`pSrc`	Pointer to the source buffer.
`SizeOfBuffer`	Size of the source buffer.

Table 2.2: MYWIDGET_SetUserData() parameter list

Return value

0, on success.

1, on error.

2.5 Widget specific default properties

Default values should be defined for every property, which is not explicitly set by the user at creation. According to the `MYWIDGET_Obj`-structure the default values might be defined as shown below:

```
const MYWIDGET_Obj MYWIDGET_Default = {
  0,                                                 // U8            Pressed;
  { GUI_DARKBLUE, GUI_LIGHTBLUE, GUI_GRAY },         // GUI_COLOR     aBkColor[3];
  { GUI_WHITE,    GUI_DARKGRAY,  GUI_LIGHTGRAY },    // GUI_COLOR     aTextColor[3];
  GUI_ORANGE,                                        // GUI_COLOR     FocusColor;
  NULL,                                              // const char *  pText;
  0                                                  // int           NumExtraBytes;
};
```

These values are used when the widget is created. The function `MYWIDGET_Create()` changes as follows:

```
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style,
                                WM_CALLBACK * pfCallback, int NumExtraBytes) {
  MYWIDGET_Handle hWin;
  MYWIDGET_Obj    MyWidget;

  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent, Style,
                                pfCallback, sizeof(MYWIDGET_Obj) + NumExtraBytes);
  MyWidget               = MYWIDGET_Default;
  MyWidget.NumExtraBytes = NumExtraBytes;
  WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  return hWin;
}
```

2.6 Setting mandatory properties

Assuming it is mandatory or at least usual for a button to display a text on top, the function `MYWIDGET_Create()` can be changed as follows:

```
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style, const char * pText,
                                WM_CALLBACK * pfCallback, int NumExtraBytes) {
  MYWIDGET_Handle hWin;
  MYWIDGET_Obj    MyWidget;

  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent, Style,
                                pfCallback, sizeof(MYWIDGET_Obj) + NumExtraBytes);
  MyWidget                 = MYWIDGET_Default;
  MyWidget.NumExtraBytes = NumExtraBytes;
  if (pText) {
    MyWidget.pText = pText;
  }
  WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  return hWin;
}
```

`MyWidget.pText` is set to `pText` after the default values were applied.

2.7 Setting the callback function

The last and most important part of implementing the `Create()`-function is setting the window to use the widget specific callback function. Since the opportunity for the end user to implement a custom callback function has to be preserved, it should be checked if a pointer to a custom callback function was passed:

```
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style, const char * pText,
                                WM_CALLBACK * pfCallback, int NumExtraBytes) {
  MYWIDGET_Handle   hWin;
  MYWIDGET_Obj      MyWidget;
  WM_CALLBACK     * pfUsed;

  if (pfCallback) {
    pfUsed = pfCallback;
  } else {
    pfUsed = MYWIDGET_Callback;
  }
  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent, Style,
                                pfUsed, sizeof(MYWIDGET_Obj) + NumExtraBytes);
  MyWidget                 = MYWIDGET_Default;
  MyWidget.NumExtraBytes = NumExtraBytes;
  if (pText) {
    MyWidget.pText = pText;
  }
  WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  return hWin;
}
```

2.8 Widget specific callback function

The callback function makes the user recognize the type of the widget. It is responsible for drawing the widget as well as for reacting on other events like keyboard- or touch-events. Since this guide deals with an example implementation for a custom button widget, it is sufficient to deal with the message `WM_PAINT`, `WM_TOUCH` and `WM_SET_FOCUS`.

A complete reference of messages as well as a detailed description on how the Window Manager's callback mechanism works can be found in the emWin user manual in chapter 15 "The Window Manager".

The first step is using the starting point:

```
void MYWIDGET_Callback(WM_MESSAGE * pMsg) {
  switch (pMsg->MsgId) {
  default:
    WM_DefaultProc(pMsg);
  }
}
```

In order to have a visual impact on the widget, a reaction to the `WM_PAINT`-message is implemented. Since the colors to use are stored in the user data, they have to be fetched first:

```
void MYWIDGET_Callback(WM_MESSAGE * pMsg) {
  MYWIDGET_Handle hWin;
  MYWIDGET_Obj    MyWidget;
  GUI_RECT        WinRect;
  int             ColorIndex;

  hWin = pMsg->hWin;
  //
  // Get the window coordinates and widget specific data
  //
  WM_GetWindowRectEx(hWin, &WinRect);
  GUI_MoveRect(&WinRect, -WinRect.x0, -WinRect.y0);
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  switch (pMsg->MsgId) {
  case WM_PAINT:
    //
    // Determine the color according to the current state
    //
    if (WM_IsEnabled(hWin)) {
      if (MyWidget.Pressed) {
        ColorIndex = MYWIDGET_CI_PRESSED;
      } else {
        ColorIndex = MYWIDGET_CI_UNPRESSED;
      }
    } else {
      ColorIndex = MYWIDGET_CI_DISABLED;
    }
    //
    // Draw the background
    //
    GUI_SetColor(MyWidget.aBkColor[ColorIndex]);
    GUI_FillRectEx(&WinRect);
    //
    // Draw the focus rectangle
    //
    if (WM_HasFocus(hWin)) {
      GUI_SetColor(MyWidget.FocusColor);
      GUI_DrawRectEx(&WinRect);
    }
    //
    // Display the text.
    //
    GUI_SetColor(MyWidget.aTextColor[ColorIndex]);
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_DispStringInRect(MyWidget.pText, &WinRect, GUI_TA_HCENTER | GUI_TA_VCENTER);
    break;
  default:
    WM_DefaultProc(pMsg);
  }
}
```

At this point the code can be run and the widget can be recognized as a rectangle on the screen. Since this is going to be a button, one may expect a change when it is clicked, but nothing happens. The reason is that the variable `MyWidget.Pressed` is never changed, so the widget is drawn the same way regardless the pressed state. In order to update the state, it is required to react to the message `WM_TOUCH`:

```
void MYWIDGET_Callback(WM_MESSAGE * pMsg) {
  MYWIDGET_Handle   hWin;
  GUI_PID_STATE   * pState;
  MYWIDGET_Obj      MyWidget;
  GUI_RECT          WinRect;
  int               ColorIndex;

  hWin = pMsg->hWin;
  WM_GetWindowRectEx(hWin, &WinRect);
  GUI_MoveRect(&WinRect, -WinRect.x0, -WinRect.y0);
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  switch (pMsg->MsgId) {
  case WM_PAINT:
    if (WM_IsEnabled(hWin)) {
      if (MyWidget.Pressed) {
        ColorIndex = MYWIDGET_CI_PRESSED;
      } else {
        ColorIndex = MYWIDGET_CI_UNPRESSED;
      }
    } else {
      ColorIndex = MYWIDGET_CI_DISABLED;
    }
    GUI_SetColor(MyWidget.aBkColor[ColorIndex]);
    GUI_FillRectEx(&WinRect);
    if (WM_HasFocus(hWin)) {
      GUI_SetColor(MyWidget.FocusColor);
      GUI_DrawRectEx(&WinRect);
    }
    GUI_SetColor(MyWidget.aTextColor[ColorIndex]);
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_DispStringInRect(MyWidget.pText, &WinRect, GUI_TA_HCENTER | GUI_TA_VCENTER);
    break;
  case WM_TOUCH:
    if (pMsg->Data.p) {
      pState = (GUI_PID_STATE *)pMsg->Data.p;
      if (MyWidget.Pressed != pState->Pressed) {
        MyWidget.Pressed = pState->Pressed;
        WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
        if (MyWidget.Pressed) {
          WM_SetFocus(hWin);
        }
        WM_InvalidateWindow(hWin);
      }
    } else {
      MyWidget.Pressed = 0;
      WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
      WM_InvalidateWindow(hWin);
    }
    break;
  default:
    WM_DefaultProc(pMsg);
  }
}
```

The additional code retrieves the PID state which is sent along with the `WM_TOUCH`-message and sets the user data accordingly. After updating the widget specific data the widget is invalidated in order to receive a new `WM_PAINT`-message.

When running this code clicking the widget will cause the color to change. Nevertheless the focus rectangle is not drawn, although the function `WM_SetFocus()` is called when the widget receives a pressed PID state. The reason for this behavior is that the widget never "accepts" the focus. To do so the callback function needs to react to the `WM_SET_FOCUS`-message as shown below:

```
void MYWIDGET_Callback(WM_MESSAGE * pMsg) {
  MYWIDGET_Handle   hWin;
  GUI_PID_STATE   * pState;
  MYWIDGET_Obj      MyWidget;
  GUI_RECT          WinRect;
  int               ColorIndex;

  hWin = pMsg->hWin;
  WM_GetWindowRectEx(hWin, &WinRect);
  GUI_MoveRect(&WinRect, -WinRect.x0, -WinRect.y0);
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  switch (pMsg->MsgId) {
  case WM_PAINT:
    if (WM_IsEnabled(hWin)) {
      if (MyWidget.Pressed) {
        ColorIndex = MYWIDGET_CI_PRESSED;
      } else {
        ColorIndex = MYWIDGET_CI_UNPRESSED;
      }
    } else {
      ColorIndex = MYWIDGET_CI_DISABLED;
    }
    GUI_SetColor(MyWidget.aBkColor[ColorIndex]);
    GUI_FillRectEx(&WinRect);
    if (WM_HasFocus(hWin)) {
      GUI_SetColor(MyWidget.FocusColor);
      GUI_DrawRectEx(&WinRect);
    }
    GUI_SetColor(MyWidget.aTextColor[ColorIndex]);
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_DispStringInRect(MyWidget.pText, &WinRect, GUI_TA_HCENTER | GUI_TA_VCENTER);
    break;
  case WM_TOUCH:
    pState = (GUI_PID_STATE *)pMsg->Data.p;
    if (MyWidget.Pressed != pState->Pressed) {
      MyWidget.Pressed = pState->Pressed;
      WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
      if (MyWidget.Pressed) {
        WM_SetFocus(hWin);
      }
      WM_InvalidateWindow(hWin);
    }
    break;
  case WM_SET_FOCUS:
    if (pMsg->Data.v) {
      pMsg->Data.v = 0;
    }
    WM_InvalidateWindow(hWin);
    break;
  default:
    WM_DefaultProc(pMsg);
  }
}
```

`pMsg->Data.v = 1` means that the widget receives the focus. In order to let the Window Manager "know" that the focus was accepted, the value has to be set to 0. Otherwise the focus is not received. Once the widget gains or loses the focus, it should be invalidated so the appearance changes accordingly, if it is intended to do so.

Chapter 3

Example implementation

This chapter shows an example application, which includes the complete source code which was explained in the previous chapter as well as the usage of a custom user callback function and user data. Additionally a `BUTTON` widget is used to make the change of the focus recognizable.

3.1 Sample description

The following sample shows the internal implementation of the custom widget type, which actually is a button, and also shows an end user application, which uses a custom callback function to display the text which was previously stored as widget user data.

3.2 Internal source code

```
#include <string.h>        // Definition of NULL
#include <stdlib.h>        // malloc() and free()
#include "GUI.h"
#include "WM.h"
#include "BUTTON.h"

/*********************************************************************
*
*       Color indices
*/
#define MYWIDGET_CI_UNPRESSED 0
#define MYWIDGET_CI_PRESSED   1
#define MYWIDGET_CI_DISABLED  2

/*********************************************************************
*
*       Typedef
*/
typedef WM_HMEM MYWIDGET_Handle;

/*********************************************************************
*
*       MYWIDGET_Obj
*/
typedef struct {
  U8           Pressed;
  GUI_COLOR    aBkColor[3];
  GUI_COLOR    aTextColor[3];
  GUI_COLOR    FocusColor;
  const char * pText;
  int          NumExtraBytes;
} MYWIDGET_Obj;

/*********************************************************************
*
*       MYWIDGET_Default
*/
const MYWIDGET_Obj MYWIDGET_Default = {
  0,
  { GUI_DARKBLUE, GUI_LIGHTBLUE, GUI_GRAY },
  { GUI_WHITE,    GUI_DARKGRAY,  GUI_LIGHTGRAY },
  GUI_ORANGE,
  NULL,
  0
};
```

```
/*********************************************************************
*
*       MYWIDGET_Callback
*/
void MYWIDGET_Callback(WM_MESSAGE * pMsg) {
  MYWIDGET_Handle   hWin;
  GUI_PID_STATE   * pState;
  MYWIDGET_Obj      MyWidget;
  GUI_RECT          WinRect;
  int               ColorIndex;

  hWin = pMsg->hWin;
  WM_GetWindowRectEx(hWin, &WinRect);
  GUI_MoveRect(&WinRect, -WinRect.x0, -WinRect.y0);
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  switch (pMsg->MsgId) {
  case WM_PAINT:
    if (WM_IsEnabled(hWin)) {
      if (MyWidget.Pressed) {
        ColorIndex = MYWIDGET_CI_PRESSED;
      } else {
        ColorIndex = MYWIDGET_CI_UNPRESSED;
      }
    } else {
      ColorIndex = MYWIDGET_CI_DISABLED;
    }
    GUI_SetColor(MyWidget.aBkColor[ColorIndex]);
    GUI_FillRectEx(&WinRect);
    if (WM_HasFocus(hWin)) {
      GUI_SetColor(MyWidget.FocusColor);
      GUI_DrawRectEx(&WinRect);
    }
    GUI_SetColor(MyWidget.aTextColor[ColorIndex]);
    GUI_SetTextMode(GUI_TM_TRANS);
    GUI_DispStringInRect(MyWidget.pText, &WinRect, GUI_TA_HCENTER | GUI_TA_VCENTER);
    break;
  case WM_TOUCH:
    if (pMsg->Data.p) {
      pState = (GUI_PID_STATE *)pMsg->Data.p;
      if (MyWidget.Pressed != pState->Pressed) {
        MyWidget.Pressed = pState->Pressed;
        WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
        if (MyWidget.Pressed) {
          WM_SetFocus(hWin);
        }
        WM_InvalidateWindow(hWin);
      }
    } else {
      MyWidget.Pressed = 0;
      WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
      WM_InvalidateWindow(hWin);
    }
    break;
  case WM_SET_FOCUS:
    if (pMsg->Data.v) {
      pMsg->Data.v = 0;
    }
    WM_InvalidateWindow(hWin);
    break;
  default:
    WM_DefaultProc(pMsg);
  }
}
```

```
/*********************************************************************
*
*       MYWIDGET_GetUserData
*/
int MYWIDGET_GetUserData(MYWIDGET_Handle hWin, void * pDest, int SizeOfBuffer) {
  MYWIDGET_Obj   MyWidget;
  int            NumBytes;
  U8           * pExtraBytes;

  if (SizeOfBuffer <= 0) {
    return 0;
  }
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  pExtraBytes = (U8 *)malloc(sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
  if (pExtraBytes) {
    WM_GetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    if (SizeOfBuffer >= MyWidget.NumExtraBytes) {
      NumBytes = MyWidget.NumExtraBytes;
    } else {
      NumBytes = SizeOfBuffer;
    }
    GUI_MEMCPY(pDest, pExtraBytes + sizeof(MYWIDGET_Obj), NumBytes);
    free(pExtraBytes);
    return NumBytes;
  }
  return 0;
}

/*********************************************************************
*
*       MYWIDGET_SetUserData
*/
int MYWIDGET_SetUserData(MYWIDGET_Handle hWin, void * pSrc, int SizeOfBuffer) {
  MYWIDGET_Obj   MyWidget;
  int            NumBytes;
  U8           * pExtraBytes;

  if (SizeOfBuffer <= 0) {
    return 1;
  }
  WM_GetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  pExtraBytes = (U8 *)malloc(sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
  if (pExtraBytes) {
    WM_GetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    if (SizeOfBuffer >= MyWidget.NumExtraBytes) {
      NumBytes = MyWidget.NumExtraBytes;
    } else {
      NumBytes = SizeOfBuffer;
    }
    GUI_MEMCPY(pExtraBytes + sizeof(MYWIDGET_Obj), pSrc, NumBytes);
    WM_SetUserData(hWin, pExtraBytes,
                   sizeof(MYWIDGET_Obj) + MyWidget.NumExtraBytes);
    free(pExtraBytes);
    return 0;
  }
  return 1;
}
```

```
/*********************************************************************
*
*       MYWIDGET_Create
*/
MYWIDGET_Handle MYWIDGET_Create(int x0, int y0, int xSize, int ySize,
                                WM_HWIN hWinParent, U32 Style, const char * pText,
                                WM_CALLBACK * pfCallback, int NumExtraBytes) {
  MYWIDGET_Handle   hWin;
  MYWIDGET_Obj      MyWidget;
  WM_CALLBACK     * pfUsed;

  if (pfCallback) {
    pfUsed = pfCallback;
  } else {
    pfUsed = MYWIDGET_Callback;
  }
  MyWidget               = MYWIDGET_Default;
  MyWidget.NumExtraBytes = NumExtraBytes;
  if (pText) {
    MyWidget.pText = pText;
  }
  hWin = WM_CreateWindowAsChild(x0, y0, xSize, ySize, hWinParent, Style,
                                pfUsed, sizeof(MYWIDGET_Obj) + NumExtraBytes);
  WM_SetUserData(hWin, &MyWidget, sizeof(MYWIDGET_Obj));
  return hWin;
}
```

3.3 End user application

```
/*********************************************************************
*
*       _cbMyWidget
*/
void _cbMyWidget(WM_MESSAGE * pMsg) {
  GUI_RECT WinRect;
  char     acText[20] = { 0 };

  switch (pMsg->MsgId) {
  case WM_PAINT:
    MYWIDGET_Callback(pMsg);
    MYWIDGET_GetUserData(pMsg->hWin, acText, sizeof(acText));
    GUI_SetColor(GUI_WHITE);
    GUI_SetTextMode(GUI_TM_TRANS);
    WM_GetWindowRectEx(pMsg->hWin, &WinRect);
    GUI_MoveRect(&WinRect, -WinRect.x0, -WinRect.y0);
    GUI_DispStringInRect(acText, &WinRect, GUI_TA_HCENTER | GUI_TA_VCENTER);
    break;
  default:
    MYWIDGET_Callback(pMsg);
  }
}

/*********************************************************************
*
*       MainTask
*/
void MainTask(void) {
  MYWIDGET_Handle hMyWidget;
  char            acExtraBytes[] = "Extrabytes";

  GUI_Init();
  hMyWidget = MYWIDGET_Create(10, 10, 100, 50, WM_HBKWIN, WM_CF_SHOW, NULL,
                              _cbMyWidget, strlen(acExtraBytes));
  MYWIDGET_SetUserData(hMyWidget, acExtraBytes, strlen(acExtraBytes));
  BUTTON_Create(10, 100, 100, 50, 0, WM_CF_SHOW);
  while (1) {
    GUI_Delay(100);
  }
}
```

Index