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

Document Revision Q
October 2012
# Table of Contents

1 Introduction .................................................................................................................. 1  
1.1 About QP-nano™ ........................................................................................................ 2  
1.2 About QM™ .................................................................................................................. 3  
1.3 About the QP-nano™ Port to ARM7/9 ........................................................................ 4  
1.4 Licensing QP-nano™ .................................................................................................. 4  
1.5 Licensing QM™ .......................................................................................................... 4  

2 Directories and Files ....................................................................................................... 5  
2.1 Building the Examples ............................................................................................... 6  
2.2 Downloading to Flash and Debugging the Examples .................................................. 9  
2.3 Running the Examples ............................................................................................... 13  

3 The Linker Script ........................................................................................................... 14  
3.1 Linker Options ............................................................................................................ 17  

4 Non-Preemptive Configuration of QP-nano .................................................................. 19  
4.1 Configuration and Customizing QP-nano (qpn_port.h) .................................................. 19  
4.2 ARM-Specific QF-nano Port (qfn_port.s) .................................................................... 21  

5 Preemptive Configuration with QK-nano ...................................................................... 34  
5.1 Configuration and Customizing QP-nano .................................................................... 34  
5.2 Handling Interrupts in QK-nano .................................................................................. 35  
5.3 Idle Loop Customization in the QK Port ..................................................................... 39  

6 Related Documents and References .............................................................................. 40  

7 Contact Information ....................................................................................................... 41
1 Introduction

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

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

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

Figure 1: AT91SAM7S-EK evaluation board from Atmel and the J-Link pod

To focus the discussion, this Application Note uses the AT91SAM7S-EK development board from Atmel as an example hardware platform. The actual hardware/software used is as follows:
1. **QP-nano** version **4.5.02** or higher available from [www.state-machine.com/downloads](http://www.state-machine.com/downloads).


7. AT91SAM7S-EK evaluation board from Atmel (AT91SAM7S64 MCU)


1.1 **About QP-nano™**

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


![Figure 2 QP-nano components and their relationship with the target hardware, board support package (BSP), and the application](image-url)
1.2 About QM™

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

QM™ provides intuitive diagramming environment for creating good looking hierarchical state machine diagrams and hierarchical outline of your entire application. QM™ eliminates coding errors by automatic generation of compact C or C++ code that is 100% traceable from your design. Please visit [state-machine.com/qm](http://state-machine.com/qm) for more information about QM™.

The code accompanying this App Note contains three application examples: the PEdestrian LIght CONtrolled [AN-PELICAN] crossing for the AT91SAM7S-EK board (see Chapter 1 in [PSiCC2] all modeled with QM.

NOTE: The provided QM model files assume QM version 2.2.03 or higher.

Figure 3: The PELICAN example model opened in the QM™ modeling tool

![Image of the PELICAN example model opened in the QM™ modeling tool]
1.3 About the QP-nano™ Port to ARM7/9

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

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

1. The ARM processor executes in both ARM and THUMB states, meaning that some parts of the code are compiled to ARM and others to THUMB instruction sets, and calls between ARM and THUMB functions are allowed. Such approach is supported by the "interwork" option of the ARM compilers and linkers. This choice is optimal for most ARM-based microcontrollers, where large parts of the code execute from slower Flash ROM that in many cases is only 16-bit wide. The higher code density of the THUMB instruction set in such cases improves performance compared to ARM, even though THUMB is a less powerful instruction set.

2. The ARM processor operates in the SYSTEM mode \((\text{CPSR}[0:4] = 0x1F)\) while processing task-level code, and briefly switches to the IRQ mode \((\text{CPSR}[0:4] = 0x12)\) or FIQ mode \((\text{CPSR}[0:4] = 0x11)\) to process IRQ or FIQ interrupts, respectively. The System mode is used for its ability to execute the \text{MSR}/\text{MRS} instructions necessary to quickly lock and unlock interrupts. NOTE: The SYSTEM mode is the default mode assumed by the IAR compiler for execution of applications.

3. The ARM processor uses only single stack (the USER/SYSTEM stack) for all tasks, interrupts, and exceptions. The private (banked) stack pointers in SUPERVISOR, ABORT, UNDEFINED, IRQ, and FIQ modes are used only as working registers, but not to point to the private stacks. This means that you don’t need to allocate any RAM for the SUPERVISOR, ABORT, UNDEFINED, IRQ, or the FIQ stacks and you don’t need to initialize these stack pointers. The only stack you need to allocate and initialize is the USER/SYSTEM stack.

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

1.4 Licensing QP-nano™

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

- The GNU General Public License version 2 (GPL) as published by the Free Software Foundation and appearing in the file \text{GPL.TXT} included in the packaging of every Quantum Leaps software distribution.

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

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

1.5 Licensing QM™

The QM™ graphical modeling tool available for download from the www.state-machine.com/downloads website is free to use, but is not open source. During the installation you will need to accept a basic End-User License Agreement (EULA), which legally protects Quantum Leaps from any warranty claims, prohibits removing any copyright notices from QM, selling it, and creating similar competitive products.
2 Directories and Files

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

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

```plaintext
<qpn>/                     - QP-nano Root Directory
+-examples/              - QP-nano examples
  +-arm\                 - examples for ARM
    +-gnu\               - examples compiled with the GNU compiler
      +-metadata         - directory with Eclipse workspace
        +pelican-at91sam7s-ek\ - PELICAN crossing example for AT91SAM7S-EK (Vanilla)
          +-dbg\          - directory containing the debug build
          +-rel\          - directory containing the release build
          +-Makefile      - Makefile for building the application
          +-startup.s     - ARM startup code in assembly
          +-pelican.ld    - linker command file for this application
        +-bsp.h          - Board Support Package include file
        +-bsp.c          - Board Support Package implementation
        +-isr.c          - Interrupt Service Routines (ISRs)
          +-low_level_init.c - Low-level initialization of the MCU
        +-main.c         - main function
        +-pelican.h     - PELICAN HSM declarations (generated by QM)
        +-pelican.c     - PELICAN HSM implementation (generated by QM)
          +-pelican.qm   - QM model of the PELICAN crossing
          +-pelican.e wp - IAR project for the PELICAN crossing example
          +-pelican.ew w - IAR workspace for the PELICAN crossing example
          +-qp n_port.h  - QP-nano port (Vanilla)
    +-include\         - subdirectory containing the QP-nano public interface
    +-source/          - QP-nano source files
      +-ports\         - subdirectory containing the QP-nano ports
```
2.1 Building the Examples

This Application Note uses the AT91SAM7S-EK development board from Atmel as an example hardware platform. The examples included in this Application Note are based on the standard PELICAN crossing example (see Chapter 12 in [PSiCC2] and the Application Note “PEdestrian LIght CONtrolled Crossing Example” [AN-PELICAN] included in the QDK-nano download).

2.1.1 The Makefile

Whichever way you’ll choose to build the QP-nano examples (command-line or Eclipse), you are going to use the same Makefile, provided in the directory <qpn>/examples/arm/gnu/pelican-at91sam7s-ek/.

The most important part of this Makefile, which you need to customize for your own projects, is the files section. For the classic ARM architecture (ARM7/ARM9), the file section allows you to specify which modules are going to be compiled in ARM mode and which in the THUMB mode. This determination is quite important for achieving optimal performance. Generally interrupt processing and system-level code performs better when compiled to ARM, while the application-level code (e.g., state machines) perform better and take less codespace when compiled to THUMB. The following Listing 2 shows the “files” section of the Makefile.

```
Listing 2 The “files” section of the Makefile allows you to specify files to be compiled to ARM and to THUMB
```

```makefile
# files
#

# assembler source files
ASM_SRCS := $(wildcard *.s) \
  qfn_port.s

# C ARM source files
C_ARM_SRCS := isr.c bsp.c \
  qepn.c qfn.c

# C THUMB source files
C_THUMB_SRCS := low_level_init.c main.c pelican.c

# C++ ARM source files
CPP_ARM_SRCS :=

# C++ THUMB source files
CPP_THUMB_SRCS :=

LD_SCRIPT := pelican.ld```

...
2.1.2 Building the Examples from Command Line

The example directory <qpn>/examples/arm-gnu/pelican-at91sam7s-ek\ contains the Makefile you can use to build the application. The Makefile supports two build configurations: Debug (default), and Release. You choose the build configuration by defining the CONF symbol at the command line, as shown in the table below. Figure 4 shows and example command-line build of the default (Debug) configuration.

<table>
<thead>
<tr>
<th>Build Configuration</th>
<th>Build command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug (default)</td>
<td>make</td>
</tr>
<tr>
<td>Release</td>
<td>make CONF=rel</td>
</tr>
<tr>
<td>Clean the Debug configuration</td>
<td>make clean</td>
</tr>
<tr>
<td>Clean the Release configuration</td>
<td>make CONF=rel clean</td>
</tr>
</tbody>
</table>

Figure 4 Building the PELICAN application with the provided Makefile from command-line

2.1.3 Building the Examples from Eclipse

The example code contains the Eclipse workspaces and projects for building and debugging the PELICAN examples with the Eclipse IDE. The following screen shot shows how to open the provided workspace in Eclipse.
NOTE: The “How To” section of the YAGARTO website at [http://www.yagarto.de/howto/yagarto2](http://www.yagarto.de/howto/yagarto2) provides a detailed tutorial for installing and configuring Eclipse for ARM development. You can create and configure the build configurations from the Project | Build Configurations | Manage... sub-menu. For the Release and Spy configurations, you should set the make command to `make CONF=rel` and `make CONF=spy`, respectively. The provided Makefile also correctly supports the clean targets, so invoking Project | Clean... menu for any build configuration works as expected.

The provided Eclipse projects are Makefile-type projects, which use the same Makefiles that you can call from the command line. In fact the Makefiles are specifically designed to allow building all supported configurations from Eclipse, as shown in Figure 5.
2.2 Downloading to Flash and Debugging the Examples

You have several options for downloading and debugging the examples. The devkitPro project includes the Insight GDB front-end ([https://sourceforge.net/projects/devkitpro/files/Insight](https://sourceforge.net/projects/devkitpro/files/Insight) [Insight-GDB]), which is perhaps the simplest way. The other option is to use Eclipse with the Zylin Embedded CDT plugin. Please refer to the Zylin website at [http://opensource.zylin.com/embeddedcdt.html](http://opensource.zylin.com/embeddedcdt.html) for more information how to install the CDT plugin in Eclipse.

2.2.1 Starting the GDB Server

Regardless whether you use Insight, Eclipse, or even just the raw GDB, you would need a J-Tag pod and a GDB Server software. This Application Note uses the J-Link pod from SEGGER and the J-Link GDB Server software v4.08l available for download from [www.segger.com/cms/downloads.html](http://www.segger.com/cms/downloads.html). However, you might just as well use other J-Tag pods and other GDB server software, such as OpenOCD ([http://openocd.berlios.de](http://openocd.berlios.de)).

You launch the J-Link GDB server by clicking on the JLinkGDBServer.exe application in the SEGGER directory.

**NOTE:** You need a license to run the J-Link GDB Server. SEGGER now offers free licenses for Non-Commercial-Use, see [http://www.segger.com/cms/j-link-arm-for-non-commercial-use.html](http://www.segger.com/cms/j-link-arm-for-non-commercial-use.html).
2.2.2 Downloading to Flash and Debugging with Insight

The example directory `<qpn>/examples/arm/gnu/pelican-at91sam7s-ek` contains the GDB command file `jlink.gdb` file as well as the `insight-jlink.bat` batch file to launch the Insight debugger. To start debugging you can simply double-click on the `drom_jlink.bat` batch file, which will launch the Insight debugger and load the `dbg\pelican.elf` image to the flash memory of the AT91SAM7S microcontroller. Figure 7 shows a screen shot from the Insight Debug session.

As the application is running, the status LEDs at the top of the board should blink indicating the changing status of the lights in the PELICAN crossing.
2.2.3 Downloading to Flash and Debugging with Eclipse

You can also use Eclipse to debug the PELICAN application. Configuring the Eclipse debugger is somewhat involved and requires a plugin to improve Embedded debugging with the Eclipse CDT. The YAGARTO website provides a step-by-step tutorial (see http://www.yagarto.de/howto/yagarto2/) of how to install Eclipse and the Zylin Embedded CDT plugin. The following discussion assumes that you have installed the Zylin plugin.

Figure 8 shows screen shots that illustrate how to open the Debug dialog box and configure a debug session in Eclipse. The Main tab of the Debug dialog box shows how the project is setup for debugging. The Debugger tab of the Debug dialog box shows how to configure the devkitPro debugger and the GDB command file. Here you use the same jlink.gdb GDB command file as for debugging with Insight, because you use the same J-Link GDB server shown in Figure 6.
Figure 8 Configuring the Debugger in Eclipse

1. Launch the Debug dialog box.
2. Main tab of the Debug dialog box.
3. Debugger tab of the Debug dialog box.
The following screen shot shows a debugging session in Eclipse with various views. The SEGGER J-Link GDB Server is shown in the lower-right corner of the screen.

2.3 Running the Examples

After you successfully build and download the PELICAN crossing example to the AT91SAM7S-EK board, you can run the example (e.g., by pressing the Go button). As the application starts running, the LEDs 1 and 4 of the AT91SAM7S-EK should light up. This means that Cars get green light and Pedestrians get DON'T WALK signal. To activate the crossing, you need to push the PB1 button in the corner of the board (see Figure 1).
3 The Linker Script

The linker script must match the startup code for all the section names and other linker symbols. The linker script cannot be generic, because it must define the specific memory map of the target device, as well as other application-specific information. The linker script therefore named here `pelican.ld`, which corresponds to the PELICAN crossing example application.

Listing 3 Linker script for the PELICAN crossing example application (AT91SAM7S64 MCU)

(1) OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
(2) OUTPUT_ARCH(arm)
(3) ENTRY(_vectors)

(4) MEMORY { /* memory map of AT91SAM7S64 */
(5) ROM (rx) : ORIGIN = 0x00100000, LENGTH = 64k
(6) RAM (rwx): ORIGIN = 0x00200000, LENGTH = 16k
}

/* The size of the stack used by the application. NOTE: you need to adjust */
(7) STACK_SIZE = 512;

/* The size of the heap used by the application. NOTE: you need to adjust */
(8) HEAP_SIZE = 0;

SECTIONS {

(9) .reset : { /* startup code (ARM vectors and reset handler) */
(10) *startup.o (.text) /* startup code (ARM vectors and reset handler) */
(11) . = ALIGN(0x4);
(12) } >ROM

(13) .ramvect : { /* used for vectors remapped to RAM */
(14) _ram_start = .;
(15) . = 0x40;
(16) } >RAM

(17) .fastcode : {
(18) __fastcode_load = LOADADDR (.fastcode);
(19) __fastcode_start = .;
(20) *(.text.fastcode) /* all functions explicitly placed in .fastcode */
(21) *(.text.QF_tick)
(22) *(.text.QF_postISR)
(23) /* add other modules here ... */
(24) . = ALIGN (4);
(25) __fastcode_end = .;

(26) .text : {
(27) . = ALIGN(4);
(28) *(.text) /* .text sections (code) */
(29) *(.text*) /* .text* sections (code) */
(30) *(.rodata) /* .rodata sections (constants, strings, etc.) */

Copyright © Quantum Leaps, LLC. All Rights Reserved.


/* .rodata* sections (constants, strings, etc.) */
*(.rodata*)

/* .glue_7 */ glue arm to thumb (NOTE: placed already in .fastcode) */
*(.glue_7)

/* .glue_7t */ glue thumb to arm (NOTE: placed already in .fastcode) */
*(.glue_7t)

(26)

(27)

KEEP (*(.init))
KEEP (*(.fini))

. = ALIGN(4);
_etext = .;
/* global symbol at end of code */
(28)

(29)

.proinit_array : {
    PROVIDE_HIDDEN (__preinit_array_start = .);
    KEEP (*(.preinit_array*))
    PROVIDE_HIDDEN (__preinit_array_end = .);
} >ROM

(30)

.init_array : {
    PROVIDE_HIDDEN (__init_array_start = .);
    KEEP (*(SORT(.init_array.*)))
    KEEP (*(.init_array*))
    PROVIDE_HIDDEN (__init_array_end = .);
} >ROM

(31)

.fini_array : {
    PROVIDE_HIDDEN (__fini_array_start = .);
    KEEP (*(.fini_array*))
    KEEP (*(SORT(.fini_array.*)))
    PROVIDE_HIDDEN (__fini_array_end = .);
} >ROM

(32)

.data : {
    __data_load = LOADADDR (.data);
    __data_start = .;
    *(.data) /* .data sections */
    *(.data*) /* .data* sections */
    . = ALIGN(4);
    _edata = .;
} >RAM AT>ROM

(33)

.bss : {
    __bss_start__ = .;
    *(.bss)
    *(.bss*)
    *(COMMON)
    . = ALIGN(4);
    _ebss = .;
    __bss_end__ = .; /* define a global symbol at bss end */
} >RAM

(34)

(35)

(36)

PROVIDE ( end = _ebss );
PROVIDE ( _end = _ebss );
PROVIDE ( __end__ = _ebss );

(37)

.heap : {
    __heap_start__ = .;
    . = . + HEAP_SIZE;

Listing 3 shows the linker script for the PELICAN example application. The script is almost identical for C and C++ versions, with the minor differences discussed later in this section. The highlights of the linker script are as follows:

1. The OUTPUT_FORMAT directive specifies the format of the output image (elf32, little-endian, ARM).
2. OUTPUT_ARCH specifies the target machine architecture.
3. ENTRY explicitly specifies the first instruction to execute in a program.
4. The MEMORY command describes the location and size of blocks of memory in the target.
5. The region ROM corresponds to the on-chip flash of the AT91SAM7S64 device. It can contain read-only and executable sections (rx), it starts at 0x00100000 and is 64KB in size.
6. The region RAM corresponds to the on-chip SRAM of the AT91SAM7S64 device. It can contain read-only, read-write and executable sections (rwx), it starts at 0x00200000 and is 16KB in size.
7. The STACK_SIZE symbol determines the sizes of the ARM stack. You need to adjust the sizes for your particular application. The stack size cannot be zero.

**NOTE:** The QP port to ARM uses only one stack (the USER/SYSTEM stack).

8. The HEAP_SIZE symbol determines the sizes of the heap. You need to adjust the sizes for your particular application. The heap size can be zero.
9. The .reset section contains the startup code (including the ARM vectors) and must be located as the first section in ROM.
10. This line section locates all .text sections from the startup.o object module.
11. The section size is aligned to the 4-byte boundary.
12. This section is loaded directly to the ROM region defined in the MEMORY command.
13. The .ramvect section contains the RAM-based ARM vector table and the secondary jump table and must be loaded as the first section in RAM.
14. The ARM vector table and the secondary jump table have known size of 0x40 bytes. The current location counter is simply incremented to reserve 0x40 bytes for the section.
15. The .ramvect section goes into the RAM region.
16. The .fastcode section is used for RAM-based code, which needs to be loaded to ROM, but copied and executed from RAM.
17. The .fastcode section has different load memory address (LMA) than the virtual memory address (VMA). The symbol __fastcode_load corresponds to the LMA in ROM and is needed by the startup code to copy the section from ROM to RAM.
18. The __fastcode_start symbol corresponds to the VMA of the .fastcode section and is needed by the startup code to copy the section from ROM to RAM.
(19) The `.glue_7t` and `.glue_7` sections are synthesized by the compiler when you specify the ARM-THUMB interworking option. The sections contain the “call veeners” between THUMB and ARM code and are accessed frequently by every call between ARM and THUMB. It’s typically advantageous to place this small amount of hot-spot code in RAM.

(20) The `.text.fastcode` section is assigned explicitly to individual functions in the C/C++ code by means of the `__attribute__((section(".text.fastcode")))` command.

(21) The GNU compiler is also capable of placing each function in the separate section named after the function (requires specifying the option `-ffunction-sections`). This allows you to be very selective and to place individual functions (e.g. the function `QF_run()`) in RAM.

(22) You can place more hot-spot functions in RAM during the fine-tuning stage of the project.

(23) The `.fastcode` section is located in RAM, but is loaded at the ROM address.

(24) The `.text` section is for code and read-only data accessed in place.

(25) The `.rodata` is used for read-only (constant) data, such as look-up tables. Just as code, you might choose to place some frequently accessed constants in RAM by locating these sections in the `.fastcode` section.

(26) If you repeat sections already located in the `.fastcode` section, the earlier location will take precedence. However, if you decide to remove these sections from `.fastcode`, they will be located per the second specification.

(27) The `.init` and `.fini` sections are synthesized by the GNU C++ compiler and are used for static constructors and destructors. These sections are empty in C programs.

(28) The `.text` section is located and loaded to ROM.

(29,30) The `.preinit_array` and `.inint_array` sections hold arrays of function pointers that are called by the startup code to initialize the program. In C++ programs these hold pointers to the static constructors that are called by `__libc_init_array()` before `main()`.

(31) The `.fini_array` section holds an array of function pointers that are called before terminating the program. In C++ programs this array holds pointers to the static destructors.

(32) The `.data` section contains initialized data.

(33) The `.data` section is located in RAM, but is loaded to ROM and copied to RAM during startup.

(34) The `.bss` section contains uninitialized data. The C/C++ standard requires that this section must be cleared at startup.

(35) The `.bss` section is located in RAM only.

(36) The symbols marking the end of the `.bss` sections are used by the startup code to allocate the beginning of the heap.

(37) The `.heap` section contains the heap (please also see the `HEAP_SIZE` symbol definition in line (8))

---

**NOTE:** Even though the linker script supports the heap, it is almost never a good idea to use the heap in embedded systems. Therefore the examples provided with this Application Note contain the file `no_heap.c/cpp`, which contains dummy definitions of `malloc()`, `free()`, and `realloc()` functions. Linking this file saves some 2.8KB of code space compared to the actual implementation of the memory allocating functions.

(38) The `.stack` section contains the C stack (please also see the `STACK_SIZE` symbol definition in line (7)). The stack memory is initialized with a given bit-pattern at startup.

---

**NOTE:** The QP port to ARM uses only one stack (the USER/SYSTEM stack). Other ARM stacks, such as IRQ, FIQ, SUPERVISOR, or ABORT are not used and not even initialized.

### 3.1 Linker Options

The linker options for C and C++ are the same and are defined in the Makefile located in the PELICAN directory. The most
Linker options for C and C++ builds.

(1) LINKFLAGS = -T ./$(LD_SCRIPT) \
(2) -o $(BINDIR)/$(OUTPUT).elf \
(3) -Wl,-Map,$(BINDIR)/$(OUTPUT).map,--cref,--gc-sections

(1) –T option specifies the name of the linker script (pelican.ld in this case).
(2) –o option specifies the name of image file (pelican.elf in this case).
(3) --gc-sections enable garbage collection of unused input sections.

NOTE: This bare-metal application replaces the standard startup sequence defined in crt0.o with the customized startup code. Even so, the linker option -nostartfiles is not used, because some parts of the standard startup code are actually used. The startup code is specifically important for the C++ version, which requires calling the static constructors before calling main().
4 Non-Preemptive Configuration of QP-nano

The example of using QP-nano with the cooperative “Vanilla” kernel built into the QF-nano is located in the directory: `<qpn>/examples/arm/gnu/pelican-at91sam7s-ek/`. This section describes the generic QP-nano configuration, which consists of the `qpn_port.h` header file and `qfn_port.s` ARM-specific port in assembly. The board-specific elements are common for both the non-preemptive and preemptive (QK-nano) configuration and will be covered in Section 5.

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

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

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

```c
#ifndef qpn_port_h
#define qpn_port_h

(1) #define Q_NFSM
(2) #define Q_PARAM_SIZE 4
(3) #define QF_TIMEEVT_CTR_SIZE 2
/* maximum # active objects--must match EXACTLY the QF_active[] definition */
(4) #define QF_MAX_ACTIVE 6
/* fast unconditional interrupt locking/unlocking for ARM state */
(5) #define QF_INT_LOCK_32() __asm volatile("MSR cpsr_c,#(0x1F | 0x80 | 0x40)" ::: "cc")
(6) #define QF_INT_UNLOCK_32() __asm volatile("MSR cpsr_c,#(0x1F)" ::: "cc")

(7) #ifdef __thumb__                                             /* THUMB mode? */
     /* interrupt locking policy for task-level */
     (8) #define QF_INT_LOCK() QF_int_lock_SYS()
     (9) #define QF_INT_UNLOCK() QF_int_unlock_SYS()
(10) void QF_int_lock_SYS(void);
(11) void QF_int_unlock_SYS(void);
     /* interrupt locking policy for ISR-level */
(12) #define QF_ISR_NEST
(13) #define QF_ISR_KEY_TYPE unsigned long
(14) #define QF_ISR_LOCK(key_) ((key_) = QF_isr_lock_SYS())
(15) #define QF_ISR_UNLOCK(key_) (QF_isr_unlock_SYS(key_))
(16) QF_ISR_KEY_TYPE QF_isr_lock_SYS(void);
(17) void QF_isr_unlock_SYS(QF_ISR_KEY_TYPE key);

(18) #elif (defined __arm__)                                        /* ARM mode? */
     /* interrupt locking policy for task-level */
     (19) #define QF_INT_LOCK() QF_INT_LOCK_32()
     (20) #define QF_INT_UNLOCK() QF_INT_UNLOCK_32()
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.
Listing 4 qpn_port.h header file for the non-preemptive QF-nano configuration and GNU compiler

(1) Defining the macro Q_NFSM eliminates the code for the simple non-hierarchical FSMs.
(2) The macro Q_PARAM_SIZE defines the size (in bytes) of the scalar event parameter. The allowed values are 0 (no parameter), 1, 2, or 4 bytes. If you don't define this macro in qpn_port.h, the default of 0 (no parameter) will be assumed.
(3) The macro QF_TIMEEVT_CTR_SIZE defines the size (in bytes) of the time event down-counter. The allowed values are 0 (no time events), 1, 2, or 4 bytes. If you don't define this macro in qpn_port.h, the default of 0 (no time events) will be assumed.
(4) You must define the QF_MAX_ACTIVE macro as the exact number of active objects used in the application. The provided value must be between 1 and 8 and must be consistent with the definition of the QF_active[] array in main.c.
(5-6) The macros QF_INT_LOCK_32()/QF_INT_UNLOCK_32() perform a very efficient unconditional interrupt locking/unlocking using just one MSR instruction with immediate argument. As indicated by the _32 suffix, these macros can only be called in the 32-bit ARM state, because only ARM state supports the MSR instruction.

NOTE: In this QP-nano port to the ARM processors, the C-level code executes exclusively in the SYSTEM mode. The interrupt locking/unlocking functions that can be called only from the C code, might as well take advantage of the known CPU mode.
(7) Interrupt locking/unlocking cannot be accomplished in the THUMB state, because the MSR/MRS instructions necessary to manipulate the CPSR register are not available in THUMB. Therefore, the only option to accomplish interrupt locking/unlocking from THUMB is to call an ARM function, such as QF_int_lock_SYS() / QF_int_unlock_SYS().

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(30-31) The BSP functions encapsulate the particular interrupt controller. These functions are described in the upcoming Section 4.2.3.

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

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

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

### 4.2 ARM-Specific QF-nano Port (qfn_port.s)

Due to the programming complexity of the ARM7/ARM9 cores (see Section 1.3), the QP-nano port to ARM requires some assembly programming. The assembly module qfn_port.s defines the task-level and ISR-level interrupt locking and the ARM exception handlers.

The following sub-sections covers the main elements of the qfn_port.s implementation, starting with the critical section, through interrupt “wrapper” functions in assembly, and finally all other exception “wrapper” functions.
NOTE: The QP-nano port to the cooperative "Vanilla" kernel *qfn_port.s* is generic and should not need to change for other ARM systems.

### 4.2.1 The Critical Section Implementation

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

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

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

#### Listing 5 Task-level and ISR-level critical section definitions in *qfn_port.s*.

```assembly
.equ    NO_INT,      0xC0   /* mask to disable interrupts (FIQ and IRQ) */
.equ    NO_IRQ,      0x80   /* mask to disable interrupts (FIQ and IRQ) */
.equ    NO_FIQ,      0x40   /* mask to disable interrupts (FIQ and IRQ) */
.equ    FIQ_MODE,    0x11
.equ    IRQ_MODE,    0x12
.equ    SYS_MODE,    0x1F

.text
(1)     .arm
/* use the special section (.text.fastcode), to enable fine-tuning */
/* of the placement of this section inside the linker script */
(2)     .section .text.fastcode

/*****************************************************************************
* void QF_int_lock_SYS(void);
*/
.global QF_int_lock_SYS
.func   QF_int_lock_SYS
QF_int_lock_SYS:
(3)     MSR     cpsr_c,#(SYS_MODE | NO_INT) /* disable both IRQ/FIQ */
(4)     BX      lr

.size   QF_int_lock_SYS, . - QF_int_lock_SYS
.endfunc

/*****************************************************************************
* void QF_int_unlock_SYS(void);
*/
.global QF_int_unlock_SYS
.func   QF_int_unlock_SYS
QF_int_unlock_SYS:
(5)     MSR     cpsr_c,#(SYS_MODE)  /* enable both IRQ/FIQ */
     BX      lr
```

22 of 41
The functions require the 32-bit ARM state to be able to use the \texttt{MSR/MRS} instructions.

The module is declared in the section \texttt{.fastcode} located in RAM. For almost all ARM-based MCUs, the code executing from RAM is significantly faster (sometimes as much as 3-4 times faster) than code executing from ROM due to wait states necessary to access slow, and often only 16-bit wide Flash ROM. At the cost of just several bytes of RAM you get significant performance boost for the "hot-spot" interrupt locking/unlocking code.

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

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

The function returns via the \texttt{BX} instruction, which causes the T-bit to be set in the \texttt{CPSR\_c} if the return address is a THUMB label.

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

The original value of \texttt{CPSR} is moved to \texttt{r0}, which is then returned from the \texttt{QF\_int\_lock} function.
Copyright © Quantum Leaps, LLC. All Rights Reserved.
4.2.3 Handling Interrupts

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

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

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

```
ORG 0x18
LDR pc,[pc, #-0xF20]
```

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

```
0x20 – 0xF20 = 0xFFFFF100
```

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

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

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

**Figure 9 Typical ARM system with an interrupt controller (external to the ARM core).**

![Diagram of ARM system with interrupt controller](image-url)

**Figure 9** shows a typical ARM-based system with an interrupt controller. Typically, the interrupt prioritization is performed only with respect to the IRQ line, while the FIQ line is routed directly to the ARM core. The ARM core itself performs a 2-level interrupt prioritization between the FIQ and IRQ, whereas FIQ is higher priority than the IRQ. This second-level of prioritization is performed by the I and F bits of the CPSR register, which are also used for interrupt locking and unlocking policy.

Even though “auto vectoring” is not used, vectoring to a specific interrupt handler can still occur, but is done later, at the C-level interrupt handler functions implemented typically in the Board Support Package. Examples of such BSP functions for the popular interrupt controllers are provided later in this manual.
Figure 10 shows the interrupt processing sequence in the presence of an interrupt controller (Atmel's AIC is assumed in this example). The ARM vector addresses 0x18 and 0x1C point to the assembler “wrapper” functions QF_irq and QF_fiq, respectively. Each of these “wrapper” functions, for example QF_irq, performs context save, switches to the SYSTEM mode, and invokes a C-level function BSP_irq (or BSP_fiq for the FIQ interrupt). BSP_irq encapsulates the particular interrupt controller, from which it explicitly obtains the interrupt vector. Because the interrupt controller is used in this case, it must be initialized with the addresses of all used interrupt service routines (ISRs), such as tickIRQ() shown in Figure 10. Please note that these ISRs are regular C-functions and not __irq type functions because the interrupt entry and exit code is already provided in the assembly “wrapper” functions QF_irq and QF_fiq.

4.2.4 The IRQ “Wrapper” Function in Assembly

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

Listing 6 The QF_irq assembly wrapper for the “vanilla” QF-nano port (qfn_port.s).

```assembly
;------------------------------------------------------------------------------
; IRQ assembly wrapper
;------------------------------------------------------------------------------
(1)  SECTION .text rw:DATA:NOROOT(2)
```
The IRQ wrapper `QF_irq` is defined in section .text, declared as DATA to be placed in RAM for fastest execution. Such time-critical ARM code is best executed from 32-bit wide memory with minimal number of wait states.

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

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

```assembly
PUBLIC QF_irq
EXTERN BSP_irq

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

  ; IRQ exit }}}
  (7) MSR  cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM, no IRQ, but FIQ enabled!
  (8) STMFD sp!,{r0,r1} ; save SPSR and PC on SYS stack
  (9) STMFD sp!,{r2-r3,r12,lr} ; save APCS-clobbered regs on SYS stack
 (10) MOV  r0,sp ; make the sp_SYS visible to IRQ mode
 (11) SUB  sp,sp,#(2*4) ; make room for stacking (r0_SYS, r1_SYS)
 (12) MSR  cpsr_c,#(IRQ_MODE | NO_IRQ) ; IRQ mode, IRQ disabled
 (13) STMFD r0!,{r13,r14} ; finish saving the context (r0_SYS,r1_SYS)
 (14) MSR  cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM mode, IRQ disabled
        ; IRQ entry }}

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

  ; IRQ exit {{
 (18) MSR  cpsr_c,#(SYS_MODE | NO_INT) ; make sure IRQ/FIQ are disabled
 (19) MOV  r0,sp ; make sp_SYS visible to IRQ mode
 (20) ADD  sp,sp,#(8*4) ; fake unstacking 8 registers from sp_SYS
 (21) MSR  cpsr_c,#(IRQ_MODE | NO_INT) ; IRQ mode, both IRQ/FIQ disabled
 (22) MOV  sp,r0 ; copy sp_SYS to sp_IRQ
 (23) LDR  r0,[sp,#(7*4)] ; load the saved SPSR from the stack
 (24) MSR  spsr_cxsf,r0 ; copy it into spsr_IRQ
 (25) LDMFD sp,[r0-r3,r12,lr]^ ; unstack all saved USER/SYSTEM registers
 (26) NOP  ; can't access banked reg immediately
 (27) LDR  lr,[sp,#(6*4)] ; load return address from the SYS stack
 (28) MOVS pc,lr ; return restoring CPSR from SPSR
        ; IRQ exit }}}
```
NOTE: As part of the IRQ startup sequence, the ARM processor sets the I bit in the CPSR \((\text{CPSR}[7] = 1)\), but leaves the F bit unchanged (typically cleared), meaning that further IRQs are disabled, but FIQs are not. This means that FIQ can be recognized while the ARM core is in the IRQ mode. This IRQ handler does not disable the FIQ and preemption, and the provided FIQ handler can safely preempt this IRQ until FIQs are explicitly disabled later in the sequence.

(4) Now r0 can be clobbered with the return address from the interrupt that needs to be saved to the SYSTEM stack.
(5) At this point the banked \(l r\_I R Q\) register can be reused to temporarily hold r1 from the SYSTEM context.
(6) Now r1 can be clobbered with the value of spsr\_IRQ register (Saved Program Status Register) that needs to be saved to the SYSTEM stack.
(7) Mode is changed to SYSTEM with IRQ interrupt disabled, but FIQ explicitly enabled. This mode switch is performed to get access to the SYSTEM registers.

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

(8) The SPSR register and the return address from the interrupt (PC after the interrupt) are pushed on the SYSTEM stack.
(9) All registers (except \(r0\) and \(r1\)) clobbered by the AAPCS (ARM Architecture Procedure Call Standard) [ARM 06] are pushed on the SYSTEM stack.
(10) The SYSTEM stack pointer is placed in \(r0\) to be visible in the IRQ mode.
(11) The SYSTEM stack pointer is adjusted to make room for two more registers of the saved IRQ context. By adjusting the SYSTEM stack pointer, the IRQ handler can still keep FIQ enabled without the concern of corrupting the SYSTEM stack space reserved for the IRQ context.
(12) The mode is switched back to IRQ with IRQ interrupt disabled, but FIQ still enabled. This is done to get access to the rest of the context sitting in the IRQ-banked registers.
(13) The context is entirely saved by pushing the original \(r0\) and \(r1\) (still sitting in the banked IRQ Registers \(r14\_IRQ\) and \(r13\_IRQ\), respectively) to the SYSTEM stack. At this point the saved SYSTEM stack frame contains 8 registers and looks as follows (this is exactly the ARM v7-M interrupt stack frame [ARM 06]):

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

(14) The mode is switched once more to SYSTEM with IRQ disabled and FIQ enabled. Please note that the stack pointer \(sp\_SYS\) points to the top of the stack frame, because it has been adjusted after the first switch to the SYSTEM mode at line (11).
(15-17) The board-specific function \(\text{BSP}\_\text{irq}()\) is called to perform the interrupt processing at the application-level. Please note that \(\text{BSP}\_\text{irq}()\) is now a regular C function in ARM or Thumb. Typically, this function uses the silicon-vendor specific interrupt controller (such as the Atmel AIC) to vector into the current interrupt.

NOTE: The \(\text{BSP}\_\text{irq}()\) function is entered with IRQ disabled (and FIQ enabled), but it can internally unlock IRQs, if the MCU is equipped with an interrupt controller that performs prioritization of IRQs in hardware.
All interrupts (IRQ and FIQ) are locked to execute the following instructions atomically.

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

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

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

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

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

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

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

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

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

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

### The FIQ “Wrapper” Function in Assembly

NOTE: Generally, you should avoid, if only possible, using the FIQ as general purpose interrupt, because the FIQ interrupt is NOT prioritized by the interrupt controller in most ARM chips (see Figure 9). Handling the FIQ as general purpose interrupt is actually more complicated (and thus actually more expensive) than handling of IRQ-type interrupts that typically are prioritized in the interrupt controller.

Perhaps the best use of the FIQ is as a Non-Maskable-Interrupt (NMI) for handling very special functions. To use FIQ as a NMI, you must modify the critical section definition. Specifically, you would need to set only the I bit in the CPSR (see Listing 5):

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

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

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

#### Listing 7 The QF_fiq assembly wrapper for the “vanilla” QF port defined in qfn_port.s.

```
SECTION .textrw:DATA:NOROOT(2)
PUBLIC   QF_fiq
EXTERN   BSP_fiq
```
The QF_fiq() “wrapper” shown in Listing 7 is very similar to the IRQ wrapper (Listing 6), except the FIQ mode is used instead of the IRQ mode. The following comments explain only the slight, but important differences in disabling interrupts and the responsibilities of the C-level handler BSP_fiq() function.

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

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

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

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

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

### 4.2.6 Other ARM Exception “Wrapper” Functions in Assembly

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

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

#### Listing 8 ARM Exception “Wrapper” Functions in Assembly defined in qfn_port.s.

```assembly
;------------------------------------------------------------------
; Exception assembler wrappers
;------------------------------------------------------------------
PUBLIC QF_reset
PUBLIC QF_undef
PUBLIC QF_swi
PUBLIC QF_pAbort
PUBLIC QF_dAbort
PUBLIC QF_reserved
EXTERN Q_onAssert

SECTION .text:CODE:NORoot(2)
CODE32

QF_reset:
   LDR    r0,=Cstring_reset
   B      QF_except

QF_undef:
   LDR    r0,=Cstring_undef
   B      QF_except

QF_swi:
   LDR    r0,=Cstring_swi
   B      QF_except

QF_pAbort:
   LDR    r0,=Cstring_pAbort
   B      QF_except
(1) QF_dAbort:
(2)   LDR    r0,=Cstring_dAbort
(3)   B      QF_except

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

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

END

(1)  All exceptions are handled uniformly, such as the PREFETCH ABORT exception.
(2)  The first argument to the Q_onAssert() callback function is prepared in r0. This argument is a pointer to the C-string with the name of the exception.
(3-4) The common exception code is handled in at the QF_except label.
(5)  The address of the exception is saved in r1, which is the second argument to the Q_onAssert() callback function.
(6)  The mode is switched to SYSTEM with both IRQ and FIQ interrupts disabled.
(7-9) The Q_onAssert() callback function is called. Because the call happens via the BX instruction, the Q_onAssert() function can be in ARM or THUMB.

4.2.7 QP-nano Idle Processing Customization in QF_onIdle()

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

The following Listing 9 shows the QF_onIdle() callback that puts AT91 MCU into the idle power-saving mode.

Listing 9 QF_onIdle() for the non-preemptive QF-nano configuration

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

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

(1)  The QF_onIdle() function is called with interrupts locked and must unlock interrupts internally.
(2) Writing 1 to the Power Module Control SCDR register stops the CPU clock. Note that interrupts remain locked at the ARM core level. The interrupts are not disabled at the interrupt controller level.

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

This section describes how to use QP-nano with ARM7/ARM9-based MCUs with fully preemptive QK-nano real-time kernel. The benefit is very fast, fully deterministic task-level response and that responsiveness of the high-priority tasks (active objects) will be virtually insensitive to any changes in the lower-priority tasks. The downside is bigger RAM requirement for the stack (see Chapter 10 in [PSiCC2]). Also, as with any preemptive kernel, you must be very careful to avoid any sharing of resources among concurrently executing active objects, or if you do need to share resources, you need to protect them with the QK-nano priority-ceiling mutex (again see see Chapter 10 in [PSiCC2]).

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

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

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

5.1 Configuration and Customizing QP-nano

You configure and customize QP-nano through the header file qpn_port.h, which is included by the QP-nano source files (qepn.c, qfn.c, and qkn.c) as well as in all your application C modules. The following Listing 10 shows the qpn_port.h header file for the QK-nano port. Except for the highlighted fragments, the listing is identical as in the non-preemptive case (Listing 4)

Listing 10 qpn_port.h header file for the preemptive QK-nano configuration

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

... (the same as in the non-preemptive configuration in Listing 4)

void QF_reset(void);
void QF_undef(void);
void QF_sw1(void);
void QF_pAbort(void);
void QF_dAbort(void);
void QF_reserved(void);

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

5.2 Handling Interrupts in QK-nano

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

5.2.1 The IRQ “Wrapper” Function for QK

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

Listing 11 The \texttt{QK_irq} assembly wrapper for the QK port defined in \texttt{qkn_port.s}.

```assembly
SECTION .textrw:CODE:NOROOT(2)
PUBLIC QK_irq
EXTERN BSP_irq
EXTERN QK_intNest_, QK_schedule_
CODE32
QK_irq:
; IRQ entry {{{
    MOV r13,r0              ; save r0 in r13_IRQ
    SUB r0,lr,#4            ; put return address in r0_SYS
    MOV lr,r1               ; save r1 in r14_IRQ (lr)
    MRS r1,spsr             ; put the SPSR in r1_SYS
    MSR cpsr_c,#(SYS_MODE | NO_IRQ) ; SYSTEM, no IRQ, but FIQ enabled!
    STMFD sp!,{r0,r1}        ; save SPSR and PC on SYS stack
    STMFD sp!,{r2-r3,r12,lr} ; save APCS-clobbered regs on SYS stack
    MOV r0,sp               ; make the sp_SYS visible to IRQ mode
    SUB sp,sp,#(2*4)        ; make room for stacking (r0_SYS, r1_SYS)
    MSR cpsr_c,#(IRQ_MODE | NO_IRQ) ; IRQ mode, IRQ disabled
    STMFD r0!,{r13,r14}     ; finish saving the context (r0_SYS,r1_SYS)
    MSR cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
    ; IRQ entry }}}
    (1) LDR r0,=QK_intNest_  ; load address in already saved r0
    (2) LDRB r12,[r0]       ; load original QK_intNest_ into the saved r12
    (3) ADD r12,r12,#1      ; increment the nesting level
    (4) STRB r12,[r0]       ; store the value in QK_intNest_
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.
Listing 11 shows the QK_irq() "wrapper" function in assembly. Please note that the interrupt entry (delimited with the comments “IRQ entry {{{" and “IRQ entry }}}}”) and interrupt exit (delimited with the comments “IRQ exit {{{" and “IRQ exit }}}}”) are identical as in the QF_irq() “wrapper” function. Please refer to the notes after 4.2.3 for detailed explanation of these sections of the code. The following notes explain only the QK-specific additions made in the QK_irq() “wrapper” function.

1-4 The QK interrupt nesting level QK_intNest_ is incremented and saved.
5 Interrupts are locked to access the QK interrupt nesting level QK_intNest_ and to call the QK scheduler.
6-9 The current QK interrupt nesting level QK_intNest_ is decremented. Please note the use of the special version of SUBS in line (8), which sets the test flags if the nesting level QK_intNest_ drops to zero.
(10) The branch is taken when the QK interrupt nesting level `QK_intNest_` is not zero. In this case the QK scheduler should not be called, because the IRQ is returning to a preempted IRQ, rather than the task level.

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

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

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

(16-18) Otherwise scheduling is necessary, so the function `QK_sched_()` is called via the `BX` instruction.

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

5.2.2 The FIQ “Wrapper” Function for QK-nano

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

Listing 12 The QK_fiq assembly wrapper for the QK port defined in qk_port.s

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

QK_fiq:
; FIQ entry {{
    MOV r13,r0              ; save r0 in r13_FIQ
    SUB r0,lr,#4            ; put return address in r0_SYS
    MOV lr,r1               ; save r1 in r14_FIQ (lr)
    MRS r1,spsr             ; put the SPSR in r1_SYS
    MSR cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
    STMFD sp!,{r0,r1}       ; save SPSR and PC on SYS stack
    STMFD sp!,{r2-r3,r12,lr} ; save APCS-clobbered regs on SYS stack
    MOV r0,sp               ; make the sp_SYS visible to FIQ mode
    SUB sp,sp,#(2*4)        ; make room for stacking (r0_SYS, r1_SYS)
    MSR cpsr_c,#(FIQ_MODE | NO_INT) ; FIQ mode, IRQ/FIQ disabled
    STMFD r0!,{r13,r14}     ; finish saving the context (r0_SYS, r1_SYS)
    MSR cpsr_c,#(SYS_MODE | NO_INT) ; SYSTEM mode, IRQ/FIQ disabled
; FIQ entry }}}
    LDR r0,=QK_intNest_     ; load address in already saved r0
    LDRB r12,[r0]            ; load original QK_intNest_ into the saved r12
    ADD r12,r12,#1          ; increment interrupt nesting
    STRB r12,[r0]            ; store the value in QK_intNest_

; NOTE:
; Because FIQ is typically NOT prioritized by the interrupt controller
; BSP_fiq must not enable IRQ/FIQ to avoid priority inversions!
; ```
Listing 12 shows the QK_fiq() “wrapper” function in assembly. Please note that the interrupt entry (delimited with the comments “FIQ entry {{{” and “FIQ entry }}}) and interrupt exit (delimited with the comments “FIQ exit {{{” and “FIQ exit }}}) are almost identical as in the QF_fiq() “wrapper” function. Please refer to the notes after 4.2.5 for detailed explanation of these sections of the code. The following notes explain only the QK-specific additions made in the QK_fiq() “wrapper” function.

(1) The interrupted status register SPSR is loaded from the stack.
(2) The mode bits of the interrupted status register are isolated in r0.
(3) The mode bits of the interrupted status register are compared against the IRQ mode.
(4) Branch is taken if the interrupted mode was IRQ, which means that the FIQ preempted IRQ. In this case, the QK scheduler should not be called.
NOTE: FIQ can always preempt an IRQ in the ARM system architecture, because the F bit is not set in hardware in the IRQ startup sequence. Consequently, the FIQ can preempt the IRQ before the QK_intNest_ is incremented. The code in Listing 12(1-4) detects this situation and prevents calling the QK scheduler.

5.3 Idle Loop Customization in the QK Port

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

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

The following Listing 13 shows an example implementation of QK_onIdle() for the AT91SAM7 MCU. Other ARM-based embedded microcontrollers (e.g., LPC chips from NXP) handle the power-saving mode very similarly.

Listing 13 QK_onIdle() callback for the AT91SAM7 MCU.

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

<table>
<thead>
<tr>
<th>Document</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>[devkitARM] devkitARM download from the devkitPro project at SourceForge.net</td>
<td><a href="https://sourceforge.net/projects/devkitpro/files/devkit">https://sourceforge.net/projects/devkitpro/files/devkit</a> ARM</td>
</tr>
<tr>
<td>[Insight-GDB] Insight-GDB download from the devkitPro project at SourceForge.net</td>
<td><a href="https://sourceforge.net/projects/devkitpro/files/Insight">https://sourceforge.net/projects/devkitpro/files/Insight</a></td>
</tr>
<tr>
<td>[AN-PELICAN] Quantum Leaps Application Note “PEdestrian Light CONtrolled Crossing Example”</td>
<td><a href="http://www.state-machine.com/resources/-">http://www.state-machine.com/resources/-</a> AN_PELICAN.pdf</td>
</tr>
</tbody>
</table>

Copyright © Quantum Leaps, LLC. All Rights Reserved.
7 Contact Information

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

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