Quantum™ LeaPs
innovating embedded systems

QDK™
PIC18 with MPLAB C18

Document Revision D
August 2008

Table of Contents

1 Introduction

This **QP Development Kit™** (QDK) describes how to use QP state machine frameworks with the Microchip® PIC18 processors and the MPLAB C18 compiler. The actual hardware/software used to test this QDK is described below (see also Figure 1):

1. Microchip demo board PICDEM 2 PLUS with PIC18F452 target device inserted into the 40-pin socket of the board.
2. Microchip MPLAB ICD 2 In-Circuit Debugger.
3. MPLAB® C18, PICmicro® 18Cxx C Compiler v3.00 Student Edition.
4. QP/C **4.0.01** or higher.

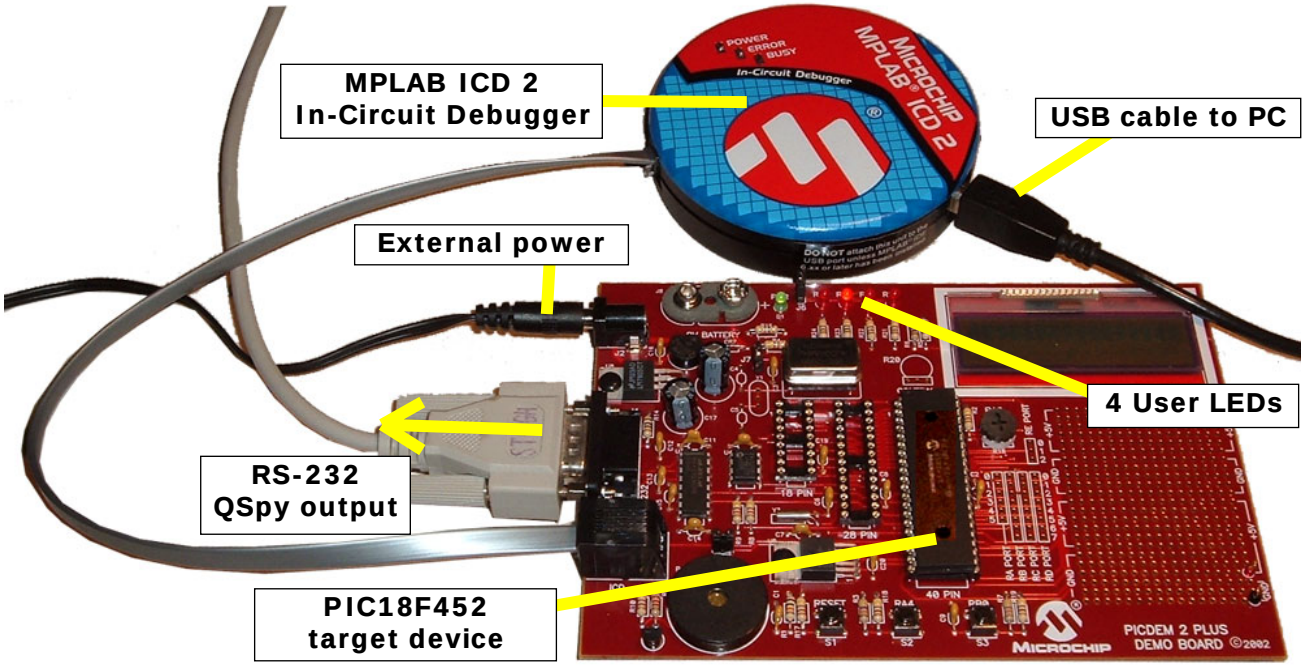


Figure 1 Microchip demo board PICDEM 2 PLUS.

As shown in Figure 1, the Microchip demo board PICDEM 2 PLUS connects to the MPLAB ICD 2 In-Circuit debugger that connects directly to the host development workstation via a standard USB cable. The board can be used for other 40-pin, 28-pin, and 18-pin devices. This QDK has been tested with the PIC18F452 device with 1.5 KB of RAM and 32KB of onboard flash. However, the described port should be applicable to all PIC18 devices big enough to accommodate QP, that is, with RAM > 0.5KB, and ROM > 16KB.

NOTE: The code density of PICmicro devices is low compared to other MCUs, and consequently QP requires more code space than usual. For example, the QP non-preemptive configuration compiled for PIC18F452 with the MPLAB C18 compiler takes more than 11KB of code space. The same exact code compiled for MSP430 with the IAR compiler takes only 2.3KB of code space.

1.1 What's Included in the QDK-PIC18-MPLAB-C18?

This QDK-nano provides the QP port to PICmicro with the MPLAB-C18 toolset, the Board Support Package (BSP) and two versions of the Dining Philosopher Problem (DPP) example application described in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08].

NOTE: Currently, the QDK-PIC18-MPLAB-C18 contains only the non-preemptive QP port. The fully preemptive configuration with QK is not provided in this version, but might be added in the future.

1.2 Licensing QDK-PIC18-MPLAB-C18

The **Generally Available (GA)** distribution of QDK-PIC18-MPLAB-C18 available for download from the www.state-machine.com/pic18 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.

For more information, please visit the licensing section of our website at: www.state-machine.com/licensing.

2 Getting Started with QDK-PIC18-MPLAB-C18

This section describes how to install, build, and use QDK-PIC18-MPLAB-C18 based on the DPP example. This information is intentionally included early in this document, so that you could start using the QDK as soon as possible. The main focus of this section is to walk you quickly through the main points without slowing you down with full-blown detail.

NOTE: This QDK assumes that the standard QP distribution consisting of QEP, QF, QK, and QS has been installed, before installing this QDK. It is also strongly recommended that you read the QP Tutorial (www.quantum-leaps.com/doxygen/qpc/tutorial_page.html) before you start experimenting with this QDK.

2.1 Installation

The QDK code is distributed in a ZIP archive (qdkc_pic18-mplab-c18_<ver>.zip, where <ver> stands for a specific QDK-PIC18-MPLAB-C18 version, such as 4.0.01). You need to uncompress the archive into the same directory into which you've installed all the standard QP components. The installation directory you choose will be referred henceforth as QP Root Directory <qp>. The following Listing 1 shows the directory structure and selected files included in the QDK-PIC18-MPLAB-C18 distribution. (Please note that the QP directory structure is described in detail in a separate Application Note: "QP Directory Structure"):

```
<qp>/                       - QP-root directory for Quantum Platform (QP)
  |
  +-include/                - QP public include files
  | +-qassert.h             - Quantum Assertions platform-independent public include
  | +-qep.h                 - QEP platform-independent public include
  | +-qf.h                  - QF platform-independent public include
  | +-qk.h                  - QK platform-independent public include
  | +-qequeue.h             - native QF event queue include
  | +-qmpool.h              - native QF memory pool include
  | +-qpset.h               - native QF priority set include
  |
  +-ports/                  - QP ports
  | +-pic18/                - PIC18 port
  | | +-vanilla/            - "vanilla" ports (non-preemptive scheduler)
  | | | +-mplab-c18/        - MPLAB C18 compiler
  | | | | +-dbg/            - Debug build directory
  | | | | | +-qep.lib       - QEP library (debug build)
  | | | | | +-qf.lib        - QF library (debug build)
  | | | | +-rel/            - Release build directory
  | | | | | +-qep.lib       - QEP library (release build)
  | | | | | +-qf.lib        - QF library (release build)
  | | | | +-spy/            - Spy build directory
  | | | | | +-qep.lib       - QEP library (instrumented Qspy build)
  | | | | | +-qf.lib        - QF library (instrumented Qspy build)
  | | | | | +-qs.lib        - QS library (instrumented Qspy build)
  | | | | |
  | | | | +-make.bat        - Batch file to build the QP libraries
  | | | | +-qep_port.h      - QEP port header file
  | | | | +-qf_port.h       - QF port header file
  | | | | +-qs_port.h       - QS port header file
  | | | | +-qp_port.h       - QP port header file
  |
  +-examples/               - subdirectory containing the QP example files
  | +-pic18/                - PIC18 port
  | | +-vanilla/            - "vanilla" ports (non-preemptive scheduler)
```

```
|  |  | +-mplab-c18/       - MPLAB C18 compiler
|  |  |  +-dpp-picdem2plus/ - Dining Philosophers example for the PICDEM 2 PLUS board
|  |  |  |  +-dbg/         - directory containing the Debug build
|  |  |  |  +-rel/         - directory containing the Release build
|  |  |  |  +-spy/         - directory containing the Spy build
|  |  |  |  |
|  |  |  |  +-bsp.c        - Board Support Package for PICDEM 2 PLUS
|  |  |  |  +-bsp.h        - BSP header file
|  |  |  |  +-isr.c        - Interrupt Service Routines (ISRs) for the application
|  |  |  |  +-main.c       - the main function
|  |  |  |  +-philo.c      - the Philosopher active object
|  |  |  |  +-dpp.h        - the DPP header file
|  |  |  |  +-table.c      - the Table active object
|  |  |  |  +-18f452.lkr   - the linker script for PIC18F452 devices (release)
|  |  |  |  +-18f452i.lkr  - the linker script for PIC18F452 devices (debugging)
|  |  |  |  +-dpp-dbg.mcp  - the MPLAB project for debug build
|  |  |  |  +-dpp-rel.mcp  - the MPLAB project for release build
|  |  |  |  +-dpp-spy.mcp  - the MPLAB project for spy build
|  |  |  |  +-dpp-picdem2plus.mcw  - the MPLAB workspace containing all the projects
```

Listing 1 Selected QP directories and files after installing QDK-PIC18-MPLAB-C18. The highlighted elements are included in the QDK-PIC18-MPLAB-C18 distribution.

2.2 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for QEP, QF, and QS are provided inside the <qp>\ports\ directory (see Listing 1). This section describes steps you need to take to rebuild the libraries yourself.

NOTE: To achieve commonality among different development tools, Quantum Leaps software does not use the vendor-specific IDEs, such as the MPLAB IDE, for building the QP libraries. Instead, QP supports *command-line* build process based on simple batch scripts.

The code distribution contains all the batch file make.bat for building all the libraries located in <qp>**ports**\pic18\vanilla\mplab-c18\ directory. For example, to build the debug version of all the QP libraries for the PIC18, with the MPLAB C18 compiler, you open a console window on a Windows PC, change directory to <qp>**ports**\pic18\vanilla\mplab-c18\, and invoke the batch by typing at the command prompt the following command:

```
make
```

The make process should produce the QP libraries in the location: <qp>\ports\pic18\vanilla\mplab-c18**dbg**. The make.bat assumes that the MPLAB C18 toolset has been installed in the directory C:\tools\Microchip\MCC18.

NOTE: You need to adjust the symbol **MPLAB_C18** at the top of the make.bat file if you've installed the MPLAB C18 compiler into a different directory. You might also need to adjust the symbol **CCFLAGS** to use a different compiler options.

In order to take advantage of the QSpy instrumentation, you need to build the QSpy version of the QP libraries. You achieve this by invoking the make.bat utility with the "spy" target, like this:

```
make spy
```

The make process should produce the QP libraries in the directory: <qp>\ports\pic18\vanilla\-mplab-c18**spy**.

You choose the build configuration by providing a target to the make.bat utility. The default target is "dbg". Other targets are "rel", and "spy" respectively. The following table summarizes the targets accepted by make.bat.

Software Version	Build command
Debug (default)	make
Release	make rel
Spy	make spy

Table 1 Make targets for the Debug, Release, and Spy library versions

2.3 Building the Example

This QDK-PIC18-MPLAB-C18 includes the DPP ("Dining Philosophers") example described in the Application Note "Dining Philosophers Problem Example" [QP AN-DPP 08]. The QDK includes a MPLAB workspace to build the three configurations of the example (Debug, Release, and Spy). The workspace is located in <qp>\examples\pic18\vanilla\mplab-c18\dpp-picdem2plus\dpp-picdem2plus.mcw for the "vanilla" version.

2.3.1 Loading the Project into MPLAB IDE

1. Connect the PICDEM 2 PLUS board to the MPLAB ICD 2 In-Circuit debugger and to the PC as described in the Quick Start Guide. The jumper configuration for this application is:
    a. J6 jumper installed (external power)
    b. J7 jumper removed (open)
    c. J9 jumpers installed (closed)
2. Connect the special cable between the PICDEM2 PLUS board and MPLAB ICD 2;
3. Connect the USB cable between the MPLAB ICD 2 In-Circuit Debugger and the PC.
4. Open the MPLAB IDE and open the project dpp-dbg.mcp (located in <qp>\examples\pic18\-vanilla\mplab-c18\dpp-picdem2plus\). Figure 2 shows the screen shot of the MPLAB IDE after opening the project.
5. Build the project by select Project->Make menu or by pressing F10.

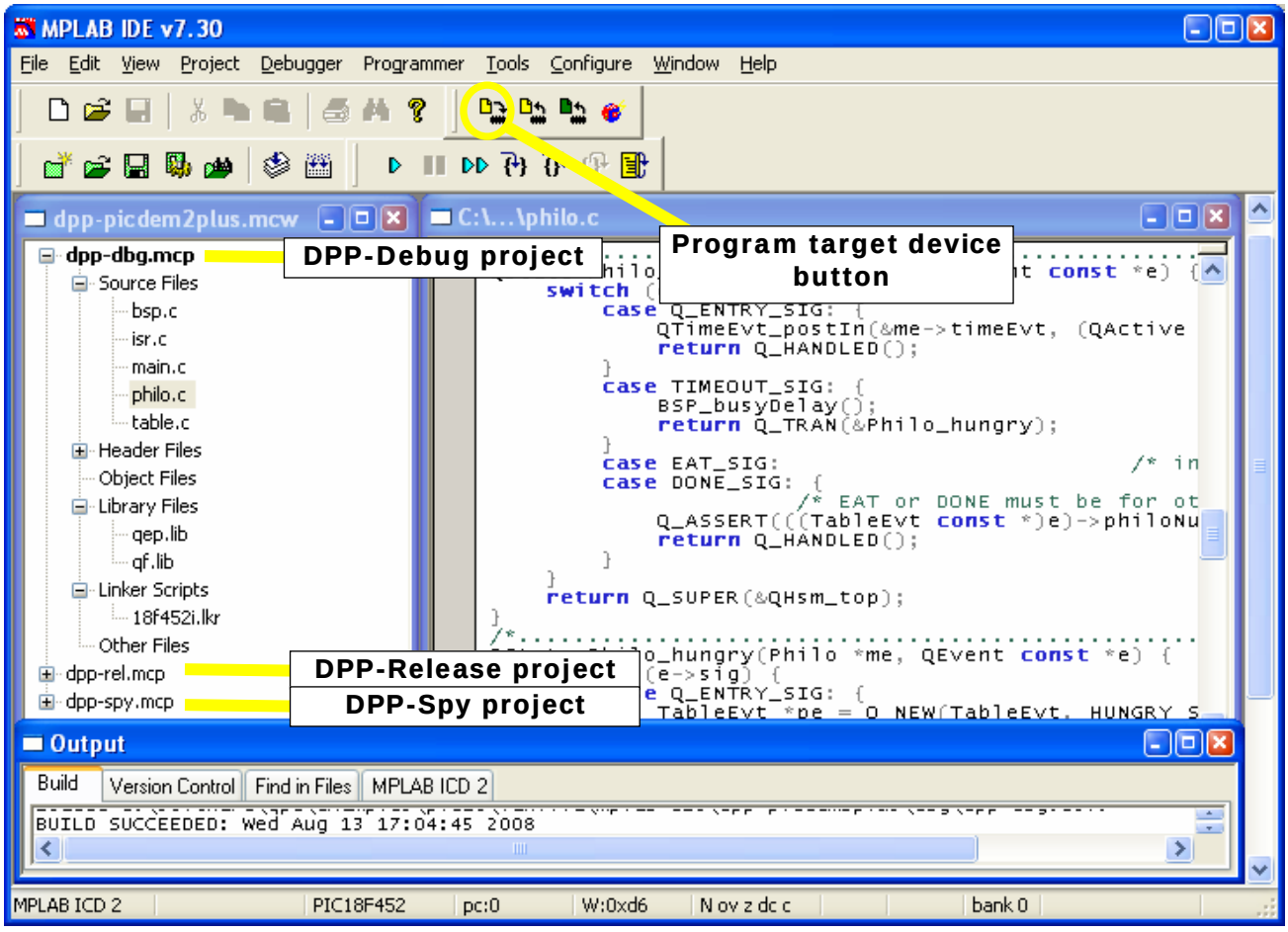


Figure 2 MPLAB IDE with the dpp-dbg.mcw workspace and 3 DPP projects

2.3.2 Running / Debugging the Examples

The MPLAB debugger can directly use the ICD 2 In-Circuit debugger (see Figure 1). To use the ICD 2 box, you need to configure the debugger by selecting the menu Debugger/Select Tool/MPLAB ICD 2. To load the code into the MCU's flash, click the "Program device" icon, as shown in Figure 3. This will start loading the code into the device. When the programmer is done, the "Run", "Animate", "Step Into", and "Step Over" icons become active and you'd be able to run the program.

Figure 3 shows the MPLAB IDE with the `dpp-debug.mcp` project active. Once the DPP example starts running, it used the 4 User LEDs start blinking. Each glowing LED corresponds to an "eating" Philosopher. The extinguished LED corresponds to "thinking" or "hungry" Philosopher.

2.3.3 Programming the Example to Run Standalone

The DPP example application comes with the `dpp-rel.mcp` project, which is intended for standalone execution on the PICDEM 2 PLUS board without the debugger. Once you activate and build this project in the MPLAB IDE, you need to de-select the ICD 2 as the Debugger and select ICD 2 as the Programmer (you cannot configure ICD 2 as a programmer and debugger simultaneously). To program the code into the PIC18 device, you click the "Program Target Device" icon. After the programming is done, you can release the device out of reset by clicking the appropriate icon. You can also disconnect the ICD 2 completely and power-cycle the board. The programmed PELICAN application should start running standalone.

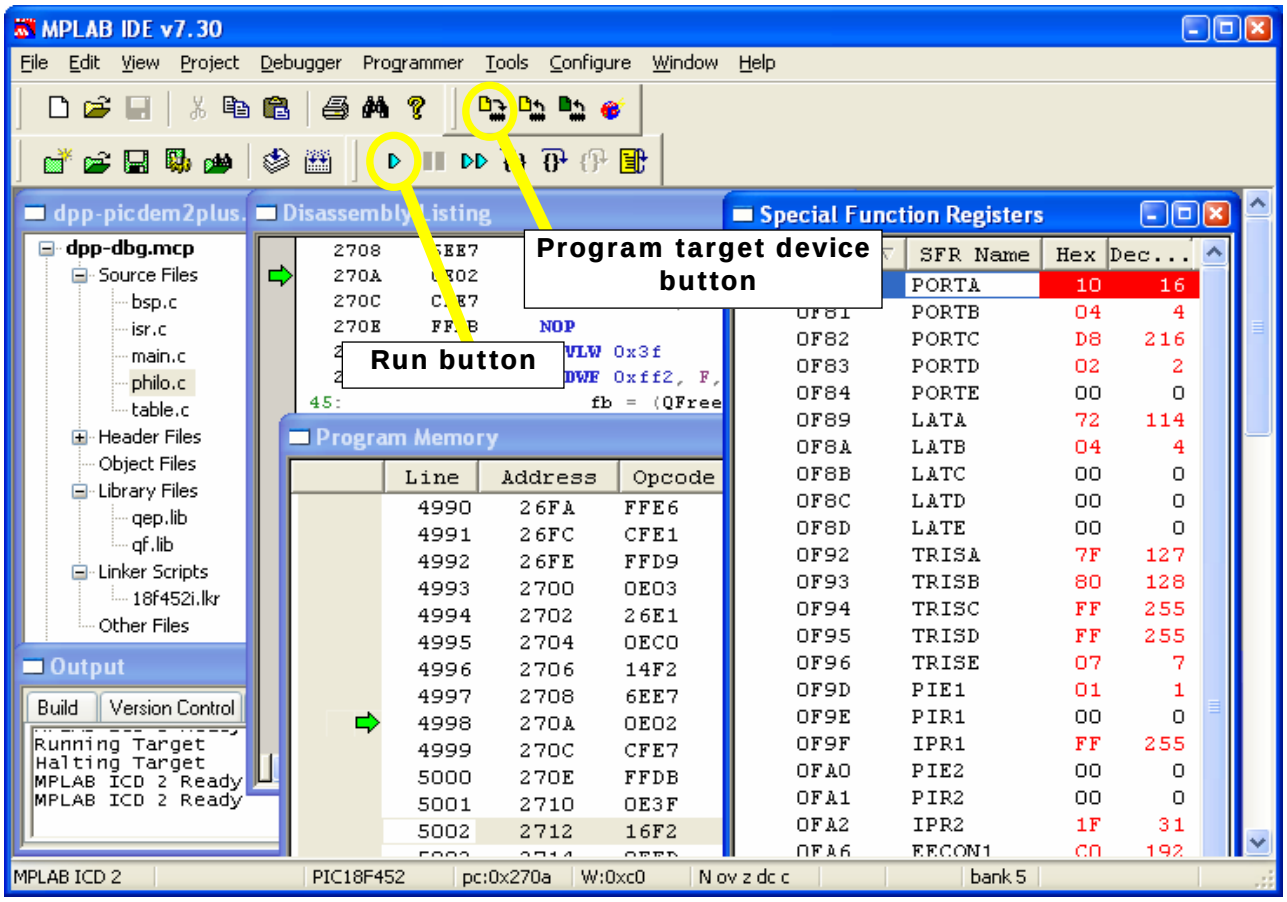


Figure 3 MPLAB Debugger with various views of the executing DPP application.

2.3.4 Collecting the QS software trace

QS is a software tracing facility built into all QP components and also available to the Application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (see Chapter 11 in [PSiCC2]).

The `dpp-spy.mcp` project contains the Spy configuration that enables the QS software tracing and links with the Spy-versions of the QP libraries (see Listing 1). You select the Spy configuration by activating the `dpp-spy.mcp` project in the MPLAB IDE (see Figure 2).

Before you download and start the DPP-Spy configuration to the PICDEM 2 PLUS board, you should connect the board's build-in DB9 connector to your workstation via a straight serial cable (see Figure 1). You can connect the QS output cable to the second COM port of the same workstation that you use for debugging, or to any COM port of another workstation (tracing workstation). Figure 4 shows the QSPY output received from the DPP-Spy configuration.

NOTE: Please note that only a small subset of the QSPY records are allowed through the QS filters. Please refer to the source code (`bsp.c`) and the "QSPY Reference Manual" [QSPY 08] for more information about using the QS filters.

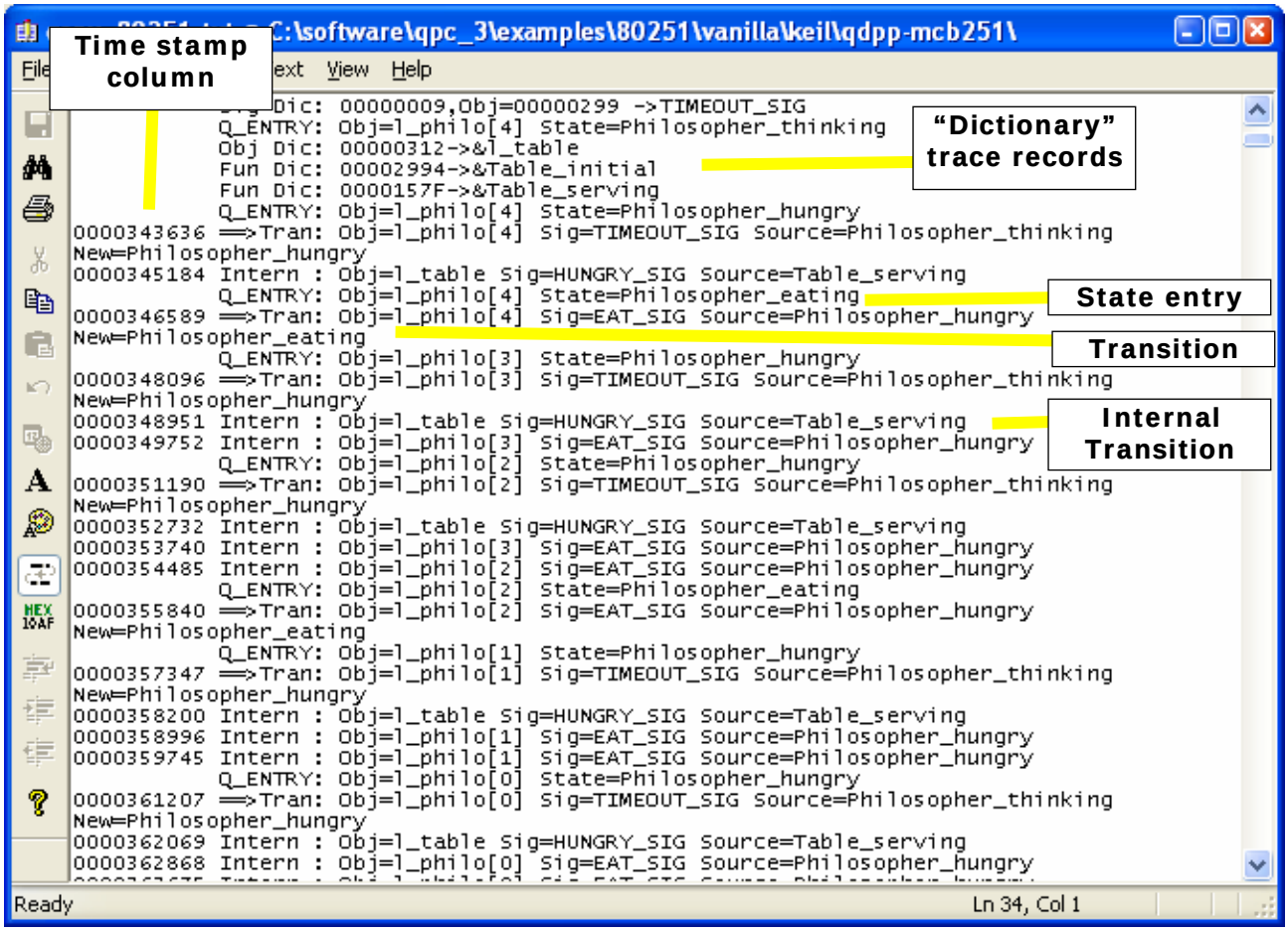


Figure 4 QSpy output from the DPP-QSpy configuration

You launch the QSPY utility on a Windows PC as follows. (1) Change the directory to the QSPY host utility `<qp>\tools\qspy\win32\mingw\rel` and execute:

```
qspy -cCOM1 -b19200 -O2 -F2 -E1 -P1 -B1
```

The meaning of the parameters is as follows:

-cCOM1 specifies COM port (change to actual COM port number you're using)

-b**19200** specifies the baud rate (depends on the oscillator frequency of your board, see Section 4)

-O2 specifies the size of an object pointer to 2 bytes

-F2 specifies the size of a function pointer to 2 bytes.

-E1 specifies the size of an event to 1 byte (an event may have up to 255 bytes).

-P1 specifies the size of a memory pool counter to 1 byte (a memory pool can manage up to 255 memory blocks).

-B1 specifies the size of a memory block to 1 byte (a memory pool can manage block up to 255 bytes).

3 The Vanilla QP Port

The "vanilla" port shows how to use QP state machine frameworks on a "bare metal" PIC18-based system without any underlying multitasking kernel. In the "vanilla" version of the QP, the only component requiring platform-specific porting is the QF. The other two components: QEP and QS require merely recompilation and will not be discussed here. Obviously, with the vanilla port you're not using the QK component. In case of PIC18, the "vanilla" QF port is very similar to the generic "vanilla" port described in Chapter 9 of [PSiCC2].

3.1 Compiler Options Used

You set the compiler options through the MPLAB IDE. The DPP crossing example has been compiled and tested with the following selections:

1. Code model: Small code model (<= 64K bytes)
2. Data model: Large data model (all RAM banks)
3. Stack model: Single-bank model
4. Optimization: Debug (for the Debug version), and Enable all (for the Release version)

However, QP does not assume any particular memory models or optimization levels, so you should be able to select any Code model, Data model, or Stack model appropriate for your PICmicro device and the application. To edit the project build options in the MPLAB IDE, you select menu Project/Build Options..., or you right click on the project `.mcp` file in the workspace window and select the Build Options... pop-up menu. Either way, you open the "Build Options".

NOTE: The Student Edition of the C18 compiler specifically disables the extended mode of the PIC18 devices, and also does not allow using the "procedural abstraction" optimization. This QDK should work with both these features, but does not require them.

3.2 Linker Options Used

You set the linker options through the MPLAB IDE and the linker command files `18f452i.lkr` (for the Debug version), and `18f452.lkr` (for the Release version), respectively. From the IDE you should select the INHX32 HEX-File format and you should suppress the COD-file generation.

The following listing shows the `18f452i.lkr` linker command file for PIC18F452:

```
    // File: 18f452i.lkr
    // Sample ICD2 linker script for the PIC18F452 processor
    LIBPATH .

(1) FILES c018iz.o
    FILES clib.lib
    FILES p18f452.lib

    CODEPAGE   NAME=vectors    START=0x0          END=0x29          PROTECTED
    CODEPAGE   NAME=page       START=0x2A         END=0x7DBF
    CODEPAGE   NAME=debug      START=0x7DC0       END=0x7FFF        PROTECTED
    CODEPAGE   NAME=idlocs     START=0x200000     END=0x200007      PROTECTED
    CODEPAGE   NAME=config     START=0x300000     END=0x30000D      PROTECTED
    CODEPAGE   NAME=devid      START=0x3FFFFE     END=0x3FFFFF      PROTECTED
    CODEPAGE   NAME=eedata     START=0xF00000     END=0xF000FF      PROTECTED
```

```
    ACCESSBANK NAME=accessram  START=0x0           END=0x7F
    DATABANK   NAME=gpr0       START=0x80          END=0xFF
    DATABANK   NAME=gpr1       START=0x100         END=0x1FF
    DATABANK   NAME=gpr2       START=0x200         END=0x2FF
    DATABANK   NAME=gpr3       START=0x300         END=0x3FF
    DATABANK   NAME=gpr4       START=0x400         END=0x4FF
    DATABANK   NAME=gpr5       START=0x500         END=0x5F3
    DATABANK   NAME=dbgspr     START=0x5F4         END=0x5FF          PROTECTED
    ACCESSBANK NAME=accesssfr  START=0xF80         END=0xFFF          PROTECTED

    SECTION    NAME=CONFIG     ROM=config

(2) STACK SIZE=0x100 RAM=gpr4
```

Listing 2 `18f452i.lkr` linker command file

(1) The startup code `c018iz.o` initializes all un-initialized static variables to zero (performs the zeroing of the BSS section), which is ANSI-C compliant. The standard PIC startup code `c018i.o` does NOT perform this standard initialization. QP assumes an ANSI-C compliance.

(2) Here you can configure the stack size for your application. The standard configuration allocates only one bank to the stack. Section 3.4.2 “Managing the Software Stack” in the “MPLAB C18 Compiler User’s Guide” [Microchip 05] describes how to allocate multiple banks to the stack.

3.3 The qep_port.h Header File

The QEP header file for PIC18 port with the MPLAB C18 compiler is located in `<qp>\ports\pic18\vanilla\mplab-c18\qep_port.h`. The most important aspect of the port is the mechanism of allocating constant objects, such as lookup tables or strings into ROM. The PIC18 is a Harvard architecture machine, and uses different instructions to access code space and data space. Therefore, the “`const`” declarator alone does not cause allocation constant objects to ROM. QP provides additional macro `Q_ROM` to enforce allocation constant objects to ROM. For the C18 compiler you need to define the `Q_ROM` macro as “`rom`” in the `qep_port.h` header file, as shown below.

```
          /* special extended keyword 'code' allocates const objects to ROM */
#define Q_ROM           rom

#define Q_SIGNAL_SIZE 1

/* exact-width integer types (HC(S)08 compiler does NOT provide <stdint.h>) */
typedef signed   char  int8_t;
typedef signed   int   int16_t;
typedef signed   long  int32_t;
typedef unsigned char  uint8_t;
typedef unsigned int   uint16_t;
typedef unsigned long  uint32_t;

#include "qep.h"              /* QEP platform-independent public interface */
```

Listing 3 The `qep_port.h` header file

3.4 The qf_port.h Header File

The QF header file for the PIC18 port with the MPLAB C18 compiler is located in `<qp>\ports\pic18\vanilla\mplab-c18\qf_port.h`. The following Listing 4 shows the `qf_port.h` header file. The following sections focus on explaining the QF configuration established by this header file.

```
             /* various QF object sizes configuration for this port, see NOTE00 */
 (1) #define QF_MAX_ACTIVE               8

 (2) #define QF_EVENT_SIZ_SIZE           1
 (3) #define QF_EQUEUE_CTR_SIZE          1
 (4) #define QF_MPOOL_SIZ_SIZE           1
 (5) #define QF_MPOOL_CTR_SIZE           1
 (6) #define QF_TIMEEVT_CTR_SIZE         2

                                    /* QF critical section entry/exit, see NOTE02 */
 (7) #define QF_INT_KEY_TYPE         unsigned char
 (8) #define QF_INT_LOCK(key_)       do { \
 (9)     (key_) = (INTCON & 0b11000000); \
(10)     INTCON &= 0b00111111; \
     } while (0)
(11) #define QF_INT_UNLOCK(key_)     (INTCON |= (key_))

                                    /* INTCON SFR declaration for PIC18 devices */
(12) extern volatile near unsigned char INTCON;

     #include "qep_port.h"                                        /* QEP port */
     #include "qvanilla.h"                      /* "Vanilla" cooperative kernel */
     #include "qf.h"                 /* QF platform-independent public interface */
```

Listing 4 The qf_port.h header file.

3.4.1 The QF Object Size Configuration

The first part of the qf_port.h header file defines limits and sizes of various internal data structures used in the QF and the applications.

Listing 4(1) `QF_MAX_ACTIVE` defines the maximum number of active objects that QF can manage. Here this limit is set to just 8, to save some RAM, but you can increase this limit up to 63, inclusive.

(2) Maximum event size is set to 1-byte, meaning that the size of a single event can be up to 255 bytes.

(3) Maximum event queue counter size is set to 1-byte, meaning that a single event queue can hold up to 255 events.

(4) Maximum memory pool element size is set to 1-byte, meaning that a pool can mange blocks of up to 255 bytes each.

(5) The memory pool counter size is set to 1-byte, meaning that a pool can manage up to 255 memory blocks.

(5) The timer counter size is set to 2-bytes, meaning that a maximum timeout can be 65535 clock ticks.

3.4.2 The QF Critical Section

The PIC18 CPU locks interrupts upon the entry to the interrupt. More precisely, if interrupt priorities are enabled, a high-priority interrupts locks all interrupts, but a low-priority interrupt locks only the low-priority interrupts. Either way, the body of an interrupt service routine (ISR) runs with interrupt locked at some level and represents thus a critical section of code. Unlocking interrupts inside ISRs is problematic and can lead to corruption of various registers and shared variables. Consequently, QF services invoked from the ISRs (such as `QF_tick()`, `QF_publish()`, `QActive_postFIFO`,

etc.) must not inadvertently unlock interrupts that are locked in hardware upon the entry to the ISR. All this means that QF must use an interrupt locking/unlocking scheme that allows for nesting critical sections. QF provides such a policy called "saving and restoring interrupt status" and described in Chapter 7 of [PSiCC2].

This QDK assumes that interrupt priorities are allowed (IPEN=1) and that high-priority interrupts can nest on low-priority interrupts. In this mode, the bit INTCON<7> is interpreted as GIEH (high-priority interrupt enable) and INTCON<6> is interpreted as GIEL (low-priority interrupt enable). The interrupt locking policy saves the current interrupt status (the bits GIEH and GIEL) and clears both GIEH and GIEL bits. The unlocking policy restores the saved state of the GIEH and GIEL bits. In particular, this policy will never enable interrupts inside high-level ISRs, and will enable only the high-level interrupts inside the low-level ISRs. You should not enable interrupts explicitly inside ISRs.

NOTE: setting IPEN=1 does not necessarily mean that interrupts will nest. You can always configure the priorities of all interrupts to high, which is equivalent to running in the compatibility mode.

Listing 4(7) "saving and restoring interrupt status" interrupt locking policy requires specifying the type of the "interrupt lock key" (see Chapter 7 of [PSiCC2]). Here `QF_INT_KEY_TYPE` is defined as `unsigned char`, consistently with the size of the `INTCON` register.

(8) The interrupt locking macro `QF_INT_LOCK()` uses the `do { } while (0)` loop for syntactically-correct grouping of instructions (and to avoid any dangling-`else` problems).

(9) The two MSB bits of the INTCON register (GIE/GIEH (INTCON<7>) and PEIE/GIEL (INTOCN<6>) are saved in the temporary "interrupt lock key" variable.

(10) The interrupts are locked by clearing the GPIE/GPIEH and PEIE/GIEL bits in the INTCON register.

(11) The saved interrupt status is restored into the INTCON register. The macro `QF_INT_UNLOCK()` restores the interrupts to exactly the same state they were before calling the `QF_INT_LOCK()` macro. In particular, inside the ISRs, the interrupts will NOT be inadvertently unlocked after exiting from the critical sections. Please note that all QF services (such as `QF_tick()`, `QActive_postFIFO()`, etc.) internally use critical sections.

(12) The INTON register is declared, independently on a particular PIC device header file.

3.5 BSP for PIC18

The Board Support Package (BSP) for PIC18 is very simple. However, there are some important details that you need to pay attention to. The BSP is minimal, but generic for most PIC18 devices.

3.5.1 BSP Header file bsp.h

```
                                                    /* The Oscillator frequency */
(1) #define BSP_FOSC_HZ          4000000UL
                                                   /* the system tick rate [Hz] */
(2) #define BSP_TICKS_PER_SEC    20UL

(3) void BSP_init(void);

(4) #include <p18C452.h>    /* SFRs for PIC18F452. Adjust for your PIC18 device */

    #ifdef Q_SPY
(5)     extern QSTimeCtr BSP_tickTime;
    #endif
```

Listing 5 BSP header file bsp.h

(1) The BSP defines the oscillator frequency used (4MHz for PICDEM 2 PLUS)

(2) The BSP defines the desired system clock-tick rate (in Hz).

(3) The `BSP_init()` function initializes the BSP and must be called at the beginning of `main()`.

(4) The BSP includes the SFR register declarations for the device family in use (here PIC18C452)

(5) If the QS software tracing is enabled, the variable `BSP_tickTime` keeps track of the timestamp at tick.

3.5.2 Board and QF Initialization

```
(1) void BSP_init(void) {
        TRISB = 0;                  /* data direction for Port B (LEDs): output */
        LATB  = 0;                                     /* extinguish all LEDs */
(2)     RCONbits.IPEN = 1;                           /* enable priority levels */

(3)     if (QS_INIT((void *)0) == 0) {    /* initialize the QS software tracing */
            Q_ERROR();
        }
    }
```

Listing 6 BSP and QF initialization

(1) The `BSP_init()` function is called from `main()` to initialize the board

(2) Consistently with the interrupt locking policy, interrupt priories are enabled.

(3) The QS component is initialized only in the Spy build configuration.

3.5.3 Starting Interrupts in QF_onStartup()

The QF_onStartup() callback is invoked from QF_run() just before starting the event loop. The QF_onStartup() function is located in the file isr.c and must configure and start the interrupts, in particular the time-tick interrupt. In this BSP only the timer tick interrupt is started. Please note that QF_onStartup() must also set the global interrupt flags GIE/GIEH and PEIE/GIEL.

NOTE: The BSP uses Timer1 as the system clock tick source. Timer1 is configured to use the external clock source, which is for the PICDEM 2 PLUS board is the 32kHz oscillator. The Timer1 is also configured not to use the clock synchronization. This is all done to enable the system tick interrupt to terminate the SLEEP power-saving mode available in PIC18.

If you are not interested in using the SLEEP mode, you can use any other timer (such as Timer0, Timer2, or Timer3) as the system clock tick source.

```
    /*..........................................................................*/
(1) void QF_onStartup(void) {                 /* entered with interrupts locked */
        TMR1H = 0;
        TMR1L = 1;                                  /* make TMR1 exprie immediately */
(2)     T1CON = 0x8F;      /* ext Timer1 clock, 16-bit, no synch, prescaler 1:1 */
(3)     IPR1bits.TMR1IP = 1;             /* set TMR1 interrupt as high priority */
(4)     PIE1bits.TMR1IE = 1;                        /* enable TMR1 interrupt */

(3)     INTCON |= 0b11000000;                       /* unlock the interrupts */
    }
```

Listing 7 QF_onStartup() function

(1) The QF_onStartup() callback is invoked from QF_run() to configure and start interrupts.

(2) The Timer1 is configured to 16-bit mode, no synchronization, 1:1 prescaler.

(3) The Timer1 interrupt priority is configured to HIGH.

(4) The interrupts are enabled by setting GIE/GIEH and PEIE/GIEL. Please note that the macro QF_INT_UNLOCK() is not used, because this macro unlocks interrupts conditionally based on the "interrupt lock key", which is not available at this point.

3.5.4 ISRs

This QDK configures the PIC18 to enable interrupt priorities (IPEN bit RCON<7> is set). This means that you might have high and low priority interrupts with vectors 0x08 and 0x18, respectively

NOTE: Using the interrupt priorities does not necessarily mean that interrupt nesting is allowed. You can always configure the priorities to all interrupts to HIGH, in which case the interrupts will be handled exactly as in the compatibility mode (with interrupt priorities disabled.)

The GIEH and GIEL flags in the INTCON register are automatically cleared upon the entry to the high-priority interrupt. The GIEH and GIEL flags are also automatically set upon the high-priority interrupt exit. This means that high-priority interrupts execute with interrupts disabled, which prevents nesting of interrupts.

In contrast, only the GIEL flag is automatically cleared upon the entry to the low-priority interrupt. This means that a high-priority interrupt can preempt a low-priority interrupt. The MPLAB C18 compiler introduces some temporary variables and sections, which might get corrupted when interrupt nesting is enabled Please read **carefully** the pertinent section in the "MPLAB

C18 Compiler User's Guide" before you decide to use low-level interrupts with the MPLAB C18 compiler..

The MPLAB C18 compiler provides mechanisms for writing ISR directly in C. This QDK can use all these mechanisms.

```
    #define BSP_TMR1_HZ          32768UL
    #define SYSTEM_TICK_TOUT     (0x10000 - BSP_TMR1_HZ/BSP_TICKS_PER_SEC)

    /*.....................................................................*/
(1) #pragma code
(2) #pragma interrupt ISR_hi
(3) void ISR_hi(void) {                                     /* Hi-priority ISR */

(4)     if (PIR1bits.TMR1IF) {                              /* TMR1 interrupt? */
(5)         PIR1bits.TMR1IF = 0;                    /* clear TMR1 interrupt flag */

                                                 /* program TMR1 for the next tick */
(6)         TMR1H = (uint8_t)(SYSTEM_TICK_TOUT >> 8);
(7)         TMR1L = (uint8_t)(SYSTEM_TICK_TOUT);

(8)         QF_tick();                      /* handle all armed time events in QF */
        }
(9)  #ifdef Q_SPY
(10)    else if (INTCONbits.TMR0IF) {                       /* Timer0 overflow? */

(11)        INTCONbits.TMR0IF = 0;              /* clear the TMR0 interrupt flag */
(12)        BSP_tickTime += (QSTimeCtr)0x10000;    /* account for TMR0 overflow */

        }
    #endif
(13)    /*
        else if (...) {                           // handle other interrupt sources
            . . .
        }
        */
    }

(14) #pragma code intVector_hi = 0x08
    void intVector_hi(void) {                   /* Hi-priority interrupt vector */
        _asm
(15)        goto ISR_hi                                     /* jump to ISR_hi */
       _endasm
    }
```

Listing 8 ISR handler for the PIC18.

(1) The ISR and interrupt vector must be located in the code section

(2) The ISR in C is always compiler-specific, as the C standard does not define how to specify ISRs. In the case of the MPLAB C18 compiler, interrupt must be preceded with the `#pragma interrupt` followed by the interrupt handler name. This tells the C18 compiler to synthesize the correct interrupt entry and exit for the ISR.

(3) The ISR in C must have a `void (void)` signature. If interrupt prioritization is not used, there is only one ISR (the High Level ISR) in the PIC18 system vectored from 0x08.

(4) The ISR tests the interrupt conditions. For the Timer1 interrupt, the TMR1IF flag is set in the PIR1 register.

(5) The TMR1IF needs to be cleared in software.

(6-7) The Timer1 16-bit counter is set to achieve the next overflow in the specified number of Timer1 clocks. As described in the "PIC18FXX2 Data Sheet" [Microchip 02], the actual writing to the Timer1 counter happens atomically at the time TMR1L register is written.

(8) The time-tick ISR must invoke `QF_tick()`, and can also perform other things, if necessary. The function `QF_tick()` cannot be reentered, that is, it necessarily must run to completion and return before it can be called again. This requirement is automatically fulfilled, because here `QF_tick()` is invoked with interrupts locked.

(9) The ISR can test other interrupt flags and process other interrupt sources.

(10) The high-priority interrupt vector is located at absolute address 0x08.

(11) As described in the "MPLAB C18 Compiler User's Guide" [Microchip 05], the interrupt vector typically contains the single assembly `GOTO` instruction to branch to the ISR.

3.6 QP Idle Processing Customization in QF_onIdle()

QP can very easily detect the situation when no events are available. In case of non-preemptive configuration, QP calls `QF_onIdle()` callback that you can use to perform some idle processing. The `QF_onIdle()` callback is invoked with interrupts **locked**, and you must necessarily unlock the interrupts inside this callback:

If he processor supports a power saving mode, `QF_onIdle()` is the ideal place to take advantage of this. The PIC18 microcontroller supports the SLEEP power saving mode. This mode is activated with the SLEEP assembly instruction.

```
(1) void QF_onIdle(void) {                      /* entered with interrupts LOCKED */

(2) #ifdef NDEBUG
(3)    Sleep();                          /* transition to SLEEP mode, see NOTE02 */
    #endif

(4)    QF_INT_UNLOCK();                         /* unlock interrupts, see NOTE01 */
   }
```

Listing 9 The QF_onIdle() callback for PIC18.

(1) The `QF_onIdle()` callback is invoked with interrupts locked.

(2) The SLEEP mode stops the CPU clock and thus can interfere with the debugger. Therefore, the following transition to the low-power SLEEP mode is conditionally compiled only when the macro NDEBUG is defined (which is defined only in the Release configuration.)

(3) According to the "PIC18FXX2 Data Sheet", the Sleep mode can be exited even if the global inetrrupt enable flag is cleared (INTCON<7> == 0). This allows for an atomic transition to the SLEEP mode. Selected interrupts, such as Timer1 interrupt with external clock (not synchronized to the CPU clock) can wake up the CPU from the SLEEP mode.

NOTE: After waking up, the CPU does not service the interrupt immediately, because interrupts are still disabled. The interrupt gets serviced only after the CPU unlocks interrupts.

3.7 Assertion Handling Policy in Q_onAssert()

As described in Chapter 6 of [PSiCC2], all QP components use internally assertions to detect errors in the way application is using the QP services. You need to define how the application reacts in case of assertion failure by providing the callback function `Q_onAssert()`. Typically, you would put the system in fail-safe state and try to reset. It is also a good idea to log some information as to where the assertion failed.

The following code fragment shows the `Q_onAssert()` callback for the PIC18. The function simply locks all interrupts and enters a for-ever loop in which it waits and preserves the system state for inspection in a debugger. This policy is only adequate for testing, but probably is **not** for production release.

```
void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
    (void)file;                                     /* avoid compiler warning */
    (void)line;                                     /* avoid compiler warning */
    QF_INT_LOCK();                                  /* cut off the interrupts */
    for (;;) {   /* NOTE: replace the loop with reset for the final version */
    }
}
```

4 The QP-Spy (QS) Instrumentation

This QDK demonstrates how to use the QS software tracing instrumentation to generate real-time trace of a running QP application. Normally, the QS instrumentation is inactive and does not add any overhead to your application, but you can turn the instrumentation on by defining the Q_SPY macro and recompiling the code.

QS is a software tracing facility built into all QP components and also available to the Application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (see Chapter 11 in PSiCC2 [PSiCC2]).

QS can be configured to send the trace data out of the serial port of the target device. On the PIC18, QS uses the built-in UART to send the trace data to the host. The PICDEM 2 PLUS board has the RS232 level-shifter already installed and available as the serial port (see Figure 1). The QS platform-dependent implementation is located in the file bsp.c and looks as follows:

```
(1) #ifdef Q_SPY

    #define QS_BUF_SIZE        252U
    #define BAUD_RATE          19200U

(2) QSTimeCtr BSP_tickTime;

(3) uint8_t QS_onStartup(void const *arg) {
(4)     static uint8_t qsBuf[QS_BUF_SIZE];           /* buffer for Quantum Spy */
        uint16_t n;

(5)     QS_initBuf(qsBuf, sizeof(qsBuf));     /* initialize the QS trace buffer */

        /* initialize the USART for transmitting the QS trace data */
(6)     SPBRG = BSP_FOSC_HZ/16/BAUD_RATE - 1;   /* set up for desired baud rate */
        RCSTAbits.SPEN = 1;                              /* enable serial port */
        TXSTAbits.SYNC = 0;                               /* asynchronous mode */
        TXSTAbits.BRGH = 1;                                    /* high TX rate */
        TXSTAbits.TXEN = 1;                                       /* enable TX */

(7)     return (uint8_t)1;            /* indicate successfull QS initialization */
    }
    /*..........................................................................*/
(8) void QS_onCleanup(void) {
    }
    /*..........................................................................*/
(9) void QS_onFlush(void) {
        uint16_t b;
(10)    while ((b = QS_getByte()) != QS_EOD) { /* next QS trace byte available? */
(11)        while (PIR1bits.TXIF == 0U) {                  /* TXREG not empty? */
            }
(12)        TXREG = (uint8_t)b;     /* stick the byte to TXREG for transmission */
        }
}
    /*..........................................................................*/
    /* NOTE: invoked within a critical section (inetrrupts disabled) */
(13) QSTimeCtr QS_onGetTime(void) {
(14)    uint8_t tlow = TMR0L;    /* read TMR0L and latch TMR0H at the same time */
(15)    if (INTCONbits.TMR0IF == 0U) {          /* TMR0 overflow accounted for? */
(16)        return BSP_tickTime
                  + (QSTimeCtr)(((uint16_t)TMR0H << 8) | (uint16_t)tlow);
        }
        else {              /* the TMR0 overflowed, but the ISR did not run yet */
```

```
(17)        return BSP_tickTime + (QSTimeCtr)0x10000
                    + (QSTimeCtr)(((uint16_t)TMR0H << 8) | (uint16_t)tlow);
            }
        }
        #endif                                                          /* Q_SPY */
```

Listing 10 QS implementation to send data out of the on-chip UART of the PIC18.

(1) The QS instrumentation is enabled only when the macro `Q_SPY` is defined

(2) The `BSP_tickTime` variable is used to keep track of the timestamp at tick to extend the 16-bit Timer0 timestamp to 32 bits.

(3) The `QS_init()` function initializes the QS software tracing facility.

(4) The QS trace buffer is statically allocated to the specified size

(5) The `QS_init()` function must call the `QS_initBuf()` function to initialize the trace buffer.

NOTE: On the PIC18 with Small memory model the QS trace buffer must fit within one RAM bank, that is, it cannot be bigger than 256 - 4 bytes.

(6) The QS instrumentation uses PIC18 USART configured as asynchronous port, high baud rate, 8-bits, no parity. The generic formula for the baud rate is in this case:

```
Baud_Rate = Fosc/16/(SDBRG + 1)
```

The best mach to the standard PC baud rate for Fosc=4MHz occurs for the 19200 baud rate. For different CPU clock frequency, you should choose the baud rate carefully to avoid transmission errors. Table 16.5 in the “PIC18FXX2 Data Sheet” [Microchip 02] provides SDBRG register values for different oscillator frequencies Fosc and different desired baud rates.

(7) The function `QS_onStartup()` returns success to the caller (QS initialized successfully).

(8) The `QS_onCleanup()` function is empty in this QS port.

(9) The `QS_onFlush()` callback function flushes the QS trace buffer to the output port. This function is only used during the initial transient to output the “dictionary” records. The function call takes up several milliseconds to output the whole buffer.

(10) The `QS_onFlush()` callback function uses the byte-oriented interface provided by QS. The `QS_getByte()` function returns a 16-bit value. The `QS_getByte()` function returns `QS_EOD` (0xFFFF) when the QS buffer is empty.

(11) This `while()` loop waits as long as the Serial transmitter is busy.

(12) The new byte (the LSB of the value returned from `QS_getByte()`) is inserted to the serial buffer register `TXREG`.

(13) The `QS_onGetTime()` callback function provides the microsecond-level time-stamps to QS records.

The platform-specific QS port must provide function `QS_oGetTime()` that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the PIC18 port uses Timer0, which is operated in “free running” 16-bit mode, that is, it naturally wraps around from 0xFFFF to 0.

To extend the 16-bit counter to 32-bit timestamp, the Timer0 overflow interrupt is used to count the number of overflows. The implementation of the function QS_onGetTime() combines information from the Timer0 counter and the interrupt to produce a monotonic 32-bit-wide time stamp.

(14) The PIC18 Timer0 hardware is designed such that reading the TMR0L register latches the high-byte content of the running Timer0 counter into the TMR0H register. This guarantees that the (TMR0L, TMR0H) pair of registers contains the valid timer value.

(15) The TMR0IF flag is examined to check if Timer2 auto-reloaded, but the clock-tick ISR has not had a chance to run yet (interrupts are disabled).

(16) If the wrap-around did not occurred the function returns the "fine time" from the (TMR0L, TMR0H) counter pair added to the timestamp-at-tick.

(17) Otherwise, the wrap-around happened, the function returns (TMR0L, TMR0H) counter pair incremented by one rollover 0x10000 added to the timestamp-at-tick.

4.1 QS Output in QF_onIdle()

The QS trace transmission is placed in the idle processing QF_onIdle(), which guarantees very-low impact of the tracing on the normal timing. The QF_onIdle() callback has been already discussed in Section 3.6. This section explains the details of the QS trace output.

```
    void QF_onIdle(void) {                          /* entered with ints. LOCKED */
        . . .
(1) #ifdef Q_SPY
(2)     if (PIR1bits.TXIF != 0U) {                              /* TXREG empty? */
(3)         uint16_t b = QS_getByte();
(4)         if (b != QS_EOD) {                  /* next QS trace byte available? */
(5)             TXREG = (uint8_t)b; /* stick the byte to TXREG for transmission */
            }
        }
    #elif defined NDEBUG
        Sleep();
    #endif

(6)     QF_INT_UNLOCK(intKey);                  /* unlock interrupts, see NOTE01 */
    }
```

Listing 11 QS trace output in QF_onIdle().

(1) The QS output is active only in the Spy version (Q_SPY defined)

(2) The transmit interrupt flag is tested to see if the transmit has completed.

(3) If so, the next byte is requested from the QS trace buffer.

NOTE: The QF_onIdle() callback is invoked with interrupts locked, so accessing the QS trace buffer is safe. (The QS_getByte() does not lock interrupts internally, so there is no risk of nesting critical sections.)

(4) The return value from QS_getByte() is checked for End-Of-Data (EOD) condition.

(5) If the data is available, the new byte is copied to the TXREG, which also clears the transmit flag.

(6) The interrupts can be unlocked.

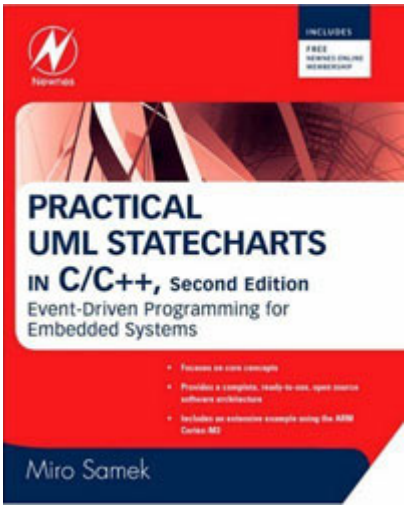
5 Related Documents and References

Document	Location
[Samek 08] "Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.quantum-leaps.com/writings/psicc2.htm
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2008	http://www.quantum-leaps.com/doxygen/qpc/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf
"PIC18FXX2 Data Sheet", 2002, [Microchip 02]	Document DS39564B available in PDF from the Microchip website www.microchip.com
"MPLAB C18 Compiler User's Guide", 2005, [Microchip 05]	Document DS51288J available in PDF from the Microchip website www.microchip.com

6 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)
e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com



"*Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems*", by Miro Samek, Newnes, 2008

Atmel Corporation
2325 Orchard Parkway
San Jose, CA 95131
USA
+1 408 441 0311
WEB: www.Atmel.com

Microchip Technologies, Inc.
Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200 Fax: 480-792-7277
Technical Support: 480-792-7627
Web Address: http://www.microchip.com