Application Note
QP-nano™ and ARM7/9 with IAR

Document Revision P
November 2011

Copyright © Quantum Leaps, LLC

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

Quantum™ LeaPs
innovating embedded systems

Table of Contents

1 Introduction.....1
1.1 About QP-nano™.....1
1.2 About the QP-nano™ Port to ARM7/9.....2
1.3 Licensing QP-nano.....3
2 Directories and Files.....4
3 Non-Preemptive Configuration of QP-nano.....5
3.1 Configuration and Customizing QP-nano (qpn_port.h).....5
3.2 ARM-Specific QF-nano Port (qfn_port.s).....8
4 Preemptive Configuration with QK-nano.....20
4.1 Configuration and Customizing QP-nano.....20
4.2 Handling Interrupts in QK-nano.....21
4.3 Idle Loop Customization in the QK Port.....25
5 Related Documents and References.....26
6 Contact Information.....27

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1 Introduction

This Application Note describes how to use the QP-nano™ state machine framework version 4 or higher with the ARM7 or ARM9 processors. This document describes the following two main implementation options:

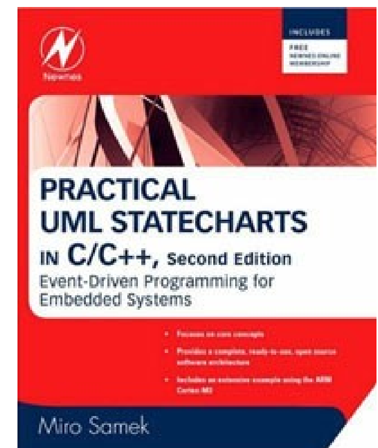
1. The cooperative "Vanilla" kernel available in the QF-nano real-time framework; and
2. The preemptive run-to-completion QK-nano kernel.

To focus the discussion, this Application Note uses the IAR Embedded Workbench® for ARM (EWARM version **5.30** KickStart™ edition, which is available as a **free** download from the IAR website www.iar.com). However, most of the code described here is generic ARM/THUMB and should be easily adapted to any ARM development toolset, such as the RealView®, Keil, Green Hills, or GNU.

This Application Note does not contain any executable examples, which are provided in the QP-nano Development Kits™ (QDKs-nano) for specific ARM7/ARM9 boards. The QDKs-nano for ARM are available as separate downloads from www.state-machine/arm.

NOTE: Even though this Application Note is based on the IAR toolset for ARM, the provided explanations are applicable to most toolsets supporting ARM7/ARM9 processors.

1.1 About QP-nano™

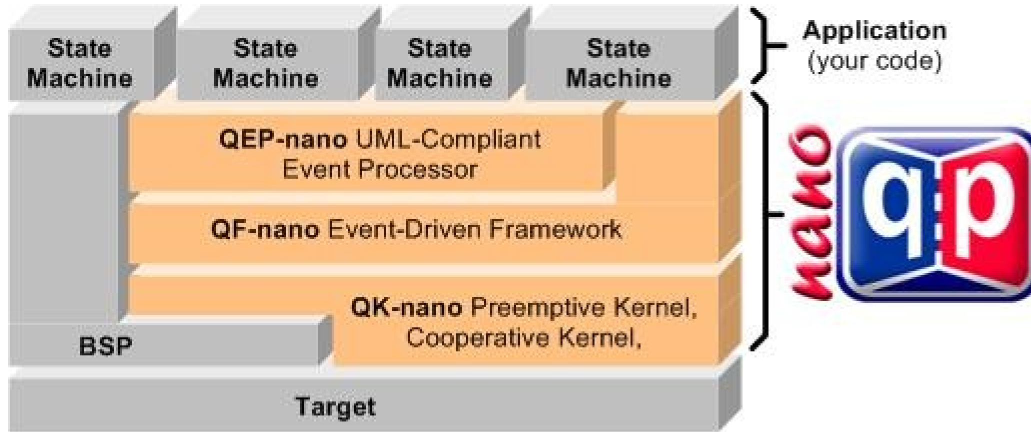


QP-nano™ is a generic, portable, ultra-lightweight, event-driven framework signed specifically for low-end 8- and 16-bit MCU. As shown in Figure 1, QP-nano consists of a universal UML-compliant event processor (QEP-nano), a portable real-time framework (QF-nano), and a tiny run-to-completion kernel (QK-nano). The ultra-lightweight **QP-nano** requires only 1-2KB of code and just several bytes of RAM.

QP-nano enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (UML statecharts) directly in C **without big tools**. QP-nano is described in the book "*Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*" [PSiCC2] (Newnes, 2008).

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 1 QP-nano components and their relationship with the target hardware, board support package (BSP), and the application



QP-nano runs on a bare-metal MCU, completely **replacing** a conventional RTOS, which typically cannot fit into available RAM of low-end MCUs. QP-nano includes a simple non-preemptive scheduler and a fully preemptive kernel (QK-nano). QK-nano is smaller and faster than most traditional preemptive kernels or RTOS, yet offers fully deterministic, preemptive execution of embedded applications. QP-nano can manage up to 8 concurrently executing tasks structured as state machines (called active objects in UML).

1.2 About the QP-nano™ Port to ARM7/9

The ARM7/9 cores are a quite complicated processors in that they support two *operating states*: ARM state, which executes 32-bit, word-aligned ARM instructions, and THUMB state, which operates with 16-bit, halfword-aligned THUMB instructions. On top of this, the CPU has several *operating modes*, such as USER, SYSTEM, SUPERVISOR, ABORT, UNDEFINED, IRQ, and FIQ. Each of these operating modes differs in visibility of registers (register banking) and sometimes privileges to execute instructions.

All these options mean that a designer must make several choices about the use of the ARM processor. This Application Note makes the following choices and assumptions:

1. The ARM processor executes in both ARM and THUMB states, meaning that some parts of the code are compiled to ARM and others to THUMB instruction sets, and calls between ARM and THUMB functions are allowed. Such approach is supported by the "`interwork`" option of the ARM compilers and linkers. This choice is optimal for most ARM-based microcontrollers, where large parts of the code execute from slower Flash ROM that in many cases is only 16-bit wide. The higher code density of the THUMB instruction set in such cases improves performance compared to ARM, even though THUMB is a less powerful instruction set.
2. The ARM processor operates in the **SYSTEM mode** (`CPSR[0:4] = 0x1F`) while processing task-level code, and briefly switches to the IRQ mode (`CPSR[0:4] =` 0x12) or FIQ mode (`CPSR[0:4] =` 0x11) to process IRQ or FIQ interrupts, respectively. The System mode is used for its ability to execute the `MSR/MRS` instructions necessary to quickly lock and unlock interrupts. NOTE: The SYSTEM mode is the default mode assumed by the IAR compiler for execution of applications.
3. The ARM processor uses only **single stack** (the USER/SYSTEM stack) for all tasks, interrupts, and exceptions. The private (banked) stack pointers in SUPERVISOR, ABORT, UNDEFINED, IRQ, and FIQ modes are used only as working registers, but not to point to the private stacks. This means that you **don't need** to allocate any RAM for the SUPERVISOR, ABORT, UNDEFINED, IRQ, or the FIQ

Copyright © Quantum Leaps, LLC. All Rights Reserved.

stacks and you don't need to initialize these stack pointers. The only stack you need to allocate and initialize is the USER/SYSTEM stack.

4. The application can use both IRQ and FIQ modes for processing of interrupts. The FIQ can preempt the IRQ. The interrupt locking policy includes locking simultaneously both IRQ and FIQ.

1.3 Licensing QP-nano

The **Generally Available (GA)** distributions of the QP-nano™ state machine frameworks available for download from the www.state-machine.com/downloads website is offered with the following two licensing options:

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file `GPL.TXT` included in the packaging of every Quantum Leaps software distribution. The GPL *open source* license allows you to use the software at no charge under the condition that if you redistribute the original software or applications derived from it, the complete source code for your application must be also available under the conditions of the GPL (GPL Section 2[b]).
- One of several Quantum Leaps commercial licenses, which are designed for customers who wish to retain the proprietary status of their code and therefore cannot use the GNU General Public License. The customers who license Quantum Leaps software under the commercial licenses do not use the software under the GPL and therefore are not subject to any of its terms.

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

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 Directories and Files

The code for the QP-nano port to ARM is available as part of any QP-nano Development Kit (QDK-nano) for ARM. The QDKs-nano assume that the generic platform-independent QP-nano™ distribution has been installed. The code of the ARM port is organized according to the Application Note: "QP_Directory_Structure". Specifically, for this port the files are placed in the following directories:

Listing 1 Directories and files after installing QP-nano baseline code and the QDK-nano (QDK-nano-ARM-AT91SAM7 in this case). The highlighted files are part of the QP-nano port to ARM

```
<qpn>/                     - QP-nano Root Directory
  +-examples/              - QP-nano examples
  | +-arm\                 - examples for ARM
  | | +-iar\               - examples compiled with the IAR compiler
  | | | +-dpp-at91sam7s-ek\ - DPP example for AT91SAM7S-EK (non-preemptive)
  | | | | +-dbg\           - directory containing the debug build
  | | | | +-rel\           - directory containing the release build
  | | | | | +-at91mc_cstartup.s - AT91 startup code in assembly
  | | | | | +-at91SAM7S64.icf - IAR ARM linker command file
  | | | | | +-bsp.h         - Board Support Package include file
  | | | | | +-bsp.c         - Board Support Package implementation
  | | | | | +-dpp.h
  | | | | | +-main.c
  | | | | | +-philo.c
  | | | | | +-table.c
  | | | | | +-qpn_port.h    - QP-nano port
  | | | | | +-qfn_port.s    - QF-nano port in assembly
  | | | | | +-dpp.eww       - IAR workspace for the DPP application example
  | | | |
  | | | +-dpp-qk-at91sam7s-ek\ - DPP example for AT91SAM7S-EK (preemptive with QK)
  | | | | +-dbg\           - directory containing the debug build
  | | | | | +-dpp-qk.out  - image of the application
  | | | | | +-dpp-qk.map  - map file of the application
  | | | | | +-at91mc_cstartup.s - AT91 startup code in assembly
  | | | | | +-at91SAM7S64.icf - IAR ARM linker command file
  | | | | | +-bsp.h         - Board Support Package include file
  | | | | | +-bsp.c         - Board Support Package implementation
  | | | | | +-dpp.h
  | | | | | +-main.c
  | | | | | +-philo.c
  | | | | | +-table.c
  | | | | | +-qpn_port.h    - QP-nano port
  | | | | | +-qkn_port.s    - QK-nano port in assembly
  | | | | | +-dpp-qk.eww    - IAR workspace for the DPP application example
  |
  +-include\                - subdirectory containing the QP-nano public interface
  | +-qassert.h             - embedded-systems-friendly assertions used in QP-nano
  | +-qepn.h                - The platform-independent QEP-nano header file
  | +-qfn.h                 - The platform-independent QF-nano header file
  | +-qkn.h                 - The platform-independent QK-nano header file
  |
  +-source/                - QP-nano source files
  | +-qepn.c               - QEP-nano
  | +-qfn.c                - QF-nano
  | +-qkn.c                - QK-nano (required only in QK-nano configuration)
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3 Non-Preemptive Configuration of QP-nano

The example of using QP-nano with the cooperative "Vanilla" kernel built into the QF-nano is located in the directory: `<qpn>\examples\arm\iar\dpp-at91sam7s-ek\`. This section describes the generic QP-nano configuration, which consist of the `qpn_port.h` header file and `qfn_port.s` ARM-specific port in assembly. The board-specific elements are common for both the non-preemptive and preemptive (QK-nano) configuration and will be covered in Section 4.

3.1 Configuration and Customizing QP-nano (qpn_port.h)

You configure and customize QP-nano through the header file `qpn_port.h`, which is included by the QP-nano source files (`qepn.c` and `qfn.c`) as well as in all your application C modules.

NOTE: The QP-nano port to the cooperative "Vanilla" kernel `qpn_port.h` is generic and should not need to change (except for the `QF_MAX_ACTIVE` definition) for other ARM systems.

```
    #ifndef qpn_port_h
    #define qpn_port_h

(1) #define Q_NFSM
(2) #define Q_PARAM_SIZE            4
(3) #define QF_TIMEEVT_CTR_SIZE     2

    /* maximum # active objects--must match EXACTLY the QF_active[] definition  */
(4) #define QF_MAX_ACTIVE           6

                /* fast unconditional interrupt locking/unlocking for ARM state */
(5) #define QF_INT_LOCK_32()        __asm("MSR cpsr_c,#(0x1F | 0x80 | 0x40)")
(6) #define QF_INT_UNLOCK_32()      __asm("MSR cpsr_c,#(0x1F)")

(7) #if (__CPU_MODE__ == 1)                                       /* THUMB mode? */

                                     /* interrupt locking policy for task-level */
(8)     #define QF_INT_LOCK()       QF_int_lock_SYS()
(9)     #define QF_INT_UNLOCK()     QF_int_unlock_SYS()

(10)    void QF_int_lock_SYS(void);
(11)    void QF_int_unlock_SYS(void);

                                      /* interrupt locking policy for ISR-level */
(12)    #define QF_ISR_NEST
(13)    #define QF_ISR_KEY_TYPE     unsigned long
(14)    #define QF_ISR_LOCK(key_)   ((key_) = QF_isr_lock_SYS())
(15)    #define QF_ISR_UNLOCK(key_) (QF_isr_unlock_SYS(key_))

(16)    QF_ISR_KEY_TYPE QF_isr_lock_SYS(void);
(17)    void QF_isr_unlock_SYS(QF_ISR_KEY_TYPE key);

(18) #elif (__CPU_MODE__ == 2)                                      /* ARM mode? */

                                     /* interrupt locking policy for task-level */
(19)    #define QF_INT_LOCK()       QF_INT_LOCK_32()
(20)    #define QF_INT_UNLOCK()     QF_INT_UNLOCK_32()
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
                                    /* interrupt locking policy for ISR-level */
(21)     #define QF_ISR_NEST
(22)     #define QF_ISR_KEY_TYPE     unsigned long
(23)     #define QF_ISR_LOCK(key_)   do { \
(24)         (key_) = __get_CPSR(); \
(25)         QF_INT_LOCK_32(); \
         } while (0)
(26)     #define QF_ISR_UNLOCK(key_) __set_CPSR(key_)

         #include <intrinsics.h>              /* for  __get_CPSR()/__set_CPSR() */

     #else

         #error Incorrect __CPU_MODE__. Must be ARM or THUMB.

     #endif
(27) void QF_reset(void);
     void QF_undef(void);
     void QF_swi(void);
     void QF_pAbort(void);
     void QF_dAbort(void);
     void QF_reserved(void);

(28) void QF_irq(void);
(29) void QF_fiq(void);

(30) void BSP_irq(void);
(31) void BSP_fiq(void);

(32) #include <stdint.h>    /* Exact-width integer types. WG14/N843 C99 Standard */

(33) #include "qepn.h"              /* QEP-nano platform-independent header file */
(34) #include "qfn.h"                /* QF-nano platform-independent header file */

     #endif                                                     /* qpn_port_h */
```

Listing 2 `qpn_port.h` header file for the non-preemptive QF-nano configuration and IAR compiler

(1) Defining the macro `Q_NFSM` eliminates the code for the simple non-hierarchical FSMs.

(2) The macro `Q_PARAM_SIZE` defines the size (in bytes) of the scalar event parameter. The allowed values are 0 (no parameter), 1, 2, or 4 bytes. If you don't define this macro in `qpn_port.h`, the default of 0 (no parameter) will be assumed.

(3) The macro `QF_TIMEEVT_CTR_SIZE` defines the size (in bytes) of the time event down-counter. The allowed values are 0 (no time events), 1, 2, or 4 bytes. If you don't define this macro in qpn_port.h, the default of 0 (no time events) will be assumed.

(4) You must define the `QF_MAX_ACTIVE` macro as the exact number of active objects used in the application. The provided value must be between 1 and 8 and must be consistent with the definition of the `QF_active[]` array in `main.c`.

(5-6) The macros `QF_INT_LOCK_32()/QF_INT_UNLOCK_32()` perform a very efficient unconditional interrupt locking/unlocking using just one MSR instruction with immediate argument. As indicated by the _32 suffix, these macros can only be called in the 32-bit ARM state, because only ARM state supports the MSR instruction.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

NOTE: In this QP-nano port to the ARM processors, the C-level code executes exclusively in the SYSTEM mode. The interrupt locking/unlocking functions that can be called only from the C code, might as well take advantage of the known CPU mode. Note also that the macros `QF_INT_LOCK_32()`/`QF_INT_UNLOCK_32()` are more efficient that the IAR intrinsic functions `__disable_interrupt()` and `__enable_interrupt()`, respectively. The IAR intrinsic functions must be generic to allow interrupt locking in any CPU mode (such as USER, SYSTEM, IRQ, FIQ, UNDEF, ABORT, SWI). In contrast, the `QF_INT_LOCK_32()`/`QF_INT_UNLOCK_32()` can be simple, because they are specific only to the SYSTEM mode.

(7) Interrupt locking/unlocking cannot be accomplished in the THUMB state, because the `MSR/MRS` instructions necessary to manipulate the CPSR register are not available in THUMB. Therefore, the only option to accomplish interrupt locking/unlocking from THUMB is to call an ARM function, such as `QF_int_lock_SYS()`/`QF_int_unlock_SYS()`.

NOTE: This QP-nano port does not use intrinsic functions, such as `__disable_interrupts()`/`__enable_interrupts()`, because they are more generic and consequently less optimal than the functions `QF_int_lock_SYS()`/`QF_int_unlock_SYS()`. The generic functions must work in all ARM modes, not just the SYSTEM mode, whereas the functions `QF_int_lock_SYS()`/`QF_int_unlock_SYS()` can be optimized to lock and unlock interrupts only in the SYSTEM mode.

(8-9) The interrupt locking/unlocking macros for the task-level resolve to the functions `QF_int_lock_SYS()` / `QF_int_unlock_SYS()`, respectively.

(10-11) The prototypes of the interrupt locking/unlocking functions for the task-level are declared. These functions are defined in assembly and are callable both in ARM and THUMB state.

(12)This QP-nano port to ARM allows nesting of interrupts.

(13)The macro `QF_ISR_KEY_TYPE` is defined, which means that ISRs use the “saving and restoring interrupt status” policy.

(14-15) The interrupt locking/unlocking macros for the ISR-level resolve to the functions `QF_isr_lock_SYS()` / `QF_isr_unlock_SYS()`, respectively.

(16-17) The prototypes of the interrupt locking/unlocking functions for the ISR-level are declared. These functions are defined in assembly and are callable the THUMB state (due to the –interworking option).

(18)For the compilation in the ARM state, the interrupt locking/unlocking macros can be defined inline, much more efficiently than in the THUMB state.

(19-20) For compilation in ARM state, the macros `QF_INT_LOCK()`/ `QF_INT_UNLOCK()` are defined very efficiently using the helper macros `QF_INT_LOCK_32()`/`QF_INT_UNLOCK_32()`, respectively

(21)This QP-nano port to ARM allows nesting of interrupts.

(22)The macro `QF_ISR_KEY_TYPE` is defined, which means that ISRs use the “saving and restoring interrupt status” policy.

(23) The interrupt locking macro for the ISR-level is defined inline in the ARM state.

(24) The interrupt key holds the CPSR value (actually only the control-bits of it), which is obtained by means of the intrinsic function `__get_CPSR()`. In the ARM state, this function expands to a single machine instruction `MRS r?,CPSR_c`.

(25) After saving the CPSR, the interrupts are efficiently locked with macro `QF_INT_LOCK_32()` described in step (8) above.

(26) The interrupt unlocking macro for the ISR-level restores the CPSR from the saved interrupt key value. The intrinsic function `__set_CPSR()` expands to a single machine instruction `MSR CPSR_c,r?`.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(27)All ARM exceptions are handled in these functions defined in assembly. The exception handlers encapsulate the various ARM modes (e.g., UNDEF, SWI, ABORT, IRQ, FIQ) and hide them from the application-level software. Application-level uses exclusively the SYSTEM mode.

(28-29) The most important exception handlers are the assembler "wrapper" functions for IRQ and FIQ. These "wrapper" functions encapsulate the IRQ and FIQ modes and hide them from the application-level software. Application-level uses exclusively the SYSTEM mode.

(30-31) The BSP functions encapsulate the particular interrupt controller. These functions are described in the upcoming Section 3.2.3.

(32)The IAR compiler provides the C99-standard exact-width integer types are defined in the standard `<stdint.h>` header file.

(33)The `qpn_port.h` must include the QEP-nano event processor interface `qepn.h`.

(34)The `qpn_port.h` must include the QF-nano real-time framework interface `qfn.h`.

3.2 ARM-Specific QF-nano Port (qfn_port.s)

Due to the complexity of the ARM core (see Section Error: Reference source not found), the QP-nano port to ARM requires some assembly programming. The assembly module `qfn_port.s` defines the task-level and ISR-level interrupt locking and the ARM exception handlers.

The following sub-sections covers the main elements of the `qfn_port.s` implementation, starting with the critical section, through interrupt "wrapper" functions in assembly, and finally all other exception "wrapper" functions.

NOTE: The QP-nano port to the cooperative "Vanilla" kernel `qfn_port.s` is generic and should not need to change for other ARM systems.

3.2.1 The Critical Section Implementation

This QP-nano port uses the simple unconditional interrupt locking and unlocking policy for the task-level and the policy of saving and restoring the interrupt status described for the ISR-level. This ISR policy allows for nesting critical sections, where the interrupts status is preserved across the critical section in a temporary stack variable. In other words, upon the exit from a critical section the interrupts are actually unlocked in the `QF_isr_lock_SYS()` function only if they were unlocked before the matching `QF_isr_unlock_SYS()` function.

As discussed in the upcoming Section "The FIQ Wrapper Function in Assembly", you'll typically not enable the ARM core interrupts during the FIQ interrupt processing, so the described critical section nesting will occur.

The critical section in QF-nano is defined as follows in `qfn_port.s`:

Listing 3 Task-level and ISR-level critical section definitions in `qfn_port.s`.

```
    ;-----------------------------------------------------------------------------
    ; Task-level Interrupt locking/unlocking
    ;-----------------------------------------------------------------------------
(1)     SECTION .textrw:DATA:NOROOT(2)
        PUBLIC  QF_int_lock_SYS, QF_int_unlock_SYS

(2)     CODE32
    QF_int_lock_SYS:
(3)     MSR     cpsr_c,#(SYS_MODE | NO_INT) ; disable both IRQ/FIQ
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
(4)     BX      lr

    QF_int_unlock_SYS:
(5)     MSR     cpsr_c,#(SYS_MODE)  ; enable both IRQ/FIQ
        BX      lr

    ;-----------------------------------------------------------------------------
    ; ISR-level Interrupt locking/unlocking
    ;-----------------------------------------------------------------------------
        PUBLIC  QF_isr_lock_SYS, QF_isr_unlock_SYS

        CODE32
    QF_isr_lock_SYS:
(6)     MRS     r0,cpsr             ; get the original CPSR in r0 to return
(7)     MSR     cpsr_c,#(SYS_MODE | NO_INT) ; disable both IRQ/FIQ
        BX      lr                  ; return the original CPSR in r0

    QF_isr_unlock_SYS:
(8)     MSR     cpsr_c,r0           ; restore the original CPSR from r0
        BX      lr                  ; return to ARM or THUMB
```

(1) The module is declared in the section `.textrw` located in the DATA area. As described in “ARM® IAR C/C++ Compiler Reference Guide” [IAR 08a], the `.textrw` section is used for functions executing in RAM. For almost all ARM-based MCUs, the code executing from RAM is significantly faster (sometimes as much as 3-4 times faster) than code executing from ROM due to wait states necessary to access slow, and often only 16-bit wide Flash ROM. At the cost of just several bytes of RAM you get significant performance boost for the “hot-spot” interrupt locking/unlocking code.

(2) The functions require the 32-bit ARM state to be able to use the `MSR`/`MRS` instructions.

(3) Both IRQ and FIQ interrupts are disabled simultaneously by means of the most efficient immediate move to the `CPSR_c` (control bits only). Using this efficient instruction is possible, because the mode bits are known to be SYSTEM. Also, the T-bit is known to be cleared since this code executes in the ARM state.

NOTE: In this QP-nano port to ARM, the C-level code executes exclusively in the SYSTEM mode. The interrupt locking/unlocking functions that can be called only from the C code, might as well take advantage of the known CPU mode. Because these specific interrupt locking/unlocking functions assume the SYSTEM mode, the names of these functions have been chosen to reflect this fact (`QF_int_lock_SYS()`, `QF_int_unlock_SYS()`).

(4) The function returns via the `BX` instruction, which causes the T-bit to be set in the `CPSR_c` if the return address is a THUMB label.

(5) Both IRQ and FIQ interrupts are unconditionally enabled by means of the most efficient immediate move to the `CPSR_c` (control bits only). Using this efficient instruction is possible, because the mode bits are known to be SYSTEM. Also, the T-bit is known to be cleared since this code executes in the ARM state.

(3) The original value of `CPSR` is moved to `r0`, which is then returned from the `QF_int_lock` function.

(4) The IRQ and FIQ are locked simultaneously by means of the most efficient immediate move to the `CPSR_c`. Using this efficient instruction is possible, because the mode bits are known to be SYSTEM. Also, the T-bit is known to be cleared since this code executes in the ARM state.

(5) The return from the function occurs via the `BX` instruction, which causes the T-bit to be set in the `CPSR_c` if the return address is a THUMB label.

(6) The original value of `CPSR` is moved to `r0` to be returned from the function.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(7) The IRQ and FIQ are locked simultaneously by means of the most efficient immediate move to the `CPSR_c` (control bits only).

(8) The original value of `CPSR`, which is passed in the argument of the `QF_isr_unlock_SYS` function is moved to `CPSR_c`. At this point interrupts are re-enabled if they were enabled before the matching call to `QF_int_lock_SYS`, or they remain disabled, if they were disabled before the call.

3.2.2 Discussion of the Critical Section

When the IRQ line of the ARM processor is asserted, and the I bit (bit `CPSR[7]`) is cleared, the core ends the instruction currently in progress and then starts the IRQ sequence, which performs the following actions ("ARM Architecture Reference Manual, 2nd Edition", Section 2.6.6 [Seal 00]):

- `R14_irq     =` address of next instruction to be executed + 4
- `SPSR_irq    = CPSR`
- `CPSR[4:0] = 0b10010` (enter IRQ mode)
- `CPSR[7]     = 1,` NOTE: `CPSR[6]` is unchanged
- `PC          = 0x00000018`

The ARM Technical Note "What happens if an interrupt occurs as it is being disabled?" [ARM 05], points out two potential problems. Problem 1 is related to using a particular subroutine as an IRQ handler and as a regular subroutine called outside of the IRQ scope and then inspecting the `SPSR_IRQ` register to detect in which context the handler function is called. This is impossible in this QF port, because the C-level IRQ handler is always called in the SYSTEM mode, where the application programmer has no access to the `SPSR_IRQ` register. Problem 2 described in the ARM Note [ARM 05] is more applicable and relates to the situation when both IRQ and FIQ are disabled simultaneously, which is actually the case in this port (see Listing 3(3 and 7)). If the IRQ is received during the `CPSR` write, FIQs could be disabled for the execution time of the IRQ handler. One of the workarounds recommended in the ARM Note is to explicitly enable FIQ very early in the IRQ handler. This is exactly done in the `QF_irq` assembler "wrapper" discussed later in this document.

For completeness, this discussion should mention the Atmel Application Note "Disabling Interrupts at Processor Level" [Atmel 98a], which describes another potential problem that might occur when the IRQ or FIQ interrupt is recognized exactly at the time that it is being disabled. While the ARM core is executing the "`MSR cpsr_c,#(SYS_MODE | NO_INT)`" instruction (see Listing 3(3 and 7)), the interrupts are disabled only on the **next** clock cycle. If, for example, an IRQ interrupt occurs exactly during the execution of this instruction, the `CPSR[7]` bit is set **both** in the `CPSR` and `SPSR_irq` (Saved Program Status Register) and the IRQ **is** entered. The problem arises when the IRQ or FIQ handler would manipulate the I or F bits in the `SPSR` register. This QF port provides the IRQ and FIQ "wrappers" in assembly that never change any bits in the `SPSR`. This approach corresponds to the Workaround 1 described in the Atmel Application Note [Atmel 98a], which is safe here because the application programmer really has no way of accessing the `SPSR` register.

NOTE: In this QP-nano port to ARM, the C-level code executes exclusively in the SYSTEM mode. Therefore, the banked registers `SPSR_irq` or `SPSR_fiq` aren't really visible or accessible to the application programmer. Also accessing these registers would necessarily require assembly programming, since no standard compiler functions use these registers.

3.2.3 Handling Interrupts

This QP-nano port to ARM can work with or without an interrupt controller, such as the Atmel's Advanced Interrupt Controller (AIC), Philips' Vectored Interrupt Controller (VIC), and others.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

When used with an interrupt controller, the “vanilla” port assumes **no** “auto-vectoring”, which is described for example in the Atmel Application Note “Interrupt Management: Auto-vectoring and Prioritization” [Atmel 98b].

SIDE NOTE: Auto-vectoring occurs when the following LDR instruction is located at the address 0x18 for the IRQ (this example pertains to the Atmel’s AIC):

```
ORG 0x18

LDR pc,[pc,#-0xF20]
```

When an IRQ occurs, the ARM core forces the PC to address 0x18 and executes the LDR pc,[pc,#-0xF20] instruction. When the instruction at address 0x18 is executed, the effective address is:

```
0x20 - 0xF20 = 0xFFFFF100
```

(0x20 is the value of the PC when the instruction at address 0x18 is executed due to pipelining of the ARM core).

This causes the ARM core to load the PC with the value read from the AIC_IVR register located at 0xFFFFF100. The read cycle causes the AIC_IVR register to return the address of the currently active interrupt service routine. Thus, the single `LDR pc,[pc,#-0xF20]` instruction has the effect of directly jumping to the correct ISR, which is called auto-vectoring.

Instead of “auto-vectoring”, both the “vanilla” and QK ports to ARM assume that the low-level interrupt handlers are directly invoked upon hardware interrupt requests. The IRQ/FIQ vectors (at 0x18 and 0x1C, respectively) load the `PC` with the address of the “wrapper” routines written in assembly.

Figure 2 Typical ARM system with an interrupt controller (external to the ARM core).

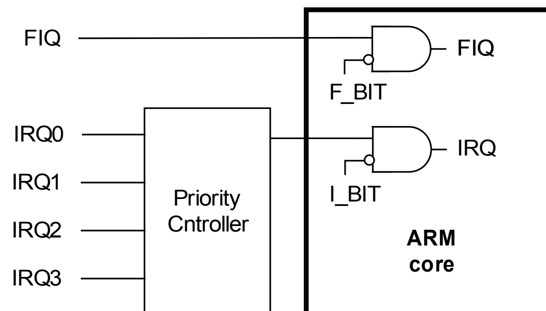


Figure 2 shows a typical ARM-based system with an interrupt controller. Typically, the interrupt prioritization is performed only with respect to the IRQ line, while the FIQ line is routed directly to the ARM core. The ARM core itself performs a 2-level interrupt prioritization between the FIQ and IRQ, whereas FIQ is higher priority than the IRQ. This second-level of prioritization is performed by the I and F bits of the CPSR register, which are also used for interrupt locking and unlocking policy.

Even though “auto vectoring” is not used, vectoring to a specific interrupt handler can still occur, but is done later, at the C-level interrupt handler functions implemented typically in the Board Support Package. Examples of such BSP functions for the popular interrupt controllers are provided later in this manual.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Figure 3 Interrupt processing in the “vanilla” port to ARM with the Atmel’s AIC interrupt controller

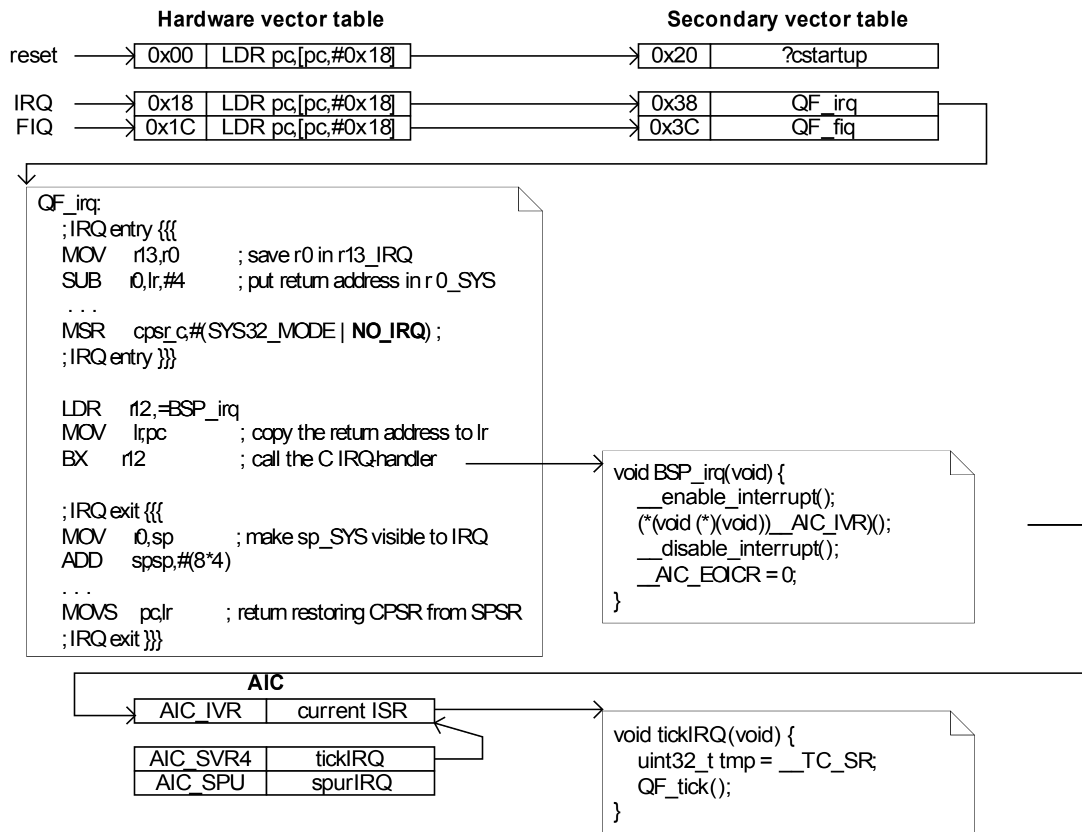


Figure 3 shows the interrupt processing sequence in the presence of an interrupt controller (Atmel’s AIC is assumed in this example). The ARM vector addresses 0x18 and 0x1C point to the assembler “wrapper” functions `QF_irq` and `QF_fiq`, respectively. Each of these “wrapper” functions, for example `QF_irq`, performs context save, switches to the SYSTEM mode, and invokes a C-level function `BSP_irq` (or `BSP_fiq` for the FIQ interrupt). BSP_irq encapsulates the particular interrupt controller, from which it explicitly obtains the interrupt vector. Because the interrupt controller is used in this case, it must be initialized with the addresses of all used interrupt service routines (ISRs), such as `tickIRQ()` shown in Figure 3. Please note that these ISRs are regular C-functions and not `__irq` type functions because the interrupt entry and exit code is already provided in the assembly “wrapper” functions `QF_irq` and `QF_fiq`.

3.2.4 The IRQ “Wrapper” Function in Assembly

The “vanilla” QF-nano port provides interrupt “wrapper” function `QF_irq()` for handling the IRQ-type interrupts. The function is coded entirely in assembly, and is located in the file `qfn_port.s`.

Listing 4 The `QF_irq` assembly wrapper for the “vanilla” QF-nano port (`qfn_port.s`).

```
    ;--------------------------------------------------------------------------
    ; IRQ assembly wrapper
    ;--------------------------------------------------------------------------

(1)     SECTION .textrw:DATA:NOROOT(2)
        PUBLIC  QF_irq
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
        EXTERN  BSP_irq

 (2)    CODE32
    QF_irq:
    ; IRQ entry {{{
 (3)    MOV     r13,r0              ; save r0 in r13_IRQ
 (4)    SUB     r0,lr,#4            ; put return address in r0_SYS
 (5)    MOV     lr,r1               ; save r1 in r14_IRQ (lr)
 (6)    MRS     r1,spsr             ; put the SPSR in r1_SYS

 (7)    MSR     cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM, no IRQ, but FIQ enabled!
 (8)    STMFD   sp!,{r0,r1}         ; save SPSR and PC on SYS stack
 (9)    STMFD   sp!,{r2-r3,r12,lr}  ; save APCS-clobbered regs on SYS stack
(10)    MOV     r0,sp               ; make the sp_SYS visible to IRQ mode
(11)    SUB     sp,sp,#(2*4)        ; make room for stacking (r0_SYS, r1_SYS)

(12)    MSR     cpsr_c,#(IRQ_MODE | NO_IRQ) ; IRQ mode, IRQ disabled
(13)    STMFD   r0!,{r13,r14}       ; finish saving the context (r0_SYS,r1_SYS)

(14)    MSR     cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM mode, IRQ disabled
    ; IRQ entry }}}

        ; NOTE: BSP_irq might re-enable IRQ interrupts (the FIQ is enabled
        ; already), if IRQs are prioritized by the interrupt controller.
        ;
(15)    LDR     r12,=BSP_irq
(16)    MOV     lr,pc               ; copy the return address to link register
(17)    BX      r12                 ; call the C IRQ-handler (ARM/THUMB)


    ; IRQ exit {{{
(18)    MSR     cpsr_c,#(SYS_MODE | NO_INT) ; make sure IRQ/FIQ are disabled
(19)    MOV     r0,sp               ; make sp_SYS visible to IRQ mode
(20)    ADD     sp,sp,#(8*4)        ; fake unstacking 8 registers from sp_SYS

(21)    MSR     cpsr_c,#(IRQ_MODE | NO_INT) ; IRQ mode, both IRQ/FIQ disabled
(22)    MOV     sp,r0               ; copy sp_SYS to sp_IRQ
(23)    LDR     r0,[sp,#(7*4)]      ; load the saved SPSR from the stack
(24)    MSR     spsr_cxsf,r0        ; copy it into spsr_IRQ

(25)    LDMFD   sp,{r0-r3,r12,lr}^  ; unstack all saved USER/SYSTEM registers
(26)    NOP                         ; can't access banked reg immediately
(27)    LDR     lr,[sp,#(6*4)]      ; load return address from the SYS stack
(18)    MOVS    pc,lr               ; return restoring CPSR from SPSR
    ; IRQ exit }}}
```

(1) The IRQ wrapper `QF_irq` is defined in section `.textrw`, declared as **DATA** to be placed in RAM for fastest execution. Such time-critical ARM code is best executed from 32-bit wide memory with minimal number of wait states.

(2) The IRQ handler must be written in the 32-bit instruction set (ARM), because the ARM core automatically switches to the ARM state when IRQ is recognized.

(3) The IRQ stack is not used, so the banked stack pointer register `r13_IRQ` (`sp_IRQ`) is used as a scratchpad register to temporarily hold `r0` from the SYSTEM context.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

NOTE: As part of the IRQ startup sequence, the ARM processor sets the I bit in the CPSR (`CPSR[7] = 1`), but leaves the F bit unchanged (typically cleared), meaning that further IRQs are disabled, but FIQs are not. This means that FIQ can be recognized while the ARM core is in the IRQ mode. This IRQ handler does not disable the FIQ and preemption, and the provided FIQ handler can safely preempt this IRQ until FIQs are explicitly disabled later in the sequence.

(4) Now r0 can be clobbered with the return address from the interrupt that needs to be saved to the SYSTEM stack.

(5) At this point the banked `lr_IRQ` register can be reused to temporarily hold r1 from the SYSTEM context.

(6) Now r1 can be clobbered with the value of spsr_IRQ register (Saved Program Status Register) that needs to be saved to the SYSTEM stack.

(7) Mode is changed to SYSTEM with IRQ interrupt disabled, but FIQ explicitly enabled. This mode switch is performed to get access to the SYSTEM registers.

NOTE: The F bit in the `CPSR` is intentionally cleared at this step (meaning that the FIQ is explicitly enabled). Among others, this represents the workaround for the Problem 2 described in ARM Technical Note "What happens if an interrupt occurs as it is being disabled?" [ARM 05].

(8) The SPSR register and the return address from the interrupt (PC after the interrupt) are pushed on the SYSTEM stack.

(9) All registers (except `r0` and `r1`) clobbered by the AAPCS (ARM Architecture Procedure Call Standard) [ARM 06] are pushed on the SYSTEM stack.

(10) The SYSTEM stack pointer is placed in `r0` to be visible in the IRQ mode.

(11) The SYSTEM stack pointer is adjusted to make room for two more registers of the saved IRQ context. By adjusting the SYSTEM stack pointer, the IRQ handler can still keep FIQ enabled without the concern of corrupting the SYSTEM stack space reserved for the IRQ context.

(12) The mode is switched back to IRQ with IRQ interrupt disabled, but FIQ still enabled. This is done to get access to the rest of the context sitting in the IRQ-banked registers.

(13) The context is entirely saved by pushing the original `r0` and `r1` (still sitting in the banked IRQ Registers `r14_IRQ` and `r13_IRQ`, respectively) to the SYSTEM stack. At this point the saved SYSTEM stack frame contains 8 registers and looks as follows (this is exactly the ARM v7-M interrupt stack frame [ARM 06]):

```
high memory
                   SPSR
                   PC (return address)
                   LR
       |           R12
       v           R3
     stack         R2
     growth        R1
                   R0 <-- sp_SYS
low memory
```

(14) The mode is switched once more to SYSTEM with IRQ disabled and FIQ enabled. Please note that the stack pointer `sp_SYS` points to the top of the stack frame, because it has been adjusted after the first switch to the SYSTEM mode at line (11).

(15-17) The board-specific function `BSP_irq()` is called to perform the interrupt processing at the application-level. Please note that `BSP_irq()` is now a regular C function in ARM or Thumb.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

Typically, this function uses the silicon-vendor specific interrupt controller (such as the Atmel AIC) to vector into the current interrupt.

NOTE: The `BSP_irq()` function is entered with IRQ disabled (and FIQ enabled), but it can internally unlock IRQs, if the MCU is equipped with an interrupt controller that performs prioritization of IRQs in hardware.

(18) All interrupts (IRQ and FIQ) are locked to execute the following instructions atomically.

(19) The `sp_SYS` register is moved to r0 to make it visible in the IRQ mode.

(20) Before leaving the SYSTEM mode, the `sp_SYS` stack pointer is adjusted to un-stack the whole interrupt stack frame of 8 registers. This brings the SYSTEM stack to exactly the same state as before the interrupt occurred.

NOTE: Even though the SYSTEM stack pointer is moved up, the stack contents have not been restored yet. At this point it's critical that the interrupts are completely locked, so that the stack contents above the adjusted stack pointer cannot be corrupted.

(21) The mode is changed to IRQ with IRQ and FIQ interrupts locked to perform the final return from the IRQ.

(22) The SYSTEM stack pointer is copied to the banked `sp_IRQ`, which thus is set to point to the top of the SYSTEM stack

(23-24) The value of `SPSR` is loaded from the stack (please note that the SPSR is now 7 registers away from the top of the stack) and placed in `SPSR_irq`.

(25) The 6 registers are popped from the SYSTEM stack. Please note the special version of the `LDM` instruction (with the '^' at the end), which means that the registers are popped from the SYSTEM/USER stack. Please also note that the special LDM(2) instruction does not allow the write-back, so the stack pointer is not adjusted. (For more information please refer to Section "LDM(2)" in the "ARM Architecture Reference Manual" [Seal 00].)

(26) It's important not to access any banked register after the special LDM(2) instruction.

(27) The return address is retrieved from the stack. Please note that the return address is now 6 registers away from the top of the stack.

(28) The interrupt return involves loading the PC with the return address and the `CPSR` with the `SPSR`, which is accomplished by the special version of the `MOVS pc,lr` instruction.

3.2.5 The FIQ "Wrapper" Function in Assembly

NOTE: Generally, you should avoid, if only possible, using the FIQ as general purpose interrupt, because the FIQ interrupt is NOT prioritized by the interrupt controller in most ARM chips (see Figure 2). Handling the FIQ as general purpose interrupt is actually more complicated (and thus actually more expensive) than handling of IRQ-type interrupts that typically are prioritized in the interrupt controller.

Perhaps the best use of the FIQ is as a Non-Maskable-Interrupt (NMI) for handling very special functions. To use FIQ as a NMI, you must modify the critical section definition. Specifically, you would need to set only the I bit in the CPSR (see Listing 3):

```
MSR     cpsr_c,#(SYS_MODE | NO_IRQ) ; disable only IRQ in SYSTEM mode
```

However, please note that if you use FIQ as a NMI, you cannot use FIQ to call any QF services because the critical section never masks the FIQ and consequently such NMI can corrupt critical QF data.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

The “vanilla” QF port provides interrupt “wrapper” function `QF_fiq()` for handling the FIQ-type interrupts. The function is coded entirely in assembly, and is located in the file `qfn_port.s`.

Listing 5 The `QF_fiq` assembly wrapper for the “vanilla” QF port defined in `qfn_port.s`.

```
        SECTION .textrw:DATA:NOROOT(2)
        PUBLIC  QF_fiq
        EXTERN  BSP_fiq

        CODE32
    QF_fiq:
    ; FIQ entry {{{
(1)     MOV     r13,r0                  ; save r0 in r13_FIQ
        SUB     r0,lr,#4                ; put return address in r0_SYS
        MOV     lr,r1                   ; save r1 in r14_FIQ (lr)
        MRS     r1,spsr                 ; put the SPSR in r1_SYS

(2)     MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
        STMFD   sp!,{r0,r1}             ; save SPSR and PC on SYS stack
        STMFD   sp!,{r2-r3,r12,lr}      ; save APCS-clobbered regs on SYS stack
        MOV     r0,sp                   ; make the sp_SYS visible to FIQ mode
        SUB     sp,sp,#(2*4)            ; make room for stacking (r0_SYS, r1_SYS)

        MSR     cpsr_c,#(FIQ_MODE | NO_INT) ; FIQ mode, IRQ/FIQ disabled
        STMFD   r0!,{r13,r14}           ; finish saving the context (r0_SYS, r1_SYS)

        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
    ; FIQ entry }}}

        ; NOTE:
        ; Because FIQ is typically NOT prioritized by the interrupt controller
        ; BSP_fiq must not enable IRQ/FIQ to avoid priority inversions!
        ;
        LDR     r12,=BSP_fiq
        MOV     lr,pc                   ; store the return address
        BX      r12                     ; call the C FIQ-handler (ARM/THUMB)


    ; FIQ exit {{{                      ; both IRQ/FIQ disabled (see NOTE above)
        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; make sure IRQ/FIQ are disabled
        MOV     r0,sp                   ; make sp_SYS visible to FIQ mode
        ADD     sp,sp,#(8*4)            ; fake unstacking 8 registers from sp_SYS

        MSR     cpsr_c,#(FIQ_MODE | NO_INT) ; FIQ mode, IRQ/FIQ disabled
        MOV     sp,r0                   ; copy sp_SYS to sp_FIQ
        LDR     r0,[sp,#(7*4)]          ; load the saved SPSR from the stack
        MSR     spsr_cxsf,r0            ; copy it into spsr_FIQ

        LDMFD   sp,{r0-r3,r12,lr}^      ; unstack all saved USER/SYSTEM registers
        NOP                             ; can't access banked reg immediately
        LDR     lr,[sp,#(6*4)]          ; load return address from the SYS stack
        MOVS    pc,lr                   ; return restoring CPSR from SPSR
    ; FIQ exit }}}
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

The QF_fiq() "wrapper" shown in Listing 5 is very similar to the IRQ wrapper (Listing 4), except the FIQ mode is used instead of the IRQ mode. The following comments explain only the slight, but important differences in disabling interrupts and the responsibilities of the C-level handler BSP_fiq() function.

(1) The FIQ handler is always entered with both IRQ and FIQ disabled, so the FIQ mode is not visible in any other modes.

(2) The mode is switched to SYSTEM to get access to the SYSTEM stack pointer. Please note that both IRQ and FIQ interrupts are kept disabled throughout the FIQ handler.

(3) The C-function BSP_fiq() is called to perform the interrupt processing at the application-level. Please note that BSP_fiq() is now a regular C function in ARM or THUMB. Unlike the IRQ, the FIQ interrupt is often not covered by the priority controller, therefore the BSP_fiq() should NOT unlock interrupts.

NOTE: The BSP_fiq() function is entered with both IRQ and FIQ interrupts disabled and it should never enable any interrupts. Typically, the FIQ line to the ARM core does not have a priority controller, even though the IRQ line typically goes through a hardware interrupt controller.

In particular, the BSP_fiq() function must **NEVER** enable the IRQ interrupt, because this could corrupt the banked lr_IRQ register in case the FIQ nests on top of IRQ (which is unavoidable in the ARM processor architecture).

3.2.6 Other ARM Exception "Wrapper" Functions in Assembly

The "vanilla" QF port provides also assembly "wrapper" functions for all other ARM exceptions, which are: RESET, UNDEFINED INSTRUCTION, SOFTWARE INTERRUPT, PREFETCH ABORT, DATA ABORT, and the RESERVED exception. All these exception handlers are coded entirely in assembly, and are located in the file qfn_port.s.

The policy of handling the ARM hardware exceptions in QF is to raise an assertion, an assumption here being that no ARM exception should occur during normal program execution. You can easily substitute this standard behavior for selected ARM exceptions by simply initializing the ARM vector table to your own implementations.

Listing 6 ARM Exception "Wrapper" Functions in Assembly defined in qfn_port.s.

```
;-----------------------------------------------------------------------------
; Exception assembler wrappers
;-----------------------------------------------------------------------------

    PUBLIC  QF_reset
    PUBLIC  QF_undef
    PUBLIC  QF_swi
    PUBLIC  QF_pAbort
    PUBLIC  QF_dAbort
    PUBLIC  QF_reserved
    EXTERN  Q_onAssert

    SECTION .text:CODE:NOROOT(2)
    CODE32

QF_reset:
    LDR     r0,=Csting_reset
    B       QF_except
QF_undef:
    LDR     r0,=Csting_undef
    B       QF_except
QF_swi:
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
        LDR     r0,=Csting_swi
        B       QF_except
    QF_pAbort:
        LDR     r0,=Csting_pAbort
        B       QF_except
(1) QF_dAbort:
(2)     LDR     r0,=Csting_dAbort
(3)     B       QF_except
    QF_reserved:
        LDR     r0,=Csting_reserved
        B       QF_except
(4) QF_except:
        ; r0 is set to the string with the exception name
(5)     SUB     r1,lr,#4              ; set line number to the exception address
(6)     MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
(7)     LDR     r12,=Q_onAssert
(8)     MOV     lr,pc                 ; store the return address
(9)     BX      r12                   ; call the assertion-handler (ARM/THUMB)
        ; the assertion handler should not return, but in case it does
        ; hang up the machine in this endless loop
        B       .

        LTORG  ; strings enclosed in "" are zero-terminated
    Csting_reset:       DC8     "Reset"
    Csting_undef:       DC8     "Undefined"
    Csting_swi:         DC8     "Software Int"
    Csting_pAbort:      DC8     "Prefetch Abort"
    Csting_dAbort:      DC8     "Data Abort"
    Csting_reserved:    DC8     "Reserved"

        END
```

(1) All exceptions are handled uniformly, such as the PREFETCH ABORT exception.

(2) The first argument to the `Q_onAssert()` callback function is prepared in `r0`. This argument is a pointer to the C-string with the name of the exception.

(3-4) The common exception code is handled in at the `QF_except` label.

(5) The address of the exception is saved in `r1`, which is the second argument to the `Q_onAssert()` callback function .

(6) The mode is switched to SYSTEM with both IRQ and FIQ interrupts disabled.

(7-9) The `Q_onAssert()` callback function is called. Because the call happens via the `BX` instruction, the `Q_onAssert()` function can be in ARM or THUMB.

3.2.7 QP-nano Idle Processing Customization in QF_onIdle()

QP-nano can very easily detect the situation when no events are available, in which case `QF_run()` calls the `QF_onIdle()` callback. You can use `QF_onIdle()` to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QF_onIdle()` is called repetitively from the event loop whenever the event loop has no more events to process, in which case only an interrupt can provide new events. The `QF_onIdle()` callback is called with interrupts **locked**, because any enabled interrupt can posting new events to the active objects and thus invalidate the idle condition. If this would happen, some active objects would have urgent events to process, yet the CPU will enter the low-power sleep mode.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

The following Listing 7 shows the `QF_onIdle()` callback that puts AT91 MCU into the idle power-saving mode.

Listing 7 QF_onIdle() for the non-preemptive QF-nano configuration

```
(1) __arm __ramfunc void QF_onIdle(void)          {  /* NOTE: interrupts LOCKED */
    #ifdef NDEBUG     /* only if not debugging (power saving hinders debugging) */
(2)     AT91C_BASE_PMC->PMC_SCDR = 1;/* Power-Management: disable the CPU clock */
        /* NOTE: an interrupt starts the CPU clock again */
     #endif

(3)     QF_INT_UNLOCK_ARM();   /* unlock interrupts as soon as CPU clock starts */
    }
```

(1) The `QF_onIdle()` function is called with interrupts locked and must unlock interrupts internally.

(2) Writing 1 to the Power Module Control SCDR register stops the CPU clock. Note that interrupts remain locked at the ARM core level. The interrupts are not disabled at the interrupt controller level.

(3) An active interrupt re-starts the CPU clock. The code starts executing and unlocks interrupts at the ARM core level. Only at this point the interrupt can be serviced.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 Preemptive Configuration with QK-nano

This section describes how to use QP-nano with ARM7/ARM9-based MCUs with fully preemptive QK-nano real-time kernel. The benefit is very fast, fully deterministic task-level response and that responsiveness 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 (see Chapter 10 in [PSiCC2]). Also, 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-nano priority-ceiling mutex (again see see Chapter 10 in [PSiCC2]).

NOTE: QK-nano incurs less overhead and provides responsiveness exceeding that of any traditional multiple-stack preemptive kernel, at the fraction of the RAM/ROM footprint (see the ESD article "Build a Super Simple Tasker", [Samek+ 06])

The QP-nano port tp ARM with the QK-nano preemptive kernel is located in the directory: `<qpn>\-examples\arm\iar\dpp-`**`qk`**`-at91sam7s-ek\` and consists of the following files:

- `qpn_port.h` contains the platform-specific customization of QP-nano (the QP-port)
- `qkn_port.s` contains the ARM-specific port of the QK-nano kernel
- `bsp.h` contains the Board Support Package interface (BSP)
- `bsp.c` contains the implementation of the BSP, which includes all ISRs and all platform-specific QP-nano callbacks.
- `at91mc_cstartup.s` — startup code in assembly
- `at91SAM7S64.icf` — linker command file for executing the code out of Flash

4.1 Configuration and Customizing QP-nano

You configure and customize QP-nano through the header file `qpn_port.h`, which is included by the QP-nano source files (`qepn.c`, `qfn.c`, and `qkn.c`) as well as in all your application C modules. The following Listing 8 shows the `qpn_port.h` header file for the QK-nano port. Except for the highlighted fragments, the listing is identical as in the non-preemptive case (Listing 2)

Listing 8 `qpn_port.h` header file for the preemptive QK-nano configuration

```
     #define Q_NFSM
     #define Q_PARAM_SIZE            4
     #define QF_TIMEEVT_CTR_SIZE     2

     . . . (the same as in the non-preemptive configuration in Listing 2)

     void QF_reset(void);
     void QF_undef(void);
     void QF_swi(void);
     void QF_pAbort(void);
     void QF_dAbort(void);
     void QF_reserved(void);

(1)  void QK_irq(void);
(2)  void QK_fiq(void);
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
void BSP_irq(void);
void BSP_fiq(void);

#include <stdint.h>    /* Exact-width integer types. WG14/N843 C99 Standard */

#include "qepn.h"              /* QEP-nano platform-independent header file */
#include "qfn.h"                /* QF-nano platform-independent header file */
#include "qkn.h"           /* QK-nano platform-independent public interface */
```

(1-2) The QK-nano port uses different assembler “wrapper” functions for the IRQ and FIQ interrupts. The next section covers these “wrapper” functions in detail.

4.2 Handling Interrupts in QK-nano

Interrupt handling with QK-nano is also very similar as in the non-preemptive “vanilla” configuration, except that the assembler “wrapper” functions additionally invoke the QK-nano scheduler upon the interrupt exit to perform asynchronous preemptions (see Chapter 10 in [PSiCC2]). The following sections explain the IRQ and FIQ “wrapper” functions for QK-nano.

4.2.1 The IRQ “Wrapper” Function for QK

The QK-nano port provides interrupt “wrapper” function `QK_irq()` for handling the IRQ-type interrupts. The function is coded entirely in assembly and is located in the file `qkn_port.s`.

Listing 9 The QK_irq assembly wrapper for the QK port defined in qk_port.s.

```
        SECTION .textrw:CODE:NOROOT(2)
        PUBLIC  QK_irq
        EXTERN  BSP_irq
        EXTERN  QK_intNest_, QK_schedule_
        CODE32

    QK_irq:
    ; IRQ entry {{{
        MOV     r13,r0               ; save r0 in r13_IRQ
        SUB     r0,lr,#4             ; put return address in r0_SYS
        MOV     lr,r1                ; save r1 in r14_IRQ (lr)
        MRS     r1,spsr              ; put the SPSR in r1_SYS

        MSR     cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM, no IRQ, but FIQ enabled!
        STMFD   sp!,{r0,r1}          ; save SPSR and PC on SYS stack
        STMFD   sp!,{r2-r3,r12,lr}   ; save APCS-clobbered regs on SYS stack
        MOV     r0,sp                ; make the sp_SYS visible to IRQ mode
        SUB     sp,sp,#(2*4)         ; make room for stacking (r0_SYS, r1_SYS)

        MSR     cpsr_c,#(IRQ_MODE | NO_IRQ) ; IRQ mode, IRQ disabled
        STMFD   r0!,{r13,r14}        ; finish saving the context (r0_SYS,r1_SYS)

        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
    ; IRQ entry }}}

(1)     LDR     r0,=QK_intNest_      ; load address in already saved r0
(2)     LDRB    r12,[r0]             ; load original QK_intNest_ into the saved r12
(3)     ADD     r12,r12,#1           ; increment the nesting level
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
 (4)      STRB    r12,[r0]             ; store the value in QK_intNest_

          MSR     cpsr_c,#(SYS_MODE | NO_IRQ) ; enable FIQ

          ; NOTE: BSP_irq might re-enable IRQ interrupts (the FIQ is enabled
          ; already), if IRQs are prioritized by the interrupt controller.
          ;
          LDR     r12,=BSP_irq
          MOV     lr,pc                ; copy the return address to link register
          BX      r12                  ; call the C IRQ-handler (ARM/THUMB)

 (5)      MSR     cpsr_c,#(SYS_MODE | NO_INT) ; make sure IRQ/FIQ are disabled
 (6)      LDR     r0,=QK_intNest_      ; load address
 (7)      LDRB    r12,[r0]             ; load original QK_intNest_ into the saved r12
 (8)      SUBS    r12,r12,#1           ; decrement the nesting level
 (9)      STRB    r12,[r0]             ; store the value in QK_intNest_
(10)      BNE     QK_irq_exit          ; branch if interrupt nesting not zero

(11)      LDR     r12,=QK_schedPrio_
(12)      MOV     lr,pc                ; copy the return address to link register
(13)      BX      r12                  ; call QK_schedPrio_ (ARM/THUMB)
(14)      CMP     r0,#0                ; check the returned priority
(15)      BEQ     QK_irq_exit          ; branch if priority zero

(16)      LDR     r12,=QK_sched_
(17)      MOV     lr,pc                ; copy the return address to link register
(18)      BX      r12                  ; call QK_sched_ (ARM/THUMB)


      QK_irq_exit:
      ; IRQ exit {{{                   ; IRQ/FIQ disabled--return from scheduler
          MOV     r0,sp                ; make sp_SYS visible to IRQ mode
          ADD     sp,sp,#(8*4)         ; fake unstacking 8 registers from sp_SYS

          MSR     cpsr_c,#(IRQ_MODE | NO_INT) ; IRQ mode, both IRQ/FIQ disabled
          MOV     sp,r0                ; copy sp_SYS to sp_IRQ
          LDR     r0,[sp,#(7*4)]       ; load the saved SPSR from the stack
          MSR     spsr_cxsf,r0         ; copy it into spsr_IRQ

          LDMFD   sp,{r0-r3,r12,lr}^   ; unstack all saved USER/SYSTEM registers
          NOP                          ; can't access banked reg immediately
          LDR     lr,[sp,#(6*4)]       ; load return address from the SYS stack
          MOVS    pc,lr                ; return restoring CPSR from SPSR
      ; IRQ exit }}}
```

Listing 9 shows the `QK_irq()` "wrapper" function in assembly. Please note that the interrupt entry (delimited with the comments "`IRQ entry {{{`" and "`IRQ entry }}}`") and interrupt exit (delimited with the comments "`IRQ exit {{{`" and "`IRQ exit }}}`") are identical as in the `QF_irq()` "wrapper" function. Please refer to the notes after 3.2.3 for detailed explanation of these sections of the code. The following notes explain only the QK-specific additions made in the `QK_irq()` "wrapper" function.

(1-4) The QK interrupt nesting level `QK_intNest_` is incremented and saved.

(5) Interrupts are locked to access the QK interrupt nesting level `QK_intNest_` and to call the QK scheduler.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(6-9) The current QK interrupt nesting level `QK_intNest_` is decremented. Please note the use of the special version of `SUBS` in line (8), which sets the test flags if the nesting level `QK_intNest_` drops to zero.

(10) The branch is taken when the QK interrupt nesting level `QK_intNest_` is not zero. In this case the QK scheduler should not be called, because the IRQ is returning to a preempted IRQ, rather than the task level.

(11-13) The QK scheduler function `QK_schedPrio_()` is called via the `BX` instruction to determine the priority of the next task to run. The `QK_schedPrio_()` function can be compiled in ARM or THUMB mode, but perhaps using the ARM instruction set would deliver somewhat better performance.

(14)The priority, returned in `r0`, is tested against zero. Priority of zero means that no higher-priority task has been found, which would be above the priority of the preempted task.

(15)The branch is taken if the priority is zero (no scheduling is necessary).

(16-18) Otherwise scheduling is necessary, so the function `QK_sched_()` is called via the `BX` instruction. This function re-enables interrupts internally to launch a task, but it always returns with interrupts disabled. The `QK_sched_()` function can be compiled in ARM or THUMB mode, but perhaps using the ARM instruction set would deliver somewhat better performance.

4.2.2 The FIQ "Wrapper" Function for QK-nano

The QK port provides interrupt "wrapper" function `QK_fiq()` for handling the FIQ-type interrupts. The function is coded entirely in assembly and is located in the file `qkn_port.s`.

Listing 10 The QK_fiq assembly wrapper for the QK port defined in qk_port.s

```
        SECTION .textrw:CODE:NOROOT(2)
        PUBLIC  QK_fiq
        EXTERN  BSP_fiq
        EXTERN  QK_intNest_, QK_schedule_
        CODE32

    QK_fiq:
    ; FIQ entry {{{
        MOV     r13,r0                ; save r0 in r13_FIQ
        SUB     r0,lr,#4              ; put return address in r0_SYS
        MOV     lr,r1                 ; save r1 in r14_FIQ (lr)
        MRS     r1,spsr               ; put the SPSR in r1_SYS

        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
        STMFD   sp!,{r0,r1}           ; save SPSR and PC on SYS stack
        STMFD   sp!,{r2-r3,r12,lr}    ; save APCS-clobbered regs on SYS stack
        MOV     r0,sp                 ; make the sp_SYS visible to FIQ mode
        SUB     sp,sp,#(2*4)          ; make room for stacking (r0_SYS, r1_SYS)

        MSR     cpsr_c,#(FIQ_MODE | NO_INT) ; FIQ mode, IRQ/FIQ disabled
        STMFD   r0!,{r13,r14}         ; finish saving the context (r0_SYS, r1_SYS)

        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
    ; FIQ entry }}}

        LDR     r0,=QK_intNest_       ; load address in already saved r0
        LDRB    r12,[r0]              ; load original QK_intNest_ into the saved r12
        ADD     r12,r12,#1            ; increment interrupt nesting
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
        STRB    r12,[r0]            ; store the value in QK_intNest_


        ; NOTE:
        ; Because FIQ is typically NOT prioritized by the interrupt controller
        ; BSP_fiq must not enable IRQ/FIQ to avoid priority inversions!
        ;
        LDR     r12,=BSP_fiq
        MOV     lr,pc               ; copy the return address to link register
        BX      r12                 ; call the C FIQ-handler (ARM/THUMB)


        MSR     cpsr_c,#(SYS_MODE | NO_INT) ; make sure IRQ/FIQ are disabled
        LDR     r0,=QK_intNest_     ; load address
        LDRB    r12,[r0]            ; load original QK_intNest_ into the saved r12
        SUBS    r12,r12,#1          ; decrement the nesting level
        STRB    r12,[r0]            ; store the value in QK_intNest_
        BNE     QK_fiq_exit         ; branch if interrupt nesting not zero

(1)     LDR     r0,[sp,#(7*4)]      ; load the saved SPSR from the stack
(2)     AND     r0,r0,#0x1F         ; isolate the SPSR mode bits in r0
(3)     CMP     r0,#IRQ_MODE        ; see if we interrupted IRQ mode
(4)     BEQ     QK_fiq_exit         ; branch if interrutped IRQ


        ; We have interrupted a task. Call QK scheduler to handle preemptions
        LDR     r12,=QK_schedPrio_
        MOV     lr,pc               /* copy the return address to link register */
        BX      r12                 /* call QK_schedPrio_ (ARM/THUMB) */
        CMP     r0,#0               /* check the returned priority */
        BEQ     QK_fiq_exit         /* branch if priority zero */

        LDR     r12,=QK_sched_
        MOV     lr,pc               /* copy the return address to link register */
        BX      r12                 /* call QK_sched_ (ARM/THUMB) */

    QK_fiq_exit:
    ; FIQ exit {{{                  ; both IRQ/FIQ disabled (see NOTE above)
        MOV     r0,sp               ; make sp_SYS visible to FIQ mode
        ADD     sp,sp,#(8*4)        ; fake unstacking 8 registers from sp_SYS

        MSR     cpsr_c,#(FIQ_MODE | NO_INT) ; FIQ mode, IRQ/FIQ disabled
        MOV     sp,r0               ; copy sp_SYS to sp_FIQ
        LDR     r0,[sp,#(7*4)]      ; load the saved SPSR from the stack
        MSR     spsr_cxsf,r0        ; copy it into spsr_FIQ

        LDMFD   sp,{r0-r3,r12,lr}^  ; unstack all saved USER/SYSTEM registers
        NOP                         ; can't access banked reg immediately
        LDR     lr,[sp,#(6*4)]      ; load return address from the SYS stack
        MOVS    pc,lr               ; return restoring CPSR from SPSR
    ; FIQ exit }}}
```

Listing 10 shows the `QK_fiq()` "wrapper" function in assembly. Please note that the interrupt entry (delimited with the comments "`FIQ entry {{{`" and "`FIQ entry }}}`") and interrupt exit (delimited with the comments "`FIQ exit {{{`" and "`FIQ exit }}}`") are almost identical as in the `QF_fiq()` "wrapper" function. Please refer to the notes after 3.2.5 for detailed explanation of these sections of the code. The following notes explain only the QK-specific additions made in the `QK_fiq()` "wrapper" function.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(1) The interrupted status register SPSR is loaded from the stack.

(2) mode bits of the interrupted status register are isolated in r0.

(3) The mode bits of the interrupted status register are compared against the IRQ mode.

(4) Branch is taken if the interrupted mode was IRQ, which means that the FIQ preempted IRQ. In this case, the QK scheduler should **not** be called.

NOTE: FIQ can always preempt an IRQ in the ARM system architecture, because the F bit is not set in hardware in the IRQ startup sequence. Consequently, the FIQ can preempt the IRQ **before** the QK_intNest_ is incremented. The code in Listing 10(1-4) detects this situation and prevents calling the QK scheduler.

4.3 Idle Loop Customization in the QK Port

As described in Chapter 10 of [PSiCC2], the QK idle loop executes only when there are no events to process. The QK allows you to customize the idle loop processing by means of the callback QK_onIdle(), which is invoked by every pass through the QK idle loop. You can define the platform-specific callback function QK_onIdle() to save CPU power, or perform any other “idle” processing (such as Quantum Spy software trace output).

NOTE: The idle callback QK_onIdle() is invoked with interrupts unlocked (which is in contrast to QF_onIdle() that is invoked with interrupts locked, see Section).

The following Listing 11 shows an example implementation of QK_onIdle() for the AT91SAM7 MCU. Other ARM-based embedded microcontrollers (e.g., LPC chips from NXP) handle the power-saving mode very similarly.

Listing 11 QK_onIdle() callback for the AT91SAM7 MCU.

```
__ramfunc void QK_onIdle(void) {
#ifdef NDEBUG     /* only if not debugging (power saving hinders debugging) */
    AT91C_BASE_PMC->PMC_SCDR = 1;/* Power-Management: disable the CPU clock */
    /* NOTE: an interrupt starts the CPU clock again */
#endif
}
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 Related Documents and References

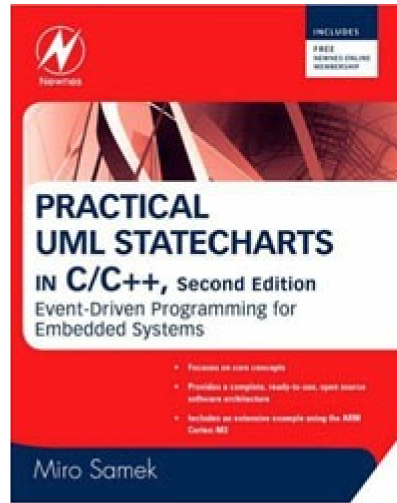
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.	http://www.embedded.com/showArticle.jhtml?articleID=190302110
[Seal 00] "ARM Architecture Reference Manual", Seal, David, Addison Wesley 2000.	Available from most online book retailers, such as amazon.com. ISBN 0-201-73719-1.
[Atmel 98] Application Note "Disabling Interrupts at Processor Level", Atmel 1998	http://www.atmel.com/dyn/resources/-prod_documents/DOC1156.PDF
[ARM 05] ARM Technical Support Note "Writing Interrupt Handlers", ARM Ltd. 2005	www.arm.com/support/faqdev/1456.html
[Philips 05] Application Note AN10391 "Nesting of Interrupts in the LPC2000", Philips 2005	www.semiconductors.philips.com/acrobat_download/-applicationnotes/AN10381_1.pdf
[Seal 00] "ARM Architecture Manual, 2nd Edition", David Seal Editor, Addison Wesley 2000	Available from most online book retailers, such as amazon.com.
[ARM 06] "ARM v7-M Architecture Application Level Reference Manual", ARM Limited	www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html .
[IAR 09a] "ARM® IAR C/C++ Compiler Reference Guide v5.30", IAR 2009	Available in PDF as part of the ARM KickStart™ kit in the file EWARM_CompilerReference.pdf.
[IAR 09b] "IAR Linker and Library Tools Reference Guide v5.30", IAR 2009	Available in PDF as part of the ARM KickStart™ kit
[IAR 09c] "ARM® Embedded Workbench® IDE User Guide v5.30", IAR 2009	Available in PDF as part of the ARM KickStart™ kit in the file EWARM_UserGuide.pdf.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

6 Contact Information

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

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



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

Copyright © Quantum Leaps, LLC. All Rights Reserved.