QDK™

H8/SKP36077-Renesas

Document Revision A
October 2008

Copyright © Quantum Leaps, LLC

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

Table of Contents

1 Introduction 1
- 1.1 What's Included in the QDK? 2
- 1.2 Licensing QP 2

2 Getting Started 3
- 2.1 Installation 3
- 2.2 Building the QP Libraries 5
- 2.3 Building the Examples 6
  - 2.3.1 Loading the Project into HEW 6
- 2.4 Running the Examples 7

3 The Vanilla QP Port 9
- 3.1 The qep_port.h Header File 9
- 3.2 The qf_port.h Header File 9
  - 3.2.1 The QF Object Size Configuration 10
  - 3.2.2 The QF Critical Section 10
- 3.3 ISRs in the Non-preemptive "Vanilla" Configuration 11
- 3.4 QP Idle Loop Customization in QF_onIdle() 12

4 The QK Port 13
- 4.1 The qk_port.h Header File 13
- 4.2 ISRs in the Preemptive Configuration with QK 14
- 4.3 Idle Loop Customization in the QK Port 14

5 Board Support Package 15
- 5.1 Compiler Options Used 15
- 5.2 Linker Options Used 16
  - 5.2.1 Specifying Program Sections 16
  - 5.2.2 Specifying Stack and Heap Sizes 17
- 5.3 The BSP header file bsp.h 18
- 5.4 BSP initialization 18
- 5.5 Starting Interrupts in QF_onStartup() 19
- 5.6 Assertion Handling Policy in Q_onAssert() 19

6 The Quantum Spy (QS) Instrumentation 20

7 Related Documents and References 22

8 Contact Information 23

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1 Introduction

This **QP™ Development Kit** (QDK) describes how to use QP™ event-driven platform with the Renesas H8 family of processors, the Renesas H8, H8S, H8/300 Standard C/C++ Toolchain, and the High Performance Embedded Workshop 4 (HEW4).

Figure 1 Renesas H8/SKP36077 Starter Kit with E8 In-Circuit Debugger.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

The actual hardware/software used to validate this QDK is described below (see Figure 1):

1. Renesas H8/SKP36077 development board with E8 In-Circuit Debugger.
2. High Performance Embedded Workshop 4.04 (HEW4)
3. Renesas H8S, H8/300 Standard C/C++ Toolchain v6.2.1.0.
4. QP/C/C++ v4.0.01 or higher.

As shown in Figure 1, the Renesas H8/SKP36077 Starter Kit (SKP) contains the SKP36077 board and the E8 In-Circuit Debugger that connects directly to the host development workstation via the provided USB cable. The USB connection also powers up the board, so no additional power is required. This QDK has been tested with the H8/36077 device with 4KB of RAM and 56KB of onboard flash. However, the described port should be applicable to most H8S, H8/300 devices.

1.1 What's Included in the QDK?

This QDK provides the QP port to H8 with the Renesas HEW4 IDE and Renesas H8 compiler, the Board Support Package (BSP) and two versions of the Dining Philosopher Problem (DPP) example application described in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08].

1. DPP with the cooperative "Vanilla" kernel; and
2. DPP with the preemptive run-to-completion QK kernel.

1.2 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

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2 Getting Started

This section describes how to install, build, and use the QDK-H8-Renesas-SKP36077 based two examples. This information is intentionally included early in this document, so that you could start using the QDK as soon as possible. The main focus of this section is to walk you quickly through the main points without slowing you down with full-blown detail.

> **NOTE:** This QDK assumes that the standard QP distribution consisting of QEP, QF, QK, and QS has been installed, before installing this QDK. It is also strongly recommended that you read the QP Tutorial (www.quantum-leaps.com/doxygen/qpc/tutorial_page.html) before you start experimenting with this QDK.

2.1 Installation

The QDK code is distributed in a ZIP archive (qdkc_h8-renesas_skp36077_<ver>.zip, where <ver> stands for a specific QDK-H8-Renesas-SKP36077 version, such as 4.0.01). You should uncompress the archive into the same directory into which you've installed all the standard QP components. The installation directory you choose will be referred henceforth as QP Root Directory <qp>. The following Listing 1 shows the directory structure and selected files included in the QDK-H8-Renesas-SKP36077 distribution. (Please note that the QP directory structure is described in detail in a separate Application Note: "QP Directory Structure"):

```
<qp>\                        - QP-root directory for Quantum Platform (QP)
 |
 +-include/                  - QP public include files
 | +-qassert.h               - Quantum Assertions platform-independent public include
 | +-qep.h                   - QEP platform-independent public include
 | +-qf.h                    - QF platform-independent public include
 | +-qk.h                    - QK platform-independent public include
 | +-qequeue.h               - native QF event queue include
 | +-qmpool.h                - native QF memory pool include
 | +-qvanilla.h              - native QF cooperative "Vanilla" kernel
 |
 +-ports\                    - QP ports
 | +-h8\                     - H8 port
 | | +-vanilla\              - "vanilla" ports
 | | | +-renesas\            - Renesas H8 compiler
 | | | | +-dbg\              - QP libraries - Debug configuration
 | | | | | +-qep_300HN.lib - QEP library for H8/300 target
 | | | | | +-qf_300HN.lib  - QF library for H8/300 target
 | | | | +-dbg\              - QP libraries - Release configuration
 | | | | +-spy\              - QP libraries - Spy configuration
 | | | | |
 | | | | +-make_300HN.bat  - make script for building QP libraries for H8/300HN target
 | | | | +-qep_port.h        - QEP port
 | | | | +-qf_port.h         - QF port
 | | | | +-qs_port.h         - QS port
 | | | | +-qp_port.h         - QP port
 | | |
 | | +-qk\                   - QK (Quantum Kernel) ports
 | | | +-renesas\            - Renesas H8 compiler
 | | | | +-dbg\              - QP libraries - Debug configuration
 | | | | | +-qep_300HN.lib - QEP library for H8/300 target
 | | | | | +-qf_300HN.lib  - QF library for H8/300 target
 | | | | | +-qk_300HN.lib  - QK library for H8/300 target
 | | | | +-dbg\              - QP libraries - Release configuration
 | | | | +-spy\              - QP libraries - Spy configuration
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
| | | | +-make_300HN.bat  - make script for building QP libraries for H8/300HN target
| | | | +-qep_port.h      - QEP port
| | | | +-qf_port.h       - QF port
| | | | +-qk_port.h       - QK port
| | | | +-qs_port.h       - QS port
| | | | +-qp_port.h       - QP port
|
+-examples\               - subdirectory containing the QP example files
| +-h8\                   - H8 port
| | +-vanilla\            - "vanilla" ports
| | | +-renesas\          - Renesas H8 compiler
| | | | +-dpp_skp36077\   - Dining Philosophers example for SKP36077 board
| | | | | +-Debug\        - directory containing the Debug build
| | | | | +-Release\      - directory containing the Release build
| | | | | +-Spy\          - directory containing the Spy build
| | | | | +-bsp.c         - Board Support Package for SKP36077 board
| | | | | +-bsp.h         - BSP header file
| | | | | +-dpp.h         - the DPP header file
| | | | | +-main.c        - the main function
| | | | | +-philo.c       - the Philosopher active object
| | | | | +-table.c       - the Table active object
| | | | | +-dpp_skp36077.hwp - HEW project for this application
| | | | | +-dpp_skp36077.hws - HEW workspace for this application
| | | | | +-skp36077def.h - Definitions for the SKP36077 board
| | | | | +-iodefine.h    - I/O definitions for H8/36077 (generated startup code)
| | | | | +-dbsct.c       - Definitions of program sections (generated startup code)
| | | | | +-intprg.c      - dummy definitions of interrupt handlers (generated code)
| | | | | +-resetprog.c   - reset handler (generated startup code)
| | | | | +-sbrk.h        - heap section definition (generated startup code)
| | | | | +-sbrk.c        - heap section definition (generated startup code)
| | | | | +-stacksct.h    - stack section definition (generated startup code)
| | | | | +-typedef.h     - Exact-width types (generated startup code)
| | |
| | +-qk\                 - QK examples
| | | +-renesas\          - Renesas H8 compiler
| | | | +-dpp_qk_skp36077\  - DPP example with QK for SKP36077 board with QK
| | | | | +-Debug\        - directory containing the Debug build
| | | | | +-Release\      - directory containing the Release build
| | | | | +-Spy\          - directory containing the Spy build
| | | | | +-bsp.c         - Board Support Package for SKP36077 borad
| | | | | +-bsp.h         - BSP header file
| | | | | +-dpp.h         - the DPP header file
| | | | | +-main.c        - the main function
| | | | | +-philo.c       - the Philosopher active object
| | | | | +-table.c       - the Table active object
| | | | | +-dpp_qk_skp36077.hwp - HEW project for this application
| | | | | +-dpp_qk_skp36077.hws - HEW workspace for this application
| | | | | +-skp36077def.h - Definitions for the SKP36077 board
| | | | | +-iodefine.h    - I/O definitions for H8/36077 (generated startup code)
| | | | | +-dbsct.c       - Definitions of program sections (generated startup code)
| | | | | +-intprg.c      - dummy definitions of interrupt handlers (generated code)
| | | | | +-resetprog.c   - reset handler (generated startup code)
| | | | | +-sbrk.h        - heap section definition (generated startup code)
| | | | | +-sbrk.c        - heap section definition (generated startup code)
| | | | | +-stacksct.h    - stack section definition (generated startup code)
| | | | | +-typedef.h     - Exact-width types (generated startup code)
```

Listing 1 Selected QP directories and files after installing QP and this QDK. The highlighted elements are included in the QDK-H8-Renesas-SKP36077 distribution.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.2 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-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\h8\` directory (see Listing 1). This section describes steps you need to take to rebuild the libraries yourself.

> **NOTE:** To streamline and simplify the QP-library build process, Quantum Leaps software does not use the vendor-specific IDEs, such as the IAR Embedded Workbench IDE, for building the QP libraries. Instead, this QDK provides *command-line* build process based on simple batch scripts.
>
> The build process for your application is largely independent on the QP-library builds. In fact, once you have the QP libraries, you typically don't need to rebuild them—at least not on the daily basis as you work on your application. This QDK uses the Renesas H8 compiler to build the example applications, but you are free to use any other build strategy.

The code distribution contains all the batch file `make_300HN.bat` for building all the libraries located in `<qp>\`**`ports`**`\h8\vanilla\renesas\` directory. For example, to build the debug version of all the QP libraries for the H8 CPU, with the Renesas H8 compiler, you open a console window on a Windows PC, change directory to `<qp>\`**`ports`**`\h8\vanilla\renesas\`, and invoke the batch by typing at the command prompt the following command:

```
make_300HN
```

The make process should produce the QP libraries in the location: `<qp>\ports\h8\vanilla\renesas\-`**`dbg`**`\`. The `make_300HN.bat` assumes that the Renesas H8 toolset has been installed in the directory `C:\tools\Renesas\Hew\Tools\Renesas\H8\6_2_1`.

> **NOTE:** You need to adjust the symbol **`RENESAS_H8_DIR`** at the top of the `make_300HN.bat` file if you've installed the Renesas H8 toolset in a different directory.

In order to take advantage of the Q-SPY instrumentation, you need to build the Spy version of the QP libraries. You achieve this by invoking the `make_300HN.bat` utility with the "spy" target, like this:

```
make_300HN spy
```

The make process should produce the QP libraries in the directory: `<qp>\ports\h8\vanilla\-renesas\`**`spy`**`\`.

You choose the build configuration by providing a target to the `make_300HN.bat` utility. The default target is "dbg". Other targets are "rel", and "spy" respectively. The following table summarizes the targets accepted by `make_300HN.bat`.

Software Version	Build command
Debug (default)	`make_300HN`
Release	`make_300HN rel`
Spy	`make_300HN spy`

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

Copyright © Quantum Leaps, LLC. All Rights Reserved.

2.3 Building the Examples

The examples included in this QDK are based on the standard Dining Philosopher Problem application implemented with active objects (see Quantum Leaps Application Note: "Dining Philosopher Problem Application" [QL AN-DPP 08] includes in this QDK).

The example directory contains the HEW workspace file `dpp_skp36077.hws` that you can load into the HEW IDE. The workspace contains two build configurations (Debug, Release, and Spy) that you can select with the drop-down list, as shown in Figure 2.

NOTE: The DPP example workspace comes with three build configurations: Debug, Release, and Spy. You select the build configuration through the drop-down list, as shown in Figure 2.

2.3.1 Loading the Project into HEW

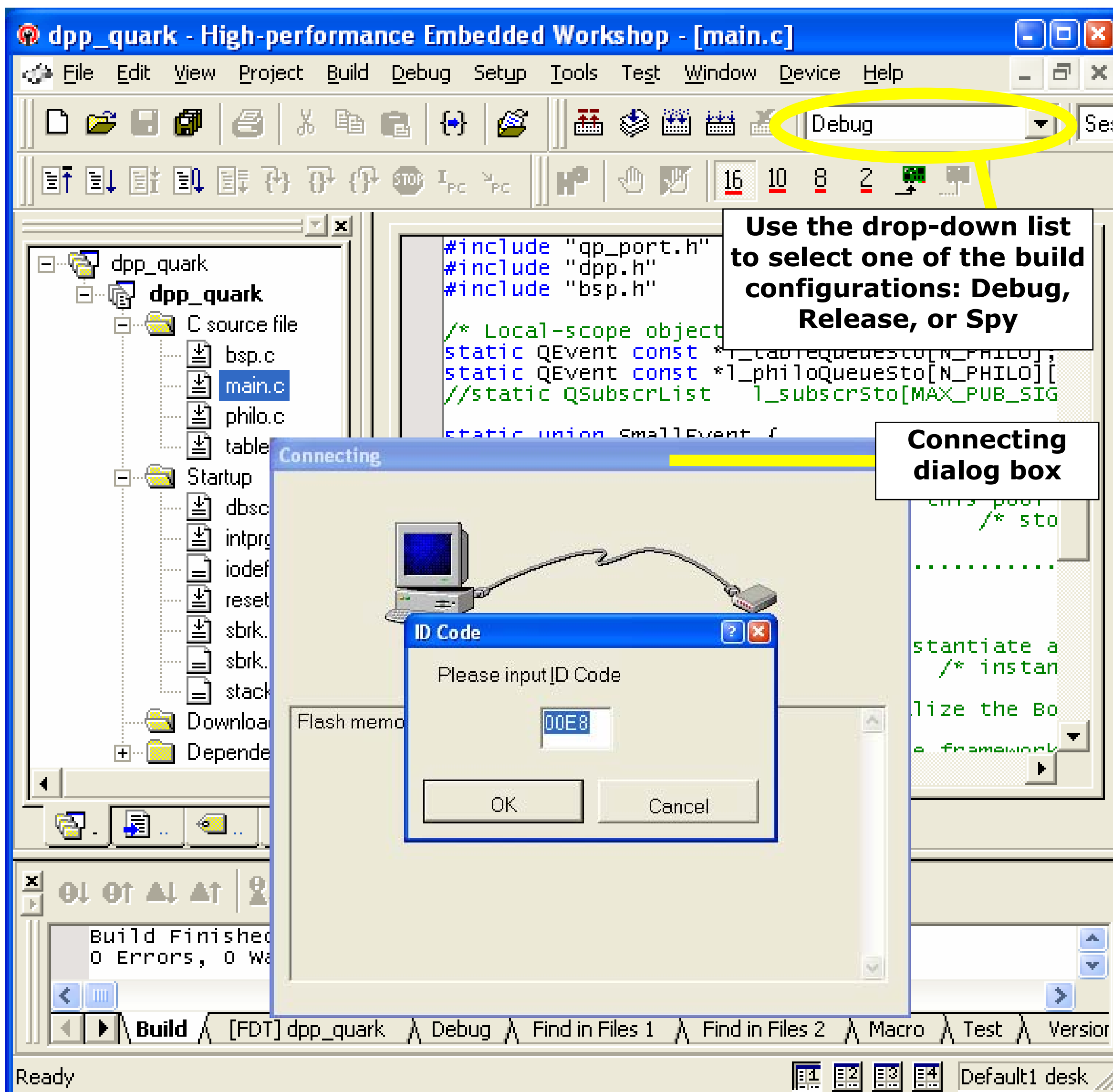


Figure 2 HEW IDE with the dpp_`skp36077` **project.**

Copyright © Quantum Leaps, LLC. All Rights Reserved.

1. Connect the H8/SKP36077 board to the PC with the provided USB cable
2. Launch HEW IDE and open the project `dpp_skp36077.hws` (located in `<qp>\examples\h8\vanilla\-renesas\dpp_skp36077\`). Figure 2 shows the screen shot of the HEW IDE after opening the project.
3. Build the project by select Build | Build menu or by pressing F7.

2.4 Running the Examples

The HEW debugger can directly communicate with the E8 In-Circuit Debugger (see Figure 1). To load the code into the MCU's flash, select Debug | Download Modules menu (see Figure 2). This will launch the progress-bar message box. When the message box window goes away, the MCU is programmed. You can run the example by selecting menu Debug | Go (see Figure 3).

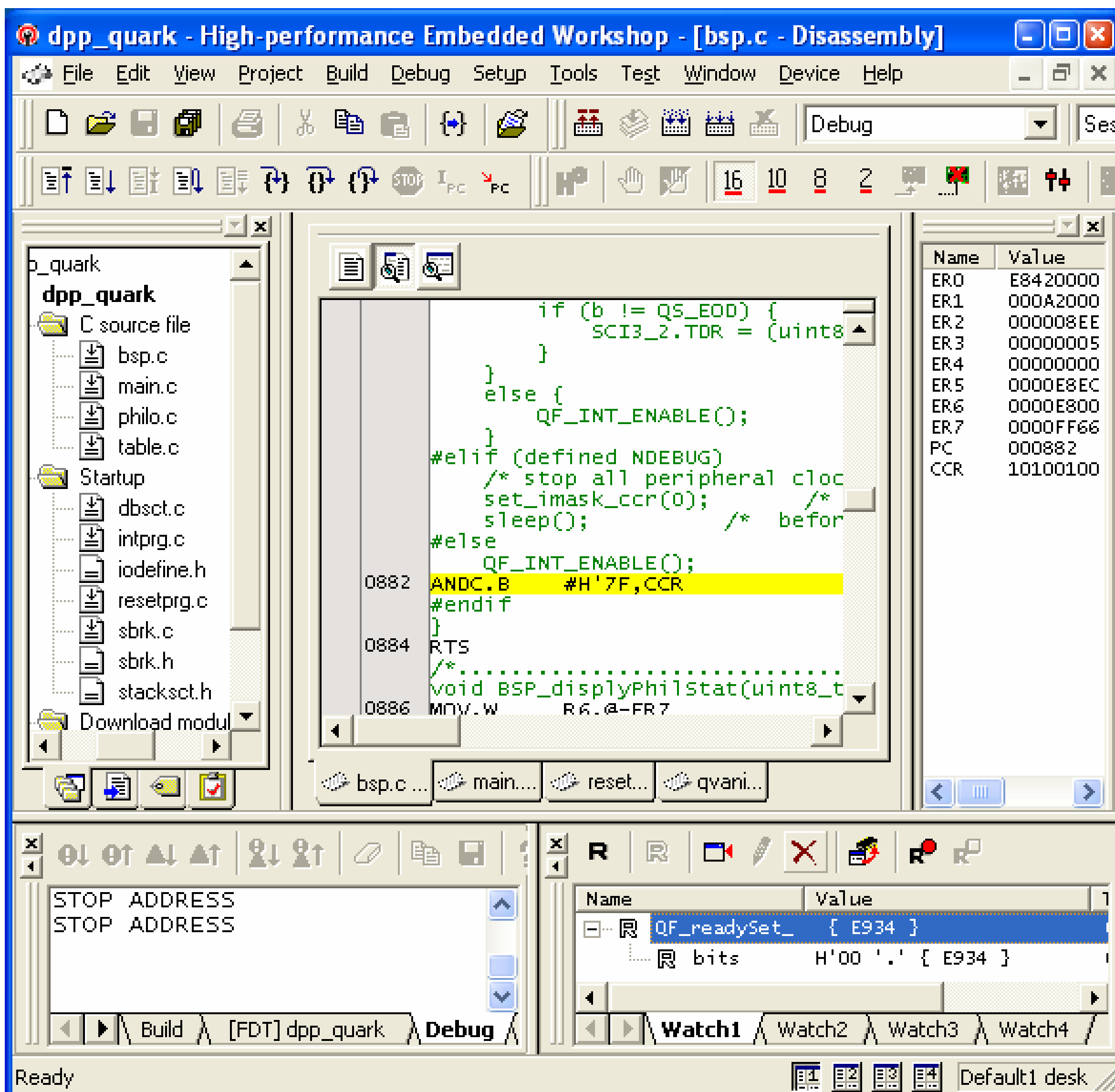


Figure 3 HEW Debugger with various views of the executing DPP application.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

An example run of the DPP application is shown in Figure 1. The four user LEDs should start blinking. The LEDs 0 through 3 represent Philosophers 0 through 3. An LED-on represents an "eating" philosopher. Extinguished LED represents philosopher "thinking" or "hungry".

To see the QS software trace output, you also need to download the Spy build configuration to the target board. Next you need to launch the QSPY host utility to observe the output in the human-readable format. You launch the QSPY utility on a Windows PC as follows: (1) Change the directory to the QSPY host utility <qp>\tools\qspy\win32\mingw\rel and execute:

```
qspy -c COM2 -b 38400 -O2 -F2
```

This will start the QSPY host application to listen on COM2 serial port with baud rate **38400**. (Please use the actual virtual COM port number on your PC.) The screen shot in Figure 4 shows the QSPY output from the DPP run:

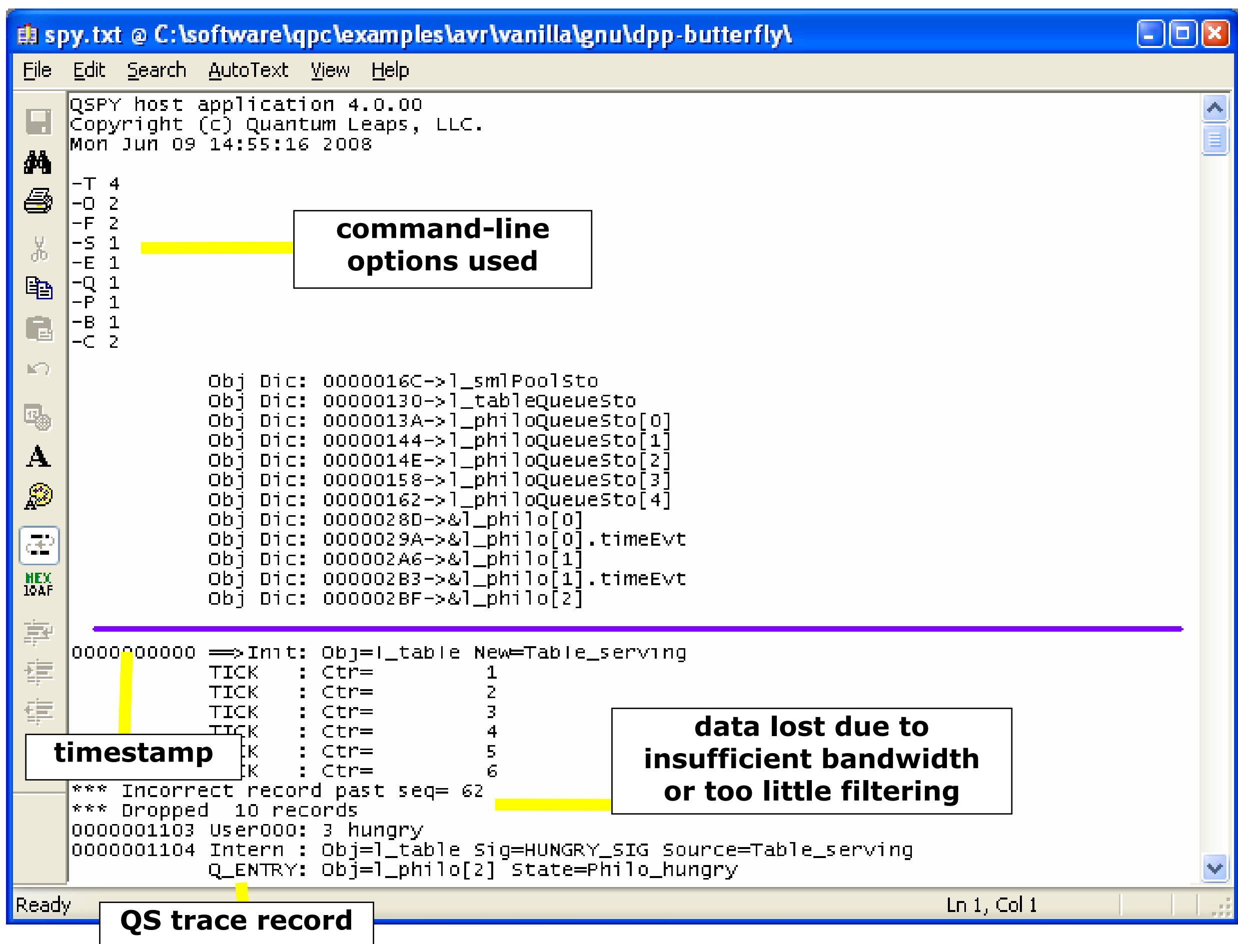


Figure 4 Screen shot from the QSPY output.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3 The Vanilla QP Port

The "vanilla" port shows how to use QP on a "bare metal" H8-based system without any underlying multitasking kernel. In the "vanilla" version of the QP, the only component requiring platform-specific porting is the QF. The other two components: QEP and QS require merely recompilation and will not be discussed here. Obviously, with the vanilla port you're not using the QK component. In case of H8, the "vanilla" QF port is very similar to the generic "vanilla" port described in Chapter 9 of [PSiCC2].

3.1 The qep_port.h Header File

The QEP header file for the H8 port with the Renesas H8 compiler is located in `<qp>\ports\h8\-vanilla\renesas\qep_port.h`.

```
                                         /* 1-byte signal space (255 signals) */
#define QP_SIGNAL_SIZE     1

/* exact-width integer types (Renesas H8 compiler does NOT provide <stdint.h>) */
typedef unsigned char  uint8_t;
typedef signed   char  int8_t;
typedef unsigned short uint16_t;
typedef signed   short int16_t;
typedef unsigned long  uint32_t;
typedef signed   long  int32_t;

#include "qep.h"               /* QEP platform-independent public interface */
```

Listing 2 The qep_port.h header file

The event signal size `QP_SIGNAL_SIZE` is configured to use 1-byte (256 signals). The next good choice for H8 is a 2-byte signal (64K signals). The Renesas H8 compiler is a pre C99-standard compiler and does not provide the exact-width integer types header file `<stdint.h>` (see C99 Standard, Section 7.18.1.1). For such a compiler, the standard types are defined using the `typedef` directives.

3.2 The qf_port.h Header File

The QF header file for the H8 port with the Renesas H8 compiler is located in `<qp>\ports\h8\-vanilla\renesas\qf_port.h`. The following sub-sections focus on explaining the QF configuration established by the `qf_port.h` header file shown in Listing 3.

```
              /* various QF object sizes configuration for this port, see NOTE01 */
(1) #define QF_MAX_ACTIVE               8
(2) #define QF_EVENT_SIZ_SIZE           2
(3) #define QF_EQUEUE_CTR_SIZE          1
(4) #define QF_MPOOL_SIZ_SIZE           2
(5) #define QF_MPOOL_CTR_SIZE           2
(6) #define QF_TIMEEVT_CTR_SIZE         2

                               /* interrupt locking policy for Renesas H8 compiler */
(7) #define QF_INT_DISABLE()          set_imask_ccr(1)
(8) #define QF_INT_ENABLE()           set_imask_ccr(0)

                                /* critical section policy for Renesas H8 compiler */
(9) #define QF_INT_KEY_TYPE            uint8_t
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
(10) #define QF_INT_LOCK(key_)       do { \
(11)     (key_) = get_imask_ccr(); \
(12)     set_imask_ccr(1); \
(13) } while (0)
(14) #define QF_INT_UNLOCK(key_)     set_imask_ccr(key_)

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

Listing 3 The qf_port.h header file.

3.2.1 The QF Object Size Configuration

The first part of the `qf_port.h` header file defines limits and sizes of various internal data structures used in the QF and the applications, as shown in Listing 3(1-6).

(1) `QF_MAX_ACTIVE` defines the maximum number of active objects that QF can manage. Here this limit is set to just 8, to save some RAM, but you can increase this limit up to 63, inclusive.

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

(3) Maximum event queue counter size is set to 1-byte, meaning that a single event queue can hold up to 255 events.

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

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

(6) The timer counter size is set to 2-bytes, meaning that a maximum timeout can be 2^{16}-1 clock ticks.

3.2.2 The QF Critical Section

The H8 microcontrollers do not provide any advanced interrupt prioritization in hardware. In other words, if you enable interrupts at the CPU level (by clearing the I flag in the CCR), the H8 hardware will allow all interrupts, including the interrupt level currently being serviced. For that reason unlocking interrupts inside ISRs is not advisable, because ISRs are typically not reentrant. Consequently, QF services invoked from the ISRs (such as `QF_tick()`, `QF_publish()`, `QActive_postFIFO`, etc.) must not inadvertently unlock interrupts that are locked in hardware upon the entry to the interrupt processing. All this means that QF must use an interrupt locking/unlocking scheme that allows for nesting critical sections. QF provides such a policy called "saving and restoring interrupt status" and described in Chapter 7 of [PSiCC2].

NOTE: The H8 hardware "prioritizes" only those interrupts that arrive while the interrupts are disabled. If multiple interrupts arrive during that time, the H8 hardware picks the one with the highest hardware priority (see H8 Hardware Manual [Renesas H8 HW]). However, the H8 has no prioritized interrupt controller, which would keep track of the priority of the currently serviced interrupt and allow only interrupts of higher priority to preempt the currently running interrupt.

The QF critical section policy for H8 is defined in Listing 3(7-14) as follows:

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(7-8) The macros QF_INT_DISABLE() and QF_INT_ENABLE() define simple, unconditional interrupt disabling and enabling, respectively. For the H8, the inline function set_imask_ccr(1) expands to a single assembly instruction ORC.B #H'80,CCR. Similarly, the inline function set_imask_ccr(0) expands to a single assembly instruction ANDC.B #H'7F,CCR.

(9) The interrupt lock key type is defined, which means that the interrupt status is saved and restored into the interrupt "key" variable (see Chapter 7 of [PSiCC2]).

(10-13) The macro QF_INT_LOCK() defines the CPU and compiler-specific interrupt locking mechanism. The interrupt locking macro QF_INT_LOCK() is defined with the intrinsic functions get_imask_ccr() and set_imask_ccr(1). (Please note that the do {…} while (0) loop around the macro is only necessary for syntactically correct grouping of instructions.)

(14) The macro QF_INT_UNLOCK() defines the CPU and compiler-specific interrupt unlocking mechanism. This macro restores the interrupt status provided in the interrupt "key" variable.

3.3 ISRs in the Non-preemptive "Vanilla" Configuration

The Renesas H8 compiler supports writing interrupts in C. In the "vanilla" port, the ISRs are identical as in the simplest of all "superloop" (main+ISRs), and there is nothing QP-specific in the structure of the ISRs.

```
(1) __interrupt(vect = 26)
(2) void timer_z_isr(void) {
(3)     /* clear any level-sensitive interrupt sources, if necessary ... */

(4)     QF_tick();

        /* perform other work of the ISR, e.g., switch debouncing */
    }
```

Listing 4 Time tick interrupt calling QF_tick() function to manage armed time events.

(1) The ISR in C is always compiler-specific, as the C standard does not define how to specify ISRs. In the case of the Renesas H8 compiler for interrupt must be declared with the __interrupt extended keyword. You also need to specify the interrupt vector number in the parentheses as shown.

(2) The ISR in C that follows the interrupt vector specification must have a void (void) signature.

(3) The level-sensitive interrupt sources should be cleared somewhere in the ISR body

(4) The time-tick ISR must invoke QF_tick(), and can also perform other things, if necessary. The function QF_tick() cannot be reentered, that is, it necessarily must run to completion and return before it can be called again. This requirement is automatically fulfilled, because the H8 hardware locks interrupts upon the interrupt entry and will not allow the same interrupt to preempt currently running interrupt.

NOTE: You should **not** enable interrupts inside ISRs, because the H8 CPU does not prioritize interrupts in hardware.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

3.4 QP Idle Loop Customization in QF_onIdle()

The cooperative "vanilla" kernel 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 the determination of the idle condition might change by any interrupt posting an event.

H8 microcontrollers support several power-saving levels (consult the specific data sheet for details). The following piece of code shows the `QF_onIdle()` callback that puts H8 core into the idle power-saving mode. Please note that H8 architecture allows for very **atomic** setting the low-power mode and enabling interrupts at the same time (see the article "Using Low-Power Modes in Foreground/Background Systems" [Samek 07a]).

```
(1) void QF_onIdle(void) {
    #ifdef Q_SPY
        /* perform QS output ... */
(2) #elif (defined NDEBUG)          /* low-power mode interferes with debugging */
(3)     /* stop all peripheral clocks that you can in your applicaiton ... */
(4)     set_imask_ccr(0);        /* the following SLEEP instruction will execute */
(5)     sleep();           /*  before entering any pending interrupt, see NOTE01 */
    #else
(6)     QF_INT_ENABLE();                                 /* just unlock interrupts */
    #endif
    }
```

Listing 5 `QF_onIdle()` **for the non-preemptive ("vanilla") QP port to H8.**

(1) The `QF_onIdle()` callback is always called with interrupts locked to prevent any race condition between posting events from ISRs and transitioning to the sleep mode.

(2) The low-power mode is entered only in the Release (not DEBUG) configuration.

(3) The clock management `registers` are setup the desired sleep mode. Please note that the sleep mode is not active until the SLEEP command.

(4) The interrupts are unlocked with the `set_imask_ccr(0)` intrinsic function.

(5) The sleep mode is activated with the `sleep()` intrinsic function.

> **NOTE:** As described in "H8/300H Series Software Manual", Section 2.2.5 "ANDC" [Renesas H8 Sw], **no** interrupt requests, including NMI, are accepted immediately after execution of the `ANDC` instruction. This means that the `ANDC` instruction and the immediately following `SLEEP` instruction are executed **atomically**.
>
> **CAUTION:** The instruction pair (`ANDC.B #H'7F,CCR, SLEEP`) should never be separated by any other instruction.

(6) In the DEBUG configuration the interrupts are simply enabled.

> **NOTE:** Every path through `QF_onIdle()` callback function must ultimately unlock interrupts.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

4 The QK Port

The QP port with the preemptive kernel (QK) is remarkably simple and very similar to the "vanilla" port. In particular, the interrupt locking/unlocking policy is the same, and the BSP is identical, except some small additions to the ISRs.

The DPP example for the QK port is provided in the directory `<qp>\examples\h8\qk\renesas\dpp_qk_skp36077`.

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

4.1 The qk_port.h Header File

In the QK port, you use very similar configuration as the "Vanilla" port described earlier. This section describes only the differences, specific to the QK component.

You configure and customize QK through the header file `qk_port.h`, which is located in the QP ports directory `<qp>\ports\h8\qk\renesas\`.

```
                                                /* interrupt entry code */
(1) #define QK_ISR_ENTRY()    (++QK_intNest_)
                                                 /* interrupt exit code */
(2) #define QK_ISR_EXIT()     do { \
(3)     --QK_intNest_; \
(4)     if (QK_intNest_ == (uint8_t)0) { \
(5)         QK_schedule_(0x00); \
        } \
    } while (0)

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

Listing 6 qk_porth.h header file

(1) The macro `QK_ISR_ENTRY()` is designed to be called upon the entry to an ISR programmed in C. The macro informs the QK preemptive kernel about entering an interrupt. This macro increments the nesting up-down counter `QK_intNest_` to account for entering a new interrupt level. This informs the QK scheduler about the interrupt context, so that if a QF API is called or another interrupt preempts this one, the QK scheduler will not be called.

NOTE: The hardware interrupt entry sequence for H8 includes setting the global interrupt mask in the CCR register, so that interrupts remain locked unless unlocked explicitly.

(2) The macro `QK_ISR_EXIT()` is designed to be called upon the exit from an ISR programmed in C. The macro informs the QK preemptive kernel about exiting an interrupt.

(3) The QK interrupt nesting up-down counter `QK_intNest_` is decremented to the level before entering the ISR.

NOTE: The whole ISR in this H8 port runs with interrupts disabled, so the wole body of the ISR runs in a critical section. You should **not** re-enable interrupts inside ISRs.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

(4) Only if the interrupt nesting drops to zero (ISR nested on task-level code)…

(5) The QK scheduler is called to perform the asynchronous preemptions.

(6) The QK configuration is specified simply by including the "qk.h" QK interface.

4.2 ISRs in the Preemptive Configuration with QK

As all preemptive kernels, QK must be notified about interrupt entry and exit. You achieve this by means of the QK macros QK_ISR_ENTRY() and QK_ISR_EXIT(), as shown in Listing 7.

```
__interrupt(vect = 26)
void timer_z_isr(void) {
    QK_ISR_ENTRY();                          /* inform QK about ISR entry */
    TZ0.TSR.BIT.IMFA = 0;                     /* clear compare match flag */
#ifdef Q_SPY
    l_tickTime += (uint32_t)((f1_CLK_SPEED/8 + BSP_TICKS_PER_SEC/2)
                             / BSP_TICKS_PER_SEC);
#endif
    QF_tick();                           /* process all armed time events */
    QK_ISR_EXIT();                            /* inform QK about ISR exit */
}
```

Listing 7 Time tick interrupt calling QF_tick() function to manage armed time events and QK ISR entry/exit macros.

4.3 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 kernel 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 3.4).

The following Listing 8 shows an example implementation of QK_onIdle() for the H8.

```
/*..........................................................................*/
void QK_onIdle(void) {
#ifdef Q_SPY
    /* perform QS output ... */
#elif (defined NDEBUG)         /* low-power mode interferes with debugging */
    /* stop all peripheral clocks that you can in your applicaiton ... */
    sleep();                     /*  before entering any pending interrupt */
#endif
}
```

Listing 8 QK_onIdle() callback for H8.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

5 Board Support Package

The Board Support Package (BSP) for H8 with the non-preemptive Vanilla kernel is located in the directory: `<qp>\examples\h8\vanilla\renesas\dpp_skp36077\` and consists of the following files:

1. `bsp.h` contains the Board Support Package interface (BSP)
2. `bsp.c` contains the implementation of the BSP, which includes all ISRs and all platform-specific QP-nano callbacks.

NOTE: This QDK-nano uses the recommended by Renesas C-startup for H8. The C-startup code is comprised of the following files:

1. `stacksct.h` – stack size definition (**customize for your application**)
2. `sbrk.h` – heap size definition (**customize for your application**)
3. `sbrk.c`
4. `iodefine.c`
5. `dbsct.h`
6. `intprg.c` – default interrupt handlers (**customize for your application**)
7. `resetprg.c`
8. `typedefine.h`

Out of these 8 files only the indicated 3 files need customization for a specific application.

5.1 Compiler Options Used

You set the compiler and linker options through the HEW IDE. The compiler options are as follows:

```
-cpu=300HN -include="$(PROJDIR)\.","$(PROJDIR)\..\..\..\..\include" -
object="$(CONFIGDIR)\$(FILELEAF).obj" -debug -nolist -chgincpath -nologo
```

The –debug option denotes that the compiler should generate the Debug information. This option is only used in the Debug build. Both the Debug and Release builds use the generic CPU type "With no specification".

NOTE: You can access the compiler options by selecting the Build | Renesas H8S, H8/300 Standard Toolchain..., or by right-clicking on any of the C modules in the Workspace window and choosing the Build Options/Renesas H8S, H8/300 Standard Toolchain... pop-up menu.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

5.2 Linker Options Used

5.2.1 Specifying Program Sections

The H8 devices have typically RAM divided into two separate blocks, as shown in the memory map of the H8/36077 device published in Section 6.3 of the "SKP36077 User's Manual" [Renesas SKP36077] and reproduced in Figure 5 below.

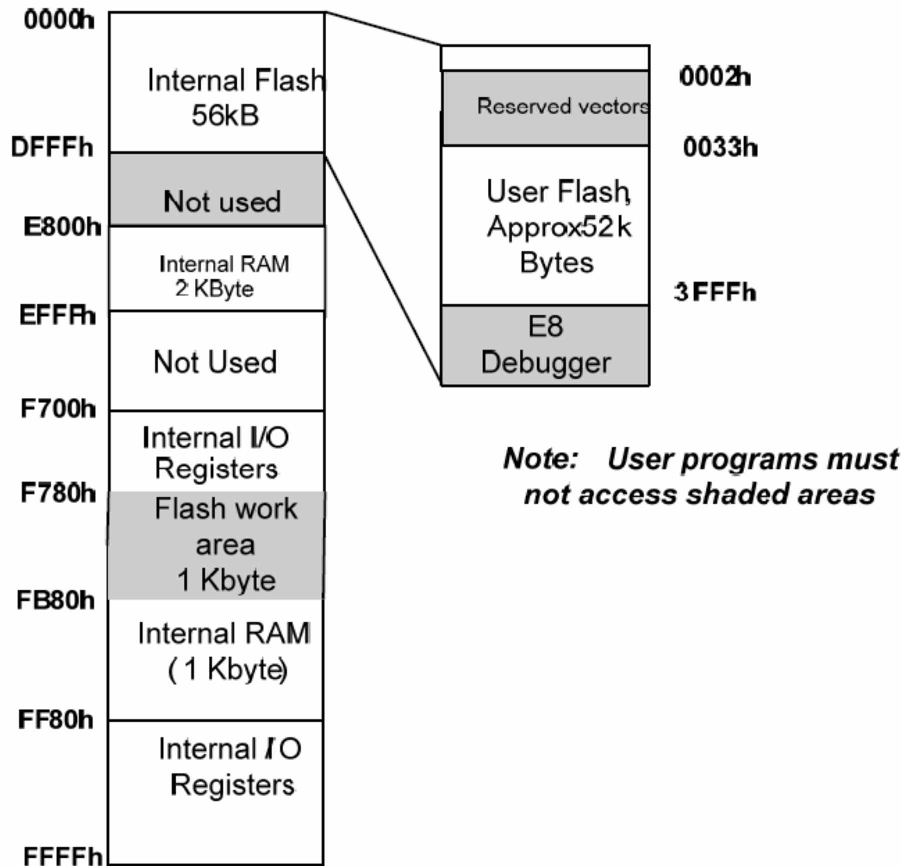


Figure 5 Memory map of the H8/36077 device with the "Kernel" program (Renesas ROM monitor program).

The HEW IDE allows to specify very precisely all the program sections, as shown in the screen shot in Figure 6. For the SKP36077 board, the sections are configured according to the memory map for the H8/36077 device.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

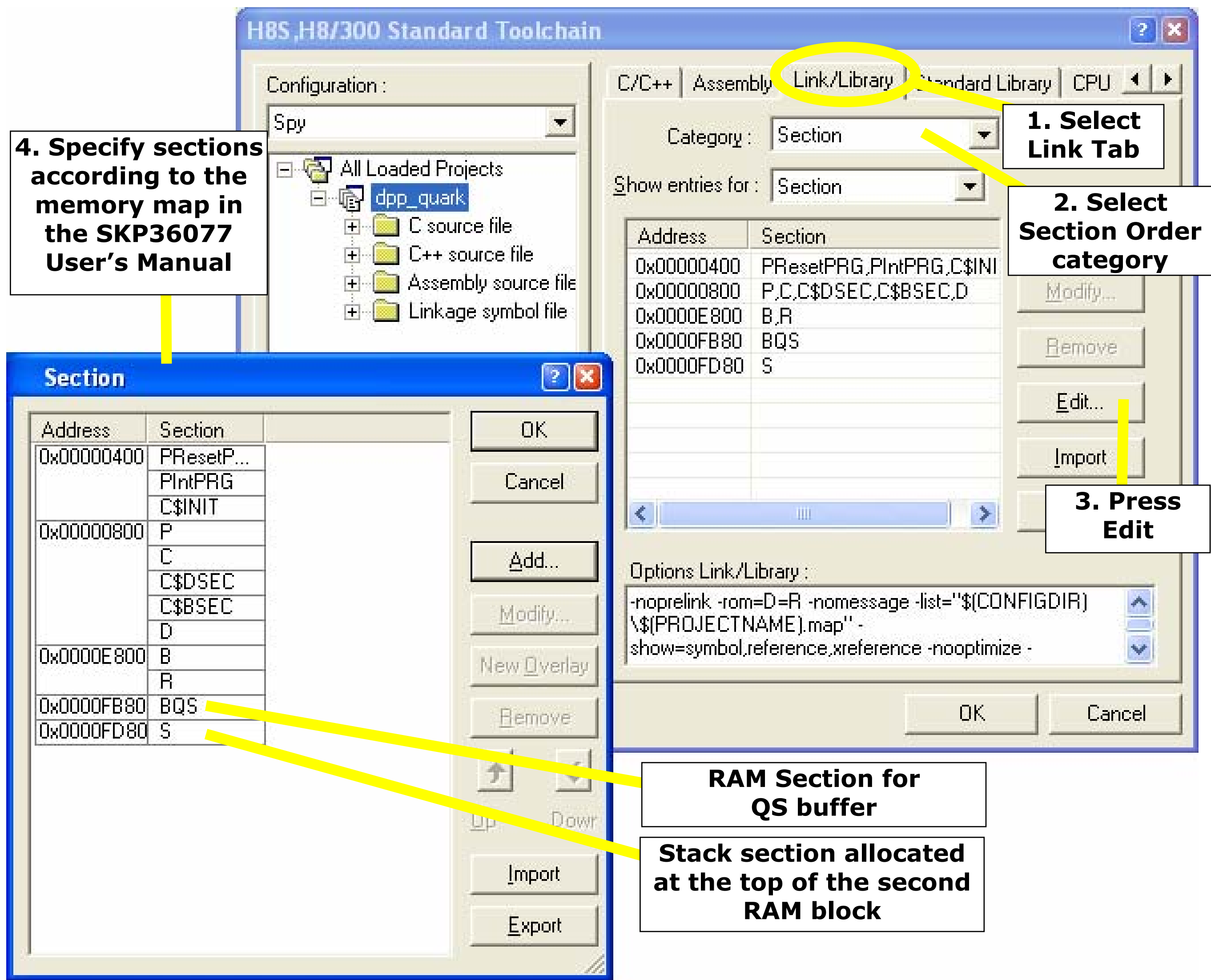


Figure 6 Specifying the sections names and addresses in the HEW IDE.

In particular, the Section dialog box shows two special RAM sections **BQS** for the QS trace buffer (see the upcoming Section 6), and **S** for the stack located at 0x0000FD80. Both sections are allocated in the **second** RAM block of the H8/36077 device.

NOTE: The stack must entirely fit in the RAM block, so the start address for the stack section plus the stack size must be less than the end of the RAM block (at 0x0000FF80 in case of H8/36077). The stack size is configured in the file `stacksct.h` described below.

5.2.2 Specifying Stack and Heap Sizes

You specify the size of the stack in the file stacksct.h, which contains the single `#pragma` directive:

```
    #pragma stacksize 0x200
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

NOTE: The QK preemptive kernel generally requires more stack space than the cooperative "Vanilla" kernel. You need to adjust the stacksize value for your system.

You specify the size of the heap in the file sbrk.h, which is shown below:

```
#define HEAPSIZE 1
```

This QDK-nano-H8 does not use the heap[1], and consequently the heapsize is defined to the minimum value of 1.

5.3 The BSP header file bsp.h

```
(1) #include "iodefine.h"              /* I/O register definitions for H8/36077 */

    /*--------------------------------------------------------------------------*/
(2) #define BSP_TICKS_PER_SEC   50

    void BSP_init(void);
    void BSP_displyPhilStat(uint8_t n, char const *stat);
```

Listing 9 The bsp.h for the PELICAN crossing example.

(1) The header file "iodefine.h" provides the definitions of the H8 special function registers.

(2) The BSP defines the desired ticking rate. This constant is useful for defining timeouts, which are always specified in units of clock ticks.

5.4 BSP initialization

The following BSP_init() function from the PELICAN crossing application for the SKP36077 board configures the PIO lines for the User LEDs, the User switches, and the system clock tick. Finally, the BSP_init() function initializes the QS software tracing (only in the Q_SPY configuration):

```
void BSP_init(void) {
    WDT.TCSRWD.BYTE = 0x10;                             /* disable Watchdog */
    WDT.TCSRWD.BYTE = 0x00;

    MSTCR2.BIT.MSTTZ   = 0;                               /* turn on TimerZ */
    TZ0.TCR.BIT.TPSC   = 3;                         /* internal clock phi/8 */
    TZ0.TCR.BIT.CCLR   = 1;
    TZ0.GRA            = (uint16_t)(((f1_CLK_SPEED/8 + BSP_TICKS_PER_SEC/2)
                                     / BSP_TICKS_PER_SEC) - 1);
    TZ0.TIER.BIT.IMIEA = 1;                  /* compare match interrupt enable */

                                                  /* enable the User LEDs... */
    LED0_DDR_1();                              /* configure LED0 pin as output */
    LED1_DDR_1();                              /* configure LED1 pin as output */
    LED2_DDR_1();                              /* configure LED2 pin as output */
    LED3_DDR_1();                              /* configure LED3 pin as output */
```

[1] Using the heap in real-time embedded devices can cause many problems, such as heap fragmentation, non-determinism, concurrency issues, etc.

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
        LED0 = LED_OFF;
        LED1 = LED_OFF;
        LED2 = LED_OFF;
        LED3 = LED_OFF;
                                                    /* enable the Switch... */
        SW1_DDR = 0;

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

5.5 Starting Interrupts in QF_onStartup()

QP-nano invokes the `QF_onStartup()` callback just before starting the event loop inside `QF_run()`. The `QF_onStartup()` function must start the interrupts configured earlier. In this BSP only the system tick interrupt is started.

```
    void QF_onStartup(void) {
        TZ.TSTR.BIT.STR0 = 1;                                   /* start TimerZ */
    }
```

5.6 Assertion Handling Policy in Q_onAssert()

As described in Chapter 6 of [PSiCC2], all QP components use internally assertions to detect errors in the way application is using the QP services. You need to define how the application reacts in case of assertion failure by providing the callback function `Q_onAssert()`. Typically, you would put the system in fail-safe state and try to reset. It is also a good idea to log some information as to where the assertion failed.

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

```
   void Q_onAssert(char const Q_ROM * const Q_ROM_VAR file, int line) {
        QF_INT_LOCK();                                  /* lock the interrupts */
        for (;;) {                               /* hang in this for-ever loop */
        }
   }
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

6 The Quantum Spy (QS) Instrumentation

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

QS can be configured to send the trace data out of the serial port of the target device. On the H8/SKP36077 baord, QS uses the built-in UART2 to send the trace data to the host. The QS platform-dependent implementation is located in the file bsp.c and looks as follows:

```
 (1) #ifdef Q_SPY

 (2) #define BAUD_RATE 38400

 (3) #pragma section QS
 (4) static uint8_t l_qsBuf[0x200 - 4];                          /* buffer for QS */
 (5) static uint32_t l_tickTime;
     #pragma section

     enum AppRecords {
              /* application-specific trace records */
         PHILO_STAT = QS_USER
     };

 (6) uint8_t QS_onStartup(void const *arg) {
         uint16_t n;

 (7)     QS_initBuf(l_qsBuf, sizeof(l_qsBuf));

 (8)     IO.PMR1.BIT.TXD2  = 1;                         /* enable SCI3 TX output */
 (9)     MSTCR1.BIT.MSTS3 = 0x0;        /* clear the SCI3 module stop control bit */
         SCI3_2.SCR3.BYTE = 0x00;               /* clear TE & RE bits to 0 in SCR */
         SCI3_2.SSR.BYTE &= ~(0x8 | 0x10 | 0x20);              /* clear error flags */
         SCI3_2.SMR.BYTE  = 0x00;        /* set 8-bit data, 1-stop bit, no parity */
         SCI3_2.BRR       = (uint8_t)((f1_CLK_SPEED + 32*BAUD_RATE/2)
                                / (32*BAUD_RATE) - 1);
         SCI3_2.SCR3.BYTE |= 0x20;                         /* enable transmission */

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

(12)     return (uint8_t)1;            /* indicate successfull QS initialization */
     }
     /*......................................................................*/
(13) void QS_onCleanup(void) {
     }
     /*......................................................................*/
(14) void QS_onFlush(void) {
         uint16_t b;
         while ((b = QS_getByte()) != QS_EOD) { /* next QS trace byte available? */
             while ((SCI3_2.SSR.BYTE & 0x80) == 0) {          /* TDRE not ready? */
```

Copyright © Quantum Leaps, LLC. All Rights Reserved.

```
            }
            SCI3_2.TDR = (uint8_t)b;  /* stick the byte to the TX data register */
        }
    }
    /*..........................................................................*/
    /* NOTE: invoked within a critical section (inetrrupts disabled) */
(15) QSTimeCtr QS_onGetTime(void) {
(16)     if (TZ0.TSR.BIT.IMFA == 0) {       /* input compare match flag NOT set? */
            return l_tickTime + (uint32_t)TZ0.TCNT;
        }
        else {       /* the output compare occured, but the ISR did not run yet */
(17)         return l_tickTime
(18)                + (uint32_t)((f1_CLK_SPEED/8 + BSP_TICKS_PER_SEC/2)
                             / BSP_TICKS_PER_SEC)
(19)                + (uint32_t)TZ0.TCNT;
        }
    }
    #endif                                                            /* Q_SPY */
```

Listing 10 QS implementation to send data out of the UART2 serial port of the H8

(1) The QS instrumentation is enabled only when the macro `Q_SPY` is defined

(2) You might want to adjust the UART baud rate to your particular system (the 38400 baud rate works well with the 10MHz f1 clock)

(3) The most expensive (in terms of RAM) trace buffer is allocated in the QS section, which is placed in the second RAM block (see Section 5.2.1).

(4) You should adjust the QS buffer size (in bytes) to your particular application

(5) The local variable l_tickTime is used to hold the clock count at system clock tick. This variable is used to generate 32-bit timestamp.

(6) The `QS_onStartup()` callback performs the initialization of QS

(7) You always need to call `QS_initBuf()` from `QS_onStartup()` to initialize the trace buffer. This particular QS port initializes USART for data transfer at the given baud rate (`BAUD_RATE = 38400` bits per second)

(8-9) UARTA2 is configured.

(10-11) The QS filters are configured.

(12) Return 1 from `QS_onStartup()` callback means that configuration of QS output was successfull.

(13) The `QS_onCleanup()` callback performs the cleanup of QS. Here nothing needs to be done.

(14) 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 Chapter 11 in [PSiCC2])

(15) The `QS_getTime()` callback provides the time-stamp to the QS trace records. The QS time-stamping implementation uses Timer A1.

(16) Here the input compare match flag is tested to find out whether Timer Z0 rolled over. This flag being set means that the timer rolled over from 0x0000 to 0xFFFF.

(17-18) In case of the rollover the clock count at tick must be incremented by the number of timer counts per system clock tick.

(19) The 32-bit timestamp is the combination of the clock count at tick (the coarse time) and the current value of the Timer Z0 counter (the fine time).

Copyright © Quantum Leaps, LLC. All Rights Reserved.

7 Related Documents and References

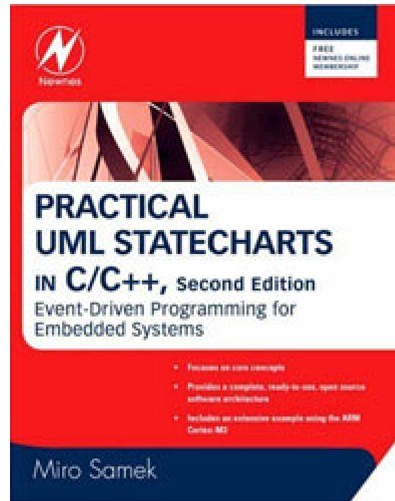
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.quantum-leaps.com/psicc2.htm
[QP 08] "QP/C/C++ Reference Manuals", Quantum Leaps, LLC, 2008	http://www.quantum-leaps.com/doxygen/qpc/ http://www.quantum-leaps.com/doxygen/qpcpp/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.quantum-leaps.com/doc/AN_QP_Directory_Structure.pdf
[QL AN-PELICAN 08] "Application Note: PELICAN Crossing Application", Quantum Leaps, LLC, 2008	http://www.quantum-leaps.com/doc/AN_PELICAN.pdf
[Renesas H8 HW] "H8/36077 Group Hardware Manual", Renesas 2005	Renesas document REJ09B0216-0100, available online at www.renesas.com
[Renesas H8 Sw] "H8/300H Series Software Manuall", Renesas, 2004.	Renesas document REJ09B0213-0300, available online at www.renesas.com
[Renesas H8 compiler] "H8S, H8/300 Series C/C++ Compiler, Assembler, Optimizing Linkage Editor Compiler Package Ver.6.01 User's Manual", Renesas 2004	Renesas document REJ10B0161-0100, available online at www.renesas.com.
[Renesas SKP36077] "SKP36077 StarterKit Plus User's Manual", Reneas 2007	Document included in PDF with SKP36077 kit.
[Samek 06b] "Build a Super Simple Tasker", Embedded Systems Design, Miro Samek and Robert Ward, July 2006	http://www.embedded.com/shared/-printableArticle.jhtml?articleID=190302110
[Samek 07a] "Using Low-Power Modes in Foreground/Background Systems", Miro Samek, to be published in ESD, October 2007	www.embedded.com/mag.htm

Copyright © Quantum Leaps, LLC. All Rights Reserved.

8 Contact Information

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

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



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

Renesas Technology Corp.
2-6-2, Ote-machi Chiyoda-ku,
Tokyo,100-0004
Japan
WEB: www.renesas.com

Copyright © Quantum Leaps, LLC. All Rights Reserved.