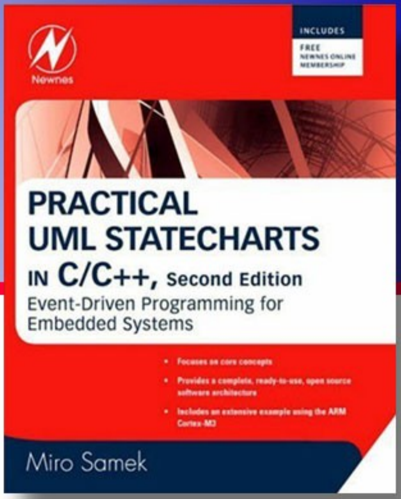

Newnes
INCLUDES FREE NEWNES ONLINE MEMBERSHIP
PRACTICAL UML STATECHARTS IN C/C++, Second Edition
Event-Driven Programming for Embedded Systems
Focuses on core concepts
Provides a complete, ready-to-use, open source software architecture
Includes an extensive example using the ARM Cortex-M3
Miro Samek

QDK™
TMS320C55x-C5500

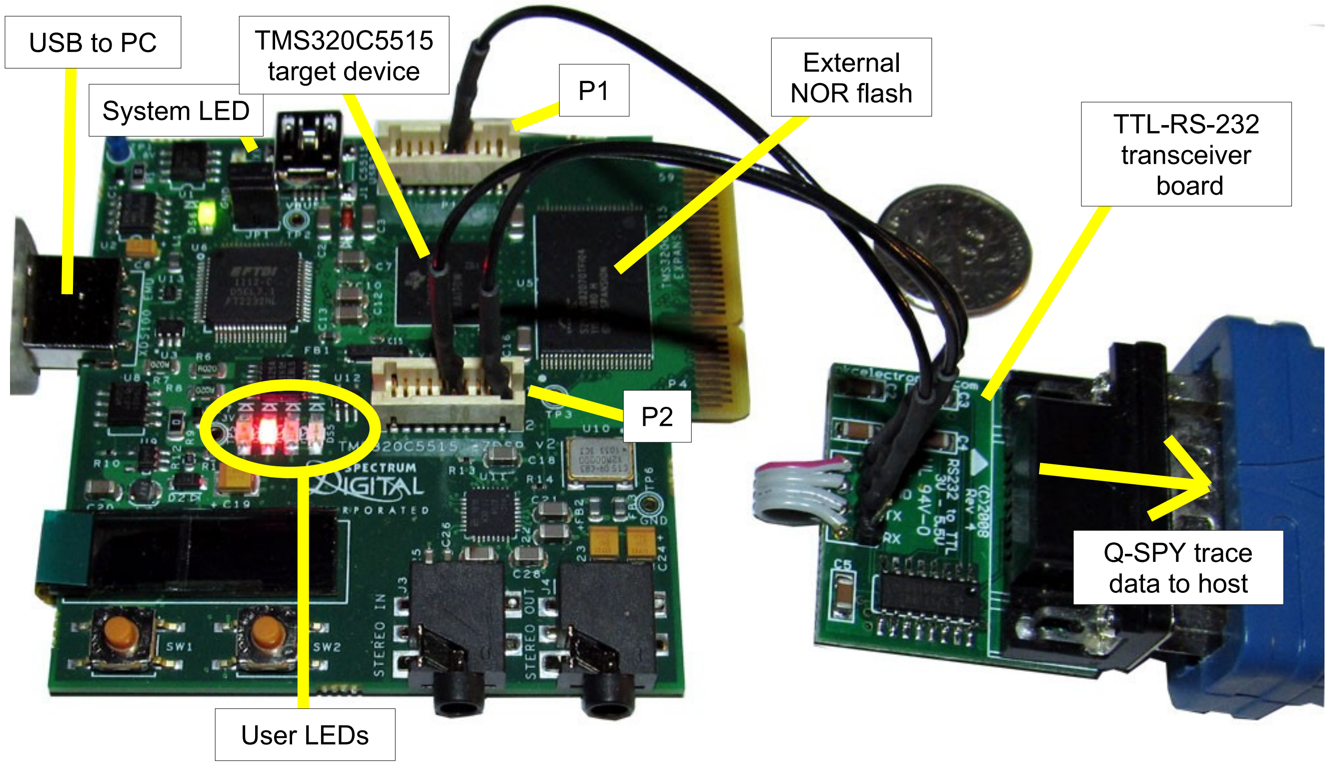
Document Revision B
February 2012

Table of Contents

1 Introduction

This **QP Development Kit**™ (QDK) describes how to use the QP™ state machine framework with the Texas Instruments TMS320C55x DSPs and the Texas Instruments C5500 C/C++ compiler.

Figure 1: Spectrum Digital eZdsp USB Stick C5515 and the TTL-RS-232 transceiver board



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

1. Spectrum Digital eZdsp-C5515 USB Stick
2. Texas Instruments Code Composer Studio IDE v**5.1**
3. Texas Instruments C5500 Optimizing C/C++ Compiler v**4.4.0**
4. QP/C or QP/C++ **4.4.00** or higher

NOTE: This QDK covers both the C and C++ version of QP. The concrete code examples are taken from the C version, but the C++ version is essentially identical except for some trivial syntax differences.

As shown in Figure 1, the eZdsp-C5515 USB Stick includes the integrated USB J-tag debugger and the target TMS320C5515 target device as well as the external NOR flash. However, the described QP port should be applicable to almost all TMS320C55x devices.

NOTE: For the QS (Q-SPY) software tracing output, you also need a TTL-to-RS-232 transceiver board. Such boards are available from a number of vendors. For example Figure 1 shows the RS232 to TTL converter board 3.3V to 5V from NKC Electronics ($9.99 http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html).

1.1 What's Included in the QDK-TMS320C55x-C5500?

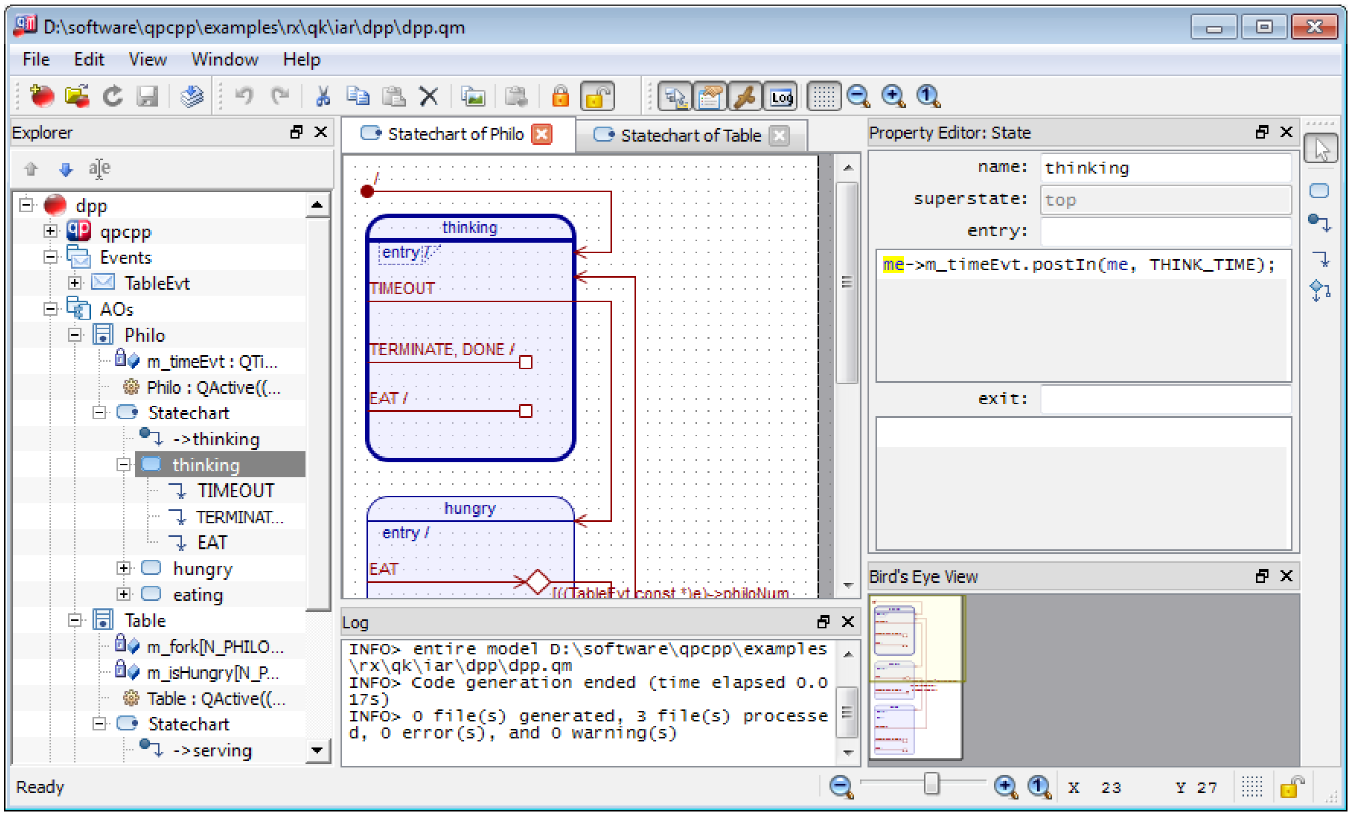
This QDK provides a basic Board Support Package (BSP) for TMS320C55x and two versions of the Dining Philosopher Problem (DPP) example application described in Chapter 7 of [PSiCC2] as well as in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08] (included in the QDK distribution):

- The Dining Philosopher Problem example with the cooperative "vanilla" kernel; and
- The Dining Philosopher Problem example with the preemptive QK kernel described in Chapter 10 of [PsiCC2].
- Texas Instruments Chip Support Library (CSL) for C55x devices in the directory `c55xx_csl`, which allows accessing the C55x peripherals in the standard way documented in the TI literature [SPUR433J].
- The `hex_util` utility for converting the .out COFF files into binary format to program in ROM
- The `nor_writer` project to perform programming of the NOR-Flash device on the USBSTK5515 board

NOTE: The significant parts of the source code (files `dpp.h`, `philo.c`, and `table.c`) have been generated by the QM™ modeling tool from the `dpp.qm` model, which is the same for the Vanilla and QK versions of the DPP application (see Figure 2). These files can be edited by hand (after unchecking the read-only property), but the changes made at the code level won't be incorporated back into the model.

NOTE: This QDK Manual covers both the 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.

Figure 2: The DPP example model opened in the QM™ modeling tool

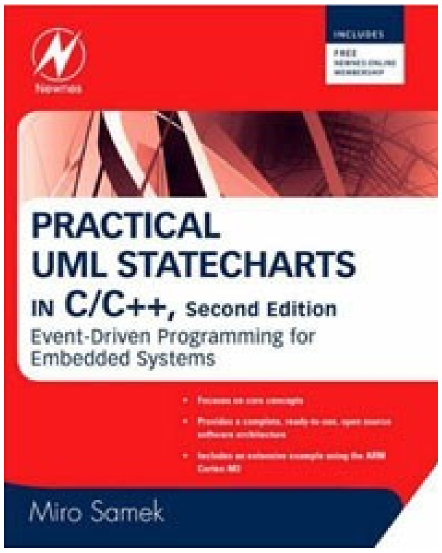


```
name:       thinking
superstate: top
entry:
me->m_timeEvt.postIn(me, THINK_TIME);
exit:
```

```
INFO> entire model D:\software\qpcpp\examples
\rx\qk\iar\dpp\dpp.qm
INFO> Code generation ended (time elapsed 0.0
17s)
INFO> 0 file(s) generated, 3 file(s) processe
d, 0 error(s), and 0 warning(s)
```

1.2 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++ **without big 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).



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. QP can work with or without a traditional RTOS or OS. In the simplest configuration, QP can completely **replace** a traditional RTOS. QP can manage up to 63 concurrently executing tasks structured as state machines (called active objects in UML).

1.3 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™ is available for 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™.

1.4 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.

1.5 Licensing QM™

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

2 Getting Started

This section describes how to install, build, and use QDK-TMS320C55x-C5500.

2.1 Software Installation

The QDK code is distributed in a ZIP archive (`qdk_tms320c55x-c5500.zip`. You can decompress the archive into any directory. The installation directory you choose will be referred henceforth as `<qpc>`. The following Listing 1 shows the directory structure and selected files included in the QP distribution. (Please note that the QP directory structure is described in detail in a separate Quantum Leaps Application Note: "QP Directory Structure").

NOTE: Every QDK™ contains only example(s) pertaining to the specific MCU and compiler, but does not include the platform-independent baseline code of QP™, which is available for a separate download. It is strongly recommended that you read Chapter 12 in [PSiCC2] before you start with this QDK™.

This QDK contains also the Texas Instruments Chip Support Library (CSL) for C55x devices in the directory `c55xx_csl`, which is copied from the code accompanying TI Application Note "TMS320C55x Chip Support Library API Reference Guide" [SPUR433J]. This Texas Instruments code allows accessing the C55x peripherals in the standard way documented in the TI literature.

Listing 1: Selected QP directories and files after installing QDK-TMS320C55x-C5500

```
<qpc>/                   - QP/C Root Directory
  |
  +-doc\
  | +-AN_DPP.pdf         - Application Note "Dining Philosopher Problem Example"
  | +-QDK_TMS320C55x-C5500.pdf - This QDK Manual "QDK TMS320C55x-C5500"
  |
  +-ports/                 - QP ports
  | +-tms320c55x\          - TMS320C55x ports
  | | +-vanilla\           - Ports to the "vanilla" (non-preemptive) kernel
  | | | +-c5500\           - TI's C5500 compiler
  | | | | +-dbg\           - Debug build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-rel\             - Release build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-spy\             - Spy build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-src\           - Platfom-specific source directory
  | | | | | +-qf_port.c    - Platfom-specific source code for the QF port
  | | | | | +-qs_port.c    - Platfom-specific source code for the QS port
  | | | | +-make_5515_large.bat  - Batch to build QP for the C5515 core, large model
  | | | | +-qep_port.h     - QEP platform-dependent public include
  | | | | +-qf_port.h      - QF platform-dependent public include
  | | | | +-qs_port.h      - QS platform-dependent public include
  | | |
  | | +-qk\                - QK (preemptive kernel) ports
  | | | +-c5500\           - TI's C5500 compiler
  | | | | +-dbg\           - Debug build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-rel\              - Release build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-spy\              - Spy build
  | | | | | +-qp5515_large.lib  - QP library for the C5515 core, large memory model
  | | | | +-src\            - Platfom-specific source directory
  | | | | | +-qf_port.c     - Platfom-specific source code for the QF port
  | | | | | +-qs_port.c     - Platfom-specific source code for the QS port
  | | | | +-make_5515_large.bat  - Batch to build QP for the C5515 core, large model
  | | | | +-qep_port.h      - QEP platform-dependent public include
  | | | | +-qf_port.h       - QF platform-dependent public include
  | | | | +-qk_port.h       - QK platform-dependent public include
  | | | | +-qs_port.h       - QS platform-dependent public include
  |
  +-examples\               - subdirectory containing the QP example files
  | +-tms320C55x\           - TMS320C55x ports
  | | +-c55xx_csl\          - Texas Instruments Chip Support Library for C55x DSP
  | | | +-build\
  | | | | +-cslVC5505\      - CCSv5 project for building the library
  | | | | | +-Debug\
  | | | | | | +-cslVC5505_Lib.lib - Debug version of the CSL for C55x
  | | | | | +-Release\
  | | | | | | +-cslVC5505_Lib.lib - Release version of the CSL for C55x
  | | | +-inc\             - header files for the CSL library
  | | | +-src\             - source files for the CSL library
  | | | +-hex_utility\     - utility for generating images to burn into Flash
  | | | | +-hex55.exe      - utility for generating images to burn into Flash
  | | | | +-dpp.cmd        - example command file for converting DPP image
  | | | +-nor_writer\      - CCSv5 project for writing images to the NOR flash
  | | | |
  | | +-vanilla\           - Ports to the non-preemptive “vanilla” kernel
  | | | +-C5500\           - TI’s C5500 compiler
  | | | | +-dpp_usbstk5515\ - DPP example for (non-preemptive)
  | | | | | +-Debug\       - directory containing the Debug build
  | | | | | | +-dpp.out    - image of the application
  | | | | | | +-dpp.map    - map file of the application
  | | | | | +-Release\     - directory containing the Release build
  | | | | | +-Spy\         - directory containing the Spy build
  | | | | | |
  | | | | | +-.ccsproject  - Eclipse CCS project for the DPP application
  | | | | | +-.cproject    - Eclipse CCS project for the DPP application
  | | | | | +-.project     - Eclipse CCS project for the DPP application
  | | | | | +-bsp.c        - BSP for the USBSTK5515 (non-preemptive)
  | | | | | +-bsp.h        - BSP header file
  | | | | | +-main.c       - the main function
  | | | | | +-dpp.qm       - QM model for the DPP application
  | | | | | +-dpp.h        - the DPP application header file
  | | | | | +-philo.c      - the Philosopher active objects
  | | | | | +-table.c      - the Table active object
  | | | | | +-C5515.cmd    - Linker script for the C5515 DSP
  | | | | | +-usbstk5515.ccxml - Debugger configuration for USBSK5515
  | | | |
  | | | | +-dpp-qk_usbstk5515\ - DPP example for (with preemptive QK kernel)
  | | | | | +-Debug\       - directory containing the Debug build
  | | | | | | +-dpp-qk.out - image of the application
  | | | | | | +-dpp-qk.map - map file of the application
  | | | | | +-Release\     - directory containing the Release build
  | | | | | +-Spy\          - directory containing the Spy build
  | | | | | |
  | | | | | +-.ccsproject  - Eclipse CCS project for the DPP application
  | | | | | +-.cproject    - Eclipse CCS project for the DPP application
  | | | | | +-.project     - Eclipse CCS project for the DPP application
  | | | | | +-bsp.c        - BSP for the USBSTK5515 (non-preemptive)
  | | | | | +-bsp.h        - BSP header file
  | | | | | +-main.c       - the main function
  | | | | | +-dpp.qm       - QM model for the DPP application
  | | | | | +-dpp.h        - the DPP application header file
  | | | | | +-philo.c      - the Philosopher active objects
  | | | | | +-table.c      - the Table active object
  | | | | | +-C5515.cmd    - Linker script for the C5515 DSP
  | | | | | +-usbstk5515.ccxml - Debugger configuration for USBSK5515
```

2.2 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for Debug, Release, and Spy build configurations are provided inside the `<qp>\ports\tms320C55x` directory. This section describes steps you need to take to rebuild the libraries yourself.

NOTE: To achieve commonality among different development tools, Quantum Leaps software does not use the vendor-specific IDEs, such as the Code Composer Studio IDE, for building the QP libraries. Instead, QP supports *command-line* build process based on simple batch scripts.

The code distribution contains the batch file `make_<core>_<model>.bat` for building all the libraries located in the `<qp>\`**`ports\`**`tms320c55x\...` directory. For example, to build the debug version of all the QP libraries for TMS320C55x, with the TI C5500 compiler, QK kernel, you open a console window on a Windows PC, change directory to `<qp>\`**`ports`**`\tms320C55x\qk\C5500\`, and invoke the batch by typing at the command prompt the following command:

```
make_5515_large.bat
```

The build process should produce the QP library in the location: `<qp>\`**`ports`**`\tms320C55x\qk\C5500\-dbg\`. The `make_5515_large.bat` files assume that the Texas Instruments toolset has been installed in the directory `C:\tools\TI\ccsv5\tools\compiler\c5500`.

NOTE: You need to adjust the symbol `TI_C5500` at the top of the batch scripts if you've installed the TI C5500 compiler into a different directory.

In order to take advantage of the QS ("spy") instrumentation, you need to build the QS version of the QP library. You achieve this by invoking the `make_5515_large.bat` utility with the "spy" target, like this:

```
make_5515_large.bat spy
```

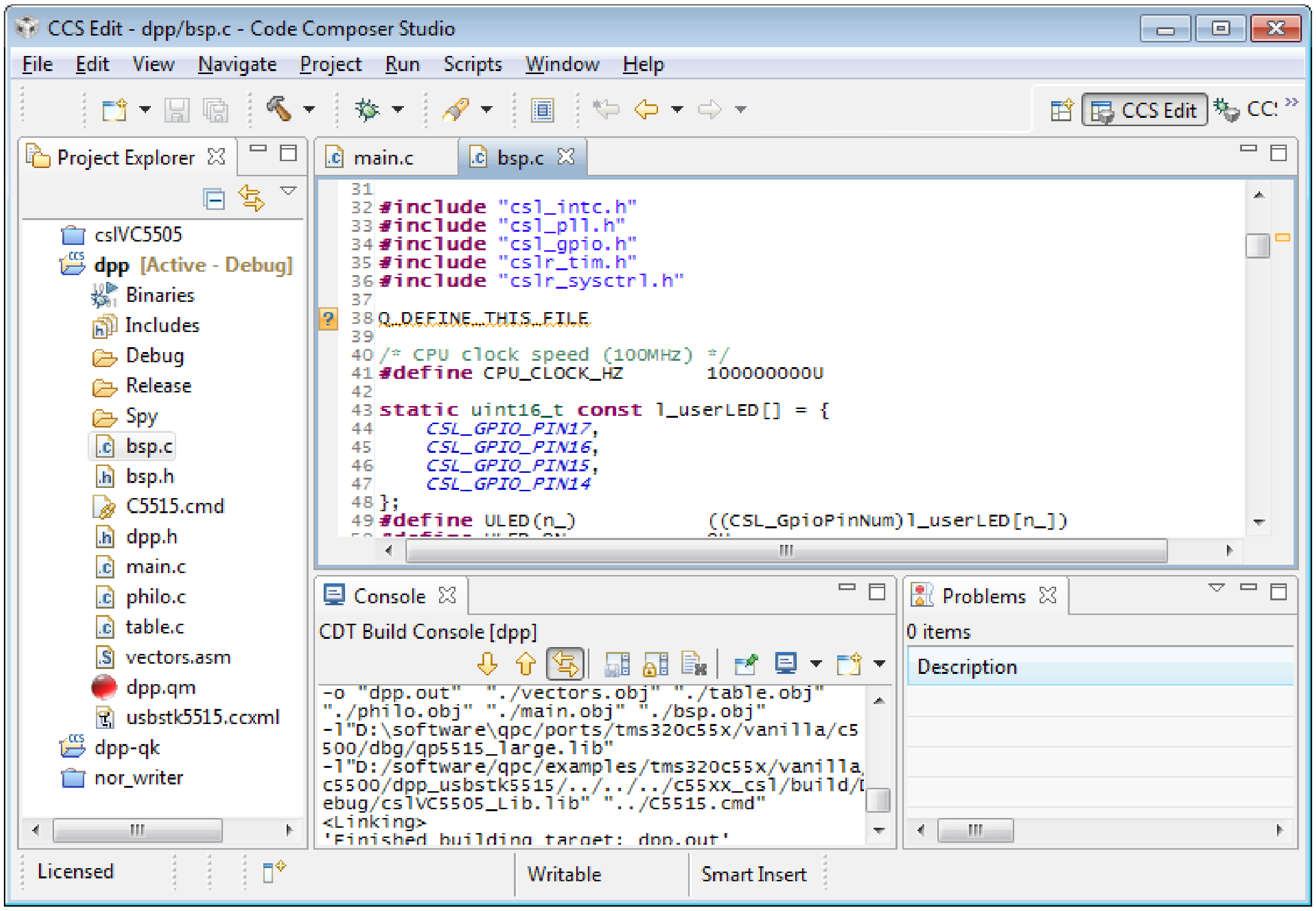
The make process should produce the QP libraries in the directory: `<qp>\ports\tms320C55x\qk\-C5500\`**`spy\`**.

You choose the build configuration by providing a target to the `make_5515_large.bat` batch. The default target is "`dbg`". Other targets are "`rel`", and "`spy`", respectively.

2.3 Building and Running the Examples

The examples included in this QDK are based on the standard DPP application implemented with active objects (see Quantum Leaps Application Note: “Dining Philosophers Problem Application” [QL AN-DPP 08] included in this QDK). The example directory contains the Eclipse-based Code Composer Studio v 5.1 (CCS5.1) projects that you can import into the CCS IDE, as shown in Figure 3.

Figure 3: The Code Composer Studio 5.1 with the DPP example



```
#include "csl_intc.h"
#include "csl_pll.h"
#include "csl_gpio.h"
#include "cslr_tim.h"
#include "cslr_sysctrl.h"

Q_DEFINE_THIS_FILE

/* CPU clock speed (100MHz) */
#define CPU_CLOCK_HZ          100000000U

static uint16_t const l_userLED[] = {
    CSL_GPIO_PIN17,
    CSL_GPIO_PIN16,
    CSL_GPIO_PIN15,
    CSL_GPIO_PIN14
};
#define ULED(n_)              ((CSL_GpioPinNum)l_userLED[n_])
```

```
-o "dpp.out"  "./vectors.obj" "./table.obj"
"./philo.obj" "./main.obj" "./bsp.obj"
-l"D:\software\qpc/ports/tms320c55x/vanilla/c5
500/dbg/qp5515_large.lib"
-l"D:/software/qpc/examples/tms320c55x/vanilla
c5500/dpp_usbstk5515/../../../c55xx_csl/build/D
ebug/cslVC5505_Lib.lib" "../C5515.cmd"
<Linking>
'Finished building target: dpp.out'
```

2.3.1 *Building the Example Projects in the Code Composer Studio*

The following instructions assume that you have already installed and configured the Code Composer Studio 5.x on your host PC.

1. Launch CCS and import the project located in `<qp>\examples\tms320C55x\vanilla\C5500\-dpp_usbstk5515\` directory.

2. Build the project by select **Project | Build Project** menu or by pressing Ctrl-B. The project file contains three build configurations Debug, Release, and Spy. You can select the build configuration by means of the **Project | Build Configurations | Set Active** menu.

2.3.2 *Running the DPP Examples*

The linker script `C5515.cmd` (see Listing 1), links the image into RAM. The CCS debugger, configured for the USBSTK5515 board loads the image to RAM and stops at the `main()` function.

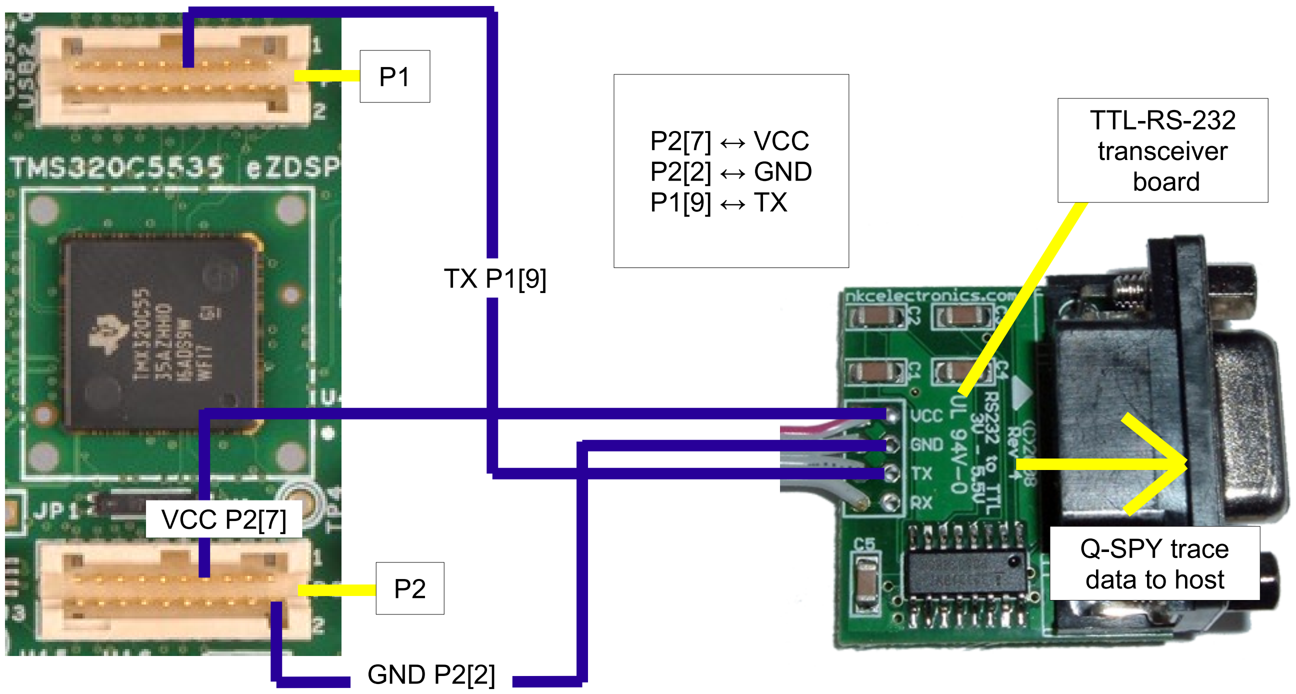
NOTE: The USBSTK5515 comes factory-programmed with a demo application, which initializes the OLED display with a scrolling text. You might be surprised that the OLED keeps animating the text, even when you stop the main processor and download your own code to it. This is normal, because the OLED display has its own memory and keeps scrolling the text independently from the main CPU.

NOTE: Because the program is loaded to RAM, it is gone when you stop debugging and disconnect the board.

After successfully programming the device, you can run the program by selecting the Debug | Run menu (F5), or by clicking on the Run button. The four User LEDs of the USBSTK5515 should start blinking (see Figure 1). The LEDs correspond to Philosophers 0-3 (Philosopher number 4 is not shown). Please refer to the Quantum Leaps Application Note [QL AN-DPP 08] for details of the DPP example.

For the QS (Q-SPY) software tracing output, you need to connect a TTL to RS-232 transceiver to the USBSTK5515 board, as shown in Figure 4. The figure shows the RS232 to TTL converter board 3.3V to 5V from NKC Electronics (http://www.nkcelectronics.com/rs232-to-ttl-converter-board-33v232335.html), but you can use any other equivalent board.

Figure 4: Connecting RS232-TTL board to the P1 and P2 connectors of the USBSTK5515 board.

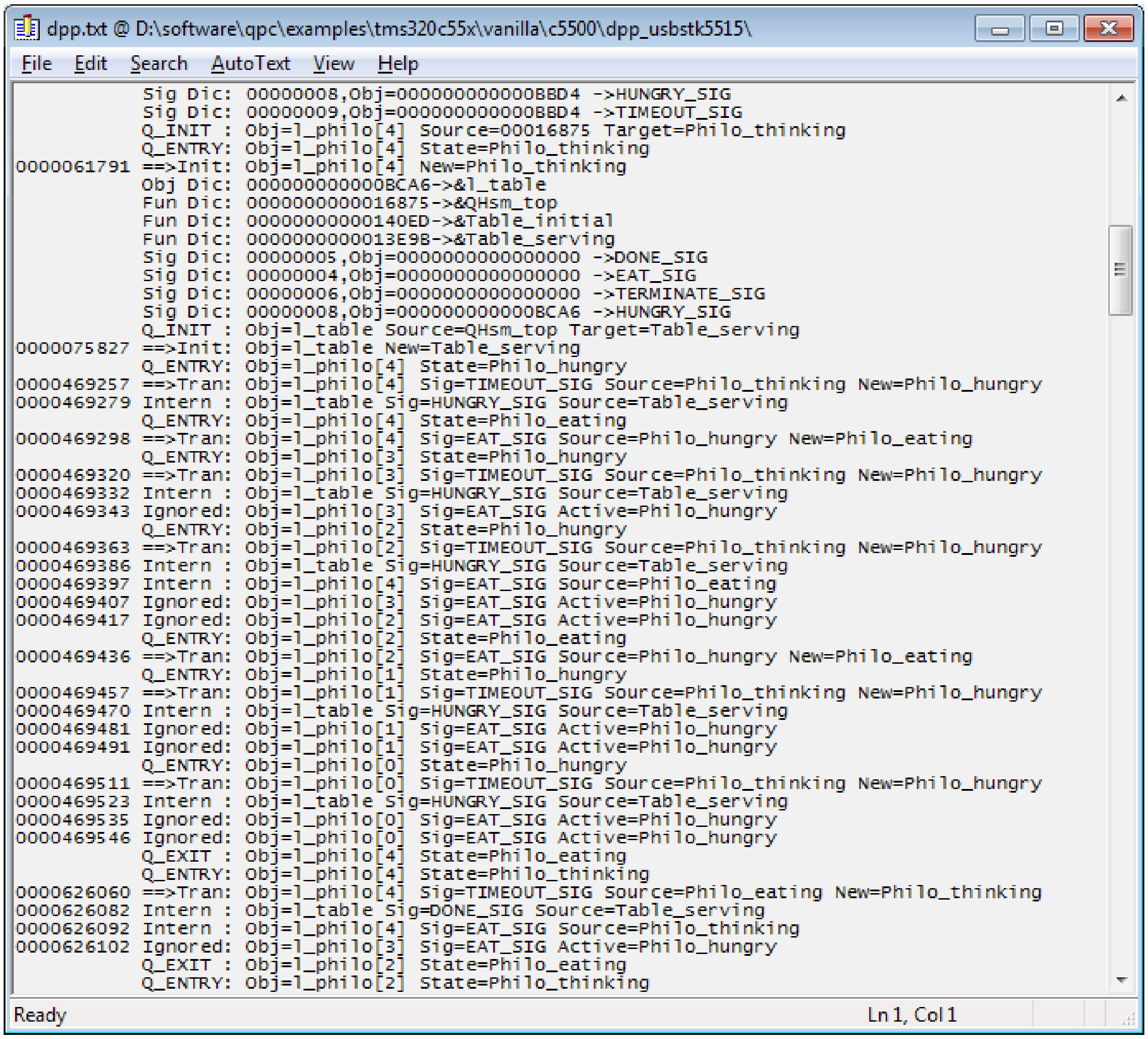


To see the QS software trace output, you also need to download the Spy build configuration to the target board. Next you need to launch the QSPY host utility to observe the output in the human-readable format. You launch the QSPY utility on a Windows PC as follows: (1) Change the directory to the QSPY host utility (QSPY is provided as part of Qtools collection) and execute:

```
qspy -cCOM1 -b115200
```

This will start the QSPY host application to listen on COM1 serial port with baud rate 115200. (Please use the actual virtual COM port number on your PC.) The screen shot in Figure 5 shows the QSPY output from the DPP run:

Figure 5: Screen shot from the QSPY output


dpp.txt @ D:\software\qpc\examples\tms320c55x\vanilla\c5500\dpp_usbstk5515\
File Edit Search AutoText View Help
Ready
Ln 1, Col 1

```
             Sig Dic: 00000008,Obj=000000000000BBD4 ->HUNGRY_SIG
             Sig Dic: 00000009,Obj=000000000000BBD4 ->TIMEOUT_SIG
             Q_INIT : Obj=l_philo[4] Source=00016875 Target=Philo_thinking
             Q_ENTRY: Obj=l_philo[4] State=Philo_thinking
0000061791 ==>Init: Obj=l_philo[4] New=Philo_thinking
             Obj Dic: 000000000000BCA6->&l_table
             Fun Dic: 0000000000016875->&QHsm_top
             Fun Dic: 00000000000140ED->&Table_initial
             Fun Dic: 0000000000013E9B->&Table_serving
             Sig Dic: 00000005,Obj=0000000000000000 ->DONE_SIG
             Sig Dic: 00000004,Obj=0000000000000000 ->EAT_SIG
             Sig Dic: 00000006,Obj=0000000000000000 ->TERMINATE_SIG
             Sig Dic: 00000008,Obj=000000000000BCA6 ->HUNGRY_SIG
             Q_INIT : Obj=l_table Source=QHsm_top Target=Table_serving
0000075827 ==>Init: Obj=l_table New=Table_serving
             Q_ENTRY: Obj=l_philo[4] State=Philo_hungry
0000469257 ==>Tran: Obj=l_philo[4] Sig=TIMEOUT_SIG Source=Philo_thinking New=Philo_hungry
0000469279 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
             Q_ENTRY: Obj=l_philo[4] State=Philo_eating
0000469298 ==>Tran: Obj=l_philo[4] Sig=EAT_SIG Source=Philo_hungry New=Philo_eating
             Q_ENTRY: Obj=l_philo[3] State=Philo_hungry
0000469320 ==>Tran: Obj=l_philo[3] Sig=TIMEOUT_SIG Source=Philo_thinking New=Philo_hungry
0000469332 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000469343 Ignored: Obj=l_philo[3] Sig=EAT_SIG Active=Philo_hungry
             Q_ENTRY: Obj=l_philo[2] State=Philo_hungry
0000469363 ==>Tran: Obj=l_philo[2] Sig=TIMEOUT_SIG Source=Philo_thinking New=Philo_hungry
0000469386 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000469397 Intern : Obj=l_philo[4] Sig=EAT_SIG Source=Philo_eating
0000469407 Ignored: Obj=l_philo[3] Sig=EAT_SIG Active=Philo_hungry
0000469417 Ignored: Obj=l_philo[2] Sig=EAT_SIG Active=Philo_hungry
             Q_ENTRY: Obj=l_philo[2] State=Philo_eating
0000469436 ==>Tran: Obj=l_philo[2] Sig=EAT_SIG Source=Philo_hungry New=Philo_eating
             Q_ENTRY: Obj=l_philo[1] State=Philo_hungry
0000469457 ==>Tran: Obj=l_philo[1] Sig=TIMEOUT_SIG Source=Philo_thinking New=Philo_hungry
0000469470 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000469481 Ignored: Obj=l_philo[1] Sig=EAT_SIG Active=Philo_hungry
0000469491 Ignored: Obj=l_philo[1] Sig=EAT_SIG Active=Philo_hungry
             Q_ENTRY: Obj=l_philo[0] State=Philo_hungry
0000469511 ==>Tran: Obj=l_philo[0] Sig=TIMEOUT_SIG Source=Philo_thinking New=Philo_hungry
0000469523 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000469535 Ignored: Obj=l_philo[0] Sig=EAT_SIG Active=Philo_hungry
0000469546 Ignored: Obj=l_philo[0] Sig=EAT_SIG Active=Philo_hungry
             Q_EXIT : Obj=l_philo[4] State=Philo_eating
             Q_ENTRY: Obj=l_philo[4] State=Philo_thinking
0000626060 ==>Tran: Obj=l_philo[4] Sig=TIMEOUT_SIG Source=Philo_eating New=Philo_thinking
0000626082 Intern : Obj=l_table Sig=DONE_SIG Source=Table_serving
0000626092 Intern : Obj=l_philo[4] Sig=EAT_SIG Source=Philo_thinking
0000626102 Ignored: Obj=l_philo[3] Sig=EAT_SIG Active=Philo_hungry
             Q_EXIT : Obj=l_philo[2] State=Philo_eating
             Q_ENTRY: Obj=l_philo[2] State=Philo_thinking
```

2.4 Programming the NOR-Flash

As mentioned in the previous section, the CCS debugger loads the program image directly to RAM of the C5515 device and executes it from there. But at some point, you might like to burn the program **permanently** into the ROM, which in case of the USBSTK5515 board is the external NOR Flash (see Figure 1). This QDK provides all tools you need to program the NOR flash.

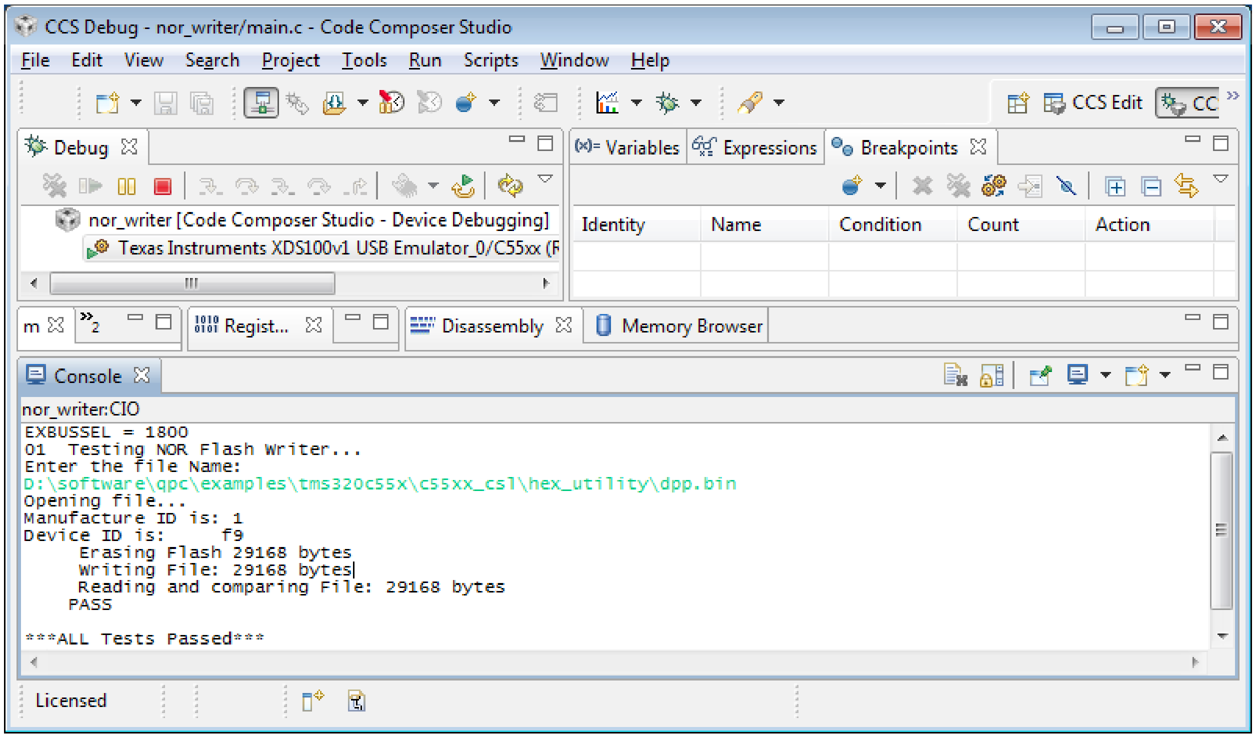
First, you need to convert the output file `.out` generated by the linker (COFF format) to the binary format suitable to burning into flash. The utility to do such conversion is called `hex55.exe` and is located in the directory `<qpc>\examples\tms320c55x\c55xx_csl\hex_utility\` (see Listing 1). This directory contains also the `dpp.cmd` "command file" with all the options required for the USBSTK5515 board. You need to edit this command file to convert your own `.out` files. You call `hex55.exe` as follows:

```
hex55.exe dpp.cmd
```

In the next step you will program this image into the NOR-Flash. The application to do this is called `nor_writer` and is a C5515 project that you can import into the CCS v5 IDE. The projects is located in the directory `<qpc>\examples\tms320c55x\c55xx_csl\nor_writer\` (see Listing 1). This application uses semi-hosting I/O to input data from the host and copy it to the target using the debugger connection.

Once you build the `nor_writer` project, you load it to the USB5515 board via the debugger and start executing it. The application opens a console, into which you type the full path of the image to burn into NOR-Flash and hit Return (see Figure 6). After a short while, the application should print "`***ALL Tests Passed***`" to the console. The image is now permanently programmed and you can power-cycle the board.

Figure 6: Screen shot from the run of the nor_writer project



```
EXBUSSEL = 1800
01  Testing NOR Flash Writer...
Enter the file Name:
D:\software\qpc\examples\tms320c55x\c55xx_csl\hex_utility\dpp.bin
Opening file...
Manufacture ID is: 1
Device ID is:      f9
     Erasing Flash 29168 bytes
     Writing File: 29168 bytes
     Reading and comparing File: 29168 bytes
    PASS

***ALL Tests Passed***
```

3 Non-Preemptive “Vanilla” Port

The “vanilla” port shows how to use QP™ on a bare metal TMS320C55x-based system with the cooperative “vanilla” kernel. In this version you’re using the non-preemptive kernel built-into the QF framework and you are not using the QK component.

3.1 The qep_port.h Header File

The QEP header file for the TMS320C55x port is located in `<qp>\ports\tms320C55x\vanilla\-C5500\qep_port.h`. Listing 2 shows the `qep_port.h` header file for TMS320C55x.

NOTE: The QP port to the cooperative “Vanilla” kernel `qep_port.h` is generic and should not need to change for other TMS320C55x applications.

Listing 2: qep_port.h header file for the non-preemptive QP configuration and C5500 compiler

```
                                      /* 2-byte (64K) signal space, see NOTE01 */
(1) #define Q_SIGNAL_SIZE 2

                  /* Exact-width types. WG14/N843 C99 Standard, Section 7.18.1.1 */
(2) #include <stdint.h>

(3) typedef int16_t  int8_t;                                     /* see NOTE01 */
(4) typedef uint16_t uint8_t;

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

    /*****************************************************************************
    * NOTE01:
    * The TMS320C55x cannot separately address 8-bit bytes (the smallest
    * separately-addressable entity is a 16-bit word). Therefore the TMS320C55x
    * char is 16 bits (to make it separately addressable).
    */
```

(1) The macro `Q_SIGNAL_SIZE` defines the size (in bytes) of event signals. The allowed values are 1, 2, or 4 bytes. Here the value of `Q_SIGNAL_SIZE` is set to 2, so that QP will use `uint16_t`, which is the smallest separately addressable entity on TMS320C55x DSP.

(2) The C99-standard exact-width integer types are included from the `<stdint.h>` header file.

(3-4) The standard `<stdint.h>` header file does not provide definitions for `uint8_t` or `int8_t`. However QP uses these integer types, so they are approximated with 16-bit wide types.

NOTE: The TMS320C55x `char` is 16 bits (to make it separately addressable). Therefore the types `int8_t` and `uint8_t` are actually 16-bit integers on TMS320C55x. This yields results you may not expect; for example: `sizeof(int8_t) == sizeof(uint8_t) == sizeof(int16_t) == sizeof(uint16_t) == 1`.

3.2 The qf_port.h Header File

The QF header file for the TMS320C55x port is located in `<qp>\ports\tms320C55x\vanilla\C5500\qf_port.h`. This file specifies the interrupt locking/unlocking policy (QF critical section) as well as the configuration constants for QF (see Chapter 8 in [PSiCC2]).

The most important porting decision you need to make in the `qf_port.h` header file is the policy for disabling and enabling interrupts. The TMS320C55x DSP needs the policy of "saving and restoring interrupt status" (see Section 7.3.1 of the book "Practical UML Statecharts in C/C++, Second Edition" [PSiCC2]). Listing 3 shows the `qf_port.h` header file for TMS320C55x.

Listing 3: The qf_port.h header file for TMS320C55x.

```
          /* The maximum number of active objects in the application, see NOTE01 */
 (1) #define QF_MAX_ACTIVE               8

 (2) #define QF_EVENT_SIZ_SIZE           2
 (3) #define QF_EQUEUE_CTR_SIZE          2
 (4) #define QF_MPOOL_SIZ_SIZE           2
 (5) #define QF_MPOOL_CTR_SIZE           2
 (6) #define QF_TIMEEVT_CTR_SIZE         2

                                                /* QF interrupt disable/enable */
 (5) #define QF_INT_DISABLE()            _disable_interrupts()
 (6) #define QF_INT_ENABLE()             _enable_interrupts()

                                       /* QF critical section policy, see NOTE02 */
 (7) #define QF_CRIT_STAT_TYPE           unsigned int
 (8) #define QF_CRIT_ENTRY(stat_)        ((stat_) = _disable_interrupts())
 (9) #define QF_CRIT_EXIT(stat_)         _restore_interrupts(stat_)

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

(10) void QF_zero(void);                            /* zero the .bss QF variables */
(11) void bzero(uint8_t *ptr, uint16_t len);     /* helper to clear other objects */
```

(1) The `QF_MAX_ACTIVE` specifies the maximum number of active object priorities in the application. You always need to provide this constant. Here, `QF_MAX_ACTIVE` is set to 8 to conserve some RAM, but you can increase this value up to 63 inclusive.

(2-6) The other object sizes in QF are all set to 2, so that QP will use `uint16_t` to represents these objects, which is the most natural entity in the TMS320C55x DSP.

NOTE: The `qf_port.h` header file does not change the default settings for all the rest of various object sizes inside QF. Please refer to Chapter 8 of [PSiCC2] for discussion of all configurable QF parameters.

(5) The interrupt disable macro calls the intrinsic function `_disable_interrupts()`, which resolves to the instruction `setc INTM` (set the INTM mask in ST1 register).

(6) The interrupt disable macro calls the intrinsic function `_enable_interrupts()`, which resolves to the single instruction `clrc INTM` (clear the INTM mask in ST1 register).

(7) The `QF_CRIT_STAT_TYPE` is defined, which means that the policy of "saving and restoring critical section status" is applied (see Section 7.2 in [PSiCC2]).

NOTE: This policy allows nesting of critical sections. The TMS320C55x automatically disables interrupts upon the entry to an ISR by setting the INTM mask. This means that the whole body of an ISR is a critical section. Therefore, in order to call QP services from ISRs, this port uses policy that allows nesting critical sections.

(8) Entry to the critical section calls again the intrinsic function `_disable_interrupts()`, but this time the interrupt status returned by this function is saved into the `stat_` argument.

(9) Exit from the critical section calls the intrinsic function `_restore_interrupts()`, which restores the interrupt status from the `stat_` argument.

3.2.1 Additional Functions for Clearing Uninitialized Variables

The `qf_port.h` (Listing 3(10-11)) file additionally declares two functions that are intended to clear the critical variables that get **not** initialized to zero in the startup code.

CAUTION: The standard TI startup code (`c_int00`) does **not** clear to zero all the uninitialized variables, as required by the C-standard. This BSP includes a workaround for this non-standard behavior for the QP objects. However, you need to be careful in **your** application code not to rely on clearing the unitialized variables.

3.3 ISRs in the Non-Preemptive "Vanilla" Configuration

The C5500 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. The only QP-specific requirement is that you provide a periodic time-tick ISR and you invoke `QF_tick()` in it. The ISRs are located in the `bsp.c` file found in the application directory.

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

```
(1) interrupt void TINT_isr(void) {
(2)     CSL_SYSCTRL_REGS->TIAFR |= 0x0001U;                 /* clear Timer0 bit */

(3)     QF_TICK(&l_TINT_isr);                          /* handle the QF time events */
    }
```

(1) The ISR must be declared with the extended keyword `interrupt`. An ISR must also be a `void (void)` function.

NOTE: All interrupts must also be "plugged" into the interrupt vector table, as described later.

(2) The interrupt source must be cleared, if necessary.

(3) The time-tick ISR must invoke `QF_TICK()`, and can also perform other actions.

3.4 The qf_port.c Source File

The need for a port-specific implementation of the QF framework arises only because of the non-standard behavior of the Texas Instruments startup code, which does **not** clear the unitialized static variables, as required by the C standard. The platform-specific implementation file `qf_port.c`, shown in Listing 5, implements functions `bzero()`, `QF_zero()`, and overrides function `QF_psInit()`, to clear the critical variables.

Listing 5: The qf_port.c source file

```
#include "qf_pkg.h"
#include "qassert.h"

Q_DEFINE_THIS_MODULE(qf_port)

/* Global objects ----------------------------------------------------------*/
QSignal      QF_maxSignal_;
QSubscrList *QF_subscrList_;

/*..........................................................................*/
void bzero(uint8_t *ptr, uint16_t len) {
    while (len-- != (uint16_t)0) {
        *ptr++ = (uint8_t)0;
    }
}
/*..........................................................................*/
void QF_zero(void) {                                         /* see NOTE01 */
    extern uint8_t QF_intLockNest_;

    QF_intLockNest_      = (uint8_t)0;
    QF_maxPool_          = (uint8_t)0;
    QF_timeEvtListHead_ = (QTimeEvt *)0;
    bzero((uint8_t *)QF_active_,    sizeof(QActive*)*(QF_MAX_ACTIVE + 1));
    bzero((uint8_t *)&QF_readySet_, sizeof(QF_readySet_));
}
/*..........................................................................*/
void QF_psInit(QSubscrList *subscrSto, QSignal maxSignal) {   /* see NOTE01 */
    QF_subscrList_  = subscrSto;
    QF_maxSignal_   = maxSignal;
    bzero((uint8_t *)subscrSto, maxSignal*sizeof(QSubscrList));
}
```

3.5 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.

The TMS320C55x DSP supports several power-saving levels (consult the [TI Piccolo 08] data sheet for details). The following piece of code shows the `QF_onIdle()` callback that puts TMS320C55x DSP into the IDLE power-saving mode. Please note that TMS320C55x architecture allows for an **atomic** setting the low-power mode and enabling interrupts at the same time.

Listing 6: QF_onIdle() for the non-preemptive (“vanilla”) QP port to TMS320C55x

```
(1) void QF_onIdle(void) {

(2)     USBSTK5515_LED_ON();                /* switch the System LED on and off */
        asm(" nop");
        . . .
        USBSTK5515_LED_OFF();

(3) #ifdef Q_SPY

        QF_INT_ENABLE();
        if (CSL_FEXT(l_uartObj.uartRegs->LSR, UART_LSR_THRE)) {
            uint16_t count = UART_FIFO_DEPTH;
            uint8_t const *pBuf;

            QF_INT_DISABLE();
            pBuf = QS_getBlock(&count);
            QF_INT_ENABLE();

            for (; count > 0U; --count, ++pBuf) {
                CSL_FSET(l_uartObj.uartRegs->THR, 7U, 0U, (*pBuf & 0xFFU));
            }
        }

(4) #elif defined NDEBUG
        /* Put the CPU and peripherals to the low-power mode.
        * you might need to customize the clock management for your application,
        * see the datasheet for your particular TMS320C5500 device.
        */
(5)     asm(" IDLE");
(6)     QF_INT_ENABLE();

(7) #else

(8)     QF_INT_ENABLE();

    #endif
    }
```

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

(2) The System LED is turned on and off, after a brief delay. These “glitches” cause the System LED to glow at the intensity, which is proportional to the rate of invocation of `QF_onIdle()`. Please note that System LED is on with interrupts disabled, so the interrupt processing time does not add to the ON-time.

(3) If QSPY software tracing is enabled, the `QF_onIdle()` callback tries to output one byte from the trace buffer to the SCI. See Section 6 for more information about QSPY software tracing.

(4) The low-power mode stops the CPU clock, so it can interfere with the debugger. Here, the low-power mode is activated only in the Release build configuration when the macro NDEBUG is defined.

(5) The IDLE mode is activated by executing the `IDLE` instruction.

NOTE: The TMS320C55x DSP allows for atomic transition to sleep mode with interrupts still **disabled**, as it should be done to avoid non-deterministic sleep (see [Samek 07a]).

(6) Only after the CPU wakes up, interrupts are unlocked to service the interrupt. (In the debug mode the machine is not put to sleep, because sleep mode interferes with debugging.)

NOTE: Every path through `QF_onIdle()` callback function must ultimately enable interrupts.

(7-8) In the debug mode the interrupts are simply re-enabled. (In the debug mode the machine is not put to sleep, because sleep mode might interfere with debugging.)

3.6 The qf_port.h Header File

The QF header file for the TMS320C55x port is located in `<qp>\ports\tms320C55x\vanilla\C5500\qf_port.h`. This file specifies the interrupt locking/unlocking policy (QF critical section) as well as the configuration constants for QF (see Chapter 8 in [PsiCC2]).

4 Preemptive QK Port

The QP port to TMS320C55x DSP 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 almost identical, except some small additions to the ISRs.

You configure and customize QK through the header file `qk_port.h` shown in Listing 7. Except for the highlighted fragments, the listing is identical as in the non-preemptive case (see Listing 3). The QK port header file for the TMS320C55x port is located in `<qp>\ports\tms320C55x\`**`qk`**`\C5500\qk_port.h`.

Listing 7: qk_port.h header file for the preemptive QP port with QK

```
(1) #define QK_ISR_ENTRY()  (++QK_intNest_)

(2) #define QK_ISR_EXIT()  do { \
(3)     --QK_intNest_; \
(4)     if (QK_intNest_ == (uint8_t)0) { \
(5)         uint8_t p = QK_schedPrio_(); \
(6)         if (p != (uint8_t)0) { \
(7)             QK_sched_(p); \
            } \
        } \
    } while (0)

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

(1) Every ISR needs to call the `QK_ISR_ENTRY()` macro upon the entry. The `QK_ISR_ENTRY()` macro increments the interrupt nesting up-down-counter `QK_intNest_`, which informs the QK kernel that the interrupt context is active.

(2) Every ISR needs to call the `QK_ISR_EXIT()` macro upon the exit.

(3) The `QK_ISR_EXIT()` macro decrements the interrupt nesting up-down-counter `QK_intNest_`.

(4) When the up-down-counter `QK_intNest_` drops to zero the ISR is about to return to the task-level. In that case the QK scheduler is called to process the asynchronous preemptions (see Chapter 10 in [PSiCC2]).

(5) The `QK_schedPrio_()` function finds the highest-priority task ready to run and returns its priority. If no such task is found, the function returns zero.

(6-7) If the highest-priority task ready to run exists, the QK scheduler `QK_sched()` is called.

4.1 qk_port.c Source File

The need for a port-specific implementation of the QK kernel arises only because of the non-standard behavior of the Texas Instruments startup code, which does **not** clear the unitialized static variables, as required by the C standard. The platform-specific implementation file `qk_port.c`, shown in Listing 8, implements the `QK_init()` function to clear the critical variables.

Listing 8: The qk_port.c source file

```
    #include "qk_pkg.h"
```

```
#include "qassert.h"

void QK_init(void) {
    QK_intNest_   = (uint8_t)0;
    QK_currPrio_ = (uint8_t)(QF_MAX_ACTIVE + 1);
    bzero((uint8_t *)&QK_readySet_, sizeof(QK_readySet_));

#ifndef QK_NO_MUTEX
    QK_ceilingPrio_ = (uint8_t)0;
#endif
}
```

4.2 Idle Loop Customization in the QK Port

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

NOTE: The idle callback `QK_onIdle()` is invoked with interrupts enabled (which is in contrast to `QF_onIdle()` that is invoked with interrupts disabled, see Section 6).

The following Listing 9 shows an example implementation of `QK_onIdle()` for the TMS320C55x DSP.

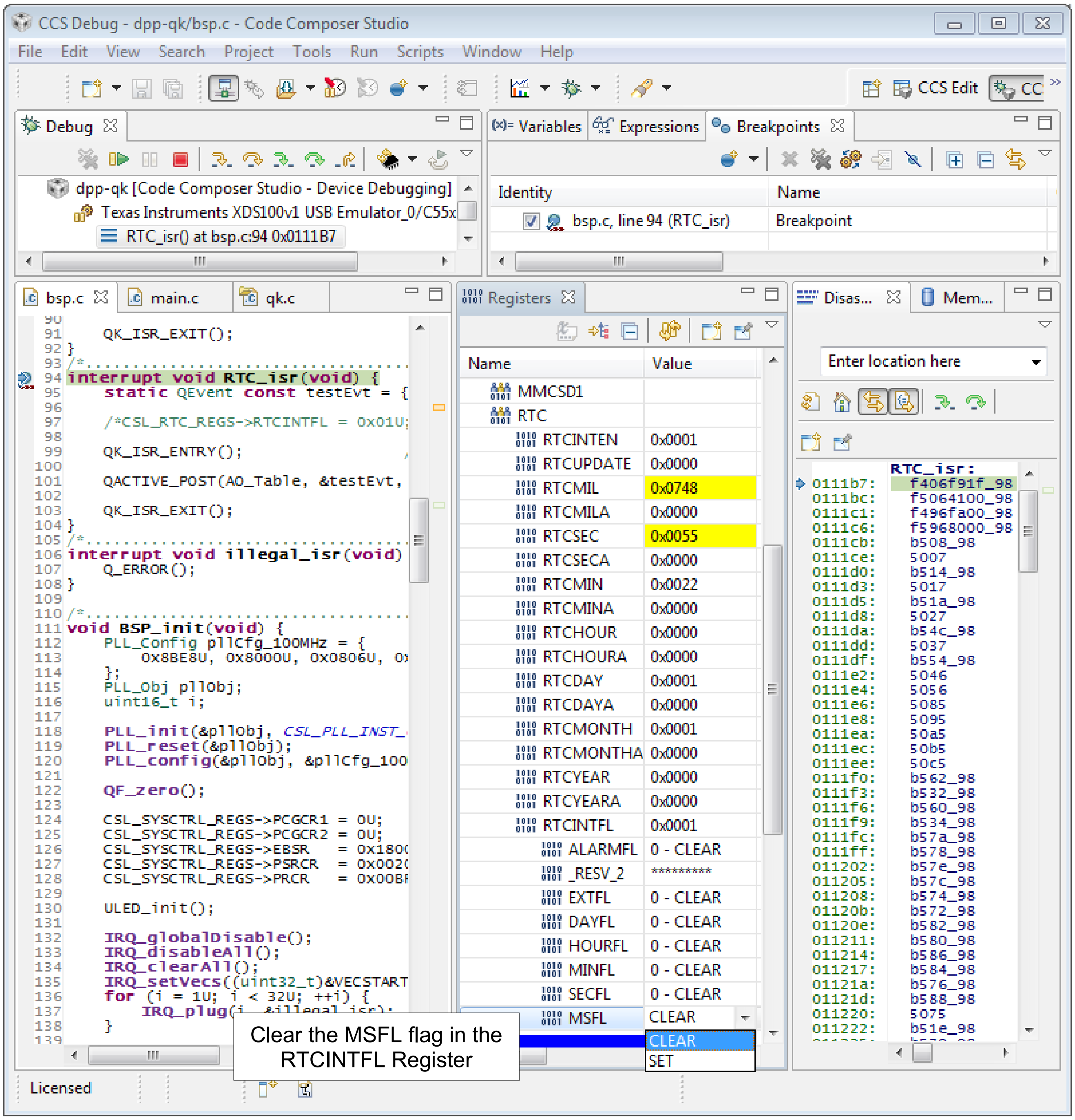
Listing 9: QK_onIdle() callback for TMS320C55x.

```
void QK_onIdle(void) {

    QF_INT_DISABLE();
    SLED_ON();                          /* switch the System LED on and off */
    asm(" nop");
    . . .
    SLED_OFF();
    QF_INT_ENABLE();

#ifdef Q_SPY
    if (CSL_FEXT(l_uartObj.uartRegs->LSR, UART_LSR_THRE)) {
        uint16_t count = UART_FIFO_DEPTH;
        uint8_t const *pBuf;

        QF_INT_DISABLE();
        pBuf = QS_getBlock(&count);
        QF_INT_ENABLE();

        for (; count > 0U; --count, ++pBuf) {
            CSL_FSET(l_uartObj.uartRegs->THR, 7U, 0U, (*pBuf & 0xFFU));
        }
    }
#elif defined NDEBUG
    asm(" IDLE");
#endif
}
```

4.3 Testing QK Preemption Scenarios

The DPP example application includes special instrumentation for convenient testing of various preemption scenarios, such as those illustrated in Figure 7.

Figure 7: Triggering the RTC interrupt from the CCS debugger.


CCS Debug - dpp-qk/bsp.c - Code Composer Studio
File
Edit
View
Search
Project
Tools
Run
Scripts
Window
Help
CCS Edit
CC
Debug
dpp-qk [Code Composer Studio - Device Debugging]
Texas Instruments XDS100v1 USB Emulator_0/C55x
RTC_isr() at bsp.c:94 0x0111B7
Variables
Expressions
Breakpoints
bsp.c
main.c
qk.c
Registers
Disas...
Mem...
Enter location here
Clear the MSFL flag in the RTCINTFL Register
Licensed

Identity	Name
✓ bsp.c, line 94 (RTC_isr)	Breakpoint

```
    QK_ISR_EXIT();
}
/*..........................................
interrupt void RTC_isr(void) {
    static QEvent const testEvt = {

    /*CSL_RTC_REGS->RTCINTFL = 0x01U;

    QK_ISR_ENTRY();                        /

    QACTIVE_POST(AO_Table, &testEvt,

    QK_ISR_EXIT();
}
/*..........................................
interrupt void illegal_isr(void)
    Q_ERROR();
}

/*..........................................
void BSP_init(void) {
    PLL_Config pllCfg_100MHz = {
        0x8BE8U, 0x8000U, 0x0806U, 0x
    };
    PLL_Obj pllObj;
    uint16_t i;

    PLL_init(&pllObj, CSL_PLL_INST_
    PLL_reset(&pllObj);
    PLL_config(&pllObj, &pllCfg_100

    QF_zero();

    CSL_SYSCTRL_REGS->PCGCR1 = 0U;
    CSL_SYSCTRL_REGS->PCGCR2 = 0U;
    CSL_SYSCTRL_REGS->EBSR   = 0x180
    CSL_SYSCTRL_REGS->PSRCR  = 0x002
    CSL_SYSCTRL_REGS->PRCR   = 0x00B

    ULED_init();

    IRQ_globalDisable();
    IRQ_disableAll();
    IRQ_clearAll();
    IRQ_setVecs((uint32_t)&VECSTART
    for (i = 1U; i < 32U; ++i) {
        IRQ_plug(i, &illegal_isr);
    }
```

Name	Value
MMCSD1	
RTC	
RTCINTEN	0x0001
RTCUPDATE	0x0000
RTCMIL	0x0748
RTCMILA	0x0000
RTCSEC	0x0055
RTCSECA	0x0000
RTCMIN	0x0022
RTCMINA	0x0000
RTCHOUR	0x0000
RTCHOURA	0x0000
RTCDAY	0x0001
RTCDAYA	0x0000
RTCMONTH	0x0001
RTCMONTHA	0x0000
RTCYEAR	0x0000
RTCYEARA	0x0000
RTCINTFL	0x0001
ALARMFL	0 - CLEAR
_RESV_2	*********
EXTFL	0 - CLEAR
DAYFL	0 - CLEAR
HOURFL	0 - CLEAR
MINFL	0 - CLEAR
SECFL	0 - CLEAR
MSFL	CLEAR
	CLEAR
	SET

```
         RTC_isr:
0111b7:    f406f91f_98
0111bc:    f5064100_98
0111c1:    f496fa00_98
0111c6:    f5968000_98
0111cb:    b508_98
0111ce:    5007
0111d0:    b514_98
0111d3:    5017
0111d5:    b51a_98
0111d8:    5027
0111da:    b54c_98
0111dd:    5037
0111df:    b554_98
0111e2:    5046
0111e4:    5056
0111e6:    5085
0111e8:    5095
0111ea:    50a5
0111ec:    50b5
0111ee:    50c5
0111f0:    b562_98
0111f3:    b532_98
0111f6:    b560_98
0111f9:    b534_98
0111fc:    b57a_98
0111ff:    b578_98
011202:    b57e_98
011205:    b57c_98
011208:    b574_98
01120b:    b572_98
01120e:    b582_98
011211:    b580_98
011214:    b586_98
011217:    b584_98
01121a:    b576_98
01121d:    b588_98
011220:    5075
011222:    b51e_98
```

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 RTC interrupt. The ISR for this interrupt is shown below:

```
interrupt void RTC_isr(void) {
    static QEvent const testEvt = { MAX_SIG, 0U, 0U };
    /*CSL_RTC_REGS->RTCINTFL = 0x01U;*/       /* clear the interrupt source */

    QK_ISR_ENTRY();                        /* inform the QK kernel about ISR entry */

    QACTIVE_POST(AO_Table, &testEvt, 0);      /* post a test event to Table */

    QK_ISR_EXIT();                          /* inform the QK kernel about ISR exit */
}
```

The ISR, as all interrupts in the system, 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.

Figure 7 hows how to trigger the RTC interrupt from the CCS debugger. From the debugger you need to first open the register window and select RTC registers. You scroll down to the RTCINTFL register and expand it as well. Finally, you get to the MSFL flag. When you right-click on the value (SET), you get a drop-down box, which allows you to CLEAR this flag. After selecting CLEAR, you hit the Enter key.

NOTE: The logic of triggering the RTC interrupt is actually inverted here. You trigger the RTC interrupt by clearing the RTCINTFL flag.

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 RTC test interrupt. Next, you need to free-run the code (don't use single stepping) so that the interrupt. You observe the order in which the breakpoints are hit. This procedure will become clearer after a few examples.

4.3.1 *Interrupt Preemption Test*

The first interesting test is verifying that interrupts do indeed preempt QK tasks. Please note that every QK task is either directly or indirectly launched from an ISR through the call to the QK scheduler occurring in the `QK_ISR_EXIT()` macro (see Listing 7). This happens before executing interrupt-return, so you might wonder whether or not the CPU can service interrupts in the middle of a task, as opposed to servicing interrupts only while executing the QK idle loop.

To test this scenario, you suspend the running DPP-QK application (click the Suspend button) and you set a breakpoint anywhere in the `Table` 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 `RTC_isr()` interrupt service routine. Next you trigger the RTC interrupt per the instructions given in the previous section. You click the Resume button or hit F8.

NOTE: The priority of the `Table` active object is higher than any of the `Philo` active objects, so you expect that QK should handle the test event posted to `Table` from the `RTC_isr()` **before** continuing to processing of any Philo event (asynchronous preemption).

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `RTC_isr ()` function, which means that RTC ISR preempted the `Philo` task. After verifying this, you click Resume again.
2. The second breakpoint hit is the one in the `Table` state machine, which means that the task continues after the ISR.

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

4.3.2 Task Preemption Test

The next interesting test is verifying that tasks can preempt each other. You suspend the DPP-QK application and set a breakpoint anywhere in the `Philo` 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 `RTC_isr()` interrupt handler and on the first instruction of the `Table_serving()` state handler. Next you trigger the RTC interrupt per the instructions given in the previous section. You click the Resume button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `RTC_isr()` function, which means that RTC ISR preempted the `Philo` task. After verifying this, you click Resume again.
2. The second breakpoint hit is the one in `Table_serving() state` handler, which means that the Table state machine is activated before the control returns to the preempted `Philo` task. After verifying this, you click Resume again.
3. The final breakpoint hit is the one inside the `Philo` state machine. This validates that the preempted task continues executing only after the preempting task (the `Table` state machine) completes.

4.3.3 Other Tests

Other interesting preemption test could check that interrupts cannot preempt interrupts on C5515, especially after any QP call from inside an ISR, because the INTM mask is never cleared throughout the duration of the ISR. This would confirm that the critical section policy assumed in this QP port is correct (see Section 3.2).

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

5 BSP for TMS320C55x DSP

This section highlights a few less-obvious aspects ot the Board Support Package for TMS320C55x DSPs, such as vector table initialization and the linker file.

5.1 Vector Table Initialization

The BSP contains the C55x vector table in assembly (file `vectors.asm`), which is aligned at 256-byte boundary and placed in the special RAM section "vectors". Additionally, this vector table is set-up with the CSL call `IRQ_setVecs()`, which sets up the Interrupt Vector Pointers (IVPD, IVPH). After hooking up, the vector table is pre-filled with the `illegal_isr()` vectors, which will cause an assertion in case any unexpected interrupt should occur.

Listing 10: Initialization of the vector table in bsp.c

```
#include "csl_intc.h"
void VECSTART(void);
. . .

interrupt void illegal_isr(void) {
    Q_ERROR();                                           /* assert an error */
}
 . .

void BSP_init(void) {
    . . .
    IRQ_setVecs((uint32_t)&VECSTART);                /* set the vector table */
    for (i = 1U; i < 32U; ++i) {                 /* pre-fill the Vector table */
        IRQ_plug(i, &illegal_isr);                        /* with illegal ISR */
    }

    /* plug in all ISRs into the vector table...*/
    IRQ_plug(TINT_EVENT, &TINT_isr);
    IRQ_plug(RTC_EVENT,  &RTC_isr);
    /* ... */
}
```

5.2 The Linker Command File

The BSP contains the `C5515.cmd` linker command file, which directs the linker how to place the various code and data sections in the memory of the TMS320C55x device. The linker command-file can also specify options to the linker, such as size of the stacks and the heap.

Listing 11: Linker command file C5515.cmd

```
-stack    1000       /* PRIMARY STACK SIZE    */
-sysstack 1000       /* SECONDARY STACK SIZE  */
-heap     0          /* HEAP AREA SIZE        */
```

```
MEMORY {
    MMR      (RW) : origin = 0000000h length = 0000c0h      /* MMRs  */
    DARAM    (RW) : origin = 00000c0h length = 00FF40h      /* on-chip DARAM */
    SARAM    (RW) : origin = 0010000h length = 03E000h      /* on-chip SARAM */

    EMIF_CS0 (RW) : origin = 0050000h  length = 07B0000h    /* mSDR          */
    EMIF_CS2 (RW) : origin = 0800000h  length = 0400000h    /* ASYNC1 : NAND */
    EMIF_CS3 (RW) : origin = 0C00000h  length = 0200000h    /* ASYNC2 : NAND */
    EMIF_CS4 (RW) : origin = 0E00000h  length = 0100000h    /* ASYNC3 : NOR  */
    EMIF_CS5 (RW) : origin = 0F00000h  length = 00E0000h    /* ASYNC4 : SRAM */
}

SECTIONS {
    vectors (NOLOAD)
    vector       : > DARAM  ALIGN = 256
    .bss         : > SARAM
    .stack       : > SARAM
    .sysstack    : > SARAM
    .text        : > SARAM  ALIGN = 4
    .cinit       : > SARAM
    .const       : {} > SARAM PAGE 0
    .data        : {} > SARAM PAGE 0
    .cio         : {} > SARAM
    .pinit       : {} > SARAM

    .emif_cs0    : > EMIF_CS0
    .emif_cs2    : > EMIF_CS2
    .emif_cs3    : > EMIF_CS3
    .emif_cs4    : > EMIF_CS4
    .emif_cs5    : > EMIF_CS5
}
```

NOTE: You need to adjust the sizes of stack, syststack and the heap for your application. Please note that the preemptive **QK kernel** requires more stack than the cooperative Vanilla kernel.

6 The QS Software Tracing Instrumentation

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]).

This QDK demonstrates how to use the QS software tracing instrumentation to generate real-time trace of a running QP application. The QS port to the TMS320C55x DSP is more involved than most other ports due to the fact that TMS320 DSP cannot address individual 8-bit bytes, which leads to unexpected behavior of the exact-width integer type `uint8_t`.

NOTE: The exact-width integer type `uint8_t` is in fact 16-bit wide on the TMS320 DSPs. This unusual aspect breaks the whole slew of assumptions made in the original implementation of the QS target-resident component.

6.1 The qs_port.c Source File

As it turns out, the QS API does not need to change in any way, but the unexpected wider dynamic range of the `uint8_t` data type on the TMS320 DSP requires careful re-implementing of most of the QS target-based code. The modifications are necessary to efficiently manage the target-resident trace buffer as a *packed array of 8-bit bytes* and because the `uint8_t`-type variables must be additionally masked off with 0xFF to get the true value of the least-significant 8-bits for comparisons with transparent bytes (see Chapter 11 in [PSiCC2]). Listing 12 illustrates the most representative fragments of the `qs_port.c` source file, which is located in the `qpc\ports\tms320c55x\vanilla\c5500\src\` directory for the "vanilla" port and also in `qpc\ports\tms320c55x\qk\c5500\src\` directory for the QK port.

Listing 12: The most representative fragments of the qs_port.c source file

```
     #include "qs_pkg.h"
     #include "qassert.h"

     //Q_DEFINE_THIS_MODULE("qs_port")

     #undef  QS_INSERT_BYTE
     #define QS_INSERT_BYTE(b_) \
(1)      if ((QS_head_ & 1U) == 0U) { \
(2)          QS_ring_[QS_head_ >> 1] = (b_) << 8; \
         } \
         else { \
(3)          QS_ring_[QS_head_ >> 1] |= (b_) & 0xFFU; \
         } \
         if (++QS_head_ == QS_end_) { \
             QS_head_ = (QSCtr)0; \
         } \
         ++QS_used_;

     /*..........................................................................*/
     void QS_initBuf(uint8_t sto[], uint32_t stoSize) {
         QS_ring_   = &sto[0];
(4)     QS_end_    = (QSCtr)(stoSize * 2);             /* number of 8-bit bytes */
         QS_head_   = (uint16_t)0;
         . . .
         bzero(QS_glbFilter_, sizeof(QS_glbFilter_));
     }
     /*..........................................................................*/
     void QS_begin(uint8_t rec) {
         uint8_t b;
         QS_chksum_ = (uint8_t)0;                          /* clear the checksum */
         ++QS_seq_;                                   /* increment the sequence num */
(5)     b = QS_seq_ & 0xFFU;
         QS_INSERT_ESC_BYTE(b)                        /* store the sequence number */
         QS_INSERT_ESC_BYTE(rec)                            /* store the record ID */
     }
     /*..........................................................................*/
     void QS_u8(uint8_t format, uint8_t d) {
         QS_INSERT_ESC_BYTE(format)
(6)     d &= 0xFFU;
         QS_INSERT_ESC_BYTE(d)
     }
      . . .
     /* from qs_byte.c =========================================================*/
     uint16_t QS_getByte(void) {
         uint8_t byte;
         if (QS_used_ == (QSCtr)0) {
             return QS_EOD;                               /* return End-Of-Data */
         }
(7)     if ((QS_tail_ & 1U) == 0U) {
(8)         byte = QS_ring_[QS_tail_ >> 1] >> 8;
         }
         else {
(9)         byte = QS_ring_[QS_tail_ >> 1] & 0xFFU;
         }
         ++QS_tail_;                                        /* advance the tail */
         if (QS_tail_ == QS_end_) {                        /* tail wrap around? */
             QS_tail_ = (QSCtr)0;
         }
         --QS_used_;                                      /* one less byte used */
         return byte;                                        /* return the byte */
     }
      . . .
```

(1) As noted before, the TMS320 DSP cannot separately address 8-bit quantities, so the addressing of bytes is implemented in software. An even value of the `QS_head_` index addresses the most-significant byte in the word, and the odd value the least-significant byte.

(2) The argument `b_` is shifted to the most-significant byte position and stored in the `QS_ring_[]` buffer. Simultaneously, the least-significant byte in the same word is set to zero, but this is OK, because this byte will be overwritten anyway in the next write.

(3) The argument `b_` is masked and or-ed with the most-significant byte in the the `QS_ring_[]` buffer.

(4) On the TMS320 DSP `sizeof(uint16_t) == 1`. The trace buffer indexes are counting 8-bit bytes, so a buffer with a size computed by the sizeof() operator needs to be **multiplied by 2**.

(5,6) The byte quantities must be explicitly masked with 0xFF before comparing them to the transparent bytes in the `QS_INSERT_ESC_BYTE()` macro.

(7-9) The `QS_getByte()` function unpacks the byte from the word at `QS_ring_[]` buffer by reversing the steps (1-3).

NOTE: The block-oriented interface `QS_getBlock()` is **not** available in the TMS320 port of QS. The reason is that the DSP cannot individually address densely packed bytes in the trace buffer.

6.2 QS Initialization

On the eZdsp-C5515 USB Stick board QS is configured to send the trace data via the built-in UART. The QS platform-dependent implementation is located in the file `bsp.c` and looks as shown in Listing 13:

Listing 13: QS implementation to send data out of the UART of the TMS320C5515 DSP

```
(1) #ifdef Q_SPY

    uint8_t QS_onStartup(void const *arg) {
(2)     static uint8_t qsBuf[2*256];                   /* buffer for Quantum Spy */
        CSL_UartSetup uartSetup;

(3)     QS_initBuf(qsBuf, sizeof(qsBuf));

(4)     uartSetup.clkInput = CPU_CLOCK_HZ;             /* input clock freq in Hz */
        uartSetup.baud = UART_BAUD_RATE;                             /* baud rate */
        uartSetup.wordLength = CSL_UART_WORD8;                 /* word length of 8 */
        uartSetup.stopBits = 0;                        /* to generate 1 stop bit */
        uartSetup.parity = CSL_UART_DISABLE_PARITY;             /* disable parity */
        uartSetup.fifoControl = CSL_UART_FIFO_DMA1_DISABLE_TRIG01;/*enable FIFO */
        uartSetup.loopBackEnable = CSL_UART_NO_LOOPBACK;           /* no loopback */
        uartSetup.afeEnable = CSL_UART_NO_AFE;            /* no auto flow control */
        uartSetup.rtsEnable = CSL_UART_NO_RTS;                          /* no RTS */

(5)     UART_init(&l_uartObj, CSL_UART_INST_0, UART_POLLED);/* init. UART oject */
(6)     CSL_SYSCTRL_REGS->EBSR = 0x1800U;    /* re-configure I/O muxing for LEDs */
(7)     UART_setup(&l_uartObj, &uartSetup);          /* configure UART registers */

        /* initialize the CPU Timer1 used for QS timestamp */
(8)     CSL_TIM_1_REGS->TCR = 0U;                                  /* stop Timer1 */
        CSL_TIM_1_REGS->TIMPRD1 = ~0U;
        CSL_TIM_1_REGS->TIMPRD2 = ~0U;
        CSL_TIM_1_REGS->TIMCNT1 = ~0U;
        CSL_TIM_1_REGS->TIMCNT2 = ~0U;
        CSL_TIM_1_REGS->TCR     = 0x801BU;      /* autoReload | prescaler = 128 */

                                                     /* setup the QS filters... */
(9)     QS_FILTER_ON(QS_SIG_DIC);
        QS_FILTER_ON(QS_OBJ_DIC);
        QS_FILTER_ON(QS_FUN_DIC);
        . . .

        return (uint8_t)1;                                      /* return success */
```

```
     }
     /*..................................................................*/
(10) QSTimeCtr QS_onGetTime(void) {          /* invoked with interrupts disabled */
         uint32_t tmr32;

         tmr32  = (uint32_t)CSL_TIM_1_REGS->TIMCNT2 << 16;
         tmr32 |= (uint32_t)CSL_TIM_1_REGS->TIMCNT1;

         return (QSTimeCtr)(0xFFFFFFFFUL - tmr32);
     }
     /*..................................................................*/
     void QS_onFlush(void) {
         uint16_t b;
         while ((b = QS_getByte()) != QS_EOD) {       /* while not End-Of-Data... */
             while (!CSL_FEXT(l_uartObj.uartRegs->LSR, UART_LSR_THRE)) {
             }
             CSL_FSET(l_uartObj.uartRegs->THR, 7U, 0U, b);    /* output the byte */
         }
     }
     #endif                                                            /* Q_SPY */
```

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

(2) You should adjust the QS buffer size to your particular application

(3) The function `QS_initBuf()` takes the size of the QS buffer as returned by the `sizeof()` operator, which is the usual practice (note that on TMS320 DSP, the `sizeof()` operator returns the number of **16-bit words**).

(4) The UART setup structure defined in the CSL is initialized for the specific settings

(5) The CSL function initializes the UART and sets up the EBSR register to multiplex UART TX and RX pins.

(6) The EBSR mode is modified to allow correct GPIO configuration for the User LEDs

(7) The CSL function sets up the UART to the selected configuration.

(8) Timer1 is configured for providing the 32-bit time stamp. The timer runs off the 100MHz system clock. The pre-scaler of 128 sets the timer count granularity of 100/128 count per microsecond. You can change this granularity by selecting a different pre-scaler.

(9) The QS filters are configured. Here some of the filters are commented out to avoid overrunning the QS buffer due to high volume of trace data.

(10) The QS time stamp is generated from the free running 32-bit Timer1. Please note that Timer1 counts down, so it must be subtracted from 0xFFFFFFFF to make it to an up-counter.

(11) The block-oriented interface QS_block() is used to extract the trace data one byte at a time. (See also Listing 6 and Listing 9).

7 Related Documents and References

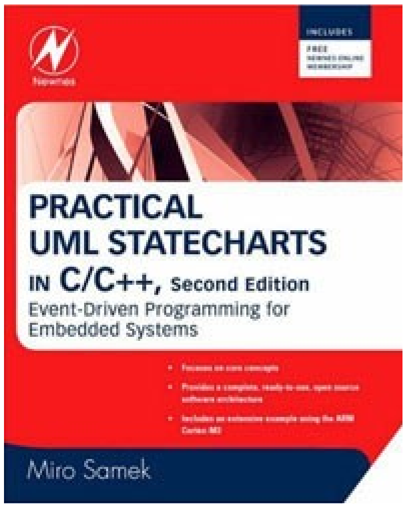
Document	Location
[PSiCC2] “Practical UML Statecharts in C/C++, Second Edition”, Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP 08] “QP Reference Manual”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/doxygen/qpn/
[QL AN-Directory 07] “Application Note: QP Directory Structure”, Quantum Leaps, LLC, 2007	http://www.state-machine.com/resources/AN_QP_Directory_Structure.pdf
[QL AN-DPP 08] “Application Note: Dining Philosopher Problem Application”, Quantum Leaps, LLC, 2008	http://www.state-machine.com/resources/AN_DPP.pdf
[TI SWPU073e 09] “C55x v3.x CPU Reference Guide”, Texas Instruments, 2009	Texas Instruments document SWPU073E: http://www.ti.com/lit/ug/swpu073e/swpu073e.pdf
[TI SPRU281F 03] “TMS320C55x Optimizing C/C++ Compiler User's Guide”, Texas Instruments, 2003	Texas Instruments literature number SPRU281F: http://www.ti.com/lit/ug/spru281f/spru281f.pdf
[SD USBSTK5515 10] “TMS320C5515 eZdsp USB Stick”, Spectrum Digital, 2010	Spectrum Digital: http://support.spectrumdigital.com/boards/usbstk5515/reva/files/usbstk5515_TechRef_RevA.pdf
[TI SPRU433J 04] “TMS320C55x Chip Support Library API Reference Guide”, Texas Instruments, 2004	Texas Instruments literature number SPRU433J http://www.ti.com/lit/ug/spru433j/spru433j.pdf .
[Samek 07a] “Using Low-Power Modes in Foreground/Background Systems”, Miro Samek, Embedded System Design, October 2007	http://www.embedded.com/design/202103425

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
Miro Samek
Newnes

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

Texas Instruments, Inc.
Web: http://www.ti.com