# Table of Contents

1 Introduction ............................................................................................................................................... 1  
   1.1 About the QP Ports to ARM Cortex-M ...................................................................................... 2  
   1.2 Cortex Microcontroller Software Interface Standard (CMSIS) ........................................... 2  
   1.3 About QP™ .................................................................................................................................. 2  
   1.4 About QM™ .................................................................................................................................. 3  
   1.5 Licensing QP™ ............................................................................................................................. 3  
   1.6 Licensing QM™ ............................................................................................................................. 3  

2 Directories and Files ................................................................................................................................. 4  
   2.1 Building and Debugging the Examples ....................................................................................... 6  

3 Interrupts in the QP Ports to ARM Cortex-M ......................................................................................... 7  
   3.1 “Kernel-Aware” and “Kernel-Unaware” Interrupts .................................................................... 7  
   3.2 Assigning Interrupt Priorities ...................................................................................................... 9  
   3.3 Interrupts and the FPU (Cortex-M4F) .......................................................................................... 10  

4 The Cooperative QV Kernel .................................................................................................................... 11  
   4.1 The qep_port.h Header File ........................................................................................................ 11  
   4.2 The qf_port.h Header File ........................................................................................................... 11  
   4.3 The qv_port.h Header File ........................................................................................................... 13  
   4.4 Writing ISRs for QV .................................................................................................................... 14  
   4.5 Using the FPU in the QV Port (Cortex-M4F) ............................................................................... 15  
   4.6 QV Idle Processing Customization in QV_onIdle() .................................................................... 16  

5 The Preemptive, Run-to-Completion QK Kernel ...................................................................................... 17  
   5.1 Single-Stack, Preemptive Multitasking on ARM Cortex-M ..................................................... 17  
   5.2 The qf_port.h Header File ........................................................................................................... 19  
   5.3 The qk_port.h Header File ........................................................................................................... 20  
   5.4 QK Assembly Code for ARM Cortex-M .................................................................................... 21  
   5.5 Writing ISRs for QK .................................................................................................................... 27  
   5.6 Using the FPU with the preemptive QK kernel (Cortex-M4F) ................................................... 28  
   5.7 QK Idle Processing Customization in QK_onIdle() ................................................................... 28  
   5.8 Testing QK Preemption Scenarios ............................................................................................... 29  

6 The Preemptive, Blocking QXK Kernel .................................................................................................. 32  
   6.1 Traditional, Preemptive Multitasking on ARM Cortex-M ....................................................... 32  
   6.2 The qf_port.h Header File ........................................................................................................... 33  
   6.3 The qkx_port.h Header File ......................................................................................................... 33  
   6.4 QXK Assembly Code for ARM Cortex-M .................................................................................. 34  
   6.5 Writing ISRs for QXK .................................................................................................................. 41  
   6.6 Using the FPU in the QXK Port (Cortex-M4F) .......................................................................... 41  
   6.7 QXK Idle Processing Customization in QXK_onIdle() .............................................................. 42  

7 QS Software Tracing Instrumentation .................................................................................................. 43  
   7.1 QS Time Stamp Callback QS_onGetTime() .............................................................................. 44  
   7.2 QS Trace Output in the Idle Callback ......................................................................................... 45  
   7.3 Invoking the QSpy Host Application ......................................................................................... 46  

8 Related Documents and References ..................................................................................................... 47  

9 Contact Information ............................................................................................................................... 48
1 Introduction

This Application Note describes how to use the QP™ active object frameworks (QP/C, QP/C++, and QP-nano) with the ARM Cortex-M processors (Cortex M0/M0+/M3/M4). Three main implementation options are covered: the cooperative, priority-based QV kernel, the preemptive, run-to-completion QK kernel, and the traditional, preemptive, blocking QXK kernel. All three of these kernels are available in QP. This document assumes QP version 5.6.4 or higher.

To focus the discussion, this Application Note uses the GNU-ARM toolset and the Eclipse-based Code Composer Studio (from Texas Instruments), as well as the EK-TM4C123GXL board (also from Texas Instruments), as shown in Figure 1. However, the general implementation strategy applies equally to all toolsets for ARM Cortex-M, such as ARM-KEIL, IAR EWARM, and TI-ARM. The QP code downloads contain also examples for other boards, such as STM32 Nucleo, NXP mbed-1768, and others.

The provided application examples illustrate also using the QM™ modeling tool for designing QP™ applications graphically and generating code automatically.

Figure 1: The TivaC LaunchPad board (EK-TM4C123GXL) used to test the ARM Cortex-M ports
1.1 About the QP Ports to ARM Cortex-M

In contrast to the “classic” ARM7/ARM9 cores, ARM Cortex-M cores contain such standard components as the Nested Vectored Interrupt Controller (NVIC) and the System Timer (SysTick). With the provision of these standardized components, it is now possible to provide fully portable system-level software for ARM Cortex-M. Therefore, the QP ports to ARM Cortex-M can be much more complete than any ports to the “classic” ARM7/ARM9 and the software is guaranteed to work on any ARM Cortex-M silicon.

**NOTE:** The code listings in this Application Note refer to the GNU-ARM toolset, but this document is applicable to all toolsets for the ARM Cortex-M family, such as ARM-KEIL, IAR EWARM, TI CCS, etc.

**NOTE:** The interrupt disabling policy for ARM-Cortex-M3/M4 has changed in QP 5.1. Interrupts are now disabled more selectively using the BASEPRI register, which allows to disable only interrupts with priorities below a certain level and never disables interrupts with priorities above this level (“zero interrupt latency”). This means that leaving the interrupt priority at the default value of zero (the highest priority) is most likely incorrect, because the free-running interrupts cannot call any QP services. See Section 3 for more information.

1.2 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).

1.3 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++. QP is described in detail in the book “Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems” [PSiCC2] (Newnes, 2008).

**Figure 2:** QP components and their relationship with the target hardware, board support package (BSP), and the application code
As shown in Figure 2, QP consists of a universal UML-compliant event processor (QEP), a portable real-time framework (QF), a choice of build-in real-time kernels, 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.

QP can completely replace a traditional RTOS. QP includes a simple non-preemptive QV kernel, a preemptive, run-to-completion kernel (QK), and the traditional preemptive blocking QXK kernel. 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 and Windows, as well as to ThreadX, uC/OS-II, and embOS.

1.4 About QM™

QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is fully traceable from your design. Please visit state-machine.com/qm for more information about QM™.

The code accompanying this App Note contains several version of the Dining Philosopher Problem [AN-DPP] example for the EK-TM4C123GXL board, all modeled with QM.

1.5 Licensing QP™

The QP active object frameworks are offered under the following licensing options (see www.state-machine.com/licensing):

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file GPL.TXT included in the packaging of every Quantum Leaps software distribution. The GPL open source license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).

- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

1.6 Licensing QM™

The QM™ graphical modeling tool available for download from the www.state-machine.com/downloads website is free to use, but is not open source. During the installation you will need to accept a basic End-User License Agreement (EULA), which legally protects Quantum Leaps from any warranty claims, prohibits removing any copyright notices from QM, selling it, and creating similar competitive products.
2 Directories and Files

The QP ports to ARM Cortex-M are available in the standard QP distribution, which also contains example applications. Specifically, the ports and examples are placed in the following directories:

Listing 1: Directories and files pertaining to the ARM Cortex-M QP ports included in the standard QP distribution.

```
qpc/                      - QP/C directory (qpcpp for QP/C++)
  +-3rd_party/             - 3-rd Party Software
    |+-CMSIS/              - CMSIS files
    |+-ek-tm4c123gxl/      - header files and startup code for EK-TM4C123GXL
    |+-...                 - other 3-rd party boards/RTOSes

  +-include/              - QP public include files
    |+-cassert.h        - QP platform-independent public include
    |+-g.ep.h            - QEP platform-independent public include
    |+-q.f.h             - QF platform-independent public include
    |+-q.k.h             - QK platform-independent public include
    |+-q.v.h             - QV platform-independent public include
    |+-q.kk.h            - QXK platform-independent public include
    |+-q.s.h             - QS platform-independent public include
    |+-...               - QP platform-dependent public include

  +-ports/                - QP ports
    |+-arm-cm/             - ARM-Cortex-M ports
      |+-qv/                - QV ports (cooperative)
      |  |+-arm/              - ARM-KEIL toolset
      |  |+-gnu/              - GNU-ARM toolset
      |  |  |+-qf_port.h       - QF platform-dependent public include
      |  |  |+-qv_port.h       - QV platform-dependent public include
      |  |  |+-qs_port.h       - QS platform-dependent public include
      |  |+-iar/              - IAR toolset
      |  |+-ti/               - TI toolset
      |  |+-qgk/              - QK ports (preemptive, run-to-completion)
      |  |  |+-arm/             - ARM-KEIL toolset
      |  |  |+-gnu/             - GNU-ARM toolset
      |  |  |  |+-qf_port.h     - QF platform-dependent public include
      |  |  |  |+-qk_port.h      - QK platform-dependent public include
      |  |  |  |+-qs_port.h      - QS platform-dependent public include
      |  |  |+-iar/             - IAR toolset
      |  |  |+-ti/              - TI toolset
      |  |+-qgk/              - QXK ports (preemptive, blocking)
      |  |  |+-arm/             - ARM-KEIL toolset
      |  |  |+-gnu/             - GNU-ARM toolset
      |  |  |  |+-qf_port.h     - QF platform-dependent public include
      |  |  |  |+-qkxk_port.h    - QXK platform-dependent public include
      |  |  |  |+-qkxk_port.s    - QXK platform-dependent source code
      |  |  |  |+-qs_port.h      - QS platform-dependent public include
      |  |  |+-iar/             - IAR toolset
      |  |  |+-ti/              - TI toolset
```
2.1 Building and Debugging the Examples

The example applications for ARM Cortex-M have been tested with the EK-TM4C123GXL board (TivaC LaunchPad) from Texas Instruments (see Figure 1) and the GNU-ARM toolset. The examples contain the Makefiles, which can be used standalone, but are also designed to work in Eclipse as makefile-based projects. The provided Makefiles and projects support building the Debug, Release, and Spy configurations.

**NOTE:** The provided Makefiles for building the QP applications assume that the GNU-ARM toolset has been installed in the directory C:/tools/gnu_arm-eabi/. You need to adjust the symbol GNU_ARM at the top of the Makefile to the location of the GNU-ARM installation directory on your system. Alternatively, you can define the GNU_ARM symbol as an environment variable, in which case you don't need to modify the Makefile.

### 2.1.1 Building the Examples from Command Line

The example directory `<qp>/examples/arm-cm/dpp_ek-tm4c123gxl/qk/gnu` 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.

<table>
<thead>
<tr>
<th>Build Configuration</th>
<th>Build command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug (default)</td>
<td>make</td>
</tr>
<tr>
<td>Release</td>
<td>make CONF=rel</td>
</tr>
<tr>
<td>Spy</td>
<td>make CONF=spy</td>
</tr>
<tr>
<td>Clean the Debug configuration</td>
<td>make clean</td>
</tr>
<tr>
<td>Clean the Release configuration</td>
<td>make CONF=rel clean</td>
</tr>
<tr>
<td>Clean the Spy configuration</td>
<td>make CONF=spy clean</td>
</tr>
</tbody>
</table>

### 2.1.2 Building the Examples from Eclipse

The example code contains the Eclipse projects for building and debugging the DPP examples with the CCS IDE from Texas Instruments. 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.

**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.
3 Interrupts in the QP Ports to ARM Cortex-M

All QP frameworks (QP/C, QP/C++, and QP-nano), like all real-time kernels, need to disable interrupts in order to access critical sections of code and re-enable interrupts when done. This section describes the general policy that is used in all built-in real time kernels in QP, such as QV, QK, and QXK (see Figure 2).

3.1 “Kernel-Aware” and “Kernel-Unaware” Interrupts

The QP ports to ARM Cortex-M3/M4 never completely disables interrupts, even inside the critical sections. On Cortex-M3/M4 (ARMv7-M architectures), the QP port disables interrupts selectively using the BASEPRI register. This policy divides interrupts into “kernel-unaware” interrupts, which are never disabled, and “kernel-aware” interrupts, which are disabled in the QP critical sections.

NOTE: The BASEPRI register is not implemented in the ARMv6-M architecture (Cortex-M0/M0+), so Cortex-M0/M0+ need to use the PRIMASK register to disable interrupts globally. In other words, in Cortex-M0/M0+ ports, all interrupts are “kernel-aware”.

Only “kernel-aware” interrupts are allowed to call QP services. “Kernel-unaware” interrupts are not allowed to call any QP services and they can communicate with QP only by triggering a “kernel-aware” interrupt (which can post or publish events).

As illustrated in Figure 3 and Figure 4, the number of interrupt priority bits actually available is implementation dependent, meaning that the various ARM Cortex-M silicon vendors can provide different number of priority bits, varying from just 3 bits (which is the minimum for ARMv7-M architecture) up to 8 bits. For example, the TI Stellaris/Tiva-C microcontrollers implement only 3 priority bits (see Figure 3). On the other hand, the STM32 MCUs implement 4 priority bits (see Figure 4). The CMSIS standard provides the macro __NVIC_PRIO_BITS, which specifies the number of NVIC priority bits defined in a given ARM Cortex-M implementation.

Figure 3: Kernel-aware and kernel-unaware interrupts with 3 priority bits implemented in NVIC

<table>
<thead>
<tr>
<th>Interrupt type</th>
<th>NVIC priority bits</th>
<th>Priority for CMSIS NVIC_SetPriority()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel-unaware interrupt</td>
<td>0 0 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>0 0 1 0 0 0</td>
<td>1 = QF_AWARE_ISR_CMSIS_PRI</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>0 1 0 0 0 0</td>
<td>2</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>0 1 1 0 0 0</td>
<td>3</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>1 0 0 0 0 0</td>
<td>4</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>1 0 1 0 0 0</td>
<td>5</td>
</tr>
<tr>
<td>Kernel-aware interrupt</td>
<td>1 1 0 0 0 0</td>
<td>6</td>
</tr>
<tr>
<td>PendSV interrupt for QK</td>
<td>1 1 1 0 0 0</td>
<td>7</td>
</tr>
</tbody>
</table>

Copyright © Quantum Leaps, LLC. All Rights Reserved.
Another important fact to note is that the ARM Cortex-M core stores the interrupt priority values in the most significant bits of its eight bit interrupt priority registers inside the NVIC (Nested Vectored Interrupt Controller). For example, if an implementation of a ARM Cortex-M microcontroller only implements three priority bits, then these three bits are shifted up to be bits five, six and seven respectively. The unimplemented bits can be written as zero or one and always read as zero.

And finally, the NVIC uses an inverted priority numbering scheme for interrupts, in which priority zero (0) is the highest possible priority (highest urgency) and larger priority numbers denote actually lower-priority interrupts. So for example, interrupt of priority 2 can preempt an interrupt with priority 3, but interrupt of priority 3 cannot preempt interrupt of priority 3. The default value of priority of all interrupts out of reset is zero (0).

**NOTE:** Never leave the priority of any interrupt at the default value.

The CMSIS provides the function NVIC_SetPriority() which you should use to set priority of every interrupt.

**NOTE:** The priority scheme passed to NVIC_SetPriority() is different again than the values stored in the NVIC registers, as shown in Figure 3 and Figure 4 as “CMSIS priorities”
3.2 Assigning Interrupt Priorities

The example projects accompanying this Application Note demonstrate the recommended way of assigning interrupt priorities in your applications. The initialization consist of two steps: (1) you enumerate the “kernel-unaware” and “kernel-aware” interrupt priorities, and (2) you assign the priorities by calling the \texttt{NVIC\_SetPriority()} CMSIS function. \textbf{Listing 2} illustrates these steps with the explanation section following immediately after the code.

\textbf{Listing 2: Assigning the interrupt priorities (see file bsp.c in the example projects).}

```c
/*!!!!!!!!!!!!!!!!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */
/* Assign a priority to EVERY ISR explicitly by calling NVIC\_SetPriority(). */
/* DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE! */
(1) enum KernelUnawareISRs { /* see NOTE0 */
    /* ... */
(2)     MAX\_KERNEL\_UNAWARE\_CMSIS\_PRI /* keep always last */
};
/* "kernel-unaware" interrupts can't overlap "kernel-aware" interrupts */
(3) Q\_ASSERT\_COMPILE(MAX\_KERNEL\_UNAWARE\_CMSIS\_PRI <= QF\_AWARE\_ISR\_CMSIS\_PRI);

(4) enum KernelAwareISRs {
(5)     GPIOPORTA\_PRI = QF\_AWARE\_ISR\_CMSIS\_PRI, /* see NOTE00 */
        SYSTICK\_PRIO,
    /* ... */
(6)     MAX\_KERNEL\_AWARE\_CMSIS\_PRI /* keep always last */
};
/* "kernel-aware" interrupts should not overlap the PendSV priority */
(7) Q\_ASSERT\_COMPILE(MAX\_KERNEL\_AWARE\_CMSIS\_PRI <= (0xFF>>(8-\_NVIC\_PRIO\_BITS)));

(8) void QF\_onStartup(void) {
    /* set up the SysTick timer to fire at BSP\_TICKS\_PER\_SEC rate */
    SysTick\_Config(ROM\_SysCtlClockGet() / BSP\_TICKS\_PER\_SEC);
    /* assign all priority bits for preemption-prio. and none to sub-prio. */
    (9) NVIC\_SetPriorityGrouping(0U);
    /* set priorities of ALL ISRs used in the system, see NOTE00 */
    /* !!!!!!!!!!!!!!!!!!!!!!!!!!!!!! CAUTION !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! */
    /* Assign a priority to EVERY ISR explicitly by calling NVIC\_SetPriority(). */
    /* DO NOT LEAVE THE ISR PRIORITIES AT THE DEFAULT VALUE! */
    (10) NVIC\_SetPriority(SysTick\_IRQn, SYSTICK\_PRIO);
    (11) NVIC\_SetPriority(GPIOPorta\_IRQn, GPIOPORTA\_PRI0);
    /* enable IRQs... */
    (12) NVIC\_EnableIRQ(GPIOPorta\_IRQn);
}
```

(1) \textbf{The enumeration} \texttt{KernelUnawareISRs} \textbf{lists the priority numbers for the "kernel-unaware" interrupts. These priorities start with zero (highest possible). The priorities are suitable as the argument for the NVIC\_SetPriority()} CMSIS function.
NOTE: The NVIC allows you to assign the same priority level to multiple interrupts, so you can have more ISRs than priority levels running as “kernel-unaware” or “kernel-aware” interrupts.

(2) The last value in the enumeration MAX_KERNEL_UNAWARE_CMSIS_PRI keeps track of the maximum priority used for a “kernel-unaware” interrupt.

(3) The compile-time assertion ensures that the “kernel-unaware” interrupt priorities do not overlap the “kernel-aware” interrupts, which start at QF_AWARE_ISR_CMSIS_PRI.

(4) The enumeration KernelAwareISRs lists the priority numbers for the “kernel-aware” interrupts.

(5) The “kernel-aware” interrupt priorities start with the QF_AWARE_ISR_CMSIS_PRI offset, which is provided in the qf_port.h header file.

(6) The last value in the enumeration MAX_KERNEL_AWARE_CMSIS_PRI keeps track of the maximum priority used for a “kernel-aware” interrupt.

(7) The compile-time assertion ensures that the “kernel-aware” interrupt priorities do not overlap the lowest priority level reserved for the PendSV exception (see Section Error: Reference source not found).

(8) The QF_onStartup() callback function is where you set up the interrupts.

(9) This call to the CMSIS function NVIC_SetPriorityGrouping() assigns all the priority bits to be preempt priority bits, leaving no priority bits as subpriorities to preserve the direct relationship between the interrupt priorities and the ISR preemption rules. This is the default configuration out of reset for the ARM Cortex-M3/M4 cores, but it can be changed by some vendor-supplied startup code. To avoid any surprises, the call to NVIC_SetPriorityGrouping(0U) is recommended.

(10-11) The interrupt priorities fall all interrupts (“kernel-unaware” and “kernel-aware” alike) are set explicitly by calls to the CMSIS function NVIC_SetPriority().

(12) All used IRQ interrupts need to be explicitly enabled by calling the CMSIS function NVIC_EnableIRQ().

### 3.3 Interrupts and the FPU (Cortex-M4F)

The QP ports described in this Application Note support also the ARM Cortex-M4F. Compared to all other members of the Cortex-M family, the Cortex-M4F includes the single precision variant of the ARMv7-M Floating-Point Unit (Fpv4-SP). The hardware FPU implementation adds an extra floating-point register bank consisting of S0–S31 and some other FPU registers. This FPU register set represents additional context that need to be preserved across interrupts and task switching (e.g., in the preemptive QK kernel).

The Cortex-M4F has a very interesting feature called lazy stacking [ARM AN298]. This feature avoids an increase of interrupt latency by skipping the stacking of floating-point registers, if not required, that is:

- if the interrupt handler does not use the FPU, or
- if the interrupted program does not use the FPU.

If the interrupt handler has to use the FPU and the interrupted context has also previously used by the FPU, then the stacking of floating-point registers takes place at the point in the program where the interrupt handler first uses the FPU. The lazy stacking feature is programmable and by default it is turned ON.

NOTE: All built-in kernels in QP are designed to take advantage of the lazy stacking feature.
4 The Cooperative QV Kernel

The non-preemptive QV kernel executes active objects one at a time, with priority-based scheduling performed after processing of each event. Due to naturally short duration of event processing in state machines, the simple QV kernel is often adequate for many real-time systems.

In the QV port, the only components requiring platform-specific porting are QF and QV itself. The other two components: QEP and QS require merely recompilation and will not be discussed here. With the QV port you’re not using the QK or QXK kernels. The QV port to ARM Cortex-M is located in the folder <qp>/ports/arm-cm/qv/.

4.1 The qep_port.h Header File

The QEP header file for the ARM Cortex-M port is located in <qp>/ports/arm-cm/qv/gnu/qep_port.h. Listing 3 shows the qep_port.h header file for ARM Cortex-M/GNU. The GNU-ARM compiler is a standard C99 compiler, so it simply includes the <stdint.h> header file that defines the platform-specific exact-with integer types.

Listing 3: The qep_port.h header file for ARM Cortex-M

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

4.2 The qf_port.h Header File

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

The ARM Cortex-M allows you to use the simplest “unconditional interrupt disabling” policy (see Section 7.3.2 in [PSiCC2]), because ARM Cortex-M is equipped with the standard nested vectored interrupt controller (NVIC) and generally runs ISRs with interrupts enabled (so the body of an ISR is not a critical section). Listing 4 shows the qf_port.h header file for ARM Cortex-M with the GNU-ARM toolset. Other toolsets use slightly different conditional compilation macros to select the Cortex-M variants, but implement the same policies.

Listing 4: The qf_port.h header file for ARM Cortex-M/GNU.

```c
/* The maximum number of active objects in the application, see NOTE1 */
(1) #define QF_MAX_ACTIVE           32

/* The maximum number of system clock tick rates */
(2) #define QF_MAX_TICK_RATE        2

/* QF interrupt disable/enable and log2()... */
(3) #if (__ARM_ARCH == 6) /* Cortex-M0/M0+/M1 ?, see NOTE2 */
(4)     #define QF_INT_DISABLE()    __asm volatile("cpsid i")
(5)     #define QF_INT_ENABLE()     __asm volatile("cpsie i")

/* QF-aware ISR priority for CMSIS function NVIC_SetPriority(), NOTE2 */
(6) #define QF_AWARE_ISR_CMSIS_PRI  0
```
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 32, but it can be increased up to the maximum limit of 63 active object priorities in the system.

**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.

The `QF_MAX_TICK_RATE` specifies the maximum number of clock tick rates for QP time events. If you don’t need to specify this limit, in which case the default of a single clock rate will be chosen.

As described in Section 3, the interrupt disabling policy for the ARMv6-M architecture (Cortex-M0/M0+) is different than the policy for the ARMv7-M. The macro `__ARM_ARCH` is defined as 6 for the ARMv6-M architecture (Cortex-M0/M0+), and 7 for ARMv7-M (Cortex-M3/M4/M4F).

(4-5) For the ARMv6-M architecture, the interrupt disabling policy uses the PRIMASK register to disable interrupts globally. The `QF_INT_DISABLE()` macro resolves in this case to the inline assembly instruction “CPSD i”, which sets the PRIMASK. The `QF_INT_ENABLE()` macro resolves to the inline assembly instruction “CPSE i”, which clears the PRIMASK.

For the ARMv6-M architecture, the `QF_AWARE_ISR_CMSIS_PRI` priority level is defined as zero, meaning that all interrupts are “kernel-aware”, because all interrupt priorities are disabled by the kernel.

(7) As described in Section 3, the interrupt disabling policy for the ARMv7-M architecture (Cortex-M3/M4/M4F) uses the BASEPRI register.
For the ARMv7-M architecture, the `QF_SET_BASEPRI()` macro sets the BASEPRI register, using the advanced syntax of the inline GNU assembler. The designation "I" (val) tells the assembler that 'val' is a constant and the designation "cc" r0 informs the compiler that the register r0 is clobbered in this operation.

For the ARMv7-M architecture, the `QF_INT_DISABLE()` macro sets the BASEPRI register to the value specified in `QF_BASEPRI` argument (see step (11) below). The `QF_INT_ENABLE()` macro sets the BASEPRI register to zero, which disables BASEPRI interrupt masking.

The `QF_BASEPRI` value is defined such that it is the lowest priority for the minimum number of 3 priority-bits that the ARM7-M architecture must provide. This partitions the interrupts as "kernel-unaware" and "kernel-aware" interrupts, as shown in Figure 2.

For the ARMv7-M architecture, the `QF_AWARE_ISR_CMSIS_PRI` priority level suitable for the CMSIS function `NVIC_SetPriority()` is determined by the `QF_BASEPRI` value.

The macro `QF_LOG2()` is defined to take advantage of the `CLZ` instruction (Count Leading Zeroes), which is available in the ARMv7-M architecture.

NOTE: The CLZ instruction is not implemented in the ARMv6M architecture. If the `QF_LOG2()` macro is not defined, the QP framework will use the log2 implementation based on a lookup table.

The `QF_CRIT_STAT_TYPE` is not defined, which means that the simple policy of "unconditional interrupt disabling and enabling" is applied (see Section 3).

For critical section entry and exit (the `QF_CRIT_ENTRY()`/`QF_CRIT_EXIT()` macros) are defined in terms of unconditional interrupt disabling and enabling, respectively.

The macro `QF_CRIT_EXIT_NOP()` provides the protection against merging two critical sections occurring back-to-back in the QP code.

### 4.3 The `qv_port.h` Header File

The QV header file for the ARM Cortex-M port is located in `<qp>\ports\arm-cm\qv\gnu\qv_port.h`. This file provides the macro `QV_CPU_SLEEP()`, which specifies how to enter the CPU sleep mode safely in the cooperative QV kernel (see also Section 4.6) and [Samek 07]).

NOTE: To avoid race conditions between interrupts waking up active objects and going to sleep, the cooperative QV kernel calls the `QV_CPU_SLEEP()` callback with interrupts disabled.

Listing 5: The `qv_port.h` header file for ARM Cortex-M

```c
#ifdef ARM_ARCH_V6M /* Cortex-M0/M0+/M1 ? */
    /* macro to put the CPU to sleep inside QV_onIdle() */
    #define QV_CPU_SLEEP() do { \
        __asm volatile("wfi"); \
        QF_INT_ENABLE(); \ 
    } while (0)
#endif /* Cortex-M3/M4/M4F */

    /* macro to put the CPU to sleep inside QV_onIdle() */
    #define QV_CPU_SLEEP() do { \
        __asm volatile("cpsid i"); \ 
    } while (0)
```
QP_INT_ENABLE(); \ 
  __asm volatile ("wfi"); \ 
  __asm volatile ("cpsie i"); \ 
} while (0)

#endif

#include "qv.h" /* QV platform-independent public interface */

For the ARMv6-M architecture, the macro QV_CPU_SLEEP() only stops the CPU with the WFI instruction (Wait For Interrupt). After the CPU is woken up by an interrupt, interrupts are re-enabled with the PRIMASK.

For the ARMv7-M architecture, the macro QV_CPU_SLEEP() first disables interrupts by setting the PRIMASK, then clears the BASEPRI to enable all "kernel-aware" interrupts and only then stops the CPU with the WFI instruction (Wait For Interrupt). After the CPU is woken up by an interrupt, interrupts are re-enabled with the PRIMASK. This sequence is necessary, because the ARM Cortex-M3/M4 cores cannot be woken up by any interrupt blocked by the BASEPRI register.

4.4 Writing ISRs for QV

Even though the ARM Cortex-M CPU is designed to use regular C functions as exception and interrupt handlers, some toolchains require additional considerations. For example, in the GNU-ARM toolchain functions that are used directly as interrupt handlers should be annotated with __attribute__((__interrupt__)). This tells the GNU-ARM 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.

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).

```c
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    ~ ~ ~
    QF_TICK_X(0U, &l_SysTick_Handler); /* 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.
4.5 Using the FPU in the QV Port (Cortex-M4F)

If you have the Cortex-M4F CPU and your application uses the hardware FPU, it should be enabled because it is turned off out of reset. The CMSIS-compliant way of turning the FPU on looks as follows:

```
SCB->CPACR |= (0xFU << 20);
```

**NOTE:** The FPU must be enabled before executing any floating point instruction. An attempt to execute a floating point instruction will fault if the FPU is not enabled.

Depending on whether or not you use the FPU in your ISRs, the “Vanilla” QP port allows you to configure the FPU in various ways, as described in the following sub-sections.

4.5.1 FPU **NOT** used in the ISRs

If you use the FPU only at the task-level (inside active objects) and **none** of your ISRs use the FPU, you can setup the FPU **not** to use the automatic state preservation and **not** to use the lazy stacking feature as follows:

```
FPU->FPCCR &= ~(1U << FPU_FPCCR_ASPEN_Pos) | (1U << FPU_FPCCR_LSPEN_Pos));
```

With this setting, the Cortex-M4F processor handles the ISRs in the exact same way as Cortex-M0-M3, that is, only the standard interrupt frame with R0-R3,R12,LR,PC,xPSR is used. This scheme is the fastest and incurs no additional CPU cycles to save and restore the FPU registers.

**NOTE:** This FPU setting will lead to **FPU errors**, if any of the ISRs indeed starts to use the FPU.

4.5.2 FPU used in the ISRs

If you use the FPU both at the task-level (inside active objects) and in any of your ISRs as well, you should setup the FPU to use the automatic state preservation and the lazy stacking feature as follows:

```
FPU->FPCCR |= (1U << FPU_FPCCR_ASPEN_Pos) | (1U << FPU_FPCCR_LSPEN_Pos);
```

This will enable the “lazy stacking feature” of the Cortex-M4F processor. The the “automatic state saving” and “lazy stacking” are enabled by default, so you typically don’t need to change these settings.

**NOTE:** As described in the ARM Application Note “Cortex-M4(F) Lazy Stacking and Context Switching” [ARM AN298], the FPU automatic state saving requires **more stack** plus additional CPU time to save the FPU registers, but only when the FPU is actually used.
4.6 QV Idle Processing Customization in QV_onIdle()

When no events are available, the non-preemptive QV kernel invokes the platform-specific callback function QV_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 QV_onIdle() must be invoked with interrupts disabled, because the idle condition can be changed by any interrupt that posts events to event queues. QV_onIdle() must internally enable interrupts, ideally atomically with putting the CPU to the power-saving mode (see also [Samek 07] and Chapter 7 in [PSiCC2]).

Because QV_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-M port, the QV_onIdle() takes no parameters. Listing 6 shows an example implementation of QV_onIdle() for the TM4C MCU. Other ARM Cortex-M embedded microcontrollers (e.g., NXP’s LPC1114/1343) handle the power-saving mode very similarly.

Listing 6: QV_onIdle() callback.

(1) void QV_onIdle(void) { /* entered with interrupts DISABLED, see NOTE01 */
(2)  #if defined NDEBUG
(3)     /* Put the CPU and peripherals to the low-power mode */
(4)     #else
(5)     QF_INT_ENABLE(); /* just enable interrupts */
(6)     #endif
(7) }

(1) The cooperative QV kernel calls the QV_onIdle() callback with interrupts disabled, 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 macro QV_CPU_SLEEP() is used to put the CPU to the low-power sleep mode safely. The macro QV_CPU_SLEEP() is defined in the qv_port.h header file for the QV kernel and depends on the interrupt disabling policy used.
(4) When a sleep mode is not used, the QV_onIdle() callback simply re-enables interrupts.
5 The Preemptive, Run-to-Completion QK Kernel

This section describes how to use QP on ARM Cortex-M with the preemptive, run-to-completion 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 QK kernel uses only one stack for all tasks (see the next section), but the stack must be bigger than in the non-preemptive QV kernel. You need to adjust the size of this stack to be large enough for your application.

5.1 Single-Stack, Preemptive Multitasking on ARM Cortex-M

The ARM Cortex-M 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-M.

1. The ARM Cortex-M 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 NMI exception (number 2) to perform asynchronous preemption and return to the preempted task, respectively (see Chapter 10 in [PSiCC2]). The application code (your code) must initialize the Interrupt Vector Table with the addresses of PendSV_Handler and NMI_Handler exception handlers.
4. The application code (your code) must call the function QK_init() to set the priority of the PendSV exception to the lowest level in the whole system (0xFF). The function QK_init() sets the priority of exception 14 to the numerical value of 0xFF. The priority is set with interrupts disabled, but the interrupt status is restored upon the function return. (NOTE: The TivaC ARM Cortex-M 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 (0xFF) to any interrupt in your application. With 3 MSB-bits of priority, 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).
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 5.5).
7. ARM Cortex-M enters interrupt context without disabling interrupts (without setting the PRIMASK bit or the BASEPRI register). Generally, you should not disable interrupts inside your ISRs. In particular, the QF services QF_PUBLISH(), QF_TICK_X(), and QACTIVE_POST() 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-M 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 pushing of the 8 registers comprising the ARM Cortex-M interrupt stack frame upon entry to NMI exception is wasteful in a single-stack kernel (see Figure 5(10) and (12)), but is necessary to perform full interrupt return to the preempted context through the NMI’s return.

5.1.1 Examples of Various Preemption Scenarios in QK

Figure 5 illustrates several preemption scenarios in QK.

Figure 5: Various preemption scenarios in the QK preemptive kernel for ARM Cortex-M.

(0) The timeline in Figure 5 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-M CPU performs tail-chaining to the pending PendSV exception.

(3) The PendSV exception discovers whether there is a higher-priority task to run then the preempted one (by calling QK_schedPrio()) and if so, to to synthesize an exception stack frame to return to the QK scheduler (QK_sched()) to run this new task.
NOTE: The QK scheduler must run in the task context, while PendSV executes in the exception context. The change of the context is accomplished by returning from the PendSV exception directly to the QK scheduler.

To return directly to the QK scheduler, PendSV synthesizes an exception stack frame, which contains the exception return address set to `QK_sched()`. The QK 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 a low-priority 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.

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.

The High-priority ISR sets the pending flag for the PendSV exception by means of the `QK_ISR_EXIT()` macro. 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 an exception return to the preempted Low-priority interrupt, which finally completes.

Upon the exit from the Low-priority ISR, it too sets the pending flag for the PendSV exception by means of the `QK_ISR_EXIT()` macro. The PendSV is already pended from the High-priority interrupt, so pending is again is redundant, but it is not an error. At the ISR return, the ARM Cortex-M CPU performs tail-chaining to the pending PendSV exception.

The PendSV exception synthesizes an interrupt stack frame to return to the QK scheduler. 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 only job of the NMI exception is to discard its own interrupt stack frame, re-enable interrupts, and return using the interrupt stack frame that has been on the stack from the moment of task preemption.

The Low-priority task, which has been preempted all that time, resumes and finally runs to completion and returns to the QK scheduler. The QK scheduler does not find any more tasks to launch and causes the NMI exception to return to the preempted task.

The NMI exception discards its own interrupt stack frame and returns using the interrupt stack frame from the preempted task context.

5.2 The qf_port.h Header File

The QF header file for the ARM Cortex-M port is located in `<qp>\ports\arm-cm\qk\gnu\qf_port.h`. This file is almost identical to the QV port in that the QK port uses the same interrupt disabling policy as the QV port. Please refer to the earlier Section 4.2 for the description of the critical section implementation.
5.3 The qk_port.h Header File

You configure and customize QK through the header file `qk_port.h`, which is located in the QP ports directory `<qp>/ports/arm-cm/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 7: `qk_port.h` header file

```c
/* determination if the code executes in the ISR context */
(1) #define QK_ISR_CONTEXT() (QK_get_IPSR() != (uint32_t)0)

__attribute__((always_inline))
(2) static inline uint32_t QK_get_IPSR(void) {
    uint32_t regIPSR;
    __asm volatile("mrs %0,ipsr" : "=r" (regIPSR));
    return regIPSR;
}

/* QK interrupt entry and exit */
(3) #define QK_ISR_ENTRY() ((void)0)
(4) #define QK_ISR_EXIT()  
    (*((Q_UINT2PTR_CAST(uint32_t, 0xE000ED04U) = (uint32_t)(1U << 28)))
#include "qk.h" /* QK platform-independent public interface */
```

(1) The macro `QK_ISR_CONTEXT()` returns `true` when the code executes in the ISR context and `false` otherwise. The macro takes advantage of the ARM Cortex-M register IPSR, which is non-zero when the CPU executes an exception (or interrupt) and is zero when the CPU is executing thread code.

**NOTE:** QK needs to distinguish between ISR and thread contexts, because threads need to perform synchronous context switch (when a higher-priority task becomes ready to run), while ISRs should not do that.

(2) The inline function `QK_get_IPSR()` obtains the IPSR register and returns it to the caller. This function is defined explicitly for the GNU-ARM toolset, but many other toolsets provide this function as an intrinsic, built-in facility.

(3) The `QK_ISR_ENTRY()` macro notifies QK about entering an ISR. The macro is empty, because the determination of the ISR vs thread context is performed independently in the `QK_ISR_CONTEXT()` macro (see above).

(4) The `QK_ISR_EXIT()` macro set the PENDSV Pend bit [28] in the ICSR register (Interrupt Control and State Register). The register is mapped at address 0xE000ED04 in all ARM Cortex-M cores.

**NOTE:** Triggering the PendSV exception in every ISR is critical for proper operation of QK.

**NOTE:** The PendSV exception is pended unconditionally, even though some interrupts will not make any higher-priority task ready to run. This means that the PendSV exception will sometimes run unnecessarily. However, the call to PendSV is very fast (tail-chaining taking only 6 clock cycles). The PendSV exception checks if a new task needs to be scheduled, and simply returns if not.
5.4 QK Assembly Code for ARM Cortex-M

The QK port to ARM Cortex-M requires coding the PendSV and NMI exceptions in assembly. This ARM Cortex-M-specific code is located in the file `<qp>\ports\arm-cm\qk\gnu\qk_port.s`.

**NOTE:** The single assembly module `qk_port.s` contains common code for all Cortex-M variants (Architecture v6M and v7M) as well as options with and without the VFP. The CPU variants are distinguished by conditional compilation, when necessary.

---

**Listing 8: QK_init() function for ARM Cortex-M (file qk_port.s)**

```assembly
/* The QK_init function sets the priorities of PendSV to 0xFF (lowest).
 * The priority is set within a critical section. */

.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,#8]        /* load the System 12-15 Priority Register */
(6-7)    MOVLS   r3,#0xFF         /* set PRI_14 (PendSV) to 0xFF */
(8)     STR     r2,[r1,#8]        /* write the System 12-15 Priority Register */
(9)     MSR     PRIMASK,r0        /* restore the original PRIMASK */
(10)    BX      lr                /* return to the caller */

.size   QK_init, . - QK_init
```

1. The QK_init() function sets the priorities of the PendSV exception (number 14) the to the lowest level 0xFF. The priority of SVCall exception (number 11) is set to the highest level 0x00 to avoid preemption of this exception.
2. The PRIMASK register is stored in r0.
3. Interrupts are disabled 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 2 (note the offset of 8) is loaded into r2.
6. The mask value of 0xFF0000 is synthesized in r3.
8. The mask is then applied to set the priority byte PRI_14 to 0xFF without changing other priority bytes in this register.
9. The contents of r2 is stored in the NVIC System Handler Priority Register 2 (note the offset of 8).
10. The original PRIMASK value is restored.
11. The function QK_init returns to the caller.
Listing 9: PendSV_Handler() function for ARM Cortex-M (file qk_port.s)

```assembly
.section .text.PendSV_Handler
.global PendSV_Handler    /* CMSIS-compliant exception name */
.type   PendSV_Handler, %function
(1)     .type   PendSV_sched_ret, %function /* to ensure the label is THUMB */

(2) PendSV_Handler:

    .if  __ARM_ARCH == 6        /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
    CPSID   I                 /* disable interrupts at processor level */
    .else                       /* M3/M4/M7 */
    MOVS    r0,#QF_BASEPRI    /* selectively disable interrupts */
    MSR     BASEPRI,r0        /* reset the instruction pipeline */
    .endif                      /* M3/M4/M7 */
    ISB                       /* reset the instruction pipeline */

    .ifdef  __FPU_PRESENT       /* if VFP available... */
    PUSH    {r0,lr}           /* push lr (EXC_RETURN) plus stack "aligner" */
    .endif                      /* VFP available */

    BL      QK_schedPrio_     /* call QK_schedPrio_() */
    CMP     r0,#0             /* is the returned next prio 0? */
    BNE.N   PendSV_sched      /* if next prio != 0, branch to scheduler */

    PendSV_ret:
    .if  __ARM_ARCH == 6        /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
    CPSIE   I                 /* enable interrupts (clear PRIMASK) */
    .else                       /* M3/M4/M7 */
    /* NOTE: r0 == 0 at this point */
    MSR     BASEPRI,r0        /* enable interrupts (clear BASEPRI) */
    .ifdef  __FPU_PRESENT       /* if VFP available... */
    POP     {r0,pc}           /* pop stack "aligner" and EXC_RETURN to PC */
    .else                       /* no VFP */
    MOVS    r0,#6
    MVNS    r0,r0             /* r0 := ~6 == 0xFFFFFFF9 */
    BX      r0                /* exception-return to the task */
    .endif                      /* M3/M4/M7 */

    PendSV_sched:                 /* call the QK scheduler... */
    /* NOTE: The QK scheduler must be called in a task context, while */
    /* we are still in the PendSV exception context. The switch to the */
    /* task context is accomplished by returning from PendSV using a */
    /* fabricated exception stack frame, where the return address is */
    /* the QK scheduler. */
    /* NOTE: the QK scheduler is called with interrupts DISABLED. */
    MOVS    r3,#1
    LSLS    r3,r3,#24         /* r3:=(1 << 24), set the T bit (new xpsr) */
    LDR     r2,=QK_sched_     /* address of the QK scheduler (new pc) */
(25)     LDR     r1,=PendSV_sched_ret /* return address after the call (new lr) */
```

(26)     SUB     sp,sp,#8*4        /* reserve space for exception stack frame */
(27)     STR     r0,[sp]           /* save the prio argument (new r0) */
(28)     ADD     r0,sp,#5*4        /* r0 := 5 registers below the top of stack */
(29)     STM     r0!,{r1-r3}       /* save xpsr,pc,lr */

(30)     MOVs    r0,#6
(31)     MVNS    r0,r0             /* r0 := ~6 == 0xFFFFFFF9 */
(32)     BX      r0                /* exception-return to the QK scheduler */

(33) PendSV_sched_ret:
     /* NOTE: After the QK scheduler returns, we need to resume the preempted */
     /* task. However, this must be accomplished by a return-from-exception, */
     /* while we are still in the task context. The switch to the exception */
     /* context is accomplished by triggering the NMI exception. */
     /* NOTE: The NMI exception is triggered with interrupts DISABLED, */
     /* because QK scheduler disables interrupts before return. */
     /* */
     /* before triggering the NMI exception, make sure that the */
     /* VFP stack frame will NOT be used... */
     
(34)   .ifdef  __FPU_PRESENT       /* if VFP available... */
(35)     MRS     r0,CONTROL        /* r0 := CONTROL */
(36)     BICS    r0,r0,#4          /* r0 := r0 & ~4 (FPCA bit) */
(37)     MSR     CONTROL,r0        /* CONTROL := r0 (clear CONTROL[2] FPCA bit) */
     .endif                      /* VFP available */

     /* trigger NMI to return to preempted task... */
(38)     LDR     r0,=0xE000ED04    /* Interrupt Control and State Register */
(39)     MOVs    r1,#1
(40)     LSLS    r1,r1,#31         /* r0 := (1 << 31) (NMI bit) */
(41)     STR     r1,[r0]           /* ICSR[31] := 1 (pend NMI) */
(42)     B       .                 /* wait for preemption by NMI */

.size   PendSV_Handler, . - PendSV_Handler

(1) The PendSV_sched_ret label is declared as a function so that the GNU-ARM assembler/linker
synthesize the right (THUMB) address for that label. (see also step (16) and NOTE following that
step).

(2) The PendSV_Handler exception is always entered via tail-chaining from the last nested interrupt
(see Section 5.1).

(3) For the ARMv6-M architecture (Cortex-M0/M0+), interrupts are globally disabled by setting
PRIMASK (see Section 3)

(4-5) For the ARMv7-M architecture (Cortex-M3/M4), interrupts are selectively disabled by setting the
BASEPRI register (see Section 3).

NOTE: The value moved to BASEPRI must be identical to the QF_BASEPRI macro defined in
qf_port.h, see Section 3

(6) For the ARMv7-M architecture, when the symbol __FPU_PRESENT is defined...

NOTE: The symbol __FPU_PRESENT must be defined on the command-line to the GNU-ARM
assembler if you intend to use the FPU on the Cortex-M4F cores.
(7) The lr register (EXC_RETURN) is pushed to the stack along with r0, to keep the stack aligned at 8-byte boundary.

**NOTE:** In the presence of the FPU (Cortex-M4F), the EXC_RETURN[4] bit carries the information about the stack frame format used, whereas EXC_RETURN[4] == 0 means that the stack contains room for the S0-S15 and FPSCR registers in addition to the usual R0-R3, R12, LR, PC, xPSR registers. This information must be preserved, in order to properly return from the exception at the end.

(8) The function QK_schedPrio_ is called to find the highest-priority task ready to run. The function is designed to be called with interrupt disabled and returns the priority of this task (in r0), or zero if the currently preempted task is the highest-priority.

(9) The returned priority is tested against zero.

(10) The branch to the QK scheduler (label PendSV_sched) is taken if the priority is not zero.

(11) If the previous branch is not taken, this part of code simply returns from the PendSV exception to the preempted task.

(12) For the ARMv6-M architecture, interrupts are enabled by clearing the PRIMASK.

(13-15) For the ARMv6-M architecture, the constant 0xFFFFFFF9 is fabricated in the r0 register (it is the only EXC_RETURN value possible in this case), and PendSV returns via BX r0.

(16) For the ARMv7-M architecture, interrupts are enabled by setting the BASEPRI register to zero. 
(Please note that r0 must be zero at this point, so MOV r0, #0 is skipped).

(17) If the FPU is used, the EXC_RETURN and the “stack aligner” saved at step (7) are popped from the stack into r0 and pc, respectively. Updating the pc causes the return from PendSV.

(18-20) If the FPU is not used, the constant 0xFFFFFFF9 is fabricated in the r0 register (it is the only EXC_RETURN value possible in this case), and PendSV returns via BX r0.

(21) The PendSV_sched label is reached only when the function QK_schedPrio_ has returned non-zero task priority. This means that the QK scheduler needs to be invoked to call this task and potentially any tasks that nest on it. The call to the QK scheduler must also perform the mode switch from the exception context to the task context.

(22-23) The value (1 << 24) is synthesized in r3. This value is going to be stacked and later restored to xPSR register (only the T bit set).

(24) The address of the QK scheduler function QK_sched_ is loaded into r2. This will be pushed to the stack as the PC register value.

(25) The address of the PendSV_sched_ret label is loaded into r1. This will be pushed to the stack as the lr register value.

**NOTE:** The address of the PendSV_sched_ret label must be a THUMB address, that is, the least-significant bit of this address must be set (this address must be odd number). This is essential for the correct return of the QK scheduler with setting the THUMB bit in the PSR. Without the LS-bit set, the ARM Cortex-M CPU will clear the T bit in the PSR and cause the Hard Fault. The GNU-ARM assembler/linker will synthesize the correct THUMB address of the svc_ret label only if this label is declared with the .type PendSV_sched_ret, %function attribute (see step (1)).

(26) The stack pointer is adjusted to leave room for 8 registers.

(27) The original priority returned in r0 from QK_schedPrio_ is stored on the stack. This will be restored to r0 register value.

(28) The stack is adjusted by 5 registers (r0, r1, r2, r3, and r12) and loaded to r0.
(29) The values of xpsr, pc, and lr prepared in r3, r2, and r1, respectively, are stored on the stack. This operation completes the synthesis of the exception stack frame. After this step the stack looks as follows:

Hi memory

{(optionally S0-S15, FPSCR), if EXC_RETURN[4]==0
  xPSR
  pc (interrupt return address)
  lr
  r12
  r3
  r2
  r1
  r0
EXC RETURN (pushed in Listing 9(7) if FPU is present)
"aligner" (pushed in Listing 9(7) if FPU is present)
  xPSR == 0x01000000
  PC == QK_sched_
  lr == PendSV_sched_ret
  r12 don’t care
  r3 don’t care
  r2 don’t care
  r1 don’t care
  SP --> r0 == priority returned from QK_schedPrio_()

Low memory

(30-31) The special exception-return value 0xFFFFFFFF9 is synthesized in r0 (two instructions are used to make the code compatible with Cortex-M0, which has no barrel shifter). NOTE: the r0 register is used instead of lr because the Cortex-M0 instruction set cannot manipulate the higher-registers (r9-r15).

NOTE: The exception-return value is consistent with the synthesized stack-frame with the lr[4] bit set to 1, which means that the FPU registers are not included in this stack frame.

(32) PendSV exception returns using the special value of the r0 register of 0xFFFFFFFF9 (return to Privileged Thread mode using the Main Stack pointer). The synthesized stack frame causes actually a function call to QK_sched_ function in C.

NOTE: The return from the PendSV exception just executed switches the ARM Cortex-M core to the Privileged Thread mode. The QK_sched_function internally re-enables interrupts before launching any task, so the tasks always run in the Thread mode with interrupts enabled and can be preempted by interrupts of any priority.

NOTE: In the presence of the FPU, the exception-return to the QK scheduler does not change any of the FPU status bit, such as CONTROL.FPCA or LSPACT.

(28) The QK scheduler QK_sched_() returns to the svc_ret label, because this return address is pushed to the stack in step (14). Please note that the address of the svc_ret label must be a THUMB address (see also NOTE after step (15)).

(29) For the ARMv6-M architecture, interrupts are enabled by clearing the PRIMASK.

(30-31) For the ARMv7-M architecture, interrupts are enabled by setting the BASEPRI register to zero.

(32) The following code is assembled conditionally only when the FPU is actually used.
The PendSV_sched_ret label is reached when the QK scheduler returns. Please note that at this point the CPU is still in the Thread mode and the interrupts are disabled.

If the FPU is used, the read-modify-write code clears the CONTROL[2] bit [2]. This bit, called CONTROL.FPCA (Floating Point Active), would cause generating the FPU-type stack frame, which you want to avoid in this case (because the NMI exception will certainly not use the FPU).

NOTE: Clearing the CONTROL.FPCA bit occurs with interrupts disabled, so it is protected from a context switch.

The asynchronous NMI exception is triggered by setting ICSR[31]. The job of this exception is to put the CPU into the exception mode and correctly return to the thread level.

NOTE: The NMI exception is triggered with interrupts disabled, but this exception type is non-maskable, so it can run.

Listing 10: NMI_Handler() function for ARM Cortex-M (file qk_port.s).

```
.section .text.NMI_Handler
.global NMI_Handler    /* CMSIS-compliant exception name */
.type   NMI_Handler, %function

NMI_Handler:
    ADD     sp,sp,#(8*4)      /* remove one 8-register exception frame */
    .if  __ARM_ARCH == 6        /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
        CPSIE   i                 /* enable interrupts (clear PRIMASK) */
        BX      lr                /* return to the preempted task */
    .else                       /* M3/M4/M7 */
        MOVS    r0,#0            /* enable interrupts (clear BASEPRI) */
        MSR     BASEPRI,r0       /* enable interrupts (clear BASEPRI) */
        .ifdef  __FPU_PRESENT      /* if VFP available... */
            POP     {r0,pc}        /* pop stack "aligner" and EXC_RETURN to PC */
        .else                       /* no VFP */
            BX      lr                /* return to the preempted task */
            .endif               /* VFP available */
            .endif               /* M3/M4/M7 */
    .endif                      /* VFP available */
```

The NMI exception is triggered after returning from the QK scheduler. It's job is to discard its own stack frame and cause the exception-return to the original preempted task context. The stack contents just after entering NMI is shown below:

Hi memory

(exampleally S0-S15, FPSCR), if EXC_RETURN[4]==0
xPSR
pc (interrupt return address)
lr
r12
r3
r2
r1
r0

SP --> EXC_RETURN (pushed in Listing 9(7) if FPU is present)
"aligner" (pushed in Listing 9(7) if FPU is present)
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

(2) The stack pointer is adjusted to un-stack the 8 registers of the interrupt stack frame corresponding to the NMI exception itself. This moves the stack pointer from the “old SP” to “SP” in the picture above, which “uncovers” the original exception stack frame left by the PendSV exception.

(3) For the ARMv6-M architecture...

(4) Interrupts are enabled by clearing the PRIMASK.

(5) The NMI exception returns to the preempted task using the standard EXC_RETURN, which is in lr.

(6-7) For the ARMv6-M architecture, interrupts are enabled by writing 0 into the BASEPRI register.

(8-9) If the FPU is used, the EXC_RETURN and the “stack aligner” saved in PendSV step (7) are popped from the stack into r0 and pc, respectively. Updating the pc causes the return from PendSV.

(10) If the FPU is not used, the NMI exception returns to the preempted task using the standard EXC_RETURN, which is in lr.

5.5 Writing ISRs for QK

QK must be informed about entering and exiting every ISR, so that it can perform asynchronous preemption. 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-cm\dpp_ek-tm4c123gxl\qk\bsp.c.

Listing 11: Example ISRs for QK (file bsp.c).

```c
void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    ~ ~ ~
    QK_ISR_ENTRY(); /* inform QK about entering an ISR */
    QF_TICK_X(0U, &l_SysTick_Handler); /* process time events for rate 0 */
    ~ ~ ~
    QK_ISR_EXIT(); /* inform QK about exiting an ISR */
}
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.
5.6 Using the FPU with the preemptive QK kernel (Cortex-M4F)

If you have the Cortex-M4F CPU and your application uses the hardware FPU, it should be enabled because it is turned off out of reset. The CMSIS-compliant way of turning the FPU on looks as follows:

\[
\text{SCB->CPACR} |= (0xFU << 20);
\]

**NOTE:** The FPU must be enabled before executing any floating point instruction. An attempt to execute a floating point instruction will fault if the FPU is not enabled.

Depending on how you use the FPU in your tasks (active objects) and ISRs, the QK QP port allows you to configure the FPU in various ways, as described in the following sub-sections.

5.6.1 FPU used in **ONE** task only and not in any ISRs

If you use the FPU only at a single task (active object) and none of your ISRs use the FPU, you can setup the FPU not to use the automatic state preservation and not to use the lazy stacking feature as follows:

\[
\text{FPU->FPCCR} &= \sim((1U << \text{FPU_FPCCR_ASPEN_Pos}) | (1U << \text{FPU_FPCCR_LSPEN_Pos}));
\]

With this setting, the Cortex-M4F processor handles the ISRs in the exact-same way as Cortex-M0-M3, that is, only the standard interrupt frame with R0-R3,R12,LR,PC,xPSR is used. This scheme is the fastest and incurs no additional CPU cycles to save and restore the FPU registers.

**NOTE:** This FPU setting will lead to **FPU errors**, if more than one task or any of the ISRs indeed start to use the FPU.

5.6.2 FPU used in more than one task or the ISRs

If you use the FPU in more than one of the tasks (active objects) or in any of your ISRs, you should setup the FPU to use the automatic state preservation and the lazy stacking feature as follows:

\[
\text{FPU->FPCCR} |= (1U << \text{FPU_FPCCR_ASPEN_Pos}) | (1U << \text{FPU_FPCCR_LSPEN_Pos});
\]

This is actually the default setting of the hardware FPU and is **recommended for the QK port**, because it is safer in view of code evolution. Future changes to the application can easily introduce FPU use in multiple active objects, which would be unsafe if the FPU context was not preserved automatically.

**NOTE:** As described in the ARM Application Note “Cortex-M4(F) Lazy Stacking and Context Switching” [ARM AN298], the FPU automatic state saving requires **more stack** plus additional CPU time to save the FPU registers, but only when the FPU is actually used.

5.7 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 QV_onIdle() callback used in the non-preemptive configuration, see Section 4.6).

The Thumb-2 instruction set used exclusively in ARM Cortex-M 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 12 shows the QK_onIdle() callback that puts ARM Cortex-M into a low-power mode.
Listing 12: QK_onIdle() for the preemptive QK configuration.

(1) void QK_onIdle(void) {
~ ~ ~
(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 Cortex-M3 MCU.
    */
    __WFI(); /* Wait-For-Interrupt */
    #endif
}

(1) The QK_onIdle() function is called with interrupts enabled.
(2) The following code is only executed when no debugging is necessary (release version).
(3) The WFI instruction is generated using inline assembly.

NOTE: In some ARM Cortex-M MCUs (e.g., STM32), the WFI instruction disables also the JTAG debug port. This renders the MCU unresponsive to JTAG debugger.

5.8 Testing QK Preemption Scenarios

The DPP example application includes special instrumentation for convenient testing of various preemption scenarios, such as those illustrated in Figure 6.

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 GPIOA interrupt (INTID == 0). The ISR for this interrupt is shown below:

```c
void GPIOPortA_IRQHandler(void) {
    QK_ISR_ENTRY(); /* inform QK about entering an ISR */
    QACTIVE_POST(AQ_Table, Q_NEW(QEvt, MAX_PUB_SIG), /* for testing... */
                    &l_GPIOPortA_IRQHandler);
    QK_ISR_EXIT(); /* inform QK about exiting an ISR */
}
```

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 6 shows how to trigger the GPIOA interrupt from the CCS 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 6). You scroll to the NVIC_SW_TRIG register, which denotes the Software Trigger Interrupt Register in the NVIC. This write-only register is useful for software-triggering various interrupts by writing various masks to it. To trigger the GPIOA interrupt you need to write 0x00 to the NVIC_SW_TRIG 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 GPIO 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.
5.8.1 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 5(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 PIOINT0 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 GPIOPortA_IRQHandler() function, which means that GPIO 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 PIOINT0 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.
5.8.2  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 GPIOPortA_IRQHandler() interrupt handler and on the first instruction of the PendSV_Handler() handler. Next you trigger the GPIOA 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 GPIOPortA_IRQHandler() function, which means that GPIO 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 Table 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 preemting task (the Table state machine) completes.

5.8.3  Testing the FPU (Cortex-M4F)

In order to test the FPU, the Board Support Package (BSP) for the Cortex-M4F EK-TM4C123GXL board (see Figure 1) uses the FPU in the following contexts:

- In the idle loop via the QK_onIdle() callback (QP priority 0)
- In the task level via the BSP_random() function called from all five Philo active objects (QP priorities 1-5).
- In the task level via the BSP_displayPhiloStat() function called from the Table active object (QP priority 6)
- In the ISR level via the SysTick_Handler() ISR (priority above all tasks)

To test the FPU, you could step through the code in the debugger and verify that the expected FPU-type exception stack frame is used and that the FPU registers are saved and restored by the “lazy stacking feature” when the FPU is actually used.

Next, you can selectively comment out the FPU code at various levels of priority and verify that the QK context switching works as expected with both types of exception stack frames (with and without the FPU).

5.8.4  Other Tests

Other interesting tests that you can perform include changing priority of the GPIOA 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 GPIOPortA_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.
The Preemptive, Blocking QXK Kernel

This section describes how to use QP on ARM Cortex-M with the preemptive, blocking QXK real-time kernel, which provides typical services of a conventional blocking RTOS (Real-Time Operating System).

QXK has been designed specifically for mixing event-driven active objects with traditional blocking code, such as commercial middleware (TCP/IP stacks, UDP stacks, embedded file systems, etc.) or legacy software.

NOTE: If you are currently using QP on top of a conventional 3rd-party RTOS, consider moving your application to the QXK kernel. QXK provides traditional blocking threads and services, such as semaphores, so you can use traditional blocking code. But QXK is much more tightly integrated with QP than any third-party RTOS, which saves you code space and additional licensing fees.

6.1 Traditional, Preemptive Multitasking on ARM Cortex-M

Traditional multitasking based on blocking threads is quite natural on the ARM Cortex-M architecture, and is implemented for the QXK kernel as follows:

1. The ARM Cortex-M 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. QXK requires every thread (including all active object threads) to have its own private stack space.

3. QXK uses both the Main Stack Pointer (for exceptions and interrupts) and the Process Stack Pointer (for running the threads). This arrangement saves stack space, because the private thread stacks don’t have to provide additional space for interrupts.

4. The QXK port uses the PendSV exception (number 14) to perform context switch. The application code (your code) must initialize the Interrupt Vector Table with the addresses of PendSV_Handler exception handler.

5. The internal function QXK_start(), among other things, sets the priority of the PendSV exception to the lowest level in the whole system (0xFF). The function sets the priority of exception 14 to the numerical value of 0xFF. The priority is set with interrupts disabled, but the interrupt status is restored upon the function return.

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

6. It is strongly recommended that you do not assign the lowest priority (0xFF) to any interrupt in your application. With 3 MSB-bits of priority, 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).

7. Every ISR must call the macros QXK_ISR_ENTRY() upon entry and QXK_ISR_EXIT() right before exiting. This latter macro calls the QXK scheduler to check for thread preemption. If such preemption is detected, the PendSV exception is triggered in the NVIC.

8. ARM Cortex-M enters interrupt context without disabling interrupts (without setting the PRIMASK bit or the BASEPRI register). Generally, you should not disable interrupts inside your ISRs. In particular, the QF services QF_PUBLISH(), QF_TICK_X(), and QACTIVE_POST() should be called with interrupts enabled, to avoid nesting of critical sections.
9. In ARM Cortex-M 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.

6.2 The qf_port.h Header File

The QF header file for the ARM Cortex-M port is located in `<qp>/ports/arm-cm/qxk GNU/qf_port.h`. This file is almost identical to the QK and QV ports in that the QXK port uses the same interrupt disabling policy. Please refer to the earlier Section 4.2 for the description of the critical section implementation.

6.3 The qxk_port.h Header File

You configure and customize QXK through the header file `qxk_port.h`, which is located in the QP ports directory `<qp>/ports/arm-cm/qxk GNU`. The most important function of `qxk_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 13: The qxk_port.h header file for ARM Cortex-M

```c
/* determination if the code executes in the ISR context */
#define QXK_ISR_CONTEXT_() (QXK_get_IPSR() != (uint32_t)0)

__attribute__((always_inline))
static inline uint32_t QXK_get_IPSR(void) {
    uint32_t regIPSR;
    __asm volatile("mrs %0,ipsr" : "=r" (regIPSR));
    return regIPSR;
}

#define QXK_CONTEXT_SWITCH_()
    (*Q_UINT2PTR_CAST(uint32_t, 0xE000ED04U) = (uint32_t)(1U << 28))

/* QXK interrupt entry and exit */
#define QXK_ISR_ENTRY() ((void)0)
#define QXK_ISR_EXIT() do { 
    QF_INT_DISABLE();
    QXK_sched();
    QF_INT_ENABLE();
} while (0)

#include "qxk.h" /* QXK platform-independent public interface */
```

(1) The macro QXK_ISR_CONTEXT() returns `true` when the code executes in the ISR context and `false` otherwise. The macro takes advantage of the ARM Cortex-M register IPSR, which is non-zero when the CPU executes an exception (or interrupt) and is zero when the CPU is executing thread code.
NOTE: QXK needs to distinguish between ISR and thread contexts, because certain operations, such as thread unblocking, can be performed both from ISRs and threads, but when in the thread context they need to additionally call the QXK scheduler.

(2) The inline function QXK_get_IPSR() obtains the IPSR register and returns it to the caller. This function is defined explicitly for the GNU-ARM toolset, but many other toolsets provide this function as intrinsic, built-in facility.

(3) The macro QXK_CONTEXT_SWITCH() encapsulates the CPU-specific context switch implementation. In ARM Cortex-M, this macro triggers the PendSV exception, which performs the context switch.

(4) The QXK_ISR_ENTRY() macro notifies QXK about entering an ISR. The macro is empty, because the determination of the ISR vs thread context is performed independently in the QXK_ISR_CONTEXT() macro (see above).

(5) The QXK_ISR_EXIT() macro runs the QXK scheduler (which must run with interrupts disabled). The scheduler determines if any tasks have been woken up by the ISR, and if so, will set the PENDSV Pend bit [28] in the ICSR register (Interrupt Control and State Register) to perform a context switch.

6.4 QXK Assembly Code for ARM Cortex-M

The QXK port to ARM Cortex-M requires coding the PendSV and NMI exceptions in assembly. This ARM Cortex-M-specific code is located in the file <qp>\ports\arm-cm\qxk\gnu\qxk_port.s.

NOTE: The single assembly module qxk_port.s contains common code for all Cortex-M variants (Architecture v6M and v7M) as well as options with and without the VFP. The CPU variants are distinguished by conditional compilation, when necessary.

Listing 14: QXK_start_() function for ARM Cortex-M (file qxk_port.s)

/*****************************************************************************/
* The QXK_start_ function starts QXK multitasking.
* The C signature: void QXK_start_(void);
* NOTE: QXK_start_() must be called with interrupts disabled and
* returns with interrupts **enabled**.
*****************************************************************************/

.section .text.QXK_start_
.global QXK_start_
.type QXK_start_, %function
.thumb

(1) QXK_start_:  
LDR r1,=0xE000ED18 /* System Handler Priority Register */
LDR r2,[r1,#8]   /* load the System 12-15 Priority Register */
MOV r3,#0xFF
LSL r3,r3,#16
ORR r2,r3       /* set PRI_14 (PendSV) to 0xFF */

(2) STR r2,[r1,#8] /* write the System 12-15 Priority Register */

/* set the MSP to the top of the C-STACK, which is the first entry
* in the vector table. (This recovers any stack used so far by main()).
*/

(3) LDR r0,=0xE000ED08 /* r0 := address of Vector Table Offset reg */
LDR r0,[r0,#0]     /* r0 := contents of VTOR */
LDR r0,[r0] /* r0 := VT[0] (first entry is top of stack) */
(4) MSR MSP,r0 /* main SP := initial top of stack */
ISB /* flush the instruction pipeline */

/* set the current QXK thread to the next QXK thread */
(5) LDR r1,QXK_attr_
(6) LDR r2,[r1,#QXK_NEXT] /* r2 := QXK_attr_.next */
(7) STR r2,[r1,#QXK_CURR] /* QXK_attr_.curr := r2 */

/* get the top of stack of the current QXK thread */
(8) LDR r0,[r2,#QACTIVE_THREAD] /* r0 := QXK_attr_.next->thread (SP) */
/* pop r4-r11 from the PSP */
MOV r1,r0 /* r1 := top of stack */
ADDS r0,r0,#(4*4) /* point r0 to the 4 high registers r7-r11 */
(9) LDMIA r0!,{r4-r7} /* pop the 4 high registers into low regs */
MOV r8,r4 /* move low registers into high registers */
MOV r9,r5
MOV r10,r6
MOV r11,r7
(10) LDMIA r1!,{r4-r7} /* pop the low registers */
/* NOTE: at this point r0 holds the new top of stack */

(11) MSR PSP,r0 /* set PSP to the thread's SP */
ISB /* flush the instruction pipeline */

/* switch CPU to using the Process Stack Pointer (PSP) */
(12) MRS r0,CONTROL
MOV r1,#(1 << 1)
ORRS r0,r0,r1 /* set the Active Stack Pointer */
(13) MSR CONTROL,r0 /* switch to PSP */
DSB /* make sure all data access completes */
ISB /* flush the instruction pipeline */

/* fake return from an exception... */
(14) POP {r0-r3} /* pop R0..R3 */
/* NOTE: R0 holds the 'par' arg of thread fun */
POP {r1,r2} /* pop R12 and LR into low registers r1,r2 */
MOV r12,r1
MOV lr,r2
(15) POP {r1,r2} /* pop PC to R1 and xPSR to R2 */
/* NOTE: it's OK to clobber R1 and R2 */
.if __ARM_ARCH == 6 /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
(16) CPSIE i /* enable interrupts (clear PRIMASK) */
.else /* M3/M4/M7 */
(17) .ifdef __FPU_PRESENT /* if VFP available... */
/* enable the VFP by enabling CP10 and CP11 coprocessors */
LDR.W r3,=0xE000ED88 /* r3 := &CPACR */
LDR r2,[r3] /* r2 := CPACR */
ORR r2,r2,#(0xF << 20) /* r2 := r2 | (CP10 | CP11) */
STR r2,[r3] /* CPACR = r2 */
DSB /* make sure all data access completes */
(19) ISB /* reset the instruction pipeline */
/* enable Automatic State Preservation and Lazy Stacking in the VFP */
(20) LDR r3,=0xE000EF34 /* r3 := &FPCCR */
LDR  r2,[r3]           /* r2 := FPCCR */
ORR  r2,r2,#(3 << 30)  /* set ASPEN | LSPEN */
(21) STR  r2,[r3]           /* FPCCR &= ~(ASPEN | LSPEN) */

/* clear the VFP Context Active (FPCA) bit in CONTROL */
(22) MRS  r2,CONTROL       /* r2 := CONTROL (NOTE: it's OK to clobber R2) */
BICS  r2,r2,#(1 << 2)    /* r2 := r2 & ~(1 << 2) (FPCA bit) */
(23) MSR  CONTROL,r2       /* CONTROL := r2 (clear CONTROL[2] FPCA bit) */
.endif                      /* VFP available */
(24) MOVS r2,#0          /* NOTE: it's OK to clobber R2 */
(25) MSR  BASEPRI,r2      /* enable interrupts (clear BASEPRI) */
.endif                      /* M3/M4/M7 */
(26) BX   r1                /* return to the "interrupted" thread (PC) */

.size  QXK_start_, . - QXK_start_

(1) The internal QXK_start_() function is called from QP_run() to start multitasking. This function is
tered with interrupts disabled and must enable interrupts internally. The function does not return.

(2) The priority of the PendSV exception is set to the lowest level (numerically 0xFF).

(3) The address of the Vector Table is fetched from the Vector Table Offset register.

(4) The first entry in the Vector Table is the top of the C-stack. This value is loaded to the Main Stack
pointer to re-claim any stack space used up to this point.

(5) The address of the QXK_attr_ structure is loaded into r1.

(6-7) The next thread is set to be the current thread.

(8) The top of the stack of the current thread is loaded into r0. The current thread's stack looks as
follows:

    Hi memory
    SP --> r11
        r10
        r9
        r8
        r7
        r6
        r5
        r4
        xPSR
        pc (interrupt return address)
        lr
        r12
        r3
        r2
        r1

    Old SP --> r0  don’t care

    Low memory

(9-10) The registers r4-r11 are restored from the stack frame of the current thread. (NOTE: the code
uses only the instructions available in Cortex-M0).

(11) The Process Stack Pointer (PSP) is set to the new top of the current thread's stack.

(12-13) The active stack pointer is switched to the PSP.
The 8-register exception stack frame is removed from the current stack (PSP). Please note that only the value in \( r0 \) and \( lr \) need to be really preserved and the rest of registers are "don't care". (NOTE: the code uses only the instructions available in Cortex-M0)

The values of the \( pc \) (Program Counter) and the \( xPSR \) from the top of stack are popped into \( r1, r2 \)

For the ARMv6-M architecture, the interrupts are enabled by clearing the PRIMASK

For other architectures (ARMv7-M) and when the FPU is present,

The VFP is enabled by enabling the CP10 and CP11 coprocessors in the CPACR.

The DSP/ISB instructions make sure that all data and instructions complete before proceeding

The VFP is also configured to enable the automatic state preservation and lazy stacking feature in the FPCCR

This read-modify-write code clears the CONTROL[2] bit [2]. This bit, called CONTROL_FPCA (Floating Point Active), would cause un-stacking the FPU-type stack frame, which you don't want when "returning" to the first thread.

For the ARMv7-M architecture, the interrupts are enabled by setting BASEPRI to 0.

The BX r1 instruction "returns" to the pc obtained from first thread, which starts running.

### Listing 15: PendSV_Handler() function for ARM Cortex-M (file qxk_port.s)

```
.global PendSV_Handler    /* CMSIS-compliant exception name */
.type   PendSV_Handler, %function

PendSV_Handler:
    MRS     r0,PSP            /* r0 := Process Stack Pointer */
    ISB                       /* flush the instruction pipeline */

    .if  __ARM_ARCH == 6        /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
    SUBS    r0,r0,#(8*4)      /* make room for 8 registers r4-r11 */
    MOVS    r1,r0             /* r1 := temporary PSP (do not clobber r0!) */
    STmia   r1!,[r4-r7]       /* save the low registers */
    MOV     r4,r8             /* move the high registers to low registers */
    MOV     r5,r9
    MOV     r6,r10
    MOV     r7,r11

    STmia   r1!,[r4-r7]       /* save the high registers */
    /* NOTE: at this point r0 still holds the top of stack */
    CPSID   i                 /* disable interrupts (set PRIMASK) */
    .else                       /* M3/M4/M7 */
    STMDB   r0!,[r4-r11]      /* save r4-r11 on top of the exception frame */
    .ifdef  __FPU_PRESENT     /* if VFP available... */
    TST     lr,#(1 << 4)      /* is it return with the VFP exception frame? */
    IT      EQ                /* if lr[4] is zero... */
    VSTMDBEQ r0!,{s16-s31}   /* ... save VFP registers s16..s31 */
    .endf                      /* VFP available */
    MOVS    r3,#QF_BASEPRI
    MSR     BASEPRI,r3        /* selectively disable interrupts */
    .endf                      /* M3/M4/M7 */
    ISB                       /* reset the instruction pipeline */
    /* un-pend any PendSV pended from the time this PendSV was called... */
```
LDR r2,=0xE000ED04    /* Interrupt Control and State Register */
MOVS r1,#1
LSLS r1,r1,#27        /* r0 := (1 << 27) (UNPENDSVSET bit) */
STR r1,[r2]            /* ICSR[27] := 1 (unpend PendSV) */

/* store the SP of the current QXK thread... */
LDR r1=QXK_attr
LDR r2,[r1,#QXK_CURR]
STR r0,[r2,#QMACTIVE_THREAD] /* QXK_attr_.curr->thread := r0 */

 ifdef __FPU_PRESENT    /* if VFP available... */
 STR lr,[r2,#QMACTIVE_OSOBJECT] /* QXK_attr_.curr->osObject := lr */
 endif /* VFP available */

/* set current to the next... */
LDR r2,[r1,#QXK_NEXT]  /* r2 := QXK_attr_.next */
STR r2,[r1,#QXK_CURR]  /* QXK_attr_.curr := r2 */

 /* restore the SP of the next thread... */
LDR r0,[r2,#QMACTIVE_THREAD] /* r0 := QXK_attr_.next->thread (SP) */

 ifdef __FPU_PRESENT    /* if VFP available... */
 STR lr,[r2,#QMACTIVE_OSOBJECT] /* lr := QXK_attr_.next->osObject */
 endif /* VFP available */

/* exit the critical section */
 ifdef __ARM_ARCH == 6    /* Cortex-M0/M0+/M1 (v6-M, v6S-M)? */
 CPSIE i                  /* enable interrupts (clear PRIMASK) */
 MOVS r1,r0              /* r1 := top of stack */
 ADDS r0,r0,#(4*4)       /* point r0 to the 4 high registers r7-r11 */
 LDMIA r0!,{r4-r7}       /* pop the 4 high registers into low regs */
 MOV r8,r4               /* move low registers into high registers */
 MOV r9,r5
 MOV r10,r6
 MOV r11,r7
 LDMIA r1!,{r4-r7}       /* pop the low registers */
 /* NOTE: at this point r0 holds the new top of stack */
 else /* M3/M4/M7 */
 MOV r3,#0
 MSR BASEPRI,r3          /* enable interrupts (clear BASEPRI) */
 ifdef __FPU_PRESENT    /* if VFP available... */
 TST lr,#(1 << 4)        /* is it return to the VFP exception frame? */
 IT EQ                   /* if lr[4] is zero... */
 VLDMAIEQ r0!,{s16-s31} /* ... restore VFP registers s16..s31 */
 endif /* VFP available */
 LDMIA r0!,{r4-r11}     /* restore r4-r11 from next thread's stack */
 endif /* M3/M4/M7 */
 DSB                      /* make sure all data access completes */
 MSR PSP,r0              /* Process Stack Pointer := r0 */
ISB    / * flush the instruction pipeline */

(35)    BX    lr    / * return to the next thread */
.size    PendSV_Handler, . - PendSV_Handler

(1)   The PendSV_Handler exception is always entered via tail-chaining from the last nested interrupt
(see Section 6.1)

(2)   The Process Stack Pointer (PSP) is moved to r0

(3-7) For the ARMv6-M architecture, the registers r4-r11 are saved to the (PSP) stack of the current
thread. (NOTE: the code uses only the instructions available in Cortex-M0).

Hi memory
SP --> r11
    r10
    r9
    r8
    r7
    r6
    r5
    r4
    xPSR
    pc (interrupt return address)
    lr
    r12
    r3
    r2
    rl
old SP --> r0   don’t care
Low memory

(8)   For the ARMv6-M architecture, interrupts are disabled by setting PRIMASK

(9)   Else (for the ARMv7-M architecture), store r4-r11 on the current stack (using the Cortex-M3/M4
instruction). At this point, the current thread’s stack looks as follows:

(10)  If additionally, the FPU is present, the EXC_RETURN[4] bit is examined to find out whether the FPU
exception stack frame is used.

(11-12) If the FPU stack frame is used the FPU registers s16-s31 are saved on the stack. These
registers complete the already saved FPU registers s0-s15 and FPCSR. At this point, the current
thread’s stack looks as follows:

Hi memory
SP --> s31 {if FPU stack frame is used}
    s30
    s29
    s28
    s27
    s26
    s25
    s24
    s23
    s22
    s21
    s20
s19
s18
s17
s16
r11
r10
r9
r8
r7
r6
r5
r4

empty (if FPU stack frame is used)
FPCSR
s15
s14
s13
s12
s11
s10
s9
s8
s7
s6
s5
s4
s3
s2
s1
s0

xPSR
pc (interrupt return address)
lr
r12
r3
r2
r1

old SP --> r0  don’t care
Low memory

NOTE: The stack frame with the FPU is apparently much bigger than without it. But this big stack frame is used only if the FPU usage has been actually detected in the hardware (EXC_RETURN[4] bit is cleared).

(13-14) For the ARMv7-M architecture, interrupts are disabled by setting BASEPRI to 0.

(15-17) The PendSV exception is un-pended to prevent any unnecessary context switch.

NOTE: The PendSV exception starts with interrupts enabled, so it can always be preempted by an interrupt. Such an interrupt will detect that a context switch is needed, and so it will pend PendSV and return to PendSV. This will cause another instance of PendSV to run just after returning from PendSV, but at this time current thread and next thread will be identical, so a context switch would be unnecessary.

(18-20) The current stack pointer is saved in the TCB of the current thread.
(21) If the FPU is present, the EXC_RETURN (lr) is saved in the "osObject" member of the current thread
(22-23) The QXK current thread is set to the next thread
(24) The r0 register is loaded with the next-thread's top of stack-pointer
(25) If the FPU is present, the EXC_RETURN (lr) is restored from the "osObject" member of the next thread
(26) For the ARMv6-M architecture, the interrupts are enabled by clearing the PRIMASK
(27-28) The registers r4-r11 are restored from next thread's stack. (NOTE: the code uses only the instructions available in Cortex-M0).
(29) Else (for the ARMv7-M architecture), interrupts are enabled by setting BASEPRI to 0.
(30-31) Additionally, if the FPU is present, the EXC_RETURN[4] is tested to find out whether the next thread uses the FPU stack frame.
(32) If so, the FPU registers s16-s31 are restored from the stack. These registers complete the FPU registers s0-s15 and FPCSR that will be restored upon the return from the PendSV exception.
(33) The registers r4-r11 are restored from next thread's stack
(34) The Process Stack Pointer is set to the top of the next thread.
(35) The BX lr instruction returns to the next thread.

6.5 Writing ISRs for QXK

QXK 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 QXK_ISR_ENTRY() and QXK_ISR_EXIT(), respectively. You need to call these macros in every ISR. The following listing shows the ISR the file <qp>/examples/arm-cm/dpp_ek-tm4c123gxl/qxk/bsp.c.

Listing 16: Example ISRs for QXK (file bsp.c).

void SysTick_Handler(void) __attribute__((__interrupt__));
void SysTick_Handler(void) {
    ~ ~ ~
    QXK_ISR_ENTRY(); /* inform QXK about entering an ISR */
    QF_TICK_X(0U, &l_SysTick_Handler); /* process time events for rate 0 */
    ~ ~ ~
    QXK_ISR_EXIT(); /* inform QXK about exiting an ISR */
}

6.6 Using the FPU in the QXK Port (Cortex-M4F)

When the FPU is enabled in the compiler, the QXK_start_() function (Listing 14(17-21)) enables the FPU as well as the Automatic Stack Preservation and the Lazy Stacking in the VFP.
6.7 QXK Idle Processing Customization in QXK_onIdle()

QXK can very easily detect the situation when no events are available, in which case QXK calls the QXK_onIdle() callback. You can use QXK_onIdle() to suspend the CPU to save power, if your CPU supports such a power-saving mode. Please note that QXK_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 QXK_onIdle() callback is called with interrupts unlocked (which is in contrast to the QV_onIdle() callback used in the non-preemptive configuration, see Section 4.6).

The Thumb-2 instruction set used exclusively in ARM Cortex-M 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 shows the QXK_onIdle() callback that puts ARM Cortex-M into the low-power mode.

Listing 17: QXK_onIdle() for the preemptive QXK configuration.

```c
(1) void QXK_onIdle(void) {
    ~ ~ ~
(2) #elif 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 Cortex-M3 MCU. 
         */
(3)     __WFI(); /* Wait-For-Interrupt */
#endif
}
(1) The QXK_onIdle() function is called with interrupts enabled.
(2) The following code is only executed when no debugging is necessary (release version).
(3) The WFI instruction is generated using inline assembly.

NOTE: In some ARM Cortex-M MCUs (e.g., STM32), the WFI instruction disables also the JTAG debug port. This renders the MCU unresponsive to JTAG debugger.
7 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” an 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 18: 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)     static uint8_t qsBuf[4*256]; /* buffer for Quantum Spy */
(7)     UARTInit(QS_BAUD_RATE); /*initialize the UART with the desired baud rate*/
(8)     NVIC_DisableIRQ(UART_IRQn); /*do not use the interrupts (QS uses polling)*/
(9)     LPS_TICK3_PER_SEC = SystemFrequency / BSP_TICKS_PER_SEC;
(10)    QS_tickPeriod_ = LPS_TICK3_PER_SEC;
(11)    QS_tickTime_ = QS_tickPeriod_; /* to start the timestamp at zero */
(12)    return (uint8_t)1; /* return success */
(13)#endif
```

This code initializes the UART, sets the baud rate, and enables polling instead of interrupts for the Quantum Spy. The code then sets the tick period and initializes the timestamp to zero, returning success if the setup is successful.
The QS instrumentation is enabled only when the macro `Q_SPY` is defined.

The QS implementation uses the UART driver provided in the LPC library.

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_`.

This enumeration defines application-specific QS trace record(s), to demonstrate how to use them.

You need to define the `QS_init()` callback to initialize the QS software tracing.

You should adjust the QS buffer size (in bytes) to your particular application.

You always need to call `QS_initBuf()` from `QS_init()` to initialize the trace buffer.

The `QS_exit()` callback performs the cleanup of QS. Here nothing needs to be done.

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” an also to Chapter 11 in [PSiCC2])

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

The platform-specific QS port must provide function `QS_onGetTime()` (Listing 18(11)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the ARM Cortex-M 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 7 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 7: Using the SysTick Current Value Register to provide 32-bit QS time stamp.

Listing 18 (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_`.

7.2 QS Trace Output in the Idle Callback

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 19: QS trace output using the UART0 of the TivaC MCU

```c
#define UART_TXFIFO_DEPTH 16
... void QK_onIdle(void) {
    ...
    #ifdef Q_SPY
```
if ((LPC_UART->LSR & LSR_THRE) != 0) { /* is THR empty? */
    uint16_t b;
    QF_INT_DISABLE();
    b = QS_getByte();
    QF_INT_ENABLE();
    if (b != QS_EOD) { /* not End-Of-Data? */
        LPC_UART->THR = (b & 0xFFU); /* 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.

7.3 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]):

```
quspy –cCOM5
```

The specific COM port obviously depends on how the Tiva-C virtual COM port enumerates on your machine. You might want to open the COM ports in the Device Manager to find out the COM port number.
### 8 Related Documents and References

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
</table>
9 Contact Information

Quantum Leaps, LLC
103 Cobble Ridge Drive
Chapel Hill, NC 27516
USA
+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)
e-mail: info@quantum-leaps.com
WEB: http://www.quantum-leaps.com
http://www.state-machine.com

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

Legal Disclaimers

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

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

All designated trademarks are the property of their respective owners.