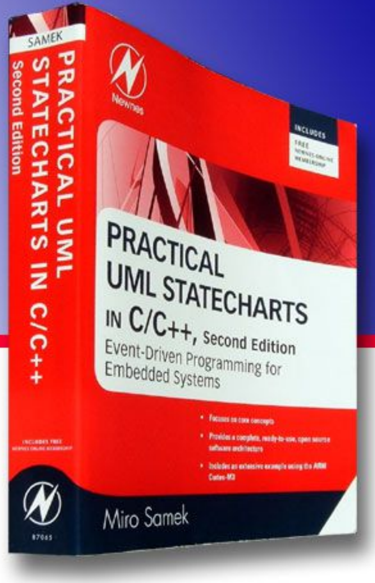
quantum® LeaPs
innovating embedded systems


SAMEK
PRACTICAL UML STATECHARTS IN C/C++
Second Edition
PRACTICAL UML STATECHARTS IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems
Miro Samek

QDK™ ARM-Cortex STM32 with GNU

Document Revision E
September 2012

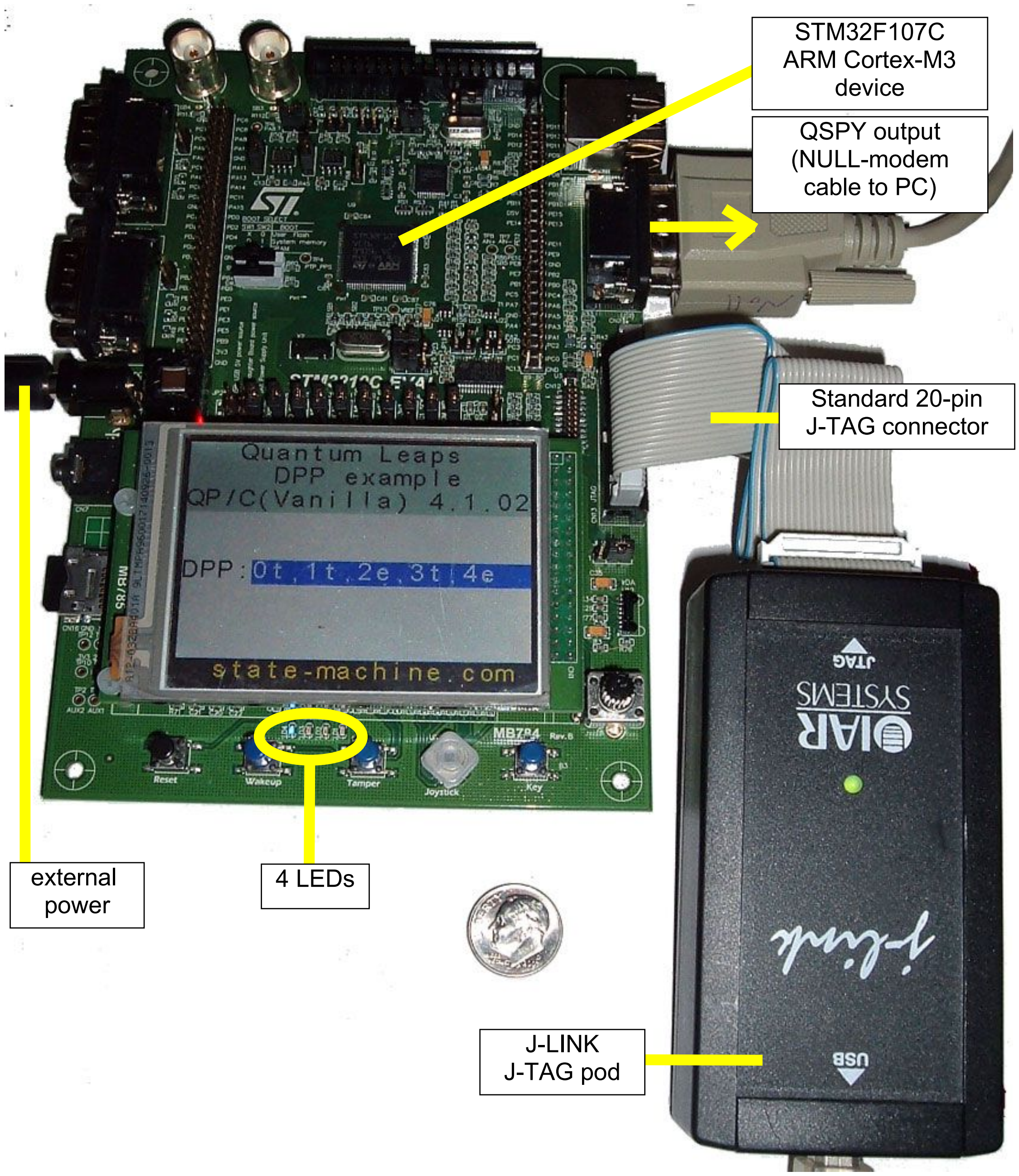
www.quantum-leaps.com
www.state-machine.com

Table of Contents

1 Introduction

This **QP Development Kit**™ (QDK) describes how to use the QP/C™ and QP™/C++ state machine frameworks version **4.5.02** or higher on the ARM Cortex-M3 based STM32 microcontrollers with the GNU toolchain. This QDK uses the SAM3210C-EVAL development board from STMicroelectronics shown in Figure 1.

Figure 1 SAM3210C-EVAL evaluation board and J-LINK pod (bottom)



The actual hardware/software used is as follows:

1. **Sourcery CodeBench** version 2011.09-60 http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview/
2. SEGGER J-Link **GDB Server** software v4.08l available for download from www.segger.com/cms/downloads.html
3. **QP/C** or **QP/C++** version **4.5.02** or higher available from www.state-machine.com/downloads.

The examples included in this QDK are based on the standard Dining Philosophers Problem implemented with active objects (see Chapter 9 in [PSiCC2] and the Application Note "Dining Philosopher Problem Example" [AN-DPP]).

NOTE: This QDK is based on the **Application Note "QP™ and ARM-Cortex with GNU"** [QP-Cortex], which describes the QP source code common for all ARM Cortex-M3 and Cortex-M0 cores. This QDK Manual pertains both to C and C++ versions of the QP™ state machine frameworks. Most of the code listings in this document refer to the QP/C version. Occasionally the C code is followed by the equivalent C++ implementation to show the C++ differences whenever such differences become important.

1.1 About the ARM-Cortex Port

In contrast to the traditional ARM7/ARM9 cores, ARM-Cortex cores contain such standard components as the Nested Vectored Interrupt Controller (NVIC) and the System Timer (SysTick). With the provision of these standard components, it is now possible to provide fully **portable** system-level software for ARM-Cortex. Therefore, this QP port to ARM-Cortex can be much more complete than a port to the traditional ARM7/ARM9 and the software is guaranteed to work on any ARM-Cortex silicon.

The non preemptive cooperative kernel implementation is very simple on ARM-Cortex, perhaps simpler than any other processor, mainly because Interrupt Service Routines (ISRs) are regular C-functions on ARM-Cortex.

However, when it comes to handling preemptive multitasking, ARM-Cortex is a unique processor unlike any other. The ARM-Cortex hardware has been designed with traditional blocking real-time kernels in mind, and implementing a simple run-to-completion preemptive kernel (such as the QK preemptive kernel described in Chapter 10 in [PSiCC2]) is a bit tricky. Application Note "QP and ARM-Cortex with GNU" [QP-Cortex] describes in detail the QK implementation on ARM-Cortex.

1.2 Cortex Microcontroller Software Interface Standard (CMSIS)

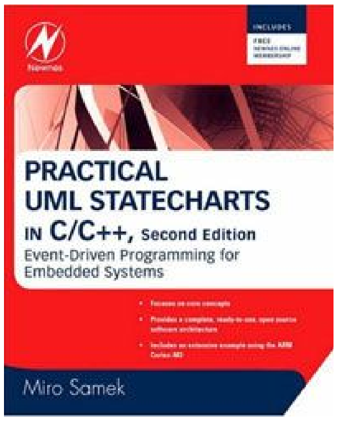
The ARM-Cortex examples provided with this Application Note are compliant with the Cortex Microcontroller Software Interface Standard (CMSIS).

1.3 What's Included in the QDK

The code accompanying this QDK contains the separate C and C++ example, each containing the standard startup code, linker script, makefiles, board support package (BSP) and two versions of the Dining Philosopher Problem (DPP) example for two different kernels available in QP. The DPP application is described in Chapter 7 of [PSiCC2] as well as in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08] (included in the QDK distribution).

NOTE: The accompanying code contains the compiled QP libraries, so that you can build the example application, but to avoid repetitions, the platform-independent source code of QP frameworks is **not** provided in this QDK. You need to download the QP source code separately from www.state-machine.com/downloads/.

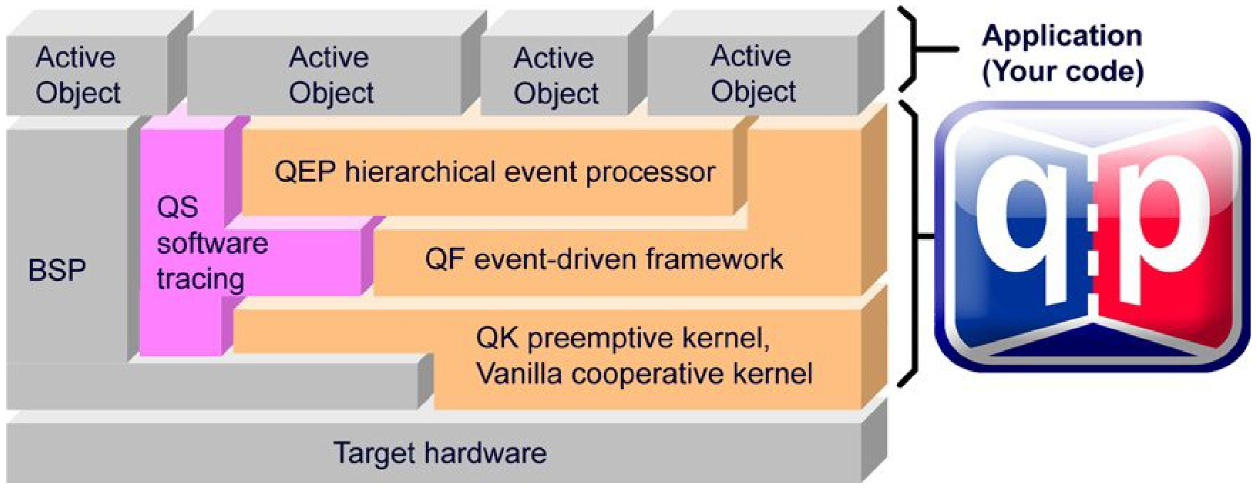
1.4 About QP™



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

As shown in Figure 2, QP consists of a universal UML-compliant event processor (QEP), a portable real-time framework (QF), a tiny run-to-completion kernel (QK), and software tracing instrumentation (QS). Current versions of QP include: QP/C™ and QP/C++™, which require about 4KB of code and a few hundred bytes of RAM, and the ultra-lightweight QP-nano, which requires only 1-2KB of code and just several bytes of RAM.

Figure 2 QP components and their relationship with the target hardware, board support package (BSP), and the application



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

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

1.5 About QM™

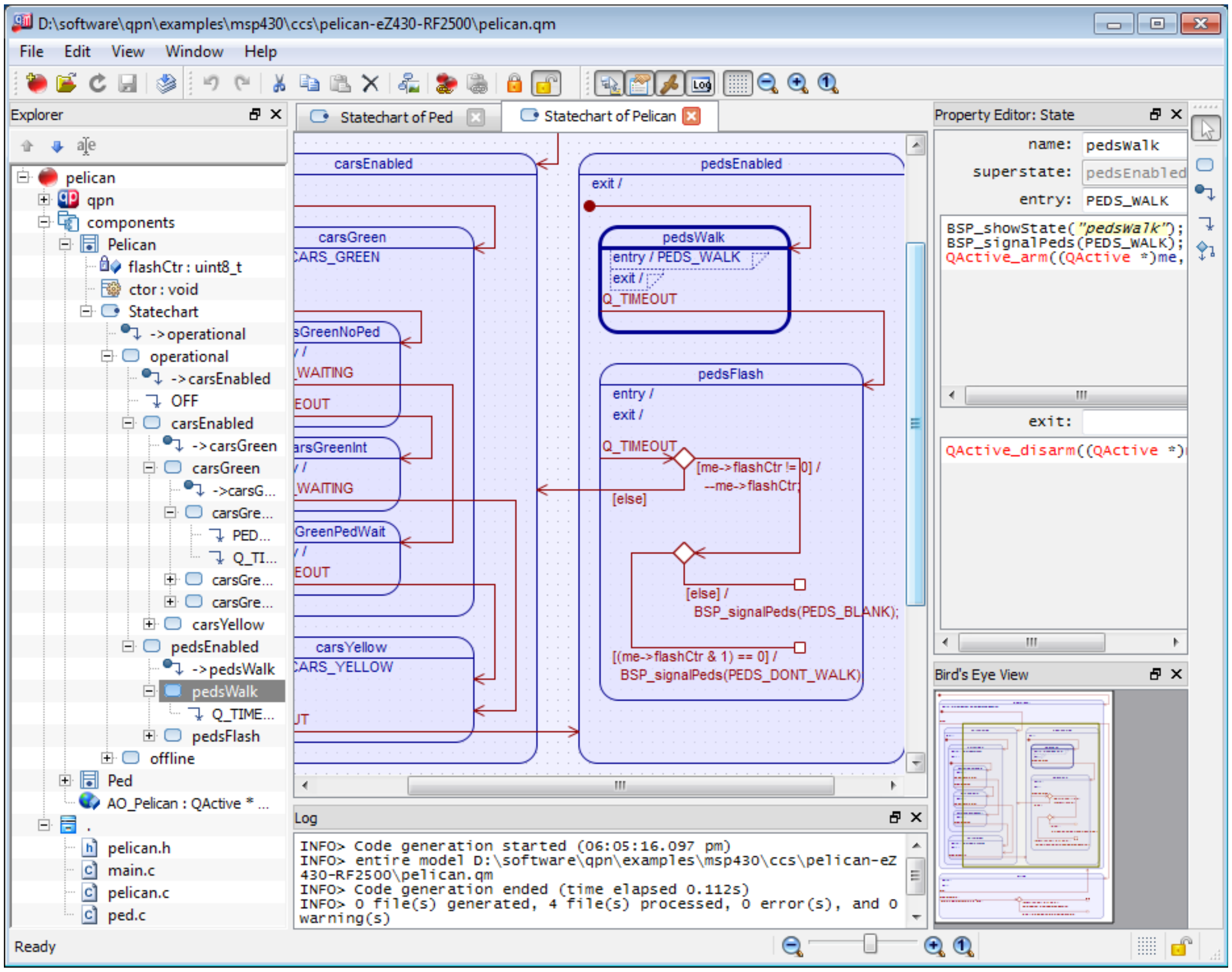
QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

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 for more information about QM™.

The code accompanying this App Note contains three application examples: the Dining Philosopher Problem [AN-DPP], the PEdestrian LIght CONtrolled [AN-PELICAN] crossing, and the “Fly 'n' Shoot” game simulation for the EK-LM3S811 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



```
name:        pedsWalk
superstate:  pedsEnabled
entry:       PEDS_WALK
BSP_showState("pedsWalk");
BSP_signalPeds(PEDS_WALK);
QActive_arm((QActive *)me,
exit:
QActive_disarm((QActive *)
```

```
INFO> Code generation started (06:05:16.097 pm)
INFO> entire model D:\software\qpn\examples\msp430\ccs\pelican-eZ
430-RF2500\pelican.qm
INFO> Code generation ended (time elapsed 0.112s)
INFO> 0 file(s) generated, 4 file(s) processed, 0 error(s), and 0
warning(s)
```

1.6 Licensing QP

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

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

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

1.7 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 Getting Started

The code for the QP port to ARM is available as part of any QP Development Kit (QDK) for ARM-Cortex. The QDKs assume that the generic platform-independent QP™ distribution has been installed. The code of the ARM-Cortex 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 Selected Directories and files of the QP after installing the QDK-ARM-Cortex-GNU. Directories and files shown in bold indicate the elements included in the ARM port.

```
<qp>/                          - QP/C or QP/C++ Root Directory
  |
  +-include/                   - QP public include files
  | +-. . .
  |
  +-ports/                     - QP ports
  | +-arm-cortex/              - ARM-Cortex port
  | | +-vanilla/               - "vanilla" ports
  | | | +-gnu/                 - GNU compiler
  | | | | +-dbg/               - Debug build
  | | | | | +-libqp_cortex-m3_cs.a  - QEP library for Cortex-M3 with CodeSourcery
  | | | | +-rel/               - Release build
  | | | | | +-...              - Release libraries
  | | | | +-spy/               - Spy build
  | | | | | +-...              - Spy libraries
  | | | | +-make_cortex-m3_cs.bat - Batch to build QP for Cortex-M3 with CodeSourcery
  | | | | +-. . .
  | | | | +-qep_port.h         - QEP platform-dependent public include
  | | | | +-qf_port.h          - QF platform-dependent public include
  | | | | +-qs_port.h          - QS platform-dependent public include
  | | | | +-qp_port.h          - QP platform-dependent public include
  | | +-qk/                    - QK (Quantum Kernel) ports
  | | | +-gnu/                 - GNU compiler
  | | | | +-dbg/               - Debug build
  | | | | | +-libqp_cortex-m3_cs.a  - QEP library for Cortex-M3 with CodeSourcery
  | | | | +-rel/               - Release build
  | | | | | +-...              - Release libraries
  | | | | +-spy/               - Spy build
  | | | | | +-...              - Spy libraries
  | | | | +-src/               - Platfom-specific source directory
  | | | | | +-qk_port.s        - Platfom-specific source code for the port
  | | | | +-make_cortex-m3_cs.bat - Batch to build QP for Cortex-M3 with CodeSourcery
  | | | | +-. . .             - Batch scripts to build QP libraries for other Cortex cores
  | | | | +-qep_port.h         - QEP platform-dependent public include
  | | | | +-qf_port.h          - QF platform-dependent public include
  | | | | +-qs_port.h          - QS platform-dependent public include
  | | | | +-qp_port.h          - QP platform-dependent public include
  |
  +-examples/                  - QP examples
  | +-arm-cortex/              - ARM-Cortex port
  | | +-vanilla/               - port for the "Vanilla" cooperative kernel
  | | | +-gnu/                 - GNU compiler
  | | | | +-.metadata          - directory with Eclipse workspace
  | | | | +-dpp-stm3210c-eval  - DPP example for the STM3210C-EVAL evaluation board
  | | | | | +-cmsis/           - CMSIS files
  | | | | | | +-core_cm3.h - Cortex-M3 core interface
  | | | | | | +-core_cm3.c - Cortex-M3 core interface implementation
  | | | | | | +-startup_stm32f10x_cl.c - Startup code for STM32F10x Connectivity Line
  | | | | | | +-stm32f10x.h - STM32F10x microcontroller peripherals interface
  | | | | | | +-system_stm32f10x.h - STM32F10x microcontroller system interface
  | | | | | | +-system_stm32f10x.c - STM32F10x microcontroller system implementation
  | | | | | +-STM32F10x/   - STMicroelectrinics drivers for the STM32F10x devices
  | | | | | | +-inc/       - Driver header files
  | | | | | | +-src/       - Driver source files
  | | | | | +-STM3210C-EVAL/ - ST support files for STM3210C-EVAL board
  | | | | | |
  | | | | | +-dbg/         - Debug build (runs from RAM)
  | | | | | | +-dpp.elf    - executable image
  | | | | | +-rel/         - Release build (runs from Flash)
  | | | | | | +-dpp.elf    - executable image
  | | | | | +-spy/         - Spy build (runs from RAM)
  | | | | | | +-dpp.elf    - executable image (instrumented with QSpy)
  | | | | | +-Makefile     - Makefile for the DPP application example
  | | | | | +-insight-jlink.bat- Batch file to launch the Insight with J-Link
  | | | | | +-jlink.dgb        - GDB configuration file for J-Link
  | | | | | +-bsp.h        - Board Support Package include file
  | | | | | +-bsp.c        - Board Support Package implementation
  | | | | | +-dpp.h        - Source code of the DPP application
  | | | | | +-main.c       - Source code of the DPP application
  | | | | | +-philo.c      - Source code of the DPP application
  | | | | | +-table.c      - Source code of the DPP applicationn
  | | | | | +-stm32f10x.ld - GNU linker command file for the STM32F10x devices
  | | | | | +-stm32f10x_conf.h - STMicroelectrinics driver library configuration
  | | | |
  | | +-qk/                  - QK preemptive kernel port
  | | | +-gnu/               - GNU compiler
  | | | | +-.metadata        - directory with Eclipse workspace
  | | | | +-dpp-qk-stm3210c-eval - DPP example for the STM3210C-EVAL evaluation board
  | | | | | +-. . .        - the same files as in qk/gnu/dpp-stm3210c-eval
  |   . . .
```

2.1 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for QEP, QF, QS, and QK are provided inside the `<qp>\ports\` directory (see Listing 1), so you don't need to re-build the libraries to get started. This section describes steps you need to take to rebuild the libraries yourself.

NOTE: To avoid repetitions, the platform-independent source code of QP frameworks is not provided in this application note. You need to download the QP source code separately from www.state-machine.com/downloads/.

The code distribution contains the batch file `make_<core>_cs.bat` for building all the libraries located in the `<qp>\`**ports**`arm-cortex\...` directory. The batch file supports building three configurations: Debug (default), Release, and Spy. You choose the build configuration by providing a target to the `make_cortex-m3_cs.bat` utility. The default target is "`dbg`". Other targets are "`rel`", and "`spy`" respectively. The following table summarizes the targets accepted by `make_<core>_cs.bat`.

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

Build Configuration	Build command
Debug (default)	`make_cortex-m3_cs.bat`
Release	`make_cortex-m3_cs.bat rel`
Spy	`make_cortex-m3_cs.bat spy`

For example, to build the debug version of all the QP libraries for the Cortex-M3 core, with the GNU compiler, QK kernel, you open a console window on a Windows PC, change directory to `<qp>\`**ports**`\-arm-cortex\qk\gnu\`, and invoke the batch by typing at the command prompt the following command:

```
make_cortex-m3_dk.bat
```

The build process should produce the QP library in the location: `<qp>\`**ports**`\arm-cortex\qk\gnu\dbg\`. The `make_<core>_cs.bat` files assume that the CodeSourcery toolset has been installed in the directory `C:\tools\CodeSourcery`.

NOTE: The batch file uses the environment variable `GNU_ARM` for the location of the GNU-ARM toolset. If this environment variable is not defined, the batch file uses the default location `C:\tools\CodeSourcery`.

In order to take advantage of the QS ("spy") instrumentation, you need to build the QS version of the QP libraries. You achieve this by invoking the `make_cortex-m3_dk.bat` utility with the "spy" target, like this:

```
make_cortex-m3_cs.bat spy
```

The make process should produce the QP library in the directory: `<qp>\ports\arm-cortex\-vanilla\gnu\`**spy**.

2.2 Building the Examples

2.2.1 Building the Examples from Command Line

The example directory `<qp>\examples\arm-cortex\vanilla\gnu\dpp-stm3210c-eval\` contains the `Makefile` you can use to build the application. The `Makefile` supports three build configurations: Debug (default), Release, and Spy. You choose the build configuration by defining the `CONF` symbol at the command line, as shown in the table below.

Table 2 Make targets for the Debug, Release, and Spy software configurations

Build Configuration	Build command
Debug (default)	`make`
Release	`make CONF=rel`
Spy	`make CONF=spy`
Clean the Debug configuration	`make clean`
Clean the Release configuration	`make CONF=rel clean`
Clean the Spy configuration	`make CONF=spy clean`

2.3 Software Tracing with Q-SPY

If you load the Spy configuration to the target and connect a serial NULL-cable between the USART connector on the board and your PC, you could launch the QSPY host utility to observe the software trace output in the human-readable format. You launch the QSPY utility on a Windows PC as follows. (1) Change the directory to the QSPY host utility `<qp>\tools\qspy\win32\mingw\rel` and execute:

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

3 Interrupt Vector Table and Startup Code

ARM-Cortex requires you to place the initial Main Stack pointer and the addresses of all exception handlers and ISRs into the Interrupt Vector Table (IVT) placed in ROM. By the Cortex Microcontroller Software Interface Standard (CMSIS), the IVT and the startup code is located in the CMSIS directory in the `startup_stm32f10x_cl.c` file. The following listing shows the beginning of the IVT for the DPP example, which uses two interrupts: SysTick and EXTI:

ARM-Cortex contains an interrupt vector table (also called the exception vector table) starting usually at address 0x00000000, typically in ROM. The vector table contains the initialization value for the main stack pointer on reset, and the entry point addresses for all exception handlers. The exception number defines the order of entries in the vector table

The IVT for the STM32F10x family of the connectivity line devices, is located in the file `<qp>\examples\arm-cortex\vanilla\gnu\dpp-stm3210c-eval\cmsis\startup_stm32f10x_cl.c`. This IVT can be easily adapted to other ARM-Cortex microcontrollers by modifying the IRQ handlers according to the datasheet of the specific ARM Cortex device.

Listing 2 Startup code and IVT for ARM-Cortex (cmsis\\`startup_stm32f10x_cl.c`)

```
(1) void __attribute__ ((weak)) Reset_Handler(void);
    void __attribute__ ((weak)) NMI_Handler(void);
    void __attribute__ ((weak)) HardFault_Handler(void);
    void __attribute__ ((weak)) MemManage_Handler(void);
    void __attribute__ ((weak)) BusFault_Handler(void);
    void __attribute__ ((weak)) UsageFault_Handler(void);
    void __attribute__ ((weak)) MemManage_Handler(void);
    void __attribute__ ((weak)) SVC_Handler(void);
    void __attribute__ ((weak)) DebugMon_Handler(void);
    void __attribute__ ((weak)) PendSV_Handler(void);
    void __attribute__ ((weak)) SysTick_Handler(void);

                                                          /* external interrupts... */
    void __attribute__ ((weak)) WWDG_IRQHandler(void);
    void __attribute__ ((weak)) PVD_IRQHandler(void);
    void __attribute__ ((weak)) TAMPER_IRQHandler(void);
    . . .
(2) void __attribute__ ((weak)) Spurious_Handler(void);

    /*------------------------------------------------------------------------------
    * weak aliases for each Exception handler to the Spurious_Handler.
    * Any function with the same name will override these definitions.
    */
(3) #pragma weak NMI_Handler                = Spurious_Handler
    #pragma weak MemManage_Handler          = Spurious_Handler
    #pragma weak BusFault_Handler           = Spurious_Handler
    #pragma weak UsageFault_Handler         = Spurious_Handler
    #pragma weak SVC_Handler                = Spurious_Handler
    . . .

    /* exception and interrupt vector table ------------------------------------*/
(4) typedef void (*ExceptionHandler)(void);
(5) typedef union {
        ExceptionHandler handler;
        void            *pointer;
    } VectorTableEntry;
```

```
 (6) extern unsigned __c_stack_top__;

     /*..........................................................................*/
 (7) __attribute__ ((section(".isr_vector")))
 (8) VectorTableEntry const g_pfnVectors[] = {
 (9)     { .pointer = &__c_stack_top__          }, /* initial stack pointer     */
(10)     { .handler = &Reset_Handler            }, /* Reset Handler             */
(11)     { .handler = &NMI_Handler              }, /* NMI Handler               */
         { .handler = &HardFault_Handler        }, /* Hard Fault Handler        */
         { .handler = &MemManage_Handler        }, /* MPU Fault Handler         */
         { .handler = &BusFault_Handler         }, /* Bus Fault Handler         */
         { .handler = &UsageFault_Handler       }, /* Usage Fault Handler       */
         { .handler = &Spurious_Handler         }, /* Reserved                  */
         { .handler = &Spurious_Handler         }, /* Reserved                  */
         { .handler = &Spurious_Handler         }, /* Reserved                  */
         { .handler = &Spurious_Handler         }, /* Reserved                  */
         { .handler = &SVC_Handler              }, /* SVCall Handler            */
         { .handler = &DebugMon_Handler         }, /* Debug Monitor Handler     */
         { .handler = &Spurious_Handler         }, /* Reserved                  */
         { .handler = &PendSV_Handler           }, /* PendSV Handler            */
         { .handler = &SysTick_Handler          }, /* SysTick Handler           */
         /* external interrupts (IRQs) ... */
         { .handler = &WWDG_IRQHandler          }, /* Window Watchdog           */
         . . .
     };
     . . .

     /*..........................................................................*/
(12) void Reset_Handler(void) __attribute__((__interrupt__));
     void Reset_Handler(void) {
         extern int main(void);
         extern int __libc_init_array(void);
(13)     extern unsigned __data_start;    /* start of .data in the linker script */
(14)     extern unsigned __data_end__;      /* end of .data in the linker script */
(15)     extern unsigned const __data_load;  /* initialization values for .data  */
(16)     extern unsigned __bss_start__;    /* start of .bss in the linker script */
(17)     extern unsigned __bss_end__;        /* end of .bss in the linker script */
         unsigned const *src;
         unsigned *dst;

                      /* copy the data segment initializers from flash to RAM... */
(18)     src = &__data_load;
(19)     for (dst = &__data_start; dst < &__data_end__; ++dst, ++src) {
             *dst = *src;
         }

                                                  /* zero fill the .bss segment... */
(20)     for (dst = &__bss_start__; dst < &__bss_end__; ++dst) {
             *dst = 0;
         }
                  /* call all static construcors in C++ (harmless in C programs) */
(21)     __libc_init_array();

                                             /* call the application's entry point */
(22)     main();
```

```
         /* in a bare-metal system main() has nothing to return to and it should
         * never return. Just in case main() returns, the assert_failed() gives
         * the last opportunity to catch this problem.
         */
(23)     assert_failed("startup_stm32f10x_cl", __LINE__);
     }
     /*..........................................................................*/
     void Spurious_Handler(void) __attribute__((__interrupt__));
(24) void Spurious_Handler(void) {
(25)     assert_failed("startup_stm32f10x_cl", __LINE__);
         /* assert_failed() should not return, but just in case the following
         * enless loop will tie up the CPU.
         */
(26)     for (;;) {
         }
     }
}
```

Listing 2 shows the startup code and IVT. The highlights of the startup sequence are as follows:

(1) Prototypes of all exception handlers and interrupt handlers are provided. According to the CMSIS standard, the Cortex exception handlers have names with the suffix `_Hanlder` and the IRQ handlers have the suffix `_IRQHandler`. The 'weak' attribute causes the declaration to be emitted as a weak symbol rather than a global. Weak symbols allow overriding them with identical symbols that are not "weak". When the linker encounters two identical symbols, but one of them is "weak", the linker discards the "weak" and takes the other symbol thus allowing re-definition of the "weak" symbol. Without the "weak" attribute, the linker would report a multiple definition error and would not link the application. Weak symbols are supported for ELF targets when using the GNU assembler and linker.

(2) The `Spurious_Handler()` function handles all unused or reserved exceptions/interrupts. In a properly designed system the spurious exceptions should never occur.

(3) All Cortex exceptions and interrupt handlers are aliased to the `Spurious_Handler()`. However, because all of them are "weak", the application can override them easily. In fact, only the overridden handlers are legal and all non-overridden handlers will call the `Spurious_Handler()` alias.

(4) This `typedef` defines the signature of the ARM Cortex exception as `ExceptionHandler`.

(5) This `union` defines the element of the ARM Cortex Interrupt Vector Table, which can be either an exception handler, or the stack pointer for the very first IVT entry.

(6) The symbol `__c_stack_top__` is provided in the linker script at the end of the stack section. As in ARM Cortex the stack grows towards the low-memory addresses the end of the stack section is the initial top of the stack.

(7) The following IVT is explicitly placed in the `.isr_vector` table to be linked at address 0x0, where the ARM Cortex core expects the IVT.

(8) The ARM Cortex IVT is an array of constant `VectorTableEntry` unions. The `const` keyword is essential to place the IVT in ROM.

(9) The very first entry of the ARM Cortex IVT is the initial stack pointer. Upon the reset, the stack register (r13) is initialized with this value.

(10) The second entry in the ARM Cortex IVT is the reset handler, which now can be a C function because the C-stack is initialized by this time.

NOTE: The `Reset_Handler()` function runs before the proper initialization of the program sections required by the ANSI-C standard.

---

(11) All other ARM Cortex exception and interrupt handlers are initialized in the IVT.

(12) The `Reset_Hanlder()` exception handler performs the low-level initialization required by the C/C++ standard and eventually calls `main()`. Even though ARM Cortex is designed to use regular C functions as exception and interrupt handlers, functions that are used directly as interrupt handlers

should be annotated with `__attribute__((__interrupt__))`. This tells the GNU compiler to add special stack alignment code to the function prologue.

NOTE: Because of a discrepancy between the ARMv7M Architecture and the ARM EABI, it is not safe to use normal C functions directly as interrupt handlers. The EABI requires the stack be 8-byte aligned, whereas ARMv7M only guarantees 4-byte alignment when calling an interrupt vector. This can cause subtle runtime failures, usually when 8-byte types are used [CodeSourcery].

(13-17) These extern declarations refer to the symbols defined in the linker script (see the upcoming section). These linker-generated symbols delimit the `.data`, `.bss` sections.

(18-19) The `.data` section requires copying the initialization values from the load address in ROM to the link address in RAM.

(20) The ANSI-C standard requires initializing the `.bss` section to zero.

(21) The GNU linker-generated function `__libc_init_array()` calls all static constructors, which by the ANSI-C++ standard are required to run before `main()`. The `__libc_init_array()` is harmless in C programs (this function is empty in C programs).

(22) Finally the `main()` entry point is called, which executes the embedded application.

(23) In a bare-metal system `main()` has no operating system to return to and it should never return. However, just in case `main()` returns, the call to `assert_failed()` gives the application the last opportunity to catch this problem. The function `assert_failed()` is used in the STM32 driver library to handle assertion violation.

(24) `Spurious_Handler()` should never occur in a properly designed system. The call to `assert_failed()` gives the application the last opportunity to catch this problem.

(25) If `assert_failed()` ever returns, this endless loop hangs the CPU. There is nothing else to do here, since continuing is not possible.

4 Linker Script

The linker script must match the startup code for all the section names and other linker symbols. The linker script cannot be quite generic, because it must define the specific memory map of the target device. The linker script for the STM32F10x devices is located in the application directory in the file `stm32f10x.ld`, which corresponds to the DPP example application.

Listing 3 Linker script for STM32F10x devices

```
 (1) OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
 (2) OUTPUT_ARCH(arm)
 (3) ENTRY(Reset_Handler)                                     /* entry Point */

 (4) MEMORY {                                    /* memory map of STM32F107C */
 (5)     ROM (rx)  : ORIGIN = 0x08000000, LENGTH = 256K
 (6)     RAM (xrw) : ORIGIN = 0x20000000, LENGTH = 64K
     }

     /* The size of the stack used by the application. NOTE: you need to adjust  */
 (7) STACK_SIZE = 1024;

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

     SECTIONS {

 (9)     .isr_vector : {                  /* the vector table goes FIRST into ROM */
(10)         KEEP(*(.isr_vector))                                 /* vector table */
(11)            . = ALIGN(4);
(12)        } >ROM

(13)        .text : {                                      /* code and constants */
                . = ALIGN(4);
(14)            *(.text)                                 /* .text sections (code) */
                *(.text*)                               /* .text* sections (code) */
(15)            *(.rodata)            /* .rodata sections (constants, strings, etc.) */
                *(.rodata*)          /* .rodata* sections (constants, strings, etc.) */

(16)            KEEP (*(.init))
(17)            KEEP (*(.fini))

                . = ALIGN(4);
(18)            _etext = .;                       /* global symbols at end of code */
            } >ROM

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

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

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

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

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

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

(27)     .heap : {
             __heap_start__ = . ;
             . = . + HEAP_SIZE;
             . = ALIGN(4);
             __heap_end__ = . ;
         } >RAM

(28)     .stack : {
             __stack_start__ = . ;
             . = . + STACK_SIZE;
             . = ALIGN(4);
             __c_stack_top__ = . ;
             __stack_end__ = . ;
         } >RAM

         /* Remove information from the standard libraries */
         /DISCARD/ : {
             libc.a ( * )
             libm.a ( * )
             libgcc.a ( * )
         }
     }
```

Listing 3 shows the linker script for the STM32F10x MCU. The script is identical for C and C++ versions. 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 STM32F10x device. It can contain read-only and executable sections (rx), it starts at 0x08000000 and is 256KB in size.
(6) The region RAM corresponds to the on-chip SRAM of the STM32F10x device. It can contain read-only, read-write and executable sections (rwx), it starts at 0x20000000 and is 64KB in size.
(7) The `STACK_SIZE` symbol determines the sizes of the ARM Main stack. You need to adjust the size for your particular application. The stack size cannot be zero.

NOTE: The QP port to ARM uses only one stack (the Main stack). The Thread stack is not used at all and is not initialized.

(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 `.isr_vector` section contains the ARM Cortex IVT and must be located as the first section in ROM.
(10) This line locates all `.isr_vector` section.
(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 `.text` section is for code and read-only data accessed in place.
(14) The `.text` section groups all individual `.text` and `.text*` sections from all modules.
(15) The section `.rodata` is used for read-only (constant) data, such as look-up tables.
(16-17) 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.

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

(19,20) The `.preinint_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()`.

(21) 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.

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

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

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

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

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

(27) 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()/realloc()` functions. Linking this file saves some 2.8KB of code space compared to the actual implementation of the memory allocating functions.

(28) 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.

5 C/C++ Compiler Options and Minimizing the Overhead of C++

The compiler options for C are defined in the `Makefile` located in the DPP directory. The `Makefile` specifies different options for building debug and release configurations.

5.1 Compiler Options for C

Listing 4 Compiler options used for C project, debug configuration (a) and release configuration (b).

```
       ARM_CORE = cortex-m3

CCFLAGS = -g -c \
(1a)   -mcpu=$(ARM_CORE) \
(2a)   -mthumb \
(3a)   -O \
       -Wall

CCFLAGS = -c \
(1b)   -mcpu=$(ARM_CORE) \
(2a)   -mthumb \
(3b)   -Os \
(4b)   -DNDEBUG \
       -Wall
```

Listing 4 shows the most important compiler options for C, which are:

(1) `-mcpu` option specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can emit when generating assembly code. For ARM Cortex-M3, the `ARM_CORE` symbol is set to cortex-m3.

(2) ARM Cortex cores use exclusively Thumb2 instruction set. For the GNU compiler you need to specify `-mthumb`.

(3) `-O` chooses the optimization level. Release configuration has a higher optimization level `-Os` (2b).

(4) the release configuration defines the macro `NDEBUG`.

5.2 Compiler Options for C++

The compiler options for C++ are defined in the `Makefile` located in the QP/C++ subdirectory. The `Makefile` specifies different options for building the Debug and Release configurations and allows compiling to ARM or Thumb on the module-by-module basis.

Listing 5 Compiler options used for C++ project.

```
CPPFLAGS = -g -gdwarf-2 -c -mcpu=$(ARM_CPU) -mthumb \
(1)    -fno-rtti \
(2)    -fno-exceptions \
       -Wall
```

The C++ `Makefile` located in the directory DPP uses the same options as C discussed in the previous section plus two options that control the C++ dialect:

(1) `-fno-rtti` disables generation of information about every class with virtual functions for use by the C++ runtime type identification features (`dynamic_cast` and `typeid`). Disabling RTTI eliminates several KB of support code from the C++ runtime library (assuming that you don't link with code that

uses RTTI). Note that the `dynamic_cast` operator can still be used for casts that do not require runtime type information, i.e. casts to `void *` or to unambiguous base classes.

(2) `-fno-exceptions` stops generating extra code needed to propagate exceptions, which can produce significant data size overhead. Disabling exception handling eliminates several KB of support code from the C++ runtime library (assuming that you don't link external code that uses exception handling).

5.3 Reducing the Overhead of C++

The compiler options controlling the C++ dialect are closely related to reducing the overhead of C++. However, disabling RTTI and exception handling at the compiler level is still not enough to prevent the GNU linker from pulling in some 50KB of library code. This is because the standard new and delete operators throw exceptions and therefore require the library support for exception handling. (The new and delete operators are used in the static constructor/destructor invocation code, so are linked in even if you don't use the heap anywhere in your application.)

Most low-end ARM-based MCUs cannot tolerate 50KB code overhead. To eliminate that code you need to define your own, non-throwing versions of global new and delete, which is done in the module `mini_cpp.cpp` located in the QP/C++ directory.

Listing 6 The mini_cpp.cpp module with non-throwing new and delete as well as dummy version of __cxa_atexit().

```
     #include <stdlib.h>                   // for prototypes of malloc() and free()
     //..............................................................................
 (1) void *operator new(size_t size) throw() {
         return malloc(size);
     }
     //..............................................................................
 (2) void operator delete(void *p) throw() {
         free(p);
     }

     extern "C" {
     //..............................................................................
 (3) void __cxa_atexit(void (*arg1)(void *), void *arg2, void *arg3) {
     }
     //..............................................................................
     void __cxa_guard_acquire() {
     }
     //..............................................................................
     void __cxa_guard_release() {
     }
     //..............................................................................
     void *__dso_handle = 0;

     }                                                                 // extern "C"
```

Listing 6 shows the minimal C++ support that eliminates entirely the exception handling code. The highlights are as follows:

(1) The standard version of the operator `new` throws `std::bad_alloc` exception. This version explicitly throws no exceptions. This minimal implementation uses the standard `malloc()`.

(2) This minimal implementation of the operator `delete` uses the standard `free()`.

(3) The function `__cxa_atexit()` handles the static destructors. In a bare-metal system this function can be empty because application has no operating system to return to, and consequently the static destructors are never called.

Finally, if you don't use the heap, which you shouldn't in robust, deterministic applications, you can reduce the C++ overhead even further (by about 2.8KB). The module `no_heap.cpp` provides dummy empty definitions of `malloc()` and `free()`:

```
#include <stdlib.h>                    // for prototypes of malloc() and free()

//............................................................................
extern "C" void *malloc(size_t) {
    return (void *)0;
}
//............................................................................
extern "C" void free(void *) {
}
```

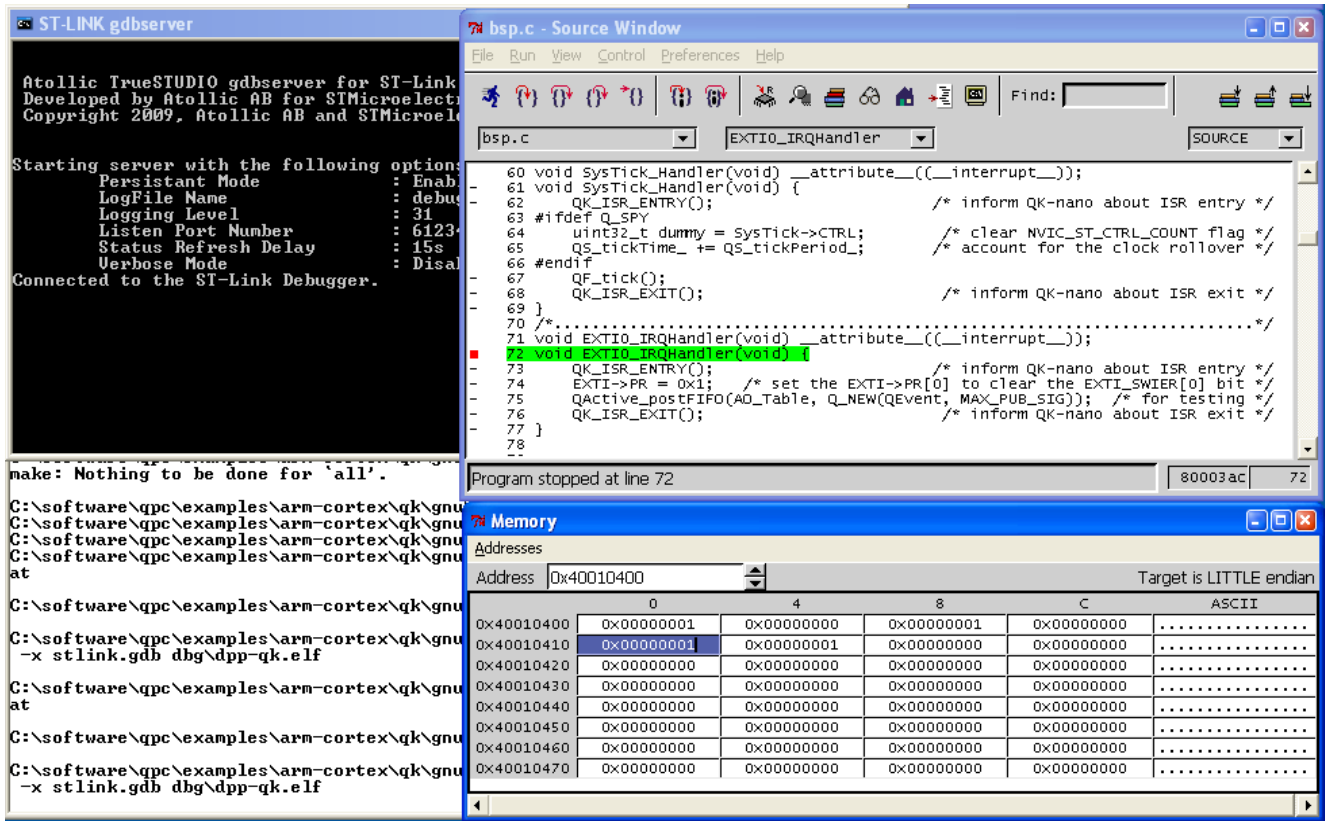
6 Testing QK Preemption Scenarios

The technique described in this section will allow you to trigger an interrupt at any machine instruction and observe the preemptions it causes. The interrupt used for the testing purposes is the EXTI0 interrupt of the STM32 microcontroller. The IRQ handler for this interrupt is shown below:

```
void EXTI0_IRQHandler(void) __attribute__((__interrupt__));
void EXTI0_IRQHandler(void) {
    QK_ISR_ENTRY();                              /* inform QK-nano about ISR entry */
    EXTI->PR = 0x1;    /* set the EXTI->PR[0] to clear the EXTI_SWIER[0] bit */
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG));  /* for testing */
    QK_ISR_EXIT();                                /* inform QK-nano about ISR exit */
}
```

The `EXTI0_IRQHandler`, as all IRQ handlers in the QK-based application, invokes the macros `QK_ISR_ENTRY()` and `QK_ISR_EXIT()`, and also posts an event to the `Table` active object, which has higher priority than any of the Philosopher active object. The NVIC priority of the EXTI0 interrupt is configured higher than the priority of the SysTick handler (see `QF_onStartup()` in `bsp.c`).

Figure 4 Triggering the EXTI0 interrupt from the Insight debugger



```
Atollic TrueSTUDIO gdbserver for ST-Link
Developed by Atollic AB for STMicroelectr
Copyright 2009, Atollic AB and STMicroele

Starting server with the following option
        Persistant Mode              : Enab
        LogFile Name                 : debu
        Logging Level                : 31
        Listen Port Number           : 6123
        Status Refresh Delay         : 15s
        Verbose Mode                 : Disa
Connected to the ST-Link Debugger.
```

```
make: Nothing to be done for `all'.

C:\software\qpc\examples\arm-cortex\qk\gnu
C:\software\qpc\examples\arm-cortex\qk\gnu
C:\software\qpc\examples\arm-cortex\qk\gnu
C:\software\qpc\examples\arm-cortex\qk\gnu
at

C:\software\qpc\examples\arm-cortex\qk\gnu

C:\software\qpc\examples\arm-cortex\qk\gnu
 -x stlink.gdb dbg\dpp-qk.elf

C:\software\qpc\examples\arm-cortex\qk\gnu
at

C:\software\qpc\examples\arm-cortex\qk\gnu

C:\software\qpc\examples\arm-cortex\qk\gnu
 -x stlink.gdb dbg\dpp-qk.elf
```

```
 60 void SysTick_Handler(void) __attribute__((__interrupt__));
 61 void SysTick_Handler(void) {
 62     QK_ISR_ENTRY();                     /* inform QK-nano about ISR entry */
 63 #ifdef Q_SPY
 64     uint32_t dummy = SysTick->CTRL;      /* clear NVIC_ST_CTRL_COUNT flag */
 65     QS_tickTime_ += QS_tickPeriod_;     /* account for the clock rollover */
 66 #endif
 67     QF_tick();
 68     QK_ISR_EXIT();                       /* inform QK-nano about ISR exit */
 69 }
 70 /*..................................................................*/
 71 void EXTI0_IRQHandler(void) __attribute__((__interrupt__));
 72 void EXTI0_IRQHandler(void) {
 73     QK_ISR_ENTRY();                     /* inform QK-nano about ISR entry */
 74     EXTI->PR = 0x1;    /* set the EXTI->PR[0] to clear the EXTI_SWIER[0] bit */
 75     QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG));  /* for testing */
 76     QK_ISR_EXIT();                       /* inform QK-nano about ISR exit */
 77 }
 78
```

Program stopped at line 72 | 80003ac | 72

Address 0x40010400 — Target is LITTLE endian

	0	4	8	C	ASCII
0x40010400	0x00000001	0x00000000	0x00000001	0x00000000	....
0x40010410	0x00000001	0x00000001	0x00000000	0x00000000	....
0x40010420	0x00000000	0x00000000	0x00000000	0x00000000	....
0x40010430	0x00000000	0x00000000	0x00000000	0x00000000	....
0x40010440	0x00000000	0x00000000	0x00000000	0x00000000	....
0x40010450	0x00000000	0x00000000	0x00000000	0x00000000	....
0x40010460	0x00000000	0x00000000	0x00000000	0x00000000	....
0x40010470	0x00000000	0x00000000	0x00000000	0x00000000	....

Figure 4 shows how to trigger the EXTI0 interrupt from the Insight debugger. From the debugger you need to first open the memory window (see the lower-right window in Figure 4). You select the EXTI address 0x40010400. To trigger the EXTI0 interrupt you need to write 1 to the address 0x40010410 by typing 1 in the indicated field, and pressing the Enter key.

The general testing strategy is to break into the application at an interesting place for preemption, set breakpoints to verify which path through the code is taken, and trigger the EXTI0 interrupt. Next, you need to free-run the code (don't use single stepping) so that the NVIC can perform prioritization. You observe the order in which the breakpoints are hit. This procedure will become clearer after a few examples.

6.1 Interrupt Nesting Test

The first interesting test is verifying the correct tail-chaining to the PendSV exception after the interrupt nesting occurs. To test this scenario, you place a breakpoint inside the `EXTI0_IRQHandler()` and also inside the `SysTick_Handler()` ISR. When the breakpoint is hit, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window) and also another breakpoint on the first instruction of the `PendSV_Handler` handler. Next you trigger the EXTI0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `EXTI0_Handler()` function, which means that EXTI0 IRQ preempted the SysTick handler.
2. The second breakpoint hit is the one in the `SysTick_Handler()`, which means that the SysTick handler continues after the EXTI0 IRQ completes.
3. The last breakpoint hit is the one in `QK_PendSV()` exception handler, which means that the PendSV exception is tail-chained only after all interrupts are processed.

You need to remove all breakpoints before proceeding to the next test.

6.2 Task Preemption Test

The next interesting test is verifying that tasks can preempt each other. You set a breakpoint anywhere in the `Philosopher` state machine code. You run the application until the breakpoint is hit. After this happens, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window). You also place a breakpoint inside the `EXTI0_Handler()` interrupt handler and on the first instruction of the `PendSV_Handler()` handler. Next you trigger the EXTI0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `EXTI0_Handler()` function, which means that EXTI0 IRQ preempted the `Philospher` task.
2. The second breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is activated before the control returns to the preempted `Philosopher` task.
3. After hitting the breakpoint in `PendSV_Handler` handler, you single step into the `QK_scheduler_()`. You verify that the scheduler invokes a state handler from the Philosopher state machine. This proves that the `Table` task preempts the `Philosopher` task.
4. After this you free-run the application and verify that the next breakpoint hit is the one inside the `Philosopher` state machine. This validates that the preempted task continues executing only after the preempting task (the `Table` state machine) completes.

6.3 Other Tests

Other interesting tests that you can perform include changing priority of the GPIOA interrupt to be lower than the priority of SysTick to verify that the PendSV is still activated only after all interrupts complete.

In yet another test you could post an event to `Philosopher` active object rather than `Table` active object from the `EXTI0_Handler()` function to verify that the QK scheduler will not preempt the `Philosopher` task by itself. Rather the next event will be queued and the `Philosopher` task will process the queued event only after completing the current event processing.

7 QS Software Tracing Instrumentation

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

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

QS can be configured to send the real-time data out of the serial port of the target device. On the STM32F10x MCU, QS uses the built-in USART2 to send the trace data out. The STM3210C-EVAL board has the USART2 connected to the DB-9 connector close to the Ethernet connector (see Figure 1), so the QSPY host application can conveniently receive the trace data on the host PC. The QS platform-dependent implementation is located in the file `bsp.c` and looks as follows:

Listing 7 QSpy implementation to send data out of the USART2 of the STM32F10x MCU.

```
(1) #ifdef Q_SPY
(2)     QSTimeCtr QS_tickTime_;
        QSTimeCtr QS_tickPeriod_;

(3)     #define QS_BUF_SIZE   (2*1024)
(4)     #define QS_BAUD_RATE  115200

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

(6) uint8_t QS_onStartup(void const *arg) {
(7)     static uint8_t qsBuf[QS_BUF_SIZE];            /* buffer for Quantum Spy */
(8)     QS_initBuf(qsBuf, sizeof(qsBuf));

                                          /* enable USART2 and GPIOA/AFIO clocks */
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_USART2, ENABLE);
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO,
                               ENABLE);
        GPIO_PinRemapConfig(GPIO_Remap_USART2, ENABLE);

                                            /* configure GPIOD.5 as push-pull... */
        GPIO_InitTypeDef  gpio_init;
        gpio_init.GPIO_Pin   = GPIO_Pin_5;
        gpio_init.GPIO_Speed = GPIO_Speed_50MHz;
        gpio_init.GPIO_Mode  = GPIO_Mode_AF_PP;
        GPIO_Init(GPIOD, &gpio_init);
                                       /* configure GPIOD.6 as input floating... */
        gpio_init.GPIO_Pin   = GPIO_Pin_6;
        gpio_init.GPIO_Mode  = GPIO_Mode_IN_FLOATING;
        GPIO_Init(GPIOD, &gpio_init);

        USART_InitTypeDef usart_init;
        usart_init.USART_BaudRate            = QS_BAUD_RATE;
        usart_init.USART_WordLength          = USART_WordLength_8b;
        usart_init.USART_StopBits            = USART_StopBits_1;
        usart_init.USART_Parity              = USART_Parity_No ;
        usart_init.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
        usart_init.USART_Mode                = USART_Mode_Tx;
        USART_Init(USART2, &usart_init);

        USART_ClockInitTypeDef usart_clk_init;
        usart_clk_init.USART_Clock               = USART_Clock_Disable;
        usart_clk_init.USART_CPOL                = USART_CPOL_Low;
        usart_clk_init.USART_CPHA                = USART_CPHA_2Edge;
        usart_clk_init.USART_LastBit             = USART_LastBit_Disable;
        USART_ClockInit(USART2, &usart_clk_init);

        USART_Cmd(USART2, ENABLE);                             /* enable USART2 */

        QS_tickPeriod_ = (QSTimeCtr)(SystemFrequency_SysClk / BSP_TICKS_PER_SEC);
        QS_tickTime_ = QS_tickPeriod_;        /* to start the timestamp at zero */

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

        . . .
        QS_FILTER_OFF(QS_QF_ACTIVE_ADD);
        QS_FILTER_OFF(QS_QF_ACTIVE_REMOVE);
        . . .

    //    QS_FILTER_OFF(QS_QK_MUTEX_LOCK);
    //    QS_FILTER_OFF(QS_QK_MUTEX_UNLOCK);
        QS_FILTER_OFF(QS_QK_SCHEDULE);

        return (uint8_t)1;                                    /* return success */
    }
    /*..........................................................................*/
 (9) void QS_onCleanup(void) {
    }
    /*..........................................................................*/
(10) void QS_onFlush(void) {
        uint16_t b;
        while ((b = QS_getByte()) != QS_EOD) { /* next QS trace byte available? */
            while ((USART2->SR & USART_FLAG_TXE) == 0) { /* while TXE not empty */
            }
            USART2->DR = (b & 0xFF);                /* put into the DR register */
        }
    }
    /*..........................................................................*/
(11) QSTimeCtr QS_onGetTime(void) {            /* invoked with interrupts locked */
(12)     if ((SysTick->CTRL & 0x00010000) == 0) {                 /* COUNT no set? */
(13)         return QS_tickTime_ - (QSTimeCtr)SysTick->VAL;
        }
(14)     else {      /* the rollover occured, but the SysTick_ISR did not run yet */
(15)         return QS_tickTime_ + QS_tickPeriod_ - (QSTimeCtr)SysTick->VAL;
        }
    }
    #endif                                                             /* Q_SPY */
```

(1) The QS instrumentation is enabled only when the macro `Q_SPY` is defined
(2) These variables are used for time-stamping the QS data records. This `QS_tickTime_` variable is used to hold the 32-bit-wide SysTick timestamp at tick. The `QS_tickPeriod_` variable holds the nominal number of hardware clock ticks between two subsequent SysTicks. The SysTick ISR increments `QS_tickTime` by `QS_tickPeriod_`.
(3) This constant determines the QS buffer size in bytes
(4) This constant determines the QS baud rate.
(5) This enumeration defines application-specific QS trace record(s), to demonstrate how to use them.
(6) You need to define the `QS_onStartup()` callback to initialize the QS software tracing.
(7) You should adjust the QS buffer size (in bytes) to your particular application
(8) You always need to call `QS_initBuf()` from `QS_init()` to initialize the trace buffer.
(9) The `QS_onCleanup()` callback performs the cleanup of QS. Here nothing needs to be done.
(10) The `QS_onFlush()` callback flushes the QS trace buffer to the host. Typically, the function busy-waits for the transfer to complete. It is only used in the initialization phase for sending the QS dictionary records to the host (see please refer to "QSP Reference Manual" section in the "QP/C Reference Manual" an also to Chapter 11 in [PSiCC2])

7.1 QS Time Stamp Callback QS_onGetTime()

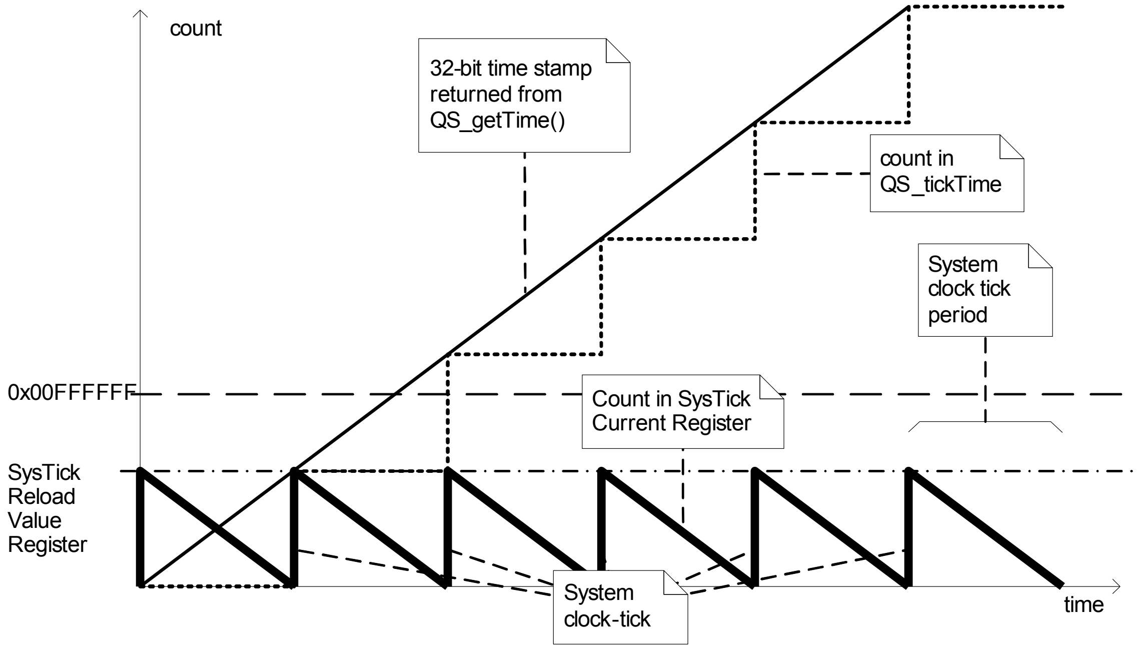
The platform-specific QS port must provide function `QS_onGetTime()` (Listing 7(11)) that returns the current time stamp in 32-bit resolution. To provide such a fine-granularity time stamp, the ARM-Cortex port uses the SysTick facility, which is the same timer already used for generation of the system clock-tick interrupt.

NOTE: The `QS_onGetTime()` callback is always called with interrupts locked.

Figure 5 shows how the SysTick Current Value Register reading is extended to 32 bits. The SysTick Current Value Register (`NVIC_ST_CURRENT`) counts down from the reload value stored in the SysTick Reload Value Register (`NVIC_ST_RELOAD`). When `NVIC_ST_CURRENT` reaches 0, the hardware automatically reloads the `NVIC_ST_CURRENT` counter from `NVIC_ST_RELOAD` on the subsequent clock tick. Simultaneously, the hardware sets the `NVIC_ST_CTRL_COUNT` flag, which "remembers" that the reload has occurred.

The system clock tick ISR `ISR_SysTick()` keeps updating the "tick count" variable `QS_tickTime_` by incrementing it each time by `QS_tickPeriod_`. The clock-tick ISR also clears the `NVIC_ST_CTRL_COUNT` flag.

Figure 5 Using the SysTick Current Value Register to provide 32-bit QS time stamp.


count
32-bit time stamp returned from QS_getTime()
count in QS_tickTime
System clock tick period
0x00FFFFFF
Count in SysTick Current Register
SysTick Reload Value Register
System clock-tick
time

Listing 7(11-15) shows the implementation of the function `QS_onGetTime()`, which combines all this information to produce a monotonic time stamp.

(12) The `QS_onGetTime()` function tests the `SysTick->CTRL[0]` flag. This flag being set means that the `SysTick->VAL` regisger has rolled over to zero, but the SysTick ISR has not run yet (because interrupts are still locked).

(13) Most of the time the `SysTick->CTRL[0]` flag is not set, and the time stamp is simply the sum of `QS_tickTime_ - SysTick->VAL)`. Please note that the `SysTick->VAL` register is negated to make it to an up-counter rather than down-counter.

(13) If the `SysTick->CTRL[0]` flag is set, the `QS_tickTime_` counter misses one update period and must be additionally incremented by `QS_tickPeriod_`.

7.2 QS Trace Output in QF_onIdle()/QK_onIdle()

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

Listing 8 QS trace output using the UART0 of the Stellaris LM3S811 MCU

```
void QK_onIdle(void) {
    . . .
#ifdef Q_SPY
    if ((USART2->SR & USART_FLAG_TXE) != 0) {              /* is TXE empty? */
        QF_INT_LOCK(dummy);
        uint16_t b = QS_getByte();
        QF_INT_UNLOCK(dummy);
        if (b != QS_EOD) {                              /* not End-Of-Data? */
            USART2->DR = (b & 0xFF);             /* put into the DR register */
        }
    }
#elif defined NDEBUG                /* sleep mode interferes with debugging */
    . . .
}
```

7.3 Invoking the QSpy Host Application

The QSPY host application receives the QS trace data, parses it and displays on the host workstation (currently Windows or Linux). For the configuration options chosen in this port, you invoke the QSPY host application as follows (please refer to “QSP Reference Manual” section in the “QP/C Reference Manual” an also to Chapter 11 in [PSiCC2]):

```
qspy -cCOM1 -b115200 -C4
```

8 References

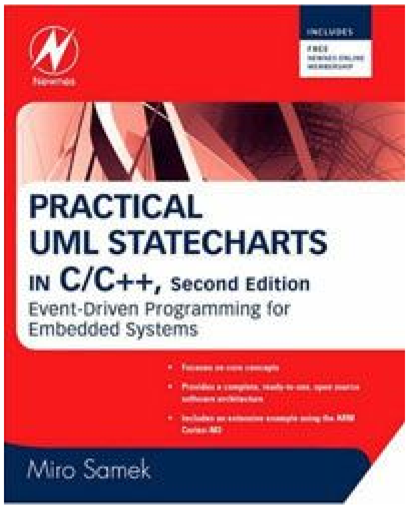
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP-Cortex] "Application Note: QP and ARM-Cortex with GNU", Quantum Leaps, 2010.	http://www.state-machine.com/arm/AN_QP_and_ARM-Cortex-GNU.pdf
[devkitPro] devkitPro website at SourceForge.net	https://sourceforge.net/projects/devkitpro.
[devkitARM] devkitARM download from the devkitPro project at SourceForge.net	https://sourceforge.net/projects/devkitpro/files/devkitARM
[GNU-make for Windows] GNU make and related UNIX-style file utilites for Windows.	http://www.state-machine.com/resources/-GNU_make_utils.zip
[Insight-GDB] Insight-GDB download from the devkitPro project at SourceForge.net	https://sourceforge.net/projects/devkitpro/files/Insight
[OpenOCD] OpenOCD on-chip debugger	http://openocd.berlios.de
[Eclipse] Eclipse IDE for C/C++ Developers	http://www.eclipse.org/downloads/
[Zylin-plugin] Zylin Embedded CDT plugin	http://opensource.zylin.com/embeddedcdt.html
[Atollic] Atollic TRUE Studio Light Eclipse-based IDE and ST-LINK debugger support	http://www.atollic.com/index.php/download
[AN-DPP] Quantum Leaps Application Note "Dining Philosophers Problem Example"	http://www.state-machine.com/resources/-AN_DPP.pdf
[CodeSourcery] Sourcery G++ Lite ARM EABI Sourcery G++ Lite 2008q3-66 Getting Started	http://www.codesourcery.com/sgpp/lite/arm/portal/doc2861/getting-started.pdf
[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.	http://www.embedded.com/showArticle.jhtml?articleID=190302110

9 Contact Information

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

+1 866 450 LEAP (toll free, USA only)
+1 919 869-2998 (FAX)

e-mail: info@quantum-leaps.com
WEB : http://www.quantum-leaps.com
http://www.state-machine.com


PRACTICAL UML STATECHARTS IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems
Miro Samek
Newnes

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