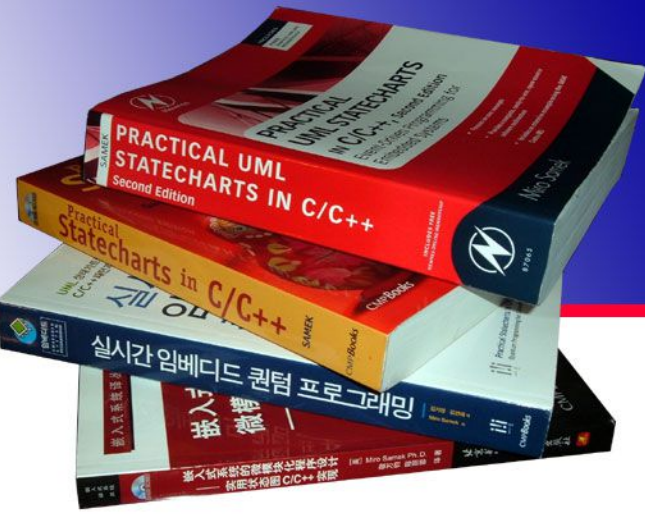
quantum® LeaPs
innovating embedded systems



Application Note
QP™ and ARM Cortex-M with ARM-KEIL

Document Revision D
October 2013

ARM® CONNECTED™

www.quantum-leaps.com
www.state-machine.com

Table of Contents

1 Introduction

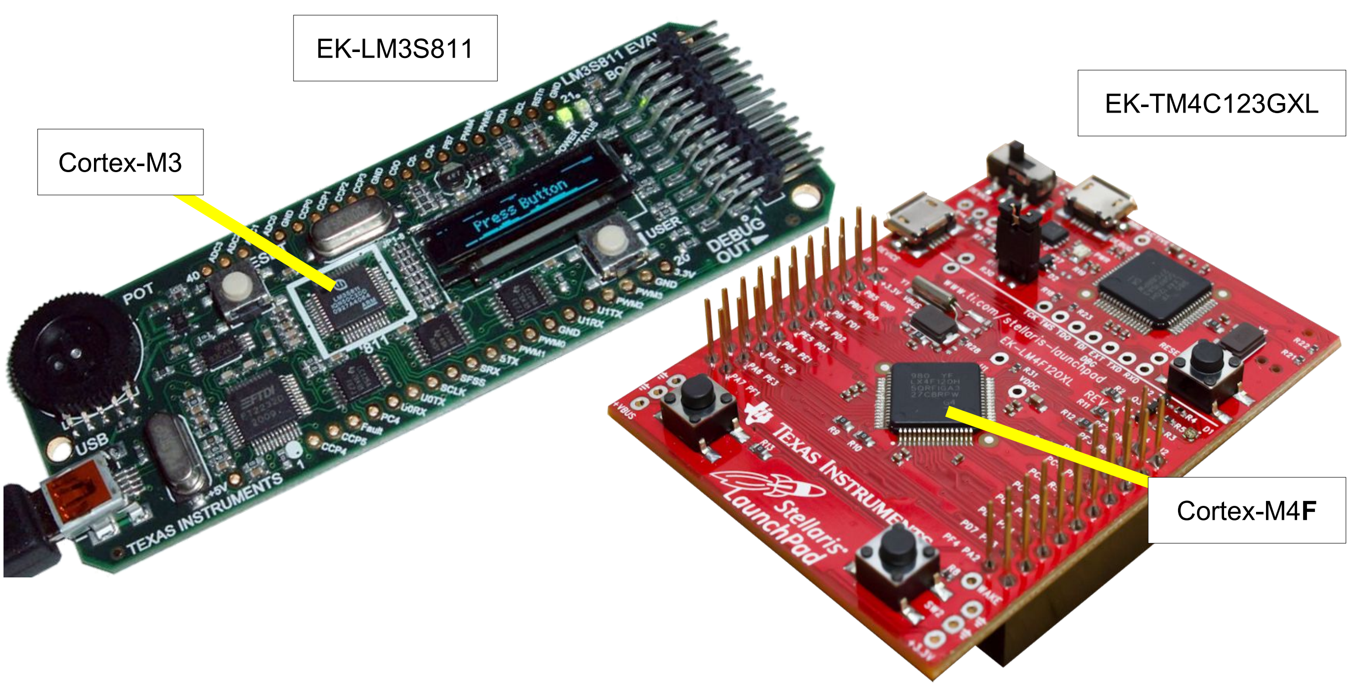
This Application Note describes how to use the QP™ state machine framework with the ARM Cortex-M processors (Cortex M0/M0+/M1/M3/M4 and **M4F** based on the ARMv6-M and ARMv7-M architectures). Two main implementation options are covered: the cooperative "Vanilla" kernel, and the preemptive QK kernel, both available in QP. The port assumes QP version **5.1.1** or higher.

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 **1.1.1** for more information.

To focus the discussion, this Application Note uses the ARM-KEIL toolset (ARMCC version 5.03.0.69) and the µVision4 IDE (MDK-Lite version 4.71.2.0), which are available as a free download from the Keil website) as well as the EK-LM3S811 and EK-TM4C123GXL boards from Texas Instruments, as shown in Figure 1. However, the source code for the QP port described here is generic for all ARM Cortex-M devices and should run without modifications on all ARM Cortex-M cores.

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

Figure 1: The EK-LM3S811 and EK-TM4C123GXL boards used to test the ARM Cortex-M port.



1.1 About the QP Port to ARM Cortex-M

In contrast to the traditional 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 standard components, it is now possible to provide fully portable system-level software for ARM Cortex-M. Therefore, this QP port to ARM Cortex-M can be much more complete than a port to the traditional ARM7/ARM9 and the software is guaranteed to work on any ARM Cortex-M silicon.

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

However, when it comes to handling preemptive multitasking, ARM Cortex-M is a unique processor unlike any other. Section 4 of this application note describes in detail the unique implementation of the preemptive, run-to-completion QK kernel (described in Chapter 10 in [PSiCC2]) on ARM Cortex-M.

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

1.1.1 "Kernel-Aware" and "Kernel-Unaware" Interrupts

Starting from QP 5.1.0, the QP port 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. As shown in Figure 2 and Figure 2, this policy divides interrupts into "kernel-unaware" interrupts, which are never disabled, and "kernel-aware" interrupts, which are disabled in the QP critical sections. **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).

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".

Figure 2: Kernel-aware and kernel-unaware interrupts with 3 priority bits implemented in NVIC

Interrupt type	NVIC priority bits	Priority for CMSIS NVIC_SetPriority()	
Kernel-unaware interrupt	000 00000	0	Never disabled
Kernel-aware interrupt	001 00000	1 = QF_AWARE_ISR_CMSIS_PRI	Disabled in critical sections
Kernel-aware interrupt	010 00000	2	
Kernel-aware interrupt	011 00000	3	
Kernel-aware interrupt	100 00000	4	
Kernel-aware interrupt	101 00000	5	
Kernel-aware interrupt	110 00000	6	
PendSV interrupt for QK	111 00000	7	Should not be used for regular interrupts

Figure 3: Kernel-aware and kernel-unaware interrupts with 4 priority bits implemented in NVIC

Interrupt type	NVIC priority bits	Priority for CMSIS NVIC_SetPriority()	
Kernel-unaware interrupt	**0 0 0 0** 0 0 0 0	**0**	**Never disabled**
Kernel-unaware interrupt	**0 0 0 1** 0 0 0 0	**1**	
Kernel-unaware interrupt	**0 0 1 0** 0 0 0 0	**2**	
Kernel-aware interrupt	**0 0 1 1** 0 0 0 0	**3 = QF_AWARE_ISR_CMSIS_PRI**	Disabled in critical sections
Kernel-aware interrupt	**0 1 0 0** 0 0 0 0	**4**	
Kernel-aware interrupt	**0 1 0 1** 0 0 0 0	**5**	
Kernel-aware interrupt	**0 1 1 0** 0 0 0 0	**6**	
Kernel-aware interrupt	**0 1 1 1** 0 0 0 0	**7**	
. . . .	. . . . .	. . .	
Kernel-aware interrupt	**1 1 1 0** 0 0 0 0	**14**	
Kernel-aware interrupt	**1 1 0 1** 0 0 0 0	**12**	
PendSV interrupt for QK	**1 1 1 1** 0 0 0 0	**15**	Should not be used for regular interrupts

As illustrated in Figure 9 and Figure 3, 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 9). On the other hand, the STM32 MCUs implement 4 priority bits (see Figure 3). 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.

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 9 and Figure 3 as "CMSIS priorities"

1.1.2 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 `NVIC_SetPriority()` CMSIS function. Error: Reference source not found illustrates these steps with the explanation section following immediately after the code.

Listing 1: Assigning the interrupt priorities (see file bsp.c in the example projects).

```
    /*!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! 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 NOTE00 */
        /* ... */
(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);

        /* assing 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_PRIO);
        /* ... */
                                                            /* enable IRQs... */
(12)    NVIC_EnableIRQ(GPIOPortA_IRQn);
    }
```

(1) The enumeration `KernelUnawareISRs` 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 `NVC_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 4.4).

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

(9) This call to the CMIS function `NVIC_SetPriorityGrouping()` assigns all the priority bits to be preempt priority bits, leaving no priority bits as subpriority bits 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 priories 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()`.

1.1.3 The Use of the FPU (Cortex-M4F)

The QP ports described in this Application Note now 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.

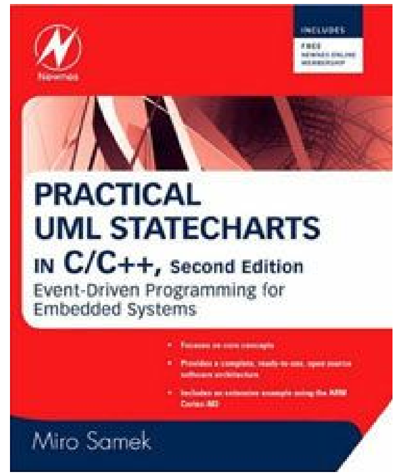
If the interrupt handler has to use the FPU and the interrupted context has also previously used by the FPU, then the stacking of floating-point registers takes place at the point in the program where the interrupt handler first uses the FPU. The lazy stacking feature is programmable and by default it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

NOTE: All QP ports to Cortex-M4F (both the cooperative Vanilla port and the preemptive QK port) are designed to **take advantage of the lazy stacking feature**.

1.1.4 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).

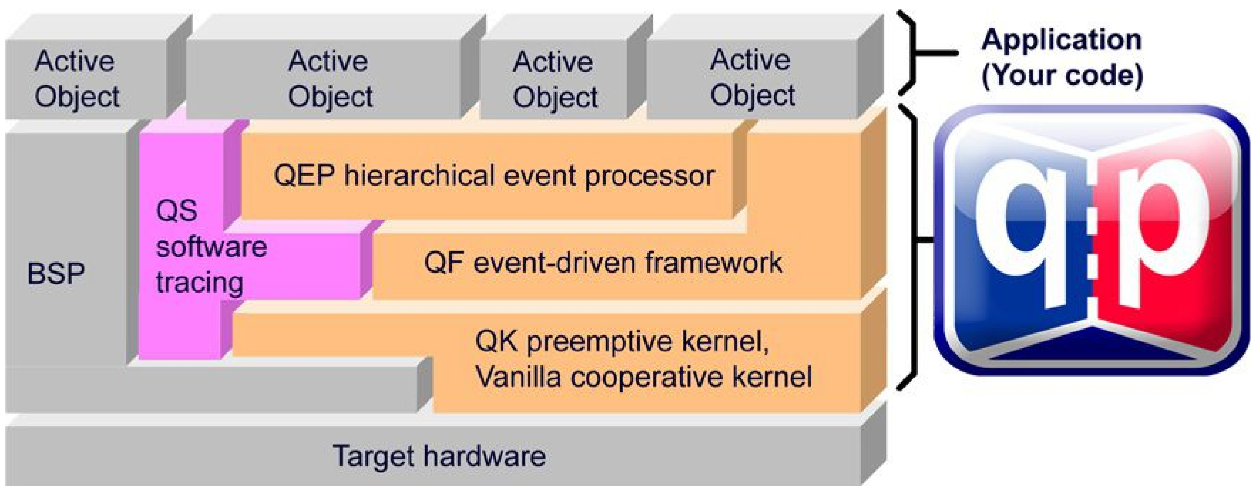
1.2 About QP™



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

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

Figure 4: QP components and their relationship with the target hardware, board support package (BSP), and the application



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

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

1.3 About QM™

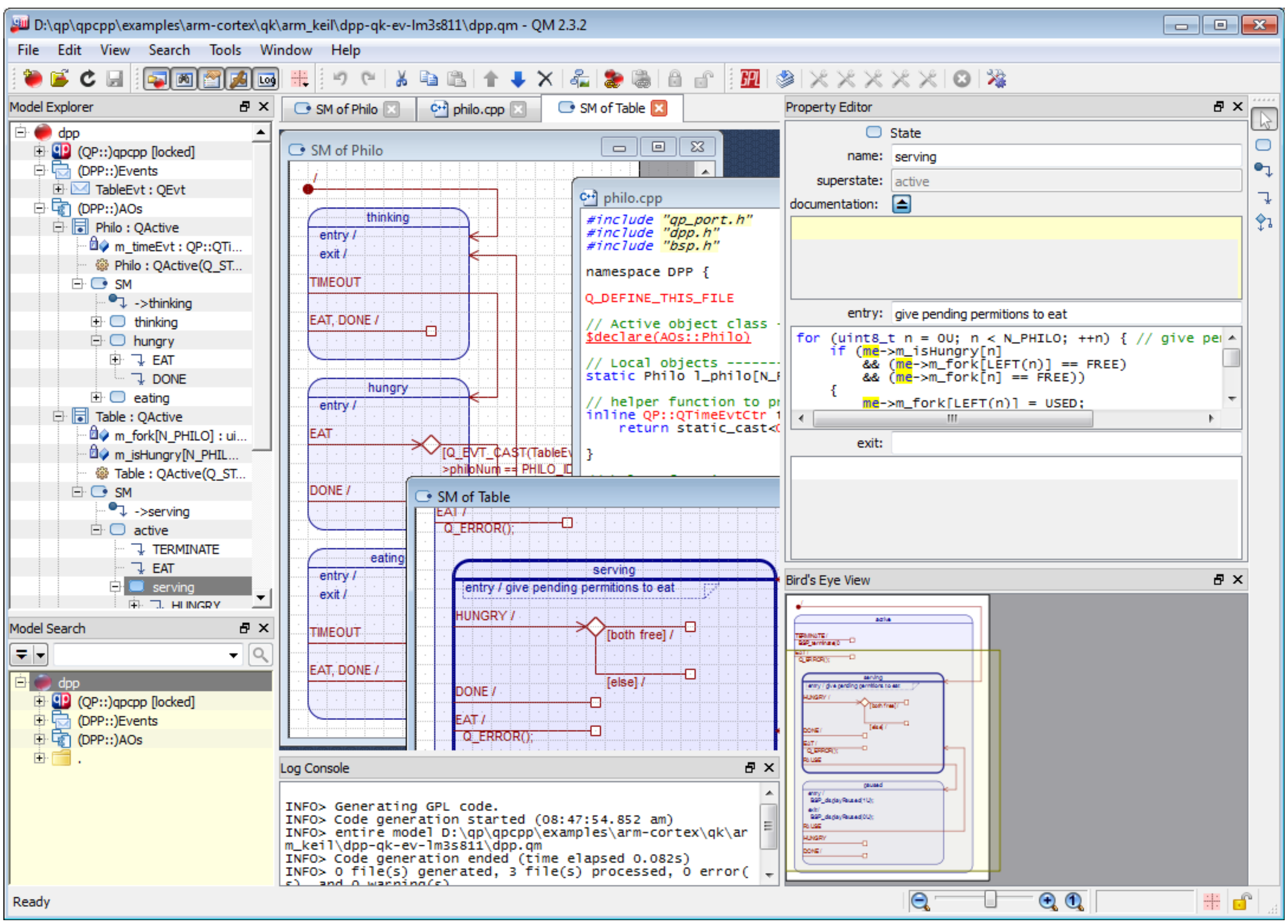
QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

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 100% traceable from your design. Please visit state-machine.com/qm for more information about QM™.

The code accompanying this App Note contains three application examples: the Dining Philosopher Problem [AN-DPP], the PEdestrian LIght CONtrolled [AN-PELICAN] crossing, and the "Fly 'n' Shoot" game simulation for the EK-LM3S811 board (see Chapter 1 in [PSiCC2] all modeled with QM.

NOTE: The provided QM model files assume QM version **2.3.2** or higher.

Figure 5: The DPP example model opened in the QM™ modeling tool



1.4 Licensing QP

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

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

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

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

1.5 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 code for the QP port to ARM Cortex-M with the ARM-KEIL toolset is available in the **standard QP distribution**. Specifically, for this port the files are placed in the following directories:

Listing 2: Directories and files pertaining to the ARM Cortex-M QP port for ARM-KEIL included in the standard QP distribution.

```
qpc/                          - QP/C directory (qpcpp for QP/C++)
  |
  +-include/                  - QP public include files
  | +-qassert.h               - QP platform-independent public include
  | +-qevt.h                  - QEvt declaration
  | +-qep.h                   - QEP platform-independent public include
  | +-qf.h                    - QF platform-independent public include
  | +-qk.h                    - QK platform-independent public include
  | +-qs.h                    - QS platform-independent public include
  | +- . . .
  | +-qp_port.h               - QP platform-dependent public include
  |
  +-ports/                    - QP ports
  | +-arm-cm/                 - ARM-Cortex-M ports
  | | +-cmsis/                - CMSIS (Cortex-M Software Interface Standard)
  | | | +-core_cm0.h
  | | | +-core_cm0plus.h
  | | | +-core_cm3.h
  | | | +-core_cm4.h
  | | | +- . . .
  | | |
  | | +-qk/                   - QK (Quantum Kernel) ports
  | | | +-arm_keil/           - ARM-KEIL compiler
  | | | | +-dbg/              - Debug build
  | | | | | +-libqp_Cortex-M3.a     - QP library for Cortex-M3
  | | | | | +-libqp_Cortex-M4.fp.a - QP library for Cortex-M4F
  | | | | +-rel/              - Release build
  | | | | +-make_Cortex-M3.bat  - Batch file to build QP libraries for Cortex-M3
  | | | | +-make_Cortex-M4.fp.bat- Batch file to build QP libraries for Cortex-M4F
  | | | | +-qep_port.h        - QEP platform-dependent public include
  | | | | +-qf_port.h         - QF platform-dependent public include
  | | | | +-qk_port.h         - QK platform-dependent public include
  | | | | +-qk_port.s         - QK platform-dependent source code
  | | | | +-qs_port.h         - QS platform-dependent public include
  | | |
  | | +-vanilla/              - "vanilla" ports
  | | | +-arm_keil/           - ARM-KEIL compiler
  | | | | +-dbg/              - Debug build
  | | | | | +-libqp_Cortex-M3.a     - QP library for Cortex-M3
  | | | | | +-libqp_Cortex-M4.fp.a - QP library for Cortex-M4F
  | | | | +-rel/              - Release build
  | | | | +-spy/              - Spy build
  | | | | +-make_Cortex-M3.bat  - Batch file to build QP libraries for Cortex-M3
  | | | | +-make_Cortex-M4.fp.bat- Batch file to build QP libraries for Cortex-M4F
  | | | | +-qep_port.h        - QEP platform-dependent public include
  | | | | +-qf_port.h         - QF platform-dependent public include
  | | | | +-qs_port.h         - QS platform-dependent public include
```

```
|
+-examples/              - subdirectory containing the QP example files
| +-arm-cm/              - ARM Cortex-M port
| | +-qk/                - QK examples (preemptive kernel)
| | | +-arm_keil/        - ARM-KEIL compiler
| | | | +-dpp-qk_ek-tm4c123gxl/ - DPP example for EK-TM4C123GXL (Cortex-M4F)
| | | | | +-dbg/         - directory containing the Debug build
| | | | | +-rel/         - directory containing the Release build
| | | | | +-spy/         - directory containing the Spy build
| | | | | +-dpp-qk.uvproj- project file for Keil uVision4
| | | | | +-bsp.c        - Board Support Package for the DPP application
| | | | | +-bsp.h        - BSP header file
| | | | | +-dpp.qm       - the DPP model file for QM
| | | | | +-dpp.h        - the DPP header file
| | | | | +-main.c       - the main function
| | | | | +-philo.c      - the Philosopher active object
| | | | | +-table.c      - the Table active object
| | | | | +-startup-qk_tm4c.s- the startup code in assembly for TM4C MCUs
| | | | |                   (NOTE: the startup code is project-dependent)
| | | | +-dpp-qk_ek-lm3s811/ - Dining Philosophers example for EK-LM3S811
| | | | | +-dbg/         - directory containing the Debug build
| | | | | +-rel/         - directory containing the Release build
| | | | | +-spy/         - directory containing the Spy build
| | | | | +-dpp.uvproj   - project file for Keil uVision4
| | | | | +-lm3s_config.h - CMSIS-compliant configuration for LM3Sxx MCUs
| | | | | +-bsp.c        - Board Support Package for the DPP application
| | | | | +-bsp.h        - BSP header file
| | | | | +-dpp.qm       - the DPP model file for QM
| | | | | +-startup-qk_lm3s.s- the startup code in assembly for LM3S MCUs
| | | | | | . . .
| | | | |
| | | | +-game-qk_ek-lm3s811/ - "Fly 'n' Shoot" game example for EK-LM3S811
| | | | | +-. . .
| | | | | +-game.uvproj   - project file for Keil uVision4
| | | | | +-lm3s_config.h - CMSIS-compliant configuration for LM3Sxx MCUs
| | | | | +-bsp.c        - Board Support Package for this application
| | | | | +-bsp.h        - BSP header file
| | | | | +-game.qm      - the "Fly 'n' Shoot" game model file for QM
| | | | | +-game.h       - the game header file
| | | | | +-main.c       - the main function
| | | | | +-missile.c    - the Missile active object
| | | | | +-ship.c       - the Ship active object
| | | | | +-tunnel.c     - the Tunnel active object
| | | | | +-startup-qk_lm3s.s- the startup code in assembly for LM3S MCUs
| | |
| | +-vanilla/           - "vanilla" examples (non-preemptive scheduler of QF)
| | | +-arm_keil/        - ARM-KEIL compiler
| | | | +-dpp_ek-tm4c123gxl/ - DPP example for EK-TM4C123GXL (Cortex-M4F)
| | | | | | . . .
| | | | +-dpp_ek-lm3s811/ - DPP example for EK-LM3S811
| | | | | | . . .
| | | | +-game_ek-lm3s811/ - Fly 'n' Shoot game for EK-LM3S811
| | | | | | . . .
```

2.1 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for QEP, QF, QS, and QK are provided inside the `<qp>\ports\arm-cm\` directory (see Listing 2). This section describes steps you need to take to rebuild the libraries yourself.

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

The code distribution contains the batch file `make_<core>.bat` for building all the libraries located in the `<qp>\`**`ports\`**`arm-cm\...` directory. For example, to build the debug version of all the QP libraries for the ARM-KEIL compiler, QK kernel, you open a console window on a Windows PC, change directory to `<qp>\`**`ports`**`\arm-cm\qk\arm_keil\`, and invoke the batch by typing at the command prompt the following command:

```
make_Cortex-M3.bat
```

The build process should produce the QP library in the location: `<qp>\`**`ports`**`\arm-cm\qk\arm_keil\-dbg\`. The `make.bat` files assume that the ARM-KEIL ARMCC compiler toolset has been installed in the directory `C:\tools\Keil\ARM\ARMCC`.

NOTE: You need to adjust the symbol `ARM_KEIL` at the top of the batch scripts if you've installed the ARM-KEIL compiler into a different directory.

In order to take advantage of the QS ("spy") instrumentation, you need to build the QS version of the QP libraries. You achieve this by invoking the `make_Cortex-M3.bat` utility with the "spy" target, like this:

```
make_Cortex-M3 spy
```

The make process should produce the QP library in the directory: `<qp>\ports\arm-cm\vanilla\-arm_keil\`**`spy\`**.

You choose the build configuration by providing a target to the `make_Cortex-M3.bat` utility. The default target is "`dbg`". Other targets are "`rel`", and "`spy`" respectively. The following table summarizes the targets accepted by `make_Cortex-M3.bat`.

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

Software Version	Build command
Debug (default)	`make_Cortex-M3` `make_Cortex-M4.fp`
Release	`make_Cortex-M3 rel` `make_Cortex-M4.fp rel`
Spy	`make_Cortex-M3 spy` `make_Cortex-M4.fp spy`

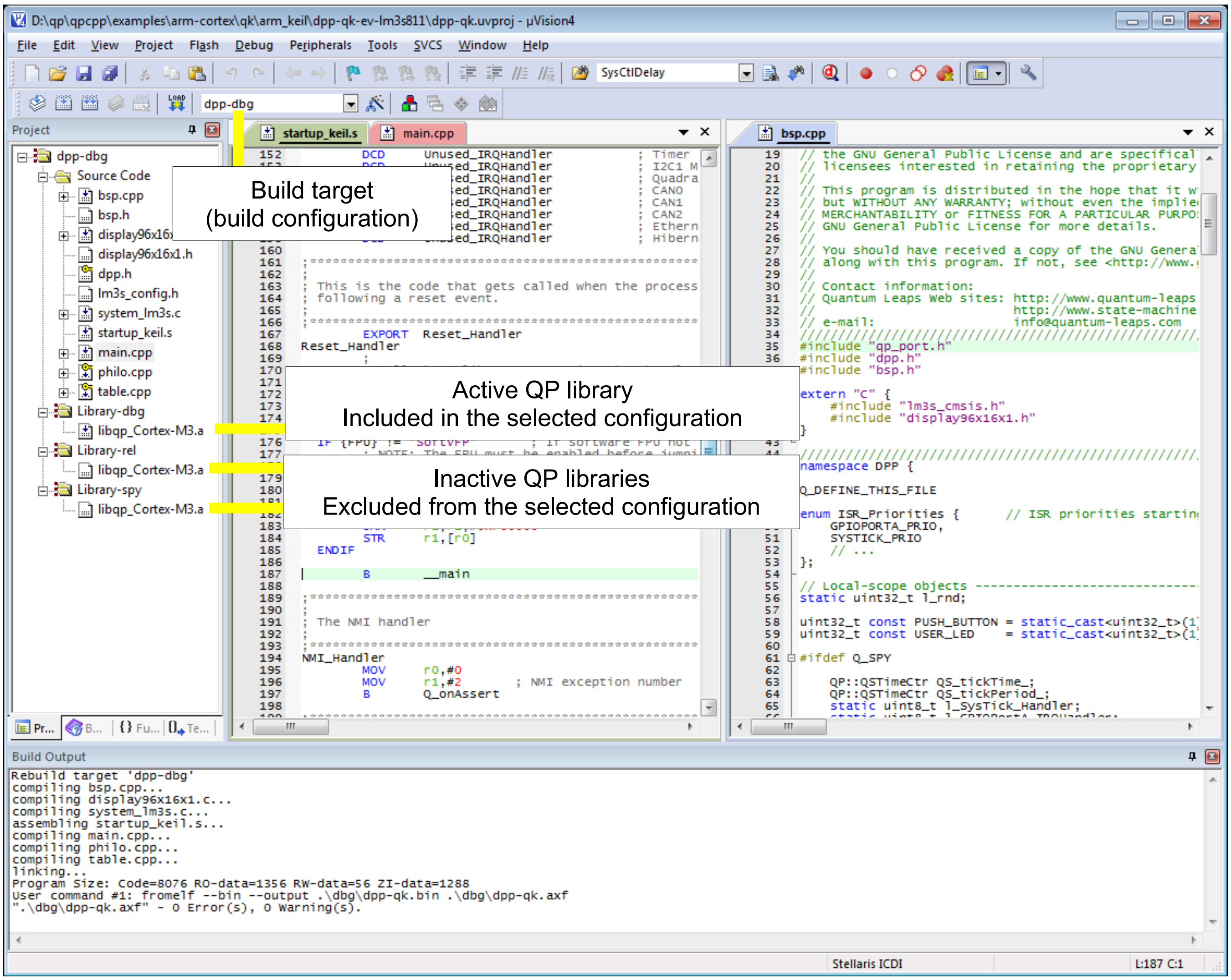
2.2 Building and Debugging the Examples

The example applications for have been tested with the EK-LM3S811 evaluation board from Texas Instruments (see Figure 1) and the ARM-KEIL toolset. The examples contain the Keil uVision project files, so that you can conveniently build and debug the examples from the Keil uVision4. The provided Keil uVision project files support building the Debug, Release, and Spy configurations.

NOTE: The provided uVision4 project files use the relative paths for the QP include directories and the QP libraries. These paths need to be adjusted for your projects, which most likely will have a different relative location with respect to the QP installation directory.

NOTE: The environment variables `QPC` (for the QP/C framework) or `QPCPP` (for the QP/C++ framework) usually used in QP projects are not utilized in the uVision4 project files, because this IDE does not seem to support external environment variables.

Figure 6: Building the DPP example the Keil uVision4 IDE



3 The Cooperative Vanilla Kernel

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

3.1 The qep_port.h Header File

The QEP header file for the port is located in `<qp>\¬ports\¬arm-cm\vanilla\arm_keil\qep_port.h`. Listing 3 shows the `qep_port.h` header file for ARM-KEIL. The ARMCC compiler is a standard C99 compiler, so the standard `<stdint.h>` header file is simply included. This header file defines the platform-specific exact-with integer types.

Listing 3: The `qep_port.h` header file for ARM-KEIL.

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

3.2 The QF Port Header File

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

The most important porting decision you need to make in the `qf_port.h` header file is the policy for disabling and enabling interrupts. The allows using the simplest “unconditional interrupt enabling” policy (see Section 7.3.2 of the book “Practical UML Statecharts in C/C++, Second Edition” [PSiCC2]), because is equipped with the standard nested vectored interrupt controller (NVIC) and generally runs ISRs with interrupts unlocked. Listing 4 shows the `qf_port.h` header file for ARM-KEIL.

Listing 4 The `qf_port.h` header file for ARM_KEIL.

```
                 /* The maximum number of active objects in the application */
(1)  #define QF_MAX_ACTIVE                   32
                                  /* The number of system clock tick rates */
(2)  #define QF_MAX_TICK_RATE                2

(3)  #if (__TARGET_ARCH_THUMB == 3) /* Cortex-M0/M0+/M1(v6-M, v6S-M)?, see NOTE2 */

(4)      #define QF_INT_DISABLE()         __disable_irq()
(5)      #define QF_INT_ENABLE()          __enable_irq()

         /* QF-aware ISR priority for CMSIS function NVIC_SetPriority(), NOTE2   */
(6)      #define QF_AWARE_ISR_CMSIS_PRI  0

                                /* macro to put the CPU to sleep inside QF_idle() */
(7)      #define QF_CPU_SLEEP() do { \
            __wfi(); \
            QF_INT_ENABLE(); \
         } while (0)
```

```
 (8) #else                                    /* Cortex-M3/M4/M4F, see NOTE3 */

 (9)     #define QF_SET_BASEPRI(val)      __asm("msr BASEPRI," #val)
(10)     #define QF_INT_DISABLE()         QF_SET_BASEPRI(QF_BASEPRI)
(11)     #define QF_INT_ENABLE()          QF_SET_BASEPRI(0U)

         /* NOTE: keep in synch with the value defined in "qk_port.s", see NOTE4 */
(12)     #define QF_BASEPRI                (0xFF >> 2)

         /* QF-aware ISR priority for CMSIS function NVIC_SetPriority(), NOTE5   */
(13)     #define QF_AWARE_ISR_CMSIS_PRI  (QF_BASEPRI >> (8 - __NVIC_PRIO_BITS))

                               /* macro to put the CPU to sleep inside QF_idle() */
(14)     #define QF_CPU_SLEEP() do { \
             __disable_irq(); \
             QF_INT_ENABLE(); \
             __wfi(); \
             __enable_irq(); \
         } while (0)

              /* Cortex-M3/M4/M4F provide the CLZ instruction for fast LOG2 */
(15)     #define QF_LOG2(n_) ((uint8_t)(32U - __clz(n_)))

     #endif

(16) /* QF_CRIT_STAT_TYPE not defined: unconditional interrupt unlocking" policy */
(17) #define QF_CRIT_ENTRY(dummy)          QF_INT_DISABLE()
(18) #define QF_CRIT_EXIT(dummy)           QF_INT_ENABLE()
(19) #define QF_CRIT_EXIT_NOP()            __nop()

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

(1) The `QF_MAX_ACTIVE` specifies the maximum number of active object priorities in the application. You always need to provide this constant. Here, `QF_MAX_ACTIVE` is set to 32 to save some memory. You can increase this limit 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.

---

(2) 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.

(3) As described in Section 1.1.1, the interrupt disabling policy for the ARMv6-M architecture (Cortex-M0/M0+) is different than the policy for the ARMv7-M. The preprocessor macro `__TARGET_ARCH_THUMB` is defined by the ARMCC toolset based on the `--cpu` setting to the compiler [ARMCC]. `__TARGET_ARCH_THUMB` is set to 3 for the v6-M and v6S-M architectures, which do not support the `CLZ` instruction..

(4) 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 intrinsic Keil function `__disable_irq()`, which in turn generates the single "`CPSD i`" Thumb2 instruction.

(5) For the ARMv6-M architecture, the `QF_INT_ENABLE()` macro resolves to the intrinsic Keil function `__enable_irq()`, which in turn generates the single "`CPSE i`" Thumb2 instruction.

(6) 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) The macro `QF_CPU_SLEEP()` specifies how to enter the CPU sleep mode safely in the cooperative Vanilla kernel (see also Section 3.6). For the ARMv6-M architecture, the macro `QF_CPU_SLEEP()` first stops the CPU with the WFI instruction (Wait For Interrupt) and after the CPU is woken up by an interrupt, re-enables interrupts with the PRIMASK. This is possible, because the ARM Cortex-M CPU can be woken up by an interrupt, even though PRIMASK is set.

(8) As described in Section 1.1.1, the interrupt disabling policy for the ARMv7-M architecture (Cortex-M3/M4/M4F) uses the BASEPRI register.

(9) For the ARMv7-M architecture, the `QF_SET_BASEPRI()` macro `sets the BASEPRI register`, which sets the BASEPRI register to the value specified in the macro argument '`val`'.

NOTE: The "`msr BASEPRI,`" `#val` is really a pseudo-instruction. The real MSR cannot load an immediate argument, but rather must use (clobber) a register. The inline Keil assembler "knows" that the value needs to be first moved into a register, and only then the register can be used in MSR.

---

(10) 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).

(11) For the ARM7-M architecture, the `QF_INT_ENABLE()` macro sets the BASEPRI register to zero, which disables BASEPRI interrupt masking.

(12) 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 9 and Figure 2.

(13) 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.

(14) The macro `QF_CPU_SLEEP()` specifies how to enter the CPU sleep mode safely in the cooperative Vanilla kernel (see also Section 3.6). For the ARMv7-M architecture, the macro `QF_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.

(15) 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 Cortex-M0/M0+/M1 (ARMv6M architecture). If the `QF_LOG2()` macro is not defined, the QP framework will use the log2 implementation based on a lookup table.

---

(16) The `QF_CRIT_STAT_TYPE` is not defined, which means that the simple policy of "unconditional interrupt locking and unlocking" is applied.

(17) The critical section entry macro disables interrupts by the policy established above

(18) The critical section exit macro re-enables interrupts by the policy established above

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

3.3 Handling Interrupts in the Non-Preemptive Vanilla Kernel

ARM Cortex-M has been specifically designed to enable writing ISRs as plain C-functions, without any special interrupt entry or exit requirements. These ISRs are perfectly adequate for the non-preemptive Vanilla kernel.

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 5).

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

```
void SysTick_Handler(void) {
    . . .
    QF_TICK_X(0U, &l_SysTick_Handler);     /* process all armed time events */
    . . .
}
```

3.3.1 The Interrupt Vector Table

The CMSIS-compliant file startup_<mcu>.s assembly module (where <mcu> stands for the specific MCU type, such as `tm4c` or `lm3s`) contains an interrupt vector table (also called the exception vector table) starting usually at address 0x00000000, typically in ROM. The vector table contains the initialization value for the main stack pointer on reset, and the entry point addresses for all exception handlers. The exception number defines the order of entries in the vector table.

ARM-Cortex-M architecture requires you to place the initial Main Stack pointer and the addresses of all exception handlers and ISRs into the Interrupt Vector Table allocated typically in ROM. In the ARM-KEIL toolset, the IDT is initialized in the `__main` startup code provided with the ARM-KEIL toolset.

NOTE: The example startup code that ships with the ARM-KEIL uVision4 toolset does **not** comply with the CMSIS standard for the names of the exception handlers and IRQ handlers. The startup code presented below has been modified by Quantum Leaps to use the CMSIS-compliant names of the exception handlers and IRQ handlers.

Listing 5: The interrupt vector table defined in startup_<mcu>.s (ARMASM assembler).

```
         ~ ~ ~ ~
(1)      AREA   RESET, CODE, READONLY
         THUMB

(2)      IMPORT assert_failed
(3)      ;IMPORT SVC_Handler
(4)      ;IMPORT PendSV_Handler

         ; add any used interrupt handlers here and in the right
         ; slot of the vector table below...
(5)      IMPORT SysTick_Handler
(6)      IMPORT GPIOPortA_IRQHandler

     ;******************************************************************************
     ; The vector table.
```

```
;*****************************************************************************
             EXPORT   __Vectors
(7)  __Vectors
(8)          DCD      StackMem + Stack              ; Top of Stack
(9)          DCD      Reset_Handler                 ; Reset Handler
             DCD      NMI_Handler                   ; NMI Handler
             DCD      HardFault_Handler             ; Hard Fault Handler
             DCD      MemManage_Handler             ; The MPU fault handler
             DCD      BusFault_Handler              ; The bus fault handler
             DCD      UsageFault_Handler            ; The usage fault handler
             DCD      0                             ; Reserved
             ~ ~ ~ ~
             DCD      SVC_Handler                   ; SVCall handler
             DCD      DebugMon_Handler              ; Debug monitor handler
             DCD      0                             ; Reserved
             DCD      PendSV_Handler                ; The PendSV handler
             DCD      SysTick_Handler               ; The SysTick handler

             ; External interrupts...
             DCD      GPIOPortA_IRQHandler          ; GPIO Port A
             DCD      Unused_IRQHandler             ; GPIO Port B
             ~ ~ ~ ~
```

(1) The vector table is placed in a read-only section `RESET`.

(2) The `assert_failed()` callback function is imported so that ARM exceptions can be handled with the same assertion failure mechanism as any other assertions in the system.

(3-4) The exception handlers for `SVC_Handler` and `PendSV_Handler` are imported only in if the preemptive QK kernel is used. For the cooperative Vanilla kernel these handlers are defined in the `startup_<mcu>.s` module.

(5-6) All interrupt handlers defined outside of the `startup_<mcu>.s` module are imported.

(7) The vector table is placed at symbol `__Vectors`

(8) The C top of stack is the first element of ARM Cortex-M vector table

(9) The addresses of exception handlers and interrupt handlers follow.

3.3.2 *Adjusting the Stack and Heap Sizes*

You can adjust the sizes of the C stack (the only one used in QP) and the heap by editing the `startup_<mcu>.s` file (located in the project directory). The following listing shows the symbols you can adjust for your specific application. Please note that the heap can be configured to zero:

Listing 6: The interrupt vector table defined in startup_<mcu>.s (ARMASM assembler).

```
         ~ ~ ~ ~
;*****************************************************************************
; <o> Stack Size (in Bytes) <0x0-0xFFFFFFFF:8>
;*****************************************************************************
Stack    EQU     0x00000200
;*****************************************************************************
;
; <o> Heap Size (in Bytes) <0x0-0xFFFFFFFF:8>
;
;*****************************************************************************
```

```
Heap    EQU     0x00000000
;******************************************************************************
; Allocate space for the stack.
;******************************************************************************
        AREA    STACK, NOINIT, READWRITE, ALIGN=3
StackMem
        SPACE   Stack
__initial_sp

;******************************************************************************
; Allocate space for the heap.
;******************************************************************************
        AREA    HEAP, NOINIT, READWRITE, ALIGN=3
__heap_base
HeapMem
        SPACE   Heap
__heap_limit
         ~ ~ ~ ~
```

3.4 Using the FPU (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 hardware reset. If the software FPU is not used, the hardware FPU is enabled in the Reset Handler before the call to `__main`, as follows:

Listing 7: The reset handler (file startup_<mcu>.s).

```
             EXPORT  Reset_Handler
(10) Reset_Handler
             ;
             ; Call the C library enty point that handles startup.  This will copy
             ; the .data section initializers from flash to SRAM and zero fill the
             ; .bss section.
(11)         IMPORT  __main

(12)   IF {FPU} != "SoftVFP"       ; If software FPU not used...
             ; NOTE: The FPU must be enabled before jumping to __main, because
             ; the initialization code downstream assumes that the FPU is present
             ; and a fault exception would result if the FPU was not enabled.
             LDR     r0,=0xE000ED88
             LDR     r1,[r0]
             ORR     r1,r1,#0xF00000
             STR     r1,[r0]
       ENDIF

             B       __main
```

NOTE: When the runtime MicroLIB library is not configured, the FPU must be enabled before the call to `__main`, because the startup code uses the FPU without initializing it. An attempt to execute a floating point instruction will fault if the FPU is not enabled.

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

3.4.1 *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

3.4.2 *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.

3.5 Using the MicrLIB Runtime Library

The QP/C framework can work completely standalone without the standard MicroLIB runtime library. Specifically, the initialization of the FPU in Reset_Handler() has been added to enable this standalone operation even for the most advanced Cortex-M4F MCUs.

However, the QP/C++ framework needs the support of the standard MicroLIB runtime library to properly invoke the static constructors.

NOTE: The C++ applications need to use the **MicroLIB** runtime library to invoke static constructors.

3.6 Idle Loop Customization in the “Vanilla” Port

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

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

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

Listing 8 shows an example implementation of `QF_onIdle()` for the Stellaris MCU. Other embedded microcontrollers (e.g., ST's STM32) handle the power-saving mode very similarly.

Listing 8: QF_onIdle() callback.

```
(1) void QF_onIdle(void) {       /* entered with interrupts DISABLED, see NOTE01 */
        . . .
(2) #if defined NDEBUG
(3)     QF_CPU_SLEEP();          /* atomically go to sleep and enable interrupts */
    #else
(4)     QF_INT_ENABLE();                                /* just enable interrupts */
    #endif
    }
```

(1) The cooperative Vanilla kernel calls the `QF_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 QF_CPU_SLEEP() is used to put the CPU to the low-power sleep mode **safely**. The macro QF_CPU_SLEEP() is defined in the qf_port.h header file for the Vanilla kernel and depends on the interrupt disabling policy used, as described in Section 3.2.

(4) When a sleep mode is not used, the `QF_onIdle()` callback simply re-enables interrupts.

4 The Preemptive QK Kernel

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

NOTE: The preemptive configuration with QK uses **more stack** than the non-preemptive "Vanilla" configuration. You need to adjust the size of this stack to be large enough for your application, as described in **Section 3.3.2**.

4.1 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 SVCall (exception number 11) to perform asynchronous preemptions and context switch, respectively (see Chapter 10 in [PSiCC2]). The application code (your code) **must** initialize the Interrupt Vector Table with the addresses of `PendSV_Handler` and `SVCall_Handler` exception handlers. Additionally, the interrupt table must be initialized with the SysTick handler that calls `QF_tick()`.
4. The application code (your code) **must** call the function `QK_init()` to set the priority of the PendSV exception to the lowest level in the whole system (0xFF), and the priority of SVCall to the highest in the system (0x00). The function `QK_init()` sets the priorities of exceptions 14 and 11 to the numerical values of 0xFF and 0x00, respectively. The priorities are set with interrupts disabled, but the interrupt status is restored upon the function return.

NOTE: The Stellaris 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 4.3.1).
7. ARM Cortex-M enters interrupt context without locking interrupts (without setting the PRIMASK bit). Generally, you should not lock interrupts inside ISRs. In particular, the QF services `QF_publish()`, `QF_tick(), and QActive_postFIFO()` should be called with interrupts enabled, to avoid nesting of critical sections.

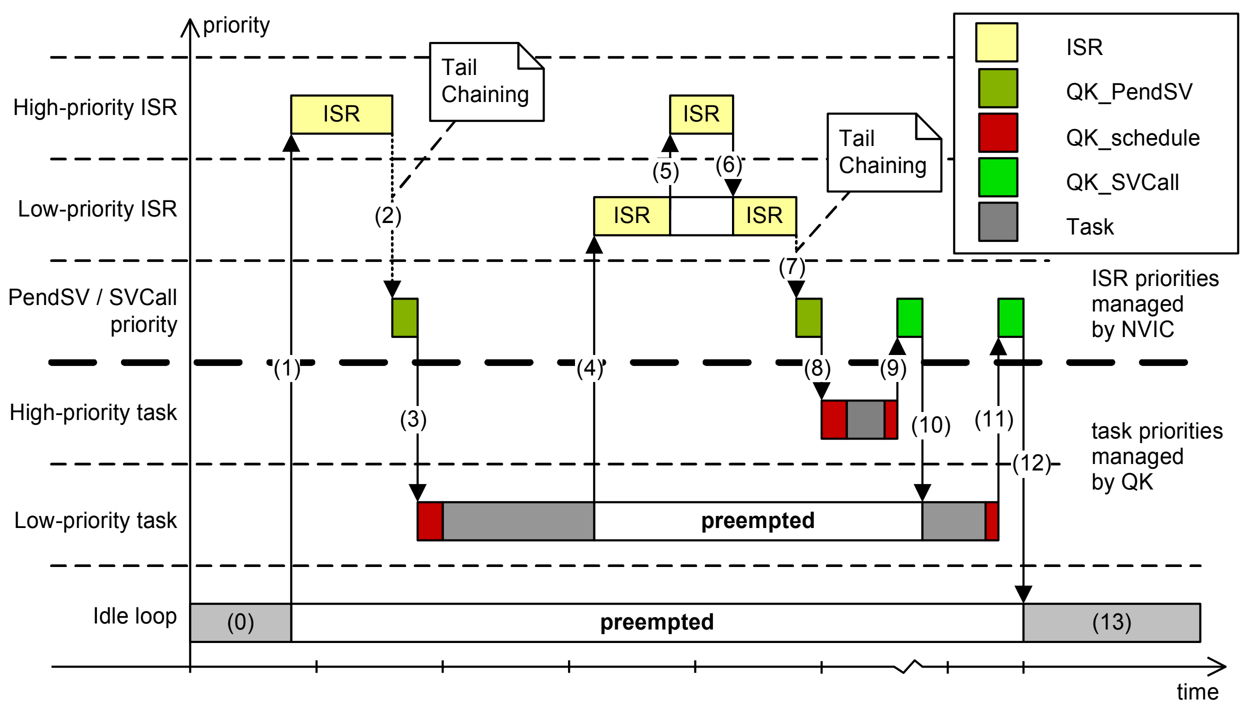
NOTE: If you don't wish an interrupt to be preempted by another interrupt, you can always prioritize that interrupt in the NVIC to a higher level (use a lower numerical value of priority).

8. In the whole prioritization of interrupts, including the PendSV exception, is performed entirely by the NVIC. Because the PendSV has the lowest priority in the system, the NVIC tail-chains to the PendSV exception only after exiting the last nested interrupt.
9. The restoring of the 8 registers comprising the interrupt stack frame in PendSV is wasteful in a single-stack kernel (see Listing 10(3) and (8)), but is necessary to perform full interrupt return from PendSV to signal End-Of-Interrupt to the NVIC.
10. The pushing of the 8 registers comprising the interrupt stack frame upon entry to SVCall is wasteful in a single-stack kernel (see Figure 7(10) and (12)), but is necessary to perform full interrupt return to the preempted context through the SVCall's return.
11. For Cortex-M4F processors with hardware FPU, the application can choose between two policies of using the FPU

4.1.1 Examples of Various Preemption Scenarios in QK

Figure 7 illustrates several preemption scenarios in QK.

Figure 7: Various preemption scenarios in the QK preemptive kernel for .



(0) The time line in Figure 7 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 CPU performs tail-chaining to the pending PendSV exception.

(3) The whole job of the PendSV exception is to synthesize an interrupt stack frame on top of the stack and perform an interrupt return.

(4) The PC (exception return address) of the synthesized stack frame is set to `QK_schedule()` (more precisely to a thin wrapper around `QK_schedule()`, see Section 4.4), so the PendSV exception returns to the QK scheduler. The scheduler discovers that the Low-priority task is ready to run (the ISR has posted event to this task). The QK scheduler enables interrupts and launches the Low-priority task, which is simply a C-function call in QK. The Low-priority task (active object) starts running. Some time later another interrupt occurs. The Low-priority task is suspended and the CPU pushes the interrupt stack frame to the Main Stack and starts executing the ISR

(5) The Low-priority ISR runs and sets the pending flag for the PendSV exception in the NVIC. Before the Low-priority ISR completes, it too gets preempted by a High-priority ISR. The CPU pushes another interrupt stack frame and starts executing the High-priority ISR.

(6) The High-priority ISR again sets the pending flag for the PendSV exception (setting an already set flag is not an error). When the High-priority ISR returns, the NVIC does not tail-chain to the PendSV exception, because a higher-priority ISR than PendSV is still active. The NVIC performs the normal interrupt return to the preempted Low-priority interrupt, which finally completes.

(7) Upon the exit from the Low-priority ISR, the NVIC performs tail-chaining to the pending `PendSV` exception

(8) The `PendSV` exception synthesizes an interrupt stack frame to return to the QK scheduler.

(9) The QK scheduler detects that the High-priority task is ready to run and launches the High-priority task (normal C-function call). The High-priority task runs to completion and returns to the scheduler. The scheduler does not find any more higher-priority tasks to execute and needs to return to the preempted task. The only way to restore the interrupted context in is through the interrupt return, but the task is executing outside of the interrupt context (in fact, tasks are executing in the Privileged Thread mode). The task enters the Handler mode by causing the synchronous **SVCall** exception

(10) The only job of the SVCall exception is to discard its own interrupt stack frame and return using the interrupt stack frame that has been on the stack from the moment of task preemption

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

(12) The SVCall exception discards its own interrupt stack frame and returns using the interrupt stack frame from the preempted task context

4.2 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 hardware reset. If the software FPU is not used, the hardware FPU is enabled in the Reset Handler before the call to `__main`, as described in Section 3.4.

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.

4.2.1 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:

```
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 more than one task or any of the ISRs indeed start to use the FPU

4.2.2 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:

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

4.3 The QK Port Header File

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

You configure and customize QK through the header file `qk_port.h`, which is located in the QP ports directory `<qp>\ports\arm-cm\`**`qk`**`\arm_keil\`. 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 9: qk_porth.h header file

```
(1) #define QK_ISR_ENTRY() do { \
(2)     QF_INT_DISABLE(); \
(3)     ++QK_intNest_; \
(4)     QF_INT_ENABLE(); \
    } while (0)

(5) #define QK_ISR_EXIT()  do { \
(6)     QF_INT_DISABLE(); \
(7)     --QK_intNest_; \
(8)     *((uint32_t volatile *)0xE000ED04U) = 0x10000000U; \
```

```
(9)      QF_INT_ENABLE(); \
     } while (0)

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

(1) The `QK_ISR_ENTRY()` macro notifies QK about entering an ISR. The macro body is surrounded by the `do {…} while (0)` loop, which is the standard way of grouping instructions without creating a dangling-`else` or other syntax problems. In ARM Cortex-M, this macro is called with interrupts unlocked, because the ARM Cortex-M hardware does not set the PRIMASK upon interrupt entry.

(2) Interrupts are disabled at the ARM Cortex-M core level to perform the following actions atomically.

(3) The QK interrupt nesting level `QK_intNest_` is incremented to account for entering an ISR. This prevents invoking the QK scheduler from event posting functions (such as `QActvie_postFIFO()` or `QActive_postLIFO()`) to perform a synchronous preemption.

(4) Interrupts are enabled at the ARM Cortex-M core level to allow interrupt preemptions.

(5) The `QK_ISR_EXIT()` macro notifies QK about exiting an ISR.

(6) Interrupts are disabled at the ARM Cortex-M core level to perform the following actions atomically.

(7) The QK interrupt nesting level `QK_intNest_` is decremented to account for exiting an ISR. This balances step (3).

(8) This write to the `NVIC_INT_CTRL` register sets the pending flag for the `PendSV` exception.

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

---

(9) Interrupts are enabled to perform regular exit from the ISR.

(10) The QK port header file must include the platform-independent QK interface `qk.h`.

4.3.1 The QK Critical Section

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

4.4 QK Platform-Specific Code for ARM Cortex-M

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

Listing 10: `QK_init()` function for ARM Cortex-M in ARMASM (file qk_port.s)

```
AREA    |.text|, CODE, READONLY
THUMB

PRESERVE8                   ; this code preserves 8-byte stack alignment

EXPORT  QK_init
EXPORT  PendSV_Handler      ; CMSIS-compliant PendSV exception name
EXPORT  SVC_Handler         ; CMSIS-compliant SVC exception name
```

```
          IMPORT  QK_schedPrio_      ; external reference
          IMPORT  QK_sched_          ; external reference

      ;*****************************************************************************
      ; The QK_init function sets the priorities of PendSV and SVCall exceptions
      ; to 0xFF and 0x00, respectively. The function internally disables
      ; interrupts, but restores the original interrupt lock before exit.
      ;*****************************************************************************
 (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)      MOVS    r3,#0xFF
 (7)      LSLS    r3,r3,#16
 (8)      ORRS    r2,r3              ; set PRI_14 (PendSV) to 0xFF
 (9)      STR     r2,[r1,#8]         ; write the System 12-15 Priority Register
(10)      LDR     r2,[r1,#4]         ; load the System 8-11 Priority Register
(11)      LSLS    r3,r3,#8
(12)      BICS    r2,r3              ; set PRI_11 (SVCall) to 0x00
(13)      STR     r2,[r1,#4]         ; write the System 8-11 Priority Register

(14)      MSR     PRIMASK,r0         ; restore the original PRIMASK
(15)      BX      lr                 ; return to the caller
```

(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 locked by setting the PRIMASK.

(4) The address of the NVIC System Handler Priority Register 0 is loaded into r1

(5) The contents of the NVIC System Handler Priority Register 2 (note the offset of 8) is loaded into r2.

(6-7) 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 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 contents of the NVIC System Handler Priority Register 1 (note the offset of 4) is loaded into r2

(11) The mask value of 0xFF000000 is synthesized in r3.

(12) The mask is then applied to set the priority byte PRI_11 to 0x00 without changing priority bytes in this register.

(13) The contents of r2 is stored in the NVIC System Handler Priority Register 1 (note the offset of 4).

(14) The original PRIMASK value is restored.

(15) The function `QK_init` returns to the caller.

Listing 11: `PendSV_Handler()` function for ARM Cortex-M (file qk_port.s).

```
 (1) PendSV_Handler
 (2)     PUSH    {lr}                ; push the exception lr (EXC_RETURN)

 (3) IF {TARGET_ARCH_THUMB} == 3     ; Cortex-M0/M0+/M1 (v6-M, v6S-M)?
 (4)     CPSID   i                   ; disable interrupts (set PRIMASK)
 (5) ELSE                            ; Cortex-M3/M4/M4F
 (6)     MOVS    r0,#(0xFF >> 2)     ; Keep in synch with QF_BASEPRI in qf_port.h!
 (7)     MSR     BASEPRI,r0          ; disable interrupts at processor level
     ENDIF

 (8)     BL      QK_schedPrio_       ; check if we have preemption
 (9)     CMP     r0,#0               ; is prio == 0 ?
(10)     BNE.N   scheduler           ; if prio != 0, branch to scheduler

     IF {TARGET_ARCH_THUMB} == 3     ; Cortex-M0/M0+/M1 (v6-M, v6S-M)?
(11)     CPSIE   i                   ; enable interrupts (clear PRIMASK)
     ELSE                            ; Cortex-M3/M4/M4F
(12)     MSR     BASEPRI,r0          ; enable interrupts (r0 == 0 at this point)
     ENDIF

(13)     POP     {r0}                ; pop the EXC_RETURN into r0 (low register)
(14)     BX      r0                  ; exception-return to the task

(15) scheduler
(16)     SUB     sp,sp,#4            ; align the stack to 8-byte boundary
(17)     MOVS    r3,#1
(18)     LSLS    r3,r3,#24           ; r3:=(1 << 24), set the T bit  (new xpsr)
(19)     LDR     r2,=QK_sched_       ; address of the QK scheduler   (new pc)
(20)     LDR     r1,=svc_ret         ; return address after the call (new lr)
(21)     PUSH    {r1-r3}             ; push xpsr,pc,lr
(22)     SUB     sp,sp,#(4*4)        ; don't care for r12,r3,r2,r1
(23)     PUSH    {r0}                ; push the prio argument        (new r0)
(24)     MOVS    r0,#0x6
(25)     MVNS    r0,r0               ; r0:=~0x6=0xFFFFFFF9
(26)     BX      r0                  ; exception-return to the scheduler

(27) svc_ret
     IF {TARGET_ARCH_THUMB} == 3     ; Cortex-M0/M0+/M1 (v6-M, v6S-M)?
(28)     CPSIE   i                   ; enable interrupts (clear PRIMASK)
     ELSE                            ; Cortex-M3/M4/M4F
(29)     MOVS    r0,#0
(30)     MSR     BASEPRI,r0          ; enable interrupts
     ENDIF

(31)   IF {FPU} != "SoftVFP"         ; If software FPU not used...
(32)     MRS     r0,CONTROL          ; r0 := CONTROL
(33)     MOVS    r1,#4               ; r1 := 0x04 (FPCA bit)
(34)     BICS    r0,r1               ; r0 := r0 & ~r1
(35)     MSR     CONTROL,r0          ; CONTROL := r0
       ENDIF

(36)     SVC     #0                  ; SV exception returns to the preempted task
```

(1) The `PendSV_Handler` exception is always entered via tail-chaining from the last nested interrupt (see Section 4.1).

(2) The exception `lr` (EXC_RETURN) is pushed to the stack.

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.

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

(4) Interrupts are disabled by setting the PRIMASK.

(5) For the ARMv7-M architecture...

(6-7) Interrupts are disabled by setting the BASEPRI register.

NOTE: The value moved to BASEPRI must be identical to `QF_BASEPRI` used in `qf_port.h`, see Section 3.2.

(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 `scheduler`) is taken if the priority is not zero.

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

(12) 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).

(13) The saved EXC_RETURN is popped from the stack to `r0`. NOTE: the `r0` register is used instead of `lr` because the Cortex-M0 instruction set cannot manipulate the higher-registers (`r9`-`r15`).

(14) This BX instruction causes exception-return to the preempted task. (Exception-return pops the 8-register exception stack frame and changes the processor state to the task-level).

(15) The scheduler 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 to the task-level.

(16) The stack pointer is aligned to the 8-byte boundary.

NOTE: The exception stack-frame that is about to be built on top of the current stack must be aligned at 8-byte boundary. This alignment has been lost in step (2), where the `EXC_RETURN` from `lr` has been pushed to the stack. In step (11), the stack is aligned again by growing the stack by four more bytes. (The stack grows towards lower addresses in ARM Cortex-M, so the stack pointer is decremented).

(17-18) 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).

(19) 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.

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

NOTE: The address of the `svc_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 ARMCC assembler/linker synthesize the correct THUMB address of the `svc_ret` label without any extra work on the programmer's part.

(21) Registers `r3`, `r2` and `r1` are pushed onto the stack.

(22) The stack pointer is adjusted to leave room for 4 registers. The actual stack contents for these registers is irrelevant.

(23) The original priority returned in `r0` from `QK_schedPrio_` is pushed to the stack. This will be restored to `r0` register value. 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 11(2))
old SP --> "aligner"  (added in Listing 11(11))
           xPSR == 0x01000000
           PC == QK_sched_
           lr == svc_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
```

(24-25) The special exception-return value 0xFFFFFFF9 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.

(26) `PendSV` exception returns using the special value of the `r0` register of 0xFFFFFFF9 (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 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`.

(27) 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 (14)).

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

(29-30) For the ARMv7-M architecture, interrupts are enabled by setting the BASEPRI register to zero.

(31) The following code is assembled conditionally only when the FPU is actually used.

(32-35) The read-modify-write code clears the `CONTROL[2]` bit [2]. This bit, called `CONTROL.FPCA` (Floating Point Active), causes generating the FPU-type stack frame, if the bit is set and the "automatic state saving" of the FPU is configured.

NOTE: Clearing the CONTROL.FPCA bit is safe in this situation, becaue the SVC exception is not using the FPU. Also, note that the CONTROL.FPCA bit is restored from ~EXC_RETURN[4] when the SVC exception retuns to the task level (see Listing 12(3)).

(36) The synchronous SVC exception is called to put the CPU into the exception mode and correctly return to the thread level.

Listing 12: `SVC_Handler()` function for ARM Cortex-M (file qk_port.s).

```
    ;*****************************************************************************
    ; The SVC_Handler exception handler is used for returning back to the
    ; interrupted task. The SVCall exception simply removes its own interrupt
    ; stack frame from the stack and returns to the preempted task.
    ;*****************************************************************************
(1) SVC_Handler
(2)     ADD     sp,sp,#(9*4)        ; remove one 8-register exception frame
                                    ; plus the "aligner" from the stack
(3)     POP     {r0}                ; pop the original EXC_RETURN into r0
(4)     BX      r0                  ; return to the preempted task

        ALIGNROM 2,0xFF             ; make sure the END is properly aligned
        END
```

(1) The job of the `SVCall` exception is to discard its own stack frame and cause the exception-return to the original preempted task context. The stack contents just before returning from `SVCall` exception is shown below:

```
Hi memory
           (optionally 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 11(2))
           "aligner"  (added in Listing 11(11))
           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 `SVCall` exception itself plus the "aligner" added to the stack in Listing 11(11).

(3) The EXC_RETURN saved in Listing 11(2) is popped from the stack into `r0` (low register for Cortex-M0 compatibility)

(4) `SVCall` exception returns to the interrupted task level using the original EXC_RETURN, which codifies the stack frame type.

4.5 Setting up and Starting Interrupts in QF_onStartup()

Setting up interrupts (e.g., `SysTick`) for the preemptive QK kernel is identical as in the non-preemptive case. Please refer to Section Error: Reference source not found.

4.6 Writing ISRs for QK

QK must be informed about entering and exiting every ISR, so that it can perform asynchronous preemptions. You inform the QK kernel about the ISR entry and exit through the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, respectively. You need to call these macros in every ISR. The following listing shows the ISR the file `<qp>\examples\arm-cm\`**`qk`**`\arm_keil\dpp-`**`qk`**`-ev-lm3s811\bsp.c`.

```
void SysTick_Handler(void) {
    QK_ISR_ENTRY();                              /* inform QK about ISR entry */
    QF_TICK(&l_SysTick_Handler);
    QK_ISR_EXIT();                                /* inform QK about ISR exit */
}
/*..........................................................................*/
void GPIOPortA_IRQHandler(void) {
    QK_ISR_ENTRY();                              /* inform QK about ISR entry */
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG));   /* for testing */
    QK_ISR_EXIT();                                /* inform QK about ISR exit */
}
```

4.7 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 **enabled** (which is in contrast to the `QF_onIdle()` callback used in the non-preemptive configuration, see Section 3.6).

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" [ARMv7-M].

The following Listing 13 shows the `QF_onIdle()` callback that puts ARM Cortex-M into the idle power-saving mode.

Listing 13 QK_onIdle() for the preemptive QK configuration.

```
(1) void QK_onIdle(void) {

        /* toggle the User LED on and then off, see NOTE01 */
(2)     QF_INT_DISABLE();
(3)     GPIOC->DATA_Bits[USER_LED] = USER_LED;         /* turn the User LED on  */
(4)     GPIOC->DATA_Bits[USER_LED] = 0;                /* turn the User LED off */
(5)     QF_INT_ENABLE();

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

(1) The `QK_onIdle()` function is called with interrupts enabled.

(2) The interrupts are disabled to prevent preemptions when the LED is on.

(3-4) This QK port uses the USER LED of the EK-LM3S811 board to visualize the idle loop activity. The LED is rapidly toggled on and off as long as the idle condition is maintained, so the brightness of the LED is proportional to the CPU idle time (the wasted cycles). Please note that the LED is on in the critical section, so the LED intensity does not reflect any ISR or other processing. The USER LED of the EV-LM3S811 board is toggled on and off.

(5) Interrupts are re-enabled.

NOTE: Obviously, toggling the USER LED is optional and is not necessary for correctness of the QK-port. You can eliminate code in lines (3-5) in your application.

(6) This part of the code is only used in the QSpy build configuration. In this case the idle callback is used to transmit the trace data using the UART of the ARM Cortex-M device.

(7) The following code is only executed when no debugging is necessary (release version).

(8) The `WFI` instruction is generated using inline assembly.

4.8 Testing QK Preemption Scenarios

The DPP example application includes special instrumentation for convenient testing of various preemption scenarios, such as those illustrated in Figure 8.

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:

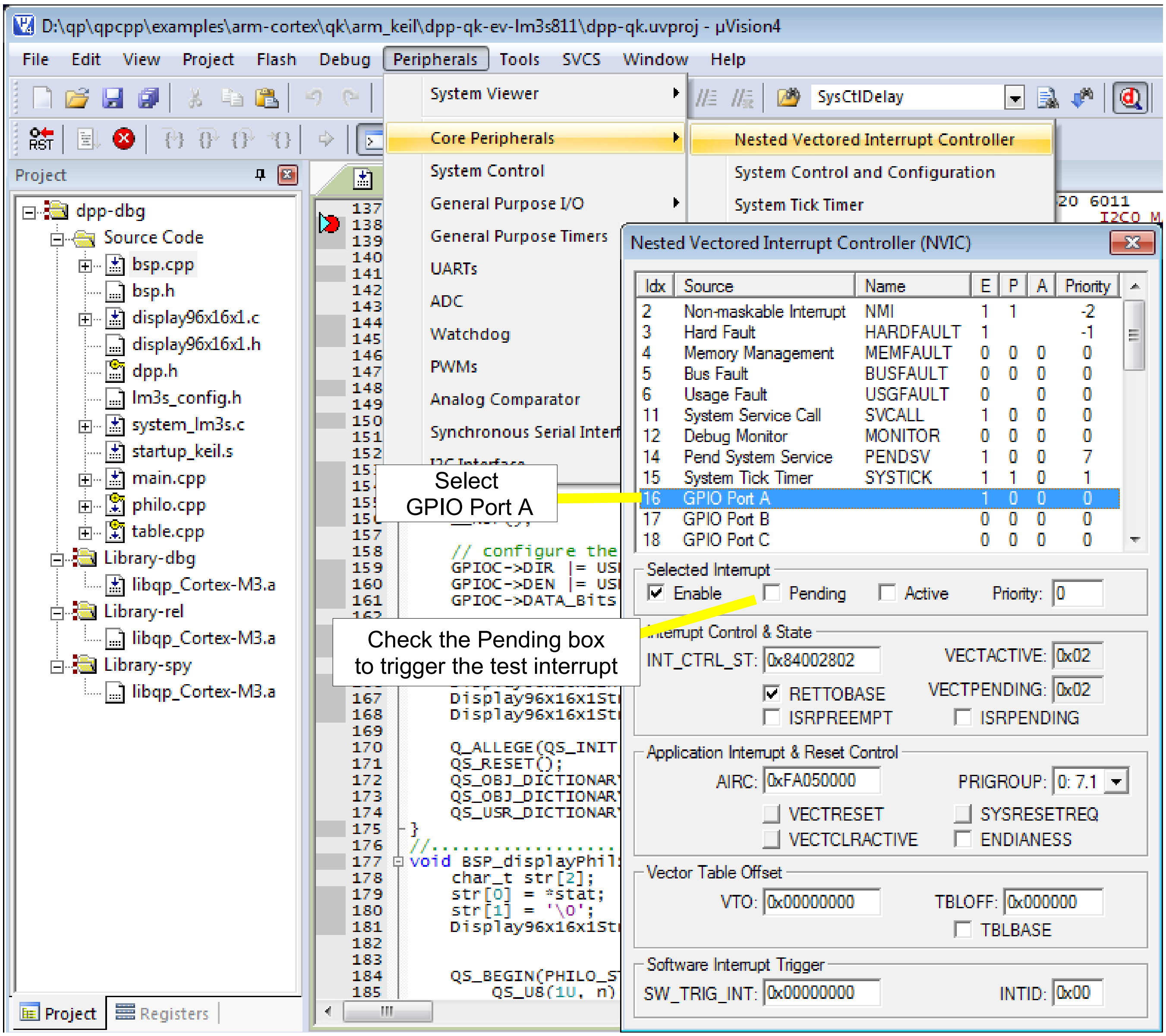
```
void GPIOPortA_IRQHandler(void) {
    QK_ISR_ENTRY();
```

```
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG));  /* for testing */
    QK_ISR_EXIT();
}
```

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

Figure 8 shows how to trigger the GPIOA interrupt from the Keil uVision4 debugger. From the debugger you need to first open the **Peripherals | Core Peripherals | Nesteed Vectored Interrupt Controller** view. Next, you select the **GPIO Port A** peripheral from the list. And finally, you check the **Pending** box to trigger the GPIOA interrupt.

Figure 8 Triggering the GPIOA interrupt from the Keil uVision4 debugger.



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.

4.8.1 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 7(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 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 SysTick ISR.
2. The second breakpoint hit is the one in the `SysTick_Handler()`, which means that the SysTick ISR continues after the GPIOA ISR completes.
3. The last breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is tail-chained only after all interrupts are processed.

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

4.8.2 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 `Philospher` 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 preempting task (the `Table` state machine) completes.

4.8.3 Testing the FPU (Cortex-M4F)

In order to test the FPU, the Board Support Package (BSP) for the Cortex-M4F EK-LM4F120XL 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 caled from the `Table` active object (QP priorty 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 stak frames (with and without the FPU).

4.8.4 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.

5 QS Software Tracing Instrumentation

Quantum Spy (QS) is a software tracing facility built into all QP components and also available to the Application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (please refer to "QSP Reference Manual" section in the "QP/C Reference Manual" 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 EK-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 14 QSpy implementation to send data out of the UART0 of the LM3S/LM4F MCUs.

```
(1) #ifdef Q_SPY

(4)     QSTimeCtr QS_tickTime_;
        QSTimeCtr QS_tickPeriod_;

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

                                    /* enable the peripherals used by the UART */
                                   /* enable the peripherals used by the UART0 */
        SYSCTL->RCGC1 |= (1 << 0);                       /* enable clock to UART0 */
        SYSCTL->RCGC2 |= (1 << 0);                       /* enable clock to GPIOA */
        __NOP();                                     /* wait after enabling clocks */
        __NOP();
        __NOP();

                                    /* configure UART0 pins for UART operation */
        tmp = (1 << 0) | (1 << 1);
        GPIOA->DIR   &= ~tmp;
        GPIOA->AFSEL |= tmp;
        GPIOA->DR2R  |= tmp;        /* set 2mA drive, DR4R and DR8R are cleared */
        GPIOA->SLR   &= ~tmp;
        GPIOA->ODR   &= ~tmp;
        GPIOA->PUR   &= ~tmp;
        GPIOA->PDR   &= ~tmp;
        GPIOA->DEN   |= tmp;

              /* configure the UART for the desired baud rate, 8-N-1 operation */
        tmp = (((SystemFrequency * 8) / UART_BAUD_RATE) + 1) / 2;
```

```
        UART0->IBRD   = tmp / 64;
        UART0->FBRD   = tmp % 64;
        UART0->LCRH   = 0x60;                      /* configure 8-N-1 operation */
        UART0->LCRH  |= 0x10;
        UART0->CTL   |= (1 << 0) | (1 << 8) | (1 << 9);

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

        return (uint8_t)1;                                    /* return success */
    }
    /*..........................................................................*/
(9) void QS_onCleanup(void) {
    }
    /*..........................................................................*/
(10) void QS_onFlush(void) {
        uint16_t fifo = UART_TXFIFO_DEPTH;                     /* Tx FIFO depth */
        uint8_t const *block;
        QF_INT_LOCK(dummy);
        while ((block = QS_getBlock(&fifo)) != (uint8_t *)0) {
            QF_INT_UNLOCK(dummy);
                                              /* busy-wait until TX FIFO empty */
            while ((UART0->FR & UART_FR_TXFE) == 0) {
            }

            while (fifo-- != 0) {                    /* any bytes in the block? */
                UART0->DR = *block++;                   /* put into the TX FIFO */
            }
            fifo = UART_TXFIFO_DEPTH;                /* re-load the Tx FIFO depth */
            QF_INT_LOCK(dummy);
        }
        QF_INT_UNLOCK(dummy);
    }
    /*..........................................................................*/
(11) QSTimeCtr QS_onGetTime(void) {           /* invoked with interrupts locked */
(12)     if ((HWREG(NVIC_ST_CTRL) & NVIC_ST_CTRL_COUNT) == 0) { /* COUNT no set? */
(13)         return QS_tickTime_ - (QSTimeCtr)SysTick->VAL;
        }
        else {     /* the rollover occured, but the SysTick_ISR did not run yet */
(14)         return QS_tickTime_ + QS_tickPeriod_ - (QSTimeCtr)SysTick->VAL;
        }
    }
    #endif                                                           /* Q_SPY */
```

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

(2-3) The QS implementation uses the UART driver provided in the Luminary Micro library.

(4) These variables are used for time-stamping the QS data records. This `QS_tickTime_` variable is used to hold the 32-bit-wide SysTick timestamp at tick. The `QS_tickPeriod_` variable holds the nominal number of hardware clock ticks between two subsequent SysTicks. The SysTick ISR increments `QS_tickTime_` by `QS_tickPeriod_`.

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

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

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

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

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

(10) The `QS_flush()` callback flushes the QS trace buffer to the host. Typically, the function busy-waits for the transfer to complete. It is only used in the initialization phase for sending the QS dictionary records to the host (see please refer to "QSP Reference Manual" section in the "QP/C Reference Manual" an also to Chapter 11 in [PSiCC2])

5.1 QS Time Stamp Callback QS_onGetTime()

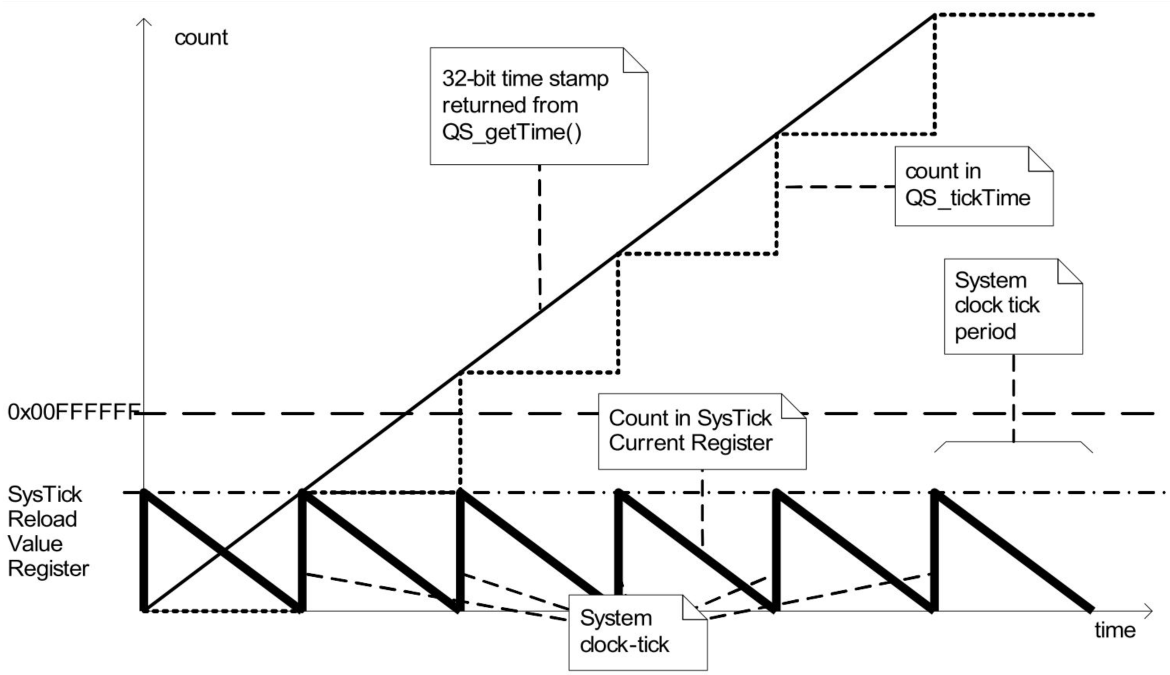
The platform-specific QS port must provide function `QS_onGetTime()` (Listing 14(11)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the ARM Cortex-M port uses the SysTick facility, which is the same timer already used for generation of the system clock-tick interrupt.

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

Figure 9 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 9 Using the SysTick Current Value Register to provide 32-bit QS time stamp.



Listing 14(11-15) shows the implementation of the function `QS_onGetTime()`, which combines all this information to produce a monotonic time stamp.

(12) The `QS_onGetTime()` function tests the `NVIC_ST_CTRL_COUNT`. This flag being set means that the `NVIC_ST_CURRENT` has rolled over to zero, but the SysTick ISR has not run yet (because interrupts are still locked).

(13) Most of the time the `NVIC_ST_CTRL_COUNT` flag is not set, and the time stamp is simply the sum of `QS_tickTime_ + (-HWREG(NVIC_ST_CURRENT))`. Please note that the `NVIC_ST_CURRENT` register is negated to make it to an up-counter rather than down-counter.

(13) If the `NVIC_ST_CTRL_COUNT` flag is set, the `QS_tickTime_` counter misses one update period and must be additionally incremented by `QS_tickPeriod_`.

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

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

Listing 15 QS trace output using the UART0 of the Tiva-C MCU

```
     #define UART_TXFIFO_DEPTH 16
     . . .
     void QK_onIdle(void) {
         . . .
     #ifdef Q_SPY
(1)      if ((UART0->FR & UART_FR_TXFE) != 0) {                      /* TX done? */
(2)          uint16_t fifo = UART_TXFIFO_DEPTH;        /* max bytes we can accept */
(3)          uint8_t const *block;
(4)          QF_INT_DISABLE();
(5)          block = QS_getBlock(&fifo);     /* try to get next block to transmit */
(6)          QF_INT_ENABLE();
(7)          while (fifo-- != 0) {                     /* any bytes in the block? */
(8)              UART0->DR = *block++;                      /* put into the FIFO */
             }
         }
     #elif defined NDEBUG                     /* sleep mode interferes with debugging */
         . . .
     }
```

(1) The `UART_FR_TXFE` flag is set when the TX FIFO becomes empty. If the flag is set, the TX FIFO can be filled with up to 16 bytes of new data.

(2) The fifo variable is initialized with the maximum number of bytes the `QS_getBlock()` function can deliver (see “QS Programmer’s Manual”).

(3) The block variable is the pointer to the contiguous data block returned from `QS_getBlock()` function (see “QS Programmer’s Manual”).

(4) Interrupts are locked to call `QS_getBlock()`.

(5) The function `QS_getBlock()` returns the contiguous data block of up-to `UART_TXFIFO_DEPTH` fifo bytes. The function also returns the actual number of bytes available in the `fifo` variable (passed as a pointer).

(6) The interrupts are unlocked after the call to `QS_getBlock()`.

(7) The `while()` loop goes over all bytes delivered from `QS_getBlock()`. (NOTE: if zero bytes are delivered, the loop does not go even once.)

(8) The next byte pointed to by the block pointer is inserted into the TX FIFO and the block pointer is advanced to the next byte.

5.3 Invoking the QSpy Host Application

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

```
qspy -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.

6 Related Documents and References

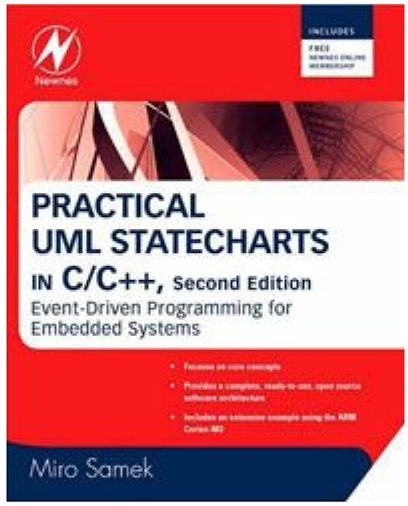
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.	http://www.embedded.com/showArticle.jhtml?articleID=190302110
[ARMv7-M] "ARM v7-M Architecture Application Level Reference Manual", ARM Limited	Available from http://infocenter.arm.com/help/.
[Cortex-M3] "Cortex™-M3 Technical Reference Manual", ARM Limited	Available from http://infocenter.arm.com/help/.
[ARM AN298] ARM Application Note 298 "Cortex-M4(F) Lazy Stacking and Context Switching", ARM 2012	Available from http://infocenter.arm.com/help/topic/com.arm.doc.dai0298a/DAI0298A_cortex_m4f_lazy_stacking_and_context_switching.pdf
[Luminary 12] "LM3S811 Microcontroller Data Sheet", Texas Instruments, 2012	Texas Instruments literature number SPMS150I
[Tiva-C 13] "Tiva ™ TM4C123GH6PM Microcontroller (identical to LM4F230H5QR)", Texas Instruments, 2013	Texas Instruments literature number SPMS376B
[ARMCC 13] "Compiler User Guide", ARM/Keil 2013	Available in the help for ARM/Keil uVision4 IDE.
[ARMASM 13] "Assembler User Guide", ARM/Keil 2013	Available in the help for ARM/Keil uVision4 IDE.
[uVision4 13] "uVision4 User's Guide", ARM/Keil 2013	Available in the help for ARM/Keil uVision4 IDE.

7 Contact Information

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

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

e-mail: info@quantum-leaps.com
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