



Quantum[®]Leaps
innovating embedded systems



Application Note

QWin[™] GUI Kit for Prototyping Embedded Systems on Windows

Document Revision K
March 2021



Table of Contents

1 Introduction.....	1
1.1 About the QWin™ GUI.....	1
1.2 About the Development Tools.....	1
1.3 Licensing.....	1
2 Getting Started.....	2
2.1 Downloading and Installing QWin from QTools Collection.....	2
2.2 Building the QWin Demo with the Visual Studio C++Toolset.....	3
2.3 Running the QWin GUI Demo on Windows.....	4
2.4 Running the QWin GUI Demo on the EK-LM3S811 Board.....	5
2.4.1 GNU-ARM Toolset.....	5
2.4.2 IAR-ARM Toolset.....	6
3 Developing Embedded Code on Windows.....	7
3.1 Dual Targeting.....	7
3.2 General Software Structure.....	7
3.3 Dialog Box GUI.....	9
3.1 Main Steps for Creating an Embedded Front Panel GUI.....	9
3.2 The Dialog Box Resource.....	10
3.2.1 The Background Image.....	11
3.3 Owner-Drawn Buttons.....	12
3.4 Graphic Displays.....	15
3.5 Segment Displays.....	18
3.6 LEDs.....	20
4 The Board Support Package.....	21
4.1 The BSP Interface (bsp.h).....	21
4.2 BSP for QWin GUI (WinMain).....	22
4.3 BSP for QWin GUI (WndProc).....	24
4.4 The Application Thread.....	28
5 The main() Function (“Superloop”).....	29
6 Internal QWin GUI Implementation.....	30
6.1 The qwin_gui.h Header File.....	30
6.1 The qwin_gui.c Source File.....	32
7 Contact Information.....	33

Legal Disclaimers

Information in this document is believed to be accurate and reliable. However, Quantum Leaps does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information.

Quantum Leaps reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

All designated trademarks are the property of their respective owners.

1 Introduction

This Application Note describes a simple, free software GUI toolkit, called **QWin™**, for prototyping embedded systems on Windows in the C or C++ programming language, including building realistic embedded front panels consisting of buttons, LEDs, and LCD displays (both segmented and graphic). The described implementation is based on the **raw Win32 API** to provide direct mapping to C for easy integration with embedded code.



1.1 About the QWin™ GUI

The QWin GUI is available as part of the [QTools Collection](#) and is also included in the [QP Frameworks](#) (in the Win32 ports). QWin consists of just two files: `qwin_gui.h` containing the interface and `qwin_gui.c` providing the implementation. Currently these files provide the following facilities:

- **Graphic displays** (pixel-addressable) such as graphical LCDs, OLEDs, etc. with up to 24-bit color
- **Segmented displays** such as segment LCDs, and segment LEDs with generic, custom bitmaps for the segments.
- **Owner-drawn buttons** with custom “depressed” and “released” bitmaps and capable of generating separate events when depressed and when released.

Additionally, the provided code shows how to handle input sources:

- **Keyboard** events
- **Mouse move** events and **mouse-wheel** events

1.2 About the Development Tools

The QWin GUI kit components have been prepared and tested with the following **free** Windows tools:

- Microsoft Visual C++ Community Edition
- Additionally, to demonstrate that the same code runs both in the Win32 emulation and in a deeply embedded target, the example code uses the free GNU-ARM toolset (included in the QTools Collection).

1.3 Licensing

The QWin GUI Kit is licensed under the permissive open source [MIT](#) license as endorsed by the [Source Initiative](#).



2 Getting Started

The QWin GUI Kit is part of the [QTools Collection](#) and is also included in the QP distributions (in the Win32 ports). This section describes how to install the QWin GUI Kit, as it is packaged in the [QTools Collection](#) for Windows, and how to build, run, and debug the example demo application included in the toolkit.

2.1 Downloading and Installing QWin from QTools Collection

You can download QTools collection for Windows it from [GitHub](#). You can install QTools into any directory on your hard drive, but the recommended default is `C:\qp\qtools`. In any case, it is recommended not to use directories with spaces or special characters. The following [Listing 1](#) shows directories and files included in this ZIP file.

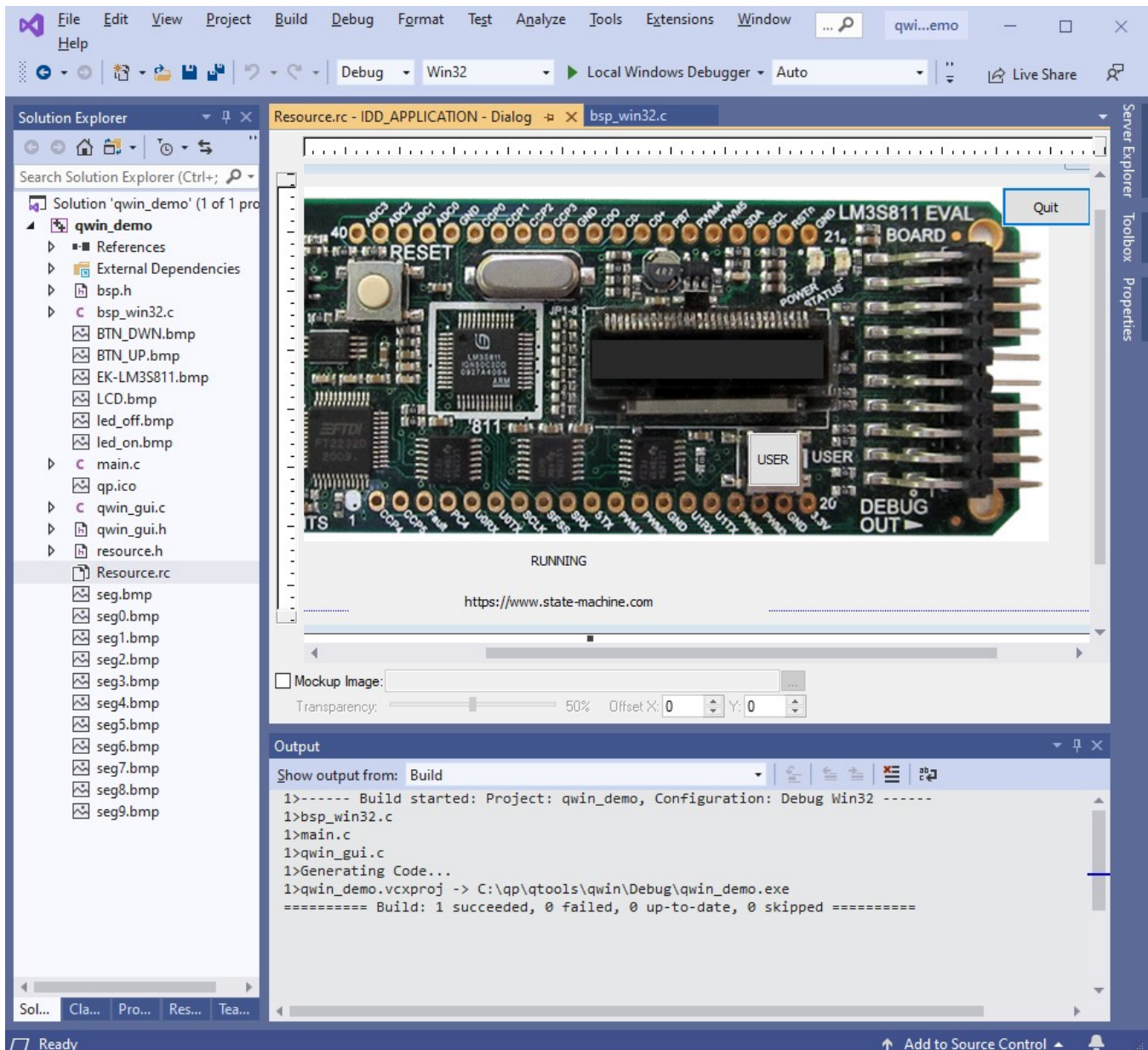
Listing 1: Selected directories and files in the QTools Collection

```
qtools\  
+-bin\                - executables (e.g., make.exe, qspy.exe, etc.)  
+-...  
+-qspy               - QSPY software tracing system (host support)  
+-...  
+-qwin\              - QWin GUI Toolkit  
| +-...  
| +-ek-lm3s811\  
| | +-bsp_lm3s811.c   - Board Support Package implementation for LM3S811 board  
| | +-display96x16x1.c - Implementation of the OLED display of LM3S811 board  
| | +-display96x16x1.h - Interface of the OLED display of LM3S811 board  
| | +-Makefile        - Makefile for GNU-ARM  
| | +-qwin_demo.ewp    - IAR Embedded Workbench project file  
| | +-qwin_demo.eww    - IAR Embedded Workbench workspace (open in IAR EWARM)  
| | +-...  
| |  
| +-bsp.h            - Board Support Package interface  
| +-bsp_win32.c       - Board Support Package implementation for Win32  
| +-main.c           - main function (the same for embedded and Win32 app)  
| +-qwin_demo.sln     - Visual Studio C++ solution  
| +-qwin_demo.vcxproj - Visual Studio C++ project  
| +-qwin_gui.h        - Interface for the QWin GUI  
| +-qwin_gui.c        - Implementation of the QWin GUI  
| +-Resource.rc       - Resource file for the example  
| +-resource.h        - Resource header file
```


2.2 Building the QWin Demo with the Visual Studio C++Toolset

To build the project with Visual Studio C++ toolset (needs to be downloaded from Microsoft), open the Visual Studio IDE and load the provided solution file `qwin_demo.sln`. Build it by pressing `F7`. This should build the code in the sub-directory `Debug`.

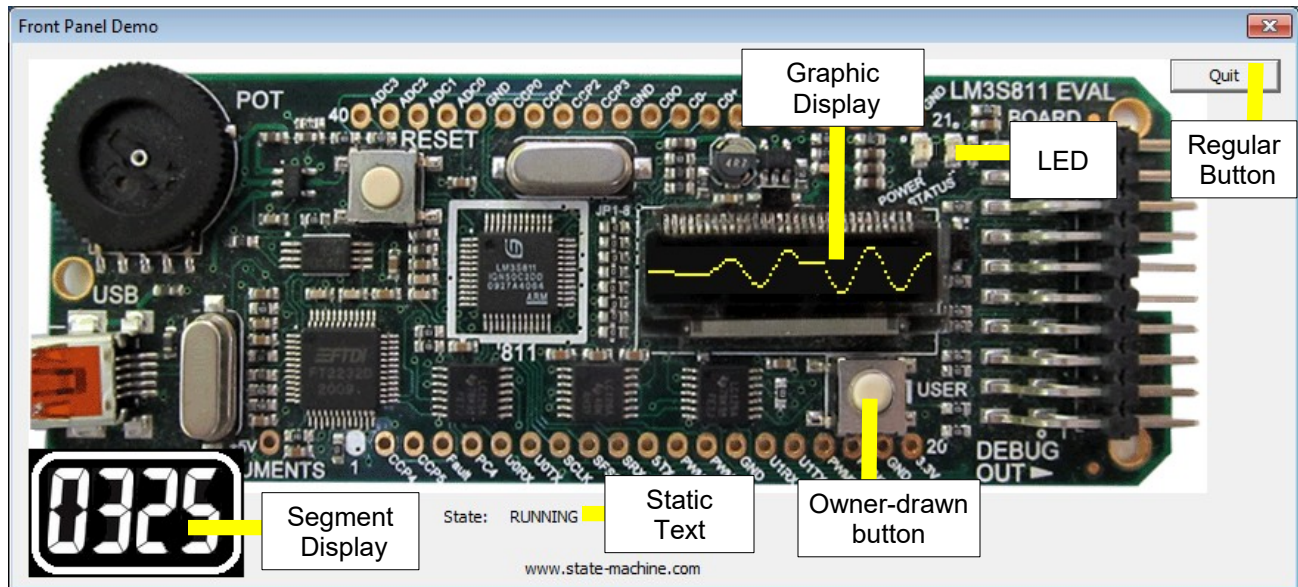
Figure 1: Building the `qwin_demo` project with Visual Studio 2019



2.3 Running the QWin GUI Demo on Windows

After the build, the Windows `qwin_demo.exe` is located in the directory `qtools\qwin\Debug\`. At this point you can either launch the application from the Visual Studio debugger (toolbar button "Local Windows Debugger") or you can double-click on the executable in the Windows Explorer. When the program starts running, you should see the application shown in the following screen shot:

Figure 2: QWin GUI demo of the EK-LM3S811 board running in Windows (`qwin_demo.exe`)



As shown in [Figure 2](#), the QWin GUI demo shows the picture of the Texas Instruments EK-LM3S811 board with a small graphic display (monochrome OLED, 96x16 pixels), an owner-drawn button, an LED, and an additional segment display in the corner. The OLED display shows the scrolling waveform, which moves from right to left at a frame rate of 30 frames per second. Also the segment display counts the frames (30 increments per second) and stops counting when the application is "Paused" (see the next paragraph).

The demo is interactive. When you click on the USER button and hold the left mouse button depressed, the application enters the PAUSED mode in which the display shows the text "PAUSED", the segment display stops counting, and the STATUS LED in the corner lights up. When you release the mouse button, the application gets back to the RUNNING mode, in which it continues to scroll the waveform.

The demo demonstrates also the **keyboard interface**. By pressing the SPACE bar on your PC, you enter the PAUSED state and pressing SPACE again you go back to RUNNING.

Finally, the demo demonstrates the **mouse-wheel interface**. By scrolling the mouse-wheel forward you enter the PAUSED state and by scrolling the mouse-wheel backward you go back to RUNNING.

NOTE: The QWin GUI demo for Windows uses the exact **same** application code (`main.c`) as the embedded project running on the actual target board EK-LM3S811, as described in [Section 2.4](#).

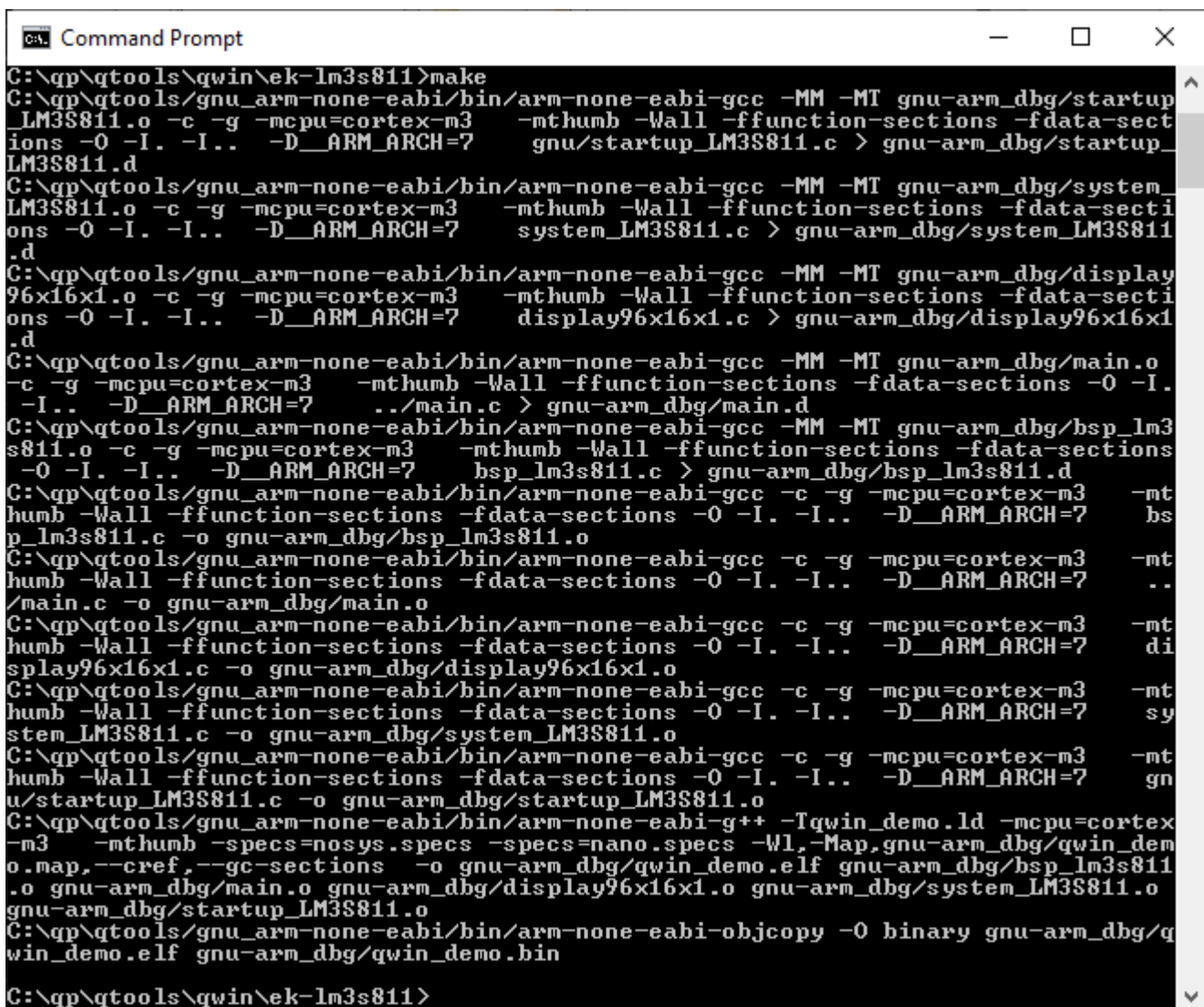
2.4 Running the QWin GUI Demo on the EK-LM3S811 Board

The code accompanying this App Note contains also the GNU-ARM Makefile and IAR-ARM workspace and project to run on the actual EK-LM3S811 board (see directory `ek-lm3s811\` in [Listing 1](#)).

2.4.1 GNU-ARM Toolset

To build the project with GNU-ARM toolset (included in the QTools collection for Windows), open Command-Prompt and change to the directory `ek-lm3s811\`. Type `make`. This should build the code in the sub-directory `gnu-arm_dbg`. Once you build the code, you can upload it to the EK-LM3S811 board by means of the LMFlash utility (available from TI).

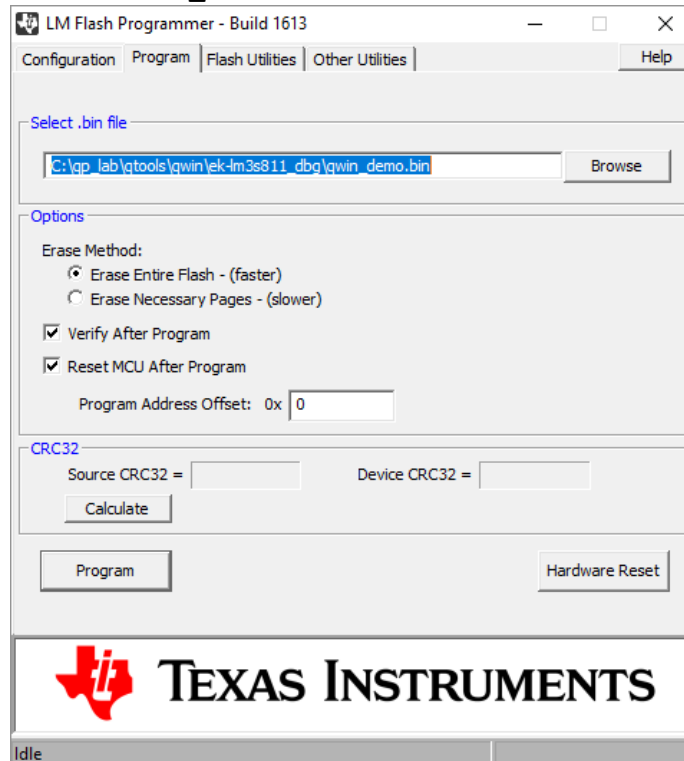
Figure 3: Building `qwin_demo` with GNU-ARM



```

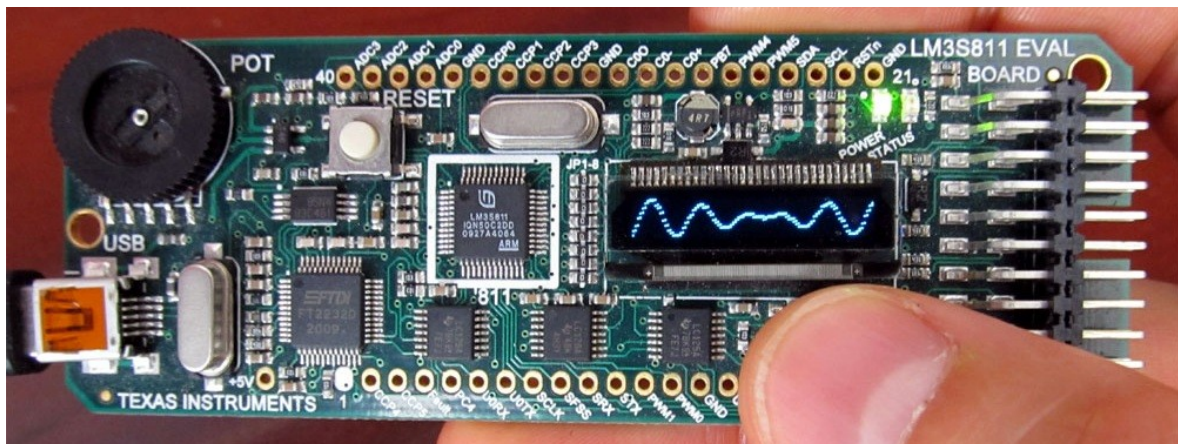
C:\qp\qtools\qwin\ek-lm3s811>make
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -MM -MT gnu-arm_dbg/startup_LM3S811.o -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 gnu/startup_LM3S811.c > gnu-arm_dbg/startup_LM3S811.d
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -MM -MT gnu-arm_dbg/system_LM3S811.o -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 system_LM3S811.c > gnu-arm_dbg/system_LM3S811.d
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -MM -MT gnu-arm_dbg/display96x16x1.o -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 display96x16x1.c > gnu-arm_dbg/display96x16x1.d
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -MM -MT gnu-arm_dbg/main.o -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 ../main.c > gnu-arm_dbg/main.d
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -MM -MT gnu-arm_dbg/bsp_lm3s811.o -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 bsp_lm3s811.c > gnu-arm_dbg/bsp_lm3s811.d
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 bsp_lm3s811.c -o gnu-arm_dbg/bsp_lm3s811.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 ../main.c -o gnu-arm_dbg/main.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 display96x16x1.c -o gnu-arm_dbg/display96x16x1.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 system_LM3S811.c -o gnu-arm_dbg/system_LM3S811.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-gcc -c -g -mcpu=cortex-m3 -mthumb -Wall -ffunction-sections -fdata-sections -O -I. -I.. -D__ARM_ARCH=7 gnu/startup_LM3S811.c -o gnu-arm_dbg/startup_LM3S811.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-g++ -Tqwin_demo.ld -mcpu=cortex-m3 -mthumb -specs=nosys.specs -specs=nano.specs -Wl,-Map,gnu-arm_dbg/qwin_demo.map,-cref,-gc-sections -o gnu-arm_dbg/qwin_demo.elf gnu-arm_dbg/bsp_lm3s811.o gnu-arm_dbg/main.o gnu-arm_dbg/display96x16x1.o gnu-arm_dbg/system_LM3S811.o gnu-arm_dbg/startup_LM3S811.o
C:\qp\qtools\gnu_arm-none-eabi\bin\arm-none-eabi-objcopy -O binary gnu-arm_dbg/qwin_demo.elf gnu-arm_dbg/qwin_demo.bin
C:\qp\qtools\qwin\ek-lm3s811>
  
```


Figure 4: Uploading qwin_demo.bin to the EK-LM3S811 with LMFlash



As shown in the picture of the EK-LM3S811 below, you can interact with the application by means of the USER button. Depressing the button enters the PAUSED mode, while releasing the button goes back to the RUNNING mode.

Figure 5: QWin Demo running on the actual EK-LM3S811 board



2.4.2 IAR-ARM Toolset

To download the project to the board, you need to open the workspace `ek-lm3s811\qwin_demo.eww` in the IAR EWARM IDE and after attaching the EK-LM3S811 board to the USB input of your host computer you can download the code to the board

3 Developing Embedded Code on Windows

The main purpose of this QWin GUI is to enable embedded engineers developing the embedded C or C++ code as much as possible on a Windows PC with fast and powerful tools, such as Visual Studio C++ in order to avoid the “target hardware bottleneck”.

NOTE: This section assumes that you have installed Visual Studio C++ on your Windows PC.

3.1 Dual Targeting

The general strategy for developing software on one platform and running it on another is known as **Dual Targeting**. This strategy is getting increasingly popular in the Test Driven Development (TDD) community, as described in the book “Test Driven Development for Embedded C” by James Grenning.

NOTE: Generally, the TDD community is mostly interested in **unit testing** of individual components, which works best with simple console applications without a GUI. However, the QWin GUI kit addresses a different need, which is developing the **complete** embedded applications, not just components, with non-trivial user interfaces.

Dual targeting means that from day one, your embedded code is designed to run on at least two platforms: the final target hardware and your PC. This in turn requires a specific way of designing the embedded software such that any target dependencies are handled through a well-defined interface often called the Board Support Package (BSP). This interface has at least two implementations: one for the actual target and one for the PC, such as Win32 API. With such interface in place, the remaining bulk of the embedded code can remain completely unaware which BSP implementation it is linked to and so it can be developed quickly on the PC, but can also run on the target hardware **without any changes**.

While some embedded programmers can view dual targeting as a self-inflicted burden, the more experienced developers generally agree that paying attention to the boundaries between software and hardware is actually beneficial, because it results in more modular, more portable, and more maintainable software with much longer useful lifetime. The investment in dual targeting has also an immediate payback in the vastly accelerated compile-run-debug cycle, which is much faster and more productive on the powerful PC compared to much slower, recourse-constrained deeply embedded target with limited visibility into the running code.

3.2 General Software Structure

The general software structure of a Windows emulation of an embedded device consists of at least two **Win32 threads**. The primary thread executes the traditional Windows message loop (the `WinMain()` function), while the simulated embedded application executes in a **secondary Win32 thread** spawned from the original thread after the initialization. This multithreaded architecture allows the embedded code to be exactly the same as in the embedded target. For example, the embedded code typically defines the `main()` function, which is executed in the secondary Win32 thread.

For simplicity, the `main()` function of the provided QWin GUI demo (see file `main.c`) is structured as the venerable “superloop”. The primitive “superloop” structure is actually not the best way of executing code on the desktop, because a “superloop” tends to take all the CPU cycles that it gets, which means that the CPU core (in a multi-core Windows PC) assigned to the secondary thread will run at 100% CPU capacity. A simple remedy to reduce this CPU usage is inserting a time delay into the “superloop” (which in the BSP for Windows calls the `Sleep()` Win32 API), and the provided example demonstrates just this. Also, the communication between the Win32 GUI thread (which plays the role of the interrupt level) with the background “superloop” occurs through global variables, which is not very robust.

However, the Windows emulation can also execute embedded code based on an RTOS or a real-time framework. For this to work, you need to have a **Windows emulation** of the RTOS or the framework, in which RTOS tasks are mapped to the Win32 threads with the proper mutual exclusion mechanisms for protecting any shared resources. Such architectures work actually better for Windows emulations.

NOTE: The [QP real-time embedded frameworks](#) provide an example of architecture ideal for prototyping on Windows. To this end, all QP frameworks (QP/C, QP/C++ and QP-nano) come with QWin GUI examples that run on Windows with almost no loading of the CPU. The screen shot below shows an interactive prototype of a “Fly ‘n’ Shoot” game included in all QP distributions, which runs also both on Windows and on the actual embedded board (EFM32-SLSTK from Silicon Labs in this case).

Figure 6: “Fly ‘n’ Shoot” game included in the QP frameworks



3.3 Dialog Box GUI

The general structure of a GUI application for building embedded front panels is based on a **dialog box**, which can be designed **graphically** within Visual Studio IDE. To provide a wider range of inputs, the dialog box is not serviced by the traditional dialog procedure (DialogProc), but rather by a generic Window Procedure (WindProc). This is achieved by assigning a custom “Windows Class” to the dialog box at design time (see Section 4.2).

3.1 Main Steps for Creating an Embedded Front Panel GUI

The recommended steps for developing embedded software with realistic GUI front panel on Windows are as follows:

NOTE: The described procedure is based on copying and modifying the existing example to preserve the project settings, such as graphical resources (file **Resource.rc**). The described procedure assumes that you have installed Visual Studio C++ Community Edition.

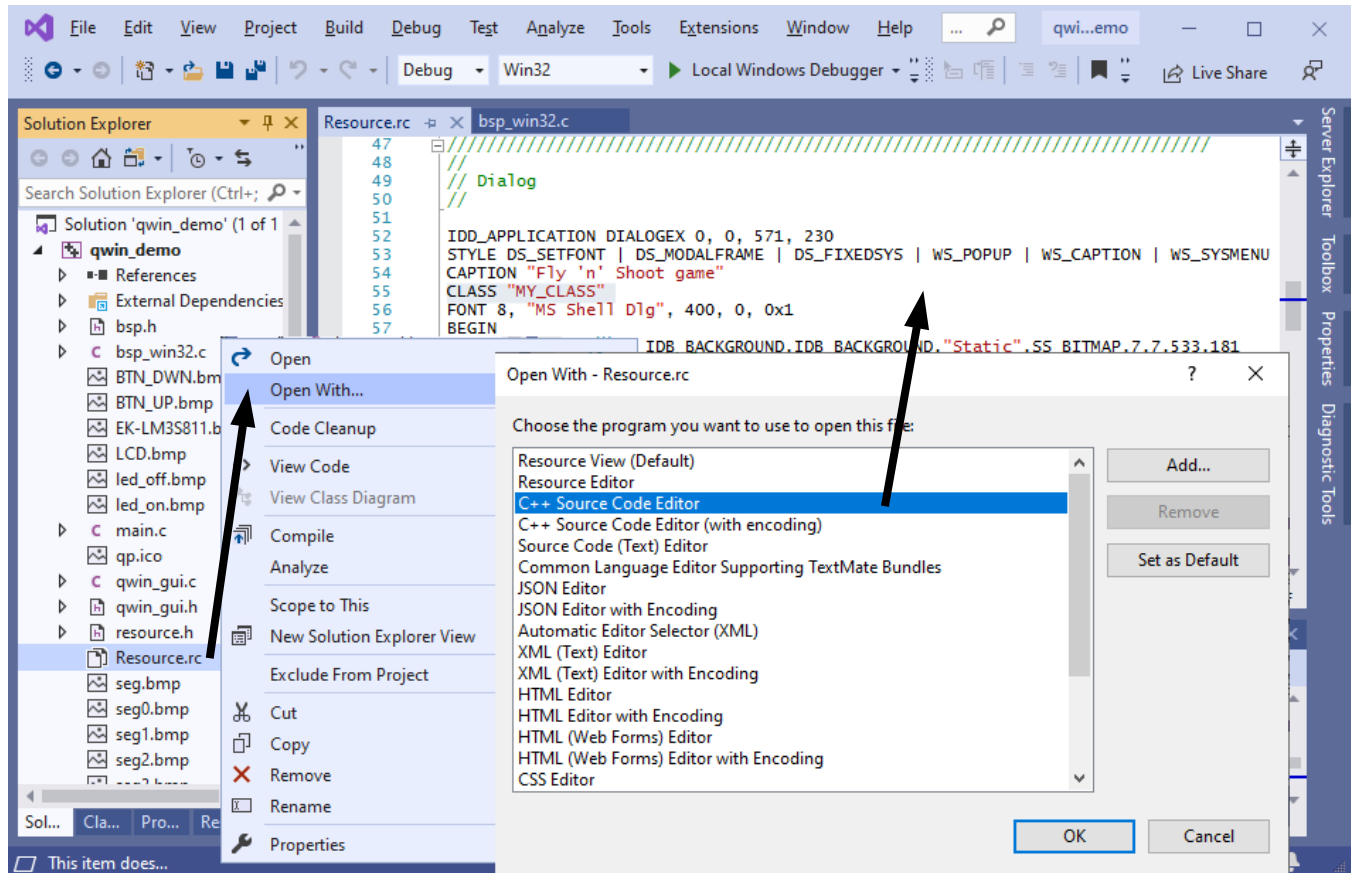
1. Copy the example GUI application directory (e.g., `qwin\`) and rename the copy to your own project.
2. Remove `*.sdf`, `*.suo`, and `*.vcxproj.user` files from the project directory (if present).
3. Rename the `.sln` (solution) file to the desired name of your project (leave the `.sln` extension). Open the renamed solution file in a text editor and replace all occurrences of the original example name with the your project name. Save the solution file.
4. Rename the `.vcxproj` (project) file to the desired name of your project (leave the `.vcxproj` extension). Open the renamed project file in a text editor and automatically replace all occurrences of the original example name with the your project name. Save the project file.
5. Open the solution file in Visual Studio C++.
6. Open the resource file in the resource editor (`Resource.rc`) and modify the front panel to your project. This step typically requires working with bitmaps, which you can either generate from digital pictures of your embedded front panel, or from your sketches. The upcoming Section describes the design of the front panel in the Visual Studio Resource Editor.
7. Adapt the `bsp_win32.c` file to handle the GUI events from your dialog box (in the `WindProc()`).

3.2 The Dialog Box Resource

The structure of a GUI application for building embedded front panels is assumed to be based on a **dialog box**, which can be designed in a resource editor included in Visual Studio.

NOTE: Due to the limitations of the Visual Studio resource editor, some necessary design steps need to be performed by editing the `Resource.rc` file manually in a text editor, as shown in [Figure 7](#).

Figure 7: Viewing the Resource.rc file in a text editor.



3.2.1 The Background Image

First, you typically want to add a background image of your embedded system to the dialog box, such as the EK-LM3S811 board shown in [Figure 2](#). The easiest way to use your own background image is to change the bitmap file associated with the `IDB_BACKGROUND` bitmap resource in the `Resource.rc` file:

```
////////////////////////////////////  
//  
// Bitmap  
//  
IDB_BACKGROUND          BITMAP          "Res\\EK-LM3S811.bmp"  
. . .
```

The most important aspect of preparing the background image is the proper **scaling**. You should scale the image carefully such that any graphic display present on the panel contains the desired number of pixels. Please note that to accommodate small pixel-count embedded displays, the provided GUI facilities for rendering graphic displays allow you to **scale** the pixels independently in the horizontal and vertical directions. For example, the OLED display of the EK-LM3S811 board has resolution of 96 x 16 pixels, which is a very small area on the modern high-resolution monitors. Therefore, the image of the board has been scaled such that the OLED display contains 192 x 32 pixels, which is exactly two times bigger than the original resolution.

NOTE: The background image needs to be converted to the BMP format, which can be done by a number of programs, for example the Paint utility bundled in every version of Windows. Save the image as a bitmap in the `Res\` directory of your project.

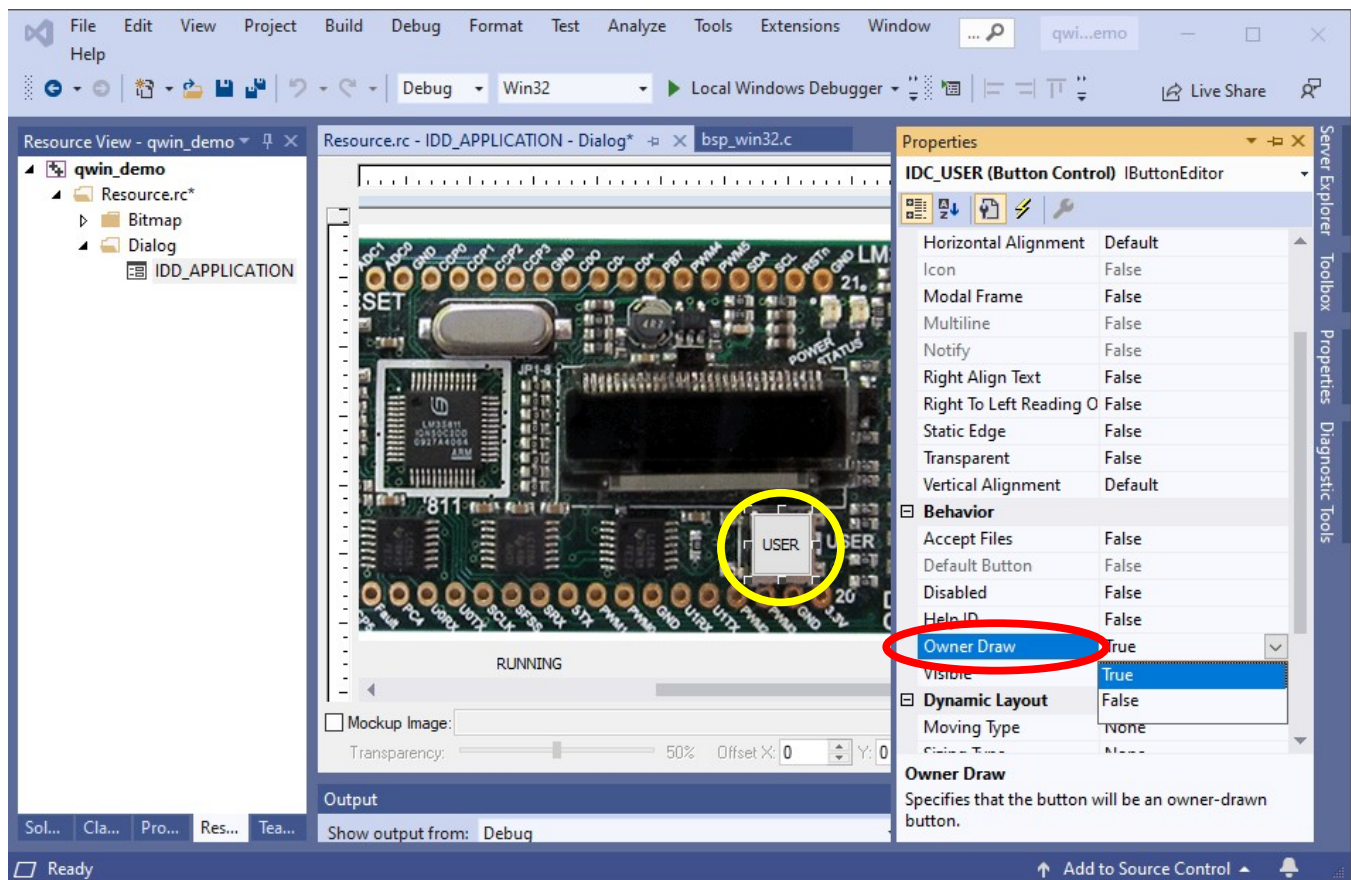
NOTE: Unfortunately, the Visual Studio resource editor does not allow you to control the **stacking order (a.k.a. z-order)** of the elements in the dialog box, so your background image might show in front of some other elements of your dialog box. However, the final stacking order is determined by the order of elements in the **Dialog resource code** (see [Listing 7](#)), regardless of how Visual Studio resource editor shows it.

3.3 Owner-Drawn Buttons

Owner-drawn buttons have two critical advantages over the regular buttons when it comes to embedded front panels. (1) they can **look** exactly as your physical buttons on your panel and, more importantly, (2) they generate events when they are **depressed** and when they are **released**. In contrast, the regular buttons are always rectangular-gray and they generate only the “clicked” event, which appears as WM_COMMAND in the WndProc.

To add an owner-drawn button to the dialog box, select the “Button” control from the Toolbox and place in the desired spot. Next, change the property “Owner drawn” to True.

Figure 8: Adding an Owner-Drawn button in Visual Studio resource editor



NOTE: The design of a front panel requires quite precise placement of the controls. To achieve the **fine control of placement**. The final fine-tuning of the position and size of the owner-drawn button can be adjusted in the **Dialog resource code** (see [Listing 7](#)).

The owner-drawn button requires also some code. which is illustrated in [Listing 2](#).

Listing 2: Implementing an owner-drawn button (file bsp_win32.c)

```
#include "qwin_gui.h"
```

```
(1) static OwnerDrawnButton l_userBtn; /* USER button of the EK-LM3S811 board */
    . . .
```

```

static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                WPARAM wParam, LPARAM lParam)
{
    switch (iMsg) {
        case WM_CREATE: {
            . . .
(2)      OwnerDrawnButton_init(&l_userBtn, IDC_USER,
                                LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_UP)),
                                LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_DWN)),
                                LoadCursor(NULL, IDC_HAND));
            return 0;
        }

        /* commands from child controls and menus... */
        case WM_COMMAND: {
            . . .
            switch (wParam) {
                case IDC_USER: { /* owner-drawn button(s) */
(3)      SetFocus(hWnd);
                    break;
                }
            }
            return 0;
        }

        /* drawing owner-drawn buttons... */
        case WM_DRAWITEM: {
            LPDRAWITEMSTRUCT pdis = (LPDRAWITEMSTRUCT)lParam;
            switch (pdis->CtlID) {
                case IDC_USER: { /* USER owner-drawn button */
(4)      switch (OwnerDrawnButton_draw(&l_userBtn, pdis)) {
                            case BTN_DEPRESSED: {
                                BSP_playerTrigger();
                                SegmentDisplay_setSegment(&l_userLED, 0U, 1U);
                                break;
                            }
                            case BTN_RELEASED: {
                                SegmentDisplay_setSegment(&l_userLED, 0U, 0U);
                                break;
                            }
                            default {
                                break;
                            }
                        }
                }
            }
            break;
        }
        return 0;
    }
    . . .
}

```

- (1) You need to provide an instance of the `OwnerDrawnButton` struct for each owner-drawn button you have added to the Dialog box.

- (2) You need to call `OwnerDrawnButton_init()` in the `WM_CREATE` message handler with the bitmaps for the “Released” state, “Depressed” state, and the mouse cursor when the mouse hovers above the button. You can specify the cursor shape to `NULL`, in which case the standard arrow cursor is used.
- (3) You need to call `SetFocus(hWnd)` in the `WM_COMMAND` message handler to set the keyboard focus for the parent window of the owner-draw button.
- (4) You need to call `OwnerDrawButton_draw()` in the `WM_DRAWITEM` message handler to render the button in the current state.

3.4 Graphic Displays

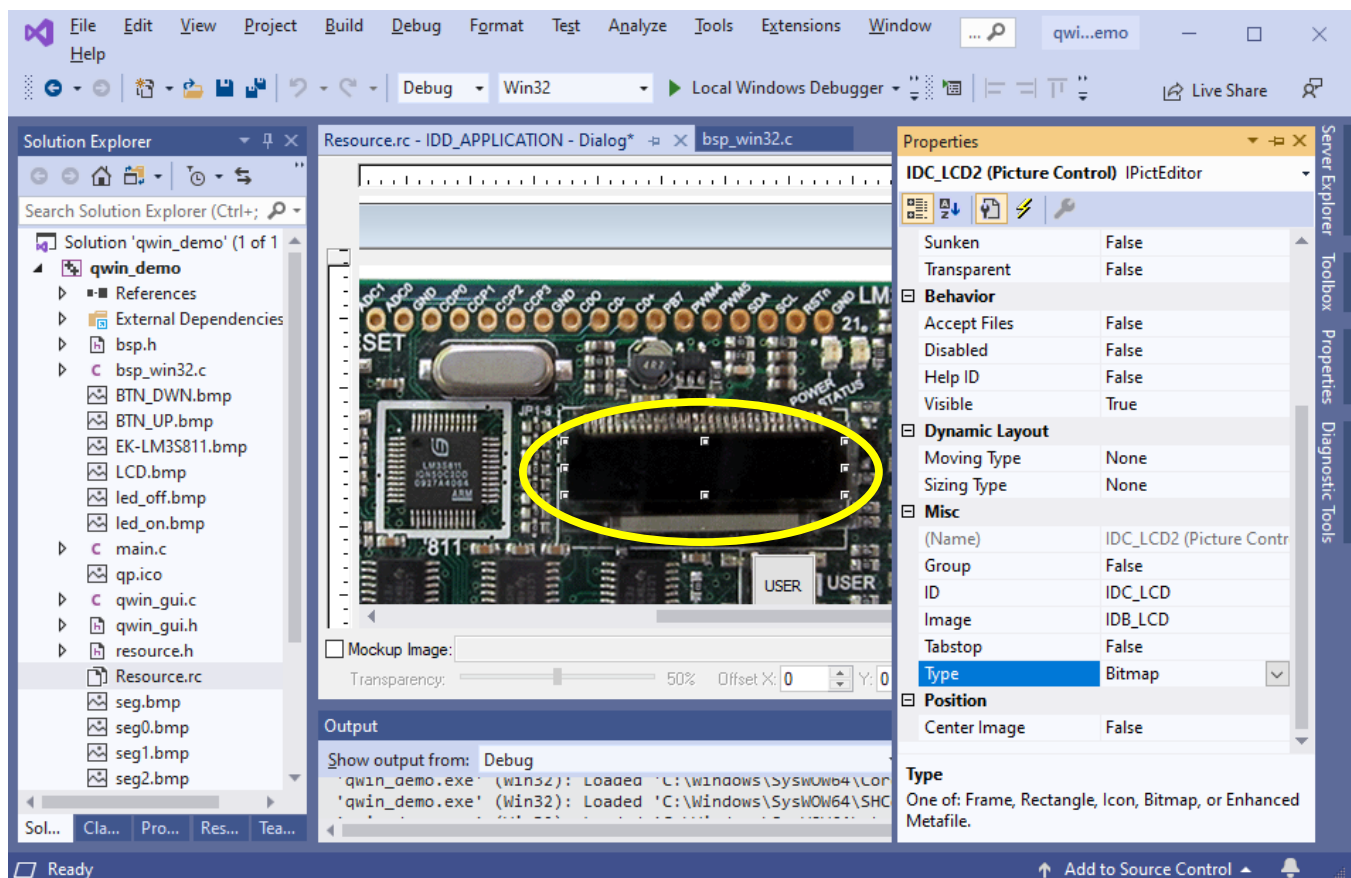
Graphic displays of various types (LCDs, OLED, etc.) become increasingly popular in embedded systems. The QWin GUI kit provides a generic, efficient implementation of a pixel-addressable graphic display with up to 24-bit color (RGB).

The use model of the provided graphic display is to perform multiple, efficient pixel updates in-memory and then render the updated bitmap to the screen. Also, to accommodate low-resolution displays, the graphic facility allows **scaling** the display independently in horizontal and vertical directions. For example, the OLED display of the EK-LM3S811 board has been scaled up by factor of 2 horizontally and 2 vertically, to the total size of 192 x 32 pixels (each original pixel corresponding to 2x2 pixels on the graphic display.) Of course, the 2x2 scaling is just an example and you can use other scaling factors.

Under the hood, the implementation is based on the Device-Independent Bitmap (DIB) section of an in-memory bitmap (`CreateDIBSection()` Win32 API, see `qwin_gui.c` source file in the QWin GUI.) The `GraphicDisplay` bitmap uses the **standard coordinate system**, in which the origin (0,0) is placed in the top-left corner, the x-coordinate extends horizontally and grows to the right, while the y-coordinate extends vertically and grows down.

To add a graphic display to the dialog box in the resource editor, select the “Picture Control” from the Toolbox and place in the desired spot. In the Property Editor of the newly added Picture Control, you change the type to “Bitmap” and the ID to the carefully scaled LCD bitmap prepared in the pervious step.

Figure 9: Adding an an LCD as a Bitmap to the dialog ox



The graphic display requires also some code, which is illustrated in [Listing 3](#).

Listing 3: Implementing a graphic display

```
#include "qwin_gui.h"

(1) static GraphicDisplay l_oled; /* the OLED display of the EK-LM3S811 board */
/* (R,G,B) colors for the OLED display */
(2) static BYTE const c_onColor [3] = { 255U, 255U, 0U }; /* yellow */
(3) static BYTE const c_offColor[3] = { 15U, 15U, 15U }; /* very dark gray */
. . .
/*.....*/
static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                WPARAM wParam, LPARAM lParam)
{
    switch (iMsg) {
        /* Perform initialization after all child windows have been created */
        case WM_INITDIALOG: {
(4)            GraphicDisplay_init(&l_oled,
                                BSP_SCREEN_WIDTH, BSP_SCREEN_HEIGHT,
                                IDC_LCD, c_offColor);
            . . .
        }
        /*.....*/
(5) void BSP_drawBitmap(uint8_t const *bitmap) {
    UINT x, y;
    /* map the EK-LM3S811 OLED pixels to the GraphicDisplay pixels... */
    for (y = 0; y < BSP_SCREEN_HEIGHT; ++y) {
        for (x = 0; x < BSP_SCREEN_WIDTH; ++x) {
            uint8_t bits = bitmap[x + (y/8)*BSP_SCREEN_WIDTH];
            if ((bits & (1U << (y & 0x07U))) != 0U) {
(6)                GraphicDisplay_setPixel(&l_oled, x, y, c_onColor);
            }
            else {
(7)                GraphicDisplay_clearPixel(&l_oled, x, y);
            }
        }
    }
(8)    GraphicDisplay_redraw(&l_oled); /* re-draw the updated display */
}
```

- (1) You need to provide an instance of the `GraphicDisplay` struct for each graphic display you have added to the Dialog box.
- (2-3) To simulate a monochrome display, the colors for the “on” and “off” pixel states are defined.
- (4) You need to call `GraphicDisplay_init()` in the `WM_INITDIALOG` message handler with the dimensions of the display and the color of the blank pixels.
- (5) The `BSP_drawBitmap()` function renders the whole 96x16 pixel OLED display of the EK-LM3S811 board. This function performs a re-mapping of the specific pixel representation for the OLED display to the pixels of the `GraphicDisplay` device-independent bitmap. In this particular case, the monochrome pixels are represented as tightly-packed bitmasks of 8-pixels per byte arranged vertically.

NOTE: The particular pixel re-mapping is just a demonstration here. You need to adapt it for your specific display. Also, you don't need to always update the whole pixel array. You might choose to update only a part of it, as is illustrated in the function `BSP_drawString()` in `bsp.c` (not shown in [Listing 3](#)).

- (6) The macro `GraphicDisplay_setPixel()` very efficiently sets a given pixel at (x, y) to the given color.
- (7) The macro `GraphicDisplay_clearPixel()` very efficiently clears a given pixel at (x, y) by setting it to the blank color specified in `GraphicDisplay_init()`.

NOTE: The pixels are set/cleared only in memory and do not appear on the screen until the call to `GraphicDisplay_redraw()`.

- (8) The function `GraphicDisplay_redraw()` re-draws the current state of the in-memory bitmap on the screen.

NOTE: You can use the function `GraphicDisplay_clear()` to clear the entire internal device-independent bitmap by setting all the pixels of the to the blank color.

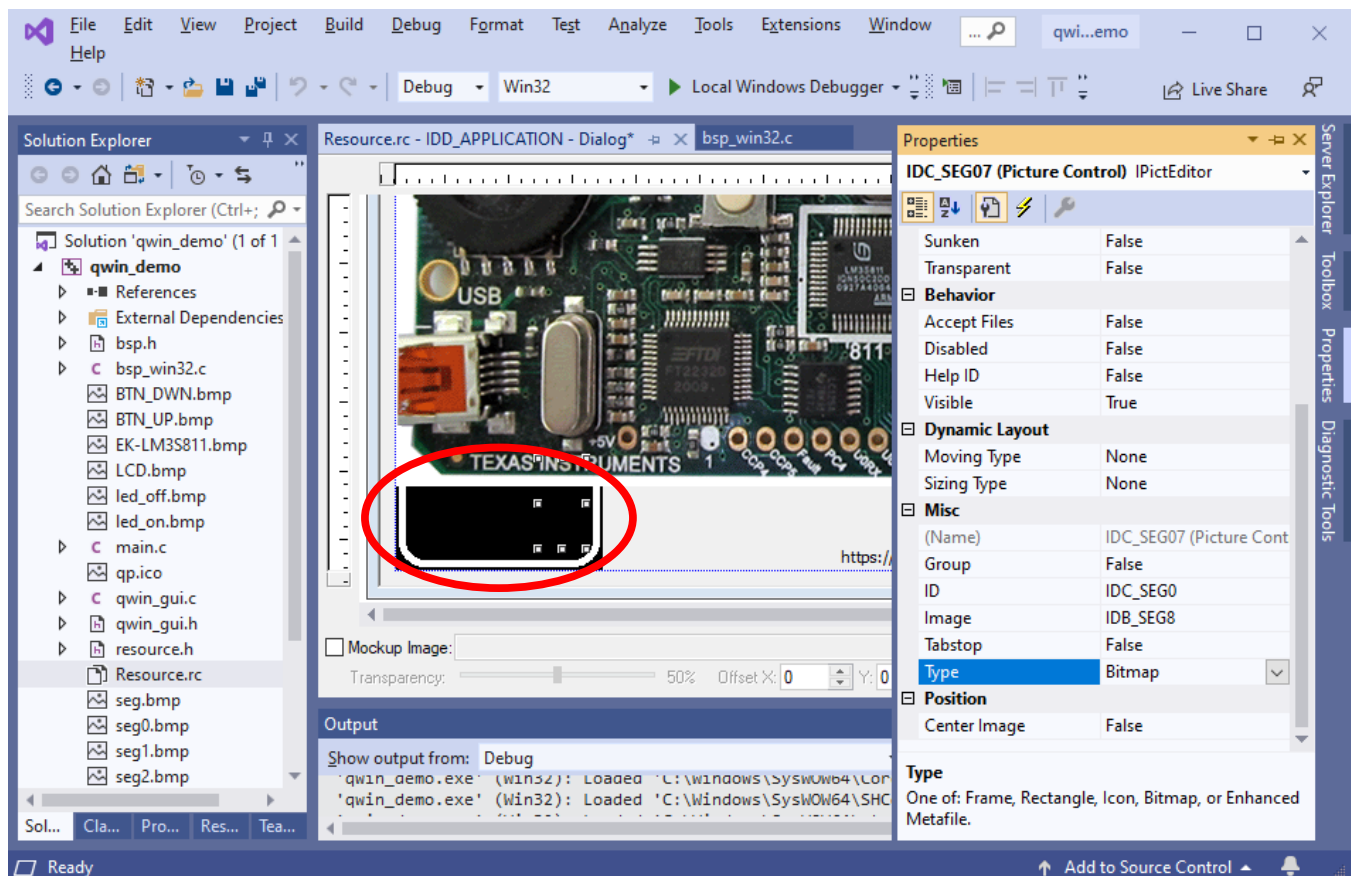
3.5 Segment Displays

The QWin GUI kit provides also a generic facility for rendering segment displays, such as found in digital watches, calculators, thermostats, microwave ovens, washing machines, digital meters, etc.

The chosen implementation is based on selective displaying bitmaps representing the various states of the related groups of segments. In this design, you need to provide all the bitmaps for all states of segment groups that you anticipate in your application, but the advantage is that the bitmaps can be very realistic and match exactly your specific display.

To add a segment display to the dialog box in the resource editor, you add all the bitmaps for the segments and a “Picture Control” for each segment group. You also need to place the Picture Controls at the exact locations of the segments. Please note that you add one Picture Control for the whole group of segments not for every individual segment. For example, as illustrated in [Figure 10](#), a single picture control represents a whole 7-segment digit. Also, as you keep adding the segment groups to the dialog box, you don't need to make sure that the resource IDs assigned to them are consecutive or in any specific order. The provided implementation of the Segment Display does not require any specific order.

Figure 10: Adding a segment display in the resource editor



NOTE: The design of a segment display requires quite precise placement of the bitmaps. To achieve the **fine control of placement**. The final fine-tuning of the position and size of the segments can be adjusted in the **Dialog resource code** (see [Listing 7](#)).

The segment display requires also some code. which is illustrated in [Listing 4](#).

Listing 4: Implementing a segment display

```
#include "qwin_gui.h"

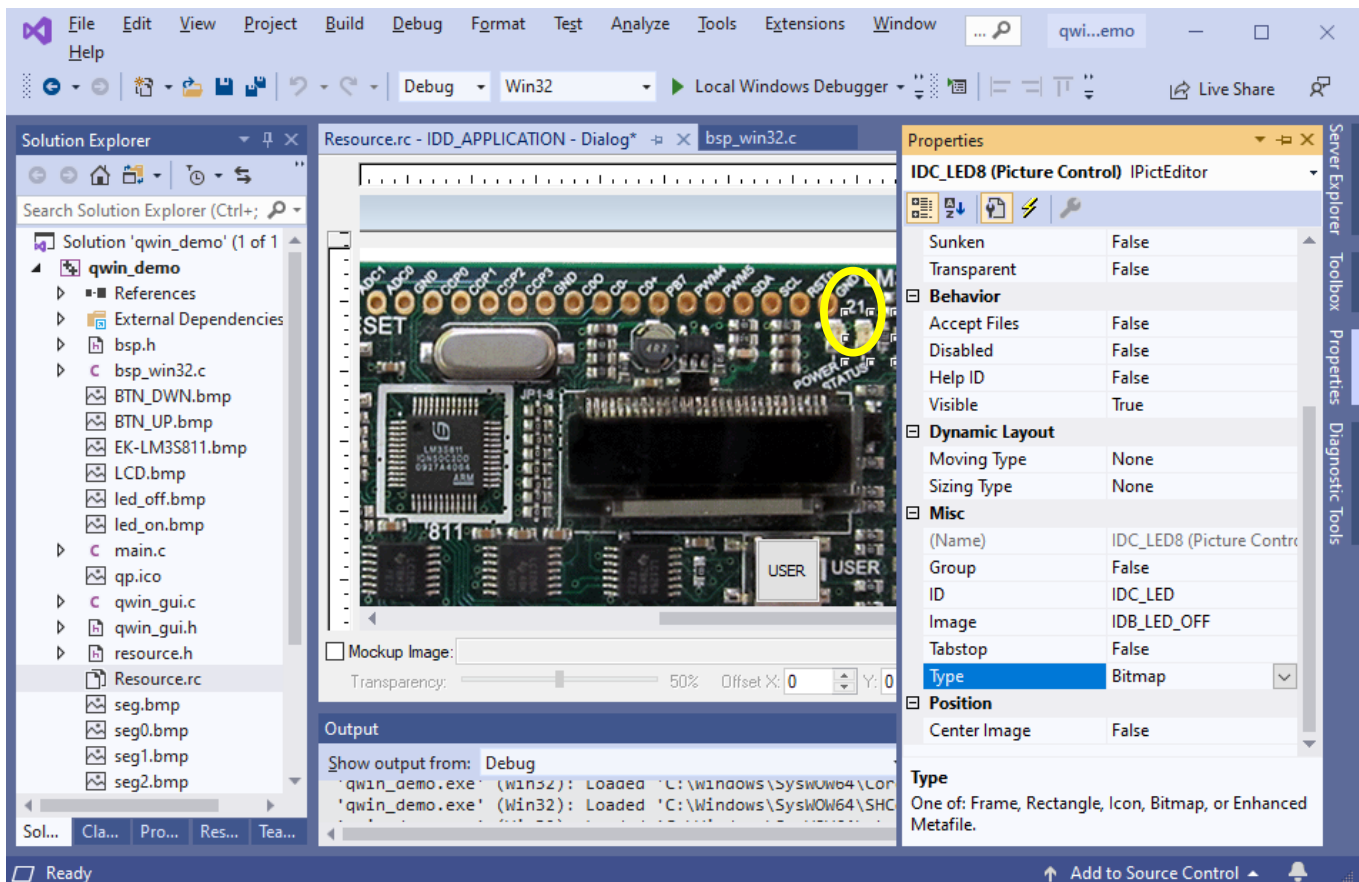
(1) static SegmentDisplay l_scoreBoard; /* segment display for the score */
/*.....*/
static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                WPARAM wParam, LPARAM lParam)
{
    switch (iMsg) {
        /* Perform initialization after all child windows have been created */
        case WM_INITDIALOG: {
            . . .
(2)         SegmentDisplay_init(&l_scoreBoard,
                                4U, /* 4 "segments" (digits 0-3) */
                                10U); /* 10 bitmaps (for 0-9 states) */
(3)         SegmentDisplay_initSegment(&l_scoreBoard, 0U, IDC_SEG0);
(4)         SegmentDisplay_initSegment(&l_scoreBoard, 1U, IDC_SEG1);
            . . .
(5)         SegmentDisplay_initBitmap(&l_scoreBoard,
                                       0U, LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SEG0)));
(6)         SegmentDisplay_initBitmap(&l_scoreBoard,
                                       1U, LoadBitmap(hInst, MAKEINTRESOURCE(IDB_SEG1)));
            . . .
        }
    }
    /*.....*/
    void BSP_updateScore(uint16_t score) {
        /* update the score in the l_scoreBoard SegmentDisplay */
(7)         SegmentDisplay_setSegment(&l_scoreBoard, 0U, (UINT)(score % 10U));
        score /= 10U;
(8)         SegmentDisplay_setSegment(&l_scoreBoard, 1U, (UINT)(score % 10U));
        score /= 10U;
(9)         SegmentDisplay_setSegment(&l_scoreBoard, 2U, (UINT)(score % 10U));
        score /= 10U;
(10)        SegmentDisplay_setSegment(&l_scoreBoard, 3U, (UINT)(score % 10U));
    }
}
```

- (1) You need to provide an instance of the `SegmentDisplay` struct for each segment display you have added to the Dialog box.
- (2) You need to call `SegmentDisplay_init()` in the `WM_INITDIALOG` message handler with the total number of segment groups and the total number of bitmaps associated with the display. For instance, the 4-digit `l_scoreBoard` display used in the game example uses 4 segment groups and 10 bitmaps (for digits 0..9).
- (3-4) For each segment group, you need to call `SegmentDisplay_initSegment()` with the index of the segment group and the dialog control corresponding to this segment group. This way, you establish a consecutive numbering of segments (by the index number).
- (5-6) For each bitmap, you need to call `SegmentDisplay_initBitmap()` with the index of the bitmap and the bitmap itself loaded from the resource pool. This way, you establish a consecutive numbering of bitmaps (by the index number).
- (7-10) You change the bitmaps in the segments by calling `SegmentDisplay_setSegment()` with the index of the segment and index of the bitmap.

3.6 LEDs

LEDs can be efficiently handled as segment displays with just one “segment group” (the LED itself) and two bitmaps (“off” and “on” states of the bitmap). This way, it is also very easy to build groups of LEDs and multi-color LEDs just by increasing the number of “segment groups” and bitmaps for each color.

Figure 11: Adding an LED in the resource editor



4 The Board Support Package

As described in the earlier Section 3.1, the best practice of “dual targeting” is to handle any target dependencies through a well-defined interface often called the Board Support Package (BSP). This interface has at least two implementations: one for the actual target and one for the PC, such as Win32 API. With such interface in place, the remaining bulk of the embedded code can remain completely unaware which BSP implementation it is linked to and so it can be developed quickly on the PC, but can also run on the target hardware **without any changes**.

4.1 The BSP Interface (bsp.h)

The `bsp.h` header file specifies the BSP interface, which is the same for the embedded target and for the Windows emulation. The following Listing shows the

Listing 5: The BSP interface for the demo application (file `bsp.h`).

```
(1) #define BSP_TICKS_PER_SEC    33U
    #define BSP_SCREEN_WIDTH    96U
    #define BSP_SCREEN_HEIGHT   16U

    void BSP_init(void);
(2) void BSP_sleep(uint32_t ticks);
    . . .

    /* special adaptations for QWin GUI applications */
(3) #ifndef QWIN_GUI
    /* replace main() with main_gui() as the entry point to a GUI app. */
(4)     #define main() main_gui()
(5)     int main_gui(); /* prototype of the GUI application entry point */
    #endif
```

- (1) The ticks-per-second constant defines how fast the “superloop” goes through each pass. This constant is convenient to implement timing delays in the application.
- (2) The actual implementation of the `sleep()` function depends on the environment. It can be implemented with polling in the “superloop” on the embedded target, but on Windows it can be efficient blocking, so that the Windows application won’t hog the CPU on your desktop machine.
- (3) The macro `QWIN_GUI` indicates that the BSP is for the Windows GUI application.

NOTE: The macro **QWIN_GUI** must be defined in the Windows GUI build, preferably on the command-line to the compiler.

- (4) When this macro is defined, the `main()` function is aliased to `main_gui()`.
- (5) The prototype of `main_gui()` is provided.

NOTE: Please note that the BSP interface avoids any explicit dependency on the Win32 API. This has a very nice side effect of **very fast compilation** of the platform-neutral source code, as the huge `<windows.h>` header file does not need to be processed in that portion of the project.

4.2 BSP for QWin GUI (WinMain)

In case of a Windows emulation, the Board Support Package implementation is located in the `bsp_win32.c` module. This module encapsulates completely the QWin GUI.

This section describes the `WinMain()`, which starts the Windows GUI application and executes the main event loop, as well as the Windows Procedure for the dialog box (`WndProc`), which handles the GUI events, such as button presses, mouse moves, etc. Listing 6 shows the relevant fragment of the `bsp_win32.c` file. This section describes the `WinMain()` function.

Listing 6: WinMain() (file `bsp_win32.c`).

```
(1) #include <stdint.h>
(2) #include "bsp.h"

(3) #include "qwin_gui.h" /* QWin GUI */
(4) #include "resource.h" /* GUI resource IDs generated by the resource editor */

/* local variables -----*/
static HINSTANCE l_hInst; /* this application instance */
(5) static HWND    l_hWnd; /* main window handle */
static LPSTR      l_cmdLine; /* the command line string */

(6) static GraphicDisplay l_oled; /* the OLED display of the EK-LM3S811 board */
(7) static SegmentDisplay l_userLED; /* USER LED of the EK-LM3S811 board */
(8) static SegmentDisplay l_scoreBoard; /* segment display for the score */
(9) static OwnerDrawnButton l_userBtn; /* USER button of the EK-LM3S811 board */

/* (R,G,B) colors for the OLED display */
static BYTE const c_onColor [3] = { 255U, 255U, 0U }; /* yellow */
static BYTE const c_offColor[3] = { 15U, 15U, 15U }; /* very dark gray */
...
/* Local functions -----*/
static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                WPARAM wParam, LPARAM lParam);

/*.....*/
(10) int WINAPI WinMain(HINSTANCE hInst, HINSTANCE hPrevInst,
                        LPSTR cmdLine, int iCmdShow)
{
    HWND hWnd;
    MSG msg;

    (void)hPrevInst; /* avoid compiler warning about unused parameter */

(11)    l_hInst = hInst; /* save the application instance */
(12)    l_cmdLine = cmdLine; /* save the command line string */

    /* create the main custom dialog window */
(13)    hWnd = CreateCustDialog(hInst, IDD_APPLICATION, NULL,
                              &WndProc, "MY_CLASS");
(14)    ShowWindow(hWnd, iCmdShow); /* show the main window */

    /* enter the message loop... */
(15)    while (GetMessage(&msg, NULL, 0, 0)) {
        TranslateMessage(&msg);
```



```
        DispatchMessage(&msg);  
    }  
    return msg.wParam;  
}
```

- (1-2) The module starts with including the standard integer types and board support package (BSP) interface `bsp.h`.
- (3-4) The GUI application includes the header file `qwin_gui.h` discussed in Section 6.1 and `resource.h` generated by the resource editor (such as ResEdit).
- (5) The main window handle is one of the most important local variables.
- (6-9) The GUI elements specific for your embedded panel, such as graphic displays, segmented displays, LEDs, and owner-drawn buttons are allocated statically. You can have any number of instances of each component type.
- (10) As in every Win32 GUI application, you need the `WinMain()` function, which is the main entry point to every GUI application on Windows.

NOTE: The structure of `WinMain()` is very standard as for any other typical Windows GUI application. In particular, the `WinMain()` function contains the standard “message pump”.

- (11-12) The current instance handle and the command line are saved in static variables.
- (13) The function `CreateCustDialog()` creates the main application dialog box window with the **customized Windows class** and regular window procedure (`WndProc`) as opposed to the dialog box procedure typical for dialog boxes.

NOTE: The created dialog box is based on a customized Windows class with the name specified as the last parameter ("`MY_CLASS`" in this case). For this to work, the dialog box resource must be associated with the same Windows class in the “Class Name” property of the resource editor, as shown in .

- (14) The created dialog box window must be shown.
- (15) The `WinMain()` function enters the standard “message pump”, in which it processes all messages until the application is closed.

NOTE: The “message pump” can be extended to handle keyboard accelerators and perhaps other events.

The Windows Class Name can also be visible directly in the resource file. Listing 7 shows the definition of the dialog box resource in the `Resource.rc` resource file. Please note the highlighted definition of the `CLASS "MY_CLASS"`.

Listing 7: The dialog resource in the QWin demo project (file `Resource.rc`).

```
////////////////////////////////////  
//  
// Dialog  
//  
  
IDD_APPLICATION DIALOGEX 0, 0, 571, 230  
STYLE DS_SETFONT | DS_MODALFRAME | DS_FIXEDSYS | WS_POPUP | WS_CAPTION | WS_SYSMENU  
CAPTION "Fly 'n' Shoot game"
```

```
CLASS "MY CLASS"
FONT 8, "MS Shell Dlg", 400, 0, 0x1
BEGIN
    CONTROL          IDB_BACKGROUND,IDB_BACKGROUND,"Static",SS_BITMAP,7,7,533,181
    DEFPUSHBUTTON     "Quit",IDOK,514,7,50,20
    CONTROL           "USER",IDC_USER,"Button",BS_OWNERDRAW | WS_TABSTOP,374,132,28,27
    CONTROL           IDB_LCD,IDC_LCD,"Static",SS_BITMAP,287,85,128,20
    CONTROL           IDB_SEG,IDC_STATIC,"Static",SS_BITMAP,7,169,97,54
    CONTROL           IDB_SEG8,IDC_SEG3,"Static",SS_BITMAP,16,177,19,36
    CONTROL           IDB_SEG8,IDC_SEG2,"Static",SS_BITMAP,36,177,19,36
    CONTROL           IDB_SEG8,IDC_SEG1,"Static",SS_BITMAP,56,177,19,36
    CONTROL           IDB_SEG8,IDC_SEG0,"Static",SS_BITMAP,76,177,19,36
    CONTROL           IDB_LED_OFF,IDC_LED,"Static",SS_BITMAP,417,36,19,18
    LTEXT              "RUNNING",IDC_PAUSED,254,193,65,8
    CTEXT              "https://www.state-machine.com",IDC_STATIC,153,214,232,10
END
~ ~ ~
```

4.3 BSP for QWin GUI (WndProc)

The `WinMain()` function creates the main dialog box window and associates the Window Procedure (traditionally called `WndProc`) with it (see [Listing 6\(13\)](#)). The main job of the `WndProc` is processing windows messages such as mouse clicks, button presses, or menu commands.

NOTE: Even though the QWin GUI example is based on a dialog box as the main window, the application uses a custom Windows class with a **regular Window Procedure** rather than a Dialog Procedure typically associated with dialog boxes. A standard Dialog Procedure is not used, because it is too specialized, which would be too limiting for the embedded front panels (e.g., the Dialog Procedure contains a specialized message loop that processes keyboard input, tab order, etc. and does not allow to override these aspects easily.)

Listing 8: WndProc() (file `bsp_win32.c`).

```
(1) static LRESULT CALLBACK WndProc(HWND hWnd, UINT iMsg,
                                     WPARAM wParam, LPARAM lParam)
{
    switch (iMsg) {
        /* Perform initialization upon creation of the main dialog window
        * NOTE: Any child-windows are NOT created yet at this time, so
        * the GetDlgItem() function can't be used (it will return NULL).
        */
        (2) case WM_CREATE: {
            (3)     l_hWnd = hWnd; /* save the window handle */

            /* initialize the owner-drawn buttons...
            * NOTE: must be done *before* the first drawing of the buttons,
            * so WM_INITDIALOG is too late.
            */
            (4)     OwnerDrawnButton_init(&l_userBtn, IDC_USER,
                                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_UP)),
                                           LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_BTN_DWN)),
                                           LoadCursor(NULL, IDC_HAND));

            return 0;
        }
    }
}
```

```
/* Perform initialization after all child windows have been created */
(5) case WM_INITDIALOG: {
(6)     GraphicDisplay_init(&l_oled,
                           BSP_SCREEN_WIDTH, BSP_SCREEN_HEIGHT,
                           IDC_LCD, c_offColor);

(7)     SegmentDisplay_init(&l_scoreBoard,
                           4U, /* 4 "segments" (digits 0-3) */
                           10U); /* 10 bitmaps (for 0-9 states) */
     SegmentDisplay_initSegment(&l_scoreBoard, 0U, IDC_SEG0);
     SegmentDisplay_initSegment(&l_scoreBoard, 1U, IDC_SEG1);
     . . .
     SegmentDisplay_initBitmap(&l_scoreBoard,
                               0U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG0)));
     SegmentDisplay_initBitmap(&l_scoreBoard,
                               1U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG1)));
     . . .
     SegmentDisplay_initBitmap(&l_scoreBoard,
                               9U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_SEG9)));

     BSP_updateScore(0U);

(8)     SegmentDisplay_init(&l_userLED,
                           1U, /* 1 "segment" (the LED itself) */
                           2U); /* 2 bitmaps (for LED OFF/ON states) */
     SegmentDisplay_initSegment(&l_userLED, 0U, IDC_LED);
     SegmentDisplay_initBitmap(&l_userLED,
                               0U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_LED_OFF)));
     SegmentDisplay_initBitmap(&l_userLED,
                               1U, LoadBitmap(l_hInst, MAKEINTRESOURCE(IDB_LED_ON)));

     /* --> Spawn the application thread to run main_gui() */
(9)     CreateThread(NULL, 0, &appThread, NULL, 0, NULL);
     return 0;
}

(10) case WM_DESTROY: {
     BSP_terminate(0);
     return 0;
}

/* commands from child controls and menus... */
(11) case WM_COMMAND: {
     switch (wParam) {
         case IDOK:
         case IDCANCEL: {
             BSP_terminate(0);
             break;
         }
         case IDC_USER: { /* owner-drawn button(s) */
             SetFocus(hWnd);
             break;
         }
     }
     return 0;
}
```

```
/* drawing owner-drawn buttons... */
(12) case WM_DRAWITEM: {
(13)     LPDRAWITEMSTRUCT pdis = (LPDRAWITEMSTRUCT)lParam;
(14)     switch (pdis->CtlID) {
(15)         case IDC_USER: { /* USER owner-drawn button */
(16)             switch (OwnerDrawnButton_draw(&l_userBtn, pdis)) {
(17)                 case BTN_DEPRESSED: {
                     BSP_playerTrigger();
                     SegmentDisplay_setSegment(&l_userLED, 0U, 1U);
                     break;
                 }
(18)                 case BTN_RELEASED: {
                     SegmentDisplay_setSegment(&l_userLED, 0U, 0U);
                     break;
                 }
             }
         }
     }
    break;
}
return 0;
}

/* mouse input... */
(19) case WM_MOUSEWHEEL: {
    if ((HIWORD(wParam) & 0x8000U) == 0U) { /* wheel turned forward? */
        BSP_moveShipUp();
    }
    else { /* the wheel was turned backwards */
        BSP_moveShipDown();
    }
    return 0;
}

/* keyboard input... */
(20) case WM_KEYDOWN: {
    switch (wParam) {
        case VK_UP:
            BSP_moveShipUp();
            break;
        case VK_DOWN:
            BSP_moveShipDown();
            break;
        case VK_SPACE:
            BSP_playerTrigger();
            break;
    }
    return 0;
}

(21) return DefWindowProc(hWnd, iMsg, wParam, lParam) ;
}
```

(1) The standard WndProc is made static here, because it is not used outside the bsp.c module.

- (2) The `WM_CREATE` Windows message is sent to the `WndProc` upon the creation of the associated window.
- (3) The main window handle is copied to the static variable to use in other BSP functions.
- (4) Any owner-drawn buttons must be initialized before they are drawn, which happens after `WM_CREATE` is sent to the `WndProc`. Please see the Windows message `WM_DRAWITEM` at label (12) and the upcoming Section 3.3 for more information about the owner-drawn buttons.
- (5) The `WM_INITDIALOG` Windows message is not typically sent to regular `WndProcs`, but it is specifically generated in the `CreateCustDialog()` function (see Listing 6(13)) after all the child window controls have been created. The `WM_INITDIALOG` message is very useful to initialize any dialog controls, such as buttons, bitmaps, text labels, etc.
- (6) The `WndProc` for the game example initializes the graphic display `l_oled`. Please see the upcoming Section 3.4 for more information about graphic displays.
- (7) The `WndProc` for the game example initializes also the segment display `l_scoreBoard`. Please see the upcoming Section 3.5 for more information about segment displays.
- (8) Finally, the `WndProc` for the game example initializes the LED (`l_userLED`) as a segment display. An LED is treated here as a simple 1-segment display with two states LED-off and LED-on.
- (9) After all initialization is done, the `WndProc` creates a Win32 thread to run the embedded application. The thread routine passed to the `CreateThread()` function is just a thin wrapper around the `main_gui()` function. In other words, you call the same `main_gui()` function as the one used in the embedded target and is “unaware” that it executes in Windows. See also the next Section 4.4.

NOTE: The function `main_gui()` is an alias for the `main()` function when the symbol `QWin_GUI` is defined. This alias is created in `bsp.h` as described in the following Section 4.4).

- (10) The `WM_DESTROY` Windows message is sent to the `WndProc` when the main window is closed or the application is forced to quit by the operating system. The `BSP_terminate()` calls `PostQuitMessage()` function as well as `BSP_terminate()`, which causes termination of the application thread.
- (11) The `WM_COMMAND` Windows message is generated by users clicking on regular buttons or activating menus. The game example contains one such button “Quit”.
- (12) Owner-drawn buttons are treated differently than regular buttons such as “Quit”. They produce the `WM_DRAWITEM` Windows messages, which is sent when an owner-drawn button is pressed and again when it is released. In simulating real buttons in embedded front panels this is typically exactly what you want.

NOTE: The Win32 API for owner-drawn controls is a bit complicated, because it covers not just buttons but owner-drawn menus, combo-boxes, and many others. However, for owner-drawn buttons most of the complexities are hidden inside the `OwnerDrawnButton_draw()` function provided in the QWin GUI kit.

- (13) The `DRAWITEMSTRUCT` pointer is extracted from the Windows message parameter.
- (14) The `switch` statement discriminates based on the ID of the child-window control. Here, you decide which owner-drawn button you want to service.
- (15) For example, to service the USER button of the EK-LM3S811 board (see [Error: Reference source not found](#)), you provide a case statement labeled with `IDC_USER`, which is the ID of the owner-drawn button set in the resource editor.

- (16) The function `OwnerDrawnButton_draw()`, which is provided in the QWin GUI kit, performs the “heavy lifting” of painting the button in the current state and returns the status of the button.
- (17) When the button is in the `BTN_DEPRESSED` state, the game example calls `BSP_playerTrigger()`, which publishes the `PLAYER_TRIGGER` event to the active objects and also turns the USER LED on.
- (18) When the button is in the `BTN_RELEASED` state, the game example turns the USER LED off.
- (19) The `WM_MOUSEWHEEL` Windows message is generated by the mouse wheel. This code demonstrates how to handle the mouse-wheel input.
- (20) The `WM_KEYDOWN` Windows message is generated by pressing keys on the keyboard. This code demonstrates how to provide keyboard input to your application.

4.4 The Application Thread

Upon receiving the `WM_INITDIALOG` message, the `WinMain()` function creates the application thread that executes the user's `main_gui()` function, while `WinMain()` keeps executing the Windows event loop. The thread routine `appThread()` is defined as follows:

NOTE: The function `main_gui()` is an alias for the `main()` function when the symbol `QWin_GUI` is defined. This alias is created in `bsp.h` as follows (see also the following Section 4.4):

```
/* special adaptations for QWin GUI applications */
#ifdef QWin_GUI
    /* replace main() with main_gui() as the entry point to a GUI app.*/
    #define main() main_gui()
    int main_gui(); /* prototype of the GUI application entry point */
#endif

static DWORD WINAPI appThread(LPVOID par) {
    (void)par; /* unused parameter */
    return main_gui(); /* run the GUI application */
}
```

5 The main() Function (“Superloop”)

The `main()` function intended to run both on the embedded target and in the Windows emulation, is located in the `main.c` source file (see [Listing 9](#)). As described before, for simplicity this code is structured as a primitive “superloop” with a time delay. The most important design element exemplified by the `main.c` module is that it accesses the hardware only through the BSP interface (`bsp.c` header file) and does not use Win32 API in any way.

NOTE: The avoidance any Win32 API dependencies in the code intended for the target has a very nice side effect of **very fast compilation** of the embedded source code, as the huge `<windows.h>` header file does not need to be processed in that portion of the project.

Listing 9: Example embedded code (file `main.c`).

```
#include <stdint.h>
#include <string.h>    /* for memset(), memmove() */
#include <math.h>
#include "bsp.h"

/* Local-scope objects -----*/
static uint8_t l_frameBuf[BSP_SCREEN_WIDTH * BSP_SCREEN_HEIGHT/8U];
...
/*.....*/
int main() {
    uint32_t n = 0U;

    BSP_init(); /* initialize the Board Support Package */

    /* clear the frame buffer */
    memset(l_frameBuf, 0U, (BSP_SCREEN_WIDTH * BSP_SCREEN_HEIGHT/8U));

    for (;;) { /* for-ever */
        /* generate the new pixel column in the display based on the count */
        uint32_t pixCol = (1U << fun(n));

        /* advance the frame buffer by 1 pixel to the left */
        memmove(l_frameBuf, l_frameBuf + 1U,
                (BSP_SCREEN_WIDTH * BSP_SCREEN_HEIGHT/8U) - 1U);

        /* add the new column to the buffer at the right */
        l_frameBuf[BSP_SCREEN_WIDTH - 1U] = (uint8_t)pixCol;
        l_frameBuf[BSP_SCREEN_WIDTH + BSP_SCREEN_WIDTH - 1U] =
            (uint8_t)(pixCol >> 8);

        if (!BSP_isPaused()) { /* not paused? */
            BSP_drawBitmap(l_frameBuf); /* draw the bitmap on the display */
            BSP_drawCount(n);
            ++n; /* advance the count */
        }

        BSP_sleep(1); /* sleep for 1 clock tick */
    }
    return 0;
}
```

6 Internal QWin GUI Implementation

This section briefly describes the `qwin_gui.h` interface and `qwin_gui.c` implementation.

6.1 The `qwin_gui.h` Header File

The `qwin_gui.h` header file specifies the interface for GUI programming, which includes creating the main custom dialog box for the front panel as well as various useful facilities for owner-drawn buttons, graphic displays, segment displays, and LEDs.

Listing 10: `qwin_gui.h` header file.

```
#define WIN32_LEAN_AND_MEAN
(1) #include <windows.h> /* Win32 API */

/* create the custom dialog hosting the embedded front panel .....*/
(2) HWND CreateCustDialog(HINSTANCE hInst, int idDlg, HWND hParent,
                        WNDPROC lpfnWndProc, LPCTSTR lpWndClass);

/* OwnerDrawnButton "class" .....*/
(3) typedef struct {
    HBITMAP hBitmapUp;
    HBITMAP hBitmapDown;
    HCURSOR hCursor;
} OwnerDrawnButton;

enum OwnerDrawnButtonAction {
    BTN_NOACTION,
    BTN_PAINTED,
    BTN_DEPRESSED,
    BTN_RELEASED
};

void OwnerDrawnButton_init(OwnerDrawnButton * const me,
                          UINT itemID,
                          HBITMAP hBitmapUp, HBITMAP hBitmapDwn,
                          HCURSOR hCursor);
void OwnerDrawnButton_xtor(OwnerDrawnButton * const me);
enum OwnerDrawnButtonAction OwnerDrawnButton_draw(
    OwnerDrawnButton * const me,
    LPDRAWITEMSTRUCT lpdis);

/* GraphicDisplay "class" for drawing graphic displays
 * with up to 24-bit color...
 */
(4) typedef struct {
    HDC      src_hDC;
    int      src_width;
    int      src_height;
    HDC      dst_hDC;
    int      dst_width;
    int      dst_height;
    HWND     hItem;
    HBITMAP  hBitmap;
```

```

        BYTE    *bits;
        BYTE    bgColor[3];
    } GraphicDisplay;

void GraphicDisplay_init(GraphicDisplay * const me,
                        UINT width, UINT height,
                        UINT idemID, BYTE const bgColor[3]);
void GraphicDisplay_xtor(GraphicDisplay * const me);
void GraphicDisplay_clear(GraphicDisplay * const me);

#define GraphicDisplay_setPixel(me_, x_, y_, color_) do { \
    BYTE *pixelRGB = &(me_)->bits[3*((x_) \
        + (me_)->src_width * ((me_)->src_height - 1U - (y_))]); \
    pixelRGB[0] = (color_)[0]; \
    pixelRGB[1] = (color_)[1]; \
    pixelRGB[2] = (color_)[2]; \
} while (0)

#define GraphicDisplay_clearPixel(me_, x_, y_) do { \
    BYTE *pixelRGB = &(me_)->bits[3*((x_) \
        + (me_)->src_width * ((me_)->src_height - 1U - (y_))]); \
    pixelRGB[0] = (me_)->bgColor[0]; \
    pixelRGB[1] = (me_)->bgColor[1]; \
    pixelRGB[2] = (me_)->bgColor[2]; \
} while (0)

void GraphicDisplay_redraw(GraphicDisplay * const me);

/* SegmentDisplay "class" for drawing segment displays, LEDs, etc.....*/
(5) typedef struct {
    HWND    *hSegment;    /* array of segment controls */
    UINT    segmentNum;    /* number of segments */
    HBITMAP *hBitmap;    /* array of bitmap handles */
    UINT    bitmapNum;    /* number of bitmaps */
} SegmentDisplay;

void SegmentDisplay_init(SegmentDisplay * const me,
                        UINT segNum, UINT bitmapNum);
void SegmentDisplay_xtor(SegmentDisplay * const me);
BOOL SegmentDisplay_initSegment(SegmentDisplay * const me,
                                UINT segmentNum, HWND hSegment);
BOOL SegmentDisplay_initBitmap(SegmentDisplay * const me,
                                UINT bitmapNum, HBITMAP hBitmap);
BOOL SegmentDisplay_setSegment(SegmentDisplay * const me,
                                UINT segmentNum, UINT bitmapNum);

/* generic helper functions .....*/
void DrawBitmap(HDC hdc, HBITMAP hBitmap, int xStart, int yStart);

```

- (1) The <windows.h> header file provides the APIs used in the Win32-GUI programming.
- (2) The CreateCustDialog() function creates the custom dialog box for the main application window.
- (3) The OwnerDrawnButton struct and functions form a “class” for owner-drawn buttons.

- (4) The `GraphicDisplay` struct and functions form a “class” for graphic displays (such as LCDs, OLEDs, etc) with up to 24-bit color. The `GraphicDisplay` “class” provides efficient pixel-level interface to a 24-bit “Device Independent Bitmap”.
- (5) The `SegmentDisplay` struct and functions form a “class” for segment displays. The `SegmentDisplay` “class” provides generic solution for managing multiple segment groups and multiple bitmaps corresponding to the specific configurations of the segments.

6.1 The `qwin_gui.c` Source File

The `qwin_gui.c` source file implements the facilities for rendering various GUI elements, such as LCD displays, owner-drawn buttons, and LEDs.

The implementation used in the `qwin_gui.c` source file is intentionally low-level, with raw Win32 API, which does not require any higher-level software layers (such as MFC, .NET/C#, Qt, or wxWidgets). The main advantages of this approach are: simplicity of use and deployment, pure C interface, and unbeatable efficiency. The following [Listing 11](#) shows the implementation of the `CreateCustDialog()` function, which must be called from `WinMain()` in the GUI applications.

Listing 11: `CreateCustDialog()` function (file `qwin_gui.c`).

```
(1) HWND CreateCustDialog(HINSTANCE hInst, int idDlg, HWND hParent,
                          WNDPROC lpfnWndProc, LPCTSTR lpWndClass)
{
    WNDCLASSEX wndclass;
    HWND        hWnd;

    wndclass.cbSize      = sizeof(wndclass);
    wndclass.style        = CS_HREDRAW | CS_VREDRAW;
    (2)  wndclass.lpfnWndProc = lpfnWndProc;
    wndclass.cbClsExtra   = 0;
    wndclass.cbWndExtra   = DLGWINDOWEXTRA;
    wndclass.hInstance    = hInst;
    wndclass.hIcon        = NULL;
    wndclass.hCursor      = LoadCursor(NULL, IDC_ARROW);
    wndclass.hbrBackground = (HBRUSH)COLOR_WINDOW;
    wndclass.lpszMenuName  = NULL;
    (3)  wndclass.lpszClassName = lpWndClass;
    wndclass.hIconSm       = NULL;

    (4)  RegisterClassEx(&wndclass);

    (5)  hWnd = CreateDialog(hInst, MAKEINTRESOURCE(idDlg), hParent, NULL);

    /* NOTE: WM_INITDIALOG provides stimulus for initializing dialog controls.
     * Dialog box procedures typically use this message to initialize controls
     * and carry out any other initialization tasks that affect the appearance
     * of the dialog box.
     */
    (6)  SendMessage(hWnd, WM_INITDIALOG, (LPARAM)0, (LPARAM)0);

    return hWnd;
}
```


- (1) The `CreateCustDialog()` function creates a custom dialog box based on the custom Windows class and must be called from `WinMain()`. The main job of the `CreateCustDialog()` function is registering the custom Window class for your application.
- (2) The window procedure (`WndProc`) of the custom Window class for QP is set from the parameter provided to the `CreateCustDialog()` function.

NOTE: The QP-Win32 port associates the customized dialog box with a regular Window Procedure rather than a standard Dialog Procedure. This is necessary for greater flexibility.

- (3) The name of the custom Window class for your application is set from the parameter provided to the `CreateCustDialog()` function.

NOTE: The dialog box resource must explicitly set the `CLASS` property and the name of this property must be passed to the `CreateCustDialog()` function.

- (4) The custom Window class is registered with Windows.
- (5) The main Dialog window is created from the `idDlg` resource provided by the application as parameter to the `CreateCustDialog()` function.
- (6) The `WM_INITDIALOG` message is sent to the `WndProc` associated with the dialog box. This message indicates the end of initialization and provides a convenient stimulus for starting your application.

The `qwin_gui.c` source file provides also the implementation of all the interesting and hard to find GUI facilities, such as the `GraphicDisplay` “class” for efficient rendering of graphic, pixel-addressable displays, `SegmentDisplay` “class” for segment displays of all kinds as well as LEDs, and `OwnerDrawnButton` “class” for rendering buttons with owner-defined look at both the “depressed” and “released” states and generating separate “depressed” and “released” events.

7 Contact Information

Quantum Leaps, LLC

<https://www.state-machine.com>
info@state-machine.com

+1 919 360-5668 (9-5 EST)

