



QuantumTM Leaps
innovating embedded systems

Application Note
Quantum PlatformTM
and
***emWin* Embedded GUI**
(μ C/GUI)

Revision C
May 2007

Quantum LeapsTM, LLC

www.quantum-leaps.com

Copyright © 2002-2007 **Quantum Leaps, LLC**. All Rights Reserved.

All parts of this document and the referenced software are protected by copyright law and can be used only with a valid license, as described in this document. Any other use or distribution of the referenced software or any part of this document is illegal.

1	Introduction.....	1
1.1	About the QP- <i>emWin</i> TM (μ C/GUI) Integration	2
1.2	Licensing the QP- <i>emWin</i> (μ C/GUI) Examples.....	2
1.3	Licensing <i>emWin</i> TM	2
1.4	Licensing μ C/GUI	2
2	Getting Started	3
2.1	Installing <i>emWin</i> TM (μ C/GUI)	3
2.2	Installing QP	3
2.3	Installing QDK-Windows	3
2.4	Installing QP- <i>emWin</i> (μ C/GUI) Examples	3
2.5	Executing the QP- <i>emWin</i> (μ C/GUI) Examples.....	5
2.5.1	The QP-demo Example without the Windows Manager.....	5
2.5.2	The QP-demo Example with the Windows Manager	6
2.6	Building the QP- <i>emWin</i> (μ C/GUI) Examples	7
2.7	Debugging the QP- <i>emWin</i> (μ C/GUI) Examples	8
3	Using QP-<i>emWin</i> without Window Manager	9
3.1	General Structure of the QP-GUI Application	9
3.2	GUI_Manager Active Object Implementation.....	10
3.3	Initializing <i>emWin</i> (μ C/GUI) Simulation.....	12
3.4	Generating Hardkey Events in <i>emWin</i> (μ C/GUI) Simulation	12
4	Using QP-<i>emWin</i> with the Window Manager	14
4.1	General Structure of the QP-GUI Application	14
4.2	Calling the Callbacks: WM_Exec()	15
4.3	Generating QP Events from WM Callbacks	15
4.4	Drawing from WM Callbacks	16
4.5	Handling Pointer Input Devices (PID)	16
4.5.1	Generating the PID Events in the <i>emWin</i> Simulation	17
5	References	18
6	Contact Information.....	18

emWin

μC/GUI
Embedded Graphical User Interface



1 Introduction

Graphical User Interfaces (GUIs) are becoming increasingly popular in embedded systems. The embedded software's industry response to this trend is proliferation of embedded GUI libraries, such as *emWin*[™] from SEGGER, PEG from Swell Software, and many others. Such GUI libraries provide anything from low-level LCD drivers, through drawing primitives, assortments of widgets, all the way to sophisticated window managers.

One thing, however, that these software packages do NOT provide is the high-level "screen logic" to control the overall behavior of the GUI. As it turns out, Quantum Platform[™] (QP) beautifully complements the GUI libraries by exactly providing the high-level structure to the GUI system. This is, of course, hardly surprising because GUIs are exemplary event-driven systems, which QP is exactly designed to handle.

This Application Note describes how to use QP with the *emWin*[™] Embedded GUI from SEGGER (www.segger.com/emwin.html) and also μC/GUI from Micrium (www.micrium.com/products/-gui/gui.html), which technically are the same products.

To demonstrate the working examples, this Application Note uses the *emWin*[™] Simulation on Windows[®], which is available for a free download from the SEGGERs website (www.segger.com/-downloads.html). You need only a Windows-based PC to execute the examples provided in this Application Note. Additionally, you'd need Microsoft Visual C++ 6.0 or higher to re-build and debug the provided examples.

The following software components have been used in this Application Note:

1. *emWin*[™] trial version 4.04 (black and white) for Microsoft Visual C++ compiler.
2. QP/C or QP/C++ v3.2.05 or higher.
3. QDK/C-Win32 or QDK/C++-Win32.

NOTE: Although the QP-*emWin* (μC/GUI) integration, as described in this document, runs on Windows, the application-level code uses exclusively the embedded *emWin*[™] API and is designed to run without any modifications on embedded targets.

1.1 About the QP-*emWin*™ (μC/GUI) Integration

Perhaps the most tricky part of integrating Quantum Platform with any GUI system is reconciling the event-driven multitasking models used in QP and the GUI system. Since both systems are event-driven, it is crucial to carefully avoid concurrency hazards and potential conflicts of authority (who's controlling CPU, events, event queuing, event processing, and so on).

As described in the SEGGER's manual "*emWin*™ Graphic Library with Graphical User Interface" [SEGGER 05], the *emWin*™ library supports the following three multitasking models:

1. Single-task system (superloop);
2. Multitasking system: only one task calling *emWin*™; and
3. Multitasking system: multiple tasks calling *emWin*™.

In principle QP can work with *emWin*™ in all three modes. However, the option number 2 (multitasking system with only one task calling *emWin*™) is the most recommended and only this one is described in this Application Note. This model corresponds to encapsulating *emWin*™ inside a dedicated active object, which will be called "GUI-Manager". The upcoming Section 3 and 4, describe the design in more detail.

1.2 Licensing the QP-*emWin* (μC/GUI) Examples

The **Generally Available (GA)** distribution of the QP-*emWin* integration available for download from the www.quantum-leaps.com/downloads website is offered with the following two licensing options:

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file GPL.TXT included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).
- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.



1.3 Licensing *emWin*™

The *emWin*™ Embedded GUI is a commercial product of SEGGER Microcontroller Systeme GmbH (www.segger.com) and requires a license to evaluate and use the software. Please contact SEGGER (www.segger-us.com/emWin.html) for more information.



1.4 Licensing μC/GUI

The same Embedded GUI software is also licensed as μC/GUI by Micrium Corporation (www.micrium.com) and requires a license to evaluate and use the software. Please contact Micrium (www.micrium.com/products/gui/gui.html) for more information.



2 Getting Started

This section describes how to install, execute, build, and debug QP-*emWin* applications.

NOTE: This Application Note pertains both to C and C++ versions Quantum Platform™. Most of the code listings in this document refer to the C version. Occasionally the C code is followed by the equivalent C++ implementation to show the C++ differences whenever such differences become important.

2.1 Installing *emWin*™ (μC/GUI)

This Application Note assumes that the trial version of *emWin*™ has been downloaded from SEGGER and installed on a Windows-based PC. The *emWin*™ trial versions are distributed in ZIP archives (e.g., *emwintrial_v404_1bpp.zip*). You can uncompress the file into any directory, but please keep in mind that the QP directory should be at the same level as the *emWin*™ directory (see Listing 1)

2.2 Installing QP

You need to download and install the Generally Available distribution of the QP code (QP/C or QP/C++) from www.quantum-leaps.com/downloads. QP is distributed in a ZIP archive (e.g., *qpc_3.2.05.zip* for the QP/C version, or *qpcpp_3.2.05.zip* for the QP/C++ version). The project files in this Application note assume that you uncompress the QP/C ZIP archive into directory *qpc_3* (and the QP/C++ archive into *qpcpp_3*). The directory *qpc_3* (*qpcpp_3*) must be at the same level as the *emWin*™ directory (see Listing 1).

2.3 Installing QDK-Windows

Because the *emWin*™ Simulation runs on Windows, you need to download and install the Quantum Development Kit (QDK) for Windows so that you can run the Windows-based simulation. The Generally Available distribution of QDK-Windows is available from www.quantum-leaps.com/downloads page and also consists of a ZIP archive (e.g., *qdkc_windows_1.2.05.zip* for QDK/C-Windows, and *qdkcpp_windows_1.2.05.zip* for QDK/C++-Windows). You need to extract the QDK-Windows ZIP file into the same directory into which you have installed the corresponding version of QP (i.e., *qpc_3* for QDK/C and *qpcpp_3* for QDK/C++).

NOTE: The Visual C++ workspaces provided in the QP-*emWin* examples assume that you use the multithreaded QDK-Windows. However, the examples work with the single-threaded QDK-Win32-1T, which you can also use.

2.4 Installing QP-*emWin* (μC/GUI) Examples

Finally, you install the QP-*emWin* examples, by decompressing the ZIP file (e.g., *qpc_emwin_1.0.00.zip* for the QP/C version and *qpcpp_emwin_1.0.00.zip* for the C++ version) into the *<emWin>* directory.

The following Listing 1 shows selected directories and files after installing all the components (*emWin*™, QP, QDK-Windows, and QP-*emWin*).

```

+<emWin> - emWin installation directory
+-Config\ - emWin configuration header files (inactive in Simulation)
+-GUI\
  +-Include\ - emWin platform-independent API include files
  +-Library\
    +-GUI.lib - emWin library precompiled for Simulation on Windows
+-Simulation\
  +-GUI Sim.lib - emWin simulation library (features specific to Simulation)

+-qpc_demo\ - QP/C demo (NO windows manager)
  +-Debug\
    +-qpc_demo.exe - QP/C demo executable (see Figure 1)
    +-qpc_demo.dsw - Visual C++ 6.0 workspace for the QP/C demo project

+-qpc_wm_demo\ - QP/C demo (WITH windows manager)
  +-Debug\
    +-qpc_wm_demo.exe - QP/C demo executable (see Figure 2)
    +-qpc_wm_demo.dsw - Visual C++ 6.0 workspace for the QP/C demo project

+-qpc_3/ - QP/C-root directory for Quantum Platform (QP) v3.x.yy
+-include/ - QP public include files
+-ports/ - QP ports
  +-80x86\
    +-win32\ - Win32 port (QDK-Windows)
      +-vc6\
        +-Debug\
          +-qep.lib - Pre-compiled QEP library
          +-qf.lib - Pre-compiled QF library
        +-qep_port.h - QEP-Win32 port header file
        +-qf_port.h - QF-Win32 port header file
  
```

Listing 1 Selected directories and files after completed installation. Directories and files in bold indicate the elements included in the QP-emWin distribution.

2.5 Executing the QP-emWin (μC/GUI) Examples

The executables for the two examples of the QP-emWin integration are located in <emWin>\-qpc_demo\Debug\qpc_demo.exe and <emWin>\qpc_wm_demo\Debug\qpc_pw_demo.exe (for the C version). These examples show how to use QP and *emWin*™ without the windows manager and with the windows manager, respectively.

2.5.1 The QP-demo Example without the Windows Manager

Figure 1 shows the “Dining Philosopher Problem” application (see “QP Programmer's Manual” [QL 06a]) with a GUI provided by the *emWin*™ integration. In this particular case the *emWin*™ Windows Manager has not been used. The GUI displays the application title along the top of the black-and-white LCD display and the state of every active object in the text block below. Additionally, the interface consists of the “hardkeys” placed on the edges of the LCD.

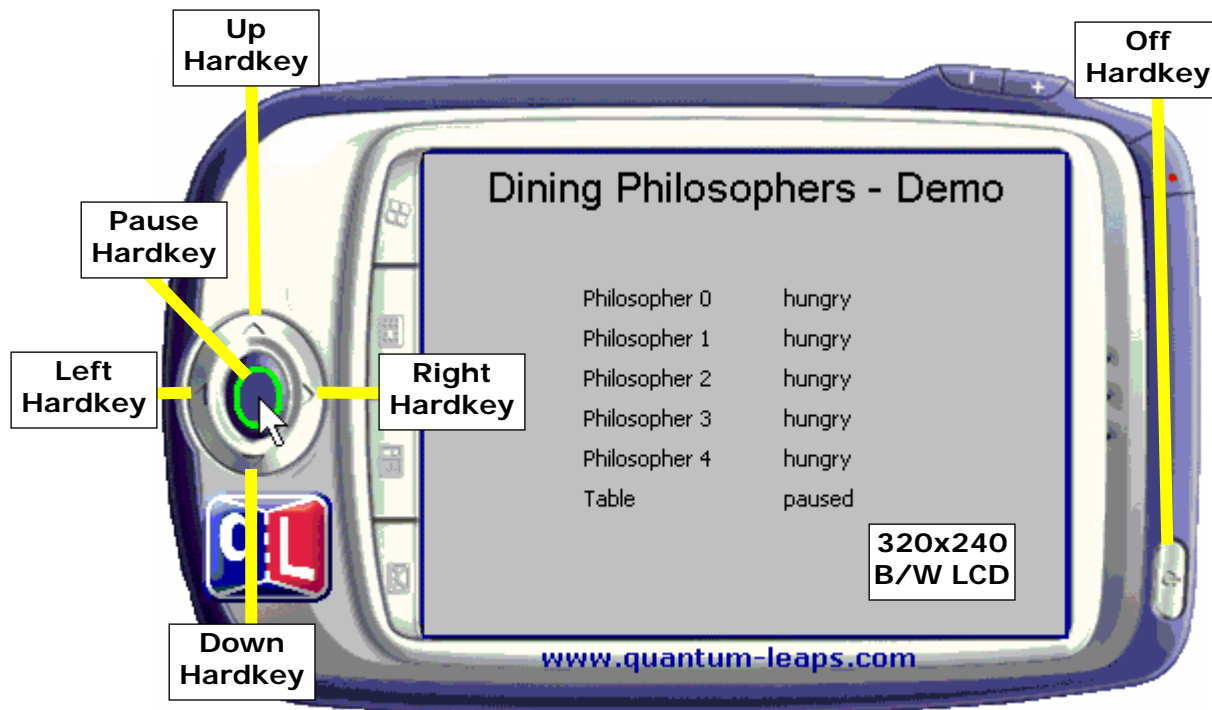


Figure 1 QP-emWin Demo without the Window Manager

You can interact with the application by clicking the mouse on the “hardkeys” indicated in Figure 1. In particular, you can pause the Dining Philosophers by pressing the “Pause” hardkey, As long as the “pause” key remains depressed, the Table active object remains in the “paused” state, in which it does not grant permissions to eat to the Philosophers (therefore quickly all Philosophers go into the “hungry” state). You can also nudge the main text block a few pixels up, down, left, and right by clicking on the “Up”, “Down”, “Left”, and “Right” hardkeys, respectively. Finally, you can terminate the application by clicking on the “Off” key. Alternatively, you terminate the application by right-clicking on the “skin” and selecting the “Exit” pop-up menu.

2.5.2 The QP-demo Example with the Windows Manager

Figure 2 shows the “Dining Philosopher Problem” application (see “[QP Programmer's Manual](#)” [OL 06a]) with the *emWin*™ Windows Manager. The main difference from the previous example is that now the states of the active objects are displayed inside a dialog box with the title “Dining Philosopher Problem”, which is managed by the *emWin*™ Windows Manager.



Figure 2 QP-emWin Demo without the Window Manager

All hardkeys described before are also functional in the Windows Manager example, except now the “Up”, “Down”, “Left”, and “Right” hardkeys move the whole dialog box. Please observe that you can cover and uncover the main application title.

The additional feature demonstrated in this example is a button “Toggle” inside the dialog box that toggles the state of the Table object. Please note how this feature interacts with the “pause” hardkey described before.

2.6 Building the QP-emWin (μC/GUI) Examples

To build and debug the QP-emWin examples on Windows you need Visual Studio v6.0 or higher. The Visual Studio workspaces are included in the QP-emWin examples and are located in <emWin>\qpc_demo\qpc_demo.dsw and <emWin>\qpc_wm_demo\qpc_pw_demo.dsw (for the C version). You build the applications just as any other Visual Studio project (press F7). The workspaces contain the resource file Simulation.res, in which you can modify the bitmaps representing the device and the hardkeys, as described in the SEGGER's manual "*emWin*™ Graphic Library with Graphical User Interface" [SEGGER 05].

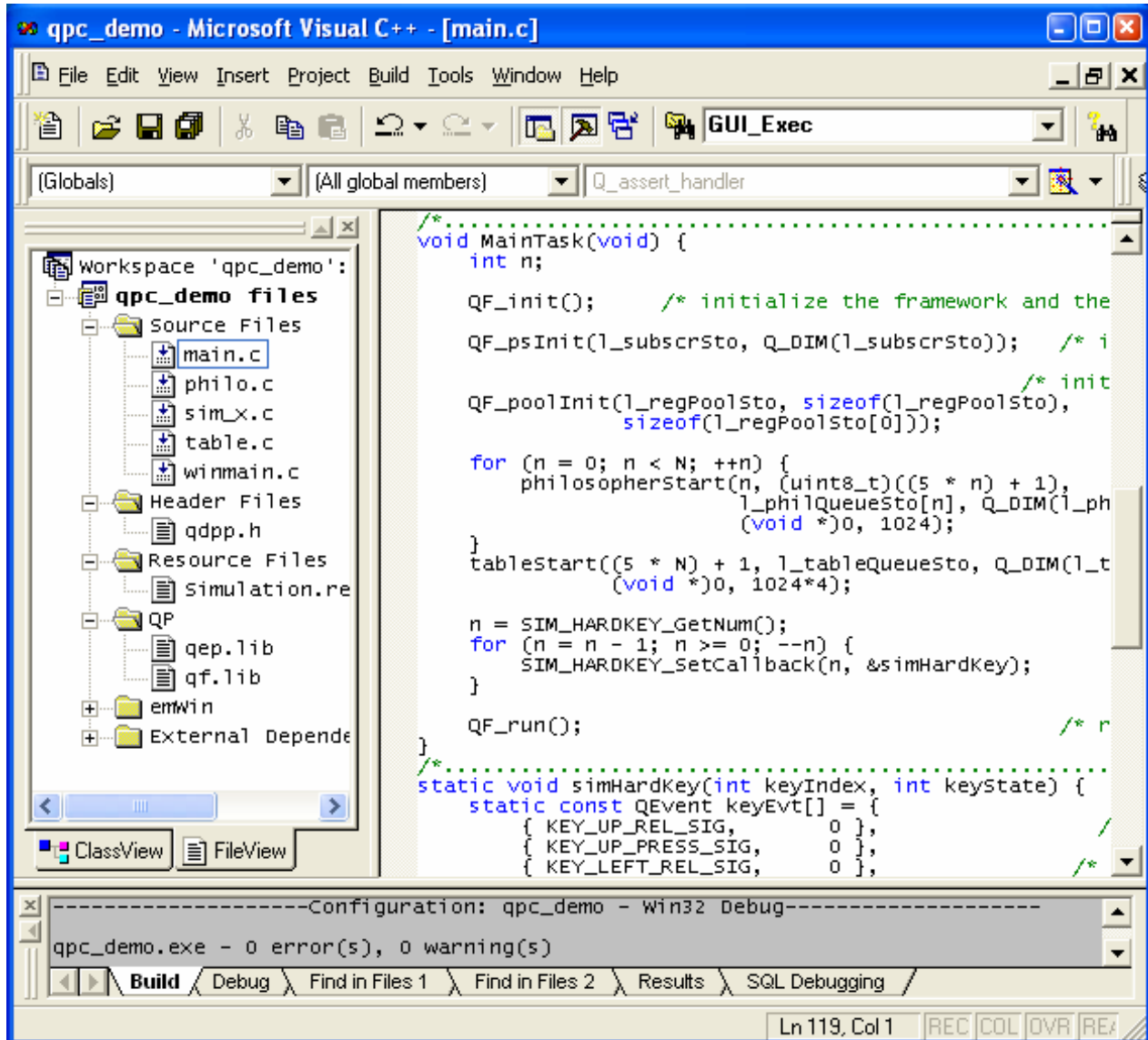


Figure 3 The qpc_demo.dsw workspace loaded in the Visual Studio v6.0.

2.7 Debugging the QP-emWin (µC/GUI) Examples

Debugging QP-emWin applications with the Visual Studio is very straightforward and not really different from other Windows applications. The one big difference is the lag in which the LCD bitmap is updated, which is due to threading model used in the *emWin*™ Simulator. To overcome these shortcomings, SEGGER provides the “Viewer” application (see Chapter 4 in the SEGGER’s manual “*emWin*™ Graphic Library with Graphical User Interface” [SEGGER 05]).

QP applications work just fine with the “Viewer”. You simply launch the “Viewer” (*emWinViewer.exe*) and start debugging the application in the Visual Studio. Figure 4 shows the *emWinViewer* application window on top of the Visual Studio debugging session.

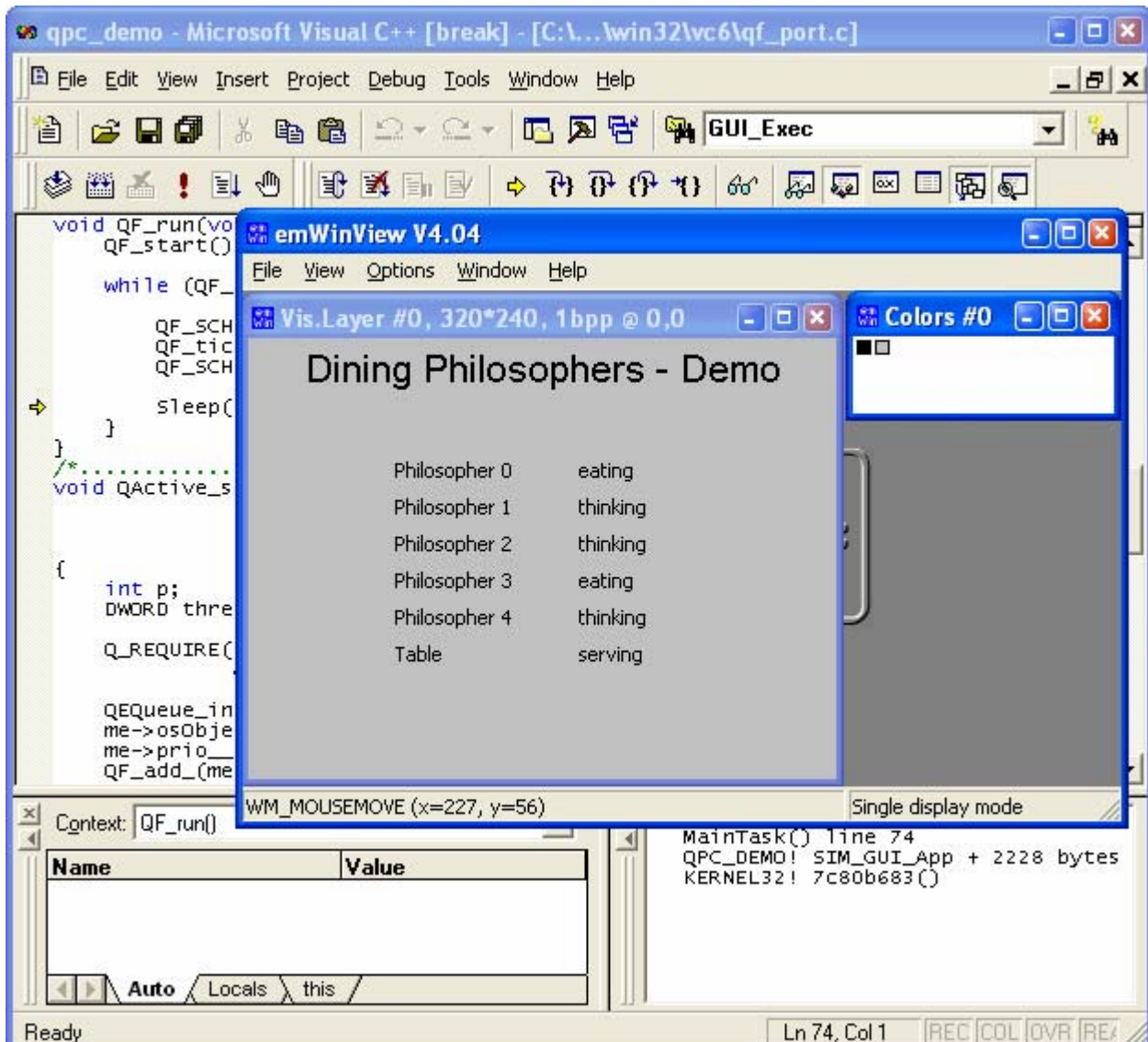


Figure 4 The *emWinViewer* application shows the updates to the LCD bitmap without any lag, which greatly improves the debugging experience.

3 Using QP-emWin without Window Manager

Smaller embedded systems often cannot tolerate the footprint of the *emWin*™ Windows Manager. Without the Windows Manager the application is entirely responsible for performing all updates to the screen. At the same time, however, the control flow is greatly simplified because no *emWin*™ callbacks are used.

3.1 General Structure of the QP-GUI Application

The most recommended architecture of a QP-GUI application is to entirely encapsulate the GUI inside a dedicated active object called the GUI_Manager for the sake of this discussion (this name is generic, of course).

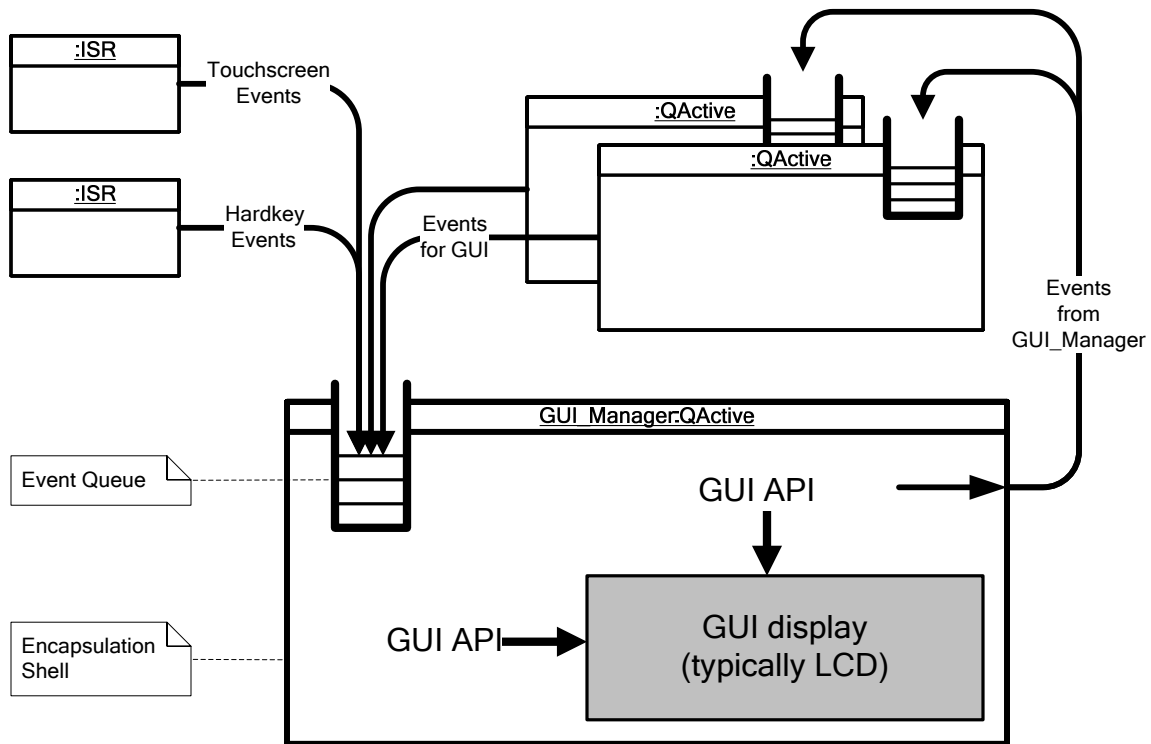


Figure 5 General structure of QP-GUI application without Window Manager. A dedicated Active Object GUI_Manager encapsulates the GUI display and has exclusive control of the GUI display.

As shown in Figure 5, the GUI_Manager active object receives all events asynchronously via its event queue. These events include hardkey events, touchscreen events, or mouse events (generically Pointer Device events). The GUI_Manager entirely owns the GUI display and uses the GUI library (*emWin*™ API in this case) for updating the display. The calls to the GUI library occur as actions of the GUI_Manager’s state machine.

The other components of the system do NOT call the GUI library directly, only post or publish events (using the QP API) to the GUI_Manager active object that performs any necessary GUI updates on their behalf. Please note that without the Windows Manager, the GUI subsystem does not use any “callbacks”, which means that there is no communication from the display to the GUI_Manager.

3.2 GUI_Manager Active Object Implementation

In the Dining Philosopher Problem example the role of the GUI_Manager is played by the Table active object. The state diagram in Figure 6 shows the state machine of the Table active object.

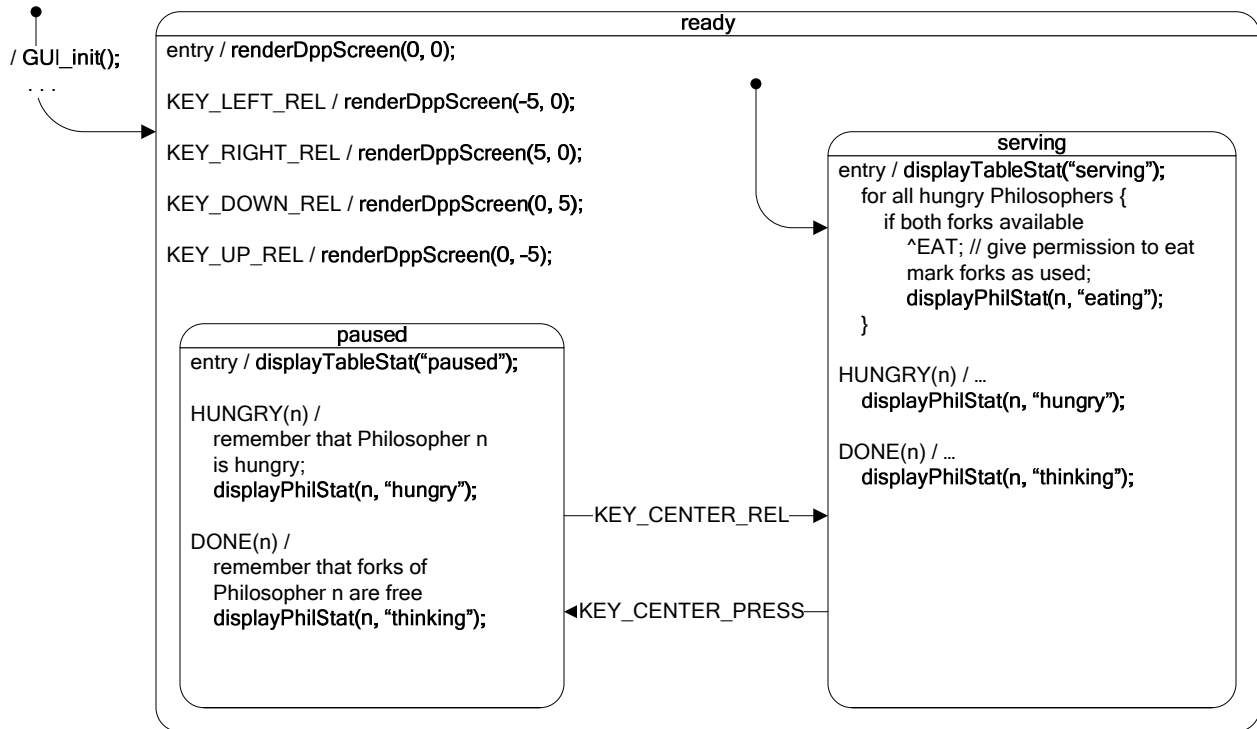


Figure 6 The Table state machine. The highlighted actions perform output to the GUI display (by indirectly calling the *emWin* API).

The GUI library initialization occurs in the top-most initial transition (`GUI_init()`). Subsequently, the helper function `renderDppScreen()` encapsulates the rendering of the screen. This, of course, accomplished with the *emWin*™ API, as shown in the following code fragment:

```

/* ..... */
static void displayPhilStat(uint8_t n, char const *state) {
    l_philoState[n] = state;
    GUI_DisplayStringAt(state, l_x0rg + STATE_X, l_y0rg + l_philoY[n]);
}
/* ..... */
static void displayTableStat(char const *state) {
    l_tableState = state;
    GUI_DisplayStringAt(state, l_x0rg + STATE_X, l_y0rg + l_tableY);
}
/* ..... */
static void renderDppScreen(int dx, int dy) {
    GUI_SetBackColor(GUI_GRAY);
    GUI_Clear();
    GUI_SetColor(GUI_BLACK);
    GUI_SetFont(&GUI_Font24_ASCII);
    GUI_DisplayStringHCenterAt("Dining Philosophers - Demo", 160, 5);
    GUI_SetFont(&GUI_Font13_ASCII);
}
  
```

```

l_xOrg += dx;
l_yOrg += dy;

GUI_DispStringAt("Philosopher 0", l_xOrg, l_yOrg + l_phiYoY[0]);
GUI_DispStringAt("Philosopher 1", l_xOrg, l_yOrg + l_phiYoY[1]);
GUI_DispStringAt("Philosopher 2", l_xOrg, l_yOrg + l_phiYoY[2]);
GUI_DispStringAt("Philosopher 3", l_xOrg, l_yOrg + l_phiYoY[3]);
GUI_DispStringAt("Philosopher 4", l_xOrg, l_yOrg + l_phiYoY[4]);
GUI_DispStringAt("Table", l_xOrg, l_yOrg + l_tableY);

di spl yPhi l Stat(0, l_phi l oState[0]);
di spl yPhi l Stat(1, l_phi l oState[1]);
di spl yPhi l Stat(2, l_phi l oState[2]);
di spl yPhi l Stat(3, l_phi l oState[3]);
di spl yPhi l Stat(4, l_phi l oState[4]);
di spl yTabl eStat(l _tabl eState);
}

```

Listing 2 Helper functions for performing screen output by means of *emWin* API (shown in boldface).

The superstate “ready” handles the “Left”, “Right”, “Up”, and “Down” hardkeys.

NOTE: Each hardkey has two events associated with it: an event generated when a hardkey is depressed (e.g., KEY_LEFT_PRESS) and when it is released (KEY_LEFT_REL). As described in the previous Section, all GUI-related events (such as the hardkeys, touchscreen, mouse, etc.) are generated externally to the GUI_Manager.

The following listing shows the state handler function for the “ready” state:

```

QSTATE Table_ready(Table *me, QEvent const *e) {
    swi tch (e->sig) {
        case Q_ENTRY_SIG: {
            renderDppScreen(0, 0);
            return (QSTATE)0;
        }
        case Q_INIT_SIG: {
            Q_INIT(&Tabl e_servi ng);
            return (QSTATE)0;
        }
        case KEY_LEFT_REL_SIG: { /* hardkey LEFT rel eased */
            renderDppScreen(-5, 0);
            return (QSTATE)0;
        }
        case KEY_RIGHT_REL_SIG: { /* hardkey RIGHT rel eased */
            renderDppScreen(5, 0);
            return (QSTATE)0;
        }
        case KEY_DOWN_REL_SIG: { /* hardkey DOWN rel eased */
            renderDppScreen(0, 5);
            return (QSTATE)0;
        }
        case KEY_UP_REL_SIG: { /* hardkey UP rel eased */
            renderDppScreen(0, -5);
            return (QSTATE)0;
        }
    }
    return (QSTATE)&QHsm_top;
}

```

Listing 3 Superstate “ready” handles the “Left”, “Right”, “Up”, and “Down” hardkeys.

3.3 Initializing *emWin* (μC/GUI) Simulation

In the embedded target, you confine all GUI library calls to the GUI_Manager (Table in this case) active object. In particular, the GUI library is initialized in the top-most initial transition of the Table's state machine:

```
void Table_initial (Table *me, QEvent const *e) {
    uint8_t n;
    (void)e;                               /* suppress the compiler warning */

    GUI_Init();                             /* initialize the emWin embedded GUI */

    QActive_subscribe_((QActive *)me, HUNGRY_SIG);
    QActive_subscribe_((QActive *)me, DONE_SIG);

    for (n = 0; n < N; ++n) {
        me->fork__[n] = FREE;
        me->isHungry__[n] = 0;
    }
    Q_INIT(&Table_ready);
}
```

The *emWin*™ Simulation requires additional initializations in order to generate hardkey events, simulate touchscreen events, or mouse events. The next section describes initializations required for generating hardkey events. The upcoming Section ?? shows how to handle touchscreen and mouse events, which are used in the QDPP example with *emWin*™ Windows Manager.

3.4 Generating Hardkey Events in *emWin* (μC/GUI) Simulation

In the embedded target the hardkey events are generated by debouncing the raw switches attached to buttons on the user panel. This typically is done from an interrupt context, in which you can directly call QP API, such as QF_publish() or QActive_postFIFO() to post the events to the GUI_Manager active object.

The SEGGER's *emWin*™ Simulation environment provides an easy way of generating hardkey events (see Section 3.2.2 in the SEGGER's Manual [SEGGER 05]). The following code snippet from the main.c module shows how to use the callbacks provided in the *emWin*™ Simulation to generate the QP events:

```
static void simHardKey(int keyIndex, int keyState); /* callback function */
/*.....*/
void MainTask(void) {
    int n;

    QF_init(); /* initialize the framework and the underlying RT kernel */
    QF_psInit(I_subscrSto, Q_DIM(I_subscrSto)); /* init publish-subscribe */
    /*.....*/
    QF_poolInit(I_regPoolSto, sizeof(I_regPoolSto),
               sizeof(I_regPoolSto[0])); /* initialize event pools... */

    for (n = 0; n < N; ++n) {
        philosopherStart(n, (uint8_t)((5 * n) + 1),
                        I_phiIQueueSto[n], Q_DIM(I_phiIQueueSto[n]),
                        (void *)0, 1024);
    }
    tableStart((5 * N) + 1, I_tableQueueSto, Q_DIM(I_tableQueueSto),
```

```

        (void *)0, 1024*4);

    n = SIM_HARDKEY_GetNum(); /* get the number of hardkeys in the bitmap */
    for (n = n - 1; n >= 0; --n) {
        SIM_HARDKEY_SetCallback(n, &simHardKey); /* register the callback */
    }

    QF_run(); /* run the QF application */
}
/*.....*/
static void simHardKey(int keyIndex, int keyState) {
    static const QEvent keyEvt[] = {
        { KEY_UP_REL_SIG, 0 }, /* hardkey UP released */
        { KEY_UP_PRESS_SIG, 0 }, /* hardkey UP pressed */
        { KEY_LEFT_REL_SIG, 0 }, /* hardkey LEFT released */
        { KEY_LEFT_PRESS_SIG, 0 }, /* hardkey LEFT pressed */
        { KEY_CENTER_REL_SIG, 0 }, /* hardkey CENTER released */
        { KEY_CENTER_PRESS_SIG, 0 }, /* hardkey CENTER pressed */
        { KEY_RIGHT_REL_SIG, 0 }, /* hardkey RIGHT released */
        { KEY_RIGHT_PRESS_SIG, 0 }, /* hardkey RIGHT pressed */
        { KEY_DOWN_REL_SIG, 0 }, /* hardkey DOWN released */
        { KEY_DOWN_PRESS_SIG, 0 }, /* hardkey DOWN pressed */
        { KEY_POWER_REL_SIG, 0 }, /* hardkey POWER released */
        { KEY_POWER_PRESS_SIG, 0 }, /* hardkey POWER pressed */
    };
    /* do not overrun the array */
    Q_REQUIRE((keyIndex * 2) + keyState < Q_DIM(keyEvt));

    /* post the hardkey event to the Table active object (GUI manager) */
    QActive_postFI FO(QDPP_table, &keyEvt[(keyIndex * 2) + keyState]);

    if ((keyIndex == 5) && (keyState == 0)) { /* hardkey POWER released? */
        PostQuitMessage(0); /* terminate the simulation */
    }
}

```

Listing 4 Initializing the QF application Generating hardkey events from the emWin Simulation

4 Using QP-emWin with the Window Manager

The *emWin*™ Window Manager supplies a set of routines which allow you to easily create, move, resize, and otherwise manipulate any number of windows on the screen. It also provides lower-level support by managing the layering of windows on the display and by alerting your application to display changes that affect its windows. To do its job, the Window Manager must communicate events from the GUI library back to the application, for example, the Window Manager communicates that a window has been resized or a button has been clicked with the mouse.

In the simple situation without the Windows Manager (the previous section), the GUI library was only used for updating the GUI display. The information was flowing from the application to the GUI library, but never back. The fundamental difference introduced by the Window Manager is that GUI library becomes an additional source of events, which must be converted to the QP events to be processed by the state machines. The QDPP example located in <emWin>\qpc_wm_demo\ demonstrates the use of QP with the *emWin*™ Window Manager.

4.1 General Structure of the QP-GUI Application

The general structure of the QP-GUI application with the Window Manager remains the same as in the case without the Window Manager. That is, a dedicated active object called generically the *GUI_Manager* encapsulates the GUI API. The only difference is that now the GUI library is also a source of events, which are posted to the event queue of the *GUI_Manager* active object:

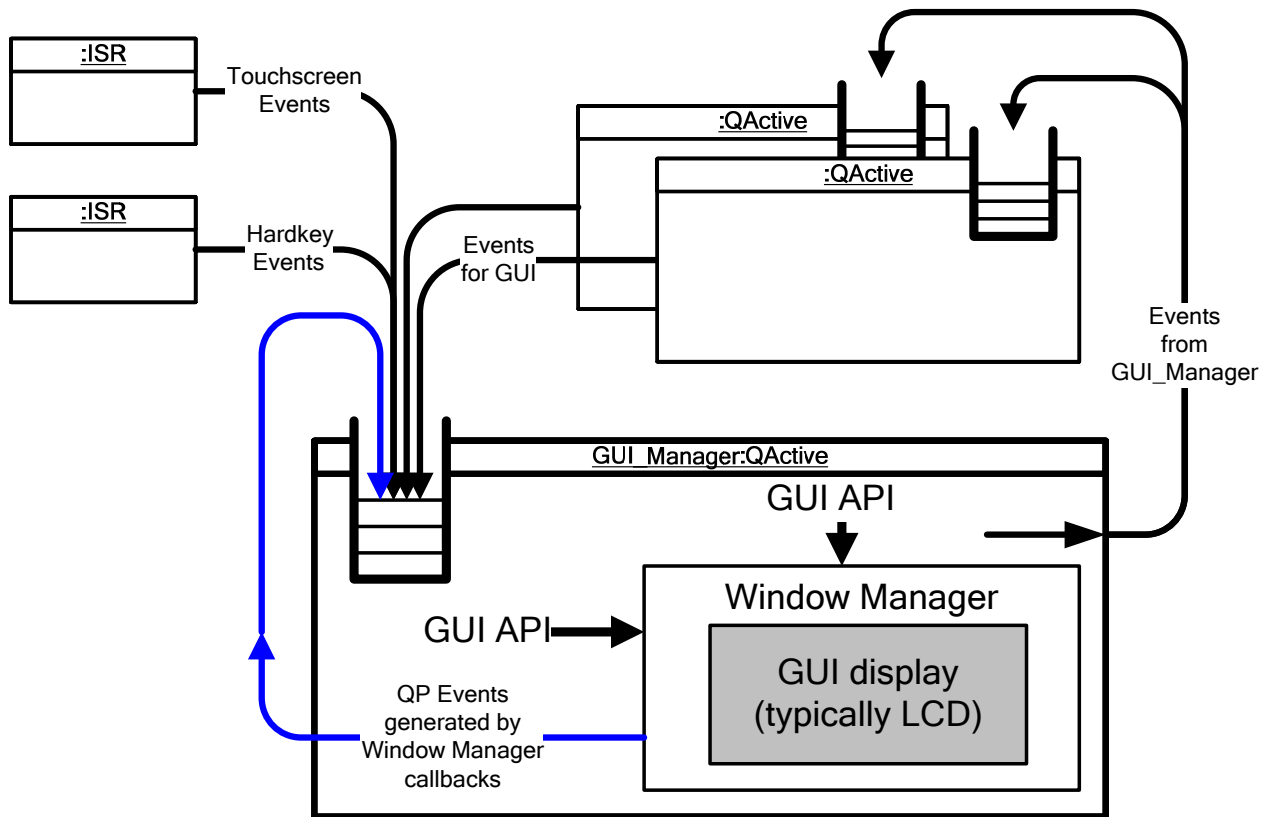


Figure 7 General structure of QP-GUI application with a Window Manager. Callbacks from the GUI are converted to QP events and posted to the event queue of the *GUI_Manager* active object.

Figure 7 shows the modified structure of the application. The GUI_Manager active object receives all events asynchronously through its event queue. These events include such GUI-related events like: hardkey events, touchscreen events, or mouse events (generically Pointer Input Device events). Additionally, the Window Manager invokes callbacks defined by the application. These callbacks are used mostly to generate QP events and post them to the event queue of the GUI_Manager active object.

4.2 Calling the Callbacks: WM_Exec()

In the presence of the Window Manager the screen is not immediately updated after you modify a window or a widget (see Section 15.1.2 “Understanding the redrawing mechanism” in the SEGGER’s *emWin*™ Manual [SEGGGER 05]). The Window Manager merely updates just the window or the widget object, but not necessarily the screen. The screen updates and calling the callbacks is performed only when you invoke the special Window Manager function WM_Exec().

In order to avoid any concurrency hazards, you should invoke WM_Exec() (as all other GUI library functions) exclusively from the GUI_Manager active object. In the QDPP with Window Manager example, the Table active object demonstrates where you should place calls to WM_Exec() to achieve timely updates of the screen.

You should also understand that the callback routines registered with the Window Manager are invoked indirectly from WM_Exec(). By performing all the calls to WM_Exec() from a single active object, which always executes in the Run-To-Completion (RTC) fashion, you avoid by design any concurrency issues.

4.3 Generating QP Events from WM Callbacks

Generally, the WM callbacks should mainly be used for generating QP events and posting them to the GUI_Manager active object (see the blue arrow in Figure 7). The following code snippet from the table.c module illustrates how to do it:

```
static void onDialogGUI(WM_MESSAGE * pMsg) {
    switch (pMsg->MsgId) {
        case WM_INIT_DIALOG: {
            break;
        }
        case WM_NOTIFY_PARENT: {
            switch (pMsg->Data.v) {
                case WM_NOTIFICATION_RELEASED: /* react only if released */
                    switch (WM_GetId(pMsg->hWinSrc)) {
                        case GUI_ID_BUTTON0: {
                            /* static PAUSE event for the Table active object */
                            static QEvent const pauseEvt = { PAUSE_SIG, 0 };
                            QActive_postFI FO(QDPP_table, &pauseEvt);
                            break;
                        }
                    }
                    break;
            }
            break;
        }
        default: {
            WM_DefaultProc(pMsg);
            break;
        }
    }
}
```

Listing 5 Using a WM callback to post events to the Table active object.

4.4 Drawing from WM Callbacks

Occasionally, it is appropriate to perform actual drawing from the WM callback, if such drawing is independent on the state of the GUI_Manager active object. (Otherwise the callback should generate a QP event and post it to the GUI_Manager object, and the drawing should be performed from the GUI_Manager's state machine).

For example, repainting of the main window background does not depend on the state of the Table active object, so it is performed directly from the callback:

```
static void onMainWndGUI (WM_MESSAGE* pMsg) {
    switch (pMsg->MsgId) {
        case WM_PAINT: {
            GUI_SetBackColor(GUI_GRAY);
            GUI_Clear();
            GUI_SetColor(GUI_BLACK);
            GUI_SetFont(&GUI_Font24_ASCII);
            GUI_DisplayStringHCeaterAt("Dining Philosophers - Demo", 160, 5);
            break;
        }
        default: {
            WM_DefaultProc(pMsg);
            break;
        }
    }
}
```

Listing 6 Painting directly from a callback.

4.5 Handling Pointer Input Devices (PID)

One of the main advantages of using the Window Manager is automatic handling of the mouse events or touch-screen events (called generically Pointer Input Devices).

In the embedded target, the PID events originate (as all other external events) from the interrupts or sometimes from polling external devices. In the QP environment, it is recommended to deliver such events as QP events to the event queue of the GUI_Manager active object and only at this level use the GUI API to update the state of the PID inside the GUI library. This strategy is illustrated in the handling of the MOUSE_CHANGE event in the "ready" state of Table active object:

```
QSTATE Table_ready(Table *me, QEvent const *e) {
    switch (e->sig) {
        case MOUSE_CHANGE_SIG: { /* mouse change (move or click) event */
            GUI_PID_STATE mouse;
            mouse.x = ((MouseEvent const *)e)->xPos;
            mouse.y = ((MouseEvent const *)e)->yPos;
            mouse.Pressed = ((MouseEvent const *)e)->buttonStates;

            GUI_PID_StoreState(&mouse); /* update the state of the Mouse PID */
            WM_Exec(); /* update the screen and invoke WM callbacks */
            return (QSTATE)0;
        }
        . . .
    }
    return (QSTATE)&QHsm_top;
}
```

Listing 7 Handling the Pointer Input Device (mouse) in the Table active object.

4.5.1 Generating the PID Events in the *emWin* Simulation

In the *emWin* Simulation, you generate the QP event `MOUSE_CHANGE` from the `GUI_MOUSE_StoreState()` function. You generate the QP event in the standard way, fill in the (x, y) position of the mouse as well as the bitmask representing the pressed-state of the mouse buttons, and finally you post the event to the `Table` active object (`GUI_Manager`).

```
void GUI_MOUSE_StoreState(const GUI_PID_STATE *pState) {  
    MouseEvt *pe = Q_NEW(MouseEvt, MOUSE_CHANGE_SIG);  
    pe->xPos = pState->x;  
    pe->yPos = pState->y;  
    pe->buttonStates = pState->Pressed;  
    QActive_postFI FO(QDPP_table, pe);  
}
```

5 References

Document	Location
[Samek 02] "Practical Statecharts in C/C++", Miro Samek, CMP Books, 2002	Available from most online book retailers, such as amazon.com . See also: http://www.quantum-leaps.com/writings/book.htm
[QPC 06] "QP Programmer's Manual", Quantum Leaps, LLC, 2006	http://www.quantum-leaps.com/doc/QP_Manual.pdf
[SEgger 05] " <i>emWin</i> ™ Graphic Library with Graphical User Interface", SEgger 2005	Document available with the purchase of the <i>emWin</i> ™ software from SEgger.
[Micrium 06] µC/GUI product website.	www.micrium.com/products/gui/gui.html .

6 Contact Information

SEgger Microcontroller Systeme GmbH

Heinrich-Hertz-Str. 5
 40721 Hilden
 Germany

+49(0)2103-2878-0
 +49(0)2103-2878-28 (FAX)

email: info@segger.com
 WEB: www.segger.com

Micrium, Inc.

949 Crestview Circle
 Weston, FL 33327
 USA

+1 954 217 2036
 +1 954 217 2037 (FAX)

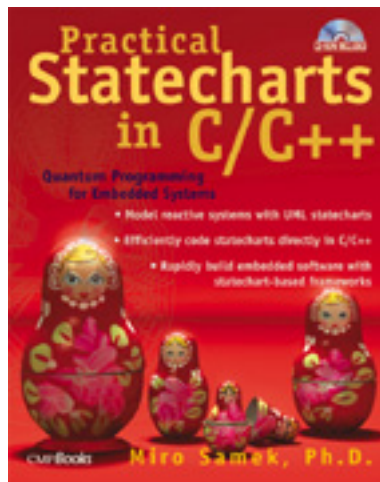
e-mail: info@micrium.com
 WEB: www.micrium.com

Quantum Leaps, LLC

103 Cobble Ridge Drive
 Chapel Hill, NC 27516
 USA

+1 866-450-LEAP (toll free)
 +1 919-869-2998 (fax)

e-mail: info@quantum-leaps.com
 WEB : <http://www.quantum-leaps.com>



"Practical Statecharts in C/C++",
 by Miro Samek, Ph.D.
 CMP Books 2002,
 ISBN 1578201101