



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

# **QDK<sup>TM</sup>** **ColdFire-IAR**

**Document Revision C**  
**August 2008**

Copyright © Quantum Leaps, LLC

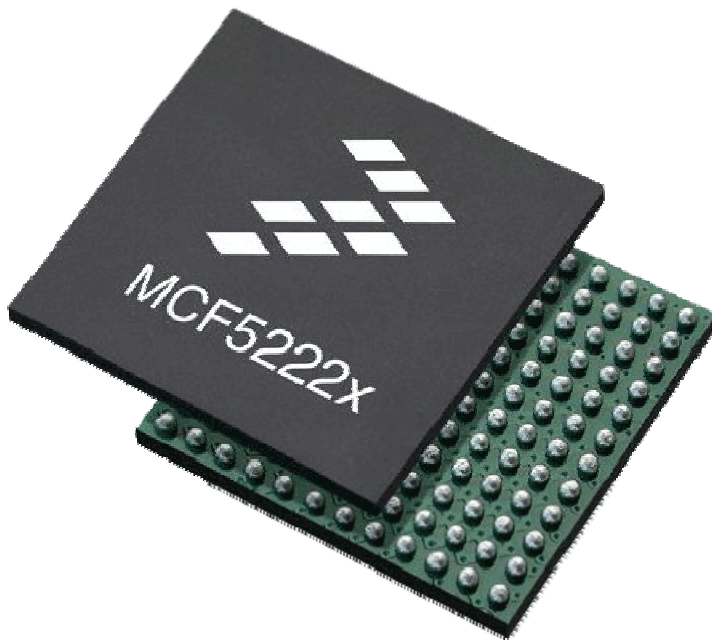
[www.quantum-leaps.com](http://www.quantum-leaps.com)  
[www.state-machine.com](http://www.state-machine.com)



# Table of Contents

<b>1</b>	<b>Introduction</b> .....	<b>1</b>
1.1	What's Included in the QDK-ColdFire-IAR?.....	2
1.2	Licensing QDK-ColdFire-IAR .....	2
<b>2</b>	<b>Getting Started</b> .....	<b>3</b>
2.1	Installation .....	3
2.2	Building the QP Libraries.....	5
2.3	Building the Examples .....	6
2.4	Running the Examples .....	7
<b>3</b>	<b>The Vanilla QP Port</b> .....	<b>8</b>
3.1	Compiler Options Used .....	8
3.2	Linker Options Used .....	8
3.2.1	Specifying Stack and Heap Sizes .....	8
3.3	The qep_port.h Header File .....	9
3.4	The qf_port.h Header File.....	9
3.4.1	The QF Object Size Configuration .....	10
3.4.2	The QF Critical Section .....	10
3.5	Startup Code.....	11
3.6	BSP for ColdFire.....	12
3.6.1	BSP Header file bsp.h .....	12
3.6.2	Initialization and the CPU Clock and Peripherals .....	12
3.6.3	Starting Interrupts in QF_onStartup() .....	13
3.6.4	Interrupt Service Routines (ISRs).....	13
3.6.5	Interrupt Vector Table and Default Exception Handlers .....	14
3.6.6	QP Idle Loop Customization in QF_onIdle() .....	15
3.6.7	Assertion Handling Policy in Q_onAssert() .....	16
<b>4</b>	<b>The QK Port</b> .....	<b>17</b>
4.1	Compiler and Linker Options Used .....	17
4.2	Linker Options Used .....	17
4.3	The qk_port.h Header File .....	17
4.4	Disabling Interrupts Before Entering an ISR.....	19
4.5	Setting up and Starting Interrupts in QF_onStart() .....	20
4.6	Writing ISRs for QK.....	20
4.7	QK Idle Processing Customization in QK_onIdle() .....	20
<b>5</b>	<b>The Quantum Spy (QS) Instrumentation</b> .....	<b>22</b>
5.1	QS Time Stamp Callback QS_onGetTime() .....	24
5.2	QS Trace Output in QF_onIdle()/QK_onIdle() .....	25
5.3	Invoking the QSpy Host Application .....	27
<b>6</b>	<b>Related Documents and References</b> .....	<b>28</b>
<b>7</b>	<b>Contact Information</b> .....	<b>29</b>

---



## 1 Introduction

This **Quantum Development Kit™** (QDK) describes how to use Quantum Platform™ (QP) with the Freescale ColdFire processors and the IAR Embedded Workbench for ColdFire [ColdFire RM 07]. The actual hardware/software used to test this QDK is described below (see also Figure 1):

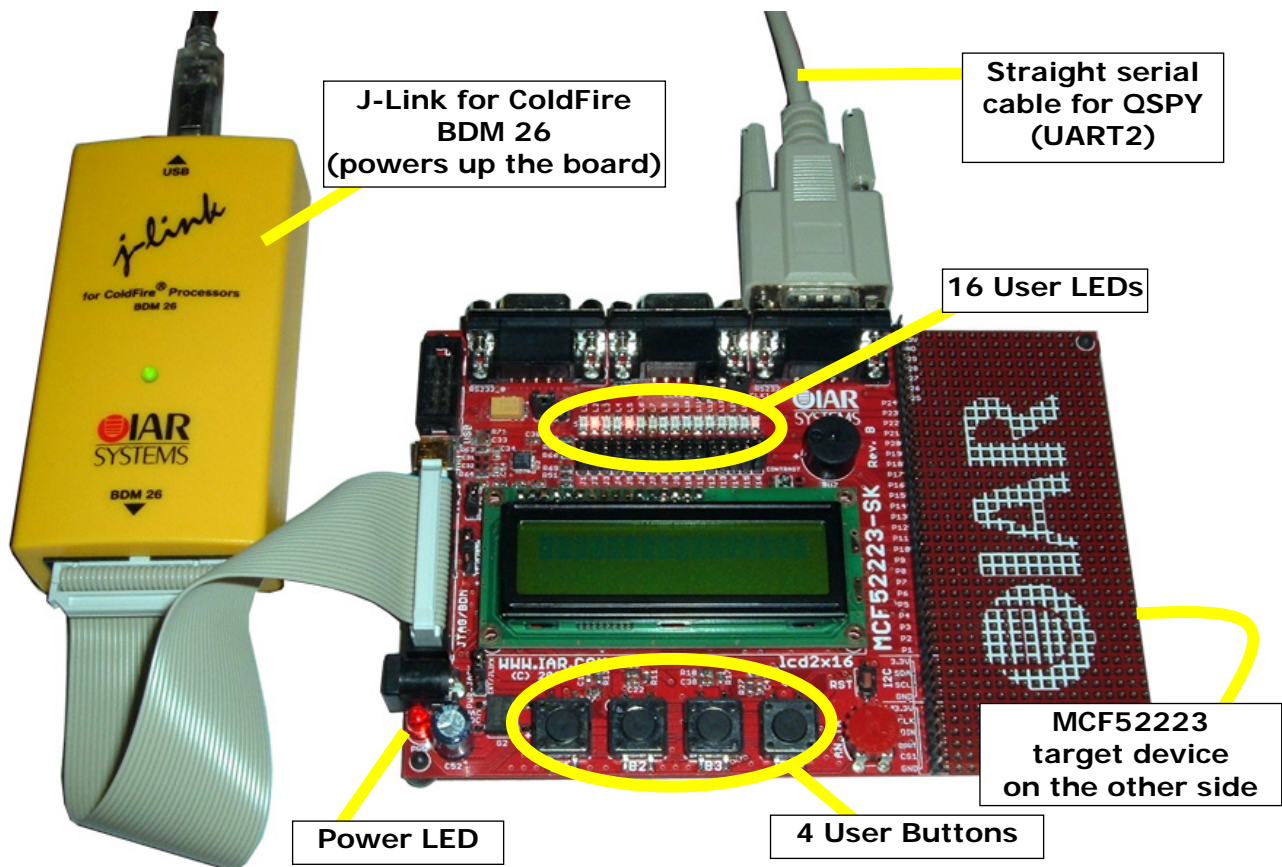


Figure 1 IAR ColdFire development kit MCF52223-SK.

1. MCF52223-SK from IAR Systems.
2. The evaluation version of IAR Embedded Workbench for ColdFire KickStart™ version 1.20A.
3. Quantum Leaps QP/C and QP/C++ v4.0.00.

As shown in Figure 1, the IAR development kit MCF52223-SK contains the ColdFire board as well as the USB debug pod J-Link for ColdFire processors that connects to the board via the 20-pin ribbon cable. This QDK has been tested with the MCF52223 device with 32KB of RAM and 256KB of on-chip flash. However, the described port should be applicable to all devices based on ColdFire V2 Instruction Set Architecture ISA\_A+.

The IAR Embedded Workbench for ColdFire KickStart Edition is included with the MCF52223-SK development kit, and is also available for download from the IAR website [www.iar.com](http://www.iar.com).

## 1.1 What's Included in the QDK-ColdFire-IAR?

This QDK provides two QP ports to ColdFire V2 and two example applications implemented with the IAR Embedded Workbench for ColdFire:

1. QP port to the cooperative “Vanilla” kernel and the Dining Philosophers Problem (DPP) example described in the Application Note [QP AN-DPP 08]; and
2. QP port to the preemptive QK kernel and the Dining Philosophers Problem (DPP) example using this port.

Both cooperative and preemptive versions of the DPP application contain the Q-SPY instrumentation that uses the UART2 of the ColdFire processor with DMA.

**NOTE:** This QDK™ is applicable to both C and C++ versions of QP. Special sections cover the C/C++ differences, whenever such differences are important or at least non-trivial.

## 1.2 Licensing QDK-ColdFire-IAR

The **Generally Available (GA)** distribution of QDK-ColdFire-IAR available for download from the [www.state-machine.com](http://www.state-machine.com) 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](http://www.state-machine.com/licensing).

## 2 Getting Started

This section describes how to install, build, and use QDK-ColdFire-IAR based on two versions of the DPP example. This information is intentionally included early in this document, so that you could start using the QDK as soon as possible. It is strongly recommended that you read the “QP Tutorial” section of the “QP Reference Manual” [QP/C 08] (available electronically in the QP Distribution) before you start experimenting with this QDK.

**NOTE:** This QDK assumes that the standard QP distribution consisting of QEP, QF, QK, and QS has been installed, before installing this QDK.

### 2.1 Installation

The QDK code is distributed in a ZIP archive (qdkc\_coldfire-iar\_<ver>.zip, where <ver> stands for a specific QDK-ColdFire-IAR version, such as 4.0.00). You should uncompress the archive into the same directory into which you’ve installed all the standard QP components. The installation directory you choose will be referred henceforth as QP Root Directory <qp>. The following Listing 1 shows the directory structure and selected files included in the QDK-ColdFire-IAR distribution. (The QP directory structure is described in detail in a separate Application Note: “[QP Directory Structure](#)”) [QP 07]:

```

<qp>/
+-ports/
  +-coldfire/
    +-qk/
      +-iar/
        +-dbg/
          +-libqep.r68
          +-libqf.r68
          +-libqk.r68
          +-libqs.r68
        +-rel/
        +-spy/
        +-make.bat
        +-qep_port.h
        +-qf_port.h
        +-qk_port.h
        +-qs_port.h
      +-vanilla/
        +-iar/
          +-dbg/
            +-libqep.r68
            +-libqf.r68
            +-libqs.r68
          +-dbg/
          +-spy/
          +-make.bat
          +-qep_port.h
          +-qf_port.h
          +-qs_port.h
    +-examples/
      +-coldfire/
  
```

- QP-root directory
- QP ports
- ColdFire port
- QK (Quantum Kernel) ports
- IAR ColdFire compiler
- QP libraries – Debug configuration
- QEP library
- QF library
- QK library
- QS library
- QP libraries – Release configuration
- QP libraries – Spy configuration
- make script for building the QP libraries
- QEP port
- QF port
- QK port
- QS port
- “vanilla” ports
- IAR ColdFire compiler
- QP libraries – Debug configuration
- QEP library
- QF library
- QS library
- QP libraries – Release configuration
- QP libraries – Spy configuration
- make script for building the QP libraries
- QEP port
- QF port
- QS port
- subdirectory containing the QP example files
- ColdFire port

```

+-qk/           - QK examples
+-iar/         - IAR ColdFire compiler
+-dpp-qk-mc52223-sk/ - Dining Philosophers example with QK for MCF52223-SK
+-dbg/        - directory containing the Debug build
+-rel/        - directory containing the Release build
+-spy/        - directory containing the Spy build

+-dpp-qk.eww   - workspace for the IAR Embedded Workbench
+-cstartM52223.s68 - Startup code for ColdFire
+-bsp.c       - Board Support Package for MCF52223-SK
+-bsp.h       - BSP header file
+-dpp.h       - the DPP header file
+-dpp-qk_vec_table.s68 - the ColdFire interrupt vector table for DPP
+-intr_handlers.h - interrupt handlers prototypes
+-intr_handlers.c - interrupt handlers definitions
+-main.c      - the main function
+-phil.o.c    - the Philosopher active object
+-table.c     - the Table active object

+-vanilla/    - "vanilla" examples (non-preemptive kernel of QF)
+-iar/         - IAR ColdFire compiler
+-dpp-mc52223-sk/ - Dining Philosophers example for MCF52223-SK
+-dbg/        - directory containing the Debug build
+-rel/        - directory containing the Release build
+-spy/        - directory containing the Spy build

+-dpp-qk.eww   - workspace for the IAR Embedded Workbench
+-cstartM52223.s68 - Startup code for ColdFire
+-bsp.c       - Board Support Package for MCF52223-SK
+-bsp.h       - BSP header file
+-dpp.h       - the DPP header file
+-dpp_vec_table.s68 - the ColdFire interrupt vector table for DPP
+-intr_handlers.h - interrupt handlers prototypes
+-intr_handlers.c - interrupt handlers definitions
+-main.c      - the main function
+-phil.o.c    - the Philosopher active object
+-table.c     - the Table active object
  
```

**Listing 1** Selected QP directories and files after installing QDK-ColdFire-IAR. The high-lighted elements are included in the standard QDK-ColdFire-IAR distribution.

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

**NOTE:** To streamline and simplify the QP-library build process, Quantum Leaps software does not use the vendor-specific IDEs, such as the IAR Embedded Workbench IDE, for building the QP libraries. Instead, this QDK provides *command-line* build process based on simple batch scripts.

The build process for your application is largely independent on the QP-library builds. In fact, once you have the QP libraries, you typically don't need to rebuild them—at least not on the daily basis as you work on your application. This QDK uses the IAR EWCF IDE to build the example applications, but you are free to use any other build strategy.

The code distribution contains all the batch file `make.bat` for building all the libraries located in `<qp>\ports\coldfire\vanilla\iar\` directory. For example, to build the debug version of all the QP libraries for the ColdFire, with the IAR ColdFire compiler, you open a console window on a Windows PC, change directory to `<qp>\ports\coldfire\vanilla\iar\`, and invoke the batch by typing at the command prompt the following command:

```
make
```

The `make` process should produce the QP libraries in the location: `<qp>\ports\coldfire\vanilla\iar\dbg\`. The `make.bat` assumes that the IAR ColdFire toolset has been installed in the directory `c:\tools\IAR\ColdFire_KS_1.20`.

**NOTE:** You need to adjust the symbol `IAR_CF` at the top of the `make.bat` file if you've installed the IAR.

In order to take advantage of the Q-SPY instrumentation, you need to build the Spy version of the QP libraries. You achieve this by invoking the `make.bat` utility with the "spy" target, like this:

```
make spy
```

The `make` process should produce the QP libraries in the directory: `<qp>\ports\coldfire\vanilla\iar\spy\`.

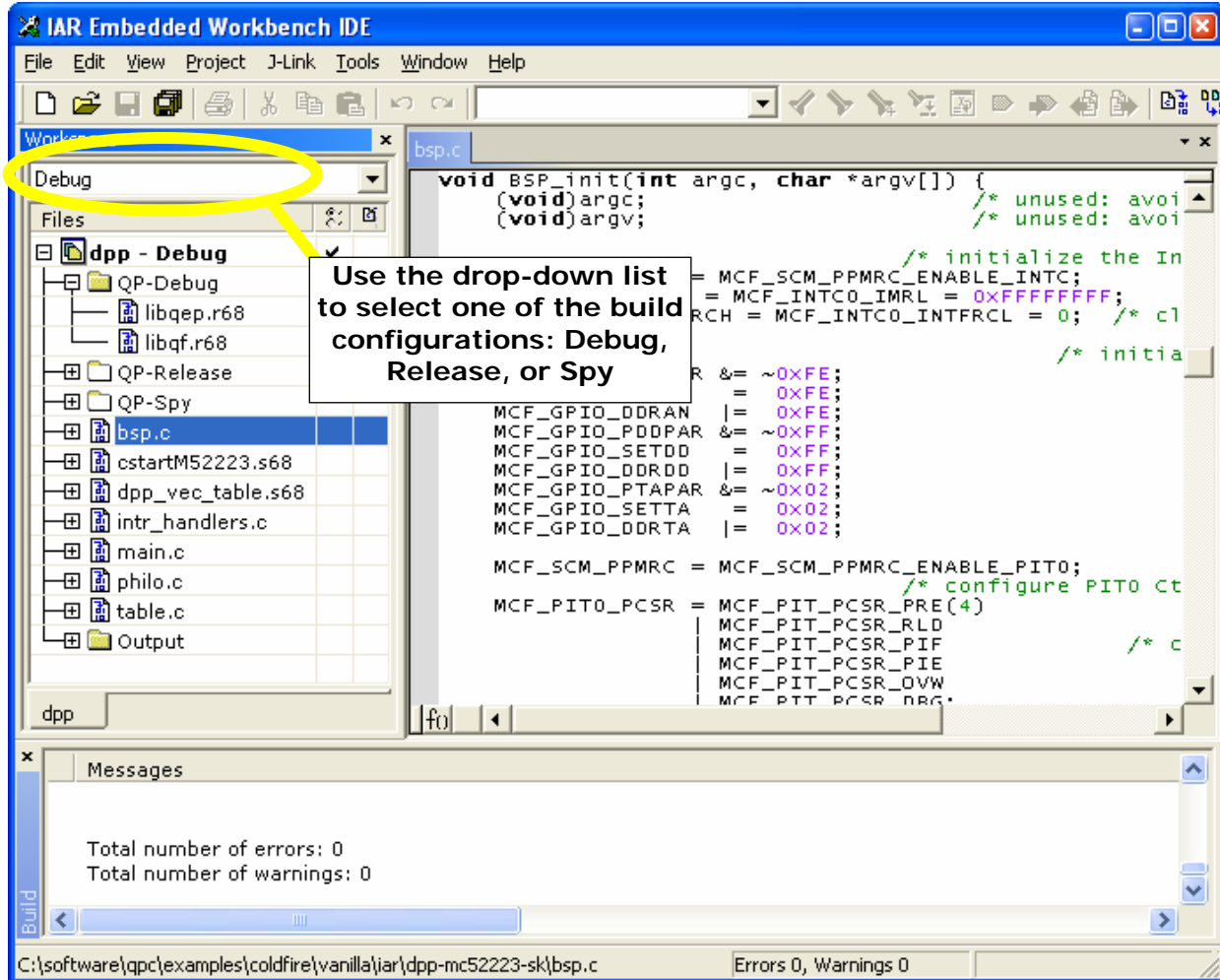
You choose the build configuration by providing a target to the `make.bat` utility. The default target is "dbg". Other targets are "rel", and "spy" respectively. The following table summarizes the targets accepted by `make.bat`.

Software Version	Build command
Debug (default)	make
Release	make rel
Spy	make spy

**Table 1 Supported build configurations for the Debug, Release, and Spy versions**

## 2.3 Building the Examples

This QDK-ColdFire-IAR uses the DPP (“Dining Philosophers Problem”) example described in Application Note [QP AN-DPP 08]. The QDK contains the IAR Embedded Workbench workspaces to build the examples. Each workspace contains three build configurations: Debug, Release, and Spy. The workspaces are located in <qp>\examples\coldfire\vanilla\iar\dpp-mcf52223-sk\dpp.esw for the “vanilla” version, and in <qp>\examples\coldfire\qk\iar\dpp-qk-mcf52223-sk\dpp-qk.esw for the QK version.



**Figure 2 IAR Embedded Workbench for ColdFire with the dpp workspace.**

Figure 2 shows the IAR Embedded Workbench IDE with the `dpp.esw` workspace open. You can use the drop-down list in the upper-left corner to select one of the build configurations: Debug, Release, or Spy. The Spy configuration requires installing the QS component.

To build any selected configuration, you simply press F7, or alternatively select Project->Make menu. To work with the MCF52223 board, the Debugger option should be set to use the “J-Link” debugger.

## 2.4 Running the Examples

You load and debug the application using the IAR Embedded Workbench (see Figure 3). The application uses the User LEDs 1-5 to display the status of the Philosophers in the DPP application. LED-16 shows the activity of the idle loop.

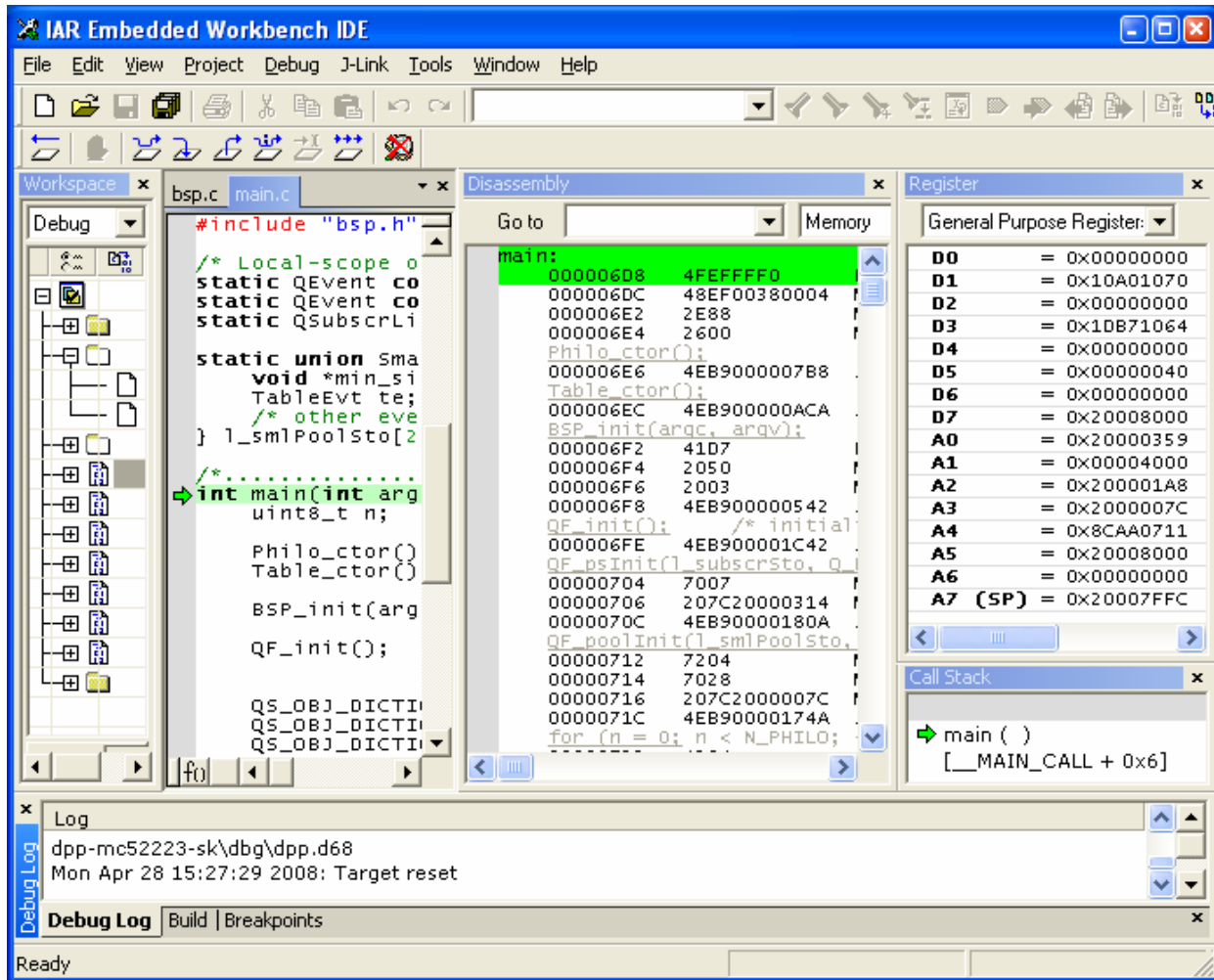


Figure 3 The DPP example stopped at the IAR EWCF debugger

## 3 The Vanilla QP Port

---

The “vanilla” port shows how to use QP with the non-preemptive Vanilla kernel built into the QF. In case of ColdFire, the “vanilla” the QP port is very similar to the generic “vanilla” port described in Chapter 7 in the PSiCC2 book [Samek 08].

In this port, the ColdFire core operates in the Supervisor mode at all times (SR[13] == 1), which is the default mode out of reset.

### 3.1 Compiler Options Used

---

You set the compiler options for building the QP libraries in the make.bat file located in the <qp>\-ports\col dFi re\i ar\ directory. The compiler options for building the examples are set through the IAR EW IDE. Either way, the most important compiler options are as follows:

```
--core v2 --isa isa_a+ --code_model far --dl ib_conf ig  
C:\tool s\IAR\Col dFi re_KS_1. 20\cf\LI B\dl cfaffn. h  
-I C:\tool s\IAR\Col dFi re_KS_1. 20\cf\INC\
```

In particular, the options **--core v2 --isa isa\_a+** mean that the ColdFire V2 device with the Instruction Set Architecture ISA+ is used. The include directories prefix (-I C:\tool s\IAR\Col dFi re\_KS\_1. 20\...) points to the installation directory of the IAR toolset and could be different on your system, if you’ve installed the IAR toolset in a different location.

**NOTE:** You select this option through the IAR EW IDE by right-clicking on the project in the “Project Workspace” pane and selecting the “Options...” pop-up menu. This opens the Options tabbed dialog box. The compiler options are determined through the “C/C++ Compiler” category and several tabs under this category.

### 3.2 Linker Options Used

---

Linker options are only relevant for building the applications (not the QP libraries). In this QDK, you set the linker options for building the applications through the IAR EW IDE. The most important linker options are as follows:

```
tool s\IAR\Col dFi re_KS_1. 20\cf\LI B\ -f  
C:\tool s\IAR\Col dFi re_KS_1. 20\cf\CONFI G\I nkm52223. xcl -rt -s __program_start -  
D__FLASHBE GI N=0 -D__FLASHEND=3FFFF -D__RAMBE GI N=20000000  
-D__RAMEND=20007FFF -D__VBR=0 -D__VBR_ADDRESS=0 -D__I PSBAR=40000000 -D_CSTACK_SI ZE=200 -  
D_HEAP_SI ZE=0 ...
```

In particular, the linker command file I nkm52223. xcl , located in the IAR directory, describes the memory map of the ColdFire device (MCF52223 in this case).

#### 3.2.1 Specifying Stack and Heap Sizes

The sizes of the Main Stack and the heap are specified through the IAR EW IDE. The IDE then adds these selections as command line parameters to the linker (see Linker Options above). Please note that the values are in hexadecimal, so the constant -D\_CSTACK\_SI ZE=200 sets the stack size to 512<sub>10</sub> bytes. Note also, that this QDK does not use the heap (heap size is zero). In the non-preemptive “Vanilla” port, you could use the heap by simply configuring a non-zero size of the heap. In the

preemptive QK port, you should be **very careful** with the heap, because it represents shared resource.

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

The QEP header file for the ColdFire port with the IAR compiler is located in <qp>\ports\coldfire\vani ll a\iar\qep\_port.h.

```

/* 2-byte signals (64K signal space) */
#define Q_SIGNAL_SIZE      2
#include <stdint.h>        /* exact-width integers, WG14/N843 C99, 7.18.1.1 */
#include "qep.h"          /* QEP platform-independent public interface */

```

#### Listing 2 The qep\_port.h header file

The event signal size Q\_SIGNAL\_SIZE is configured to use 2-bytes (64K signals). The IAR compiler supports the C99-standard <stdint.h> header file (exact-width integer types), which is included in the qep\_port.h.

### 3.4 The qf\_port.h Header File

The QF header file for the ColdFire port with the IAR compiler is located in <qp>\ports\coldfire\vani ll a\iar\qf\_port.h. The following sub-sections focus on explaining the QF configuration established by the qf\_port.h header file shown in Listing 3.

```

/* various QF object sizes configuration for this port, see NOTE01 */
(1) #define QF_MAX_ACTIVE      16
(2) #define QF_EVENT_SIZE_SIZE  2
(3) #define QF_QUEUE_CTR_SIZE  1
(4) #define QF_MPOOL_SIZE_SIZE  2
(5) #define QF_MPOOL_CTR_SIZE  2
(6) #define QF_TIMEEVT_CTR_SIZE 4

/* QF critical section entry/exit */
(7) /* QF_INT_KEY_TYPE not defined */
(8) #define QF_INT_LOCK(key_)    __disable_interrupt()
(9) #define QF_INT_UNLOCK(key_)  __enable_interrupt()

(10) #include <intrinsic.h>      /* IAR intrinsic functions */

(11) #include "qep_port.h"        /* QEP port */
(12) #include "qqueue.h"         /* Vanilla QF/C port needs event-queue */
(13) #include "qmpool.h"         /* Vanilla QF/C port needs memory-pool */
(14) #include "qsched.h"        /* Vanilla QF/C port needs non-preemptive scheduler */
(15) #include "qf.h"            /* QF platform-independent public interface */
(16) #include "qpset.h"         /* Vanilla QF/C port needs priority-set */

/* The maximum number of active objects in the application */
(1) #define QF_MAX_ACTIVE      63
(2) #define QF_EVENT_SIZE_SIZE  2
(3) #define QF_QUEUE_CTR_SIZE  2
(4) #define QF_MPOOL_SIZE_SIZE  2
(5) #define QF_MPOOL_CTR_SIZE  2
(6) #define QF_TIMEEVT_CTR_SIZE 4

```

```
/* QF critical section entry/exit */
/* "saving and restoring interrupt status" policy */
(7) #define QF_INT_KEY_TYPE      uint16_t
(8) #define QF_INT_LOCK(key_)   do { \
    (key_) = __get_status_register(); \
    __disable_interrupts(); \
} while (0)
(9) #define QF_INT_UNLOCK(key_) __set_status_register(key_)

(10) #include <intrinsics.h>          /* IAR intrinsic functions */

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

**Listing 3 The qf\_port.h header file.**

### 3.4.1 The QF Object Size Configuration

The first part of the qf\_port.h header file defines limits and sizes of various internal data structures used in the QF and the applications.

Listing 3(1) QF\_MAX\_ACTIVE defines the maximum number of active objects that QF can manage. Here this limit is set to 63, which is the maximum supported by the current version of QF. You can reduce this number to save some RAM.

Listing 3(2) Maximum event size is set to 2-bytes, meaning that the size of a single event can be up to 64K bytes.

Listing 3(3) Maximum event queue counter size is set to 2-bytes, meaning that a single event queue can hold up to 64K events.

Listing 3(4) Maximum memory pool element size is set to 2-bytes, meaning that a pool can manage blocks of up to 64K bytes each.

Listing 3(5) The memory pool counter size is set to 2-bytes, meaning that a pool can manage up to 64K memory blocks.

Listing 3(5) The timer counter size is set to 4-bytes, meaning that a maximum timeout can be  $2^{32}-1$  clock ticks.

### 3.4.2 The QF Critical Section

The ColdFire core supports interrupt prioritization. However, the software is responsible for managing the interrupt mask level encoded in three-bits of the Status Register (SR[8-10]). In fact, the only way to lock interrupts on ColdFire is to momentarily raise the mask level to maximum (level 7), and then to lower it again to the level before the critical section. This implies that ColdFire requires the policy of "saving and restoring interrupt status".

The QF critical section policy for ColdFire is defined in Listing 3(7-9) as follows:

Listing 3(7) The interrupt lock key is defined, which means that the interrupt status is saved upon the entry to critical section and then restored upon the exit from the critical section.

Listing 3(8-9) The macros QF\_INT\_LOCK() / QF\_INT\_UNLOCK() define the CPU and compiler-specific interrupt locking mechanism. The intrinsic functions \_\_get\_status\_register() \_\_disable\_interrupt() ,and \_\_enable\_interrupt() are provided by the IAR compiler. They are synthesized in-line and don't cause the function-call overhead.

Listing 3(10) The prototypes of the intrinsic functions are provided in the <i>ntrinsics.h> header file.

### 3.5 Startup Code

The IAR toolset provides standard startup code for ColdFire, which is adequate for QP. For example, the startup code for MCF52223 is provided in the assembly module cstartM52223.s68 shown below:

```

MODULE    ?M52223_CSTART
RSEG     CSTACK: DATA(2)

PUBLIC   __program_start
PUBLIC   ?CSTART_DEVICE
EXTERN  ?C_STARTUP

EXTERN  __FLASHBEGIN
EXTERN  __RAMBEGIN
EXTERN  __VBR
EXTERN  __IPSBAR

COMMON  INTVEC: CODE(2)
ORG     0x00
__program_start:
DC32    SFE(CSTACK)
DC32    ?CSTART_DEVICE

;-----
;
;      Clear CFM configuration field
;-----

RSEG     CFMCONFIG: CODE(1)
DC32    0x00000000
DC32    0x00000000
DC32    0x00000000
DC32    0x00000000
DC32    0x00000000
DC32    0x00000000

;-----
;
;      Device configuration.
;-----

RSEG     RCODE: CODE(1)

#include "i052223.h"

?CSTART_DEVICE:
MOVE.L  #__FLASHBEGIN, D7
OR.L    #1, D7
MOVEC.L D7, FLASHBAR

MOVE.L  #__RAMBEGIN, D7
OR.L    #1, D7
MOVEC.L D7, RAMBAR

MOVE.L  #__VBR, D7
MOVEC.L D7, VBR

MOVE.L  #__IPSBAR, D7
OR.L    #1, D7
MOVE.L  D7, (IPSBAR_RESET).L

```

```

JMP      (?C_STARTUP).L

ENDMOD

END

```

## 3.6 BSP for ColdFire

The Board Support Package (BSP) for ColdFire is quite straightforward. However, there are some important details that you need to pay attention to.

### 3.6.1 BSP Header file bsp.h

The BSP header file defines the intended system clock tick rate in the constant `BSP_TICKS_PER_SEC`, the BSP initialization function prototype `BSP_init()`:

```

#define BSP_TICKS_PER_SEC    30

void BSP_init(int argc, char *argv[]);
void BSP_displayPhilStat(uint8_t n, char const *stat);
void BSP_busyDelay(void); /* to artificially extend RTC processing in the DPP */

```

### 3.6.2 Initialization and the CPU Clock and Peripherals

The BSP initialization routine configures the Interrupt Controller, enables the peripherals used in your application, and configures the low-power mode. You most likely need to customize it for your application (see the datasheet for your ColdFire device and the board schematics.)

```

#define SYS_CLK              48000000L
#define PER_CLK              (SYS_CLK/2)

#define AN_TR_PIN           MCF_GPIO_PORTAN_PORTANO
#define AN_TR_CHANNEL       0

/* interrupt IDs ... */
#define INTR_ID_EPF1        ( 1+64)
#define INTR_ID_PIT0        (55+64)
#define INTR_ID_RTC         (63+64)

#define PIT_PERIOD          (PER_CLK / (BSP_TICKS_PER_SEC * 16))
#define PIT0_LEVEL          2

/* ..... */
void BSP_init(int argc, char *argv[]) {
    (void)argc; /* unused: avoid the compiler warning */
    (void)argv; /* unused: avoid the compiler warning */

    /* initialize the Interrupt Control Module */
    MCF_SCM_PPMRC = MCF_SCM_PPMRC_ENABLE_INTC; /* enable ICM clock */
    MCF_INTCO_IMRH = MCF_INTCO_IMRL = 0xFFFFFFFF; /* mask all interrupts */
    MCF_INTCO_INTFRCH = MCF_INTCO_INTFRCL = 0; /* clear forced interrupts */

    /* initialize the LEDs lines... */
    MCF_GPIO_PANPAR &= ~0xFE;
    MCF_GPIO_SETAN  = 0xFE;
    MCF_GPIO_DDRAN  |= 0xFE;
    MCF_GPIO_PDDPAR &= ~0xFF;
}

```

```

MCF_GPIO_SETDD = 0xFF;
MCF_GPIO_DDRDD |= 0xFF;
MCF_GPIO_PTAPAR &= ~0x02;
MCF_GPIO_SETTA = 0x02;
MCF_GPIO_DDRTA |= 0x02;

/* configure the low-power mode */
MCF_PMM_LPCR = MCF_PMM_LPCR_LPMDOZE;
MCF_PMM_LPCR = MCF_PMM_LPCR_XLPM_IP1(0); /* any interrupt wakes up */

if (QS_INIT((void *)0) == 0) { /* initialize the QS software tracing */
    Q_ERROR();
}
}

```

### 3.6.3 Starting Interrupts in QF\_onStartup()

ColdFire provides two Programmable Interval Timers (PIT0 and PIT1), specifically designed to deliver the periodic system time tick. This port uses PIT0. The configuration of PIT0 is done in the QF\_onStartup() callback. QP invokes the QF\_onStartup() callback just before starting the event loop inside QF\_run().

```

#define PIT_PERIOD (PER_CLK / (BSP_TICKS_PER_SEC * 16))
#define PIT0_LEVEL 2

/* ..... */
void QF_onStartup(void) {
    MCF_SCM_PPMRC = MCF_SCM_PPMRC_ENABLE_PIT0; /* enable PIT0 clock */
    /* configure PIT0 Ctrl and status register */
    MCF_PIT0_PCSR = MCF_PIT_PCSR_PRE(4) /* prescaler 1/16 */
    | MCF_PIT_PCSR_RLD /* enable reload */
    | MCF_PIT_PCSR_PIF /* clear pending interrupt */
    | MCF_PIT_PCSR_PIE /* enable interrupt */
    | MCF_PIT_PCSR_OVW /* enable overwrite */
    | MCF_PIT_PCSR_DBG; /* enable debug mode */

    MCF_PIT0_PMR = PIT_PERIOD - 1; /* set the PMR for the desired rate */
    /* configure PIT0 interrupt */
    *(&MCF_INTCO_ICR01 + INTR_ID_PIT0 - 64 - 1) = (PIT0_LEVEL << 3);
    /* enable PIT0 interrupt */
    MCF_INTCO_IMRH &= ~(1UL << (INTR_ID_PIT0 & 0x1F));

    MCF_PIT0_PCSR |= MCF_PIT_PCSR_EN; /* enable PIT0 counting */
    __enable_interrupts(); /* Global interrupts enable */
}

```

### 3.6.4 Interrupt Service Routines (ISRs)

The IAR compiler supports writing interrupts in C via the extended keyword `__interrupt`. In the non-preemptive case used in this QDK, you don't need to do anything special upon interrupt entry or exit.

The following listing shows the ISR servicing PIT0 system clock tick interrupt, which calls the QF\_tick() function. As in most interrupts in ColdFire, you must explicitly clear the interrupt source at the beginning of the ISR.

```

__interrupt void PIT0_IntrHandler(void) {
    MCF_PIT0_PCSR |= MCF_PIT_PCSR_PIF; /* clear the interrupt source */
#ifdef Q_SPY
    QS_tickTime += PIT_PERIOD; /* add the rollover */
#endif
}

```

```

} QF_tick(); /* process all armed time events */

```

### 3.6.5 Interrupt Vector Table and Default Exception Handlers

The IAR compiler toolset generates the default ColdFire interrupt vector table (IVT) in assembly. The application programmer is supposed to manually insert all active interrupt handlers into the IVT. The following listing shows how the `PIT0_IntrHandler()` ISR has been added to the IVT. Other interrupts should be added similarly.

The default IVT contains several exception handlers and default interrupt handlers. The default (do-nothing) implementation of these handlers is located in the file `intr_handlers.c`. These default handlers should be customized for the specific application.

```

PROGRAM ?COLDFIRE_VECTOR_TABLE

EXTERN ReservedHandler
EXTERN AccessErrorHandler
EXTERN AddressErrorHandler
EXTERN IllegalInstrHandler
EXTERN DivByZeroHandler
EXTERN PrivilegeViolationHandler
EXTERN TraceHandler
EXTERN UnimplementedLineA_OpcOdeHandler
EXTERN UnimplementedLineF_OpcOdeHandler
EXTERN DebugHandler
EXTERN FormatErrorHandler
EXTERN SpuriousIntrHandler
EXTERN Trap0Handler
EXTERN Trap1Handler
.
.
.
EXTERN Trap15Handler

EXTERN PIT0_IntrHandler

COMMON INTVEC: CODE(2)
ORG 0x8
DC32 AccessErrorHandler ;; 2 0x008 Access error
DC32 AddressErrorHandler ;; 3 0x00C Address error
DC32 IllegalInstrHandler ;; 4 0x010 Illegal instruction
DC32 DivByZeroHandler ;; 5 0x014 Divide by zero
DC32 ReservedHandler ;; 6 0x018 reserved
DC32 ReservedHandler ;; 7 0x01C reserved
DC32 PrivilegeViolationHandler ;; 8 0x020 Privilege violation
DC32 TraceHandler ;; 9 0x024 Trace
DC32 UnimplementedLineA_OpcOdeHandler ;; 10 0x028 Unimplemented line-a opcode
DC32 UnimplementedLineF_OpcOdeHandler ;; 11 0x02C Unimplemented line-f opcode
DC32 DebugHandler ;; 12 0x030 Debug interrupt
DC32 ReservedHandler ;; 13 0x034 reserved
DC32 FormatErrorHandler ;; 14 0x038 Format error
DC32 ReservedHandler ;; 15 0x03C reserved
.
.
.
DC32 ReservedHandler ;; 23 0x05C reserved
DC32 SpuriousIntrHandler ;; 24 0x060 Spurious interrupt
.
.
.
DC32 Trap0Handler ;; 32 0x080 Trap # 0 instructions
DC32 Trap1Handler ;; 33 0x084 Trap # 1 instructions
DC32 Trap2Handler ;; 34 0x088 Trap # 2 instructions
.
.
.
DC32 Trap15Handler ;; 47 0x0BC Trap # 15 instructions
DC32 ReservedHandler ;; 48 0x0C0 reserved

```

```

DC32   ReservedHandler          ;; 118 0x1D8 reserved
DC32   PIT0_IntrHandler         ;; 119 0x1DC <-- PIT0 interrupt flag
DC32   ReservedHandler          ;; 120 0x1E0 PIT1 interrupt flag

DC32   ReservedHandler          ;; 255 0x3FC reserved

END

```

### 3.6.6 QP Idle Loop Customization in QF\_onIdle()

The non-preemptive “Vanilla” kernel can very easily detect the situation when no events are available, in which case the scheduler calls the `QF_onIdle()` callback. You can customize the `QF_onIdle()` function to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QF_onIdle()` is called repetitively from the event loop whenever the event loop has no more events to process, in which case only an interrupt can provide new events. The `QF_onIdle()` callback is called with interrupts **locked**, because the determination of the idle condition might change by any interrupt posting an event.

The ColdFire instruction set provides a special instruction `STOP` for stopping the CPU clock. As described in the “ColdFire Reference Manual” [ColdFire RM 07], the `STOP` instruction can be executed with **interrupts locked**. The instruction takes one immediate parameter, which is loaded to the SR register. This allows entering the low-power mode and unlocking interrupts atomically, exactly how it should be done.

The following Listing 4 shows the `QF_onIdle()` callback that puts ColdFire into the low-power mode.

```

(1) void QF_onIdle(QF_INT_KEY_TYPE key) {           /* interrupts LOCKED, NOTE01 */
    /* toggle the LED-16 on and then off, see NOTE02 */
(2)   MCF_GPIO_CLRTA = ~0x02;
    MCF_GPIO_SETTA = 0x02;

    #ifdef Q_SPY
(3)   .
(4)   #if defined NDEBUG
        /* Put the CPU and peripherals to the low-power mode.
         * you might need to customize the clock management for your application,
         * see the ColdFire Reference Manual for your particular device.
         */
(5)   __stop(0x2000);                               /* privileged STOP instruction */
    #else
(6)   QF_INT_UNLOCK(key);                           /* unlock the interrupts */
    #endif
}

```

**Listing 4 QF\_onIdle() for the “vanilla” ColdFire port**

Listing 4(1) The signature of `QF_onIdle()` depends on the interrupt locking policy. For the policy of “saving and restoring interrupt status” used in this port (see Section 3.4.2), the `QF_onIdle()` callback takes the interrupt lock key parameter.

Listing 4(2) The LED-16 is used to visualize the idle loop activity. The brightness of the LED is proportional to the frequency of invocations of the idle loop. Note that the LED is toggled with interrupts locked, so no interrupt execution time contributes to the brightness of the User LED.

Listing 4(3) This part of the code is only used in the QSPY build configuration. In this case the idle callback is used to transmit the trace data using the UART2 of the ColdFire device.

Listing 4(4) The low-power mode might interfere with debugging, therefore this code is only active in the Release configuration (the macro NDEBUG defined).

Listing 4(5) The low-power mode is entered with the STOP instruction, which loads the SR with the immediate parameter. The value 0x2000 corresponds to Supervisor mode with the interrupt level of zero (all interrupts allowed).

**NOTE:** You most likely need to customize the transition to the low-power mode by configuring the Power Management Module and additionally powering down the unused peripherals.

Listing 4(6) If the low-power mode is not used (in Debug and Spy configurations), the interrupts are simply unlocked.

### 3.6.7 Assertion Handling Policy in Q\_onAssert()

All QP components internally use assertions to detect errors in the way application is using the QP services. You need to define how the application reacts in case of assertion failure by providing the callback function Q\_onAssert(). Typically, you would put the system in fail-safe state and try to reset. It is also a good idea to log some information as to where the assertion failed.

The following code fragment shows the Q\_onAssert() callback. The function simply locks all interrupts and enters a for-ever loop. This policy is only adequate for testing, but probably is not adequate for production release.

```
void Q_onAssert(char const Q_ROM *file, int line) {
    (void)file; /* avoid compiler warning */
    (void)line; /* avoid compiler warning */
    _disable_interrupts(); /* make sure that all interrupts are disabled */
    for (;;) { /* NOTE: replace the loop with reset for final version */
    }
}
```

## 4 The QK Port

---

This section describes how to use QP on ColdFire with the fully preemptive QK real-time kernel. The benefit is very fast, fully deterministic task-level response and that execution timing of the high-priority tasks (active objects) will be virtually insensitive to any changes in the lower-priority tasks. The downside is bigger RAM requirement for the stack (see Chapter 10 in PSiCC2 [Samek 08]). Additionally, as with any preemptive kernel, you must be very careful to avoid any sharing of resources among concurrently executing active objects, or if you do need to share resources, you need to protect them with the QK priority-ceiling mutex.

**NOTE:** QK incurs far less overhead and provides responsiveness exceeding that of any traditional multiple-stack real-time kernel, at the fraction of the RAM/ROM footprint (see the ESD article “Build a Super Simple Tasker”, [Samek+ 06]). The non-blocking restrictions of this kernel type are irrelevant for executing state machines.

### 4.1 Compiler and Linker Options Used

---

The compiler and linker used in the QK port are identical to those used in the “vanilla” port described earlier.

**NOTE:** With the preemptive QK kernel, you must be very careful to provide adequate stack size by setting the stack size via the General Options tab in the Options dialog box in the IAR IDE. The preemptive configuration with QK requires the stack to be significantly deeper than the non-preemptive “Vanilla” configuration. You need to adjust the size of this stack to be large enough for your application.

### 4.2 Linker Options Used

---

The linker options are identical as in the “vanilla” configuration described earlier.

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

---

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

You configure and customize QK through the header file `qk_port.h`, which is located in the QP ports directory `<qp>\ports\coldfire\qk\iar\`. The preemptive QK requires a specific initialization as well as specific interrupt entry and exit actions that are performed in the two QK macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()` defined in the `qk_port.h` header file:

```
/* QK interrupt entry and exit */
(1) #define QK_ISR_ENTRY(level_) do { \
(2)     ++QK_intNest; \
(3)     __set_status_register(__get_status_register() \
        & (0xF8FF | ((level_) << 8))); \
    } while (0)

(4) #define QK_ISR_EXIT() do { \
(5)     __asm("move.w #0x2700, sr"); \
```

```
(6)  --QK_intNest_; \
(7)  if (QK_intNest_ == (uint8_t)0) { \
(8)      QK_schedule_(0x2000); \
      } \
    } while (0)

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

### Listing 5 qk\_porth.h header file

Listing 5(1) The QK\_ISR\_ENTRY() macro must be placed at the start of every ISR to inform the QK kernel that an ISR is entered. The macro argument is the interrupt mask level 'level\_', which corresponds to the mask level of the ISR that calls the QK\_ISR\_ENTRY() macro (see also the upcoming Section "Disabling Interrupts Before Entering an ISR").

**NOTE:** As described in Section "Disabling Interrupts Before Entering an ISR", all ISRs in the QK port are entered with the interrupt mask level set to 7 (maximum). After performing critical updates, the QK\_ISR\_ENTRY() macro must then restore the mask to the level corresponding to the priority of the current interrupt, so that the ColdFire core can prioritize interrupts.

Listing 5(2) The QK\_intNest variable is incremented to inform the QK kernel about the entry to the interrupt context. As long as QK\_intNest variable is not zero, the QK kernel won't engage the scheduler to handle preemptions, as no task can preempt an interrupt.

Listing 5(3) The interrupt mask level can be lowered to the level of the current interrupt.

**NOTE:** The interrupt mask level could be set simply by loading immediate value to the SR register (`__set_status_register(0x2000 | ((level_) << 8))`). However, due to a bug in the IAR compiler, using an immediate argument to the intrinsic function `__set_status_register()` causes an error.

Listing 5(4) The QK\_ISR\_EXIT() macro must be placed at the end of every ISR to inform the QK kernel that an ISR is exited.

Listing 5(5) The interrupt exit sequence begins with disabling all interrupts by loading 0x2700 to the SR. (This is equivalent to `__set_status_register(0x2700)`, but as described above causes problem in the current version of the IAR compiler).

Listing 5(6) The QK\_intNest variable is decremented to account for exiting one interrupt level.

Listing 5(7-8) The QK scheduler is called only when the nesting level QK\_intNest variable drops to zero. The scheduler must handle asynchronous preemptions in this case.

Listing 5(9) The qk\_port.h must always include the platform-independent QK interface.

## 4.4 Disabling Interrupts Before Entering an ISR

As described in the “ColdFire Reference Manual” [ColdFire RM 07], “All ColdFire processors inhibit interrupt sampling during the first instruction of all exception handlers. This allows any handler to disable interrupts effectively, if necessary, by raising the interrupt mask level” (“ColdFire Reference Manual”, Section 3.5). This feature is essential for the correct operation of the QK kernel, because QK requires to safely disable interrupt nesting until the QK\_intNest variable is incremented to inform the QK kernel about the entry to the interrupt context (see Listing 5(2)).

Unfortunately, the ISR entry synthesized by the IAR compiler (when you use the `__interrupt` extended keyword) does **not** raise the interrupt mask level in the first instruction<sup>1</sup>. A simple work-around is to write an “interrupt veneer” assembly code that locks interrupts in the first instruction and then jumps to the ISR. The following Listing 6 shows how to add such an “interrupt veneer” to the PIT0 interrupt handler in the ColdFire IVT.

```

;-----
;          Interrupt Veneers
;-----
(1)      RSEG      RCODE: CODE(1)
(2)      EXTERN   PIT0_IntrHandler
(3)      PIT0_IntrVeneer:
(4)      MOVE.W   #0x2700, SR           ; prevent interrupt nesting
(5)      JMP      (PIT0_IntrHandler).L ; jump to the ISR in C
;-----
          EXTERN   XYZ_IntrHandler     ; add other “interrupt veneers” as follows
XYZ_IntrVeneer:
          MOVE.W   #0x2700, SR           ; prevent interrupt nesting
          JMP      (XYZ_IntrHandler).L  ; jump to the ISR in C
;-----
          COMMON  INTVEC: CODE(2)
          ORG     0x8
;-----
(6)      DC32     ReservedHandler      ;; 118 0x1D8 reserved
          DC32     PIT0_IntrVeneer     ;; 119 0x1DC <-- PIT0 interrupt flag
          DC32     ReservedHandler      ;; 120 0x1E0 PIT1 interrupt flag
;-----

```

**Listing 6 Adding the “interrupt” veneer to the PIT0 interrupt in the QK application (file \exampl es\col dfi re\col dfi re\qk\i ar\dpp-qk-mc52223-sk\dpp-qk\_vec\_tabl e.s68).**

Listing 6(1) This directive declares a relocatable code segment for the “interrupt veneers”.

Listing 6(2) This directive imports the external symbol `PIT0_IntrHandler`, which is defined in C.

Listing 6(3) This label defines the “interrupt veneer” entry point. You place this label in the IVT.

Listing 6(4) The first instruction of the interrupt locks interrupts by raising the interrupt mask level to maximum (0x7). This instruction is guaranteed to execute before any other interrupt can preempt the currently serviced interrupt.

<sup>1</sup> Other compilers for ColdFire, such as CodeWarrior 7.0, do raise the mask level, and even allow programmer to decide what value should be written to the SR in the first instruction of an ISR. This deficiency of the IAR compiler has been reported to IAR and perhaps will be resolved in the future releases of this compiler.

Listing 6(5) The “interrupt veneer” performs then a long jump to the ISR written in C (with the `__interrupt` keyword).

Listing 6(6) The “interrupt veneer” (rather than the ISR in C) is placed in the IVT.

**NOTE:** You need to add “interrupt veneers” to **all** compiler-synthesized ISRs.

## 4.5 Setting up and Starting Interrupts in QF\_onStart()

Setting up interrupts (e.g., ) for the preemptive QK kernel is identical as in the non-preemptive case. Please refer to Section 3.6.3.

## 4.6 Writing ISRs for QK

QK must be informed about entering and exiting every ISR, so that it can perform asynchronous preemptions. You inform the QK kernel about the ISR entry and exit through the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, respectively (see Listing 5(1,4)). You need to call these macros in every ISR. The following listing shows the ISR the file `<qp>\examples\coldfire\qk\iar\dpp-qk-mcf52223-sk\bsp.c`.

```

#define PITO_LEVEL      2
. . .
(1) __interrupt void PITO_IntrHandler(void) {
      MCF_PITO_PCSR |= MCF_PIT_PCSR_PIF;          /* clear the interrupt source */
      #ifdef Q_SPY
      QS_tickTime += PIT_PERIOD;                  /* add the rollover */
      #endif
(2)   QK_ISR_ENTRY(PITO_LEVEL);                    /* inform QK about entering the ISR */
(3)   QF_tick();                                  /* process all armed time events */
(4)   QK_ISR_EXIT();                              /* inform QK about exiting the ISR */
}

```

**Listing 7 Example of an ISR for QK.**

Listing 7(1) The “interrupt veneer” (rather than the ISR in C) is placed in the IVT.

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

QK can very easily detect the situation when no events are available, in which case QK calls the `QK_onIdle()` callback. You can use `QK_onIdle()` to suspended the CPU to save power, if your CPU supports such a power-saving mode. Please note that `QK_onIdle()` is called repetitively from the event loop whenever the event loop has no more events to process, in which case only an interrupt can provide new events. The `QK_onIdle()` callback is called with interrupts **unlocked** (which is in contrast to the `QF_onIdle()` callback used in the non-preemptive configuration, see Section 3.6.5).

The following Listing 8 shows the `QK_onIdle()` callback that puts ColdFire core into the low-power mode. Note that the callback `QK_onIdle()` is different from the callback `QF_onIdle()` used in the “Vanilla” kernel.

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

```
    QF_INT_KEY_TYPE intKey;

    /* toggle the LED-16 on and then off, see NOTE01 */
(2)   QF_INT_LOCK(intKey);
(3)   MCF_GPIO_CLRTA = ~0x02;           /* switch LED-16 on */
(4)   MCF_GPIO_SETTA = 0x02;          /* switch LED-16 off */
(5)   QF_INT_UNLOCK(intKey);

    #ifdef Q_SPY
(6)   . . .
(7)   #elif defined NDEBUG
        /* Put the CPU and peripherals to the low-power mode.
         * you might need to customize the clock management for your application,
         * see the ColdFire Reference Manual for your particular device.
         */
(8)   __stop(0x2000);                 /* privileged STOP instruction */
    #endif
}
```

### Listing 8 QK\_onIdle() for the preemptive QK configuration

Listing 8(1) The QK\_onIdle() function is called with interrupts **unlocked**.

Listing 8(2) The interrupts are locked to prevent preemptions when the LED is on.

Listing 8(3-4) This QK port uses the LED-16 the MCF52223-SK board to visualize the idle loop activity. The LED is rapidly toggled on and off as long as the idle condition is maintained, so the brightness of the LED is proportional to the CPU idle time (the wasted cycles). Please note that the LED is on in the critical section, so the LED intensity does not reflect any ISR or other processing.

Listing 8(5) Interrupts are unlocked.

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

Listing 8(6) This part of the code is only used in the Q-SPY build configuration. In this case the idle callback is used to transmit the trace data using the UART2 of the ColdFire device.

Listing 8(7) The following code is only executed when no debugging is necessary (release version).

Listing 8(8) The STOP instruction is generated using the intrinsic function. The instruction puts the device into a low-power mode and simultaneously writes 0x2000 (Supervisor mode, interrupts unlocked) to the SR.

**NOTE:** You most likely need to customize the transition to the low-power mode by configuring the Power Management Module and powering down the unused peripherals.

## 5 The Quantum Spy (QS) Instrumentation

This QDK demonstrates how to use the Quantum Spy (QS) to generate real-time trace of a running QP application. Normally, the QS instrumentation is inactive and does not add any overhead to your application, but you can turn the instrumentation on by defining the `Q_SPY` macro and recompiling the code.

Quantum Spy (QS) is a software tracing facility built into all QP components and also available to the Application code. QS allows you to gain unprecedented visibility into your application by selectively logging almost all interesting events occurring within state machines, the framework, the kernel, and your application code. QS software tracing is minimally intrusive, offers precise time-stamping, sophisticated runtime filtering of events, and good data compression (see Chapter 11 in PSiCC2 [Samek 08]).

QS can be configured to send the real-time data out of the serial port of the target device. On the MCF52223 MCU, QS uses the built-in UART2 to send the trace data out (see Figure 1), so the QSPY host application can conveniently receive the trace data on the host PC. Also, to make the tracing minimally intrusive, the trace output uses the DMA3 channel to send the trace data with minimal CPU overhead. The complete implementation of the QS software-tracing output consists of several steps, which are all summarized in Listing 9 (file `bsp.c`).

```

#define PIT_PERIOD      (PER_CLK / (BSP_TICKS_PER_SEC * 16))

(1) #ifndef Q_SPY
(2)     OSTimeCtr QS_tickTime;

        #define UART_BAUD  115200

(3)     enum AppRecords {                               /* application-specific trace records */
        PHILO_STAT = QS_USER
    };
    #endif

    /*.....*/
    #ifndef Q_SPY
    /*.....*/
(4) uint8_t QS_onStartup(void const *arg) {
(5)     static uint8_t qsBuf[2*1024];                    /* buffer for Quantum Spy */
        extern uint32_t __RAMBEGIN;

(6)     QS_initBuf(qsBuf, sizeof(qsBuf));

        /* set port UC to initialize UTXD2 */
(7)     MCF_GPIO_PUCPAR = MCF_GPIO_PUCPAR_UTXD2_UTXD2; /* config UTXD2 pin */
        MCF_UART2_UCR  = MCF_UART_UCR_RESET_TX;        /* reset Transmitter */
        MCF_UART2_UCR  = MCF_UART_UCR_RESET_MR;        /* reset the Mode Register */
        /* config no parity, 8-bits, no echo, no loopback, 1 stop bit */
        MCF_UART2_UMR1 = MCF_UART_UMR_PM_NONE | MCF_UART_UMR_BC_8
        | MCF_UART_UMR_CM_NORMAL | MCF_UART_UMR_SB_STOP_BITS_1;
        /* set the Tx baud rate generator from the SYSTEM CLOCK */
        MCF_UART2_UCSR = MCF_UART_UCSR_TCS_SYS_CLK;
        MCF_UART2_UBG1 = (uint8_t)((SYS_CLK / (UART_BAUD * 32)) >> 8);
        MCF_UART2_UBG2 = (uint8_t)(SYS_CLK / (UART_BAUD * 32));

        MCF_UART2_UCR  = MCF_UART_UCR_TX_ENABLED;      /* enable transmitter */

        /* set Back-Door Enable bit for DMA access in MCF_SCM_RAMBAR */
(8)     MCF_SCM_RAMBAR = (uint32_t)&__RAMBEGIN | (MCF_SCM_RAMBAR_BDE | 0x1);
(9)     MCF_SCM_MPR    |= 0x04; /* Master Privilege Register bit-2 DMA master */

```

```

(10) MCF_SCM_DMAREQC |= MCF_SCM_DMAREQC_DMACH3(0x0E); /* DMA3 for UART2 TX */

MCF_DMA3_DCR = 0x00; /* clear the DCR to avoid CE error */
MCF_DMA3_DSR = 0x01; /* reset the DSR */

QS_tickTime = PIT_PERIOD - 1; /* to start the timestamp at zero */

/* setup the QS filters... */
QS_FILTER_ON(QS_ALL_RECORDS);
QS_FILTER_OFF(QS_QF_ACTIVE_ADD);
. . .

return (uint8_t)1; /* return success */
}
/*.....*/
void QS_onCleanup(void) {
}
/*.....*/
(11) void QS_onFlush(void) {
    uint16_t len = 0xFFFF; /* request as much as possible */
    uint8_t const *block;
    QF_INT_KEY_TYPE intKeyLock;
    QF_INT_LOCK(intKeyLock);
    while ((block = QS_getBlock(&len)) != (uint8_t *)0) {
        QF_INT_UNLOCK(intKeyLock);

        /* wait in-line while DMA byte counter not zero */
        while ((MCF_DMA3_BCR & 0x00FFFFFF) != 0) {

            MCF_DMA3_DCR = 0x00; /* clear the DCR to avoid CE error */
            MCF_DMA3_DSR = 0x01; /* reset the DSR */
            MCF_DMA3_SAR = (uint32_t)block; /* set the source */
            MCF_DMA3_DAR = (uint32_t)&MCF_UART_UTB(2); /* destination UART2 */
            MCF_DMA3_BCR = MCF_DMA_BCR_BCR(len); /* byte counter */

            /* external Request, cycle Steal, dest & source = 8 bit
             * dest address NOT incremented source incremented by 1
             */
            MCF_DMA3_DCR = MCF_DMA_DCR_EEXT
                | MCF_DMA_DCR_CS
                | MCF_DMA_DCR_SINC
                | MCF_DMA_DCR_SSIZE(01)
                | MCF_DMA_DCR_DSIZE(01);

            len = 0xFFFF; /* re-load the length */
            QF_INT_LOCK(intKeyLock);
        }
        QF_INT_UNLOCK(intKeyLock);
    }
    /*.....*/
(12) QSTimeCtr QS_onGetTime(void) { /* invoked with interrupts locked */
(13) if ((MCF_PIT0_PCSR & MCF_PIT_PCSR_PIF) == 0) { /* no rollover? */
(14) return QS_tickTime - (QSTimeCtr)MCF_PIT0_PCNT;
    }
    else { /* the rollover occurred, but PIT0_IntrHandler() did not run yet */
(15) return QS_tickTime + PIT_PERIOD
        - (QSTimeCtr)(QSTimeCtr)MCF_PIT0_PCNT;
    }
}
#endif /* Q_SPY */

/*.....*/
__interrupt void PIT0_IntrHandler(void) {

```

```
        MCF_PIT0_PCSR |= MCF_PIT_PCSR_PIF;          /* clear the interrupt source */
(16) #ifdef Q_SPY
        QS_tickTime += PIT_PERIOD;                  /* add the rollover */
    #endif
        QF_tick();                                  /* process all armed time events */
    }
```

**Listing 9 Q-SPY implementation to send data out of the UART2 and DMA3 of the MCF52223 device.**

Listing 9(1) The QS instrumentation is enabled only when the macro Q\_SPY is defined

Listing 9(2) The QS\_tickTime variable is used to hold the 32-bit-wide timestamp at tick. The PIT\_PERIOD constant the nominal number of hardware clock ticks between two subsequent time ticks.

Listing 9(3) This enumeration defines application-specific QS trace record(s), to demonstrate how to use them.

Listing 9(4) You need to define the QS\_onStartup() callback to initialize the QS software tracing.

Listing 9(5) You should adjust the QS buffer size (in bytes) to your particular application

Listing 9(6) You always need to call QS\_initBuf() from QS\_onStartup() to initialize the trace buffer.

Listing 9(7) The UART2 (actually only the transmitter) is initialized.

Listing 9(8) The Back-Door Enable bit must be switched in the SCM\_RAMBAR register to allow DMA access to the software trace buffer in RAM (SCM\_RAMBAR controls the access of peripherals to the RAM. This should not be confused with RAMBAR that controls CPU's access to RAM).

**NOTE:** The SCM\_RAMBAR register always reads 0, so the BDE bit cannot be set in the following expression:

```
MCF_SCM_RAMBAR |= MCF_SCM_RAMBAR_BDE;    /* will NOT work */
```

Listing 9(9) Additionally, DMA won't work without setting the privilege level for DMA.

Listing 9(10) Finally, DMA3 is configured to control the transmitter of UART2.

Listing 9(11) The QS\_flush() callback flushes the QS trace buffer to the host. Typically, the function busy-waits for the transfer to complete. It is only used in the initialization phase for sending the QS dictionary records to the host.

## 5.1 QS Time Stamp Callback QS\_onGetTime()

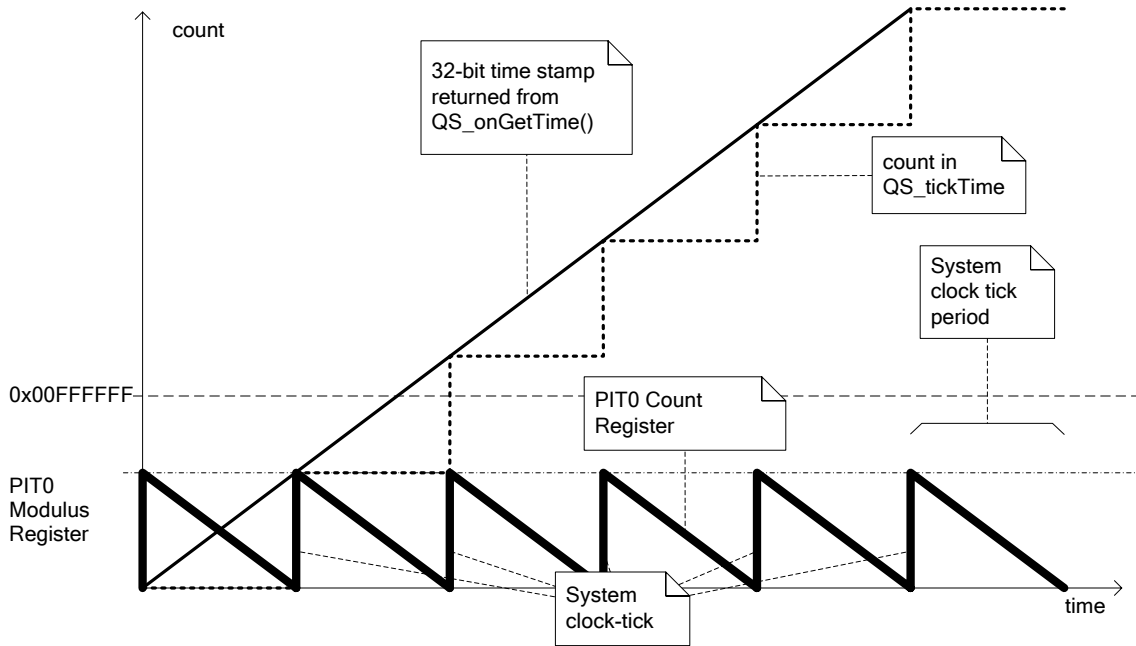
The platform-specific QS port must provide function QS\_onGetTime() (Listing 9(12)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the ColdFire port uses the PITO facility, which is the same timer already used for generation of the system clock-tick interrupt.

**NOTE:** The QS\_onGetTime() callback is always called with interrupts locked.

Figure 4 shows how the PITO Count Register (PITO\_PCNTR) reading is extended to 32 bits. The PITO\_PCNTR counts down from the reload value stored in the PITO Modulus Register (PITO\_PMR). When PITO\_PCNTR reaches 0, the hardware automatically reloads the PITO\_PCNTR counter from

PIT0\_PMR on the subsequent clock tick. Simultaneously, the hardware sets the PIT0\_PCSR flag, which triggers the interrupt and “remembers” that the reload has occurred.

The PIT0\_IntrHandler() ISR keeps updating the “tick count” variable QS\_tickTime by incrementing it each time by PIT\_PERIOD (see Listing 9(16)). The clock-tick ISR also clears the MCF\_PIT0\_PCSR flag.



**Figure 4 Using the PIT0 Count Register to provide 32-bit QS time stamp.**

Listing 9(12-15) shows the implementation of the function QS\_onGetTime(), which combines all this information to produce a monotonic time stamp.

Listing 9(13-14) Most of the time the MCF\_PIT0\_PIF flag in the MCF\_PIT0\_PCSR register is not set, and the time stamp is simply the sum of QS\_tickTime - MCF\_PIT0\_PCNTR). Note that the MCF\_PIT0\_PCNTR register is subtracted from QS\_tickTime to make it to an up-counter rather than down-counter.

Listing 9(15) If the MCF\_PIT0\_PIF flag in the MCF\_PIT0\_PCSR register is set, the PIT0\_PCNTR count has rolled over to zero, but the PIT0\_IntrHandler() ISR has not run yet (because interrupts are still locked). In this case, the MCF\_PIT0\_PCNTR counter misses one update period and must be additionally incremented by PIT\_PERIOD.

Listing 9(16) The PIT0\_IntrHandler() ISR accounts for the PIT0 rollover by incrementing the QS\_tickTime counter. The ISR also explicitly clears the MCF\_PIT0\_PCSR flag.

## 5.2 QS Trace Output in QF\_onIdle()/QK\_onIdle()

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

```

void QF_onIdle(QF_INT_KEY_TYPE key) {          /* interrupts LOCKED, NOTE01 */

    /* toggle the LED-16 on and then off, see NOTE02 */
    MCF_GPIO_CLRTA = ~0x02;
    MCF_GPIO_SETTA = 0x02;

    #ifdef Q_SPY
    QF_INT_UNLOCK(key);

(1)    if ((MCF_DMA3_BCR & 0x00FFFFFF) == 0) {      /* DMA byte counter zero? */
(2)        uint16_t len = 0xFFFF;                /* request as many bytes as possible */
(3)        uint8_t const *block;
(4)        QF_INT_LOCK(key);
(5)        block = QS_getBlock(&len);            /* try to get next block to transmit */
(6)        QF_INT_UNLOCK(key);

(7)        if (block != (uint8_t *)0) {
(8)            MCF_DMA3_DCR = 0x00;                /* clear the DCR to avoid CE error */
            MCF_DMA3_DSR = 0x01;                /* reset the DSR */
            MCF_DMA3_SAR = (uint32_t)block;        /* set the source */
            MCF_DMA3_DAR = (uint32_t)&MCF_UART_UTB(2); /* destination UART2 */
            MCF_DMA3_BCR = MCF_DMA_BCR_BCR(len);    /* byte counter */

            /* external Request, cycle Steal, dest & source = 8 bit
            * dest address NOT incremented source incremented by 1
            */
            MCF_DMA3_DCR = MCF_DMA_DCR_EEXT
                | MCF_DMA_DCR_CS
                | MCF_DMA_DCR_SINC
                | MCF_DMA_DCR_SIZE(01)
                | MCF_DMA_DCR_DSIZ(01);
        }
    }
    else {
        QF_INT_UNLOCK(key);                    /* unlock interrupts */
    }

    #ifdef NDEBUG
    /* Put the CPU and peripherals to the low-power mode.
    * you might need to customize the clock management for your application,
    * see the ColdFire Reference Manual for your particular device.
    */
    __stop(0x2000);                            /* privileged STOP instruction */
    #else
        QF_INT_UNLOCK(key);                    /* unlock the interrupts */
    #endif
}

```

### Listing 10 QS trace output using the UART2 of the MCF52223 MCU.

Listing 10(1) The DMA3 byte counter is tested. When the counter reaches zero, the DMA3 has transferred all the data and can be set up again.

Listing 10(2) The len variable is initialized with the maximum number of bytes the QS\_getBlock() function can deliver.

Listing 10(3) The block variable is the pointer to the contiguous data block returned from QS\_getBlock() function (see "QP Reference Manual" [QP/C 08]).

Listing 10(4) Interrupts are locked to call QS\_getBlock().

Listing 10(5) The function `QS_getBlock()` returns the contiguous data block of up-to `len` bytes. The function also returns the actual number of bytes available in the `len` variable (passed as a pointer).

Listing 10(6) The interrupts are unlocked after the call to `QS_getBlock()`.

Listing 10(7) If the `QS_getBlock()` returned a valid block of data...

Listing 10(8) The DMA3 channel is configured to transfer the block. At this point the DMA handles the transfer autonomously without using any CPU cycles.

## 5.3 Invoking the QSpy Host Application

---

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

```
qspy -cCOM1 -b115200 -S2 -Q2 -C4
```

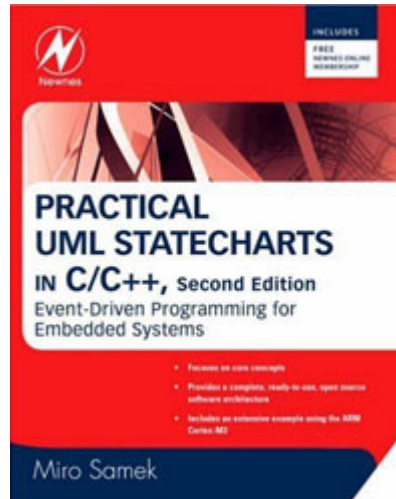
## 6 Related Documents and References

Document	Location
[ColdFire RM 07] "MCF52223 ColdFire Reference Manual", Freescale, 04-2007	<a href="http://www.freescale.com">www.freescale.com</a>
[Samek 08] "Practical UML Statecharts in C/C++, Second Edition: Event Driven Programming for Embedded Systems", Miro Samek, Newnes, 2008	Available from most online book retailers, such as <a href="http://amazon.com">amazon.com</a> . See also: <a href="http://www.state-machine.com/writings/psicc2.htm">http://www.state-machine.com/writings/psicc2.htm</a>
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2008	<a href="http://www.state-machine.com/doxygen/qpc/">http://www.state-machine.com/doxygen/qpc/</a>
[QP/C++ 08] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2008	<a href="http://www.state-machine.com/doxygen/qpcpp/">http://www.state-machine.com/doxygen/qpcpp/</a>
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	<a href="http://www.state-machine.com/doc/-AN_QP_Directory_Structure.pdf">http://www.state-machine.com/doc/-AN_QP_Directory_Structure.pdf</a>
[QL AN-DPP 08] "Application Note: Dining Philosophers Application", Quantum Leaps, LLC, 2008	<a href="http://www.state-machine.com/doc/AN_DPP.pdf">http://www.state-machine.com/doc/AN_DPP.pdf</a> (included in the QDK)

## 7 Contact Information

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

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



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

