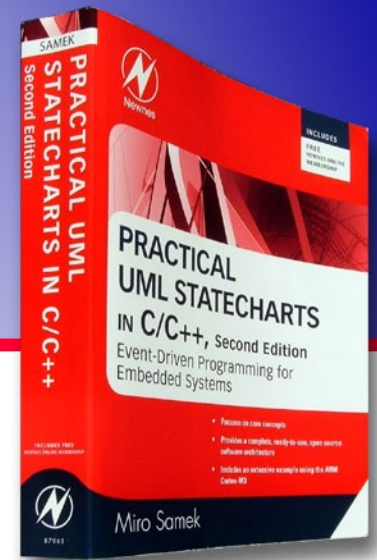




**Quantum<sup>®</sup>Leaps**  
innovating embedded systems



# Application Note

## QP<sup>™</sup> and ARM-Cortex with GNU

Document Revision C  
February 2010



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

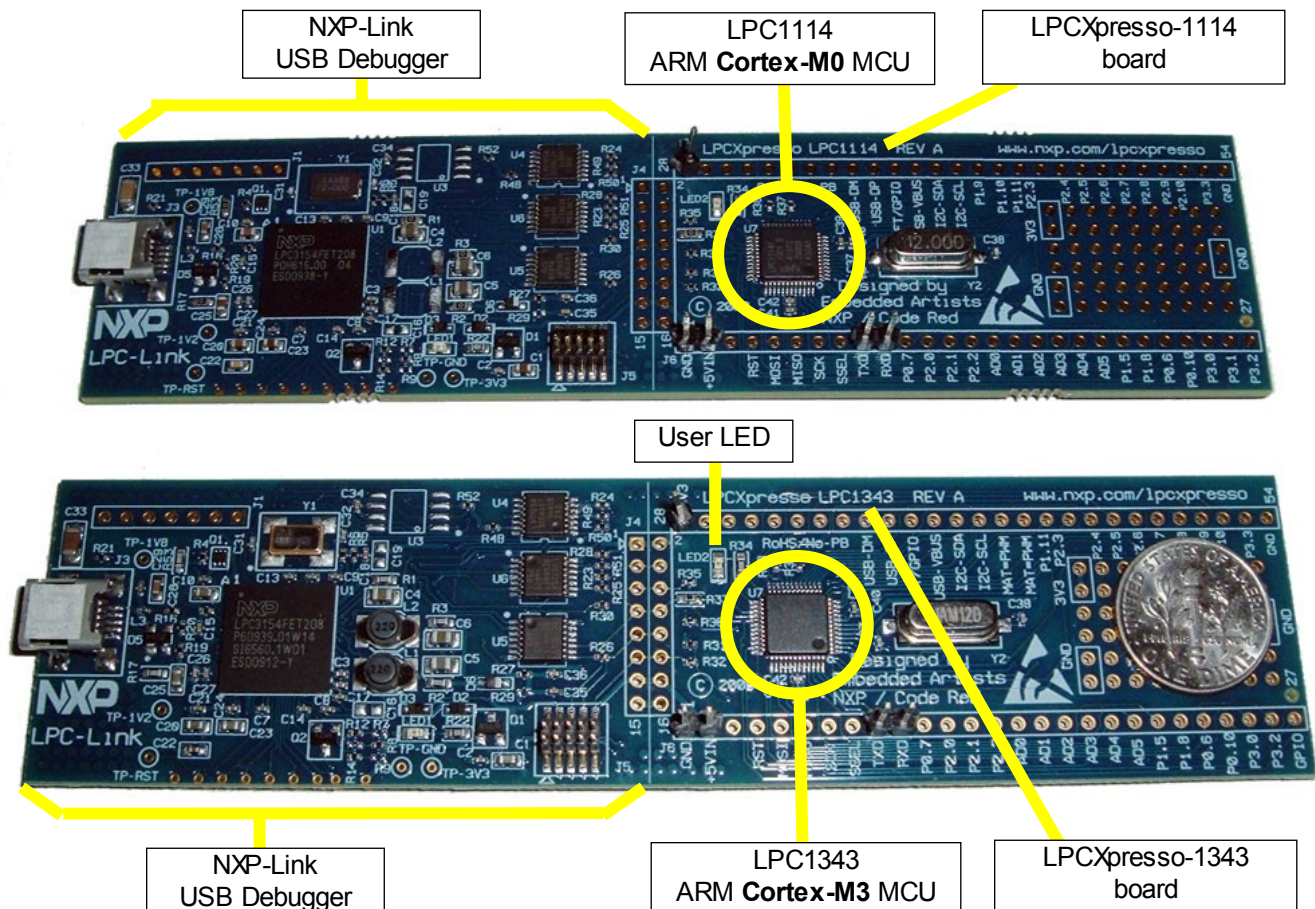
<b>1 Introduction.....</b>	<b>1</b>
1.1 About QP™.....	2
1.2 About the ARM-Cortex Port.....	2
1.3 Cortex Microcontroller Software Interface Standard (CMSIS).....	3
1.4 Licensing QP.....	3
<b>2 Directories and Files.....</b>	<b>4</b>
2.1 Building and Debugging the Examples.....	6
2.1.1 Building the Examples from Command Line.....	6
2.1.2 Building the Examples from Eclipse.....	7
2.2 Downloading to Flash and Debugging the Examples.....	9
2.2.1 Software Tracing with Q-SPY.....	10
2.3 Building the QP Libraries.....	12
<b>3 The Vanilla Port.....</b>	<b>14</b>
3.1 The qep_port.h Header File.....	14
3.2 The QF Port Header File.....	14
3.3 Handling Interrupts in the Non-Preemptive Vanilla Kernel.....	15
3.3.1 The Interrupt Vector Table.....	15
3.3.2 Starting Interrupts in QF_onStartup().....	17
3.4 Idle Loop Customization in the “Vanilla” Port.....	17
<b>4 The QK Port.....</b>	<b>19</b>
4.1 Single-Stack, Preemptive Multitasking on ARM-Cortex.....	19
4.1.1 Examples of Various Preemption Scenarios in QK.....	20
4.1.2 The Stack Utilization in QK.....	21
4.2 The QK Port Header File.....	23
4.2.1 The QK Critical Section.....	24
4.3 QK Platform-Specific Code for ARM-Cortex.....	25
4.4 Setting up and Starting Interrupts in QF_onStartup().....	29
4.5 Writing ISRs for QK.....	29
4.6 QK Idle Processing Customization in QK_onIdle().....	29
4.7 Testing QK Preemption Scenarios.....	31
4.7.1 Interrupt Nesting Test.....	32
4.7.2 Task Preemption Test.....	32
4.7.3 Other Tests.....	33
<b>5 QS Software Tracing Instrumentation.....</b>	<b>34</b>
5.1 QS Time Stamp Callback QS_onGetTime().....	35
5.2 QS Trace Output in QF_onIdle()/QK_onIdle().....	36
5.3 Invoking the QSpy Host Application.....	37
<b>6 Related Documents and References.....</b>	<b>38</b>
<b>7 Contact Information.....</b>	<b>39</b>



## 1 Introduction

This Application Note describes how to use the QP™ state machine frameworks with the ARM-Cortex processors. To focus the discussion, this Application Note uses the GNU-based LPCXpresso toolchain from Code Red Technologies Ltd (LPCXpresso version **3.2.2**) and the LPCXpresso-1114 and LPCXpresso-1343 boards from NXP, as shown in Figure 1. However, the source code for the QP port described here is generic and runs without modifications on any ARM Cortex-M0 and ARM Cortex-M3 cores.

**Figure 1** The LPCXpresso-1114 and LPCXpresso-1343 boards used to test the ARM-Cortex port.

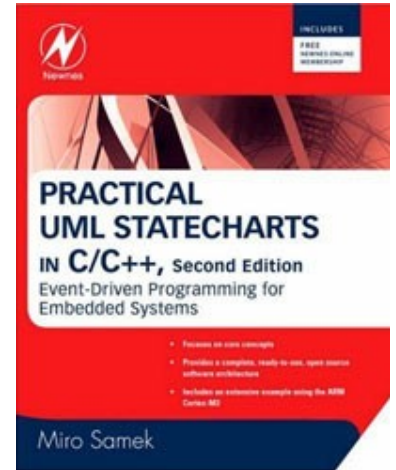




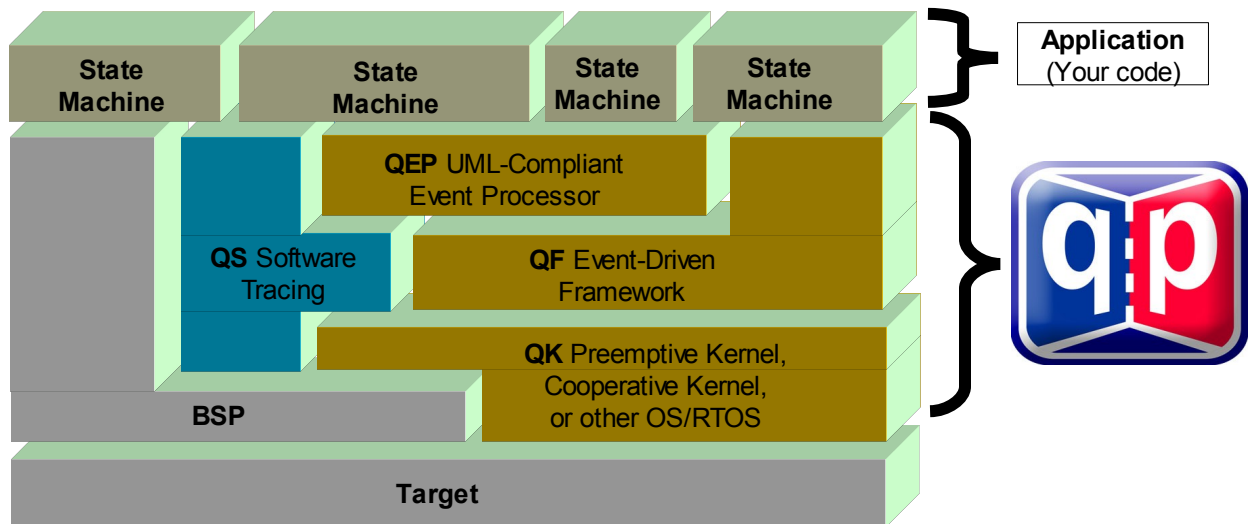
## 1.1 About QP™

QP™ is a family of very lightweight, open source, state machine-based frameworks for developing event-driven applications. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C or C++ **without big tools**. QP is described in great detail in the book “*Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*” [PSiCC2] (Newnes, 2008).

As shown in Figure 2, QP consists of a universal UML-compliant event processor (QEP), a portable real-time framework (QF), a tiny run-to-completion kernel (QK), and software tracing instrumentation (QS). Current versions of QP include: QP/C™ and QP/C++™, which require about 4KB of code and a few hundred bytes of RAM, and the ultra-lightweight QP-nano, which requires only 1-2KB of code and just several bytes of RAM.



**Figure 2 QP components and their relationship with the target hardware, board support package (BSP), and the application**



QP can work with or without a traditional RTOS or OS. In the simplest configuration, QP can completely **replace** a traditional RTOS. QP includes a simple non-preemptive scheduler and a fully preemptive kernel (QK). QK is smaller and faster than most traditional preemptive kernels or RTOS, yet offers fully deterministic, preemptive execution of embedded applications. QP can manage up to 63 concurrently executing tasks structured as state machines (called active objects in UML).

QP/C and QP/C++ can also work with a traditional OS/RTOS to take advantage of existing device drivers, communication stacks, and other middleware. QP has been ported to Linux/BSD, Windows, VxWorks, ThreadX, uC/OS-II, FreeRTOS.org, and other popular OS/RTOS.

## 1.2 About the ARM-Cortex Port

In contrast to the traditional ARM7/ARM9 cores, ARM-Cortex cores contain such standard components as the Nested Vectored Interrupt Controller (NVIC) and the System Timer (SysTick). With the provision of these standard components, it is now possible to provide fully portable system-level software for ARM-

Cortex. Therefore, this QP port to ARM-Cortex can be much more complete than a port to the traditional ARM7/ARM9 and the software is guaranteed to work on any ARM-Cortex silicon.

The non preemptive cooperative kernel implementation is very simple on ARM-Cortex, perhaps simpler than any other processor, mainly because Interrupt Service Routines (ISRs) are regular C-functions on ARM-Cortex.

However, when it comes to handling preemptive multitasking, ARM-Cortex is a unique processor unlike any other. The ARM-Cortex hardware has been designed with traditional blocking real-time kernels in mind, and implementing a simple run-to-completion preemptive kernel (such as the QK preemptive kernel described in Chapter 10 in [PSiCC2]) is a little more involved. Section 4 of this application note describes in detail the QK implementation on ARM-Cortex.

---

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

---

### 1.3 Cortex Microcontroller Software Interface Standard (CMSIS)

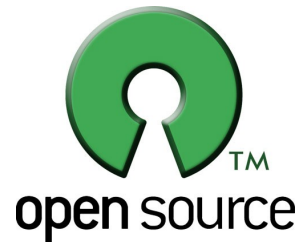
The ARM-Cortex examples provided with this Application Note are compliant with the Cortex Microcontroller Software Interface Standard (CMSIS).



### 1.4 Licensing QP

The **Generally Available (GA)** distributions of QP available for download from the [www.state-machine.com/downloads](http://www.state-machine.com/downloads) website are offered under the same licensing options as the QP baseline code. These available licenses are:

- 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).

## 2 Directories and Files

The code for the QP port to ARM-Cortex is part of the standard QP distribution, which also contains example applications.

The code of the ARM-Cortex port is organized according to the Application Note: "[QP Directory Structure](#)". Specifically, for this port the files are placed in the following directories:

### Listing 1 Directories and files pertaining to the ARM-Cortex QP port included in the standard QP distribution.

```

<qp>/                                - QP-root directory for Quantum Platform (QP) v3.x.yy
|
+-include/                            - QP public include files
| +-qassert.h                        - Quantum Assertions platform-independent public
include
| +-qevent.h                        - QEvent declaration
| +-qep.h                          - QEP platform-independent public include
| +-qf.h                           - QF platform-independent public include
| +-qk.h                           - QK platform-independent public include
| +-qs.h                           - QS platform-independent public include
| +-qs_dummy.h                     - QS "dummy" public include
| +-qequeue.h                      - native QF event queue include
| +-qmpool.h                       - native QF memory pool include
| +-qpset.h                        - native QF priority set include
| +-qvanilla.h                     - native QF non-preemptive "vanilla" kernel include
|
+-ports/                             - QP ports
| +-arm-cortex/                     - ARM-Cortex port
| | +-vanilla/                      - "vanilla" ports
| | | +-gnu/                        - GNU ARM compiler
| | | | +-dbg/                      - Debug build
| | | | | +-libqep_cortex-m0.a      - QEP library for Cortex-M0
| | | | | +-libqf_cortex-m3.a      - QF library for Cortex-M3
| | | | | +-libqep_cortex-m0.a      - QEP library for Cortex-M0
| | | | | +-libqf_cortex-m3.a      - QF library for Cortex-M3
| | | | +-rel/                     - Release build
| | | | +-...                       -
| | | | +-spy/                     - Spy build
| | | | | +-libqep_cortex-m0.a      - QEP library for Cortex-M0
| | | | | +-libqep_cortex-m3.a      - QEP library for Cortex-M3
| | | | | +-libqf_cortex-m0.a      - QF library for Cortex-M0
| | | | | +-libqf_cortex-m3.a      - QF library for Cortex-M3
| | | | | +-libqs_cortex-m0.a      - QS library for Cortex-M0
| | | | | +-libqs_cortex-m3.a      - QS library for Cortex-M3
| | | | +-make_cortex-m0.bat        - Batch file to build QP libraries for Cortex-M0
| | | | +-make_cortex-m3.bat        - Batch file to build QP libraries for Cortex-M3
| | | | +-qep_port.h               - QEP platform-dependent public include
| | | | +-qf_port.h                - QF platform-dependent public include
| | | | +-qs_port.h                - QS platform-dependent public include
| | | | +-qp_port.h                - QP platform-dependent public include
| | +-qk/                          - QK (Quantum Kernel) ports
| | +-gnu/                          - GNU ARM compiler
| | | +-dbg/                        - Debug build
| | | | +-libqep_cortex-m0.a        - QEP library for Cortex-M0

```

```

| | | | | +-libqep_cortex-m3.a - QEP library for Cortex-M3
| | | | | +-libqf_cortex-m0.a - QF library for Cortex-M0
| | | | | +-libqf_cortex-m3.a - QF library for Cortex-M3
| | | | | +-libqk_cortex-m0.a - QK library for Cortex-M0
| | | | | +-libqk_cortex-m3.a - QK library for Cortex-M3
| | | | +-rel/ - Release build
| | | | +-src/ - Platform-specific source directory
| | | | | +-qk_port.s - Platform-specific source code for the QK port
| | | | +-make_cortex-m0.bat - Batch file to build QP libraries for Cortex-M0
| | | | +-make_cortex-m3.bat - Batch file to build QP libraries for Cortex-M3
| | | | +-qep_port.h - QEP platform-dependent public include
| | | | +-qf_port.h - QF platform-dependent public include
| | | | +-qs_port.h - QS platform-dependent public include
| | | | +-qp_port.h - QP platform-dependent public include
|
+-examples/ - subdirectory containing the QP example files
| +-arm-cortex/ - ARM-Cortex port
| | +-vanilla/ - "vanilla" examples (non-preemptive scheduler of QF)
| | | +-gnu/ - GNU ARM compiler
| | | | +-dpp-lpcxpresso-1114/ - Dining Philosophers example for LPCXpresso-1114
| | | | | +-cmsis/ - directory containing the CMSIS files
| | | | | +-lpc11xx_lib/ - directory containing the LPC11xx library (from NXP)
| | | | | +-dbg/ - directory containing the Debug build
| | | | | +-rel/ - directory containing the Release build
| | | | | +-spy/ - directory containing the Spy build
| | | | |
| | | | | +- .cproject - Eclipse project file for the LPCXpresso IDE
| | | | | +- .project - Eclipse project file for the LPCXpresso IDE
| | | | | +- Makefile - external Makefile for the LPCXpresso IDE
| | | | | +-lpc1114.ld - linker command file for LPC1114
| | | | | +-bsp.c - Board Support Package for the DPP application
| | | | | +-bsp.h - BSP header file
| | | | | +-dpp.h - the DPP header file
| | | | | +-main.c - the main function
| | | | | +-philos.c - the Philosopher active object
| | | | | +-table.c - the Table active object
| | | | | +-no_heap.c - dummy heap routines to reduce code size
| | | |
| | | | +-dpp-lpcxpresso-1343/ - Dining Philosophers example for LPCXpresso-1343
| | | | | +-cmsis/ - directory containing the CMSIS files
| | | | | +-lpc13xx_lib/ - directory containing the LPC13xx library (from NXP)
| | | | | +-dbg/ - directory containing the Debug build
| | | | | +-rel/ - directory containing the Release build
| | | | | +-spy/ - directory containing the Spy build
| | | | |
| | | | | +- .cproject - Eclipse project file for the LPCXpresso IDE
| | | | | +- .project - Eclipse project file for the LPCXpresso IDE
| | | | | +- Makefile - external Makefile for the LPCXpresso IDE
| | | | | +-lpc1343.ld - linker command file for LPC1114
| | | | | +-bsp.c - Board Support Package for the DPP application
| | | | | +-bsp.h - BSP header file
| | | | | +-dpp.h - the DPP header file
| | | | | +-main.c - the main function
| | | | | +-philos.c - the Philosopher active object
| | | | | +-table.c - the Table active object
| | | | | +-no_heap.c - dummy heap routines to reduce code size

```

```

| | |
| | |--qk/                - QK examples
| | |--gnu/              - GNU ARM compiler
| | | |--dpp-qk-lpcxpresso-1114/ - DPP example for LPCXpresso-1114
| | | | |=. . .
| | | |--dpp-qk-lpcxpresso-1343/ - DPP example for LPCXpresso-1343
| | | | |=. . .

```

## 2.1 Building and Debugging the Examples

The example applications for ARM-Cortex have been tested with the LPCXpresso evaluation boards from NXP (see Figure 1) and the GNU/Eclipse-based LPCXpresso toolset from Code Red. The examples contain the Eclipse projects for the LPCXpresso IDE as well as the Makefiles, so that you can conveniently build and debug the examples from the LPCXpresso IDE. The provided Makefiles and projects support building the Debug, Release, and Spy configurations.

### 2.1.1 Building the Examples from Command Line

The example directory <qp>\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114\ contains the Makefile you can use to build the application. The Makefile supports three build configurations: Debug (default), Release, and Spy. You choose the build configuration by defining the CONF symbol at the command line, as shown in the table below. Figure 3 shows an example command-line build of the Spy configuration.

**Table 1 Make targets for the Debug, Release, and Spy software configurations**

Build Configuration	Build command
Debug (default)	make
Release	make CONF=rel
Spy	make CONF=spy
Clean the Debug configuration	make clean
Clean the Release configuration	make CONF=rel clean
Clean the Spy configuration	make CONF=spy clean



Figure 3 Building the DPP application with the provided Makefile from command-line



```

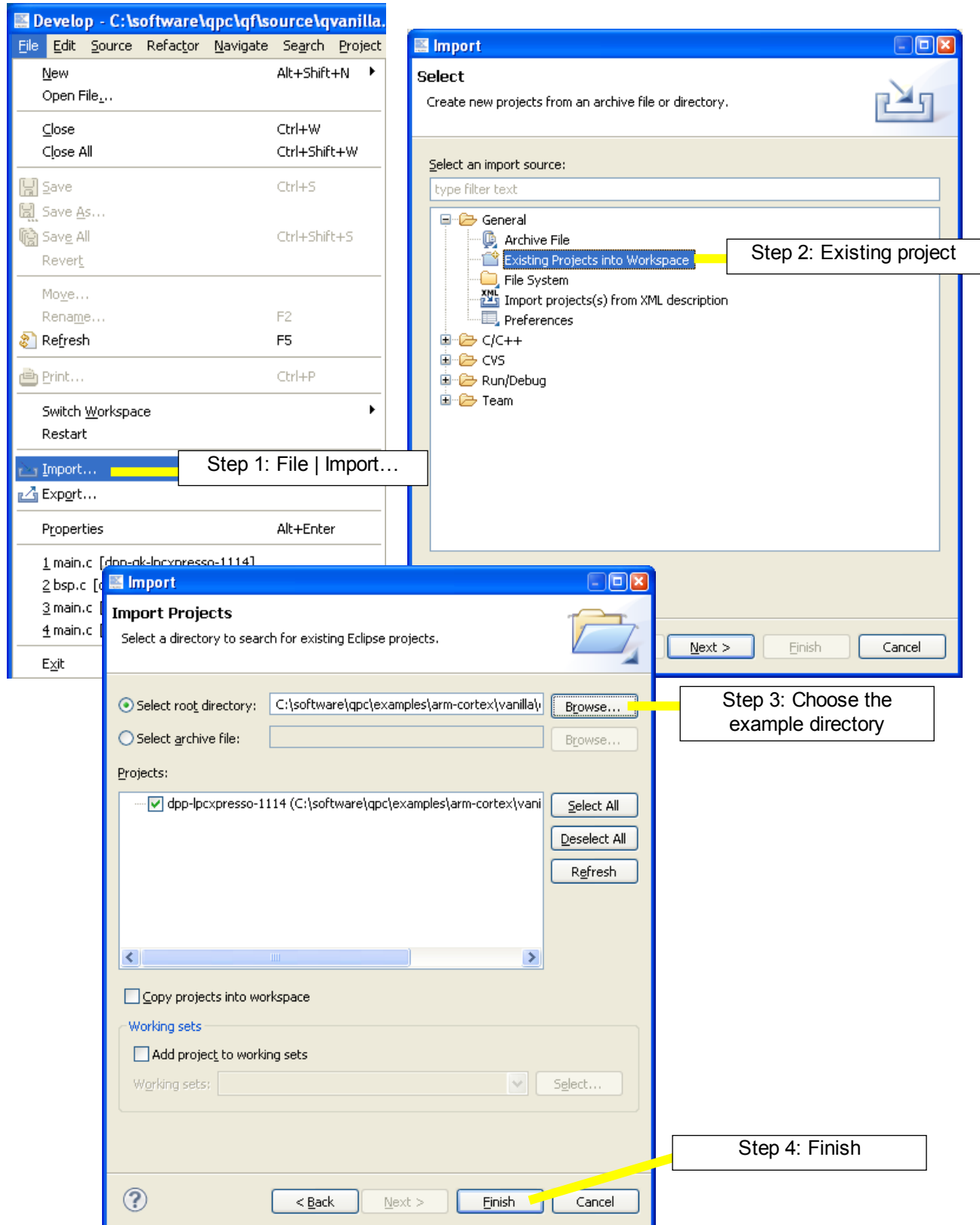
C:\software\qpc\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114>make CONF=spy
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -MM -MT spy/uart.o -mcpu=
=cortex-m0 -mthumb -Wall -g -O -I./../../../../include -I../../../../ports/a
rm-cortex/vanilla/gnu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=C
MSISv1p30_LPC11xx -DQ_SPY lpc11xx_lib/src/uart.c > spy/uart.d
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -MM -MT spy/timer32.o -m
cpu=cortex-m0 -mthumb -Wall -g -O -I./../../../../include -I../../../../ports/a
rm-cortex/vanilla/gnu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS
S=CMSISv1p30_LPC11xx -DQ_SPY lpc11xx_lib/src/timer32.c > spy/timer32.d
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -MM -MT spy/timer16.o -m
cpu=cortex-m0 -mthumb -Wall -g -O -I./../../../../include -I../../../../ports/a
rm-cortex/vanilla/gnu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS
S=CMSISv1p30_LPC11xx -DQ_SPY lpc11xx_lib/src/timer16.c > spy/timer16.d
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -MM -MT spy/gpio.o -mcp
u=cortex-m0 -mthumb -Wall -g -O -I./../../../../include -I../../../../ports/a
rm-cortex/vanilla/gnu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=C
MSISv1p30_LPC11xx -DQ_SPY lpc11xx_lib/src/gpio.c > spy/gpio.d
.
.
.
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb
-Wall -g -O -I./../../../../include -I../../../../ports/arm-cortex/vanilla/g
nu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC11xx -
DQ_SPY -c lpc11xx_lib/src/clkconfig.c -o spy/clkconfig.o
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb
-Wall -g -O -I./../../../../include -I../../../../ports/arm-cortex/vanilla/g
nu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC11xx -
DQ_SPY -c lpc11xx_lib/src/gpio.c -o spy/gpio.o
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb
-Wall -g -O -I./../../../../include -I../../../../ports/arm-cortex/vanilla/g
nu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC11xx -
DQ_SPY -c lpc11xx_lib/src/timer16.c -o spy/timer16.o
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb
-Wall -g -O -I./../../../../include -I../../../../ports/arm-cortex/vanilla/g
nu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC11xx -
DQ_SPY -c lpc11xx_lib/src/timer32.c -o spy/timer32.o
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m0 -mthumb
-Wall -g -O -I./../../../../include -I../../../../ports/arm-cortex/vanilla/g
nu -I. -Icmsis -Ilpc11xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC11xx -
DQ_SPY -c lpc11xx_lib/src/uart.c -o spy/uart.o
C:/tools/nxp/lpcxpresso_3.2/Tools/bin/arm-none-eabi-gcc -I lpc1114.ld -mcpu=corte
x-m0 -mthumb -nostdlib -Xlinker -Map=spy/dpp.map --gc-sections -L../../../../
ports/arm-cortex/vanilla/gnu/obj -o spy/dpp.elf spy/startup_LPC11.o spy/bsp.o sp
y/main.o spy/no_heap.o spy/phil.o spy/table.o spy/core_cm0.o spy/system_LPC11xx
.o spy/clkconfig.o spy/gpio.o spy/timer16.o spy/timer32.o spy/uart.o -lqf_cortex
-m0 -lqep_cortex-m0 -lqsc_cortex-m0
C:\software\qpc\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114>
  
```

### 2.1.2 Building the Examples from Eclipse

The example code contains the Eclipse projects for building and debugging the DPP examples with the LPCXpresso IDE from Code Red. The provided Eclipse projects are Makefile-type projects, which use the same Makefiles that you can call from the command line. In fact the Makefiles are specifically designed to allow building all supported configurations from Eclipse. Figure 4 shows how to import the provided projects into LPCXpresso IDE.

**NOTE:** The provided Makefiles allow you to create and configure the build configurations from the Project | Build Configurations | Manage... sub-menu. For the Release and Spy configurations, you should set the make command to make CONF=rel and make CONF=spy, respectively. The provided Makefile also correctly supports the clean targets, so invoking Project | Clean... menu for any build configuration works as expected.

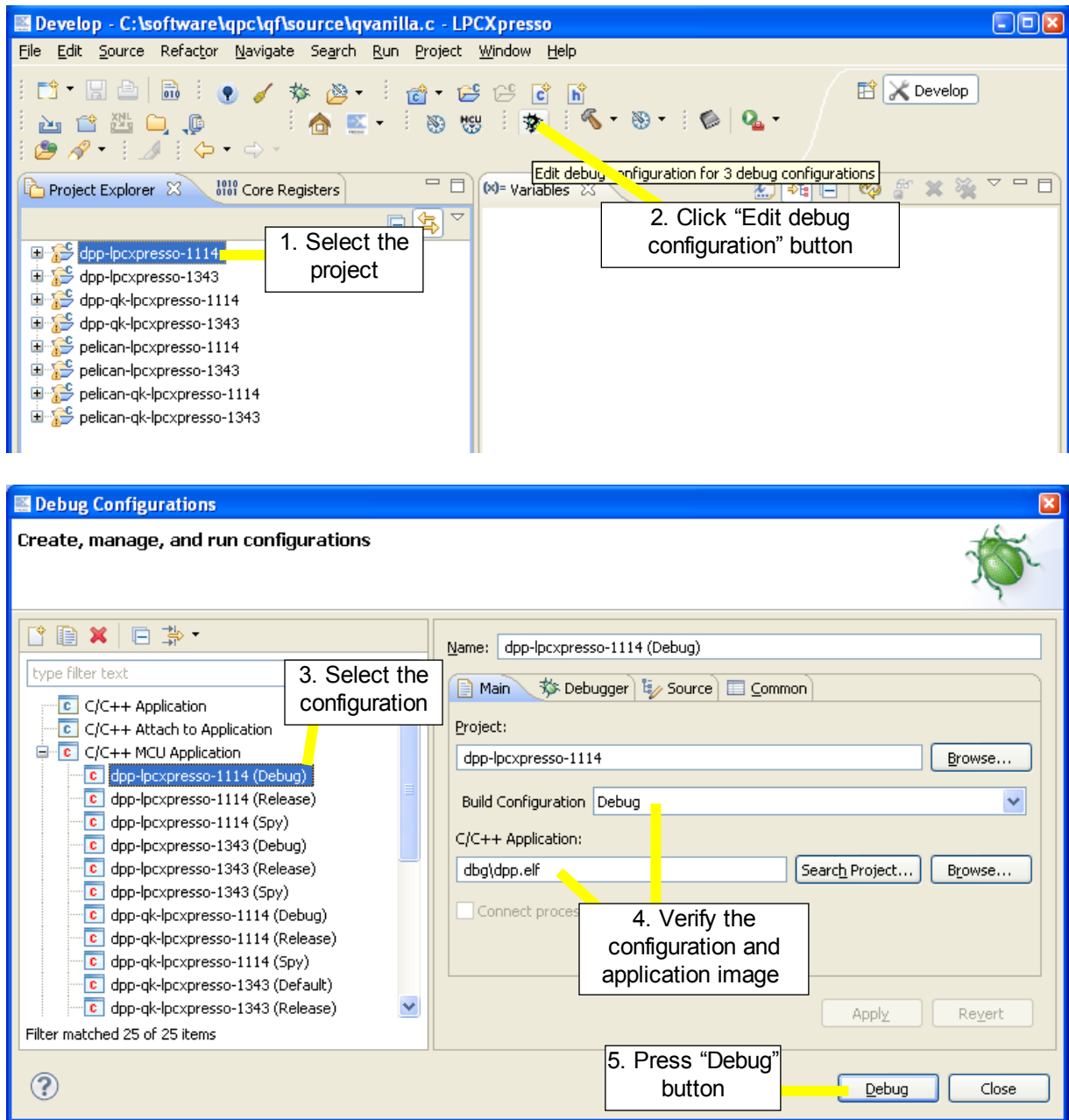
**Figure 4 Steps of importing the existing project into the Eclipse (LPCXpresso) IDE**



## 2.2 Downloading to Flash and Debugging the Examples

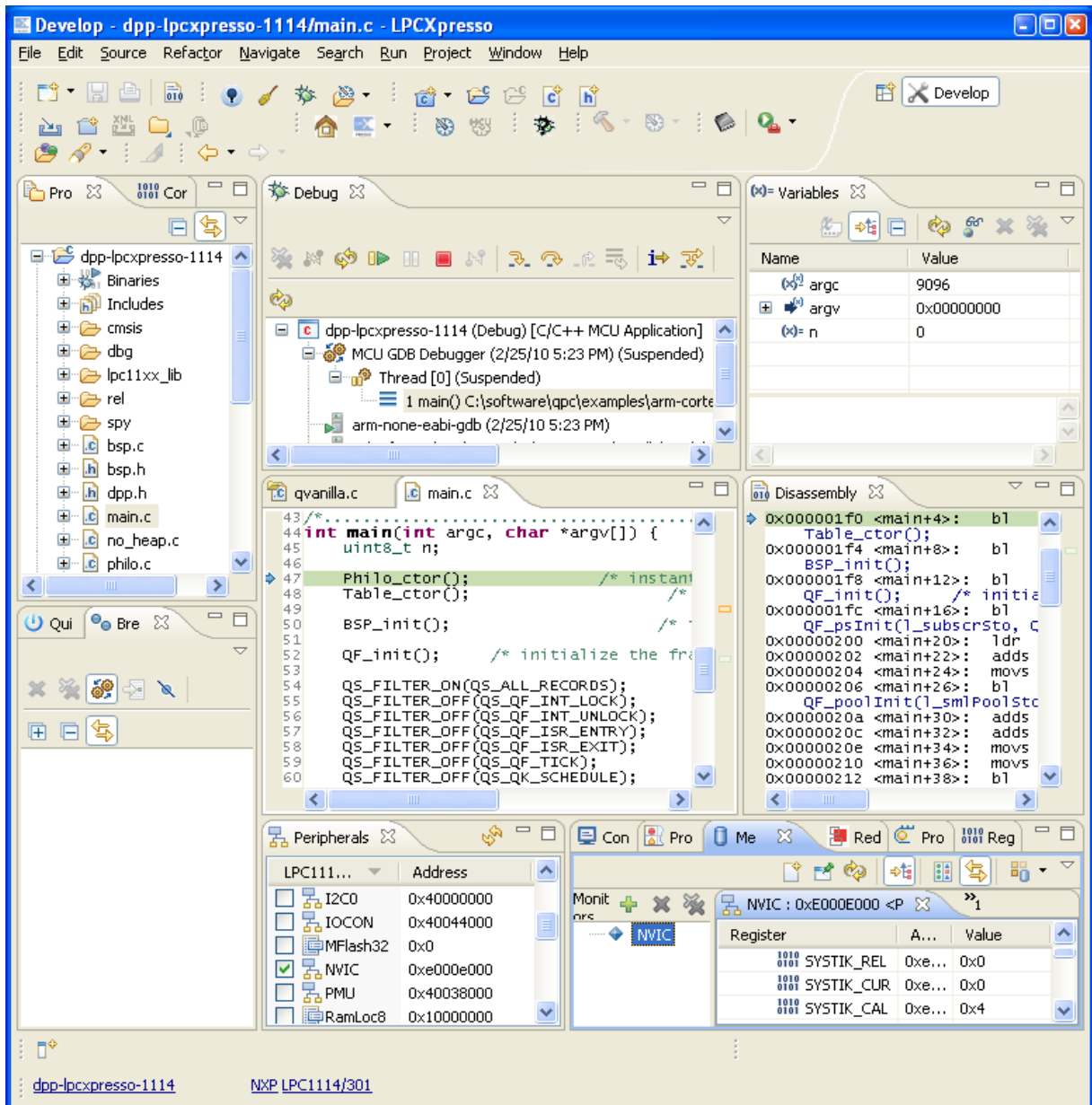
The example directory `<qp>\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114\` contains the “launch” files for the LPCXpresso IDE that contain all information required for flash-downloading and debugging all the build configurations of each project. Unlike other Eclipse-based IDEs, LPCXpresso takes care automatically for launching the GDB server application for the LPC-Link hardware debugger. Figure 5 shows the steps required to start debugging one of the provided projects with LPCXpresso IDE.

**Figure 5 Debugging the provided projects with LPCXpresso IDE**



The following screen shot in Figure 6 shows a debugging session in Eclipse with various views.

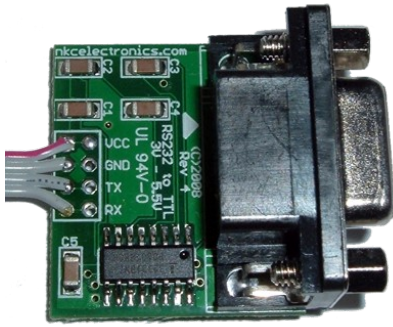
**Figure 6 Debugging with LPCXpresso IDE**



### 2.2.1 Software Tracing with Q-SPY

For the QS (Q-SPY) software tracing output, you need to connect a TTL to RS-232 transceiver to the LPCXpresso board, as shown in Figure 7. The figure shows the RS232 to TTL converter board 3.3V to 5V from NKC Electronics (<http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html>), but you can use any other equivalent board.

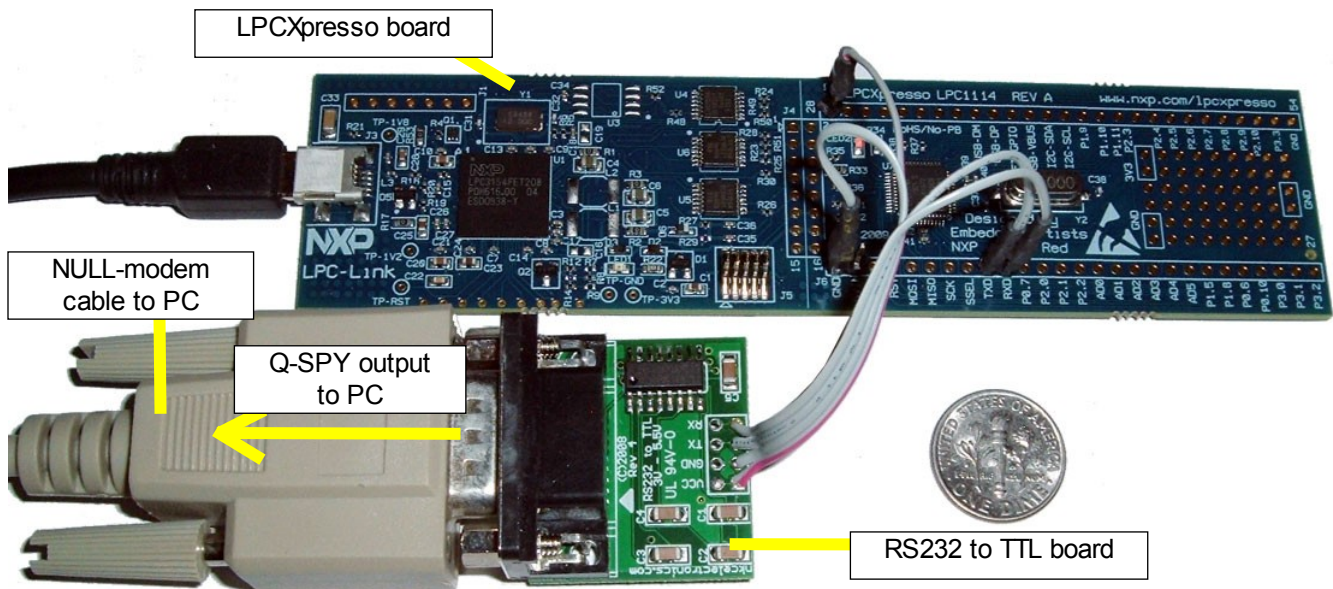




**NOTE:** For the QS (Q-SPY) software tracing output, you also need a TTL-to-RS-232 transceiver board. Such boards are available from a number of vendors. Figure 7 shows the RS232 to TTL converter board 3.3V to 5V from NKC Electronics (\$9.99 <http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html>).

The LPCXpresso board brings out all the MCU pins to the edges of the board. To connect the RS-232 transceiver you need to fit the pins into the following 3 positions: GND, 3V3, and TXD. (Figure 7 shows additionally RXD connection).

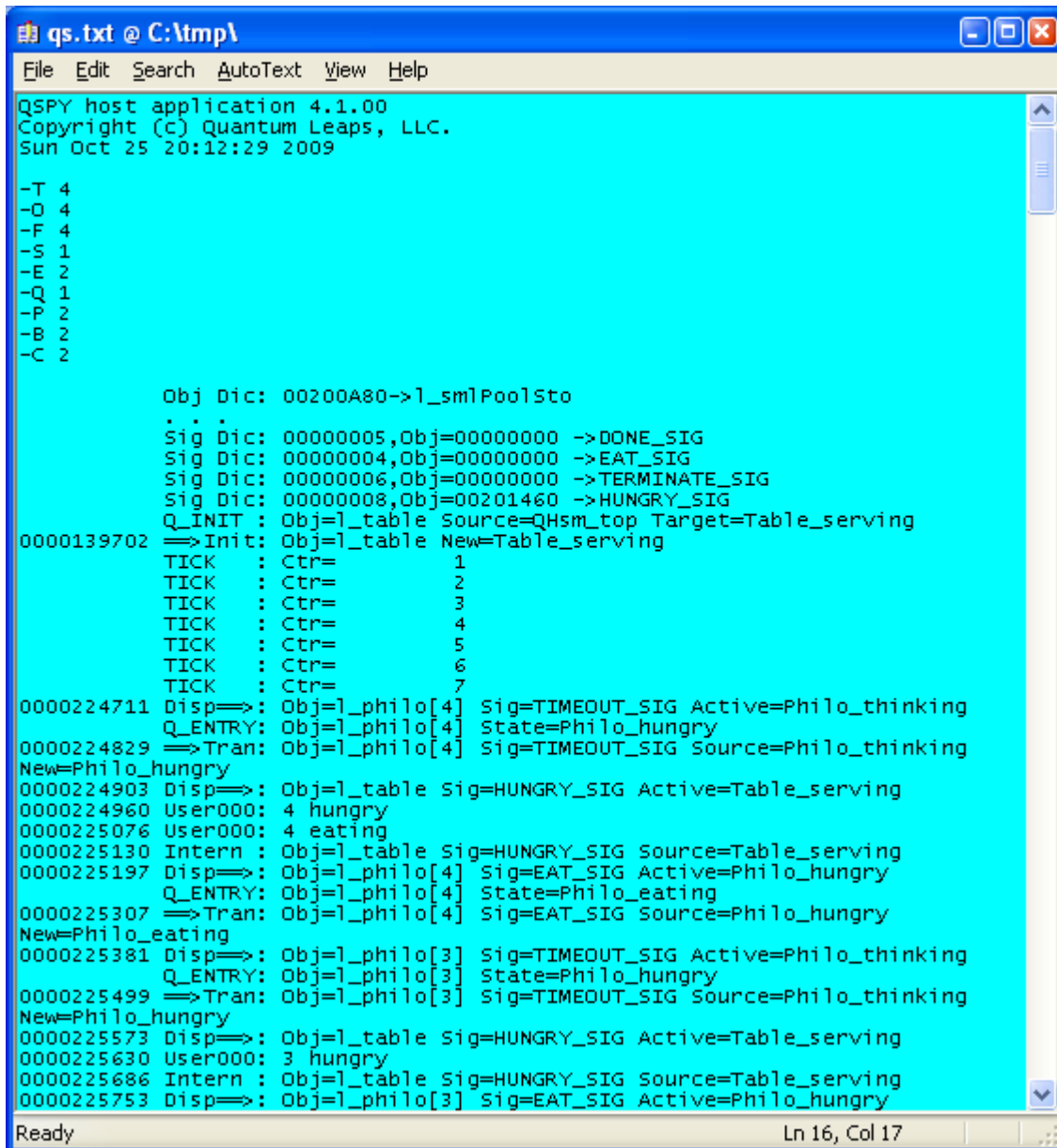
**Figure 7 Connecting RS232-TTL board to the LPCXpresso board.**



To see the QS software trace output, you also need to download the Spy build configuration to the target board. Next you need to launch the QSPY host utility to observe the output in the human-readable format. 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 -c COM1 -b 115200
```

Figure 8 Screen shot from the QSPY output



```

qs.txt @ C:\tmp\
File Edit Search AutoText View Help
QSPY host application 4.1.00
Copyright (c) Quantum Leaps, LLC.
Sun Oct 25 20:12:29 2009

-T 4
-O 4
-F 4
-S 1
-E 2
-Q 1
-P 2
-B 2
-C 2

Obj Dic: 00200A80->l_sm1Poolsto
Sig Dic: 00000005,Obj=00000000 ->DONE_SIG
Sig Dic: 00000004,Obj=00000000 ->EAT_SIG
Sig Dic: 00000006,Obj=00000000 ->TERMINATE_SIG
Sig Dic: 00000008,Obj=00201460 ->HUNGRY_SIG
Q_INIT : Obj=l_table Source=QHsm_top Target=Table_serving
0000139702 ==>Init: Obj=l_table New=Table_serving
TICK : Ctr= 1
TICK : Ctr= 2
TICK : Ctr= 3
TICK : Ctr= 4
TICK : Ctr= 5
TICK : Ctr= 6
TICK : Ctr= 7
0000224711 Disp==>: Obj=l_philo[4] Sig=TIMEOUT_SIG Active=Philo_thinking
Q_ENTRY: Obj=l_philo[4] State=Philo_hungry
0000224829 ==>Tran: Obj=l_philo[4] Sig=TIMEOUT_SIG Source=Philo_thinking
New=Philo_hungry
0000224903 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_serving
0000224960 User000: 4 hungry
0000225076 User000: 4 eating
0000225130 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000225197 Disp==>: Obj=l_philo[4] Sig=EAT_SIG Active=Philo_hungry
Q_ENTRY: Obj=l_philo[4] State=Philo_eating
0000225307 ==>Tran: Obj=l_philo[4] Sig=EAT_SIG Source=Philo_hungry
New=Philo_eating
0000225381 Disp==>: Obj=l_philo[3] Sig=TIMEOUT_SIG Active=Philo_thinking
Q_ENTRY: Obj=l_philo[3] State=Philo_hungry
0000225499 ==>Tran: Obj=l_philo[3] Sig=TIMEOUT_SIG Source=Philo_thinking
New=Philo_hungry
0000225573 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_serving
0000225630 User000: 3 hungry
0000225686 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000225753 Disp==>: Obj=l_philo[3] Sig=EAT_SIG Active=Philo_hungry
Ready Ln 16, Col 17

```

## 2.3 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, QS, and QK are provided inside the <qp>\ports\arm-cortex 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 LPCXpresso 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_<core>.bat` for building all the libraries located in the `<qp>\ports\arm-cortex\...` directory. For example, to build the debug version of all the QP libraries for ARM-Cortex, with the GNU ARM compiler, QK kernel, you open a console window on a Windows PC, change directory to `<qp>\ports\arm-cortex\qk\gnu\`, and invoke the batch by typing at the command prompt the following command:

```
make_cortex-m0
```

The build process should produce the QP libraries in the location: `<qp>\ports\arm-cortex\qk\gnu\-dbg\`. The `make.bat` files assume that the GNU toolset has been installed in the directory `C:\tools\NXP\lpcxpresso_3.2`.

---

**NOTE:** You need to adjust the symbol `GNU_ARM` at the top of the batch scripts if you've installed the GNU ARM toolchain into a different directory.

---

In order to take advantage of the QS ("spy") instrumentation, you need to build the QS version of the QP libraries. You achieve this by invoking the `make_cortex-m3.bat` utility with the "spy" target, like this:

```
make_cortex-m3 spy
```

The make process should produce the QP libraries in the directory: `<qp>\ports\arm-cortex\vanilla\gnu\spy\`. You choose the build configuration by providing a target to the `make_cortex-m3.bat` utility. The default target is "dbg". Other targets are "rel", and "spy" respectively. The following table summarizes the targets accepted by `make_cortex-m3.bat`.

**Table 2 Make targets for the Debug, Release, and Spy software configurations**

Software Version	Build command
Debug (default)	<code>make_cortex-m3</code>
Release	<code>make_cortex-m3 rel</code>
Spy	<code>make_cortex-m3 spy</code>

## 3 The Vanilla Port

The “vanilla” port shows how to use QP™ on a “bare metal” ARM Cortex-M0 or Cortex-M3 based system with the cooperative “vanilla” kernel. In the “vanilla” version of the QP, the only component requiring platform-specific porting is the QF component. The other two components: QEP and QS require merely recompilation and will not be discussed here. With the vanilla port you’re not using the QK component.

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

The QEP header file for the ARM-Cortex port is located in `<qp>\ports\arm-cortex\vanilla\gnu\qep_port.h`. Listing 2 shows the `qep_port.h` header file for ARM-Cortex/GNU. The GNU compiler is a standard C99 compiler, so I simply include the `<stdint.h>` header file that defines the platform-specific exact-width integer types.

**Listing 2 The `qep_port.h` header file for ARM-Cortex/GNU.**

```
#include <stdint.h>                /* C99-standard exact-width integer types */
#include "qep.h"                   /* QEP platform-independent public interface */
```

### 3.2 The QF Port Header File

The QF header file for the ARM-Cortex port is located in `<qp>\ports\arm-cortex\vanilla\gnu\qf_port.h`. This file specifies the interrupt locking/unlocking policy (QF critical section) as well as the configuration constants for QF (see Chapter 8 in [PSiCC2]).

The most important porting decision you need to make in the `qf_port.h` header file is the policy for locking and unlocking interrupts. The ARM-Cortex allows using the simplest “unconditional interrupt unlocking” policy (see Section 7.3.2 of the book “Practical UML Statecharts in C/C++, Second Edition” [PSiCC2]), because ARM-Cortex is equipped with the standard nested vectored interrupt controller (NVIC) and generally runs ISRs with interrupts unlocked. Listing 3 shows the `qf_port.h` header file for ARM-Cortex/GNU.

**Listing 3 The `qf_port.h` header file for ARM-Cortex/GNU.**

```
/* The maximum number of active objects in the application */
(1) #define QF_MAX_ACTIVE                63

/* QF critical section entry/exit */
(2) /* QF_INT_KEY_TYPE not defined: "unconditional interrupt unlocking" policy */
(3) #define QF_INT_LOCK(dummy)           __asm volatile ("cpsid i")
(4) #define QF_INT_UNLOCK(dummy)         __asm volatile ("cpsie i")

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

- (1) The `QF_MAX_ACTIVE` specifies the maximum number of active object priorities in the application. You always need to provide this constant. Here, `QF_MAX_ACTIVE` is set to the maximum limit of 63 active object priorities in the system. You can reduce this number to reduce the RAM footprint of the QF framework.



---

**NOTE:** The `qf_port.h` header file does not change the default settings for all the rest of various object sizes inside QF. Please refer to Chapter 8 of [PSiCC2] for discussion of all configurable QF parameters.

---

- (2) The `QF_INT_KEY_TYPE` is not defined, which means that the simple policy of “unconditional interrupt locking and unlocking” is applied.
- (3) The interrupt locking macro is the single “`CPSD i`” Thumb2 instruction.
- (4) The interrupt unlocking is the single “`CPSE i`” Thumb2 instruction.
- (5) This QF port uses the QEP event processor for implementing active object state machines.
- (6) This QF port uses the cooperative “vanilla” kernel.
- (7) The QF port must always include the platform-independent `qf.h` header file.

### 3.3 Handling Interrupts in the Non-Preemptive Vanilla Kernel

Even though ARM Cortex is designed to use regular C functions as exception and interrupt handlers, in the GNU toolchain functions that are used directly as interrupt handlers should be annotated with `__attribute__((__interrupt__))`. This tells the GNU compiler to add special stack alignment code to the function prologue.

---

**NOTE:** Because of a discrepancy between the ARMv7M Architecture and the ARM EABI, it is not safe to use normal C functions directly as interrupt handlers. The EABI requires the stack be 8-byte aligned, whereas ARMv7M only guarantees 4-byte alignment when calling an interrupt vector. This can cause subtle runtime failures, usually when 8-byte types are used [CodeSourcery].

---

Typically, ISRs are not part of the generic QP port, because it’s much more convenient to define ISRs at the application level. The following listing shows all the ISRs in the DPP example application. Please note that the `SysTick_Handler()` ISR calls the `QF_tick()` to perform QF time-event management. (The `SysTick_Handler()` updates also the timestamp used in the QS software tracing instrumentation, see the upcoming Section 8).

```
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
#ifdef Q_SPY
    uint32_t dummy = HWREG(NVIC_ST_CTRL); /* clear NVIC_ST_CTRL_COUNT flag */
    QS_tickTime_ += QS_tickPeriod_;        /* account for the clock rollover */
#endif
    QF_tick();                             /* process all armed time events */
}
```

---

**NOTE:** This Application Note complies with the CMSIS standard, which dictates the names of all exception handlers and IRQ handlers.

---

#### 3.3.1 The Interrupt Vector Table

ARM-Cortex contains an interrupt vector table (also called the exception vector table) starting usually at address 0x00000000, typically in ROM. The vector table contains the initialization value for the main stack pointer on reset, and the entry point addresses for all exception handlers. The exception number defines the order of entries in the vector table.

ARM-Cortex requires you to place the initial Main Stack pointer and the addresses of all exception handlers and ISRs into the Interrupt Vector Table allocated typically in ROM. In the GNU compiler, the IDT is initialized in the `startup_LPC11.c` C-language module located in the CMSIS directory.

**Listing 4 The interrupt vector table defined in `startup_LPC11.c` (GNU compiler).**

```

. . .
/* exception and interrupt vector table -----*/
typedef void (*ExceptionHandler)(void);
typedef union {
    ExceptionHandler handler;
    void *pointer;
} VectorTableEntry;

/* top of stack defined in the linker script */
(1) extern unsigned __c_stack_top__;

/*.....*/
(2) __attribute__((section(".isr_vector")))
VectorTableEntry const g_pfnVectors[] = {
(3)     { .pointer = &__c_stack_top__ }, /* initial stack pointer */
(4)     { .handler = &Reset_Handler }, /* Reset Handler */
        { .handler = &NMI_Handler }, /* NMI Handler */
        { .handler = &HardFault_Handler }, /* Hard Fault Handler */
        { .handler = &MemManage_Handler }, /* MPU Fault Handler */
        { .handler = &BusFault_Handler }, /* Bus Fault Handler */
        { .handler = &UsageFault_Handler }, /* Usage Fault Handler */
        { .handler = &Spurious_Handler }, /* Reserved */
        { .handler = &Spurious_Handler }, /* Reserved */
        { .handler = &Spurious_Handler }, /* Reserved */
        { .handler = &Spurious_Handler }, /* Reserved */
        { .handler = &SVC_Handler }, /* SVC Call Handler */
        { .handler = &DebugMon_Handler }, /* Debug Monitor Handler */
        { .handler = &Spurious_Handler }, /* Reserved */
        { .handler = &PendSV_Handler }, /* PendSV Handler */
        { .handler = &SysTick_Handler }, /* SysTick Handler */

        /* external interrupts (IRQs) ... */
(5)     { .handler = WAKEUP_IRQHandler }, /* PIO0_0 Wakeup */
        . . .
};

```

- (1) The main stack is allocated in its own `.stack` section in the `lpc1114.ld` linker script.

**NOTE:** The linker script is the place where you determine the stack size. You need to adjust the size of this section to suit your specific application.

**NOTE:** All QP ports, including the Vanilla port and the QK port use only the main stack (the C-stack). User stack pointer is not used at all.

- (2) The vector table is explicitly placed in the `.isr_vector` section, which the linker script locates at the beginning of Flash at address `0x00000000`.

- (3) The first entry in the IDT is the top of the main stack. The symbol `__c_stack_top` is provided by the linker script.
- (4) The subsequent entries in the IDT are exception handlers. The `Reset_Handler` is defined later in the startup file.
- (5) The third part of the IDT is for interrupt handlers supported by the specific MCU.

### 3.3.2 Starting Interrupts in `QF_onStartup()`

ARM-Cortex provides the `SysTick` facility, specifically designed to provide the periodic system time tick, which is configured to deliver the system tick at the `BSP_TICKS_PER_SEC` rate. The configuration of the `SysTick` is done in the `QF_onStartup()` callback using the CMSIS library function `SysTick_Config()`.

QP invokes the `QF_onStartup()` callback just before starting the event loop inside `QF_run()`. The `QF_onStartup()` function is located in the file `bsp.c` and must start the interrupts, in particular the time-tick interrupt.

```
enum ISR_Priorities {    /* ISR priorities starting from the highest urgency */
    GPIOPORTA_PRIO,
    SYSTICK_PRIO,
    // ...
};

void QF_onStartup(void) {
    /* set up the SysTick timer to fire at BSP_TICKS_PER_SEC rate */
    SysTick_Config(SystemFrequency / BSP_TICKS_PER_SEC);

    /* set priorities of all interrupts in the system... */
    NVIC_SetPriority(SysTick_IRQn, SYSTICK_PRIO);
    NVIC_SetPriority(GPIOPortA_IRQn, GPIOPORTA_PRIO);

    NVIC_EnableIRQ(GPIOPortA_IRQn);
}
```

### 3.4 Idle Loop Customization in the “Vanilla” Port

As described in Chapter 7 of [PSiCC2], the “vanilla” port uses the non-preemptive scheduler built into QP. If no events are available, the non-preemptive scheduler invokes the platform-specific callback function `QF_onIdle()`, which you can use to save CPU power, or perform any other “idle” processing (such as Quantum Spy software trace output).

---

**NOTE:** The idle callback `QF_onIdle()` must be invoked with interrupts locked, because the idle condition can be changed by any interrupt that posts events to event queues. `QF_onIdle()` **must** internally unlock interrupts, ideally atomically with putting the CPU to the power-saving mode (see also Chapter 7 in [PSiCC2]).

---

Because `QF_onIdle()` must enable interrupts internally, the signature of the function depends on the interrupt locking policy. In case of the simple “unconditional interrupt locking and unlocking” policy, which is used in this ARM-Cortex port, the `QF_onIdle()` takes no parameters.

shows an example implementation of `QF_onIdle()` for the LPC1114 MCU. Other ARM-Cortex embedded microcontrollers (e.g., ST’s STM32) handle the power-saving mode very similarly.

---

**Listing 5 QF\_onIdle() callback.**

```
(1) void QF_onIdle(void) {           /* entered with interrupts LOCKED, see NOTE01 */  
    . . .  
(2) #if defined NDEBUG  
    /* Put the CPU and peripherals to the low-power mode.  
     * you might need to customize the clock management for your application,  
     * see the datasheet for your particular ARM-Cortex MCU.  
     */  
(3)     __WFI();                      /* Wait-For-Interrupt */  
    #endif  
(4)     QF_INT_UNLOCK(dummy);         /* always unlock the interrupts */  
}
```

- (1) The cooperative Vanilla kernel calls the `QF_onIdle()` callback with interrupts locked, to avoid race condition with interrupts that can post events to active objects and thus invalidate the idle condition.
- (2) The sleep mode is used only in the non-debug configuration, because sleep mode stops CPU clock, which can interfere with debugging.
- (3) The Thumb2 instruction `WFI` (Wait for Interrupt) stops the CPU clock. Note that the CPU stops executing at this line and that interrupts are still **locked**. An active interrupt first starts the CPU clock again, so the CPU starts executing again. Only after unlocking interrupts in line (4) the interrupt that woke the CPU up is serviced.
- (4) The `QF_onIdle()` callback must always unlock interrupts.

---

**NOTE:** The idle callback `QF_onIdle()` must unlock interrupts in every path through the code.

---



## 4 The QK Port

This section describes how to use QP on ARM-Cortex with the preemptive QK real-time kernel described in Chapter 10 of [PSiCC2]. The benefit is very fast, fully deterministic task-level response and that execution timing of the high-priority tasks (active objects) will be virtually insensitive to any changes in the lower-priority tasks. The downside is bigger RAM requirement for the stack. Additionally, as with any preemptive kernel, you must be very careful to avoid any sharing of resources among concurrently executing active objects, or if you do need to share resources, you need to protect them with the QK priority-ceiling mutex (again see Chapter 10 of [PSiCC2]).

---

**NOTE:** The preemptive configuration with QK uses more stack than the non-preemptive “Vanilla” configuration. You need to adjust the size of this stack to be large enough for your application.

---

### 4.1 Single-Stack, Preemptive Multitasking on ARM-Cortex

The ARM-Cortex architecture provides a rather unorthodox way of implementing preemptive multitasking, which is designed primarily for the traditional real-time kernels that use multiple per-task stacks. This section explains how the run-to-completion preemptive QK kernel works on ARM-Cortex.

1. The ARM-Cortex processor executes application code in the Privileged Thread mode, which is exactly the mode entered out of reset. The exceptions (including all interrupts) are always processed in the Privileged Handler mode.
2. QK uses only the Main Stack Pointer (QK is a single stack kernel). The Process Stack Pointer is not used and is not initialized.
3. The QK port uses the PendSV (exception number 14) and the SVCcall (exception number 11) to perform asynchronous preemptions and context switch, respectively (see Chapter 10 in [PSiCC2]). The application code (your code) **must** initialize the Interrupt Vector Table with the addresses of `PendSV_Handler` and `SVCcall_Handler` exception handlers. Additionally, the interrupt table must be initialized with the SysTick handler that calls `QF_tick()`.
4. The application code (your code) **must** call the function `QK_init()` to set the priorities of the PendSV and the SVCcall exceptions to the lowest level in the whole system. The function `QK_init()` sets the priorities of exceptions 14 and 11 to the numerical value of 0xFF. The priorities are set with interrupts disabled, but the interrupt lock key is restored upon the function return.

---

**NOTE:** The Stellaris ARM-Cortex silicon supports only 3 most-significant bits of priority, therefore writing 0xFF to a priority register reads back 0xE0.

---

5. It is strongly recommended that you do **not** assign the lowest priority (0xE0) to any interrupt in your application. This leaves the following 7 priority levels for you (listed from the lowest to the highest urgency): 0xC0, 0xA0, 0x80, 0x60, 0x40, 0x20, and 0x00 (the highest priority).

---

**NOTE:** The QK configuration still works even when some interrupts use the lowest priority level 0xE0. However, the use of the stack is less than optimal in this case and the preemption scenarios are more complicated than those shown in Figure 9.

---

6. Every ISR **must** set the pending flag for the PendSV exception in the NVIC. This is accomplished in the macro `QK_ISR_EXIT()`, which **must** be called just before exiting from all ISRs (see upcoming Section 4.2.1).

7. ARM-Cortex enters interrupt context without locking interrupts (without setting the PRIMASK bit). Generally, you should not lock interrupts inside ISRs. In particular, the QF services `QF_postISR()` and `QF_tick()` should be called with interrupts enabled, to avoid nesting of critical sections.

**NOTE:** If you don't wish an interrupt to be preempted by another interrupt, you can always prioritize that interrupt in the NVIC to a higher level (use a lower numerical value of priority).

8. In ARM-Cortex the whole prioritization of interrupts, including the PendSV exception, is performed entirely by the NVIC. Because the PendSV has the lowest priority in the system, the NVIC tail-chains to the PendSV exception only after exiting the last nested interrupt.

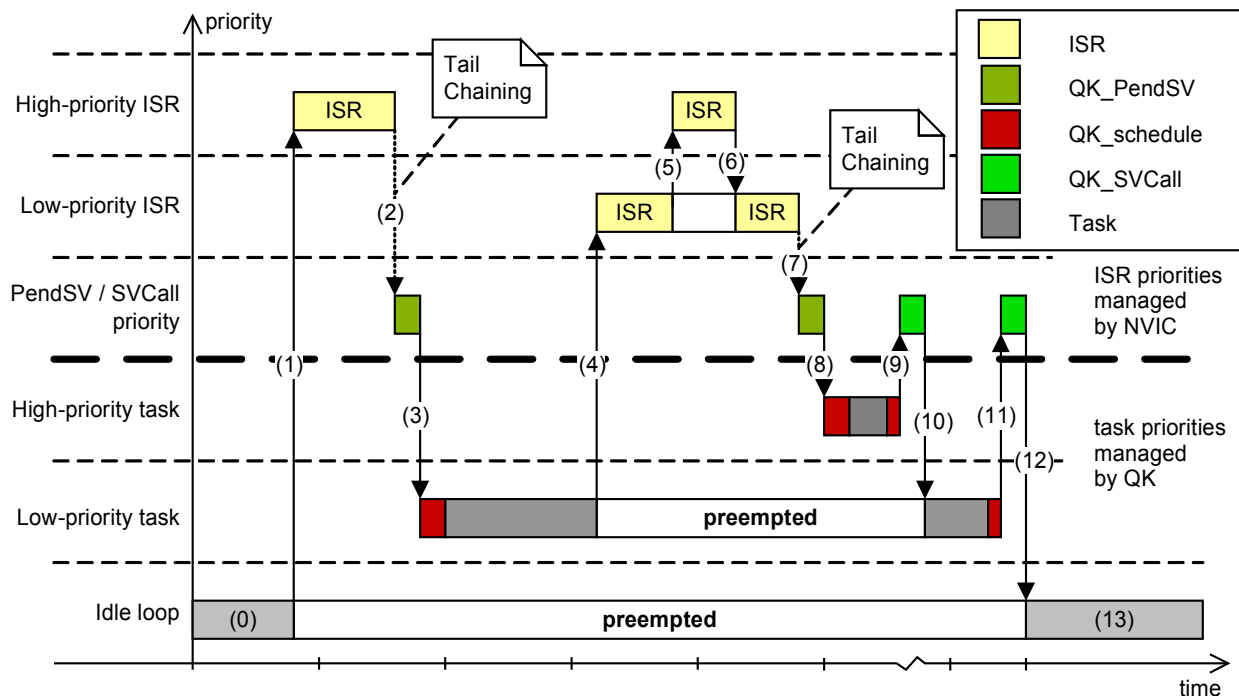
9. The restoring of the 8 registers comprising the ARM-Cortex interrupt stack frame in PendSV is wasteful in a single-stack kernel (see Listing 7(3) and (8)), but is necessary to perform full interrupt return from PendSV to signal End-Of-Interrupt to the NVIC.

10. The pushing of the 8 registers comprising the ARM-Cortex interrupt stack frame upon entry to SVCcall is wasteful in a single-stack kernel (see Figure 9(10) and (12)), but is necessary to perform full interrupt return to the preempted context upo SVCcall's return.

#### 4.1.1 Examples of Various Preemption Scenarios in QK

Figure 9 illustrates several preemption scenarios in QK.

**Figure 9 Various preemption scenarios in the QK preemptive kernel for ARM-Cortex.**



(0) The timeline in Figure 9 begins with the QK executing the idle loop.

(1) At some point an interrupt occurs and the CPU immediately suspends the idle loop, pushes the interrupt stack frame to the Main Stack and starts executing the ISR.

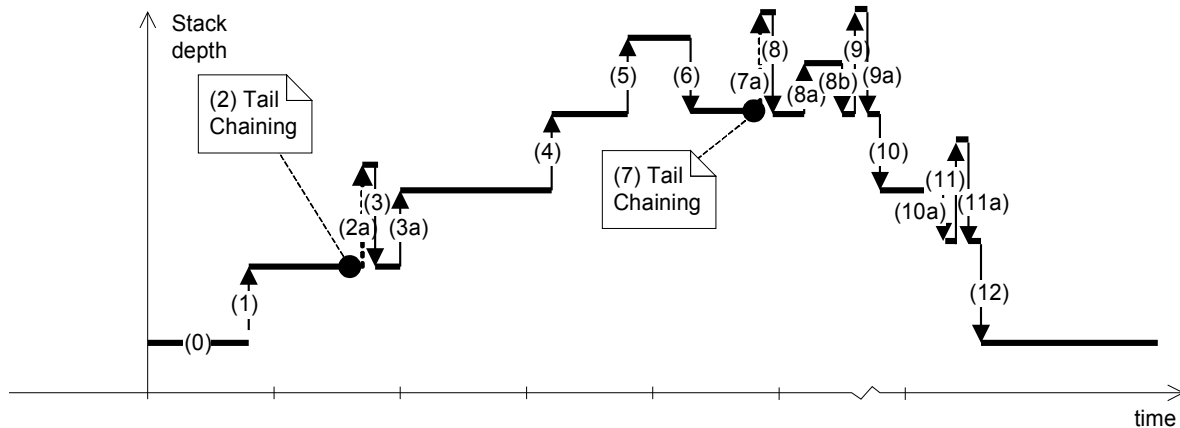
- (2) The ISR performs its work, and in QK always sets the pending flag for the **PendSV** exception in the NVIC. The priority of the PendSV exception is configured to be the lowest of all exceptions, so the ISR continues executing and PendSV exception remains pending. At the ISR return, the ARM-Cortex CPU performs tail-chaining to the pending PendSV exception.
- (3) The whole job of the PendSV exception is to synthesize an interrupt stack frame on top of the stack and perform an interrupt return.
- (4) The PC (exception return address) of the synthesized stack frame is set to `QK_schedule()` (more precisely to a thin wrapper around `QK_schedule()`, see Section 4.3), so the PendSV exception returns to the QK scheduler. The scheduler discovers that the Low-priority task is ready to run (the ISR has posted event to this task). The QK scheduler enables interrupts and launches the Low-priority task, which is simply a C-function call in QK. The Low-priority task (active object) starts running. Some time later another interrupt occurs. The Low-priority task is suspended and the CPU pushes the interrupt stack frame to the Main Stack and starts executing the ISR
- (5) The Low-priority ISR runs and sets the pending flag for the PendSV exception in the NVIC. Before the Low-priority ISR completes, it too gets preempted by a High-priority ISR. The CPU pushes another interrupt stack frame and starts executing the High-priority ISR.
- (6) The High-priority ISR again sets the pending flag for the PendSV exception (setting an already set flag is not an error). When the High-priority ISR returns, the NVIC does not tail-chain to the PendSV exception, because a higher-priority ISR than PendSV is still active. The NVIC performs the normal interrupt return to the preempted Low-priority interrupt, which finally completes.
- (7) Upon the exit from the Low-priority ISR, the NVIC performs tail-chaining to the pending `PendSV` exception
- (8) The `PendSV` exception synthesizes an interrupt stack frame to return to the QK scheduler.
- (9) The QK scheduler detects that the High-priority task is ready to run and launches the High-priority task (normal C-function call). The High-priority task runs to completion and returns to the scheduler. The scheduler does not find any more higher-priority tasks to execute and needs to return to the preempted task. The only way to restore the interrupted context in ARM-Cortex is through the interrupt return, but the task is executing outside of the interrupt context (in fact, tasks are executing in the Privileged Thread mode). The task enters the Handler mode by causing the synchronous **SVC**Call exception
- (10) The only job of the SVCCall exception is to discard its own interrupt stack frame and return using the interrupt stack frame that has been on the stack from the moment of task preemption
- (11) The Low-priority task, which has been preempted all that time, resumes and finally runs to completion and returns to the QK scheduler. The scheduler does not find any more tasks to launch and causes the synchronous SVCCall exception
- (12) The SVCCall exception discards its own interrupt stack frame and returns using the interrupt stack frame from the preempted task context
- (13) The QK idle loop resumes and runs till another interrupt preempts it

#### 4.1.2 The Stack Utilization in QK

Charting the stack utilization over time provides another, complementary view of the preemption scenarios depicted in Figure 9. To demonstrate the essential behavior, the irrelevant function calls and other unrelated stack activity will be ignored.

Figure 10 illustrates the stack utilization across the same preemption scenarios depicted in Figure 9. The timeline and labels used in Figure 9 and Figure 10 are identical to enable easy correlating of these two figures.

**Figure 10 Stack depth vs. time for various preemption scenarios in QK for ARM-Cortex**



- (0) The timeline in Figure 10 begins with the QK executing the idle loop.
- (1) At some point an interrupt occurs and the CPU immediately suspends the idle loop, pushes the interrupt stack frame to the Main Stack.
- (2) The ISR starts executing. At the ISR return, the ARM-Cortex CPU performs tail-chaining to the pending PendSV exception
- (3) Note that tail-chaining keeps the interrupt stack frame unchanged. The PendSV exception synthesizes an interrupt stack frame on top of the stack (2a) and performs an interrupt return
- (3a) The PC (exception return address) of the synthesized stack frame is set to `QK_schedule_()` (more precisely to a thin wrapper around `QK_schedule_()`, see Section 4.3), so the PendSV exception returns to the QK scheduler. The scheduler discovers that the Low-priority task is ready to run (the ISR has posted event to this task). The QK scheduler enables interrupts and launches the Low-priority task, which is simply a C-function call in QK
- (4) The Low-priority task (active object) starts running. Some time later another interrupt occurs. The Low-priority task is suspended and the CPU pushes the interrupt stack frame to the Main Stack
- (5) and starts executing the ISR. Before the Low-priority ISR completes, it too gets preempted by a High-priority ISR. The CPU pushes another interrupt stack frame and starts executing the High-priority ISR
- (6) When the High-priority ISR returns, NVIC performs the normal interrupt return to the preempted Low-priority interrupt
- (7) Upon the exit from the Low-priority ISR, the NVIC performs tail-chaining to the pending PendSV exception. Note that tail-chaining keeps the interrupt stack frame unchanged.
- (7a) The PendSV exception synthesizes an interrupt stack frame as though the interrupt occurred at the beginning of the QK scheduler.
- (8) The PendSV exception returns to the QK scheduler.
- (8a) The QK scheduler detects that the High-priority task is ready to run and launches the High-priority task via a normal C-function call.
- (8b) The High-priority task runs to completion and returns to the scheduler.
- (9) The scheduler causes the synchronous SVC call exception
- (9a) The only job of the SVC call exception is to discard its own interrupt stack frame



- (10) The SVCcall exception returns using the interrupt stack frame that has been on the stack from the moment of task preemption.
- (10a) The Low-priority task, which has been preempted all that time, resumes and finally runs to completion and returns to the QK scheduler.
- (11) The scheduler does not find any more tasks to launch and causes the SVCcall exception.
- (11a) The SVCcall exception discards its own interrupt stack frame
- (12) The SVCcall exception returns using the interrupt stack frame from the preempted task context.
- (13) The QK idle loop resumes and runs till another interrupt preempts it.

## 4.2 The QK Port Header File

In the QK port, you use very similar configuration as the “Vanilla” port described earlier. This section describes only the differences, specific to the QK component.

You configure and customize QK through the header file `qk_port.h`, which is located in the QP ports directory `<qp>\ports\arm-cortex\qk\gnu\`. The most important function of `qk_port.h` is specifying interrupt entry and exit.

---

**NOTE:** As any **preemptive** kernel, QK needs to be notified about entering the interrupt context and about exiting an interrupt context in order to perform a context switch, if necessary.

---

### Listing 6 qk\_porth.h header file

- ```

(1) #define QK_ISR_ENTRY() do { \
(2)     __asm volatile ("cpsid i"); \
(3)     ++QK_intNest_; \
(4)     QF_QS_ISR_ENTRY(QK_intNest_, QK_currPrio_); \
(5)     __asm volatile ("cpsie i"); \
    } while (0)

(6) #define QK_ISR_EXIT() do { \
(7)     __asm volatile ("cpsid i"); \
(8)     QF_QS_ISR_EXIT(QK_intNest_, QK_currPrio_); \
(9)     --QK_intNest_; \
(10)    *((uint32_t volatile *)0xE000ED04) = 0x10000000; \
(11)    __asm volatile ("cpsie i"); \
    } while (0)

(12) #include "qk.h"                                /* QK platform-independent public interface */

```
- (1) The `QK_ISR_ENTRY()` macro notifies QK about entering an ISR. The macro body is surrounded by the `do {...} while (0)` loop, which is the standard way of grouping instructions without creating a dangling-else or other syntax problems. In ARM-Cortex, this macro is called with interrupts unlocked, because the ARM-Cortex hardware does not set the PRIMASK upon interrupt entry.
  - (2) Interrupts are locked at the ARM-Cortex core level to perform the following actions atomically.
  - (3) The QK interrupt nesting level `QK_intNest_` is incremented to account for entering an ISR. This prevents invoking the QK scheduler from event posting functions (such as `QActive_postFIFO()` or `QF_publish()`) to perform a synchronous preemption.

- (4) The macro `QF_QS_ISR_ENTRY()` contains the QS instrumentation for interrupt entry. This macro generates no code when QS is inactive (i.e., `Q_SPY` is not defined).
- (5) Interrupts are unlocked at the ARM-Cortex core level to allow interrupt preemptions.
- (6) The `QK_ISR_EXIT()` macro notifies QK about exiting an ISR.
- (7) Interrupts are locked at the ARM-Cortex core level to perform the following actions atomically.
- (8) The macro `QF_QS_ISR_EXIT()` contains the QS instrumentation for interrupt exit. This macro generates no code when QS is inactive (i.e., `Q_SPY` is not defined).
- (9) The QK interrupt nesting level `QK_intNest_` is decremented to account for exiting an ISR. This balances step (3).
- (10) This write to the `NVIC_INT_CTRL` register sets the pending flag for the `PendSV` exception.

---

**NOTE:** Setting the pending flag for the `PendSV` exception in every ISR is absolutely **critical** for proper operation of QK. It really does not matter at which point during the ISR execution this happens. Here the `PendSV` is pending at the exit from the ISR, but it could as well be pending upon the entry to the ISR, or anywhere in the middle.

---

- (11) Interrupts are unlocked to perform regular exit from the ISR.
- (12) The QK port header file must include the platform-independent QK interface `qk.h`.

#### 4.2.1 The QK Critical Section

The interrupt locking/unlocking policy in the QK port is the same as in the vanilla port. Please refer to the earlier Section 3.2 for the description of the critical section implementation.

### 4.3 QK Platform-Specific Code for ARM-Cortex

The QK port to ARM-Cortex requires coding the PendSV and SVCALL exceptions in assembly. This ARM-Cortex-specific code is located in the file <qp>\ports\arm-cortex\qk\gnu\src\qk\_port.s.

**Listing 7 QK assembly code for ARM Cortex-M0/M3.**

```
.syntax unified
.cpu cortex-m3
.fpu softvfp
.thumb

/*****
*
* The QK_init function sets the priorities of SVCALL and PendSV exceptions
* to the lowest level possible (0xFF). The function internally disables
* interrupts, but restores the original interrupt lock before exit.
*
*****/
.section .text.QK_init
.global QK_init
.type QK_init, %function
(1) QK_init:
(2) MRS r0, PRIMASK /* store the state of the PRIMASK in r0 */
(3) CPSID i /* disable interrupts (set PRIMASK) */

(4) LDR r1, =0xE000ED18 /* System Handler Priority Register */
(5) LDR r2, [r1, #4] /* load the System 8-11 Priority Register */
(6) ORR r2, r2, #0xFF000000 /* set PRI_11 (SVCALL) to 0xFF */
(7) STR r2, [r1, #4] /* write the System 8-11 Priority Register */
(8) LDR r2, [r1, #8] /* load the System 12-15 Priority Register */
(9) ORR r2, r2, #0x00FF0000 /* set PRI_14 (PendSV) to 0xFF */
(10) STR r2, [r1, #8] /* write the System 12-15 Priority Register */

(11) MSR PRIMASK, r0 /* restore the original PRIMASK */
(12) BX lr /* return to the caller */
.size QK_init, . - QK_init

/*****
*
* The PendSV_Handler exception handler is used for handling asynchronous
* preemptions in QK. The use of the PendSV exception is the recommended and
* most efficient method for performing context switches with ARM Cortex.
*
* The PendSV exception should have the lowest priority in the whole system
* (0xFF, see QK_init). All other exceptions and interrupts should have higher
* priority. For Cortex-M3 with 3 priority bits, all interrupts and exceptions
* must have numerical value of priority lower than 0xE0. The seven interrupt
* priority levels available to your applications are (in the order from
* the lowest urgency to the highest urgency):
* 0xC0, 0xA0, 0x80, 0x60, 0x40, 0x20, 0x00.
*
*****/
```

```

* Also, all ISRs in the QK application *MUST* trigger the PendSV
* exception (by calling the QK_ISR_EXIT() macro).
*
* Due to tail-chaining and its lowest priority, the PendSV exception will be
* entered immediately after the exit from the last nested interrupt (or
* exception). In QK, this is exactly the time when the QK scheduler needs to
* check for the asynchronous preemptions.
*
*****/
    .section .text.PendSV_Handler
    .global PendSV_Handler
    .type PendSV_Handler, %function
(13) PendSV_Handler:
(14)     LDR     r0,=QK_readySet_ /* load the address of QK_readySet_ */
(15)     LDRB   r0,[r0] /* load the value of QK_readySet_ */
(16)     CBZ    r0,iret /* if QK_readySet_ == 0, return from interrupt*/

(17)     MOV    r1,#0x01000000 /* make up a task xPSR with only the T bit set*/
(18)     LDR    r0,=schedule /* load the address of sched wrapper (new PC) */
(19)     PUSH   {r0-r1} /* push xPSR,PC */
(20)     SUB    sp,sp,#(6*4) /* don't care for lr,r12,r3,r2,r1,r0 */
    ired:
(21)     BX     lr /* interrupt return to the task or scheduler */

(22) schedule:
(23)     CPSID  i /* disable interrupts at processor level */
(24)     BL     QK_schedule_ /* call the QK scheduler */
(25)     CPSIE  i /* enable interrupts to allow SVCcall exception*/
(26)     SVC    0 /* SV exception returns to the preempted task */
    .size PendSV_Handler, . - PendSV_Handler

/*****
*
* The SVC_Handler exception handler is used for returning back to the
* interrupted context (task or interrupt). The SVC exception should have
* the lowest priority in the whole system (see QK_init). The SVCcall
* exception simply removes its own interrupt stack frame from the stack and
* returns to the preempted task using the interrupt stack frame that must be
* at the top of the stack.
*
*****/
    .section .text.SVC_Handler
    .global SVC_Handler
    .type SVC_Handler, %function
(27) SVC_Handler:
(28)     ADD    sp,sp,#(8*4) /* remove one interrupt frame from the stack */
(29)     BX     lr /* return to the preempted task */
    .size SVC_Handler, . - SVC_Handler

.end

```

- (1) The `QK_init()` function sets the priorities of the PendSV exception (number 14) and SVCcall exception (number 11) to the lowest level 0xFF.

- (2) The PRIMASK register is stored in `r0`.
- (3) Interrupts are locked by setting the PRIMASK.
- (4) The address of the NVIC System Handler Priority Register 0 is loaded into `r1`
- (5) The contents of the NVIC System Handler Priority Register 1 (note the offset of 4) is loaded into `r2`
- (6) The priority byte `PRI_11` is set to `0xFF` without changing priority bytes in this register.
- (7) The contents of `r2` is stored in the NVIC System Handler Priority Register 1 (note the offset of 4).
- (8) The contents of the NVIC System Handler Priority Register 2 (note the offset of 8) is loaded into `r2`.
- (9) The priority byte `PRI_14` is set to `0xFF` without changing priority bytes in this register.
- (10) The contents of `r2` is stored in the NVIC System Handler Priority Register 2 (note the offset of 8).
- (11) The original PRIMASK value is restored.
- (12) The function `QK_init` returns to the caller.
- (13) The `PendSV` exception is always entered via tail-chaining from the last nested interrupt. The job of `PendSV` is to synthesize an interrupt stack frame and cause the interrupt return. The stack contents just before returning from `PendSV` is shown below:

```

Hi memory
    xPSR
    pc (interrupt return address)
    lr
    r12
    r3
    r2
    r1
old SP --> r0
    xPSR == 0x01000000
    PC == scheduler
    lr    don't care
    r12   don't care
    r3    don't care
    r2    don't care
    r1    don't care
    SP --> r0    don't care
  
```

Low memory

- (14) The address of the `QK_readySet_` is loaded to `r0`
- (15) The first byte of `QK_readySet_` is loaded to `r0`, if this byte is zero, the whole set is empty.
- (16) If the ready set is zero no tasks are pending and there is no need to run the scheduler, so branch is taken to return immediately from the ISR.
- (17) The value of `xPSR` register is synthesized in `r1`. The `xPSR` register has only the T bit set.
- (18) The address of the scheduler label is loaded into `r2`. This will be the return address
- (19) Registers `r2` and `r1` are pushed onto the stack
- (20) The stack pointer is adjusted by 6 registers. The actual stack contents for these registers is irrelevant.



- (21) `PendSV` exception returns using the special value of `lr` of `0xFFFFFFFF9` (return to Privileged Thread mode using the Main Stack pointer).
- (22) The “scheduler” label is where the `PendSV` exception returns to. It is a thin wrapper around the QK scheduler (`QK_schedule_`).
- (23) Interrupts are locked by setting the `PRIMASK` so that the `QK_schedule` function can be called with interrupts locked.
- (24) The `QK_schedule` function is called. Please note that `BL` instruction clobbers the `lr` register, but this does not matter.

---

**NOTE:** The QK scheduler runs in the Privileged Thread mode and unlocks interrupts before launching any tasks. Because the tasks run in the Thread mode with interrupts enabled, they can be preempted by interrupts of any priority, followed by tail-chaining to the `PendSV` exception. This means that as long as the QK priority of the preempting task is higher than the preempted task, the asynchronous preemption can keep repeating at higher and higher levels of nesting on the single stack.

---

- (25) Interrupts are unlocked after `QK_schedule` returns. Unlocking interrupts is necessary to cause the synchronous exception `SVCall` in the next line.
- (26) The synchronous exception `SVCall` is entered by means of the `SVC` instruction (`SWI` in traditional ARM/THUMB).
- (27) The job of `SVCall` exception is to discard its own stack frame and cause the interrupt return to the preempted context. The stack contents just before returning from `SVCall` exception is shown below:

```

Hi memory
    xPSR
    pc (interrupt return address)
    lr
    r12
    r3
    r2
    r1
SP --> r0
    xPSR don't care
    PC   don't care
    lr   don't care
    r12  don't care
    r3   don't care
    r2   don't care
    r1   don't care
old SP --> r0   don't care
  
```

Low memory

- (28) The stack pointer is adjusted to un-stack the 8 registers of the interrupt stack frame corresponding to the `SVCall` exception itself.
- (29) `SVCall` exception returns using the special value of `lr` of `0xFFFFFFFF9` (return to Privileged Thread mode using the Main Stack pointer).

## 4.4 Setting up and Starting Interrupts in QF\_onStartup()

Setting up interrupts (e.g., `SysTick`) for the preemptive QK kernel is identical as in the non-preemptive case. Please refer to Section 3.3.2.

## 4.5 Writing ISRs for QK

QK must be informed about entering and exiting every ISR, so that it can perform asynchronous preemptions. You inform the QK kernel about the ISR entry and exit through the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, respectively. You need to call these macros in every ISR. The following listing shows the ISR the file `<qp>\examples\arm-cortex\qk\gnu\dpp-qk-lpcxpresso-1114\bsp.c`.

```
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    QK_ISR_ENTRY(); /* inform QK about ISR entry */
#ifdef Q_SPY
    {
        uint32_t dummy = SysTick->CTRL; /* clear the COUNTFLAG in SysTick */
        QS_tickTime_ += QS_tickPeriod; /* account for the clock rollover */
    }
#endif
    QF_tick();
    QK_ISR_EXIT(); /* inform QK about ISR exit */
}
/*.....*/
void PIOINT0_IRQHandler(void) __attribute__((__interrupt__));
void PIOINT0_IRQHandler(void) {
    QK_ISR_ENTRY(); /* inform QK-nano about ISR entry */
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG)); /* for testing */
    QK_ISR_EXIT(); /* inform QK-nano about ISR exit */
}
```

## 4.6 QK Idle Processing Customization in QK\_onIdle()

QK can very easily detect the situation when no events are available, in which case QK calls the `QK_onIdle()` callback. You can use `QK_onIdle()` to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QK_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 `QK_onIdle()` callback is called with interrupts **unlocked** (which is in contrast to the `QF_onIdle()` callback used in the non-preemptive configuration, see Section 3.4).

The Thumb-2 instruction set used exclusively in ARM-Cortex provides a special instruction `WFI` (Wait-for-Interrupt) for stopping the CPU clock, as described in the “ARMv7-M Reference Manual” [ARM 06a]. The following Listing 8 shows the `QF_onIdle()` callback that puts ARM-Cortex into the idle power-saving mode.

**Listing 8 QK\_onIdle() for the preemptive QK configuration.**

```
(1) void QK_onIdle(void) {
    /* toggle the User LED on and then off, see NOTE01 */
```

```

(2)     QF_INT_LOCK(ignore);
(3)     //GPIOSetValue(LED_PORT, LED_BIT, LED_ON);           /* LED on */
(4)     //GPIOSetValue(LED_PORT, LED_BIT, LED_OFF);          /* LED off */
(5)     QF_INT_UNLOCK(ignore);

(6) #ifdef Q_SPY
      . . .
(7) #elif defined NDEBUG                                     /* sleep mode inteferes with debugging */
      /* put the CPU and peripherals to the low-power mode, see NOTE02
      * you might need to customize the clock management for your application,
      * see the datasheet for your particular ARM-Cortex MCU.
      */
(8)     __WFI();   /* Wait-For-Interrupt */
      #endif
    }

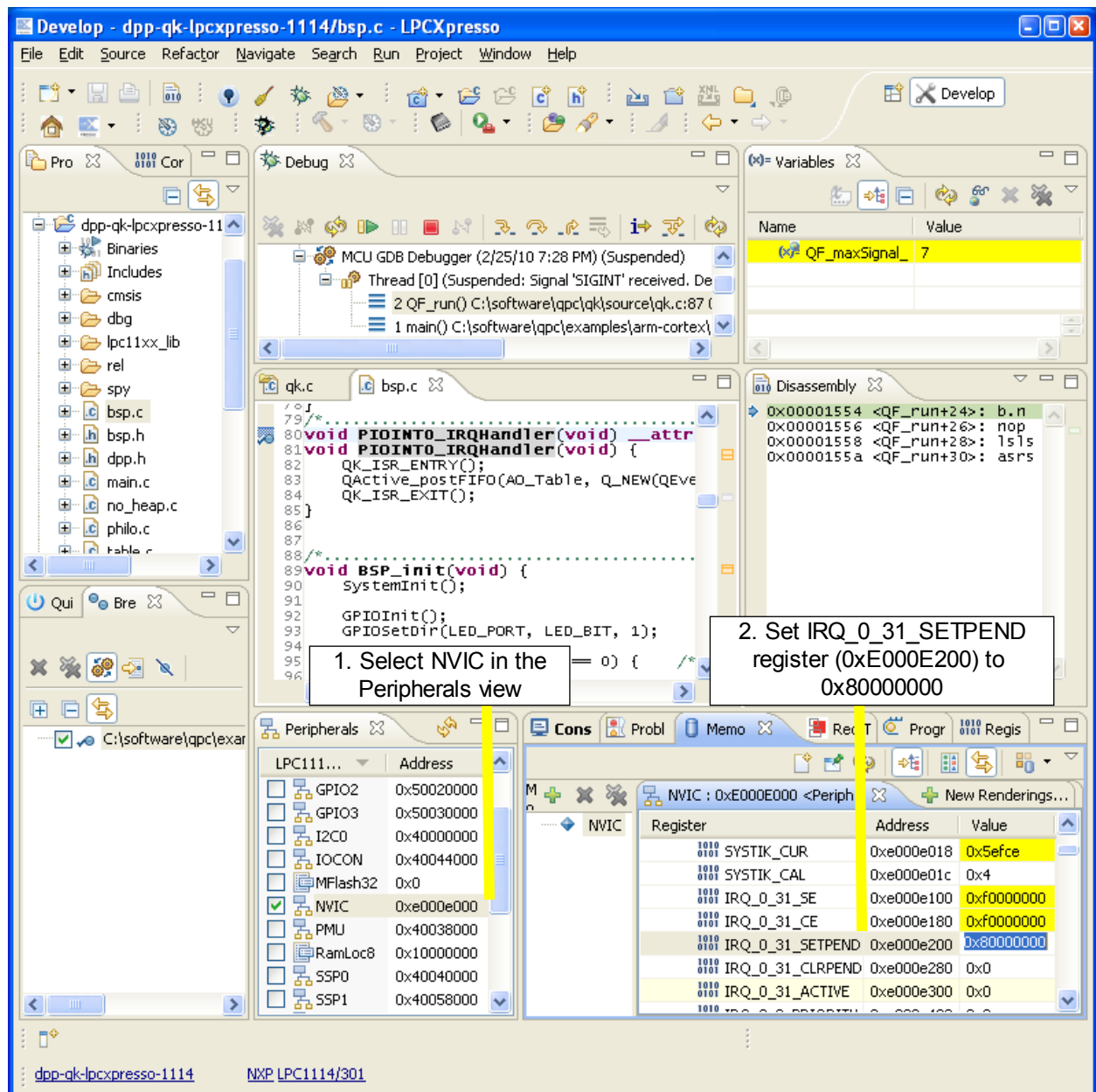
```

- (1) The `QK_onIdle()` function is called with interrupts unlocked.
- (2) The interrupts are locked to prevent preemptions when the LED is on.
- (3-4) Usually, a QK port uses one of the available LEDs to visualize the idle loop activity. However, the LPCXpresso board has only one LED, which is used for other purposes. The code commented out rapidly toggles the LED 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). Please note that the LED is on in the critical section, so the LED intensity does not reflect any ISR or other processing.
- (5) Interrupts are unlocked.
- (6) This part of the code is only used in the QSpy build configuration. In this case the idle callback is used to transmit the trace data using the UART of the ARM-Cortex device.
- (7) The following code is only executed when no debugging is necessary (release version).
- (8) The `WFI` instruction is generated using inline assembly.

## 4.7 Testing QK Preemption Scenarios

The DPP example application includes special instrumentation for convenient testing of various preemption scenarios, such as those illustrated in Figure 9. The technique described in this section will allow you to trigger an interrupt at any machine instruction and observe the preemptions it causes. The interrupt used for the testing purposes is the PIOINT0 interrupt (`INTID == 31`).

Figure 11 Triggering the PIOINT0 interrupt from the Eclipse debugger.



The ISR for this interrupt is shown below:

```
void PIOUSINT0_IRQHandler(void) __attribute__((__interrupt__));
void PIOUSINT0_IRQHandler(void) {
    QK_ISR_ENTRY(); /* inform QK-nano about ISR entry */
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG)); /* for testing */
    QK_ISR_EXIT(); /* inform QK-nano about ISR exit */
}
```

The ISR, as all interrupts in the system, invokes the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, and also posts an event to the `Table` active object, which has higher priority than any of the Philosopher active object.

Figure 11 shows how to trigger the PIOUSINT0 interrupt from the Eclipse debugger. From the debugger you need to first open the register window and select NVIC registers from the drop-down list (see right-bottom corner of Figure 11). You scroll to the STIR register, which denotes the Software Trigger Interrupt Register in the NVIC. This write-only register is useful for software-triggering various interrupts by writing the INTID to it. To trigger the PIOUSINT0 interrupt (INTID == 31) you need to write 0x80000000 to the IRQ\_0\_31\_SETPEND field by clicking on this field, entering the value, and pressing the Enter key.

The general testing strategy is to break into the application at an interesting place for preemption, set breakpoints to verify which path through the code is taken, and trigger the PIOUSINT0 interrupt. Next, you need to free-run the code (don't use single stepping) so that the NVIC can perform prioritization. You observe the order in which the breakpoints are hit. This procedure will become clearer after a few examples.

#### 4.7.1 Interrupt Nesting Test

The first interesting test is verifying the correct tail-chaining to the PendSV exception after the interrupt nesting occurs, as shown in Figure 9(7). To test this scenario, you place a breakpoint inside the `GPIOPortA_IRQHandler()` and also inside the `SysTick_Handler()` ISR. When the breakpoint is hit, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window) and also another breakpoint on the first instruction of the `QK_PendSV` handler. Next you trigger the PIOUSINT0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `PIOUSINT0_IRQHandler()` function, which means that PIOUSINT0 ISR preempted the SysTick ISR.
2. The second breakpoint hit is the one in the `SysTick_Handler()`, which means that the SysTick ISR continues after the PIOUSINT0 ISR completes.
3. The last breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is tail-chained only after all interrupts are processed.

You need to remove all breakpoints before proceeding to the next test.

#### 4.7.2 Task Preemption Test

The next interesting test is verifying that tasks can preempt each other. You set a breakpoint anywhere in the Philosopher state machine code. You run the application until the breakpoint is hit. After this happens, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window). You also place a breakpoint inside the `PIOUSINT0_IRQHandler()` interrupt handler and on the first instruction of the `PendSV_Handler()` handler. Next you trigger the PIOUSINT0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:



1. The first breakpoint hit is the one inside the `PIOINT0_IRQHandler()` function, which means that `PIOINT0` ISR preempted the `Philosopher` task.
2. The second breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the `PendSV` exception is activated before the control returns to the preempted `Philosopher` task.
3. After hitting the breakpoint in QK `PendSV_Handler` handler, you single step into the `QK_scheduler_()`. You verify that the scheduler invokes a state handler from the PED state machine. This proves that the `Table` task preempts the `Philosopher` task.
4. After this you free-run the application and verify that the next breakpoint hit is the one inside the `Philosopher` state machine. This validates that the preempted task continues executing only after the preempting task (the `Table` state machine) completes.

#### 4.7.3 Other Tests

Other interesting tests that you can perform include changing priority of the `PIOINT0` interrupt to be lower than the priority of `SysTick` to verify that the `PendSV` is still activated only after all interrupts complete.

In yet another test you could post an event to `Philosopher` active object rather than `Table` active object from the `PIOINT0_IRQHandler()` function to verify that the QK scheduler will not preempt the `Philosopher` task by itself. Rather the next event will be queued and the `Philosopher` task will process the queued event only after completing the current event processing.

## 5 QS Software Tracing Instrumentation

Quantum Spy (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 (please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” and also to Chapter 11 in [PSiCC2]).

This QDK demonstrates how to use the QS 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 can be configured to send the real-time data out of the serial port of the target device. On the LM3S811 MCU, QS uses the built-in UART to send the trace data out. The EV-LM3S811 board has the UART connected to the virtual COM port provided by the USB debugger (see Figure 1), so the QSPY host application can conveniently receive the trace data on the host PC. The QS platform-dependent implementation is located in the file `bsp.c` and looks as follows:

### Listing 9 QSpy implementation to send data out of the UART0 of the LM3S811 MCU.

```
(1) #ifdef Q_SPY

(2)     #include "uart.h"

(3)     QSTimeCtr QS_tickTime_;
(4)     QSTimeCtr QS_tickPeriod_;

(5)     enum QSDppRecords {
        QS_PHILO_DISPLAY = QS_USER
    };

    /*.....*/
(6) uint8_t QS_onStartup(void const *arg) {
(7)     static uint8_t qsBuf[4*256];          /* buffer for Quantum Spy */
(8)     QS_initBuf(qsBuf, sizeof(qsBuf));

    UARTInit(QS_BAUD_RATE); /*initialize the UART with the desired baud rate*/
    NVIC_DisableIRQ(UART_IRQn); /*do not use the interrupts (QS uses polling)*/
    LPC_UART->IER = 0;

    QS_tickPeriod_ = SystemFrequency / BSP_TICKS_PER_SEC;
    QS_tickTime_ = QS_tickPeriod_;          /* to start the timestamp at zero */

    return (uint8_t)1;                      /* return success */
}
/*.....*/
(9) void QS_onCleanup(void) {
}
/*.....*/
(10) void QS_onFlush(void) {
    uint16_t b;
    while ((b = QS_getByte()) != QS_EOD) { /* while not End-Of-Data... */
        while ((LPC_UART->LSR & LSR_THRE) == 0) { /* while TXE not empty */
        }
        LPC_UART->THR = (b & 0xFF);          /* put into the THR register */
    }
}
```

```

    }
  }
  /*.....*/
(11) QSTimeCtr QS_onGetTime(void) {          /* invoked with interrupts locked */
(12)     if ((HWREG(NVIC_ST_CTRL) & NVIC_ST_CTRL_COUNT) == 0) { /* COUNT no set? */
(13)         return QS_tickTime_ - (QSTimeCtr)SysTick->VAL;
    }
    else { /* the rollover occurred, but the SysTick_ISR did not run yet */
(14)         return QS_tickTime_ + QS_tickPeriod_ - (QSTimeCtr)SysTick->VAL;
    }
  }
}
#endif                                     /* Q_SPY */

```

- (1) The QS instrumentation is enabled only when the macro `Q_SPY` is defined
- (2) The QS implementation uses the UART driver provided in the LPC library.
- (3-4) These variables are used for time-stamping the QS data records. This `QS_tickTime_` variable is used to hold the 32-bit-wide SysTick timestamp at tick. The `QS_tickPeriod_` variable holds the nominal number of hardware clock ticks between two subsequent SysTicks. The SysTick ISR increments `QS_tickTime_` by `QS_tickPeriod_`.
- (5) This enumeration defines application-specific QS trace record(s), to demonstrate how to use them.
- (6) You need to define the `QS_init()` callback to initialize the QS software tracing.
- (7) You should adjust the QS buffer size (in bytes) to your particular application
- (8) You always need to call `QS_initBuf()` from `QS_init()` to initialize the trace buffer.
- (9) The `QS_exit()` callback performs the cleanup of QS. Here nothing needs to be done.
- (10) The `QS_flush()` callback flushes the QS trace buffer to the host. Typically, the function busy-waits for the transfer to complete. It is only used in the initialization phase for sending the QS dictionary records to the host (see please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” and also to Chapter 11 in [PSiCC2])

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

The platform-specific QS port must provide function `QS_onGetTime()` (Listing 9(11)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the ARM-Cortex port uses the SysTick facility, which is the same timer already used for generation of the system clock-tick interrupt.

---

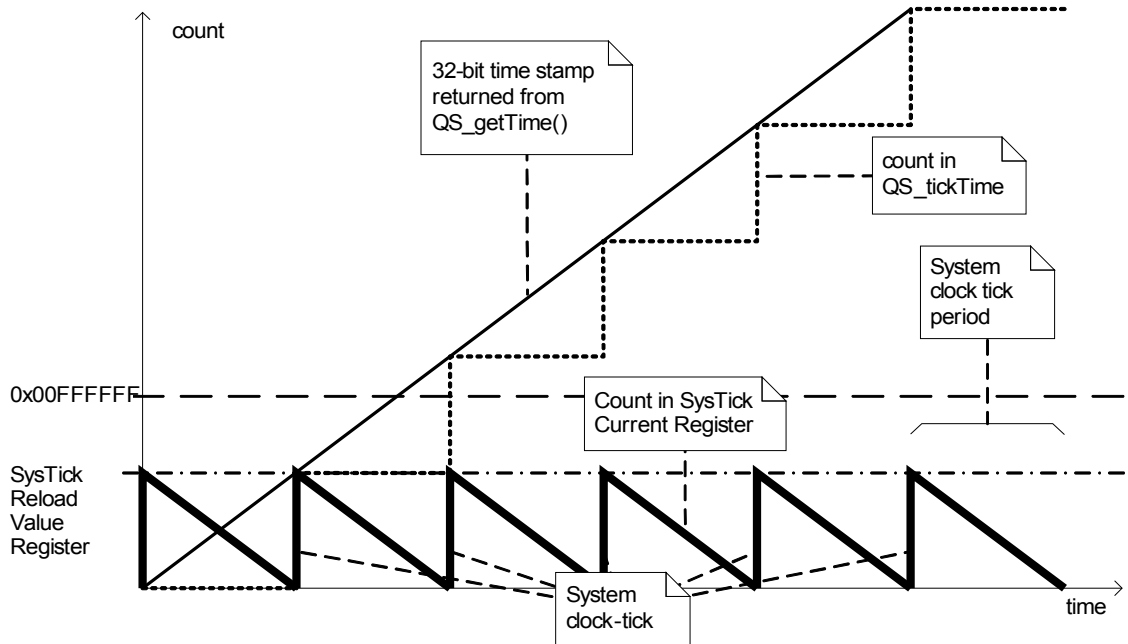
**NOTE:** The `QS_onGetTime()` callback is always called with interrupts locked.

---

Figure 12 shows how the SysTick Current Value Register reading is extended to 32 bits. The SysTick Current Value Register (`NVIC_ST_CURRENT`) counts down from the reload value stored in the SysTick Reload Value Register (`NVIC_ST_RELOAD`). When `NVIC_ST_CURRENT` reaches 0, the hardware automatically reloads the `NVIC_ST_CURRENT` counter from `NVIC_ST_RELOAD` on the subsequent clock tick. Simultaneously, the hardware sets the `NVIC_ST_CTRL_COUNT` flag, which “remembers” that the reload has occurred.

The system clock tick ISR `SysTick_Handler()` keeps updating the “tick count” variable `QS_tickTime_` by incrementing it each time by `QS_tickPeriod_`. The clock-tick ISR also clears the `NVIC_ST_CTRL_COUNT` flag.

**Figure 12 Using the SysTick Current Value Register to provide 32-bit QS time stamp.**



Listing 9(11-15) shows the implementation of the function `QS_onGetTime()`, which combines all this information to produce a monotonic time stamp.

- (12) The `QS_onGetTime()` function tests the `NVIC_ST_CTRL_COUNT`. This flag being set means that the `NVIC_ST_CURRENT` has rolled over to zero, but the SysTick ISR has not run yet (because interrupts are still locked).
- (13) Most of the time the `NVIC_ST_CTRL_COUNT` flag is not set, and the time stamp is simply the sum of `QS_tickTime_ + (-HWREG(NVIC_ST_CURRENT))`. Please note that the `NVIC_ST_CURRENT` register is negated to make it to an up-counter rather than down-counter.
- (13) If the `NVIC_ST_CTRL_COUNT` flag is set, the `QS_tickTime_` counter misses one update period and must be additionally incremented by `QS_tickPeriod_`.

## 5.2 QS Trace Output in `QF_onIdle()/QK_onIdle()`

To be minimally intrusive, the actual output of the QS trace data happens when the system has nothing else to do, that is, during the idle processing. The following code snippet shows the code placed either in the `QF_onIdle()` callback ("Vanilla" port), or `QK_onIdle()` callback (in the QK port):

**Listing 10 QS trace output using the UART0 of the Stellaris LM3S811 MCU**

```
#define UART_TXFIFO_DEPTH 16
...
void QK_onIdle(void) {
    ...
    #ifndef Q_SPY
(1)    if ((LPC_UART->LSR & LSR_THRE) != 0) {                /* is THR empty? */
```

```
uint16_t b;
(2)    QF_INT_LOCK(dummy);
(3)    b = QS_getByte();
(4)    QF_INT_UNLOCK(dummy);
(5)    if (b != QS_EOD) {                                /* not End-Of-Data? */
(6)        LPC_UART->THR = (b & 0xFF);                    /* put into the THR register */
    }
}
#elif defined NDEBUG                                    /* sleep mode interferes with debugging */
    . . .
}
```

- (1) The `LSR_THRE` flag is set when the TX FIFO becomes empty.
- (2) Interrupts are locked to call `QS_getByte()`.
- (3) The function `QS_getByte()` returns the byte or `QS_EOD` (end-of-data) when no data is available.
- (4) The interrupts are unlocked after the call to `QS_getByte()`.
- (5) If the data is available
- (6) The byte is inserted into the TX FIFO.

### 5.3 Invoking the QSpy Host Application

The QSPY host application receives the QS trace data, parses it and displays on the host workstation (currently Windows or Linux). For the configuration options chosen in this port, you invoke the QSPY host application as follows (please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” and also to Chapter 11 in [PSiCC2]):

```
qspy -cCOM5 -b115200 -C4
```



## 6 Related Documents and References

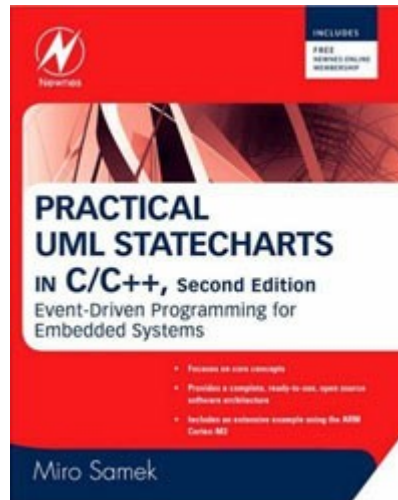
| Document                                                                                                    | Location                                                                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| [PSiCC2] “Practical UML Statecharts in C/C++, Second Edition”, Miro Samek, Newnes, 2008                     | Available from most online book retailers, such as <a href="http://amazon.com">amazon.com</a> .<br>See also: <a href="http://www.state-machine.com/psicc2.htm">http://www.state-machine.com/psicc2.htm</a>                                                         |
| [Samek+ 06b] “Build a Super Simple Tasker”, Miro Samek and Robert Ward, Embedded Systems Design, July 2006. | <a href="http://www.embedded.com/showArticle.jhtml?articleID=190302110">http://www.embedded.com/showArticle.jhtml?articleID=190302110</a>                                                                                                                          |
| [ARM 08a] “ARM v7-M Architecture Application Level Reference Manual”, ARM Limited                           | Available from <a href="http://infocenter.arm.com/help/">http://infocenter.arm.com/help/</a> .                                                                                                                                                                     |
| [ARM 08b] “Cortex™-M3 Technical Reference Manual”, ARM Limited                                              | Available from <a href="http://infocenter.arm.com/help/">http://infocenter.arm.com/help/</a> .                                                                                                                                                                     |
| [CodeSourcery] Sourcery G++ Lite ARM EABI Sourcery G++ Lite Getting Started                                 | <a href="http://www.codesourcery.com/sgpp/lite/arm/portal/doc2861/getting-started.pdf">http://www.codesourcery.com/sgpp/lite/arm/portal/doc2861/getting-started.pdf</a> .                                                                                          |
| [LPC111x] “LPC111x Preliminary user manual Rev. 00.10” — 11 January 2010                                    | Available in PDF from NXP at:<br><a href="http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc1111.lpc1112.lpc1113.lpc1114.pdf">http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc1111.lpc1112.lpc1113.lpc1114.pdf</a> |
| [LPC1343] “LPC1311/13/42/43 User manual Rev. 01.01” — 11 January 2010                                       | Available in PDF from NXP at:<br><a href="http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc13xx.pdf">http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc13xx.pdf</a>                                                 |

## 7 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*

