



Quantum™ Leaps
innovating embedded systems



Application Note

QP™ and ARM7/9-IAR

Document Revision N
November 2011



Copyright © Quantum Leaps, LLC

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

Table of Contents

1 Introduction	1
1.1 About QP™.....	1
1.2 About the ARM Port.....	2
1.3 Licensing QP.....	3
2 Directories and Files	4
2.1 Building the QP Libraries.....	5
2.2 Building the Examples.....	7
2.3 Running the Examples.....	8
3 Board Support Package	10
3.1 Startup Code in Assembly.....	10
3.2 Linker Command File.....	14
3.3 Controlling Placement of the Code in Memory and ARM/THUMB Compilation.....	15
3.4 BSP Functions in C.....	15
4 The Vanilla Port	18
4.1 Compiler Options Used.....	18
4.2 The QF Port Header File.....	18
4.3 Handling Interrupts.....	22
4.4 Idle Loop Customization in the “Vanilla” Port.....	33
5 The QK Port	34
5.1 Compiler and Linker Options Used.....	34
5.2 The QK Port Header File.....	34
5.3 Handling Interrupts.....	34
5.4 Idle Loop Customization in the QK Port.....	39
6 Controlling Placement of the Code in Memory and ARM/THUMB Compilation	40
7 References	41
8 Contact Information	42



1 Introduction

This Application Note describes how to use the QP/C™ and QP™/C++ state version **4.3.00** or higher with the ARM7 or ARM9 processors with the IAR EWARM toolchain. This document describes the following two main implementation options:

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

To focus the discussion, this Application Note uses the IAR Embedded Workbench® for ARM (EWARM version **6.30** KickStart™ edition, which is available as a **free** download from the [IAR website www.iar.com](http://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 Development Kits™ (QDKs) for specific ARM7/ARM9 boards. The QDKs 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.

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

1.1 About QP™

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

As shown in Figure 1, 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.

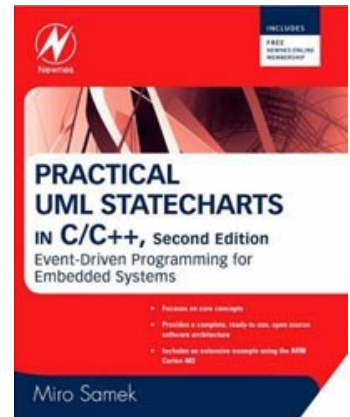
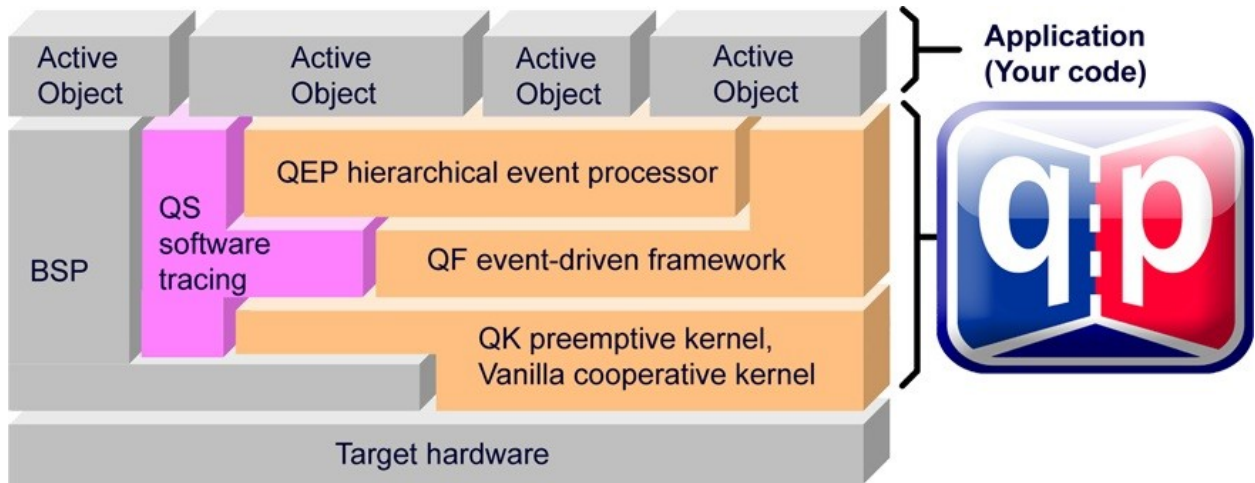


Figure 1 QP components and their relationship with the target hardware, board support package (BSP), and the application



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

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

1.2 About the ARM Port

The ARM core is a quite complicated processor in that it supports 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:

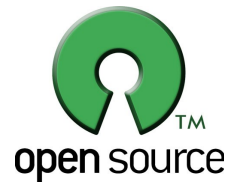
1. The ARM processor executes in both ARM and THUMB states, meaning that some parts of the code are compiled to ARM and others to THUMB instruction sets, and calls between ARM and THUMB functions are allowed. Such approach is supported by the “interwork” option of the ARM compilers and linkers. This 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 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

The **Generally Available (GA)** distribution of QP™ 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

2 Directories and Files

The code for the QP port to ARM is available as part of any QP Development Kit (QDK) for ARM. The QDKs assume that the generic platform-independent QP™ 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:

```

<qp>/          - QP/C or QP/C++ Root Directory
|
+-include/     - QP public include files
| +-qassert.h  - Quantum Assertions platform-independent public include
| +-qevent.h   - QEvent declaration
| +-qep.h      - QEP platform-independent public include
| +-qf.h       - QF platform-independent public include
| +-qk.h       - QK platform-independent public include
| +-qs.h       - QS platform-independent public include
| +-qs_dummy.h - QS "dummy" public include
| +-qevent.h   - native QF event queue include
| +-qmpool.h   - native QF memory pool include
| +-qpset.h    - native QF priority set include
| +-qvanilla.h - native QF non-preemptive "vanilla" kernel include
|
+-ports/      - QP ports
| +-arm/      - ARM port
| | +-vanilla/ - "vanilla" ports
| | | +-iar/   - IAR ARM compiler
| | | | +-dbg/  - Debug build
| | | | | +-qep_ARM7TDMI.a - QEP library for ARM7TDMI core
| | | | | +-qep_ARM966E-S.a - QEP library for ARM966E-S core
| | | | | +-qf_ARM7TDMI.a  - QF library for ARM7TDMI core
| | | | | +-qf_ARM966E-S.a - QF library for ARM966E-S core
| | | | +-rel/          - Release build
| | | | | +-...         - Release libraries
| | | | +-spy/         - Spy build
| | | | | +-...         - Spy libraries
| | | | +-src/         - Platform-specific source directory
| | | | | +-qf_port.s   - Platform-specific source code for the port
| | | | | +-make_ARM7TDMI.bat - Batch script to build QP libraries for ARM7TDMI core
| | | | | +-make_ARM966E-S.bat - Batch script to build QP libraries for ARM966E-S core
| | | | | +-...         - Batch scripts to build QP libraries for other ARM cores
| | | | +-qep_port.h   - QEP platform-dependent public include
| | | | +-qf_port.h    - QF platform-dependent public include
| | | | +-qs_port.h    - QS platform-dependent public include
| | | | +-qp_port.h    - QP platform-dependent public include
| | +-qk/             - QK (Quantum Kernel) ports
| | | +-iar/          - IAR ARM compiler
| | | | +-dbg/        - Debug build
| | | | | +-qep_ARM7TDMI.a - QEP library for ARM7TDMI core
| | | | | +-qep_ARM966E-S.a - QEP library for ARM966E-S core
| | | | | +-qf_ARM7TDMI.a  - QF library for ARM7TDMI core
| | | | | +-qf_ARM966E-S.a - QF library for ARM966E-S core
| | | | | +-qk_ARM7TDMI.a  - QK library for ARM7TDMI core
| | | | | +-qk_ARM966E-S.a - QK library for ARM966E-S core
| | | | +-rel/        - Release build
| | | | | +-...         - Release libraries
| | | | +-spy/        - Spy build

```



```

| | | | +-...           - Spy libraries
| | | | +-src/         - Platform-specific source directory
| | | | +-qk_port.s    - Platform-specific source code for the port
| | | | +-make_ARM7TDMI.bat - Batch script to build QP libraries for ARM7TDMI core
| | | | +-make_ARM966E-S.bat - Batch script to build QP libraries for ARM966E-S core
| | | | +-. . .       - Batch scripts to build QP libraries for other ARM cores
| | | | +-qep_port.h   - QEP platform-dependent public include
| | | | +-qf_port.h   - QF platform-dependent public include
| | | | +-qs_port.h   - QS platform-dependent public include
| | | | +-qp_port.h   - QP platform-dependent public include
|
+-examples/          - QP examples
| +-arm/             - ARM CPU
| | +-vanilla/       - "Vanilla" port
| | | +-iar/         - IAR ARM compiler
| | | | +-dpp-at91sam7s-ek - DPP example for the AT91SAM7S-EK evaluation board
| | | | | +-dbg/      - Debug build (runs from RAM)
| | | | | | +-dpp.out  - executable image
| | | | | +-rel/      - Release build (runs from Flash)
| | | | | | +-dpp.out  - executable image
| | | | | +-spy/      - Spy build (runs from RAM)
| | | | | | +-dpp.out  - executable image (instrumented with QSpy)
| | | | | +-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   -
| | | | | +-dpp.ewp   - IAR project for the DPP application example
| | | | | +-dpp.eww   - IAR workspace for the DPP application example
| | | |
| | +-qk/            - QK (Quantum Kernel) port
| | | +-iar/         - IAR ARM compiler
| | | | +-dpp-qk-at91sam7s-ek - DPP example for the AT91SAM7S-EK board with QK
| | | | | +-. . .     - the same files as in vanilla/iar/dpp-at91sam7s
| | | |
| . . .

```

Listing 1 Selected Directories and files of the QP after installing the ARM-IAR port. Directory and File names in bold indicate the elements included in the ARM port.

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\` directory (see Listing 1). This section describes steps you need to take to rebuild the libraries yourself.

NOTE: To achieve commonality among different development tools, Quantum Leaps software does not use the vendor-specific IDEs, such as the IAR Embedded Workbench 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\...` directory. For example, to build the debug version of all the QP libraries for the ARM7TDMI core, with the IAR ARM compiler, QK kernel, you open a console window on a Windows PC,

change directory to `<qp>\ports\arm\qk\iar\`, and invoke the batch by typing at the command prompt the following command:

```
make_ARM7TDMI
```

The build process should produce the QP libraries in the location: `<qp>\ports\arm\qk\iar\dbg\`. The `make_<core>.bat` files assume that the ARM toolset has been installed in the directory `C:\tools\IAR\ARM_KS_5.30`.

NOTE: You need to adjust the symbol `IAR_ARM` at the top of the batch scripts if you've installed the IAR ARM 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_<core>.bat` utility with the “spy” target, like this:

```
make_ARM7TDMI spy
```

The make process should produce the QP libraries in the directory: `<qp>\ports\arm\vanilla\iarspy\`.

You choose the build configuration by providing a target to the `make_<core>.bat` utility. The default target is “dbg”. Other targets are “rel”, and “spy” respectively. The following table summarizes the targets accepted by `make_<core>.bat`.

Software Version	Build command
Debug (default)	<code>make_<core></code>
Release	<code>make_<core> rel</code>
Spy	<code>make_<core> spy</code>

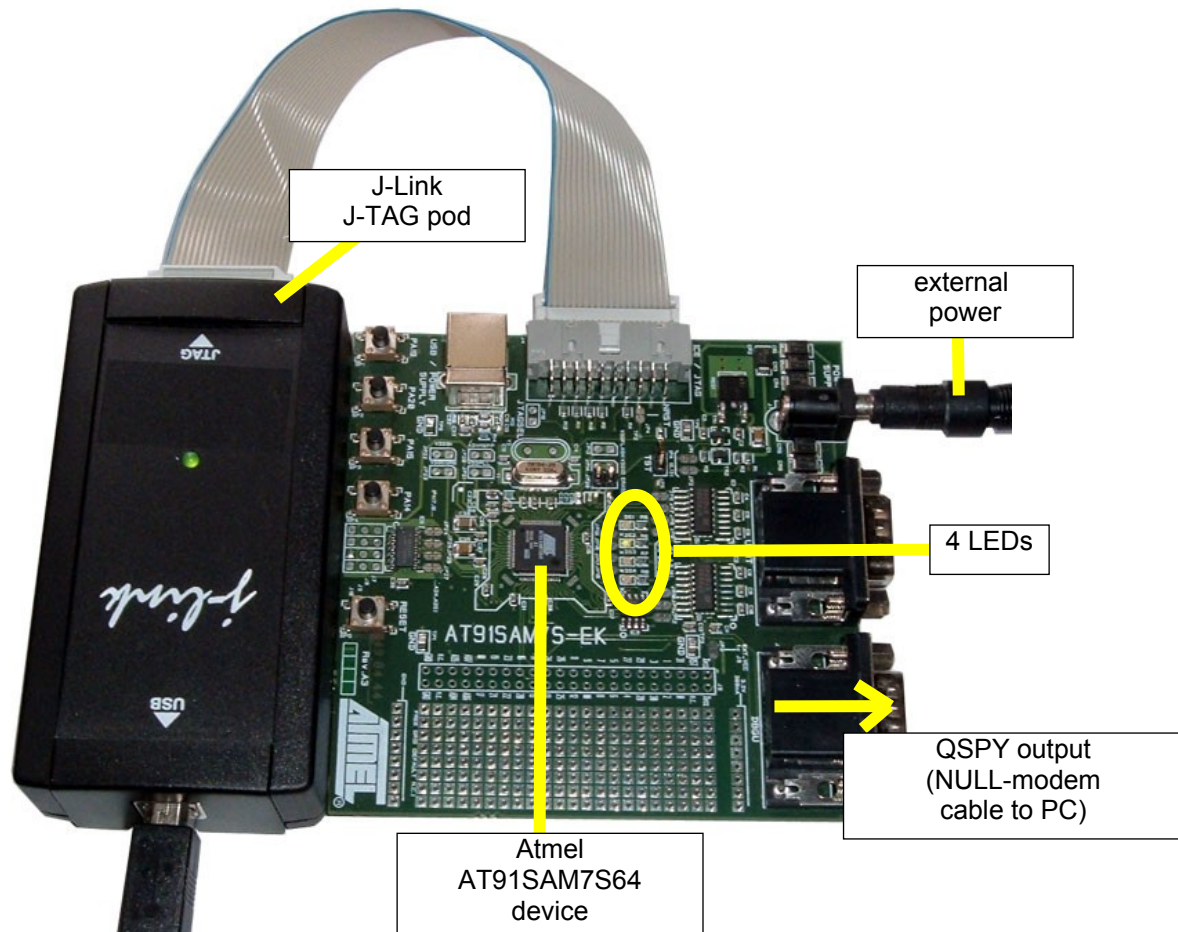
Table 1 Make targets for the Debug, Release, and Spy software configurations

2.2 Building the Examples

This Application Note uses the AT91SAM7S-EK development board from Atmel as an example hardware platform. The actual hardware/software used is as follows (see also Figure 2):

1. AT91SAM7S-EK evaluation board from Atmel (the board is based around the AT91SAM7S64 MCU) NOTE: The board, the J-Link pod, and software are bundled into the IAR AT91SAM7 Kick-Start™ Kit available from IAR Systems (www.iar.com).
2. The J-Link J-Tag pod from SEGGER (www.segger.com). NOTE: IAR Embedded Workbench supports also several other J-Tag pods.
3. IAR Embedded Workbench® for ARM version **6.30**, the free KickStart edition (www.iar.com)
4. QP/C/C++ version **4.3.00** or later

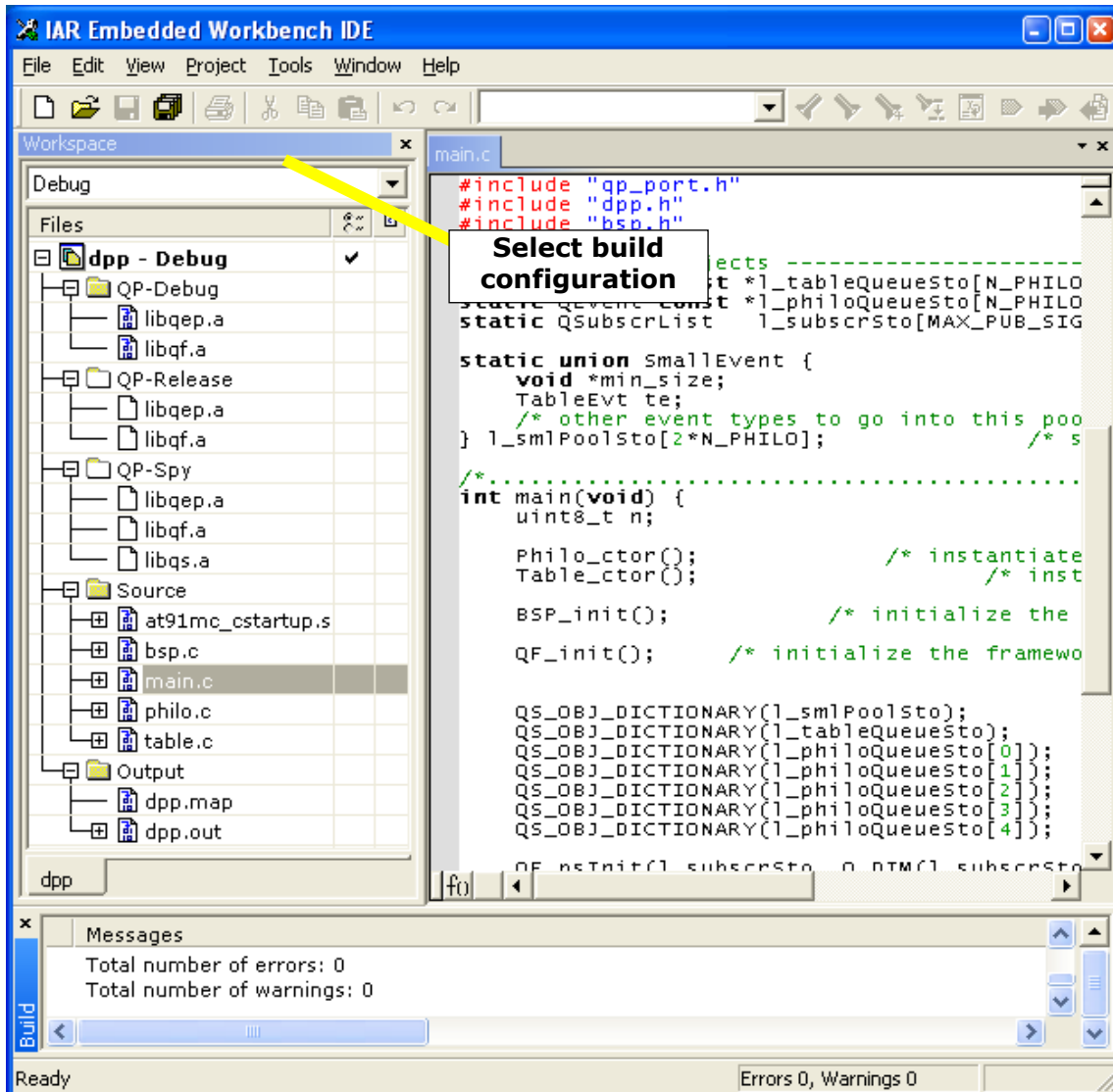
Figure 2 AT91SAM7S-EK evaluation board from Atmel and the J-Link pod



The examples included in this Application Note are based on the standard Dining Philosopher Problem implemented with active objects (see Chapter 9 in [PSiCC2]).

The example directory contains the IAR Embedded Workbench workspace file `dpp.eww` that you can load into the IAR EW IDE. The workspace contains three build configurations (Debug, Release, and Spy) that you can select with the drop-down list, as shown in Figure 3.

Figure 3 Building the DPP application in the IAR EWARM IDE



2.3 Running the Examples

You need to use the IAR EWARM IDE to download the code to the target and debug it with the IAR C-Spy debugger integrated into the IAR EWARM IDE. Before starting the debug session, you should make sure that your J-tag pod (such as J-Link) is installed and connected to the target. The AT91SAM7S-EK board should be powered up and connected to the J-Link with the 20-pin ribbon cable (see Figure 2). The status LED on the J-Link pod should not flash.

To download the code click on Project | Debug menu (or the toolbar shortcut). This should load the code and break at `main()`. To continue running, click Debug/Go, or F5, or select the toolbar shortcut.

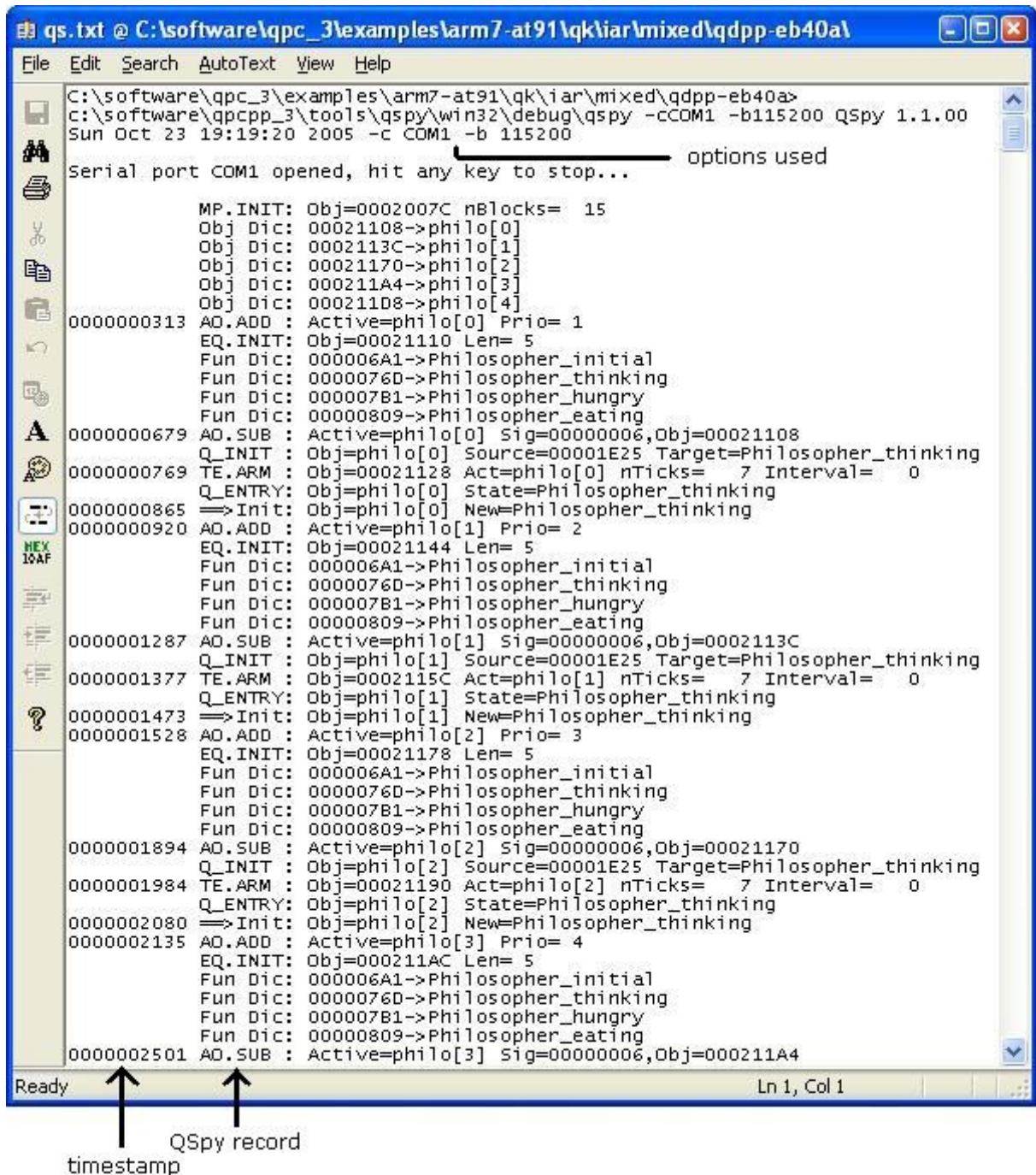
As the application is running, the status LEDs at the top of the board (see Figure 2) should blink indicating the changing status of the Dining Philosophers. If you connect the serial NULL-cable to the DBGU DB9 connector and your PC, you could launch the QSPY host utility to observe the output in the human-readable format. You launch the QSPY utility on a Windows PC as follows. (1) Change the directory to the QSPY host utility `<qp>\tools\qspy\win32\vc2005\Release` and execute:

```
qspy -c COM1 -b 115200
```

This will start the `qspy` utility to listen on COM1 serial port with baud rate 115200. (Please use `-c COM2`, or higher if you connected the NULL-cable to a different COM port on your PC.)

The following screen shot shows the QSPY output from the DPP run:

Figure 4 Screen shot from the QSPY output



3 Board Support Package

The Board Support Package (BSP) consists of the startup code, the linker script, the board initialization, and interrupt service routines (ISRs). The BSP is co-located with the application (the DPP example) and consists of the following files:

- `at91mc_cstartup.s` — startup code in assembly
- `at91SAM7S64.icf` — linker command file for executing the code out of Flash
- `bsp.c` — BSP functions and ISRs in C
- `bsp.h` — BSP header file

The BSP does not include drivers for all AT91SAM7 peripherals. However, it should be fairly complete in that it supports debugging in RAM, code downloading to Flash, software tracing with Quantum Spy (QS), fine-tuning of the release version, and more.

3.1 Startup Code in Assembly

The customized startup sequence for the AT91SAM7 is implemented in the assembly file `at91mc_cstartup.s` co-located with the DPP example application. This file is based on the IAR startup assembly module `cstartup.s`. You need to customize this code for the AT91SAM7 to provide the Memory Controller (MC) re-mapping step, among other things. The “System Startup and Termination” Section in the “ARM® IAR C/C++ Compiler Reference Guide” describes how to customize the `cstartup.s` module.

The customized `at91mc_cstartup.s` module is designed to be generic for all AT91SAM7 devices with the MC, not just to the AT91SAM7S64 and the AT91SAM7S-EK board. Typically, you should not need to modify this file to work with other members of the AT91SAM7 family. All CPU and board specifics should be handled in C, typically in the file `bsp.c`. The following sections describe the structure of the `at91mc_cstartup.s` module.

3.1.1 Exception Vectors

Listing 2 Reset and exception vectors defined in the `at91mc_cstartup.s`.

```

;-----
; Reset and other vectors
;
(1) SECTION .intvec:CODE:NOROOT(2)
    PUBLIC __vector
    PUBLIC __iar_program_start

; Execution starts here.
; After a reset, the mode is ARM, Supervisor, interrupts disabled.

(2) CODE32 ; Always ARM mode after reset
(3) __vector:
(4) LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector
    LDR pc, [pc, #24] ; load the secondary vector

```

```

    LDR    pc, [pc, #24]    ; load the secondary vector

    ; The secondary jump table starts below. It contains the
    ; addresses for the PC-relative loads from the primary vector table.
    ; The secondary vectors are to be overwritten by the application to hook
    ; up exception handlers at runtime. The default implementation provided
    ; below causes endless loops around the selected addresses.
    ; For example, a prefetch abort exception forces the PC to 0x0C and
    ; the address loaded from the secondary vector table will be again 0x0C.
    ;
(5)  DC32    __iar_program_start ; Reset
      DC32    0x04                ; Undefined Instruction
      DC32    0x08                ; Software Interrupt
      DC32    0x0C                ; Prefetch Abort
      DC32    0x10                ; Data Abort
      DC32    0x14                ; Reserved
      DC32    0x18                ; IRQ
      DC32    0x1C                ; FIQ
  
```

- (1) This module defines interrupt vectors and is placed in the `.intvec` section.
- (2) The ARM core always starts in ARM state, Supervisor mode, all interrupts disabled.
- (3) The symbol `__vector` denotes the beginning the primary vector table.
- (4) The primary vector table is pre-filled with jumps to the secondary jump table located immediately after the primary vector table at address 0x20. The secondary vector table is used for maximum flexibility. Such jump table makes it much easier to hook any exception vector at real time, by simply writing a pointer to one of the locations 0x20-0x3C, rather than having to synthesize a branch instruction at the primary vector table. Also, the `LDR` instruction can access any code within the full 32-bit address range, while the PC-relative branch instruction is limited to 0x02000000 from the current PC.

NOTE: Please note that the ARM core starts execution at address 0x0. Out of reset, the Flash memory is mapped at this address. However, immediately after executing the `LDR pc, [pc, #24]` instruction, the code starts executing from the address range above 0x00100000, in which always Flash is mapped, even after the MC remapping.

- (5) The secondary jump table is initialized in such a way that any exception (except reset) causes an endless loop to itself. For example, a Prefetch Abort exception located at address 0x0C causes a jump also to 0x0C, so the CPU hangs in an endless loop around address 0x0C. Of course this is only the initial setting, after which you application code should initialize the secondary vector table to handle exceptions any way you want.

NOTE: The simple exception vectors are only for initial setting, after which you application code should initialize the secondary vector table to handle exceptions any way you want. In particular, you will need to adjust the interrupt exceptions (IRQ and FIQ).

3.1.2 Reset and MC Remapping

Normally, when the AT91SAM7 boots from Flash, the boot sequence performs the MC remap to replace slow Flash ROM with fast RAM at the address 0x0. However, in order to guarantee that the ARM core is provided with valid vectors at all times, including the boot sequence itself, the vector table must be initialized in RAM **before** the RAM is remapped down to address 0x0.



Listing 3 Reset and Memory Controller remapping

```
SECTION FIQ_STACK:DATA:NOROOT(3)
SECTION IRQ_STACK:DATA:NOROOT(3)
SECTION SVC_STACK:DATA:NOROOT(3)
SECTION ABT_STACK:DATA:NOROOT(3)
SECTION UND_STACK:DATA:NOROOT(3)
SECTION CSTACK:DATA:NOROOT(3)

(1) SECTION .text:CODE:NOROOT(2)
    REQUIRE __vector
    EXTERN ?main
    PUBLIC __iar_program_start

    ; Embed a copyright notice prominently at the beginning of
    ; the Flash image.
    ;

(2) DC8      'Copyright (c) 2008, Quantum Leaps, LLC. All Rights Reserved.'
    ALIGNROM 2          ; re-align after the string at 2^2 boundary

    ARM                ; reset always executes in ARM

(3) __iar_program_start:
    ; Copy the exception vectors from ROM to RAM
    ;
    ; NOTE: the exception vectors must be in RAM *before* the remap
    ; in order to guarantee that the ARM core is provided with valid vectors
    ; during the remap operation.
    ;

(4) LDR     r8,=__vector          ; Source
    LDR     r9,=AT91SAM7S_RAM    ; Destination

(5) MOV     r0,#0
(6) STR     r0,[r9,#0]          ; write zero at the start of RAM
(7) LDR     r0,[r0,#0]          ; read the location zero
(8) TEQ     r0,#0                ; is the result equal to zero?

(9) LDMIA  r8!,{r0-r7}          ; Load Vectors
(10) STMIA  r9!,{r0-r7}          ; Store Vectors
(11) LDMIA  r8!,{r0-r7}          ; Load secondary vector table
(12) STMIA  r9!,{r0-r7}          ; Store secondary vector table

(13) LDR     r0,=AT91SAM7S_MC
(14) MOV     r1,#1
(15) STRNE  r1,[r0,#0]          ; Remap Memory, if not remapped already

    ; Initialize the stack pointers...

(16) MSR     cpsr_c,#(IRQ_MODE | I_BIT | F_BIT) ; Change to IRQ mode
    LDR     sp,=SFE(IRQ_STACK) ; End of IRQ_STACK

(17) MSR     cpsr_c,#(FIQ_MODE | I_BIT | F_BIT) ; Change to FIQ mode
    LDR     sp,=SFE(FIQ_STACK) ; End of FIQ_STACK

(18) MSR     cpsr_c,#(SVC_MODE | I_BIT | F_BIT) ; Change to SVC mode
    LDR     sp,=SFE(SVC_STACK) ; End of SVC_STACK

(19) MSR     cpsr_c,#(ABT_MODE | I_BIT | F_BIT) ; Change to ABT mode
    LDR     sp,=SFE(ABT_STACK) ; End of ABT_STACK
```

```

(20)   MSR     cpsr_c, #(UND_MODE | I_BIT | F_BIT) ; Change to UND mode
        LDR     sp, =SFE(UND_STACK)                ; End of UND_STACK

(21)   MSR     cpsr_c, #(SYS_MODE | I_BIT | F_BIT) ; Change to SYSTEM mode
        LDR     sp, =SFE(CSTACK)                   ; End of CSTACK

        ; Continue to ?main for more IAR specific system startup
        ;

(22)   LDR     r12,=?main
        BX     r12

        ; The code should never return from ?main, but just in case, put
        ; an endless loop here to hang the CPU.
        ; Alternatively, you might try a reset...
        ;

(23)   B      .
  
```

- (1) The code is assembled to the regular `.text` segment
- (2) It's always a good idea to embed an ASCII copyright notice directly into the binary image. Such a prominent ASCII text early in the binary image informs anybody attempting to reverse-engineer the code about the origin and rights to the code. You should replace this copyright notice with your company's name.
- (3) The symbol `__iar_program_start` is the standard entry point for the IAR startup code.
- (4) The source and destination addresses for copying vectors from ROM to RAM are prepared in registers
- (5) Register `r0` is loaded with zero.
- (6) Zero is written at the beginning of the permanent RAM location.

NOTE: Unlike the earlier AT91X40 MCUs, the AT91SAM family keeps the RAM permanently mapped to the address range starting at 0x00200000, and the MC remapping operation only creates the RAM alias at 0x0.

- (7) The memory at address zero is loaded to the register.
- (8) The loaded value is tested against zero, which is the value written before at this location. If the loaded value is indeed zero, this means that RAM is already remapped and the remap should **not** be done.

NOTE: In the AT91SAM the Memory Controller remap is a toggle, so doing a remap twice would revert the remap, which is not what you want. Also, please note that the value zero cannot be located at zero address of the ROM before the remap. This location is the ARM reset vector and must contain a valid ARM instruction. Finally, please note that the testing for remap is very handy when resetting the MCU from a debugger, which resets only the ARM core, but does not reset the Memory Controller (including the remap operation).

- (9-10) The primary vector table copied to the RAM. This vector table is identical to the ROM table in Listing 2. Please note that the `ADDR` pseudo-instruction is PC-relative, so the code is poison-independent.
- (11-12) The secondary jump table copied to the RAM. This jump table is also identical to the ROM table in Listing 2.
- (13-15) The actual Memory Controller remap is performed only if the ZERO flag hasn't been set earlier in the `TEQ` instruction (i.e., the memory remap has not occurred yet).
- (16-21) All stacks are initialized to the end of the stack sections, because the stack grows towards lower memory addresses (descending stack)

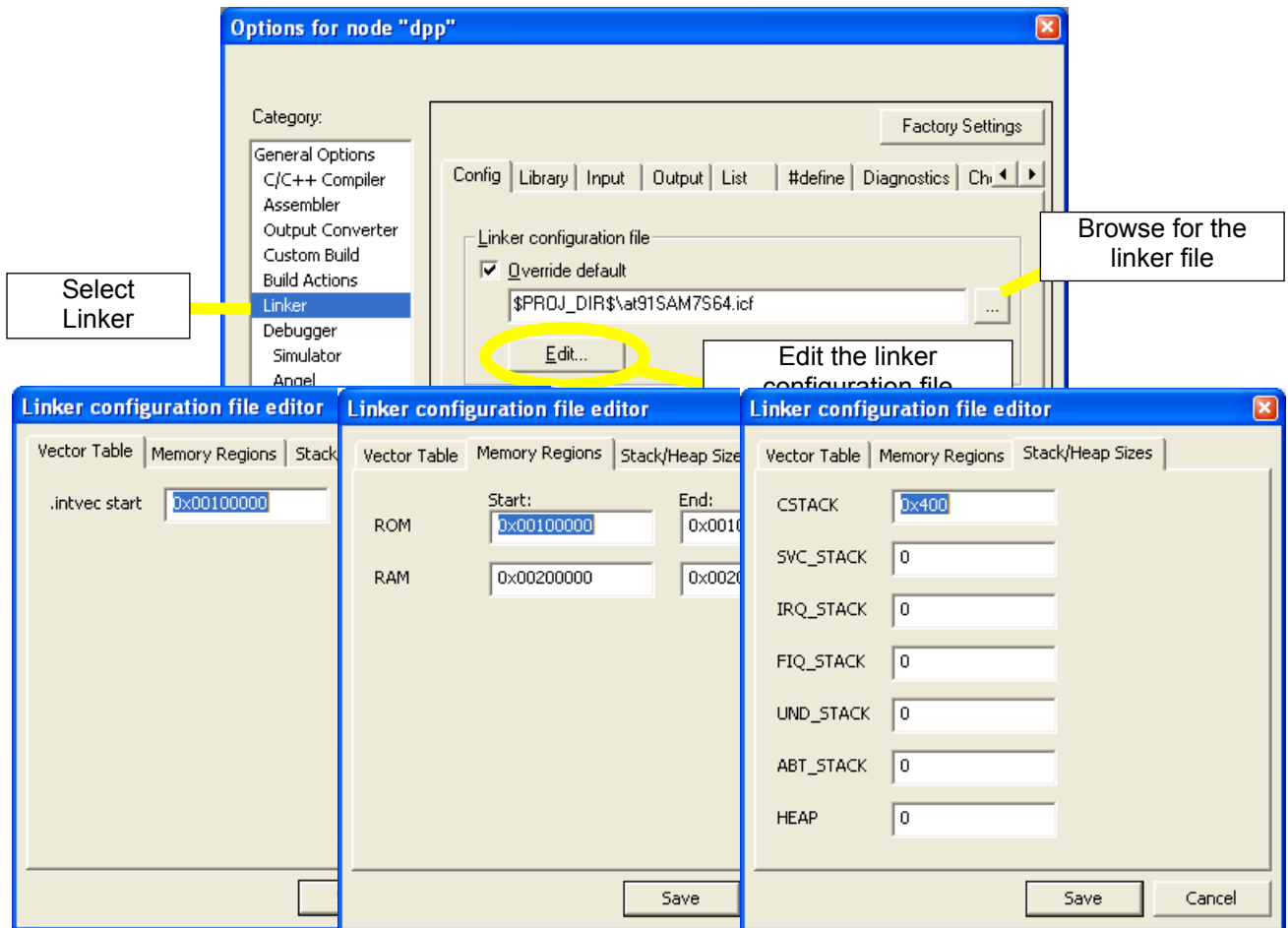
NOTE: This startup code is generic, so it allows configuring all the available stacks in the ARM core. However, the QP ports (both the vanilla and the QK versions) use only the SYSTEM/USER stack.

- (22) The code branches to the low-level initialization `?main` provided by the IAR standard library. The IAR code initializes all the RAM sections from the ROM sections and finally invokes your `main()` function in C. The “System Startup and Termination” Section in the “ARM@ IAR C/C++ Compiler Reference Guide” describes in more detail what happens from that point on and how you can customize further the initialization of the various sections.
- (23) The control should never return back to the startup code, but just in case this branch will hang the CPU in a tight loop.

3.2 Linker Command File

A linker command file controls where the linker will locate the code and data in memory. The BSP for the AT91SAM7S-EK board comes with the file `at91SAM7S64.icf` to place the complete application image in the Flash ROM. This file is not intended for manual editing, but rather only via the IAR Embedded Workbench IDE, as shown in Figure 5.

Figure 5 Editing the linker control script in the IAR Embedded Workbench IDE.



Typically, you don't need to adjust the memory addresses for the ARM device (in the Memory Regions tab in Figure 5), because they come pre-configured when you select the specific device in the General Options section. However, you should adjust the Stack and Heap Sizes on the third tab of the "Linker configuration file editor" shown in Figure 5. Again, note that QP uses only the USER/SYSTEM stack and does **not** use any other stacks. If you don't use the heap, which is highly recommended in any embedded application, you should also set the size of the heap to zero.

3.3 Controlling Placement of the Code in Memory and ARM/THUMB Compilation

A very important and often overlooked aspect for optimal system performance is controlling both the placement of the code in memory and the instruction set chosen to compile individual modules. The placement of the code in memory is most important. For example, on a typical AT91 family MCU, the code executing from the fast 32-bit wide on-chip SRAM can be three to four times faster than the same code executing from the 16-bit wide Flash. This is 300 to 400% difference!

The instruction set can contribute to additional difference of 20 to 40% in the execution speed, ARM being faster than THUMB when executed from 32-bit wide memory, and slower, when executed from 16-bit wide memory.

Therefore, this QDK puts a lot attention on giving you fine-granularity of control over the placement of the code in memory and instruction set compilation. Clearly, it's advantageous to expend some of the fast on-chip RAM to dramatically improve the performance.

In the IAR toolset, you can instruct the compiler to place the code in RAM by assigning it to the `.textrw` section (see "ARM® IAR C/C++ Compiler Reference Guide"). You can achieve this by the compiler option `--section .text=.textrw`. Similarly, you can instruct the compiler to produce either THUMB or ARM code by the compiler option `--cpu_mode thumb` or `--cpu_mode arm`, respectively.

To enable fine-tuning of the code, the Makefile for each QP component allows you to specify the code segment placement and instruction set for each individual fine-granularity module in the library. For example, the following fragment of the Makefile for building the QF library shows the choices made for several QF components:

```
. . . .
%CC% --cpu_mode thumb %CCFLAGS% %CCINC% -o%BINDIR% %SRCDIR%\qa_sub.c
%CC% --cpu_mode arm --section .text=.textrw %CCFLAGS% -o%BINDIR% %SRCDIR%\qvanilla.c
. . . .
```

The module `qa_sub.c`, for instance, which contains subscribe/unsubscribe functions for active objects, is compiled to THUMB and placed in the default location, which is ROM (Flash). This is because the functionality must be merely correct, but not necessarily fast. On the other hand, the module `qvanilla.c` is clearly a "hot-spot", because it contains the cooperative "vanilla" kernel running the whole application (see Chapter 7 in [PSiCC2]) and therefore it is compiled to ARM and placed in the fast RAM. Of course, you can fine-tune the code placement any way you like for your particular project.

You could also apply similar strategy to your application code. Additionally, you could use the IAR extended keyword `__ramfunc` to place selected functions in RAM. This option is used in the ISR code discussed in later.

3.4 BSP Functions in C

The file `bsp.c` located in the directory `<qp>\examples\arm\vanilla\iar\dpp-at91sam7s-ek` contains the ISRs, the initialization, and the QS port to the AT91SAM7S-EK board. The file `bsp.c` is mostly about accessing the hardware. It indirectly includes (through the `bsp.h` header file) the IAR header file `ioat91sam7s64.h`, which contains symbolic names for all registers defined inside the AT91SAM7S64 MCU.

3.4.1 The Low-Level Initialization

The standard IAR initialization sequence that starts from `?main` (see [2]), calls the function `__low_level_init()`. The function gives the application a chance to perform early initializations of the hardware even before the segments are copied to RAM. This function cannot use any static variables, because these have not yet been initialized in RAM. (See also “System Startup and Termination” Section in the “ARM® IAR C/C++ Compiler Reference Guide”.)

In the AT91SAM7 BSP, the `__low_level_init()` function is used to setup the PLL to speed up the rest of the startup sequence.

Listing 4 Low-level initialization in the BSP

```
(1) int __low_level_init(void) {
    AT91PS_PMC pPMC = AT91C_BASE_PMC;

    /* Set flash wait state FWS and FMCN */
(2)   AT91C_BASE_MC->MC_FMR = ((AT91C_MC_FMCN) & ((MCK + 500000)/1000000 << 16))
        | AT91C_MC_FWS_1FWS;
(3)   AT91C_BASE_WDTC->WDTC_WDMR = AT91C_WDTC_WDDIS; /* Disable the watchdog */

    /* Enable the Main Oscillator: set OSCOUNT to 6, which gives
    * Start up time = 8 * 6 / SCK = 1.4ms (SCK = 32768Hz)
    */
(3)   pPMC->PMC_MOR = ((6 << 8) & AT91C_CKGR_OSCOUNT) | AT91C_CKGR_MOSCEN;
    while ((pPMC->PMC_SR & AT91C_PMC_MOSCS) == 0) { /* Wait the startup time */
    }

    /* Set the PLL and Divider:
    * - div by 5 Fin = 3,6864 = (18,432 / 5)
    * - Mul 25+1: Fout = 95,8464 = (3,6864 * 26)
    * PLLCOUNT pll startup time estimate at : 0.844 ms
    * PLLCOUNT 28 = 0.000844 / (1/32768)
    */
    pPMC->PMC_PLLR = ((AT91C_CKGR_DIV & 0x05)
        | (AT91C_CKGR_PLLCOUNT & (28 << 8))
        | (AT91C_CKGR_MUL & (25 << 16)));
    while ((pPMC->PMC_SR & AT91C_PMC_LOCK) == 0) { /* Wait the startup time */
    }
    while ((pPMC->PMC_SR & AT91C_PMC_MCKRDY) == 0) {
    }

    /* Select Master Clock and CPU Clock select the PLL clock / 2 */
    pPMC->PMC_MCKR = AT91C_PMC_PRES_CLK_2;
    while ((pPMC->PMC_SR & AT91C_PMC_MCKRDY) == 0) {
    }

    pPMC->PMC_MCKR |= AT91C_PMC_CSS_PLL_CLK;
    while ((pPMC->PMC_SR & AT91C_PMC_MCKRDY) == 0) {
    }

(4)   return 1; /* proceed with the segment initialization */
}
```

(1) `__low_level_init()` is called from the IAR startup code before copying the DATA section and before zeroing the BSS section. (And also before calling the static constructors in C++).



- (2) The number of Flash wait states FWS is set to 1 and the number FMCN (number of MCK cycles in a microsecond) is set to the programmed MCK.
- (3) The PLL is programmed
- (4) The value returned by `__low_level_init()` determines whether or not data segments should be initialized by `__segment_init()`. If `__low_level_init()` returns 0, the data segments will NOT be initialized. For more information see the "IAR ARM C/C++ Compiler Reference Guide".

4 The Vanilla Port

The “vanilla” port shows how to use QP™ state machine frameworks on a “bare metal” ARM-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.

4.1 Compiler Options Used

The most important IAR compiler options used are as follows:

```
--interwork  
--cpu %ARM_CORE% --dlib_config %IAR_ARM%\ARM\INC\DLib_Config_Normal.h
```

In particular, the `--cpu %ARM_CORE%` option selects the ARM processor core, such as ARM7TDMI, ARM966E-S, and others. You choose this option implicitly by invoking the `make_<cpu>.bat` file.

The option `--interwork` specifies ARM/THUMB interworking code generation, which is code synthesized by the compiler and linker to perform CPU mode changes when calling ARM functions from THUMB, or THUMB functions from ARM. The option `--dlib_config %IAR_ARM%\ARM\INC\DLib_Config_Normal.h` specifies the normal configuration of the C/C++ runtime library. No locale interface, C locale, no file descriptor support, no multibytes in `printf` and `scanf`, and no hex floats in `strtod`.

The freedom of choosing the instruction set means that each individual module can be compiled either to ARM (`--cpu_mode arm`) or THUMB (`--cpu_mode thumb`) for best performance and memory usage.

4.2 The QF Port Header File

The QF header file for the ARM port is located in `<qp>\ports\arm\vanilla\iar\qf_port.h`. This file specifies the interrupt locking/unlocking policy (QF critical section) as well as the interrupt “wrapper” functions in assembly.

4.2.1 The QF Critical Section

This QF port uses the interrupt locking policy of saving and restoring the interrupt status described as in Section 7.3.1 of the “Practical UML Statecharts in C/C++, Second Edition” [PSiCC2]. This 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 enabled in the `QF_CRIT_ENTRY()` macro only if they were locked before the matching `QF_CRIT_ENTRY()` macro. Conversely, interrupts will remain locked after the `QF_CRIT_EXIT()` macro if they were locked before the matching `QF_CRIT_ENTRY()` macro.

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 is defined as follows:

```
/* fast unconditional interrupt disabling/enabling for ARM state */  
(1) #define QF_INT_DISABLE()      __asm("MSR cpsr_c,#(0x1F | 0x80 | 0x40)")  
(2) #define QF_INT_ENABLE()      __asm("MSR cpsr_c,#(0x1F)")  
  
(3) #if (__CPU_MODE__ == 1)      /* THUMB mode? */
```

```

/* QF interrupt locking/unlocking */
(4)  #define QF_CRIT_STAT_TYPE      unsigned long
(5)  #define QF_CRIT_ENTRY(stat_)  ((stat_) = QF_int_lock_SYS())
(6)  #define QF_CRIT_EXIT(stat_)   QF_int_unlock_SYS(stat_)

(7)  QF_CRIT_STAT_TYPE QF_int_lock_SYS(void);
(8)  void QF_int_unlock_SYS(QF_CRIT_STAT_TYPE key);

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

        #define QF_CRIT_STAT_TYPE      unsigned long
(9)  #define QF_CRIT_ENTRY(stat_)    do { \
(10)     (stat_) = __get_CPSR(); \
(11)     QF_INT_DISABLE(); \
        } while (0)
(12) #define QF_CRIT_EXIT(stat_)     __set_CPSR(stat_)

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

    #else

        #error Incorrect __CPU_MODE__. Must be ARM or THUMB.

    #endif

```

Listing 5 QF critical section definition

- (1-2) The macros `QF_INT_DISABLE()/QF_INT_ENABLE()` perform a very efficient unconditional interrupt disabling/enabling 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.

NOTE: In this QP 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_DISABLE()/QF_INT_ENABLE()` are more efficient than 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_DISABLE()/QF_INT_ENABLE()` can be simple, because they are specific only to the SYSTEM mode.

- (3) 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()`.
- (4) The `QF_CRIT_STAT_TYPE` is defined, which means that locking policy of “saving and restoring the interrupt status” is applied. In this method, the original ARM CPSR register is saved in the variable of the type `QF_CRIT_STAT_TYPE`.
- (5) The critical section entry macro resolves to a function call that returns the original ARM CPSR register and saves it in the argument `stat_`. The function `QF_int_lock_SYS()` is defined in assembly.

NOTE: This QP 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.

- (6) The critical section exit macro resolves to a function call that restores the original ARM CPSR register (actually only the control-bits of it) provided in the argument `stat_`. The function `QF_int_unlock_SYS()` is defined in assembly.
- (7-8) The prototypes of the interrupt locking/unlocking functions are declared. These functions are defined in assembly and are callable both in ARM and THUMB state.

NOTE: In C++ the interrupt locking/unlocking functions must be declared `extern "C"` to avoid C++ name decorating, which would make it difficult to define these functions in assembly.

- (9) The interrupt locking macro is defined inline in the ARM state.
- (10) 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`.
- (11) After saving the CPSR, the interrupts are efficiently locked with macro `QF_INT_DISABLE()` described in step (2) above.
- (12) 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?`.

The interrupt locking and unlocking functions are defined in the assembly file `<qp>\qf\arm\vanilla\iar\qf_port.s` as follows:

```

;-----
; Interrupt locking/unlocking
;-----

(1)   SECTION .text:DATA:NOROOT(2)
      PUBLIC  QF_int_lock_SYS, QF_int_unlock_SYS

(2)   CODE32
      QF_int_lock_SYS:
(3)   MRS    r0,cpsr           ; get the original CPSR in r0 to return
(4)   MSR    cpsr_c,#(SYS_MODE | NO_INT) ; disable both IRQ/FIQ in SYSTEM mode
(5)   BX    lr                ; return the original CPSR in r0

      CODE32
      QF_int_unlock_SYS:
(6)   MSR    cpsr_c,r0        ; restore the original CPSR from r0
(7)   BX    lr                ; return to ARM or THUMB

```

Listing 6 Interrupt locking and unlocking functions defined in the module `qf_port.s`.

- (1) The module is declared in the section `.text:DATA` located in the DATA area. As described in “ARM® IAR C/C++ Compiler Reference Guide” [IAR 08a], the `.text:DATA` 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 are entered in 32-bit ARM state.
- (3) The original value of `CPSR` is moved to `r0`, which is then returned from the `QF_int_lock_SYS` function.
- (4) The `IRQ` and `FIQ` 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 ARM port, 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()`).

- (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`, which is passed in the argument of the `QF_int_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.
- (7) The function `.QF_int_unlock_SYS` returns with the `BX` instruction, so the `T`-bit is set in the `CPSR_c` if the return address is a `THUMB` label.

4.2.2 Discussion of the QF 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 6(4)). 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 6(4)), 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 ARM port, 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.

4.3 Handling Interrupts

This generic “vanilla” 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.

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 6 Typical ARM system with an interrupt controller (external to the ARM core)

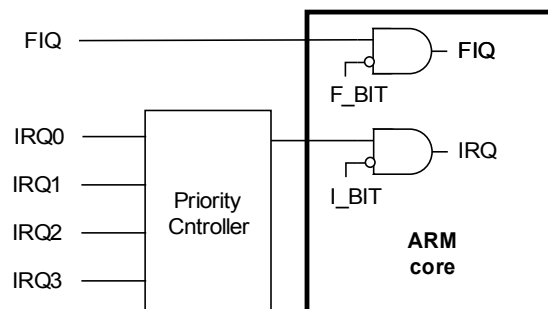
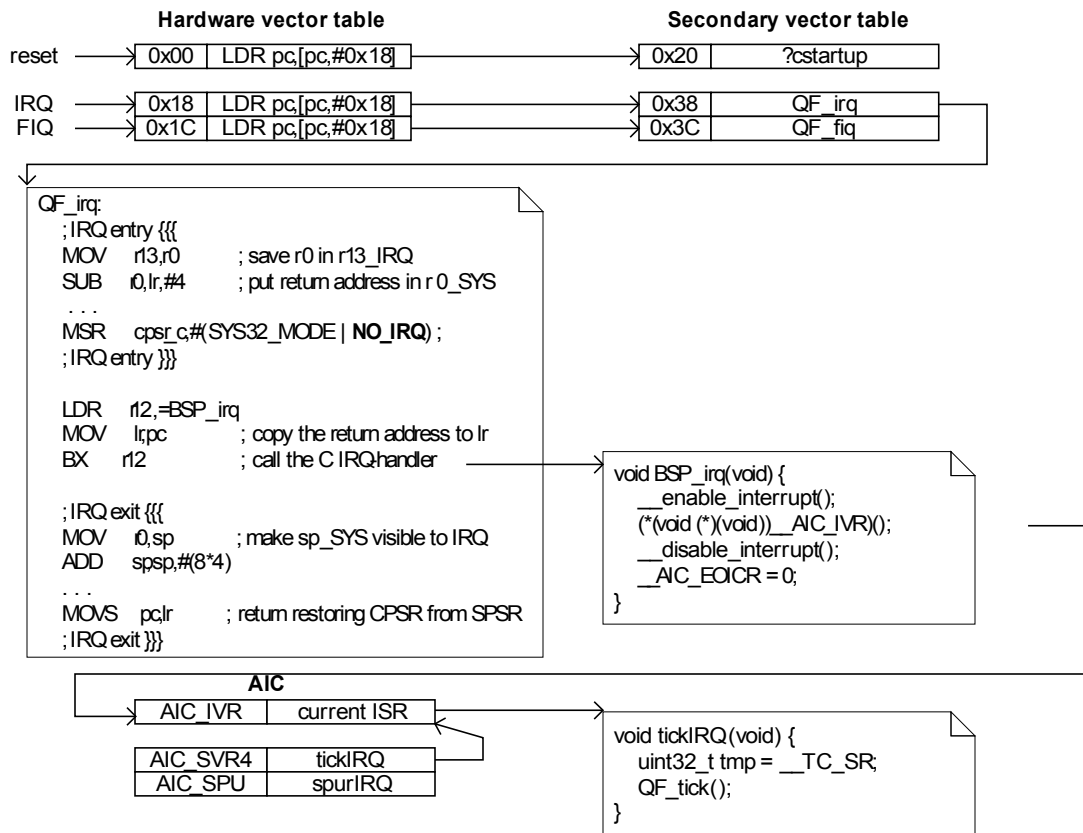


Figure 6 shows a typical ARM-based system with an interrupt controller. Typically, the interrupt prioritization is performed only with respect to the IRQ line, while the FIQ line is routed directly to the ARM core. The ARM core itself performs a 2-level interrupt prioritization between the FIQ and IRQ, 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 Application Note.

Figure 7 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 7. 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`.

Figure 7 Interrupt processing in the “vanilla” port to ARM with the Atmel’s AIC interrupt controller



4.3.1 The IRQ “Wrapper” Function in Assembly

The “vanilla” QF 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 `<qp>\ports\arm\vainlla\iar\src\qf_port.s`.

Listing 7 The QF_irq assembly wrapper for the “vanilla” QF port defined in qf_port.s.

```

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

(1)    SECTION .text:DATA:NOROOT(2)
        PUBLIC  QF_irq
        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
(27)   LDR     lr,[sp,#(6*4)] ; load return address from the SYS stack

```

```
(28)    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.

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
      v         R12
      stack     R3
      growth    R2
              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. 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.

4.3.2 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 6). 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 discussed in Section 4.2.1. Specifically, you would need to set only the I bit in the CPSR (see Listing 6(4)):

```
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.

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 `<qp>\ports\arm\vainlla\iar\src\qf_port.s`.

Listing 8 The QF_fiq assembly wrapper for the “vanilla” QF port defined in qf_port.s.

```

;-----
; FIQ assembly wrapper
;-----

SECTION .text: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
(3)  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 }}}

```

The `QF_fiq()` “wrapper” shown in Listing 8 is very similar to the IRQ wrapper (Listing 7), 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 always possible due to the ARM processor architecture).

4.3.3 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 `<qp>\ports\arm\vainlla\iar\src\qf_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 (see Listing 12(3-10)).

Listing 9 ARM Exception “Wrapper” Functions in Assembly defined in qf_port.s.

```

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:
    LDR    r0,Csting_swi
    B     QF_except
(1) QF_pAbort:
(2)    LDR    r0,Csting_pAbort
(3)    B     QF_except
QF_dAbort:
    LDR    r0,Csting_dAbort
    B     QF_except
QF_reserved:
    LDR    r0,Csting_reserved
    B     QF_except
(4) QF_except:
(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"

```

- (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.

4.3.4 Defining The BSP_irq() C-level IRQ Handler

The `QF_irq()` “wrapper” assembly function calls application-specific function `BSP_irq()` that performs interrupt processing in C. The `BSP_irq()` handler function is a regular C-function without any adornments (**no `__irq!`**), which is typically defined in the Board Support Package for the specific ARM silicon (more exactly for the specific interrupt controller integrated with the ARM core). The function can also be programmed in ARM or THUMB, but perhaps using ARM would lend a slightly better performance (if the function runs from a 32-bit wide memory).

While you can use the `BSP_irq()` callback function to implement directly the body of the IRQ interrupt, most likely you will use it to encapsulate the specific interrupt controller used with the ARM core. Typically, the function obtains the current vector address from the interrupt controller, calls this vector, and writes the EOI command to the interrupt controller (see Figure 7).

The following Listing 10 shows the implementations of the `BSP_irq()` functions for the Atmel’s AIC and Philips VIC interrupt vector controllers.

Listing 10 The `BSP_irq()` implementations for the Atmel’s AIC (top), and the Philips’ VIC (bottom).

```

/* BSP_irq() for the Atmel's AIC ..... */
(1a) __arm __ramfunc void BSP_irq(void) {

(2a)   QF_INT_ENABLE();
(3a)   (*(void (*)(void))__AIC_IVR)();           /* call the ISR handler */
(4a)   QF_INT_DISABLE();

(5a)   __AIC_EOICR = 0;                          /* write AIC_EOICR to clear interrupt */
}
/* BSP_irq() for the Philips' VIC ..... */
(1b) __arm __ramfunc void BSP_irq(void) {

(2a)   QF_INT_ENABLE();
(3b)   (*(void (*)(void))VICVectAddr)();        /* call the ISR handler */
(4a)   QF_INT_DISABLE();

(5b)   VICVectAddr = 0;                          /* write End-Of-Interrupt to the VIC */
}

```

(1a,b) The function `BSP_irq()` is defined as `__arm` to be able to use the most efficient interrupt unlocking and locking via the `QF_INT_DISABLE()/QF_INT_ENABLE()` defined as inline assembly in Listing 5(7-8). Also, compiling `BSP_irq()` in the ARM state avoids costly ARM->THUMB state switch (accomplished by a “call-veneer” synthesized by the compiler and linker). Finally, `BSP_irq()` is defined also as `__ramfunc` to execute from the RAM, which is only done for better performance. In most ARM system, the RAM is significantly faster than the ROM and also is typically 32-bit wide, while the ROM often is connected via 16-bit bus.

(2a,b) The IRQ interrupts are very efficiently unlocked (both IRQ and FIQ) via the `QF_INT_ENABLE()` macro so that the interrupt controller can handle interrupt nesting and prioritization. Please note that

enabling IRQs is safe, because the interrupt controller provides prioritization of the IRQs before they even reach the ARM core (see Figure 6).

(3a,b) The vector address is extracted from the AIC or the VIC. This address is then cast on a `(void (*)(void))` function pointer, so that the C compiler can invoke the interrupt service routine (ISR).

(4a,b) The interrupts are locked for the exit sequence.

(5a,b) After the ISR returns, the EOI command is written to the AIC or VIC. The `BSP_irq()` function must always return with interrupts **locked**.

4.3.5 The C-level ISRs

When an interrupt controller is used in `BSP_irq()`, you must initialize the interrupt controller with the addresses of the C-level ISRs and you must define these functions in C. These ISRs are normal C functions (**not** `__irq`-type functions). The ISRs can be also compiled to ARM or THUMB without restrictions. The following shows an example of the `tickISR()` ISR function for the Philips LPC213X ARM-based microcontroller.

Listing 11 An example of the `ISR_tick()` interrupt function in C.

```
(1) __arm void ISR_tick(void) {  
(2)     T1IR = 0x1;                /* clear the interrupt source */  
(3)     QF_tick();                /* perform clock-tick processing */  
}
```

(1) The C-level ISR is a regular `void (void)` C-function. The IRQ-type ISR is called in SYSTEM mode with interrupts unlocked at the ARM-core level.

NOTE: Here, the ISR is defined as an ARM function (via the `__arm` extended keyword) to avoid state switch from ARM to THUMB, because `BSP_irq()` is also defined as an ARM function. However, using ARM mode is here just a fine-tuning option and is not necessary for correctness. You could use THUMB mode (default) as well.

(2) Any level-sensitive interrupt sources must be cleared, such as the LPC2138 timer. Please note that even though the interrupts are unlocked at the ARM core level, they are still prioritized in the interrupt controller, so a level-sensitive interrupt source will not cause recursive ISR reentry.

(3) The work of the interrupt is performed with interrupts enabled at the ARM core level. Please note that the interrupt controller prevents the same level of interrupt from preempting the currently serviced level, so `QF_tick()` will never be reentered.

4.3.6 Defining The `BSP_fiq()` C-level IRQ Handler

The `QF_fiq()` “wrapper” assembly function calls application-specific function `BSP_fiq()` that performs interrupt processing in C. The `BSP_fiq()` handler function is a regular C-function without any adornments (**no** `__fiq!`), which is typically defined in the Board Support Package for the specific ARM silicon (more exactly for the specific interrupt controller integrated with the ARM core). The function can also be programmed in ARM or THUMB, but perhaps using ARM would lend a slightly better performance (if the function runs from a 32-bit wide memory).

Typically, you will use the `BSP_fiq()` callback function to implement **directly** the body of the FIQ interrupt handler. The following code snippet shows the template of the `BSP_fiq()` handler that you can customize in your applications:

```

__arm __ramfunc void BSP_fiq(void) {
    /* TBD: implement the FIQ handler directly right here */
    /* NOTE: Do NOT enable interrupts throughout the whole FIQ processing. */
    /* NOTE: Do NOT write EOI to the AIC */
}

```

NOTE: In contrast to IRQ, the FIQ line to the ARM processor core often has **no** priority controller, even though the ARM processor might be equipped with an interrupt controller (see Figure 6Figure 6). Therefore, it's important **not** to enable interrupts during handling of FIQ because without a priority controller this can lead to priority inversions. It's especially important to never enable the IRQ during processing of FIQ because this can lead to corruption of the `lr_IRQ` register if the FIQ happens to nest on top of IRQ, which is always possible in the ARM architecture.

4.3.7 Initializing the Vector Table and the Vectored Interrupt Controller

Auto-vectoring is NOT used, so the ARM vector table and the interrupt controller must be correctly initialized differently than in other QK port. The following listing shows the initialization placed in the `QF_onStart()` callback.

Listing 12 Initialization of the ARM vector table and the interrupt controller in `QF_onStart()` for the Philips LPC213x.

```

(1) __arm void QF_onStart(void) {
    /* hook the IRQ handler from the QK port */
(3)     *(uint32_t volatile *)0x24 = (uint32_t)&QF_undef;
(4)     *(uint32_t volatile *)0x28 = (uint32_t)&QF_swi;
(5)     *(uint32_t volatile *)0x2C = (uint32_t)&QF_pAbort;
(6)     *(uint32_t volatile *)0x30 = (uint32_t)&QF_dAbort;
(7)     *(uint32_t volatile *)0x34 = (uint32_t)&QF_reserved;

(8)     *(uint32_t volatile *)0x38 = (uint32_t)&QF_irq;
(9)     *(uint32_t volatile *)0x3C = (uint32_t)&QF_fiq;

    VICIntSelect = 0x0;          /* assign all interrupts to the IRQ category */

    /* Setting up Timer1 to handle the time tick interrupt.
     * Timer1 has priority 1 (second to the highest)
     */
    VICVectCntl1 = 0x25;
(10)  VICVectAddr1 = (uint32_t)&ISR_tick;
    VICIntEnable = (1 << 5);    /* enable TIMER 1 interrupt */

    . . .

    T1TCR = 0x1;                /* start the timer */

    . . .
}

```

(1) The interrupt initialization is performed in the QF callback `QF_onStartup()` [PSiCC2]. The function is defined as `__arm` to be able to use the efficient unconditional interrupt unlocking and locking via the `QF_INT_DISABLE()/QF_INT_ENABLE()` defined as inline assembly in Listing 5(7-8).

(3-9) The secondary vector table at 0x20 is initialized to the addresses of exception handlers. In particular, at label (9) the IRQ wrapper function `QF_irq()` is “hooked” at the address `0x20+0x18=0x38`.

- (10) The interrupt controller is initialized with the address of the ISR in C. Note that the ISRs in C are regular C-functions, without any special entry or exit.

4.4 Idle Loop Customization in the “Vanilla” Port

As described in Chapter 7 of [PSiCC2], the “vanilla” port uses the non-preemptive scheduler built into 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 locked, because the idle condition can be changed by any interrupt that posts events to event queues. `QF_onIdle()` **must** internally unlock interrupts, ideally atomically with putting the CPU to the power-saving mode. If the interrupt lock key is defined (the “save and restore” interrupt locking policy), `QF_onIdle()` must take the interrupt lock key as parameter, to be able to unlock the interrupts internally.

Because `QF_onIdle()` must enable interrupts internally, the signature of the function depends on the interrupt locking policy. In case of the “save and restore interrupt status” policy, which is used in this ARM port, the `QF_onIdle()` takes the interrupt status as a parameter, to be able to use the `QF_INT_ENABLE()` macro to unlock the interrupts.

The following Listing 13 shows an example implementation of `QF_onIdle()` for the Philips LPC2xxx CPU. Other ARM-based embedded microcontrollers (e.g., Atmel’s AT91) handle the power-saving mode very similarly.

Listing 13 QF_onIdle() callback for the Philips LPC2xxx CPU

```
(1) void QF_onIdle(void) { /* NOTE: called with interrupts DISABLED */  
(2)     PCON_bit.IDL = 1; /* go to idle mode to save power */  
(3)     QF_INT_ENABLE(); /* enable interrupts as soon as CPU clock starts */  
}
```

- (1) The signature of `QF_onIdle()` must be consistent with the interrupt locking policy. In case of the “save and restore interrupt status” policy, the function takes the interrupt lock key as the parameter.
- (2) Setting the `PCON_bit.IDL` bit stops the CPU clock on the LPC2xxx. Please note that the code stops executing at this line and that interrupts are still **locked**. The implementation of power saving in this microcontroller family is such that any active interrupt turns on the CPU clock, if it’s stopped.
- (3) Only after putting the CPU into low-power mode interrupts are unlocked (please note the use of the interrupt lock key).

5 The QK Port

The QK ports show how to use QP™ state machine frameworks on the ARM processor with QK, which is a very lightweight, **preemptive, priority-based kernel** designed specifically for QF (see Chapter 10 in PSiCC2). You should consider the QK port if your application requires deterministic, real-time performance and also the application can benefit from decoupling higher-priority tasks from lower-priority tasks in the time domain. Perhaps the best way to learn about QK implementation for the ARM processor is to study Chapter 10 in PSiCC2. You might also read the article “Build Super Simple Tasker” [Samek+06], which explains the inner-workings of a single-stack preemptive kernel, like QK.

One of the biggest advantages of QK is that porting QK to a new microprocessor is very easy. In fact the QK port to ARM is almost identical to the simplest “vanilla” port described in Section 3. The main difference between the two ports, which is visible at the application level, is that you use `QK_irq()` and `QK_fiq()` “wrapper” functions for interrupt processing (instead of `QF_irq()` and `QF_fiq()`, respectively). The other slight difference is that you customize the idle loop processing in a different way than in the “vanilla” port. This section focuses only on the differences from the “vanilla” port.

5.1 Compiler and Linker Options Used

The QK port uses exactly the same compiler options as the “vanilla” QP port described in Section 4.1.

5.2 The QK Port Header File

The QK header file for the ARM, IAR compiler is located in `<qp>\ports\arm\qk\iar\qk_port.h`. This file specifies the interrupt enabling/disabling policy (QK critical section) as well as the interrupt and exception handling “wrapper” functions defined in assembly.

5.2.1 The QK Critical Section

The QK port uses the same critical section as the “Vanilla” port described in Section 4.2.1. The critical section is defined in the header file `<qp>\ports\arm\qk\iar\qf_port.h`.

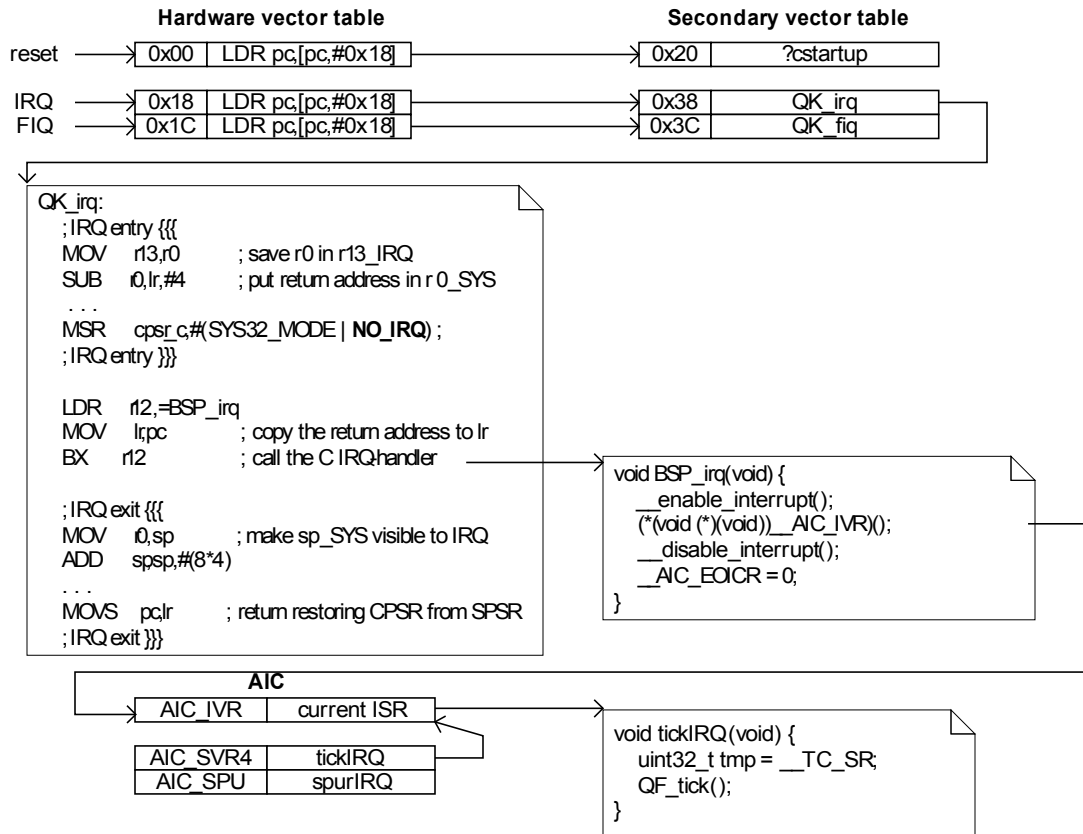
5.3 Handling Interrupts

This generic QK 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.

When used with an interrupt controller, the QK port assumes **no** “auto-vectoring”, which is described for example in the Atmel Application Note “Interrupt Management: Auto-vectoring and Prioritization” [Atmel 98b] (see also Section 4.3).

Figure 8 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 `QK_irq` and `QK_fiq`, respectively. Each of these “wrapper” functions, for example `QK_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 8. 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 `QK_irq` and `QK_fiq`.

Figure 8 Interrupt processing in the QK port to ARM with the Atmel's AIC interrupt controller



5.3.1 The IRQ “Wrapper” Function for QK

The QK 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 `<qp>\qk\arm\qk\iar\qk_port.s`.

Listing 14 The `QK_irq` assembly wrapper for the QK port defined in `qk_port.s`.

```
SECTION .text: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
STMFDB 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
(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 {{{
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

LDMFDB 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 14 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 Listing 7 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.
- (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) 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.

5.3.2 The FIQ “Wrapper” Function for QK

NOTE: Generally, you should avoid, if only possible, using the FIQ as general purpose interrupt, because in most ARM chips the FIQ interrupt is NOT prioritized by the interrupt controller. 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 discussed in Section 4.2.1. Specifically, you would need to set only the I bit in the CPSR (see Listing 6(4)):

```
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 QK or QF services because the critical section never masks the FIQ and consequently such NMI can corrupt critical QF data.

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 `<qp>\qk\arm\qk\iar\qk_port.s`.

Listing 15 The `QK_fiq` assembly wrapper for the QK port defined in `qk_port.s`

```
SECTION .text: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
    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)

    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) */

(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 interrupted IRQ

; We have interrupted a task. Call QK scheduler to handle preemptions
    MOV    r0,#SYS_MODE         ; set the interrupt unlock key
    LDR    r12,=QK_schedule_
    MOV    lr,pc                 ; copy the return address to link register
    BX    r12                    ; call QK_schedule_ (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 15 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 Listing 8 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.

- (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: Because of the ARM system architecture, the FIQ can always preempt IRQ, 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 15(1-4) detects this situation and prevents calling the QK scheduler.

5.4 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 4.4).

The following Listing 16 shows an example implementation of `QK_onIdle()` for the Philips LPC2xxx CPU. Other ARM-based embedded microcontrollers (e.g., Atmel’s AT91) handle the power-saving mode very similarly.

Listing 16 QK_onIdle() callback for the Philips LPC2xxx CPU

```

void QK_onIdle(void) {
    PCON_bit.IDL = 1;                /* go to idle mode to save power */
}

```

6 Controlling Placement of the Code in Memory and ARM/THUMB Compilation

A very important and often overlooked aspect for optimal system performance is controlling both the placement of the code in memory and the instruction set chosen to compile individual modules. The placement of the code in memory is most important. For example, on a typical ARM-based microcontroller, the code executing from the fast 32-bit wide on-chip SRAM can be three to four times faster than the same code executing from the 16-bit wide Flash. This is 300 to 400% difference!

The instruction set used can contribute to additional difference of 20 to 40% in the execution speed, ARM being faster than THUMB when executed from 32-bit wide memory, and slower, when executed from 16-bit wide memory.

Therefore, this Application Note puts a lot attention on giving you fine-level of control over the placement of the code in memory and instruction set compilation. Clearly, it's advantageous to expend some of the fast on-chip RAM to dramatically improve the performance.

In the IAR toolset, you can instruct the compiler to place the code in RAM by assigning it to the `CODE_I` section (see "ARM® IAR C/C++ Compiler Reference Guide"). You can achieve this either by the compiler option `--section .text=.textrw`, or on function-by-function basis using the `__ramfunc` extended keyword. Similarly, you can instruct the compiler to produce either THUMB or ARM code by the compiler option `--cpu_mode thumb` or `--cpu_mode arm`, respectively.

NOTE: For some reason, the IAR startup code will not correctly copy the code from ROM to RAM, unless at least one function in the application is declared with the `__ramfunc` keyword.

To enable fine-tuning of the code, the `make.bat` script for each QP component allows you to specify the code segment placement and instruction set for each individual fine-granularity module in the library. For example, the following fragment of the `make.bat` for building the QF library shows the choices made for several QF components:

```
. . .
%CC% --cpu_mode arm --section .text=.textrw %CCFLAGS% -o%BINDIR% %SRCDIR%\qf_tick.c
%CC% --cpu_mode thumb %CCFLAGS% %CCINC% -o%BINDIR% %SRCDIR%\qa_sub.c
. . .
```

The module `qf_tick.c`, for instance, which contains the `QF_tick()` function invoked from the interrupt, is compiled to ARM and placed in the fast RAM memory, because it is clearly a "hot-spot" during the execution. On the other hand, the module `qa_sub.c` performs subscriptions to events, which happens typically only during the initialization transient. Such code must be merely correct, but does not need to be fast. Therefore `qa_sub.c` is compiled to THUMB and placed in the default location, which is ROM (Flash). Of course, you can fine-tune the code placement any way you like for your particular project.

You could also apply similar strategy to your application code. Additionally, you could use the IAR extended keyword `__ramfunc` to place selected functions in RAM. This option was not used in the generic QP code, because of the non-portability of the IAR extension. However, your code, specific to the CPU and tool-chain can benefit from such proprietary extensions.

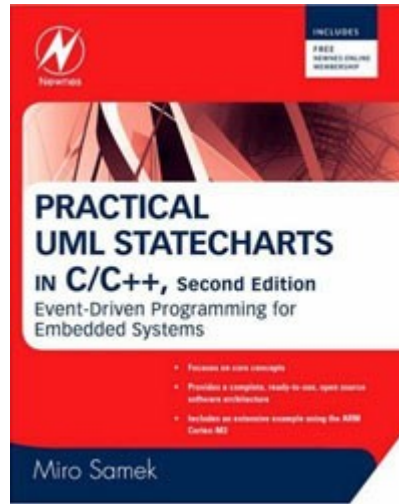
7 References

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, 2 nd 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.

8 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

