



**Quantum<sup>TM</sup> Leaps**  
innovating embedded systems

# **QDK<sup>TM</sup>**

## **80251-Keil**

**Document Revision D**  
**September 2008**

Copyright © Quantum Leaps, LLC

[www.quantum-leaps.com](http://www.quantum-leaps.com)  
[www.state-machine.com](http://www.state-machine.com)



# Table of Contents

<b>1</b>	<b>Introduction.....</b>	<b>1</b>
1.1	What's Included in the QDK-80251-Keil? .....	2
1.2	Licensing QDK-80251-Keil.....	2
1.3	Related Documentation and Resources.....	2
<b>2</b>	<b>Getting Started .....</b>	<b>3</b>
2.1	Installation .....	3
2.2	Building the QP Libraries.....	4
2.3	Building the Example.....	6
2.4	Running the Example .....	9
2.4.1	Collecting the QSpy trace.....	10
<b>3</b>	<b>The Vanilla QP Port .....</b>	<b>12</b>
3.1	Compiler Options Used .....	12
3.2	Linker Options Used .....	12
3.3	The qep_port.h Header File .....	13
3.4	The qf_port.h Header File.....	13
3.4.1	The QF Object Size Configuration .....	14
3.4.2	The QF Critical Section .....	14
3.4.3	The PCON Register Shadow for Thread-Safe Transition to Idle Mode.....	14
3.5	Startup Code.....	15
3.6	BSP for 80251 .....	15
3.6.1	BSP Header file bsp.h .....	15
3.6.2	Board Initialization and the System Timer Tick.....	16
3.6.3	Starting Interrupts in QF_onStartup() .....	16
3.6.4	ISRs .....	16
3.7	QP Idle Processing Customization in QF_onIdle() .....	18
3.8	Assertion Handling Policy in Q_onAssert() .....	19
<b>4</b>	<b>The QS Software Tracing Instrumentation .....</b>	<b>20</b>
4.1	QS Time Stamp Callback QS_onGetTime().....	20
4.2	QS Platform-Specific Implementation in bsp.c .....	22
4.3	QS Output in QF_onIdle() .....	24
<b>5</b>	<b>Related Documents and References .....</b>	<b>26</b>
<b>6</b>	<b>Contact Information.....</b>	<b>27</b>

---



# Cx51

## 8051/251 Development Tools



## 1 Introduction

This **QP Development Kit™** (QDK) describes how to use QP™ state machine frameworks with the 80251 MCUs and the Keil C51 compiler. The actual hardware/software used to test this QDK is described below (see also Figure 1):

1. Keil evaluation board MCB251 running at 11.059MHz (see “MCBx51 Evaluation Board User’s Guide”).
2. Keil C251 compiler v3.51, Keil A251 assembler v3.51, Keil L251 Linker/Locator v3.51.
3. Keil µVision IDE v2.31.
4. QP/C v4.0.01 or higher.

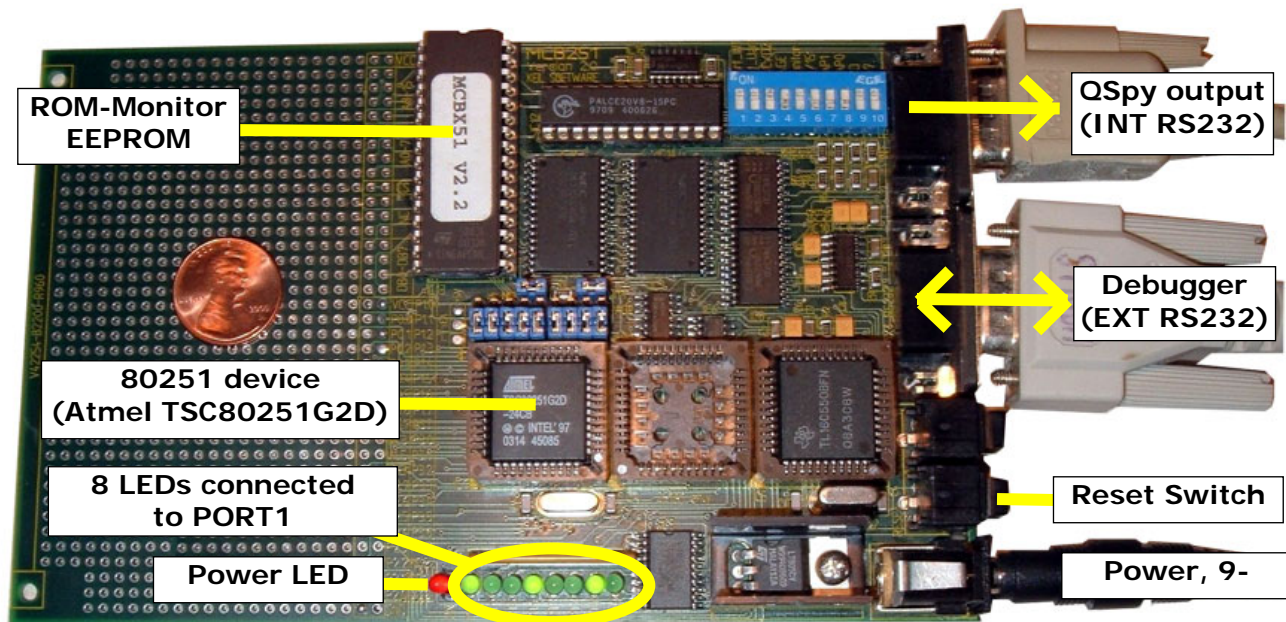


Figure 1 Keil MCB251 evaluation board connected to the external debugger (EXT RS232), the QSpy output cable (INT RS232), and external power.

As shown in Figure 1, the MCB251 board is connected to the Keil  $\mu$ Vision debugger via the EXT RS232 DB9 connector and straight serial cable. Another straight serial cable connects the INT RS232 to the QSpy host utility running on the host workstation.

This QDK has been tested with the Atmel TSC80251G2D device inserted into the IC1 socket of the board with 1KB of on-chip RAM and 32KB of onboard ROM. However, the QDK-80251-Keil should be applicable to all 80251 devices big enough to accommodate QP, that is, with RAM > 0.5KB, and ROM > 8KB.

## 1.1 What's Included in the QDK-80251-Keil?

This QDK provides the QP port to the 80251 processor, the Board Support Package (BSP) for the 80251 and the Dining Philosopher Problem (DPP) example application described in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08].

**NOTE:** Currently, the QDK-80251-Keil 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-80251-Keil

The **Generally Available (GA)** distribution of QDK-80251-Keil available for download from the [www.state-machine.com/8051](http://www.state-machine.com/8051) 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](http://www.state-machine.com/licensing).

## 1.3 Related Documentation and Resources

This QDK describes only elements of the implementation specific to 80251, the Keil Cx51 compiler, and the Keil  $\mu$ Vision IDE toolset, but does not cover other related subjects, such as: QP™ description, state machine basics, UML notation, state machine design, real-time concepts, and others. Please refer to the documents listed in the Section "References" for information about the related subjects.

## 2 Getting Started

This section describes how to install, build, and use QDK-80251-Keil based two examples. 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](http://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\_80251-keil\_<ver>.zip, where <ver> stands for a specific QDK-80251-Keil version, such as 4.0.01). You can 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-80251-Keil distribution. (Please note that the QP directory structure is described in detail in a separate Application Note: "[QP Directory Structure](#)"):

```

<qp>/
|--include/
|  |--qassert.h
|  |--qep.h
|  |--qf.h
|  |--qk.h
|  |--qs.h
|  |--qqueue.h
|  |--qmpool.h
|  |--qpset.h
|  |--qs_dummy.h
|--ports/
|  |--80251/
|  |  |--vanilla/
|  |  |  |--keil/
|  |  |  |  |--dbg/
|  |  |  |  |  |--QEP.LIB
|  |  |  |  |  |--QF.LIB
|  |  |  |  |--rel/
|  |  |  |  |  |--QEP.LIB
|  |  |  |  |  |--QF.LIB
|  |  |  |  |--spy/
|  |  |  |  |  |--QEP.LIB
|  |  |  |  |  |--QF.LIB
|  |  |  |  |  |--QS.LIB
|  |  |--make.bat
|  |  |--qep_port.h
|  |  |--qf_port.h
|  |  |--qs_port.h
|  |  |--STDINT.H
|--examples/

```

- QP-root directory for Quantum Platform (QP)
- QP public include files
  - Quantum Assertions platform-independent public include
  - QEP platform-independent public include
  - QF platform-independent public include
  - QK platform-independent public include
  - QS platform-independent public include (optional)
  - native QF event queue include
  - native QF memory pool include
  - native QF priority set include
  - dummy QS macros definition (used when QS is not configured)
- QP ports
  - 80251 port
    - "vanilla" ports (non-preemptive scheduler)
      - Keil 80251 compiler
        - Debug build directory
          - QEP library (debug build)
          - QF library (debug build)
        - Release build directory
          - QEP library (release build)
          - QF library (release build)
        - Spy build directory
          - QEP library (instrumented Qspy build)
          - QF library (instrumented Qspy build)
          - QS library (instrumented Qspy build)
      - Batch file to build the QP libraries
      - QEP port header file
      - QF port header file
      - QS port header file
      - C99-standard exact-width integers header file for Cx51
  - subdirectory containing the QP example files

```

+-80251/          - 80251 port
+-vanilla/       - "vanilla" ports (non-preemptive scheduler)
+-keil/          - Keil 80251 compiler
+-dpp-mcb251/    - Dining Philosophers example for the MCB251 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 MCB2051
+-bsp.h          - BSP header file
+-ISR.c          - Interrupt Service Routines (ISRs) for the application
+-main.c         - the main function
+-phil.o.c       - the Philosopher active object
+-dpp.h          - the DPP header file
+-table.c        - the Table active object
+-START251.A51  - the startup code (to customize the stack size, etc.)
+-dpp-mcb251.Opt - the Keil µVision project options
+-dpp-mcb251.Uv2 - the Keil µVision project to build the example
  
```

**Listing 1** Selected QP directories and files after installing QDK-80251-Keil. The highlighted elements are included in the standard QDK-80251-Keil 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 µVision IDE, for building the QP libraries. Instead, QP supports *command-line* build process based on simple batch scripts.

The code distribution contains the batch file `make.bat` for building all the libraries located in <qp>\ports\80251\vanilla\keil\ directory. For example, to build the debug version of all the QP libraries for the 80251, with the Keil C251 compiler, you open a console window on a Windows PC, change directory to <qp>\ports\80251\vanilla\keil\, 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\80251\vanilla\keil\dbg\. The `make.bat` assumes that the Keil 80251 toolset has been installed in the directory C:\tools\Keil\C251.

**NOTE:** You need to adjust the symbol `KEIL_C251_DIR` at the top of the `make.bat` file if you've installed the Keil C251 compiler into a different directory. You might also need to adjust the symbol `CCFLAGS` to use a different memory model (currently the memory model is set to `XTINY`).

In order to take advantage of the QS (Q-Spy) software tracing 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\80251\vanilla\keil\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 software versions**

## 2.3 Building the Example

This QDK-80251-Keil includes the DPP (“Dining Philosophers”) example described in the Application Note “Dining Philosophers Problem Example” [QP AN-DPP 08]. The QDK includes a  $\mu$ Vision project to build the three configurations of the example (Debug, Release, and Spy). The project is located in `<qp>\examples\80251\vanilla\keil\dpp-mcb251\dpp-mcb251.Uv2` for the “vanilla” version.

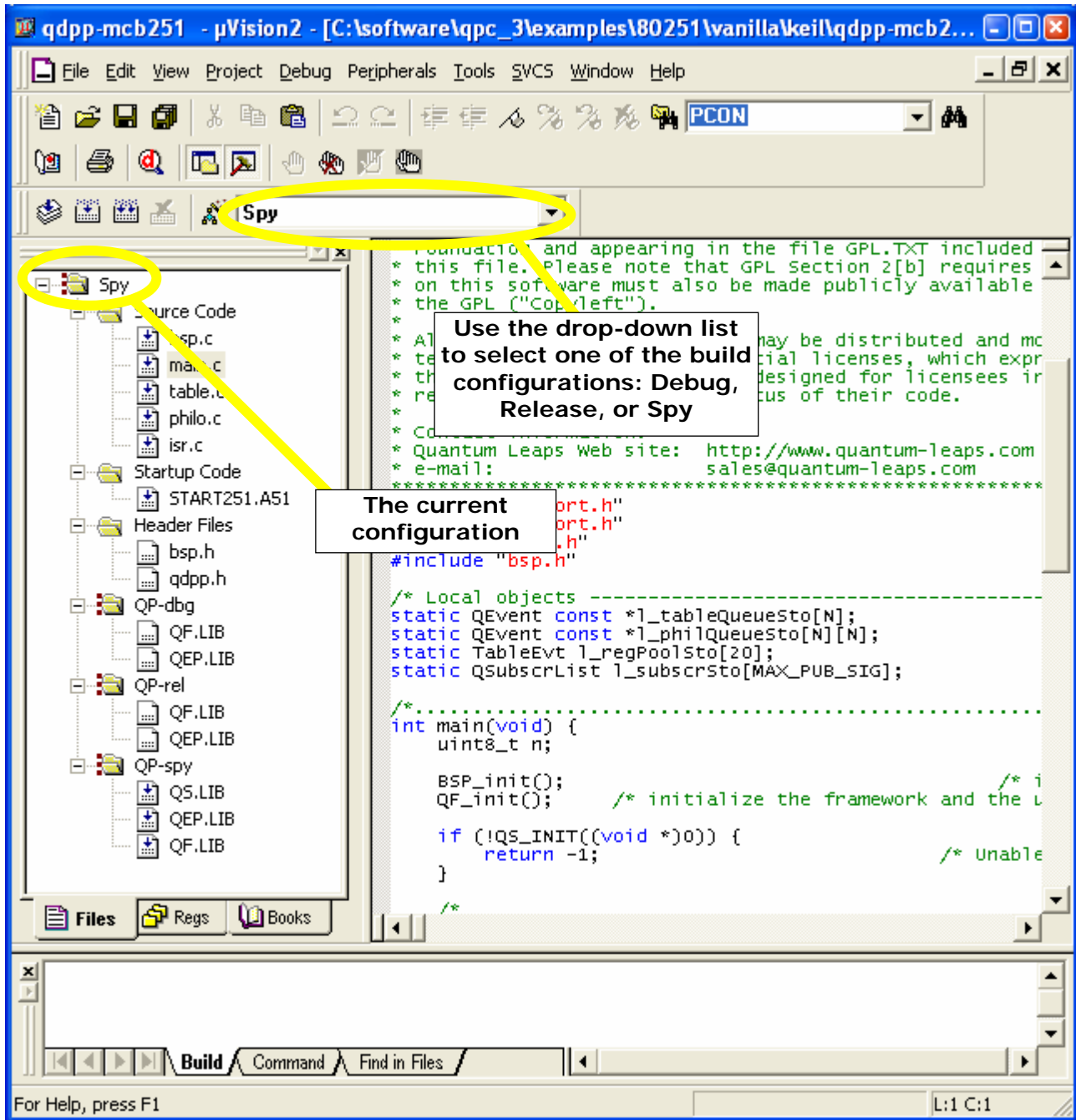


Figure 2 Keil  $\mu$ Vision IDE with the dpp-mcb251.Uv2 project

Figure 2 shows the Keil  $\mu$ Vision IDE with the DPP project open. You can use the drop-down list in to select one of the build configurations: Debug, Release, or Spy. The Spy configuration requires installing the QS component.

To build any selected configuration, you simply press F7, or alternatively select Project->Build Target ... menu. Either way, you end up opening the "Options for Target ..." tabbed dialog box. The most important options in this dialog box are as follows:

1. Device tab: Intel/8xC251SB
2. Target tab: Xtal: 12.0 (adjust to the Xtal frequency of your board), CPU Mode: Source (251 native), Memory Model: XTiny, Code Rom Size: Large 64K program, 4 Byte Interrupt Frame Size, #1 ROM start 0xFF0000 size 0x8000, #2 RAM start 0x0000 size 0x0400
3. Output tab: Select Folder for Objects (.\\dbg), Name of Executable: dpp-mcb251, Create Executable, Debug Information, Create HEX file
4. Listing tab: Select Folder for Listing (.\\dbg), Linker listing
5. C251 tab: Code Optimization 6: Constant propagation, **Generate reentrant functions**, Include Paths: . . . . \\ . . . . \\ . . . . \\ include, . . . . \\ . . . . \\ . . . . \\ ports\80251\vanilla\keil, Misc Controls: FIXDRK WARNING(disable=138) (see Figure 3(a))
6. A251 tab: Misc Controls: FIXDRK REGISTERBANK(0)
7. L251 Locate tab: Use Memory Layout from Target Dialog, Reserve: 0xFF0003-0xFF0005, 0xFF003B-0xFF003D, 0xFF007B-0xFF007D
8. L251 Misc tab: Misc controls: NOOVERLAY (see Figure 3(b))
9. Debug tab: Use: Keil Monitor-251 Driver. Settings dialog: Port Com 1, Baudrate 57600, Enable Serial Break

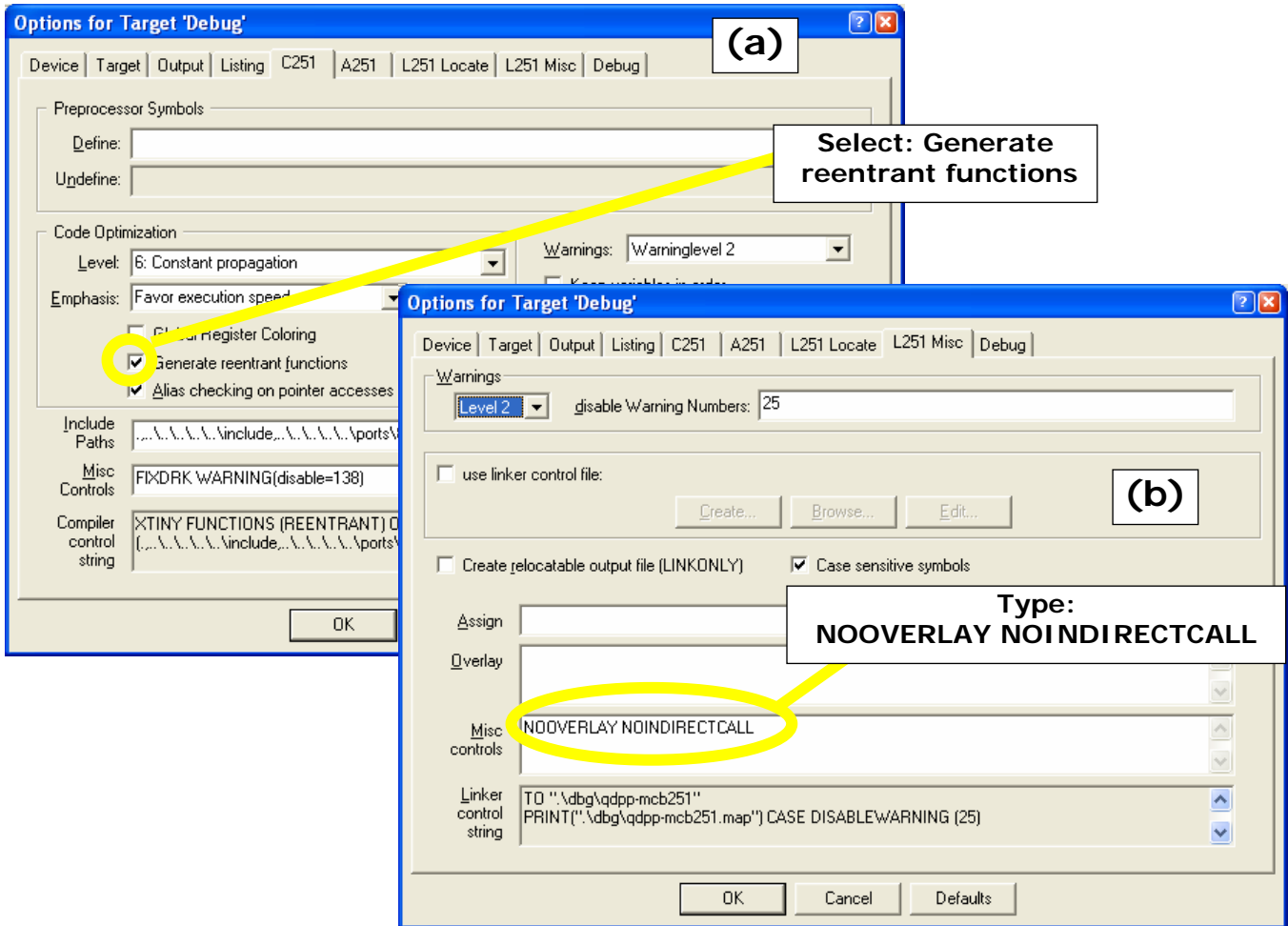


Figure 3 C251 (a) and L251 Misc (b) tabs of the project options dialog box.

## 2.4 Running the Example

Loading and executing the examples on the 80251 is performed by the ROM monitor running on the target device and communicating with the Keil  $\mu$ Vision debugger via the external UART at 57600 baud rate.

The ROM monitor downloads the code to the static RAM of the MCB251 board, which is mapped in the code address space of the target 80251 device (with the standard DIP-switch configuration). You download the code to the target and start debugging it by selecting Debug->Start Debug Session menu (Ctrl+F5 shortcut, or the debug button on the toolbar). Figure 4 shows debugger view of the DPP example application.

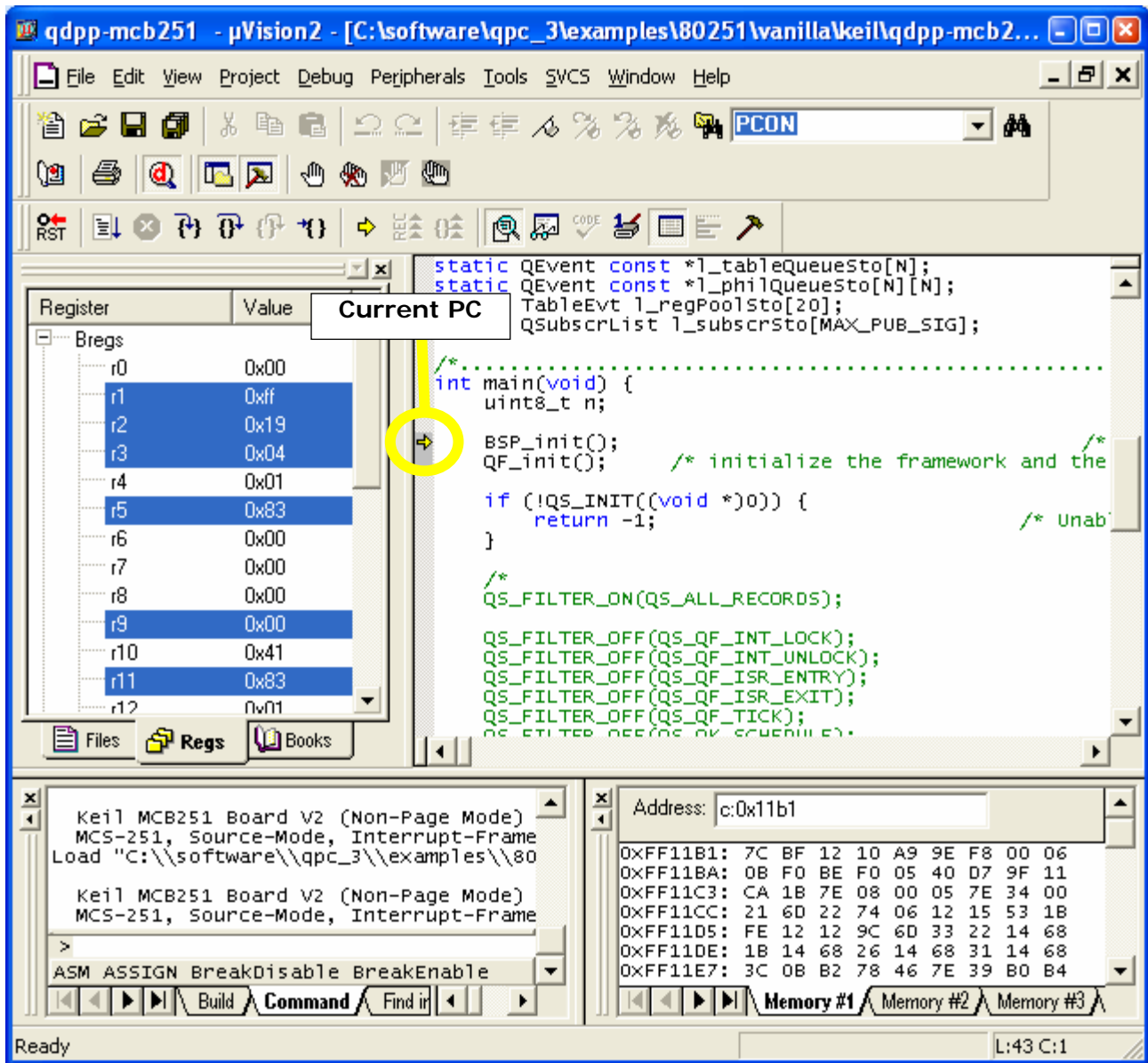


Figure 4 Debugging the DPP example

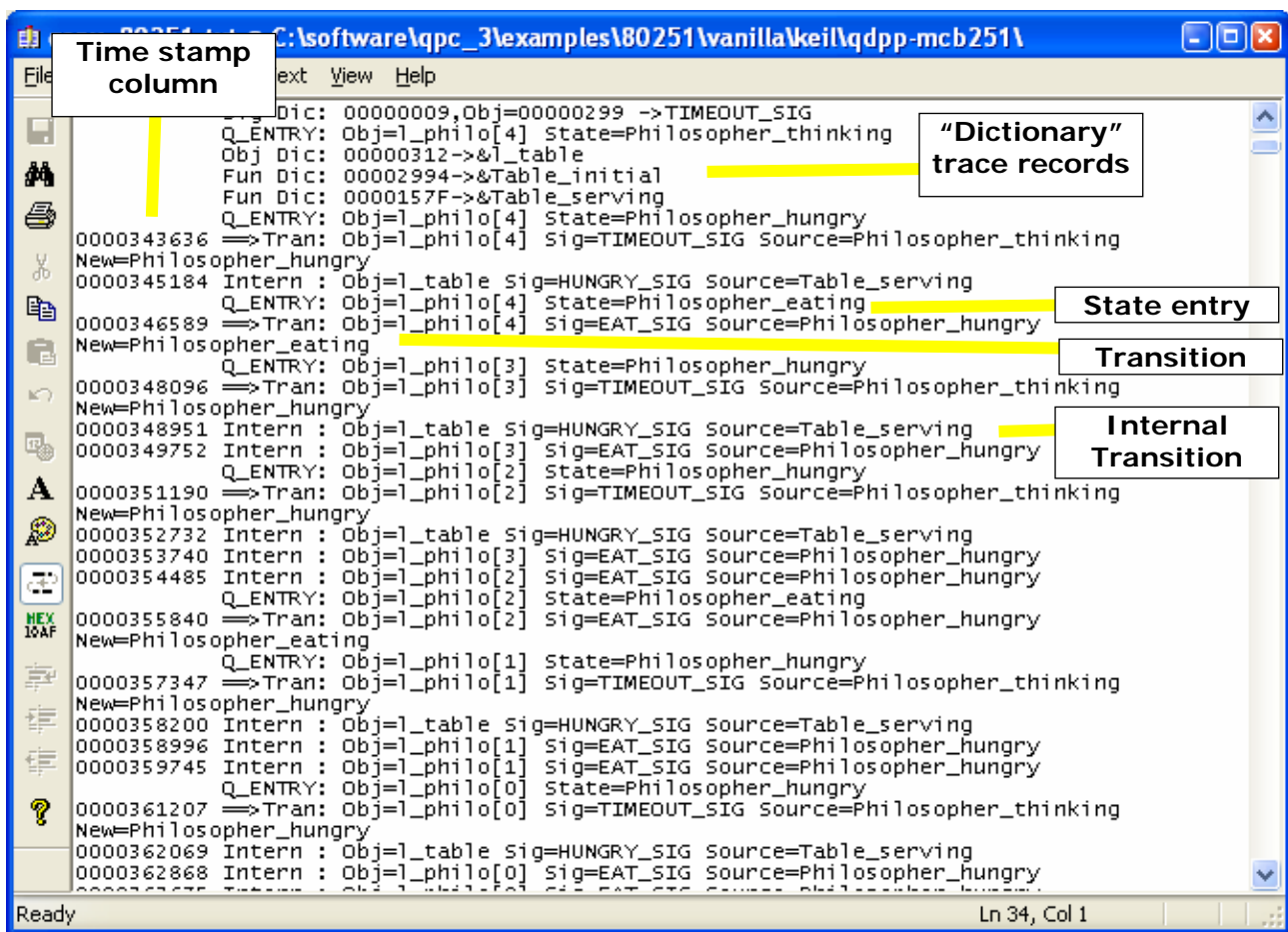
## 2.4.1 Collecting the QSpy trace

This QDK demonstrates how to use the Quantum Spy (QS) to obtain real-time trace of a running QP application.

Quantum Spy (QS) is a software tracing facility built into all Quantum Platform 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 [“QS Programmer's Manual”](#) [QL 05d]).

The DPP-MCB251 project contains the Spy configuration that enables the QSpy software tracing and links with the QSpy-versions of the QP libraries (see Listing 1). You select the QSpy configuration through the drop-down box in the Keil µVision IDE (see Figure 2).

Before you download and start the DPP-Spy configuration to the MCB251 board, you should connect the board's built-in DB9 connector to your workstation via a straight serial cable (see Figure 1). You can connect the QSpy 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).



```

Dic: 00000009,Obj=00000299 ->TIMEOUT_SIG
Q_ENTRY: Obj=1_philo[4] State=Philosopher_thinking
Obj Dic: 00000312->&1_table
Fun Dic: 00002994->&Table_initial
Fun Dic: 0000157F->&Table_serving
Q_ENTRY: Obj=1_philo[4] State=Philosopher_hungry
=>Tran: Obj=1_philo[4] Sig=TIMEOUT_SIG Source=Philosopher_thinking
New=Philosopher_hungry
0000343636 Intern : Obj=1_table Sig=HUNGRY_SIG Source=Table_serving
0000345184 Q_ENTRY: Obj=1_philo[4] State=Philosopher_eating
0000346589 =>Tran: Obj=1_philo[4] Sig=EAT_SIG source=Philosopher_hungry
New=Philosopher_eating
Q_ENTRY: Obj=1_philo[3] State=Philosopher_hungry
0000348096 =>Tran: Obj=1_philo[3] Sig=TIMEOUT_SIG Source=Philosopher_thinking
New=Philosopher_hungry
0000348951 Intern : Obj=1_table Sig=HUNGRY_SIG Source=Table_serving
0000349752 Intern : Obj=1_philo[3] Sig=EAT_SIG source=Philosopher_hungry
Q_ENTRY: Obj=1_philo[2] State=Philosopher_hungry
0000351190 =>Tran: Obj=1_philo[2] Sig=TIMEOUT_SIG Source=Philosopher_thinking
New=Philosopher_hungry
0000352732 Intern : Obj=1_table Sig=HUNGRY_SIG Source=Table_serving
0000353740 Intern : Obj=1_philo[3] Sig=EAT_SIG source=Philosopher_hungry
0000354485 Intern : Obj=1_philo[2] Sig=EAT_SIG source=Philosopher_hungry
Q_ENTRY: Obj=1_philo[2] State=Philosopher_eating
0000355840 =>Tran: Obj=1_philo[2] Sig=EAT_SIG source=Philosopher_hungry
New=Philosopher_eating
Q_ENTRY: Obj=1_philo[1] State=Philosopher_hungry
0000357347 =>Tran: Obj=1_philo[1] Sig=TIMEOUT_SIG Source=Philosopher_thinking
New=Philosopher_hungry
0000358200 Intern : Obj=1_table Sig=HUNGRY_SIG Source=Table_serving
0000358996 Intern : Obj=1_philo[1] Sig=EAT_SIG source=Philosopher_hungry
0000359745 Intern : Obj=1_philo[1] Sig=EAT_SIG source=Philosopher_hungry
Q_ENTRY: Obj=1_philo[0] State=Philosopher_hungry
0000361207 =>Tran: Obj=1_philo[0] Sig=TIMEOUT_SIG Source=Philosopher_thinking
New=Philosopher_hungry
0000362069 Intern : Obj=1_table Sig=HUNGRY_SIG Source=Table_serving
0000362868 Intern : Obj=1_philo[0] Sig=EAT_SIG source=Philosopher_hungry

```

**Figure 5 QSpy output from the DPP-QSpy configuration**

You need to start the QSpy host application on the tracing workstation. Here is the command line to launch the QSpy application from a command prompt window:

```
qspy -cCOM1 -b57600 -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)
- b57600 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).

Figure 5 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 (main.c) and the "[QS Programmer's Manual](#)" [QL 05d] for more information about using the QS filters.

## 3 The Vanilla QP Port

---

The “vanilla” port shows how to use Quantum Platform on a “bare metal” 80251-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 80251, the “vanilla” QF port is very similar to the generic “vanilla” port described in the [“QP Programmer’s Manual”](#).

### 3.1 Compiler Options Used

---

The most important C251 compiler options (used both for building the QP libraries and the final application image) are as follows:

```
XTINY  
FUNCTIONS(REENTRANT)  
OPTIMIZE(6, SPEED)  
FIXDRK  
WARNING(disable=138)  
DEBUG  
CODE  
SYMBOLS  
NOPRINT
```

In particular, you might want to adjust the memory model used (currently XTINY). The port should be insensitive to the memory model used, but of course the model must be big enough for your application. (QP itself uses some 4KB of code space and only several bytes of RAM).

You should leave the FUNCTIONS(REENTRANT) as is, because QP makes intense use of pointer-to-functions, which are difficult for meaningful overlay analysis performed by the compiler.

### 3.2 Linker Options Used

---

The linker options used in the Keil IDE are listed below. Perhaps the most important are the options NOOVERLAY and NOINDIRECTCALL.

```
NOOVERLAY  
RESERVE (0xFF0003-0xFF0005, 0xFF003B-0xFF003D, 0xFF007B-0xFF007D)  
CLASSES (EDATA (0x0-0x41F),  
HDATA (0x0-0x41F),  
HCONST (0xFF0000-0xFF7FFF),  
CONST (0xFF0000-0xFF7FFF),  
ECODE (0xFF0000-0xFF7FFF),  
CODE (0xFF0000-0xFF7FFF))
```

The **NOOVERLAY** directive disables overlay analysis and data overlaying. When this directive is used, the linker does not overlay memory used by local variables and function arguments. This directive is consistent with the FUNCTIONS(REENTRANT) compiler option.

### 3.3 The qep\_port.h Header File

The QEP header file for the 80251 port with the Keil compiler is located in <qp>\ports\80251\vani11a\keil\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 80x51 is a Harvard architecture, 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 Cx51 compiler you need to define the Q\_ROM macro as “code” in the qep\_port.h header file, as shown below.

```
#ifndef qep_port_h
#define qep_port_h

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

#define QP_SIGNAL_SIZE 1

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

#endif                    /* qep_port.h */
```

**Listing 2 The qep\_port.h header file**

### 3.4 The qf\_port.h Header File

The QF header file for the 80251 port with the Keil compiler is located in <qp>\ports\80251\vani11a\keil\qf\_port.h. The following Listing 3 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_SIZE_SIZE    1
(3) #define QF_EQUEUE_CTR_SIZE    1
(4) #define QF_MPOOL_SIZE_SIZE    1
(5) #define QF_MPOOL_CTR_SIZE     1
(6) #define QF_TIMEEVT_CTR_SIZE   2

    /* interrupt locking policy for 80x51, see NOTE01 */
(7) /* QF_INT_KEY_TYPE not defined, simple policy used, see NOTE01 */
(8) #define QF_INT_LOCK(key_)      (EA = 0)
(9) #define QF_INT_UNLOCK(key_)    (EA = 1)

#include "qep_port.h"          /* QEP port */
#include "qvani11a.h"         /* "Vanilla" cooperative kernel */
#include "qf.h"               /* QF platform-independent public interface */

(10) sbit EA = 0xA8 ^ 7;      /* global interrupt enable bit for 80x51 */

    /* shadow of the PCON register for atomic transition to Idle mode, NOTE02 */
(11) extern volatile uint8_t bdata QF_pcon;
```

**Listing 3 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 3(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 manage 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 80x51 microcontrollers provide interrupt prioritization in hardware (within 2-4 levels). Also, the interrupt entry does not include locking interrupts (the general interrupt enable bit 7 of register A8H is not cleared by the hardware upon interrupt entry). For these two reasons this QF port uses the simple interrupt locking policy of unconditional interrupt locking upon the entry to a critical section and unconditional interrupt unlocking upon the exit (see the "[QP Programmer's Manual](#)").

Listing 3(7) The interrupt lock key is not defined, which means that the interrupt status is not saved and restored. Just the simple policy of unconditional interrupt locking and unlocking is used (see Chapter 7 of [PSiCC2]).

- (8) The interrupt locking macro `QF_INT_LOCK()` clears the EA bit (bit 7 in register A8H).
- (9) The interrupt unlocking `QF_INT_UNLOCK()` sets the EA bit (bit 7 in register A8H).
- (10) The EA bit is defined as  $0xA8 \wedge 7$ , which is universal for all 80x51 machines.

### 3.4.3 The PCON Register Shadow for Thread-Safe Transition to Idle Mode

In the non-preemptive QF scheduler, just like in any foreground/background system, the low-power mode must be entered with interrupts locked to avoid an interrupt to preempt the transition to the low-power mode. However, the 80x51 architecture does not provide an **atomic** transition to the Idle mode with simultaneous unlocking interrupts. Instead, here the PCON register (which contains the IDL/PD bits controlling the Idle and Power-Down modes) is shadowed in the `QF_pcon` variable. The variable is allocated in `bdata`, to enforce the single-byte addressing. The Idle mode is set first in the shadow register `QF_pcon`. At the end of every interrupt, the ISR copies the current PCON value into the `QF_pcon` shadow. Please note that the 80x51 hardware clears the IDL and PD bits in the PCON register upon interrupt entry). Later in `QF_onIdle()` (see upcoming Section 3.7), the PCON register is restored from the shadow `QF_pcon`, which the 80x51 performs **atomically** as a single machine instruction (such as `MOV 87H, 20H`). If the interrupt gets serviced between unlocking interrupts and restoring PCON from the shadow, the IDL/PD bits will be cleared in the shadow, so the

machine will **not** go to the Idle/PD mode. Only if the IDL/PD bits survive in the shadow, the Idle/PD mode is entered.

## 3.5 Startup Code

The Keil Cx51 compiler provides a standard startup code for 80251. The file START251.A51 has been copied to the application directory (<qp>\examples\80251\vanilla\keil\dpp-mcb251\, so it can be customized for your application. Perhaps the most important parameter you should adjust is the stack size, as shown in the following snippet from the START251.A51 file:

```
.....  
-----  
: CPU Stack Size Definition  
:  
: The following EQU statement defines the stack space available for the  
: 251 application program. It should be noted that the stack space must  
: be adjusted according the actual requirements of the application.  
:  
STACKSIZE EQU    300    ; adjust to your application  
:  
.....
```

## 3.6 BSP for 80251

The Board Support Package (BSP) for 80251 is very simple. However, there are some important details that you need to pay attention to. The BSP is minimal, but generic for most 80x51 devices.

### 3.6.1 BSP Header file bsp.h

```
(1) #include <reg251s.h>                                /* SFR declarations for 80251S */  
(2) #define BSP_CPU_HZ          11059000  
(3) #define BSP_TICKS_PER_SEC   20  
(4) #define BSP_PERIPHERAL_HZ   (11059000/12)  
  
/* LEDs of the MCB251 board */  
sbit LED0 = P1 ^ 0;  
sbit LED1 = P1 ^ 1;  
sbit LED2 = P1 ^ 2;  
sbit LED3 = P1 ^ 3;  
sbit LED4 = P1 ^ 4;  
sbit LED5 = P1 ^ 5;  
sbit LED6 = P1 ^ 6;  
  
void BSP_init(void);
```

#### Listing 4 BSP header file bsp.h

- (1) The BSP includes the SFR register declarations for the device family in use (here 80251s)
- (2) The BSP defines the CPU frequency used (in Hz)
- (3) The BSP defines the desired system clock-tick rate (in Hz). Please note that you cannot go with this frequency below about 15Hz at CPU speed of 12MHz.
- (4) The BSP defines the clock for the peripherals, which traditionally in 80x51 devices is the CPU clock divided by 12.

### 3.6.2 Board Initialization and the System Timer Tick

The BSP uses Timer2 for generating the system time-tick interrupt. This timer offers the longest (16-bit) overflow values, which can deliver ticking rates as slow as 15 Hz for typical CPU clock rates around 12MHz. The clock-tick rate is defined in bsp.h as the macro BSP\_TICKS\_PER\_SEC.

```

sfr16 TMR2RL = 0xCA;          /* Timer2 reload value (16-bit register pair) */
sfr16 TMR2   = 0xCC;          /* Timer2 counter (16-bit register pair) */

(1) void BSP_init(void) {
(2)     P1 = 0x80;              /* extinguish all LEDs */

    /* setup the Timer2 overflow rate and the upcounting mode */
(3)     TR2 = 0;                /* stop Timer2 */
(4)     T2MOD = 0;              /* set Timer2 to auto-reload, up-counter */
    /* setup the reload registers for the desired tick rate... */
(5)     TMR2RL = (uint16_t)(0x10000 - BSP_PERIPHERAL_HZ/BSP_TICKS_PER_SEC + 0.5);
(6)     TMR2   = TMR2RL;
}

```

**Listing 5 BSP initialization**

- (1) The BSP\_init() function is called from main() to initialize the board
- (2) All bits of Port1, except bit 7 are cleared to extinguish the user LEDs on the MCB251
- (3) The Timer2 is stopped
- (4) The Timer2 mode is set to default (auto-reload, up-counter)
- (5) The Timer2 reload registers are loaded to generate overflow at the desired clock-tick rate BSP\_TICKS\_PER\_SEC defined in bsp.h
- (6) The Timer2 counter is initialized from the reload registers to start the first tick after the programmed period.

### 3.6.3 Starting Interrupts in QF\_onStartup()

QP invokes the QF\_onStartup() callback just before starting the event loop inside QF\_run(). The QF\_onStartup() function is located in the file isr.c and must 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 flag EA, which is cleared out of reset and up to this point hasn't been set.

```

void QF_onStartup(void) {
    TR2 = 1;                      /* start Timer2 */
    ET2 = 1;                      /* enable Timer2 interrupt */
    EA = 1;                       /* enable global interrupt flag */
}

```

**Listing 6 QF\_onStartup() function**

### 3.6.4 ISRs

The Keil C251 compiler supports writing interrupts in C. In the “vanilla” port, the ISRs are essentially identical as in the simplest of all “super-loop” (main+ISRs). However, to support a thread-safe Idle mode transition, every ISR must update the QF\_pcon shadow register from the PCON regis-

ter. Also the time-tick ISR must invoke the QF\_tick() function to perform the system clock-tick processing for QF.

```

(1) void timer2_ISR(void) interrupt 5 {          /* interrupt vector at FF:002BH */
(2)     TF2 = 0;          /* clear Timer2 overflow flag (must be cleared by software) */

    #ifdef Q_SPY
(3)     QS_tickTime += (QSTimeCtr)(BSP_PERIPHERAL_HZ/BSP_TICKS_PER_SEC + 0.5);
    #endif

(4)     QF_tick();          /* QF processing of the system clock-tick */
(5)     QF_pcon = PCON;    /* prevent low-power mode upon ISR return, see NOTE01 */
}
  
```

**Listing 7 Time Timer2 overflow interrupt calling QF\_tick() function.**

- (1) The definition of the interrupt function must use the interrupt keyword and must additionally specify the interrupt vector for this function (vector 5 for Timer2 overflow). The following table lists the lowest standard interrupt numbers supported by the Keil Cx51 compiler.

Interrupt Number	Description	Interrupt Address
0	XTERNAL 0	FF: 0003h
1	TIMER/COUNTER 0	FF: 000Bh
2	EXTERNAL 1	FF: 0013h
3	TIMER/COUNTER 1	FF: 001Bh
4	SERIAL PORT	FF: 0023h
5	TIMER/COUNTER 2	FF: 002Bh
6	PROG. COUNTER ARRAY (PCA)	FF: 0033h

**NOTE:** This non-preemptive QF port works also with ISRs that use different register banks of the 80x51, which the Keil Cx51 compiler allows you to specify by means of the “using” extended keyword. The “using” ISRs are slightly more compact and faster, because they push and pop less registers on the stack.

- (2) The Timer2 overflow flag requires clearing in software.
- (3) If the QS tracing is enabled, the clock-tick ISR updates the running 32-bit time stamp
- (4) The time-tick ISR must invoke QF\_tick(), and can also perform other actions, 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 the 80x51 hardware does not allow the same interrupt preempt itself.
- (5) The QF\_pcon shadow register is updated to the current value of the PCON register. Please note that the IDL/PD bits in the PCON register are automatically cleared in hardware upon every ISR entry, so here these bits are guaranteed to be cleared in the QF\_pcon shadow register. This, in turn, means that the Idle (or Power Down) mode is not entered, if the interrupt preempted the idle processing (see the next section).

### 3.7 QP Idle Processing Customization in QF\_onIdle()

The cooperative “vanilla” kernel can very easily detect the situation when no events are available, in which case QF\_run() calls the QF\_onIdle() callback. You can use QF\_onIdle() to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that QF\_onIdle() is called repetitively from the event loop whenever the event loop has no more events to process, in which case only an interrupt can provide new events. The QF\_onIdle() callback is called with interrupts **locked**, because the determination of the idle condition might change by any interrupt posting an event.

80x51 architecture supports two power-saving levels (Idle and Power Down). These modes are activated by setting the IDL or PD bits in the PCON register (address 0x87). Writing to the IDL or PD bits stops the CPU immediately, and it must happen with interrupts enabled. This means that it is impossible to transition to the Idle (or Power Down) mode atomically. Any enabled interrupt can preempt the idle processing after the interrupts are enabled, but before the Idle mode is actually entered.

To avoid any non-deterministic behavior such situation can cause, this QP port uses a technique to disable the Idle mode transition in every interrupt. So, if an interrupt actually preempts the idle processing in the time window described above, the Idle mode transition won't occur after the interrupt returns. To make this happen, the transition to the Idle mode must be a single machine instruction. (In particular the transition cannot be a non-atomic test-and-set operation.)

To implement such a mechanism, this QP port maintains a shadow of the PCON register (in the QF\_pcon variable). The Idle mode bit is explicitly set only in the shadow register QF\_pcon, still with interrupts disabled. Then interrupts get enabled and the register PCON is restored from the shadow. It is very important that the update of the PCON register happens as a single machine instruction. As it turns out, the Cx51 compiler translates the simple assignment PCON = QF\_pcon; into a single machine instruction such as MOV 87H, 20H.

The following piece of code shows the QF\_onIdle() callback that puts 80x51 into the idle power-saving mode. The code includes the QS output, which will be discussed in the next section.

```

(1) void QF_onIdle(void) {                               /* entered with ints. LOCKED */
(2)     LED6 = 1;                                       /* toggle LED6 on and off, see NOTE01 */
        LED6 = 0;

        #ifdef Q_SPY
        . . .
        #else
(3)     /* stop as many peripheral clocks as possible for you application... */
        QF_pcon = PCON | 0x01; /* set the IDL bit in the PCON shadow, NOTE03 */

(4)     QF_INTERRUPT_UNLOCK(intcon);                   /* unlock the interrupts */
(5)     PCON = QF_pcon;                               /* go to low-power, see NOTE03 */
        #endif

```

#### Listing 8 QF\_onIdle() for the “vanilla” 80x51 port (the QS output is discussed later)

- (1) The QF\_onIdle() function is called with interrupts locked and must unlock interrupts internally.
- (2) This QP port uses the LED6 of the MCB251 to visualize the idle loop activity. The LED is rapidly toggled on and off as long as the idle condition is maintained, so the brightness of the LED is proportional to the CPU idle time (the wasted cycles).
- (3) The IDL bit is turned on in the shadow register QF\_pcon. This happens still in the critical section.
- (4) The interrupts are unlocked.

- (5) The PCON register is updated from the shadow. This happens in just one atomic instruction (MOV 87H, 20H). If an interrupt occurs even one machine instruction before this instruction gets executed, the shadow register QF\_pcon will have the IDL/PD bits cleared and the PCON update will not cause the transition to the idle mode.

### 3.8 Assertion Handling Policy in Q\_onAssert()

As described in Chapter 6 of [PSiCC2], QF uses internally assertions to detect errors in the way application is using the QF 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 80x51. The function simply disables individually all active interrupts to preserve as much system state as possible and enters a for-ever loop. Also, the assertion handler lights up a specific LED pattern. This policy is only adequate for testing, but probably is not adequate for production release.

```
void Q_onAssert(char const Q_ROM *file, int line) {
    (void)file;          /* avoid compiler warning */
    (void)line;         /* avoid compiler warning */

    /* stop interrupts to preserve the system's state as much as possible.. */
    /* NOTE: global interrupt lock blocks the ROM-monitor, so don't do this */
    /* QF_INT_LOCK(ignore); */
    TR2 = 0;            /* stop Timer2 (system clock tick) */
    /* ... stop other interrupts in your system individually */

    LED0 = 1;
    LED1 = 1;
    LED2 = 1;
    LED3 = 1;
    LED4 = 1;
    for (;;) {         /* NOTE: replace the loop with reset for final version */
    }
}
```

**NOTE:** The assert handler refrains from globally locking the interrupts, because this apparently locks up the ROM monitor and hinders inspection of the system with the debugger.

## 4 The QS Software Tracing Instrumentation

This section describes how the QS (Q-Spy) can be used with 80x51 devices to obtain real-time trace of a running QP application.

**NOTE:** You can build the DPP example application in the QSpy configuration even without installing the separate QS components, because the QS library pre-compiled for 80251 is provided. However, to obtain the QS source code, you need to install the Quantum Spy component, which is available separately from the rest of QP and is available only under the terms of commercial licensing.

### 4.1 QS Time Stamp Callback `QS_onGetTime()`

The platform-specific QS port must provide function `QS_onGetTime()` that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the 80x51 port uses Timer2, which is the same timer already used for generation of the system clock-tick interrupt.

Timer2 is used in Mode1 (auto reload, up-counter). In this mode Timer2 uses 16-bit up-counter, organized as two cascaded 8-bit registers TL2 and TH2. This hardware timer provides time with resolution of the peripheral clock on 80x51, which is CPU frequency divided by 12. For a typical CPU clock around 12 MHz, the resolution of the time stamp is 1 microsecond.

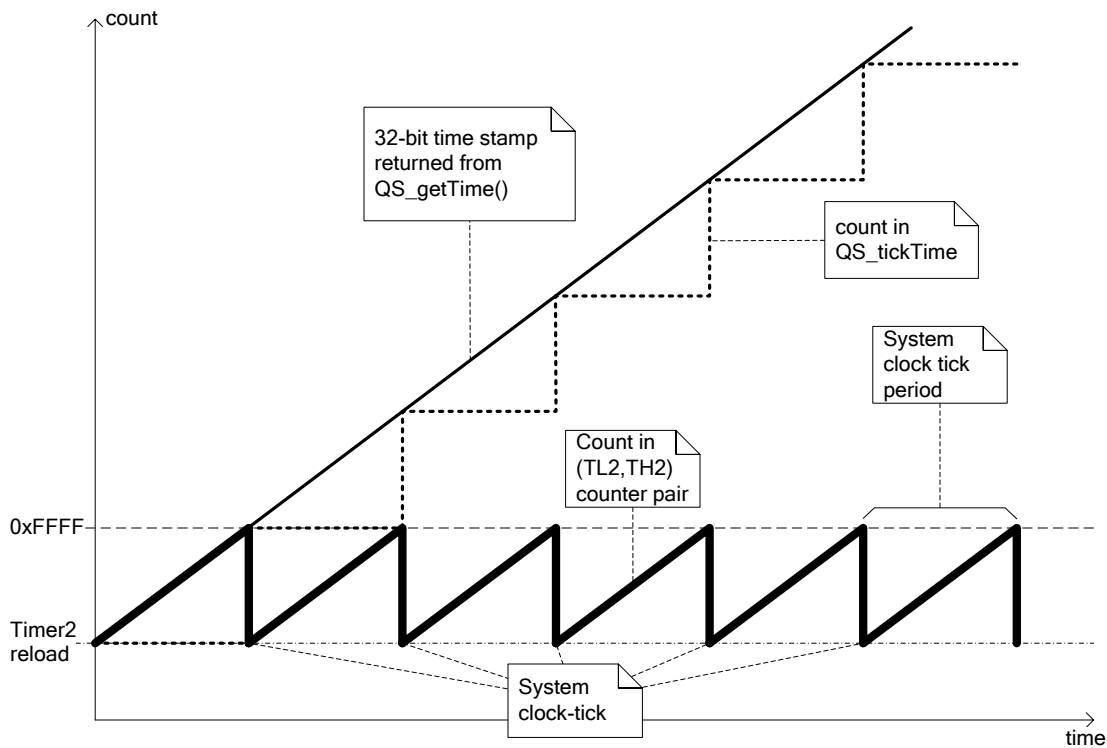


Figure 6 Using the Timer2 to provide 32-bit QS time stamp.

Figure 6 shows how the 16-bit Timer2 is extended to 32 bits. Timer2 counts up from the reload value stored in the RCAP2L, RCAP2H register pair in Listing 7(3). When TL2, TH2 counter pair saturates at 0xFFFF, the hardware automatically reloads the TL2, TH2 counters from the RCAP2L, RCAP2H register pair on the subsequent peripheral clock tick. Simultaneously, the hardware sets the TF2 flag, which “remembers” that the overflow occurred.

The system clock tick ISR keeps updating the “tick count” variable `QS_tickCount` by incrementing it each time by `0x10000 - Timer2-reload-value` (again see Figure 6). The clock-tick ISR also clears the TF2 flag.

Listing 9 shows the implementation of the function `QS_onGetTime()`, which combines all this information to produce a monotonic time stamp.

```

OSTimeCtr QS_onGetTime(void) { /* invoked with interrupts locked */
(1)  uint8_t volatile t2h1 = TH2; /* step 1 */
(2)  uint8_t volatile t2l  = TL2; /* step 2 */
(3)  uint8_t volatile t2h2 = TH2; /* step 3 */
(4)  OSTimeCtr tickTime = QS_tickTime;

(5)  if (TF2 != 0) { /* Did Timer2 auto-reload but tick ISR didn't run yet? */
(6)    tickTime += (OSTimeCtr)(BSP_PERIPHERAL_HZ/BSP_TICKS_PER_SEC + 0.5);
  }

(7)  if (t2h1 == t2h2) { /* the high-register unchanged from step 1 and 3? */
(8)    return tickTime
      + (OSTimeCtr)((uint16_t)t2l + ((uint16_t)t2h1 << 8));
  }
  else { /* the high-register changed between step 1 and 3 */
(9)    return tickTime + ((uint16_t)t2h2 << 8);
  }
}

```

### Listing 9 QS\_getTime() implementation

- (1) The first challenge in the 80x51 is that the hardware provides no way of atomically latching the 16-bit (TL2, TH2) value. The software reads first the TH2 counter.
- (2) The TL2 is read next.
- (3) The TH2 is read again.
- (4) The “tick-time” is stored in the local variable.
- (5) The TF2 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).
- (6) If the auto-reload occurred, the tick-time must be incremented by the tick period.
- (7) If the two TH2 readouts compare equal (which is the most frequent case), there was no rollover from TL2 to TH2 from the first to the third readout, so the (t2l, t2h1) register pair is consistent.
- (8) The function returns the “fine time” from the (t2l, t2h1) counter pair added to the “coarse” tick-time.
- (9) Otherwise, the rollover happened, so the pair either the pair (t2l, t2h1) or (t2l, t2h2) is inconsistent. The function does not use the t2l measurement and assumes that the rollover just happened, so TH2 is zero. The function returns the “fine time” (0, t2h2) added to the “coarse” tick-time.

## 4.2 QS Platform-Specific Implementation in bsp.c

The following Listing 10 shows the full implementation of the QS platform-specific code from bsp.c.

```

(1) #ifndef Q_SPY
(2) #if (BSP_CPU_HZ == 12000000)
        /* best match for the standard PC serial port baud rate */
        #define QS_BAUD_RATE      4800
    #elif (BSP_CPU_HZ == 11059000)
        /* best match for the standard PC serial port baud rate */
(4)     #define QS_BAUD_RATE      57600
    #else
(5)     #error "Please choose the best QSpy baud rate for your custom clock"
    #endif

(6) #define QS_BUF_SIZE      (350)

(7) OSTimeCtr QS_tickTime;

/*.....*/
(8) uint8_t QS_onStartup(void const *arg) {
(9)     static uint8_t qsBuf[QS_BUF_SIZE];          /* buffer for Quantum Spy */

        (void)arg;                                /* avoid the compiler warning */
(10)    QS_initBuf(qsBuf, sizeof(qsBuf));

        /* setup Timer1-based baud rate generator */
(11)    TR1 = 0;                                    /* stop Timer1 */
(12)    ET1 = 0;                                    /* disable Timer1 interrupt */
(13)    PCON |= 0x80;                                /* 0x80 = SMOD: set serial baud rate doubler */
        TMOD &= ~0xF0;                            /* clear Timer1 mode bits */
(14)    TMOD |= 0x20;                                /* put Timer1 into MODE 2 */
        /* set the Timer1 reload value to generate the desired baud rate */
(15)    TH1 = (uint8_t)(0x100 - 2.0*BSP_PERIPHERAL_HZ/32.0/QS_BAUD_RATE + 0.5);
        TL1 = TH1;

        /* setup serial port registers */
(16)    P3 |= (1 << 1); /* configure P3.1 as controlled by alternative function */
(17)    SCON = 0x40;                                /* serial port MODE 1 */
(18)    REN = 0;                                    /* disable serial receiver */
(19)    TR1 = 1;                                    /* start Timer1 */
(20)    TI = 1;                                    /* enable transmitting */

(21)    return (uint8_t)1;                            /* return success */
}
/*.....*/
(22) void QS_onCleanup(void) {
}
/*.....*/
(23) OSTimeCtr QS_onGetTime(void) { /* invoked with interrupts locked */
    uint8_t volatile t2h1 = TH2;                /* step 1 */
    uint8_t volatile t2l  = TL2;                /* step 2 */
    uint8_t volatile t2h2 = TH2;                /* step 3 */
    OSTimeCtr tickTime = QS_tickTime;

    if (TF2 != 0) { /* Timer2 auto-reloaded but tick ISR didn't run yet? */
        tickTime += (OSTimeCtr)(BSP_PERIPHERAL_HZ/BSP_TCKS_PER_SEC + 0.5);
    }

    if (t2h1 == t2h2) { /* the high-register unchanged from step 1 and 3? */
        return tickTime
            + (OSTimeCtr)((uint16_t)t2l + ((uint16_t)t2h1 << 8));
    }
}

```

```

        else { /* the high-register changed between step 1 and 3 */
            return tickTime + ((uint16_t)t2h2 << 8);
        }
    }
    /*.....*/
(24) void QS_onFlush(void) {
(25)     uint16_t b;
(26)     while ((b = QS_getByte()) != QS_EOD) { /* next QS trace byte available? */
(27)         while (TI == 0) { /* hang in a loop as long as transmit not ready */
(28)             TI = 0; /* clear the transmit-interrupt flag for next time */
(29)             SBUF = (uint8_t)b;
        }
    }
}
#endif /* Q_SPY */

```

### Listing 10 QSpy implementation to send data out of the on-chip UART of the 80x51.

- (1) The QS instrumentation is enabled only when the macro Q\_SPY is defined
- (2-5) The QS instrumentation uses Timer1 for the baud-rate generation. The generic formula for the baud rate is:

$$\text{Baud\_Rate} = 2^{\text{SMOD1}} \text{Timer1\_Overflow\_Rate} / 32 = 2^{\text{SMOD1}} \text{BSP\_CPU\_HZ} / (32 * (0x100 - \text{TH1}))$$

The following table shows the standard PC baud rates that best fit the formula:

BSP_CPU_HZ	0x100 – TH1 (% error)	Baud_Rate
12.000MHz	13.0208 -> 13 (1.5%)	4800
11.059MHz	1.0 -> 1 (0.00%)	57600

For different CPU clock frequency, you should choose the baud rate carefully to avoid transmission errors.

- (6) You need to decide the size of the QS buffer (in bytes). You want to make the buffer use all RAM remaining in your design.
- (7) The QS\_tickTime variable is used for extending the Timer2 count to 32-bits, as discussed in Section 4.1.
- (8) The QS\_onStartup() callback performs the initialization of QS
- (9) The QS trace buffer is statically allocated to the specified size
- (10) You always need to call QS\_initBuf() from QS\_onStartup() to initialize the trace buffer. This particular QS port initializes Timer1 for baud-rate generation and the on-board Serial Port for data transfer at the given baud rate QS\_BAUD\_RATE.
- (11) Timer1 is stopped to configure it for baud-rate generation for the serial output.

**NOTE:** This QDK uses Timer1 for baud-rate generation. The only other option would be to use Timer2, but Timer2 is already used (and much better suited) for the system time-tick and time-stamp generation.

- (12) Timer1 operating as the baud-rate generator must not generate interrupts.
- (13) The baud-rate doubling is enabled in the PCON register.

- (14) Timer1 is configured to Mode2 (8-bit timer with auto-reload).
- (15) The Timer1 reload value is set up to generate the desired baud rate QS\_BAUD\_RATE.
- (16) The Serial port TX pin (P3.1) is configured to alternative function (Serial port)

**NOTE:** This QS instrumentation takes up only a single pin (the TX pin). Please also note that QS does not take any interrupt and performs the output in the least-intrusive way during the idle processing, when the CPU has nothing else to do anyway.

- (17) The Serial port is configured to Mode 1 (full-duplex asynchronous mode).
- (18) The Serial receiver is disabled (consistently with using just a single pin).

**NOTE:** You can use the Serial receiver in your application, in any way you like, completely unrelated to QS tracing.

- (19) The Timer1 is started.
- (20) The Serial port transmission is enabled
- (21) The QS\_onStartup() function returns success.
- (22) The QS\_onCleanup() function is empty in this QS port.
- (23) The QS\_onGetTime() callback function has been discussed in Section 4.1.
- (24) 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 take up several milliseconds to output the whole buffer.
- (25) The QS\_onFlush() callback function uses the byte-oriented interface provided by QS. The QS\_getByte() function returns a 16-bit value.
- (26) The QS\_getByte() function returns QS\_EOD (0xFFFF) when the QS buffer is empty.
- (27) This while() loop waits as long as the Serial transmitter is busy.
- (28) The Serial transmitter flag is cleared to prepare for the next transmission.
- (29) The new byte (the LSB of the value returned from QS\_getByte()) is inserted to the serial buffer register SBUF.

### 4.3 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 3.7. This section explains the details of the QS trace output.

```
void QF_onIdle(void) { /* entered with ints. LOCKED */
(1) #ifndef Q_SPY
(2)     if (TI != 0) { /* ready to transmit? */
(3)         uint16_t b = QS_getByte(); /* get the next QS byte to send */
(4)         QF_INT_UNLOCK(igonre);
(5)         if (b != QS_EOD) { /* End-Of-Data not reached yet? */
(6)             TI = 0; /* clear the transmit-interrupt flag for next time */
(7)             SBUF = (uint8_t)b; /* insert the byte to the transmit register */
```

```
    }  
    }  
    else {  
(8)    QF_INT_UNLOCK(i gonre);           /* still transmitting */  
    }                                       /* either way unlock interrupts */  
    #else  
    }  
    #endif
```

**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 interrupts can be unlocked.
- (5) The return value from QS\_getByte() is checked for End-Of-Data (EOD) condition.
- (6-7) If the data is available, the transmit flag is cleared and data is placed in SBUF.
- (8) Otherwise, the interrupts are just unlocked.

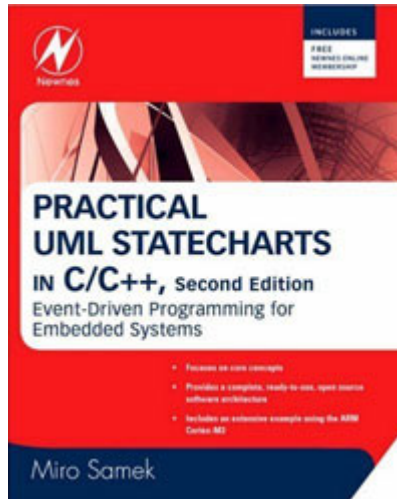
## 5 Related Documents and References

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 <a href="http://amazon.com">amazon.com</a> . See also: <a href="http://www.quantum-leaps.com/writings/psicc2.htm">http://www.quantum-leaps.com/writings/psicc2.htm</a>
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2008	<a href="http://www.quantum-leaps.com/doxygen/qpc/">http://www.quantum-leaps.com/doxygen/qpc/</a>
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	<a href="http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf">http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf</a>
[QL AN-DPP 08] "Application Note: Dining Philosophers Application", Quantum Leaps, LLC, 2008	<a href="http://www.quantum-leaps.com/doc/AN_DPP.pdf">http://www.quantum-leaps.com/doc/AN_DPP.pdf</a>
"MCBx51 Evaluation Board User's Guide", Keil Software, 2000.	The PDF version of this document is included with the Keil compiler, see also <a href="http://www.keil.com/c251/">http://www.keil.com/c251/</a> .
"C251 Compiler Optimizing C Compiler and Library Reference", Keil Software, 1997	The PDF version of this document is included with the Keil compiler, see also <a href="http://www.keil.com/c251/">http://www.keil.com/c251/</a> .
"8XC251SB Embedded Microcontroller User's Manual", Intel, 1995	<a href="http://www.intel.com/design/mcs51/-manuals/272617.htm">www.intel.com/design/mcs51/-manuals/272617.htm</a>

## 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](mailto: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

**Keil - An ARM Company**  
1501 10th Street, Suite 110  
Plano, TX 75074  
USA  
Toll Free: 800-348-8051  
Phone: 972-312-1107  
Fax: 972-312-1159  
Sales: [sales.intl@keil.com](mailto:sales.intl@keil.com)  
Support: [support.intl@keil.com](mailto:support.intl@keil.com)  
WEB : [www.Keil.com](http://www.Keil.com)

**Keil - An ARM Company**  
Bretonischer Ring 15  
D-85630 Grasbrunn  
Germany  
Phone: ++49 089/45 60 40 0  
Fax: ++49 089/46 81 62

