Quantum™ LeaPs

innovating embedded systems

QDK™/C++

Philips LPC2000

Document Version 1.04

December 2005

Preliminary Version

Quantum LeaPs™, LLC

www.quantum-leaps.com

www.DataSheet4U.com

1 Introduction

This Quantum Development Kit describes how to use Quantum Platform™ (QP) on the Philips LPC2000 family of MCUs using the IAR Embedded Workbench® for ARM (www.iar.com). This document describes QK port to the mixed ARM/THUMB mode, pure ARM mode as well as pure THUMB mode. The actual hardware/software used is as follows (see also Figure 1):

1. IAR-P213x evaluation board from IAR (the board is based around LPC2138 MCU)
2. The J-link J-TAG pod from Segger (www.segger.com).
3. IAR Embedded Workbench® for ARM version 4.30A, the free KickStart edition
4. QP/C 3.1.xx (QEP, QF, QK)

Figure 1 IAR KickStart Kit including the LPC-P213x Evaluation Board and the J-Link J-TAG pod from Segger, GmbH.

2 Getting Started

2.1 Directories and Files

The code of the port is organized according to the "Application Note: QP Directory Structure" (http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf). Specifically, for this port the files are placed in the following directories:

```
<qpcpp_3>/                - QP/C++-root directory for Quantum Platform (QP/C++) v3.1.xx
  |
  +-include/              - QP/C++ public include files
  | +-qassert.h           - Quantum Assertions platform-independent public include
  | +-qep.h               - QEP/C++ platform-independent public include
  | +-qf.h                - QF/C++ platform-independent public include
  | +-qk.h                - QK/C++ platform-independent public include
  | +-qequeue.h           - Quantum Event Queue platform-independent public include
  | +-qmpool.h            - Quantum Memory Pool platform-independent public include
  |
  +-ports/                - QP/C++ ports
  | +-arm7-lpc2000/       - ARM7 CPU, Philips LPC2000 family
  | | +-qk/               - QK (Quantum Kernel) ports
  | | | +-iar/            - IAR ARM compiler
  | | | | +-arm/          - ARM mode
  | | | | | +-dbg/        - Debug build
  | | | | | | +-libqep.r79 - QEP library
  | | | | | | +-libqf.r79  - QF library
  | | | | | | +-libqk.r79  - QK library
  | | | | | +-rel/        - Release build
  | | | | | +-spy/        - Spy build (Quantum Spy instrumented)
  | | | | | +-qep_port.h  - QEP/C++ platform-dependent public include
  | | | | | +-qf_port.h   - QF/C++ platform-dependent public include
  | | | | | +-qk_port.h   - QK/C++ platform-dependent public include
  | | | | +-mixed/        - Mixed ARM/Thumb mode
  | | | | | +-dbg/        - Debug build
  | | | | | | +-libqep.r79 - QEP library
  | | | | | | +-libqf.r79  - QF library
  | | | | | | +-libqk.r79  - QK library
  | | | | | +-rel/        - Release build
  | | | | | +-spy/        - Spy build (Quantum Spy instrumented)
  | | | | | +-qep_port.h  - QEP/C++ platform-dependent public include
  | | | | | +-qf_port.h   - QF/C++ platform-dependent public include
  | | | | | +-qk_port.h   - QK/C++ platform-dependent public include
  | | | | +-thumb/        - THUMB mode
  | | | | | +-dbg/        - Debug build
  | | | | | | +-libqep.r79 - QEP library
  | | | | | | +-libqf.r79  - QF library
  | | | | | | +-libqk.r79  - QK library
  | | | | | +-rel/        - Release build
  | | | | | +-spy/        - Spy build (Quantum Spy instrumented)
  | | | | | +-qep_port.h  - QEP/C++ platform-dependent public include
  | | | | | +-qf_port.h   - QF/C++ platform-dependent public include
  | | | | | +-qk_port.h   - QK/C++ platform-dependent public include
  | |
  +-qep/
  | +-arm7-lpc2000/       - ARM7 CPU, Philips LPC2000 family
  | | +-qk/               - QK (Quantum Kernel) ports
  | | | +-iar/            - IAR ARM compiler
  | | | | +-arm/          - ARM mode
  | | | | | +-Makefile    - Makefile to build this version of the QEP/C++ library
  | | | | +-mixed/        - Mixed ARM/Thumb mode
  | | | | | +-Makefile    - Makefile to build this version of the QEP/C++ library
```

```
| | | | +-thumb/          - THUMB mode
| | | | | +-Makefile      - Makefile to build this version of the QEP/C++ library
| +-source/               - QEP/C++ platform-independent source files
|
+-qf/
| +-arm7-lpc2000/         - ARM7 CPU, Philips LPC2000 family
| | +-qk/                 - QK (Quantum Kernel) ports
| | | +-iar/              - IAR ARM compiler
| | | | +-arm/            - ARM mode
| | | | | +-Makefile      - Makefile to build this version of the QF/C++ library
| | | | +-mixed/          - Mixed ARM/Thumb mode
| | | | | +-Makefile      - Makefile to build this version of the QF/C++ library
| | | | +-thumb/          - THUMB mode
| | | | | +-Makefile      - Makefile to build this version of the QF/C++ library
| +-source/               - QF/C++ platform-independent source files
|
+-qk/
| +-arm7-lpc2000/         - ARM7 CPU, Philips LPC2000 family
| | +-qk/                 - QK (Quantum Kernel) ports
| | | +-iar/              - IAR ARM compiler
| | | | +-arm/            - ARM mode
| | | | | +-Makefile      - Makefile to build this version of the QK/C++ library
| | | | | +-qk_port.cpp   - platform-dependent code for this version of QF/C++
| | | | +-mixed/          - Mixed ARM/Thumb mode
| | | | | +-Makefile      - Makefile to build this version of the QK/C++ library
| | | | | +-qk_port.cpp   - platform-dependent code for this version of QK/C++
| | | | +-thumb/          - THUMB mode
| | | | | +-Makefile      - Makefile to build this version of the QK/C++ library
| | | | | +-qk_vect.s79   - assembly module for QK interrupt vectors
| | | | | +-qk_port.cpp   - platform-dependent code for this version of QK/C++
| +-source/               - QK/C++ platform-independent source files
|
+-examples/               - QP/C++ examples
| +-arm7-lpc2000/         - ARM7 CPU, Philips LPC2000 family
| | +-qk/                 - QK (Quantum Kernel) ports
| | | +-iar/              - IAR ARM compiler
| | | | +-arm/            - ARM mode
| | | | | +-qdpp-p213x    - QDPP example for the LPC-P213x evaluation board
| | | | | | +-dbg/        - Debug build (runs from RAM)
| | | | | | | +-qdpp.d79  - executable image
| | | | | | +-rel/        - Release build (runs form ROM)
| | | | | | +-drivers/    - Device drivers for the LPC2138
| | | | | | +-lpc2138_ram.xcl   - IAR ARM linker command file for the LPC2138-RAM buld
| | | | | | +-lpc2138_flash.xcl- IAR ARM linker command file for the LPC2138-Flash buld
| | | | | | +-lpc2138_ram.mac   - IAR C-Spy debugger macro to remap vectors to RAM mode
| | | | | | +-Makefile    - make file to build the QDPP example
| | | | | | +-bsp.h       - Board Support Package include file
| | | | | | +-bsp.cpp     - Board Support Package implementation
| | | | | | +-cstartup.s79 - Startup code in assembly
| | | | | | +-qdpp.h
| | | | | | +-main.cpp
| | | | | | +-philo.cpp
| | | | | | +-table.cpp
| | | | | | +-qdpp_dbg.ewp - IAR project file to debug the QDPP example
| | | | | | +-qdpp_dbg.eww - IAR workspace file to debug the QDPP example
| | | | | |
| | | | +-mixed/          - Mixed ARM/Thumb mode
| | | | | +-qdpp-p213x    - QDPP example for the LPC-P213x evaluation board
| | | | | | +-dbg/        - Debug build (runs from RAM)
| | | | | | | +-qdpp.d79  - executable image
| | | | | | +-rel/        - Release build (runs form ROM)
| | | | | | +-drivers/    - Device drivers for the LPC2138
| | | | | | +-lpc2138_ram.xcl   - IAR ARM linker command file for the LPC2138-RAM buld
| | | | | | +-lpc2138_flash.xcl- IAR ARM linker command file for the LPC2138-Flash buld
```

```
  | | | | | | +-lpc2138_ram.mac  - IAR C-Spy debugger macro to remap vectors to RAM mode
  | | | | | | +-Makefile   - make file to build the QDPP example
  | | | | | | +-bsp.h      - Board Support Package include file
  | | | | | | +-bsp.cpp    - Board Support Package implementation
  | | | | | | +-cstartup.s79 - Startup code in assembly
  | | | | | | +-qdpp.h
  | | | | | | +-main.cpp
  | | | | | | +-philo.cpp
  | | | | | | +-table.cpp
  | | | | | | +-qdpp_dbg.ewp - IAR project file to debug the QDPP example
  | | | | | | +-qdpp_dbg.eww - IAR workspace file to debug the QDPP example
  | | | | | |
  | | | | +-thumb/         - Mixed ARM/Thumb mode
  | | | | | +-qdpp-p213x   - QDPP example for the LPC-P213x evaluation board
  | | | | | | +-dbg/       - Debug build (runs from RAM)
  | | | | | | | +-qdpp.d79 - executable image
  | | | | | | +-rel/       - Release build (runs form ROM)
  | | | | | | +-drivers/   - Device drivers for the LPC2138
  | | | | | | +-lpc2138_ram.xcl  - IAR ARM linker command file for the LPC2138-RAM buld
  | | | | | | +-lpc2138_flash.xcl- IAR ARM linker command file for the LPC2138-Flash buld
  | | | | | | +-lpc2138_ram.mac  - IAR C-Spy debugger macro to remap vectors to RAM mode
  | | | | | | +-Makefile   - make file to build the QDPP example
  | | | | | | +-bsp.h      - Board Support Package include file
  | | | | | | +-bsp.cpp    - Board Support Package implementation
  | | | | | | +-cstartup.s79 - Startup code in assembly
  | | | | | | +-qdpp.h
  | | | | | | +-main.cpp
  | | | | | | +-philo.cpp
  | | | | | | +-table.cpp
  | | | | | | +-qdpp_dbg.ewp - IAR project file to debug the QDPP example
  | | | | | | +-qdpp_dbg.eww - IAR workspace file to debug the QDPP example
  | | | | | |
  | | . . .
  | . . .
```

Listing 1 Directories and files of the QP port to LPC2000

2.2 Building the QP Libraries

All QP/C++ components are deployed as static class libraries that you link to your application. The pre-built libraries for QEP, QF, and QK are provided inside the <qpcpp_3>/ports/ directory (see Listing 1). This section describes steps you need to take to rebuild the libraries yourself.

All QP components and ports use the standard Quantum Leaps build procedure based on the GNU-compatible make utility. The code distribution contains all the Makefiles to automate the build.

NOTE: To achieve commonality among different platforms, Quantum Leaps software does not use the vendor-specific IDEs, such as the IAR Workbench, for building the QP libraries and applications. Instead, QP supports *command-line* build process conforming to the GNU make standard, which in turn conforms to Section 6.2 of IEEE Standard 1003.2-1992 (POSIX.2).

The GNU-compatible make utility mingw32-make.exe for Windows can be *freely* downloaded from http://www.mingw.org/download.shtml. After installing the utility, you should add the GNU-make directory to your PATH.

For example, to build the QF/C++ library for the ARM7-LPC2000, with the IAR ARM compiler, QK kernel, mixed ARM/THUMB mode, you open a console window on a Windows PC, change directory

to <qpcpp_3>/qf/arm7-lpc2000/qk/iar/mixed/, and invoke the GNU-make utility by typing at the command prompt the following command:

```
mingw32-make
```

The make utility processes the make file Makefile in the current directory to build the QF/C++ library. This Makefile assumes that the ARM toolset has been installed in the directory C:/tools/IAR/ARM_KS_4.30A. You need to adjust the symbol IAR_ARM at the top of the Makefile if you've installed the IAR ARM compiler into a different directory. The make process should produce the QEP library in the location: <qpcpp_3>/ports/arm7-lpc2000/qk/iar/mixed/dbg/libqf.r79.

Identical procedure should be applied to build the QEP and QK components as well as other modes of the ARM processor (ARM and THUMB).

2.3 Building the Examples

This QP port comes with an example application, which is the standard Dining Philosopher Problem implemented with active objects (see "Practical Statecharts in C/C++", Chapter 7).

The Figure 1 shows the example running on the LPC-P213x board. The second line of the LCD shows a state of the application where Philosopher[0] is thinking ('t'), Philosopher[1] is eating ('e'), Philosopher[2] is hungry ('h'), Philosopher[3] is thinking ('t'), and Philosopher[4] is eating ('e').

Building the various versions of the example follows the standard Quantum Leaps build procedure based on the GNU-compatible make utility. The code distribution contains all the Makefiles to automate the build.

For example, to build the QDPP example for the ARM7-LPC2000, with the IAR ARM compiler, QK kernel, mixed ARM/THUMB mode, P213x board, you open a console window on a Windows PC, change directory to <qpcpp_3>/examples/arm7-lpc2000/qk/iar/mixed/qdpp-p213x/, and invoke the GNU-make utility by typing at the command prompt the following command:

```
mingw32-make
```

The make utility processes the make file Makefile in the current directory to build the QDPP image. This Makefile assumes that the ARM toolset has been installed in the directory C:/tools/IAR/ARM_KS_4.30A. You need to adjust the symbol IAR_ARM at the top of the Makefile if you've installed the IAR ARM compiler into a different directory. The make process should produce the QDPP image in the location: <qpcpp_3>/examples/arm7-lpc2000/qk/iar/mixed/qdpp-p213x/dbg/-qdpp.d79.

Identical procedure should be applied to build the QDPP example for the ARM mode or the THUMB mode.

2.4 Running the Examples

The build process described in the previous section creates QDPP application image linked to RAM to be downloaded and executed in the target by the IAR C-Spy debugger. The code distribution provides the IAR workspace qepp-dbg.eww to run the application (see Listing 1). For example, to run the mixed ARM/THUMB version of the application, use your Windows Explorer to go to the directory <qpcpp_3>/examples/arm7-lpc2000/qk/iar/mixed/qdpp-p213x, and double-click on the workspace qepp-dbg.eww. If you installed the IAR Embedded Workbench for ARM (EWARM), this should launch the tool and open the project.

At this point, you should make sure that your J-Link pod is installed according to the "Getting Started" note in the IAR KickStart Kit distribution. The LPC-P213x board should be powered up and connected to the J-Link with the 20-pin ribbon cable (see Figure 1). 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 short-cut. You should see display similar to the Figure 1.

Identical procedure should be applied to run the QDPP example for the ARM mode or the THUMB mode.

3 About the Mixed ARM/THUMB Port

The mixed ARM/THUM port is perhaps closest to the "beaten path" ARM implementations. In particular, it is based on the following Philips Application Notes, which are available online:

1. AN10391 "Nesting of Interrupts in the LPC2000" [Philips 03]
2. AN10254 "Handling Interrupts Using IRQ and FIQ" [Philips 05a]
3. Application Note "Interrupt Management: Auto-vectoring and Prioritization", [Atmel 98a]

Generally, the port runs the task-level code in the **ARM System Mode** (mode bits 0x1F), which is the default mode in which the IAR startup code calls `main()`.

3.1 Compiler and Linker Options Used

The most important IAR compiler options used are as follows:

```
--cpu_mode thumb
--interwork
-D_DLIB_CONFIG_FILE=$(IAR_ARM)\arm\LIB\dl4tptinl8n.h
```

In particular, the option `--internetwork` specifies ARM/THUMB interworking code generation, and the option `G -D_DLIB_CONFIG_FILE=$(IAR_ARM)\arm\LIB\dl4tptinl8n.h` specifies the C-runtime library to be THUMB, interwork, normal library mode (see the IAR Compiler Reference).

The most important linker option is as follows:

```
-rt $(IAR_ARM)\arm\lib\dl4tptinl8n.r79
```

This linker option once again specifies the runtime to be the THUMB, interwork, normal library mode

3.2 The QK Port Header File

The QK header file for the LPC200, Mixed ARM/THUMB can be found in `<qpcpp_3>/ports/arm7-lpc2000/qk/iar/mixed/qk_port.h`. This file specifies the interrupt enabling/disabling policy (QK critical section) as well as the interrupt entry and exit code.

3.2.1 The QK Critical Section

This QK port uses the standard IAR intrinsic functions for disabling and enabling interrupts. These functions can be called from both ARM and THUMB mode. The IAR functions disable both IRQ and FIQ **at the ARM core level** (refer to the "ARM® IAR C/C++ Compiler Reference Guide" [IAR 05a] for more information).

```
// QK_INT_KEY_TYPE not defined
#define QK_INT_LOCK(key_)           __disable_interrupt()
#define QK_INT_UNLOCK(key_)         __enable_interrupt()
```

The QK_INT_KEY_TYPE is not defined, which means that the simplest interrupt locking scheme is applied, in which exit from a critical section always enables interrupts, regardless if the interrupts were enabled or disabled upon entry to the critical section.

This interrupt disabling policy means that the QK critical sections cannot nest. However the policy still allows **to prioritize nested interrupts** (IRQs), because the interrupt prioritization is handled in hardware by the Vectored Interrupt Controller (VIC) that operates independently to the ARM core. The operation of the VIC is explained in the "LPC213x User Manual" [Philips 05b]. The interrupt prioritization occurs between reading from the VICVectAddr register and writing to the VICVectAddr register at 0xFFFFF030. If a higher-priority interrupt occurs during this time window, the VIC will assert the IRQ line of the ARM core. The higher-priority interrupt will be recognized if the core has interrupts (IRQ) enabled.

As discussed in Section "The QK Interrupt Entry and Exit", the ARM core interrupts are typically re-enabled during the interrupt processing, so the described interrupt prioritization will occur.

NOTE: It is perhaps interesting to observe that the VIC priority encoder implements in hardware exactly the same algorithm as the QK scheduler implements in software. In other words, the prioritization works the same way for tasks (implemented in software) and for ISRs (implemented in hardware).

3.2.2 VIC Auto-vectoring

This port can be easily made to work with or without the auto-vectoring feature of the Vectored Interrupt Controller (VIC) of the LPC2000 MCUs (see the "LPC213x User Manual", [Philips 05b]).

Auto-vectoring occurs when the following LDR instruction is located at the address 0x18 for the IRQ:

```
    ORG 0x18
    LDR pc,[pc,#-0xFF0]
```

When an IRQ occurs, the ARM core forces the PC to 0x18 and executes the LDR pc,[pc,#-0xFF0] instruction. When the instruction at address 0x18 is executed, the effective address is:

```
    0x00000020 - 0x00000FF0 = 0xFFFFF030
```

(0x00000020 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 VICVectAddr register located at 0xFFFFF030. The read cycle causes the VICVectAddr register to return the address corresponding to the currently active interrupt. Thus, the single LDR pc,[pc,#-0xFF0] instruction has the effect of directly jumping to the correct interrupt service routine—auto-vectoring.

On the other hand, you can avoid auto-vectoring by placing a different LDR instruction at the address 0x18, for example LDR pc,[px,#24], will always load the same address located at 0x20+24 = 0x38.

The following discussion assumes that the auto-vectoring feature is used, that is the LDR pc,[pc,#-0xFF0] is located at the address 0x18 in the startup code. Please refer to the cstartup.s79 code located in the examples directory.

3.2.3 The QK Interrupt Entry and Exit

This QK, as any preemptive, deterministic, real-time kernel, requires specific entry and exit from the interrupts. The general interrupt handling sequence of QK is summarized by the following pseudo-code:

```
void interrupt qkISR(void) {    // an ISR is entered with interrupts disabled
    save processor registers clobbered by C-function calls
    clear the level-sensitive interrupt source
    uint8_t pin = QK_currPrio_; // save the current QK priority in a stack variable
    QK_currPrio_ = 0xFF;        // set the current QK priority above any task
    Enable interrupts

    Perform the work of the ISR

    Disable interrupts
    Signal End-Of-Interrupt to the interrupt controller
    QK_currPrio_ = pin;         // restore the initial QK priority
    QK_schedule_();             // invoke the QK scheduler
    Restore the processor registers
    Execute a return from interrupt instruction
}
```

Listing 2 QK interrupt handling sequence pseudo-code

Please note that QK inherently requires a capability of nesting interrupts, because interrupts are explicitly enabled in the ISR, and also implicitly in the QK scheduler `QK_schedule_()`.

This port uses the technique of handling nested interrupts in ARM described in the Philips Application Note "Nesting of Interrupts in the LPC2000".

The technique relies on the C compiler's ability to generate code for the ARM interrupts in C. In the IAR compiler, you define an IRQ handler in the following way:

```
__irq __arm void tickIRQ(void) {
   . . .
}
```

The use of the extended keyword `__irq` informs the IAR compiler to generate an IRQ-mode prolog and epilog of the interrupt function. However, the interrupt does not allow nesting, because the register LR_IRQ is not saved and the processor mode is not changed away from IRQ.

NOTE: The IAR compiler provides an extended keyword `__nested` to allow nesting of interrupts. However, the `__nested` interrupt code has some known problems in version 4.30A, and also for other reasons the `__nested` keyword is **not** used in this QK port.

Instead, as described in the Application Note "Nesting of Interrupts in the LPC2000", the interrupt entry and exit is coded as macros that use inline assembly. The macros are defined in the `<qpcpp_3>/ports/arm7-lpc2000/qk/iar/mixed/qk_port.h` header file as follows:

```
1: QK interrupt entry and exit
2: #define QK_IRQ_ENTRY(pin_) do { \
3:     (pin_) = QK_currPrio_; \
4:     QK_currPrio_ = 0xFF; \
5:     asm("MRS   lr,spsr"); \
6:     asm("STMFD sp!,{lr}");  \
```

```
 7:     asm("MSR   cpsr_c,#0x1F"); \
 8:     asm("STMFD sp!,{lr}"); \
 9: } while (0)
10:
11: #define QK_IRQ_EXIT(pin_) do { \
12:     asm("MSR   cpsr_c,#0xDF"); \
13:     asm("MOV   lr,#0"); \
14:     asm("STR   lr,[lr,#-0xFD0]"); \
15:     QK_currPrio_ = (pin_); \
16:     QK_schedule_(); \
17:     asm("LDMFD sp!,{lr}"); \
18:     asm("MSR   cpsr_c,#0xD2"); \
19:     asm("LDMFD sp!,{lr}"); \
20:     asm("MSR   spsr_cxsf,lr"); \
21: } while (0)
```

Listing 3 QK interrupt entry and exit macros defined in the qk_port.h header file.

As you can see, the macros quite faithfully implement the general sequence outlined in the QK interrupt handling pseudo-code from Listing 2.

Some details of the macros from Listing 3 are as follows. First, note that the `do {…} while (0)` loops around the macros are the standard way of syntactically-correct grouping of instructions. In line 7, you see the quick way of changing the ARM core mode to SYSTEM and enabling interrupts at the same time through an immediate-load to the `CPSR_c`. This technique is directly copied from the aforementioned Philips Application Note. (In fact, the whole entry sequence in lines 5-8 is identical to that described in the Application Note.)

Similarly, lines 17-20 of Listing 3 are identical as the exit sequence described in the Philips Application Note. In the preceding lines 12, you see quick mode change back to the IRQ with disabling interrupts at the same time (both I and F bits). In lines 13-14, you see writing the End-Of-Interrupt (EOI) sequence into the Vectored Interrupt Controller (VIC) VICVectAddr register at 0xFFFFF030, which is 0x0 – 0xFD0.

The intended use of these interrupt entry and exit macros is illustrated in the QF tick ISR, as follows:

```
1: __irq __arm void tickIRQ(void) {
2:     T1IR = 0x1;                                  // clear the timer interrupt
3:     uint8_t pin;                        // initial priority upon entry to the ISR
4:     QK_IRQ_ENTRY(pin, TICK_IRQ_PRIO);    // enter the nested portion of the IRQ
5:
6:     QF::tick();
7:
8:     QK_IRQ_EXIT(pin);                    // exit the nested portion of the IRQ
9: }
```

Listing 4 The QF tick ISR demonstrating the use of the QK_IRQ_ENTRY() and QK_IRQ_EXIT() macros.

3.3 Startup Code and Stack Initialization

The startup code must initialize at least the User/System stack, the IRQ stack, and optionally the FIQ stack, if the FIQ is used.

The User/System stack is the regular C stack used by the `main()` function and all functions called from main. Also, the User/System stack is used to nest preemptions as usual in the QK (see [QL 05c]).

However, the IRQ stack is also used for nesting interrupts. This is a departure for the QK, which typically uses just a single stack for nesting all tasks and interrupts. (Using a single stack in the ARM architecture is possible, but requires some assembly programming and will not be discussed in this port.)

The IRQ stack is used as well, because of the code generated by the IAR compiler when the `__irq` keyword is used (see Section 3.2.3). The following is a disassembled code emitted by the compiler for the `tickIRQ()` shown in Listing 4:

```
__irq __arm void tickIRQ(void) {
  400001D0  E24EE004  SUB         LR, LR, #0x4
  400001D4  E92D501F  STMDB       SP!, {R0,R1,R2,R3,R4,R12,LR}
    T1IR = 0x1;                                 // clear the timer interrupt
  400001D8  E3A004E0  MOV         R0, #0xE0000000
  400001DC  E3800C80  ORR         R0, R0, #0x8000
  400001E0  E3A01001  MOV         R1, #0x1
  400001E4  E5801000  STR         R1, [R0, #+0]
    QK_IRQ_ENTRY(pin);                           // enter the nested portion of the IRQ
  400001E8  E59F03B0  LDR         R0, [PC, #+944]  ; [0x400005A0] =QK_currPrio_
  400001EC  E5D00000  LDRB        R0, [R0, #+0]
  400001F0  E1A04000  MOV         R4, R0
  400001F4  E59F03A4  LDR         R0, [PC, #+932]  ; [0x400005A0] =QK_currPrio_
  400001F8  E3A010FF  MOV         R1, #0xFF
  400001FC  E5C01000  STRB        R1, [R0, #+0]
  40000200  E14FE000  MRS         LR, SPSR
  40000204  E92D4000  STMDB       SP!, {LR}
  40000208  E321F01F  MSR         CPSR_c, #31
  4000020C  E92D4000  STMDB       SP!, {LR}
    QF::tick();
  40000210  EB0009AE  BL          tick            ; 0x400028D0
    QK_IRQ_EXIT(pin);                            // exit the nested portion of the IRQ
  40000214  E321F0DF  MSR         CPSR_c, #223
  40000218  E3A0E000  MOV         LR, #0x0
  4000021C  E50EEFD0  STR         LR, [LR, #-4048]
  40000220  E59F0378  LDR         R0, [PC, #+888]  ; [0x400005A0] =QK_currPrio_
  40000224  E5C04000  STRB        R4, [R0, #+0]
  40000228  EB000725  BL          QK_schedule_            ; 0x40001EC4
  4000022C  E8BD4000  LDMIA       SP!, {LR}
  40000230  E321F0D2  MSR         CPSR_c, #210
  40000234  E8BD4000  LDMIA       SP!, {LR}
  40000238  E16FF00E  MSR         SPSR_cxsf, LR
}
  4000023C  E8FD901F  LDMIA       SP!, {R0,R1,R2,R3,R4,R12,PC}^
```

Listing 5 Disassembled code of tickIRQ() from Listing 4. The highlighted code is executed in the IRQ mode.

As shown in Listing 5, the processor mode is not changed until the instruction `MSR CPSR_c,#31` at address 40000208. Before that instruction, the following 8 registers are pushed onto the IRQ stack: R0, R1, R2, R3, R4, R12, LR, and SPSR. Consequently, the IRQ stack must be sized for 8 registers (32-bytes) for each anticipated level of preemption. In QK, theoretically the worst-case nesting could be QF_MAX_ACTIVE plus the nesting of interrupts on top of interrupts (63*32bytes + interrupt_nesting*32 = 2KB + …).

4 About the Pure ARM Port

The pure ARM port is allows for some optimizations with respect to the mixed ARM/THUMB port. These optimizations include:

1. faster inlined interrupt enabling/disabling
2. no ARM/THUMB interworking

The port is provided in the `arm-lpc2000/qk/iar/arm/` branch of the ports and examples directories (see Section 2.1). Also, just as the mixed ARM/THUMB port, the pure ARM port runs the task-level code in the **ARM System Mode** (mode bits 0x1F), which is the default mode in which the IAR startup code calls `main()`.

The pure ARM port offers some performance advantage over the mixed ARM/THUMB port when the code is executed from a fast 32-bit wide memory, such as the on-chip RAM of the LPC2000. However, when executed from slower memories, or memories only 16-bit wide, the pure ARM port would actually perform slower than the mixed ARM/THUMB port, where most of the code is compiled to THUMB. Also, the code density is lower (and thus the code size is larger) in the ARM port than it is in the mixed ARM/THUMB port.

4.1 Compiler and Linker Options Used

The most important IAR compiler options used are as follows:

```
--cpu_mode arm
-D_DLIB_CONFIG_FILE=$(IAR_ARM)\arm\LIB\dl4tpannl8n.h
```

In particular, the option `--interwork` is **not** specified, option `-D_DLIB_CONFIG_FILE= $(IAR_ARM)\arm\LIB\dl4tpannl8n.h` specifies the C-runtime library to be ARM, no-interwork, normal library mode (see the IAR Compiler Reference).

The most important linker option is as follows:

```
-rt $(IAR_ARM)\arm\lib\dl4tpannl8n.r79
```

This linker option once again specifies the runtime to be the ARM, no-interwork, normal library mode

4.2 The QK Port Header File

The QK header file for the LPC200, ARM mode can be found in `<qpcpp_3>/ports/arm7-lpc2000/qk/iar/arm/qk_port.h`. This file specifies the interrupt enabling/disabling policy (QK critical section) as well as the interrupt entry and exit code.

4.2.1 The QK Critical Section

This QK port uses the most optimal, single instruction to disable and enable interrupts (the load-immediate MSR to the CPSR_c). This instruction is only available in the pure ARM mode.

```
// QK_INT_KEY_TYPE not defined
#define QK_INT_LOCK(key_)           asm("MSR cpsr_c,#0xDF")
#define QK_INT_UNLOCK(key_)         asm("MSR cpsr_c,#0x1F")
```

The QK_INT_KEY_TYPE is not defined, which means that the simplest interrupt locking scheme is applied, in which exit from a critical section always enables interrupts, regardless if the interrupts were enabled or disabled upon entry to the critical section. This policy is adequate in the presence of the Vectored Interrupt Controller (VIC) hardware, as described in Section 3.2.1.

5 About the Pure THUMB Port

The pure THUMB port requires the most advanced techniques and represents perhaps the most radical departure from the "standard" preemptive multitasking implementations for the ARM architecture. Unlike the other two ports included in this QDK, the pure THUMB port uses only *one* stack (the System stack) for nesting both the tasks and interrupts and does not use the IRQ stack at all.

The pure THUMB port offers the following advantages with respect to the other ports:

1. only one stack with minimal context-switch stack frame
2. the best possible code density
3. no ARM/THUMB interworking overhead
4. more customizable interrupt disabling policy

The port is provided in the `arm-lpc2000/qk/iar/thumb/` branch of the ports and examples directories (see Section 2.1). Also, just as the mixed ports, the pure THUMB port runs the task-level code in the **ARM System Mode** (mode bits 0x1F), which is the default mode in which the IAR startup code calls `main()`.

The pure THUMB port should offer the best performance for executing code from slower Flash memory due to the best code density. On Flash-based MCUs, such as the LPC2000 family, the Memory Acceleration Module (MAM) should be used to further improve the code execution speed.

5.1 Compiler and Linker Options Used

The most important IAR compiler options used are as follows:

```
--cpu_mode thumb
-D_DLIB_CONFIG_FILE=$(IAR_ARM)\arm\LIB\dl4tptnnl8n.h
```

In particular, the option `--interwork` is **not** specified, option `-D_DLIB_CONFIG_FILE= $(IAR_ARM)\arm\LIB\dl4tptnnl8n.h` specifies the C-runtime library to be THUMB, **no-interwork**, normal library mode (see the "IAR Compiler Reference" [IAR 05a]).

The most important linker option is as follows:

```
-rt $(IAR_ARM)\arm\lib\dl4tptnnl8n.r79
```

This linker option once again specifies the runtime to be the THUMB, **no-interwork**, normal library mode

5.2 The QK Port Header File

The QK header file for the LPC200, THUMB mode can be found in `<qpcpp_3>/ports/arm7-lpc2000/qk/iar/thumb/qk_port.h`. This file specifies the interrupt enabling/disabling policy (QK critical section) as well as the interrupt entry and exit code.

5.2.1 The QK Critical Section

This QK port uses the customized (defined in assembly) functions for disabling and enabling interrupts. These functions have been designed to be called from the THUMB mode. The IAR functions disable both IRQ and FIQ **at the ARM core level** (refer to the "ARM® IAR C/C++ Compiler Reference Guide" [IAR 05a] for more information).

```
            // QK critical section must be the same as the QF critical section
// QK_INT_KEY_TYPE not defined
#define QK_INT_LOCK(key_)           QK_int_lock()
#define QK_INT_UNLOCK(key_)         QK_int_unlock()

extern "C" {
void QK_int_lock(void);
void QK_int_unlock(void);
}
```

The QK_INT_KEY_TYPE is not defined, which means that the simplest interrupt locking scheme is applied, in which exit from a critical section always enables interrupts, regardless if the interrupts were enabled or disabled upon entry to the critical section. This policy is adequate in the presence of the Vectored Interrupt Controller (VIC) hardware, as described in Section 3.2.1.

```
 1:         MODULE    ?INT
 2:
 3:         RSEG    CODE:CODE:NOROOT(2)
 4:         PUBLIC  QK_int_lock,QK_int_unlock
 5:
 6:         ALIGNROM 2                  ; align at 2^2 boundary
 7:         CODE16                      ; the function is called from THUMB
 8: QK_int_lock:
 9:         ADR     r0,QK_int_lock_ARM
10:         BX      r0                  ; change mode to ARM
11:         CODE32                      ; now we are in ARM
12: QK_int_lock_ARM:
13:         MSR     cpsr_c,#ARM_SYS_MODE | ARM_INT_BITS  ; SYS mode, lock int.
14:         BX      lr                  ; return changing mode back to THUMB
15:
16:         ALIGNROM 2                  ; align at 2^2 boundary
17:         CODE16                      ; the function is called from THUMB
18: QK_int_unlock:
19:         ADR     r0,QK_int_unlock_ARM
20:         BX      r0                  ; change mode to ARM
21:         CODE32                      ; now we are in ARM
22: QK_int_unlock_ARM:
23:         MSR     cpsr_c,#ARM_SYS_MODE ; SYS mode, unlock int.
24:         BX      lr                  ; return changing mode back to THUMB
25:
26:         LTORG
27:
28:         ENDMOD
```

Listing 6 Interrupt locking/unlocking for the pure THUMB port (defined in the module <qpcpp_3>/qk/arm7-lpc2000/iar/thumb/qk_vect.s79)

Listing 6 shows the assembly implementation of the C-callable functions QK_int_lock() and QK_int_unlock(). Both functions are entered in the THUMB mode (see the CODE16 directive) but must switch to the ARM mode to perform the MSR operations on the ARM status register.

The functions are slightly more efficient than the intrinsic IAR compiler functions `__disable_interrupt()` and `__enable_interrupt()`, because they assume that the mode of operation is always System, and can use the load-immediate addressing mode.

The `ARM_INT_BITS` constant is currently defined as:

```
#define ARM_INT_BITS  (ARM_I_BIT | ARM_F_BIT)
```

which means that both FIQ and IRQ are disabled.

5.2.1.1 Option for using the FIQ as a Nonmaskable Interrupt (NMI)

However, since you now have full control over the interrupt disabling policy, you could choose to disable only the IRQ bit, and thus leave the FIQ always enabled.

```
#define ARM_INT_BITS  ARM_I_BIT
```

This would allow achieving extremely short interrupt latency for servicing the FIQ, but would preclude the FIQ from using any kernel (or framework) services because the FIQ will not be disabled to access critical sections of code. Effectively, this policy would mean that the FIQ would become a **Nonmaskable Interrupt** (NMI). You can still pass parameters to and from the NMI, but the parameters must be read and written **atomically**. For the ARM architecture you could use 8-bit, 16-bit, and 32-bit variables. Conceivably you could even use larger structures, but then you must ensure that they are always accessed atomically via the LDM/STM instructions.

The FIQ used as an NMI would be completely separate from the QK and you need to arrange for the FIQ stack if you use one. To achieve the fastest possible performance, you should probably code the FIQ in assembly and make the maximum use of the 8 banked registers available in this mode.

5.3 The QK Interrupt Handling

5.3.1 No Autovectoring

The THUMB mode handles interrupts differently than the other ports. The auto-vectoring is not used (see Section 3.2.2). Instead, the instruction `ldr pc,[pc,#24]` is placed at the address 0x18 and the address of the IRQ handler is placed at 0x38:

```
        org     0x18
        ldr     pc,[pc,#24]        ; load effective address 0x20+24 = 0x38
       . . .
        org     0x38
                dc32    irq_handler
```

5.3.2 The IRQ Handler in Assembly

To avoid using the IRQ stack the ISR must be coded in assembly.

```
 1:         MODULE    ?IRQ
 2:
 3:         RSEG    CODE:CODE:NOROOT(2)
 4:         PUBLIC  irq_handler
 5:         EXTERN  QK_currPrio_, QK_schedule_
 6:         CODE32
 7:
 8: irq_handler:
 9:         ; IRQ entry {{{
10:         MOV     r13,r0              ; save r0 in r13_IRQ
11:         SUB     r0,lr,#4            ; put return address in r0
12:
13:         MSR     cpsr_c,#ARM_SYS_MODE | ARM_INT_BITS
14:         STMFD   sp,{r0}             ; save return address (PC) on user stack
15:         SUB     r0,sp,#4            ; put adjusted sp_SYS in r0
16:         STMFD   r0!,{r1-r4,r12,lr}  ; save APCS-clobbered regs
17:                                     ; on SYS stack plus r4 for the priority.
18:
19:         MSR     cpsr_c,#ARM_IRQ_MODE | ARM_INT_BITS
20:         MOV     r14,r13             ; put original r0_SYS in r14_IRQ
21:         MRS     r13,SPSR            ; put interrupted PSR in r13_IRQ
22:         STMFD   r0!,{r13,r14}       ; finish saving the context
23:
24:         MSR     cpsr_c,#ARM_SYS_MODE | ARM_INT_BITS
25:         MOV     sp,r0               ; adjust sp_SYS
26:
27:         LDR     r0,=QK_currPrio_    ; load address in already saved r0
28:         LDRB    r4,[r0]             ; load QK_currPrio into APCS-preserved r4
29:
30:         MOV     r0,#0x0
31:         LDR     r12,[r0,#-0xFD0]    ; load the vector from the VICVectAddr
32:         ; IRQ entry }}}
33:
34:         ; NOTE: the C-portion of the ISR is called with interrupts
35:         ; *disabled* because it still needs to raise the QK priority
36:         ; QK_currPrio_to the ISR level and also might need to clear
37:         ; a level-sensitive interrupt soruce. The C-ISR might then
38:         ; re-enable interrupts, if appropriate.
39:         ; NOTE: the C-portion of the ISR runs in THUMB mode and returns
40:         ; without mode change, so the following code is in THUMB
```

```
41:         ;
42:         MOV     lr,pc               ; store the return address
43:         BX      r12                 ; call the IRQ vector (THUMB)
44:
45:         CODE16 ; we're now in THUMB
46:         ADR     r0,irq_sched
47:         BX      r0
48:
49:         CODE32 ; we're now back in ARM
50: irq_sched:
51:         ; IRQ exit {{{
52:         MSR     cpsr_c,#ARM_SYS_MODE | ARM_INT_BITS    ; SYS mode, lock int.
53:
54:         ; handle the end-of-interrupt in the interrupt controller
55:         MOV     r0,#0x0
56:         STR     r0,[r0,#-0xFD0]     ; write VICVectAddr to clear interrupt
57:
58:         LDR     r0,=QK_currPrio_    ; load address
59:         STRB    r4,[r0]             ; restore intitial prio in QK_currPrio_
60:
61:         LDR     r12,=QK_schedule_
62:         MOV     lr,pc               ; store the return address
63:         BX      r12                 ; call QK_schedule_ (THUMB)
64:                                     ; NOTE: QK_schedule_ must be called
65:                                     ; with interrupts DISABLED
66:
67:         CODE16 ; we're now in THUMB
68:         ADR     r0,irq_exit
69:         BX      r0
70:
71:         CODE32 ; we're now back in ARM
72: irq_exit:
73:         MOV     r0,sp               ; make sp_SYS visible to IRQ mode
74:         ADD     sp,sp,#36           ; adjust the sp_SYS
75:
76:         MSR     cpsr_c,#ARM_IRQ_MODE | ARM_INT_BITS
77:         MOV     sp,r0               ; stick sp_SYS to sp_IRQ
78:         LDMFD   sp!,{r0}            ; grab saved PSR
79:         MSR     spsr_cxsf,r0        ; stick it into spsr_IRQ
80:
81:         LDMFD   sp,{r0-r4,r12,lr}^  ; unstack all saved SYS registers
82:         MOV     r0,r0               ; NOP: can't access banked reg immediately
83:         LDR     lr,[sp,#28]         ; grab the return address from the stack
84:         MOVS    pc,lr               ; return restoring cpsr from spsr
85:
86:         ; IRQ exit }}}
87:
88:         LTORG
89:
90:         ENDMOD
```

Listing 7 The assembly IRQ handler for the THUMB port (defined in the module <qpcpp_3>/qk/arm7-lpc2000/iar/thumb/qk_vect.s79)

The highlights of the IRQ shown in Listing 7 are as follows: The IRQ stack is not used, so the banked register r13_IRQ is used as a scratchpad register (see Listing 7 line 10). Also this code avoids accessing memory and maximizes register use for passing data. The ARM core is switched several times between the IRQ and the System modes, but the interrupts are always disabled when

the ARM core is in the IRQ mode. This makes the IRQ mode completely transparent, or in other words, it is impossible to "catch" (interrupt) the ARM processor in the IRQ mode[1].

The purpose of the lines 10-22 of Listing 7 is to build the following stack frame:

```
high memory

    PC (return address)
    LR
    R12
    R4 (pin)
    R3
    R2
    R1
    R0
    SPSR

low memory
```

This stack frame demonstrates the "QK-friendliness" to the C-code. Only the registers clobbered by the ARM Procedure Call Standard (APCS) are saved, plus the APCS-preserved R4 register that holds the initial `QK_currPrio_` value (`pin`). This is only about a half of all ARM registers that traditional kernels need to save and restore by every interrupt and context switch.

In lines 30-32 of Listing 7 you see the use of the **vectoring** feature of the VIC. The `LDR` instruction in line 34 reads the `VICVectAddr` register at 0xFFFFF030 (== 0 – 0xFD0). Subsequently, the `BX r12` instruction (line 43) jumps to the vector received from the VIC. This means that the IRQ handler in Listing 7 is a universal "shell" that services all IRQ requests in the system. Of course, as in the auto-vectoring case, the VIC must be correctly initialized with the addressed of all the IRQ service routines used in the system. The following line shows the VICVectAddr0 initialization for the tick-ISR():

```
    VICVectAddr0 = (uint32_t)&tickISR;
```

5.3.3 The IRQ Handlers in C

The C-level IRQ handlers in the THUMB port, such as the `tickISR()` are normal C functions, and not `__irq`-type functions as in the other ports. (That's why in the THUMB port they have postfix "ISR" rather than "IRQ", as in the other ports.)

The following listing shows the implementation of the `tickISR()` function:

```
void tickISR(void) {
    T1IR = 0x1;                                  // clear the timer interrupt
    QK_ISR_ENTRY(TICK_ISR_PRIO);

    QF::tick();

    QK_ISR_EXIT();
}
```

The ISR is entered with interrupts disabled, because some of the interrupts might require clearing the level-sensitive interrupt source prior to enabling interrupts to the core. Indeed, the `tickISR()` performs such clearing of the Timer1 interrupt. After this, the interrupts can be enabled and some ISR-specific work can be performed.

[1] Strictly speaking, the FIQ can interrupt the IRQ, but this discussion assumes that the FIQ is handled completely separately, as described in Section 5.2.1.1.

6 References

[Atmel 98a]	Atmel, Application Note "Interrupt Management: Auto-vectoring and Prioritization", 1998
[IAR 05a]	IAR Systems, "ARM® IAR C/C++ Compiler Reference Guide", June 2005
[Philips 03]	Philips Semiconductors: AN-10254 "Simple Interrupt Handling Using IRQ and FIQ", available online from www.philips.com
[Philips 05a]	Philips Semiconductors: AN-10381 "Nesting of Interrupts in the LPC2000", available online from www.philips.com
[Philips 05b]	Philips Semiconductors: UM10120 "LPC213x User Manual", available online from www.philips.com
[QL 05a]	Quantum Leaps, LLC, "Quantum Platform Overview" (http://www.quantum-leaps.com/doc/QP_Overview.pdf
[QL 05b]	Quantum Leaps, LLC, Application Note: QP Directory Structure" (http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf
[Samek 02]	Miro Samek, "Practical Statecharts in C/C++", CMP Books 2002.