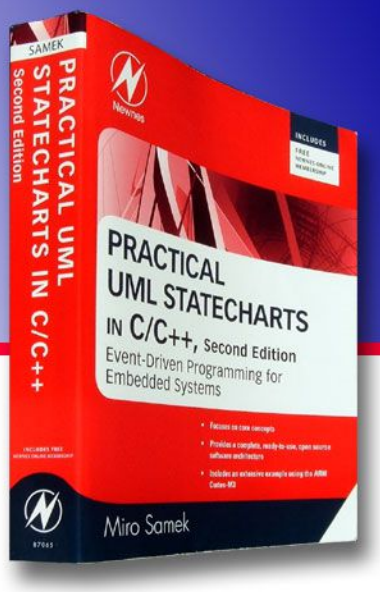
Quantum® LeaPs
innovating embedded systems



QDK™ ARM-Cortex LPCXpresso with GNU

Document Revision H
February 2013

ARM CONNECTED

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

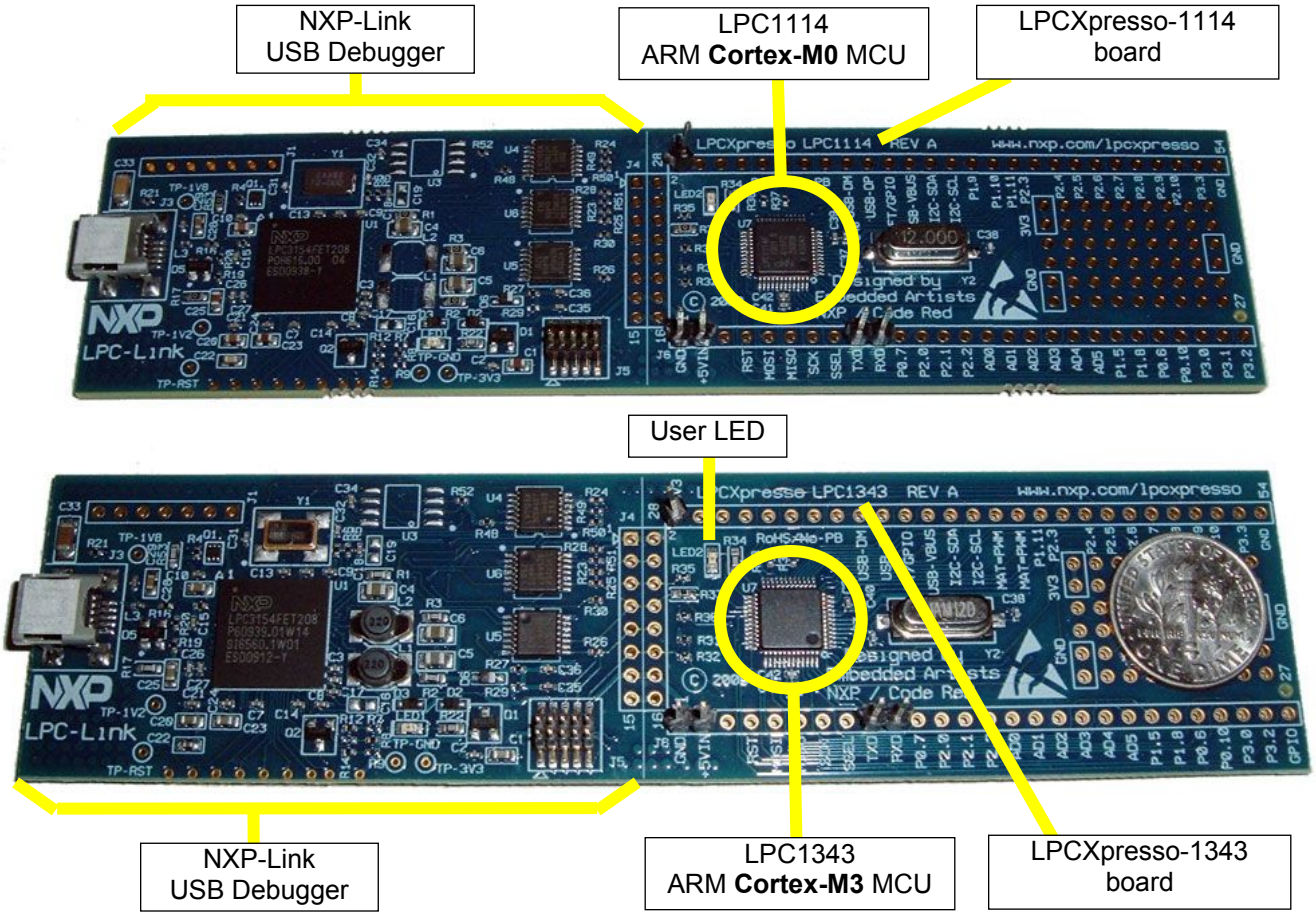
Table of Contents

1 Introduction

This **QP Development Kit**™ (QDK) describes how to use the QP/C™ and QP™/C++ state machine frameworks version **4.5.04** or higher on the ARM Cortex-M based LPC1114/1343/1769 microcontrollers with the GNU toolchain. This QDK uses the LPCXpresso™ development board from NXP shown in Figure 1.

NOTE: This QDK is based on the **Application Note "QP™ and ARM-Cortex with GNU"** [QP-Cortex] available as a separate document with the QDK. This Application Note describes the QP source code common for all ARM Cortex-M3 and Cortex-M0 cores.

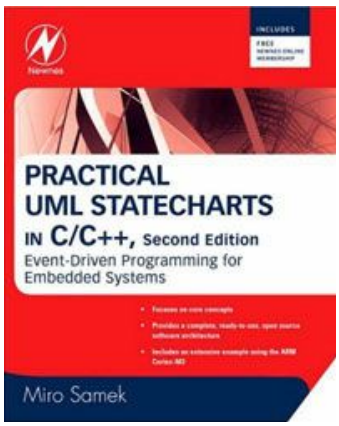
Figure 1 The LPCXpresso-1114 and LPCXpresso-1343 boards used to test the ARM-Cortex port.



NOTE: This QDK Manual pertains both to C and C++ versions of the QP™ state machine frameworks. Most of the code listings in this document refer to the QP/C version. Occasionally the C code is followed by the equivalent C++ implementation to show the C++ differences whenever such differences become important.

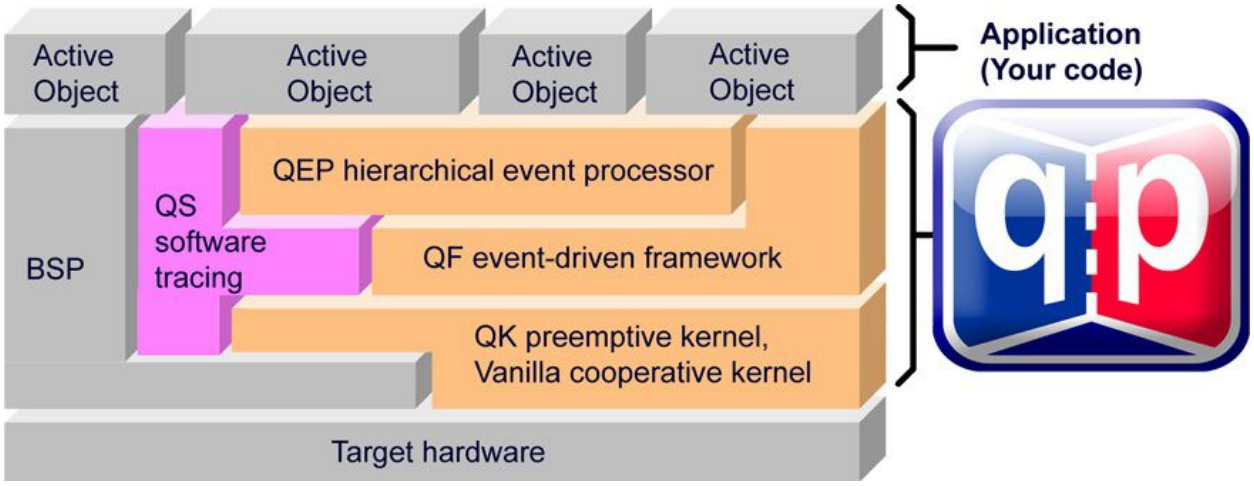
1.1 About QP™

QP™ is a family of very lightweight, open source, event-driven, active object frameworks for microcontrollers. QP enables building well-structured embedded applications as a set of concurrently executing hierarchical state machines (active objects) directly in C or C++. QP is described in great detail in the book "*Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*" [PSiCC2] (Newnes, 2008).



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

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



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

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

1.2 About QM™

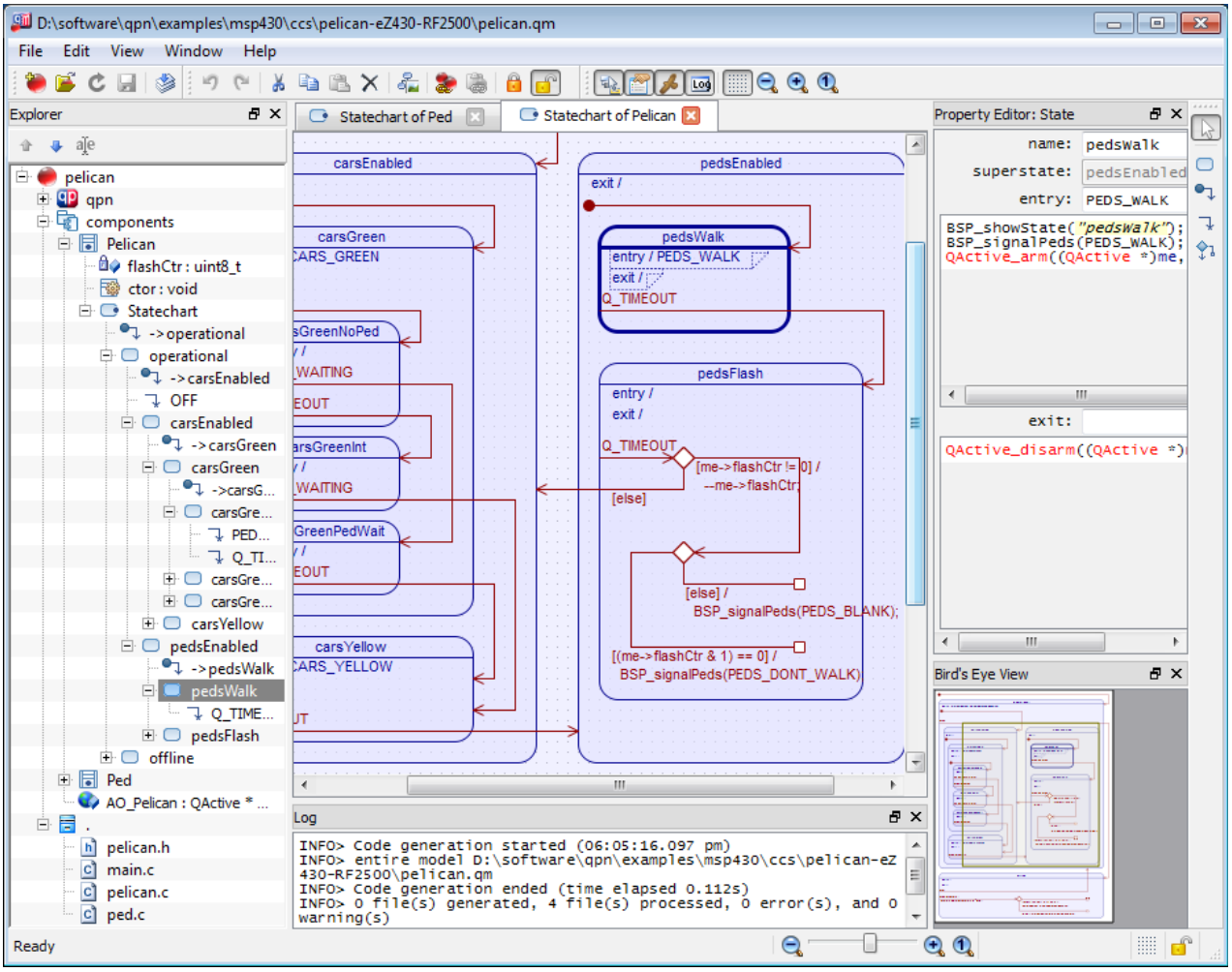
QM™ (QP™ Modeler) is a free, cross-platform, graphical UML modeling tool for designing and implementing real-time embedded applications based on the QP™ state machine frameworks. QM™ itself is based on the Qt framework and therefore runs naively on Windows, Linux, and Mac OS X.

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

The code accompanying this App Note contains three application examples: the Dining Philosopher Problem [AN-DPP], the PEdestrian LIght CONtrolled [AN-PELICAN] crossing, and the "Fly 'n' Shoot" game simulation for the EK-LM3S811 board (see Chapter 1 in [PSiCC2] all modeled with QM.

NOTE: The provided QM model files assume QM version **2.2.02** or higher.

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



1.3 About the ARM-Cortex Port

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

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

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

1.4 Cortex Microcontroller Software Interface Standard (CMSIS)

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

1.5 Licensing QP™

The **Generally Available (GA)** distributions of QP available for download from the www.state-machine.com/downloads website are offered under the same licensing options as the QP baseline code. These available licenses are:

- 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.

1.6 Licensing QM™

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

2 Getting Started

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

Listing 1 Selected directories and files of the QP after installing the QDK-ARM-Cortex-LPCXpresso. The highlighted directories and files are included in the QDKs for LPCXpresso.

```
<qp>/                       - QP-root directory for Quantum Platform (QP)
  |
  +-include/                - QP public include files
  | +-. . .
  |
  +-ports/                  - QP ports
  | +-arm-cortex/           - ARM-Cortex port
  | | +-vanilla/            - "vanilla" ports
  | | | +-gnu/              - GNU ARM compiler
  | | | | +-dbg/            - Debug build
  | | | | | +-libqp_cortex-m0_cr.a - QP library for Cortex-M0 with CodeRed tools
  | | | | | +-libqp_cortex-m3_cr.a - QP library for Cortex-M3 with CodeRed tools
  | | | | +-rel/            - Release build
  | | | | | +-. . .
  | | | | +-spy/            - Spy build
  | | | | | +-libqp_cortex-m0_cr.a - QP library for Cortex-M0 with CodeRed tools
  | | | | | +-libqp_cortex-m3_cr.a - QP library for Cortex-M3 with CodeRed tools
  | | | | +-make_cortex-m0_cr.bat - Batch to build QP for Cortex-M0 with CodeRed
  | | | | +-make_cortex-m3_cr.bat - Batch to build QP for Cortex-M3 with CodeRed
  | | | | +-qep_port.h      - QEP platform-dependent public include
  | | | | +-qf_port.h       - QF platform-dependent public include
  | | | | +-qs_port.h       - QS platform-dependent public include
  | | | | +-qp_port.h       - QP platform-dependent public include
  | | +-qk/                 - QK (Quantum Kernel) ports
  | | | +-gnu/              - GNU ARM compiler
  | | | | +-dbg/            - Debug build
  | | | | | +-libqp_cortex-m0_cr.a -  QP library for Cortex-M0 with CodeRed tools
  | | | | | +-libqp_cortex-m3_cr.a - QP  library for Cortex-M3 with CodeRed tools
  | | | | +-rel/            - Release build
  | | | | +-make_cortex-m0_cr.bat - Batch to build QP for Cortex-M0 with CodeRed
  | | | | +-make_cortex-m3_cr.bat - Batch to build QP for Cortex-M3 with CodeRed
  | | | | +-qep_port.h      - QEP platform-dependent public include
  | | | | +-qf_port.h       - QF platform-dependent public include
  | | | | +-qk_port.h       - QK platform-dependent public include
  | | | | +-qk_port.s       - Platfom-specific source code for the QK port
  | | | | +-qs_port.h       - QS platform-dependent public include
  | | | | +-qp_port.h       - QP platform-dependent public include
  |
  +-examples/               - subdirectory containing the QP example files
  | +-arm-cortex/           - ARM-Cortex port
  | | +-vanilla/            - "vanilla" examples (non-preemptive scheduler of QF)
  | | | +-gnu/              - GNU ARM compiler
  | | | | +-dpp-lpcxpresso-1114/ - Dining Philosophers example for LPCXpresso-1114
  | | | | | +-cmsis/        - directory containing the CMSIS files
```

```
| | | | | +-lpc11xx_lib/ - directory containing the LPC11xx library (from NXP)
| | | | | +-dbg/         - directory containing the Debug build
| | | | | +-rel/         - directory containing the Release build
| | | | | +-spy/         - directory containing the Spy build
| | | | | |
| | | | | +-.cproject    - Eclipse project file for the LPCXpresso IDE
| | | | | +-.project     - Eclipse project file for the LPCXpresso IDE
| | | | | +-Makefile     - external Makefile for the LPCXpresso IDE
| | | | | +-lpc1114.ld   - linker command file for LPC1114
| | | | | +-bsp.c        - Board Support Package for the DPP application
| | | | | +-bsp.h        - BSP header file
| | | | | +-dpp.qm       - QM model of the DPP application
| | | | | +-dpp.h        - the DPP header file
| | | | | +-main.c       - the main function
| | | | | +-philo.c      - the Philosopher active object
| | | | | +-table.c      - the Table active object
| | | | | +-no_heap.c    - dummy heap routines to reduce code size
| | | | |
| | | | +-dpp-lpcxpresso-1343/ - Dining Philosophers example for LPCXpresso-1343
| | | | | +-cmsis/       - directory containing the CMSIS files
| | | | | +-lpc13xx_lib/ - directory containing the LPC13xx library (from NXP)
| | | | | +-dbg/         - directory containing the Debug build
| | | | | +-rel/         - directory containing the Release build
| | | | | +-spy/         - directory containing the Spy build
| | | | | |
| | | | | +-.cproject    - Eclipse project file for the LPCXpresso IDE
| | | | | +-.project     - Eclipse project file for the LPCXpresso IDE
| | | | | +-Makefile     - external Makefile for the LPCXpresso IDE
| | | | | +-lpc1343.ld   - linker command file for LPC1114
| | | | | +-bsp.c        - Board Support Package for the DPP application
| | | | | +-bsp.h        - BSP header file
| | | | | +-dpp.qm       - QM model of the DPP application
| | | | | +-dpp.h        - the DPP header file
| | | | | +-main.c       - the main function
| | | | | +-philo.c      - the Philosopher active object
| | | | | +-table.c      - the Table active object
| | | | | +-no_heap.c    - dummy heap routines to reduce code size
| | | |
| | +-qk/                - QK examples
| | | +-gnu/             - GNU ARM compiler
| | | | +-dpp-qk-lpcxpresso-1114/ - DPP example for LPCXpresso-1114
| | | | | |=. . .
| | | | +-dpp-qk-lpcxpresso-1343/ - DPP example for LPCXpresso-1343
| | | | | |=. . .
```

2.1 Building the QP Libraries

All QP components are deployed as libraries that you statically link to your application. The pre-built libraries for QEP, QF, QS, and QK are provided inside the `<qp>\ports\arm-cortex` directory (see Listing 1). 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 LPCXpresso 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>.bat` for building all the libraries located in the `<qp>\ports\arm-cortex\...` directory. For example, to build the debug version of all the QP libraries for ARM-Cortex, with the GNU ARM compiler, QK kernel, you open a console window on a Windows PC, change directory to `<qp>\ports\arm-cortex\qk\gnu\`, and invoke the batch by typing at the command prompt the following command:

```
make_cortex-m0
```

The build process should produce the QP libraries in the location: `<qp>\ports\arm-cortex\qk\gnu\-dbg\`. The `make.bat` files assume that the Code Red GNU toolset has been installed in the directory `C:\tools\CodeRed\lpcxpresso\Tools\bin`.

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

In order to take advantage of the QS ("spy") instrumentation, you need to build the QS version of the QP libraries. You achieve this by invoking the `make_cortex-m3_cr.bat` utility with the "spy" target, like this:

```
make_cortex-m3_cr spy
```

The make process should produce the QP libraries in the directory: `<qp>\ports\arm-cortex\-vanilla\gnu\spy\`. You choose the build configuration by providing a target to the `make_cortex-m3.bat` utility. The default target is "`dbg`". Other targets are "`rel`", and "`spy`" respectively. The following table summarizes the targets accepted by `make_cortex-m3.bat`.

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

Software Version	Build commands
Debug (default)	`make_cortex-m0_cr` `make_cortex-m3_cr`
Release	`make_cortex-m0_cr rel` `make_cortex-m3_cr rel`
Spy	`make_cortex-m0_cr spy` `make_cortex-m3_cr spy`

2.2 Building and Debugging the Examples

The example applications for ARM-Cortex have been tested with the LPCXpresso evaluation boards from NXP (see Figure 1) and the GNU/Eclipse-based LPCXpresso toolset from Code Red. The examples contain the Makefile-based Eclipse projects for the LPCXpresso IDE as well as the Makefiles, so that you can conveniently build and debug the examples both from the LPCXpresso IDE and from the command prompt. The provided Makefiles and projects support building the Debug, Release, and Spy configurations.

NOTE: The provided Make files for building the QP applications assume that the GNU ARM toolchain has been installed in the directory `C:/tools/CodeRed/lpcxpresso/Tools/bin`. You need to adjust the symbol **`GNU_ARM`** at the top of the Makefile to the location of the LPCXpresso installation directory on your system. Alternatively, you can define the **`GNU_ARM`** symbol as an environment variable, in which case you don't need to modify the Makefile.

NOTE: The provided Make files also assume that you have defined the environment variable `QPC`, if you are using the QP/C framework or the environment variable `QPCPP`, if you are using the `QP/C++` framework. These environment variables must contain the paths to the installation directories of the QP/C and QP/C++ frameworks, respectively.

Defining the QP framework locations in environment variables allows you to locate your application in any directory or file system, regardless of the relative path to the QP frameworks.

2.2.1 Building the Examples from Command Line

The example directory `<qp>\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114\` contains the `Makefile` you can use to build the application. The `Makefile` supports three build configurations: Debug (default), Release, and Spy. You choose the build configuration by defining the `CONF` symbol at the command line, as shown in the table below. Figure 4 shows and example command-line build of the Spy configuration.

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

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

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

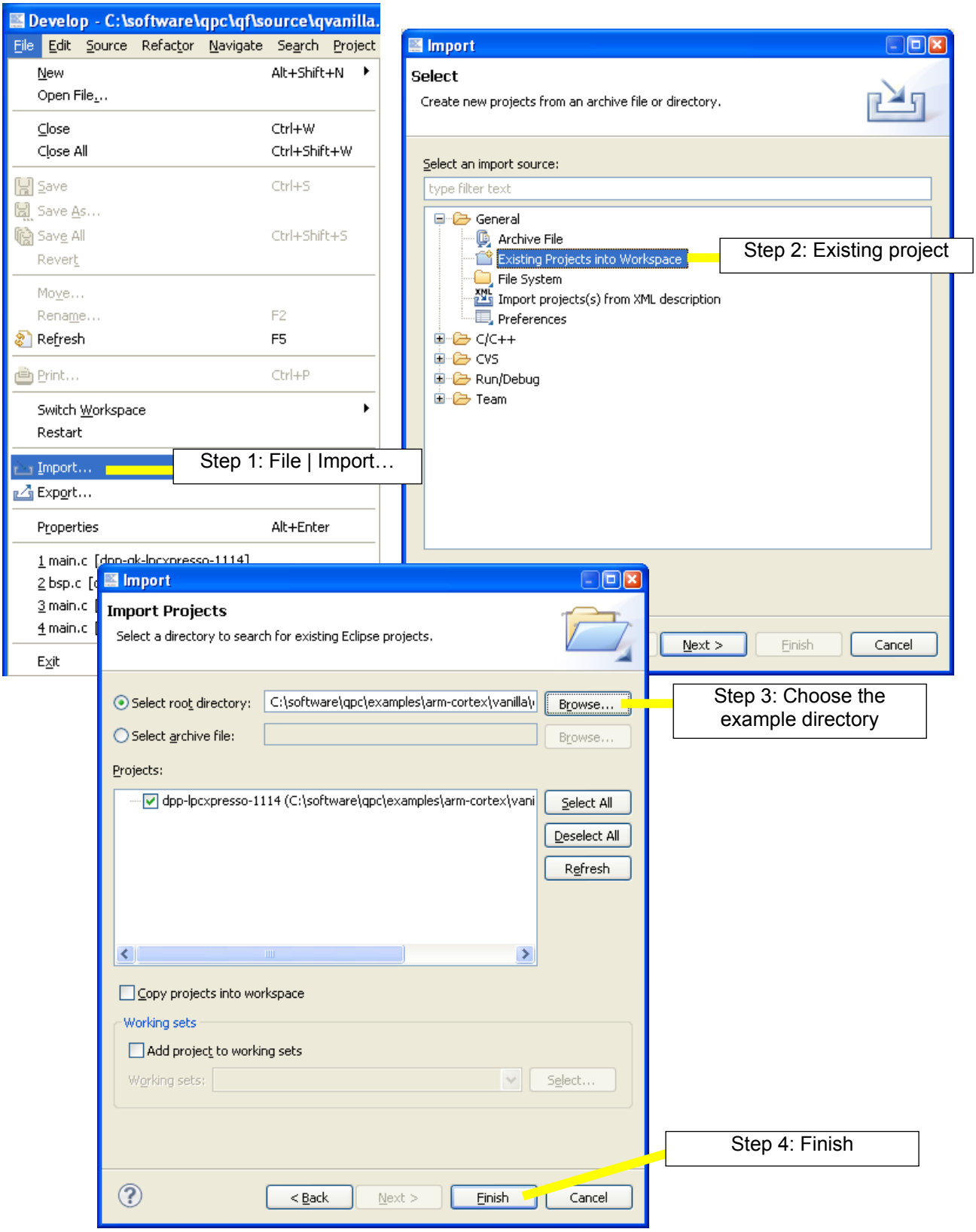
```
Administrator: Command Prompt
D:\software\qpc\examples\arm-cortex\qk\gnu\dpp-qk-lpcxpresso-1343>make CONF=spy
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -MM -MT spy/uart.o -mcpu
=cortex-m3 -mthumb -Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/p
orts/arm-cortex/qk/gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=
CMSISv1p30_LPC13xx -DQ_SPY lpc13xx_lib/src/uart.c > spy/uart.d
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -MM -MT spy/timer32.o -m
cpu=cortex-m3 -mthumb -Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc
"/ports/arm-cortex/qk/gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMS
IS=CMSISv1p30_LPC13xx -DQ_SPY lpc13xx_lib/src/timer32.c > spy/timer32.d
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -MM -MT spy/timer16.o -m
cpu=cortex-m3 -mthumb -Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc
"/ports/arm-cortex/qk/gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMS
IS=CMSISv1p30_LPC13xx -DQ_SPY lpc13xx_lib/src/timer16.c > spy/timer16.d
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -MM -MT spy/gpio.o -mcpu
=cortex-m3 -mthumb -Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/p
orts/arm-cortex/qk/gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=
CMSISv1p30_LPC13xx -DQ_SPY lpc13xx_lib/src/gpio.c > spy/gpio.d
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -MM -MT spy/clkconfig.o
-mcpu=cortex-m3 -mthumb -Wall -g -O -I"D:\software\qpc"/include -I"D:\software\q
pc"/ports/arm-cortex/qk/gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_C

                    .     .     .     .

C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb
-Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/ports/arm-cortex/qk/
gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC13xx
-DQ_SPY -c lpc13xx_lib/src/clkconfig.c -o spy/clkconfig.o
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb
-Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/ports/arm-cortex/qk/
gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC13xx
-DQ_SPY -c lpc13xx_lib/src/gpio.c -o spy/gpio.o
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb
-Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/ports/arm-cortex/qk/
gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC13xx
-DQ_SPY -c lpc13xx_lib/src/timer16.c -o spy/timer16.o
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb
-Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/ports/arm-cortex/qk/
gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC13xx
-DQ_SPY -c lpc13xx_lib/src/timer32.c -o spy/timer32.o
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb
-Wall -g -O -I"D:\software\qpc"/include -I"D:\software\qpc"/ports/arm-cortex/qk/
gnu -I. -Icmsis -Ilpc13xx_lib/inc -D__REDLIB__ -D__USE_CMSIS=CMSISv1p30_LPC13xx
-DQ_SPY -c lpc13xx_lib/src/uart.c -o spy/uart.o
C:/tools/CodeRed/lpcxpresso/Tools/bin/arm-none-eabi-g++ -Tlpc1343.ld -mcpu=corte
x-m3 -mthumb -nostdlib -Xlinker -Map=spy/dpp-qk.map --gc-sections -L"D:\software
\qpc"/ports/arm-cortex/qk/gnu/spy -o spy/dpp-qk.elf spy/startup_LPC13.o spy/main
.o spy/table.o spy/philo.o spy/bsp.o spy/no_heap.o spy/core_cm3.o spy/system_LPC
13xx.o spy/clkconfig.o spy/gpio.o spy/timer16.o spy/timer32.o spy/uart.o -lqk_co
rtex-m3_cr -lqf_cortex-m3_cr -lqep_cortex-m3_cr -lqs_cortex-m3_cr

D:\software\qpc\examples\arm-cortex\qk\gnu\dpp-qk-lpcxpresso-1343>_
```

2.2.2 *Building the Examples from Eclipse*

The example code contains the Eclipse projects for building and debugging the DPP examples with the LPCXpresso IDE from Code Red. The provided Eclipse projects are Makefile-type projects, which use the same `Makefiles` that you can call from the command line. In fact the `Makefiles` are specifically designed to allow building all supported configurations from Eclipse. Figure 5 shows how to import the provided projects into LPCXPresso IDE.

NOTE: The provided Makefiles allow you to create and configure the build configurations from the Project | Build Configurations | Manage… sub-menu. For the Release and Spy configurations, you should set the make command to `make CONF=rel` and `make CONF=spy`, respectively. The provided `Makefile` also correctly supports the clean targets, so invoking Project | Clean… menu for any build configuration works as expected.

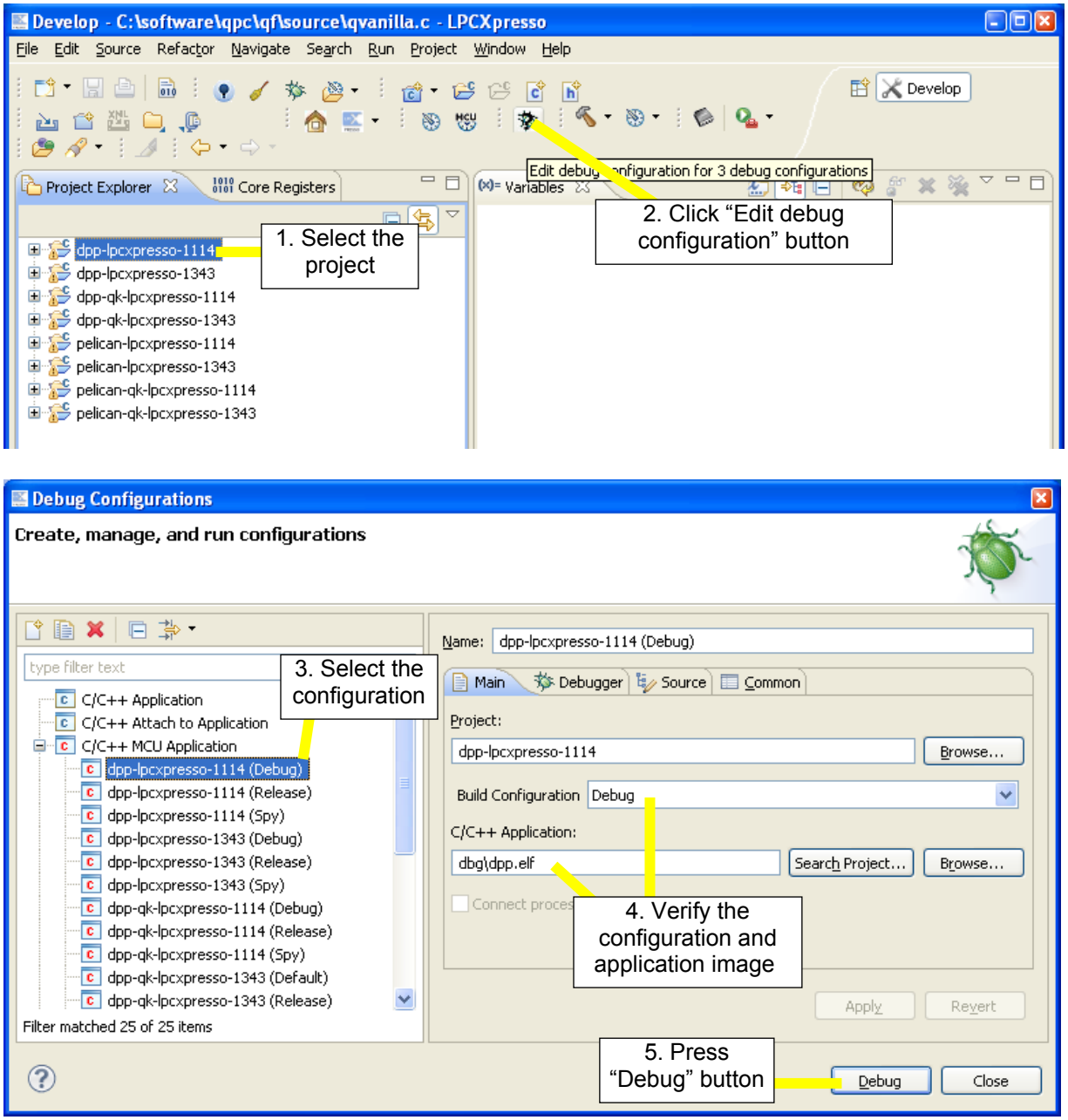
Figure 5 Steps of importing the existing project into the Eclipse (LPCXpresso) IDE



2.3 Downloading to Flash and Debugging the Examples

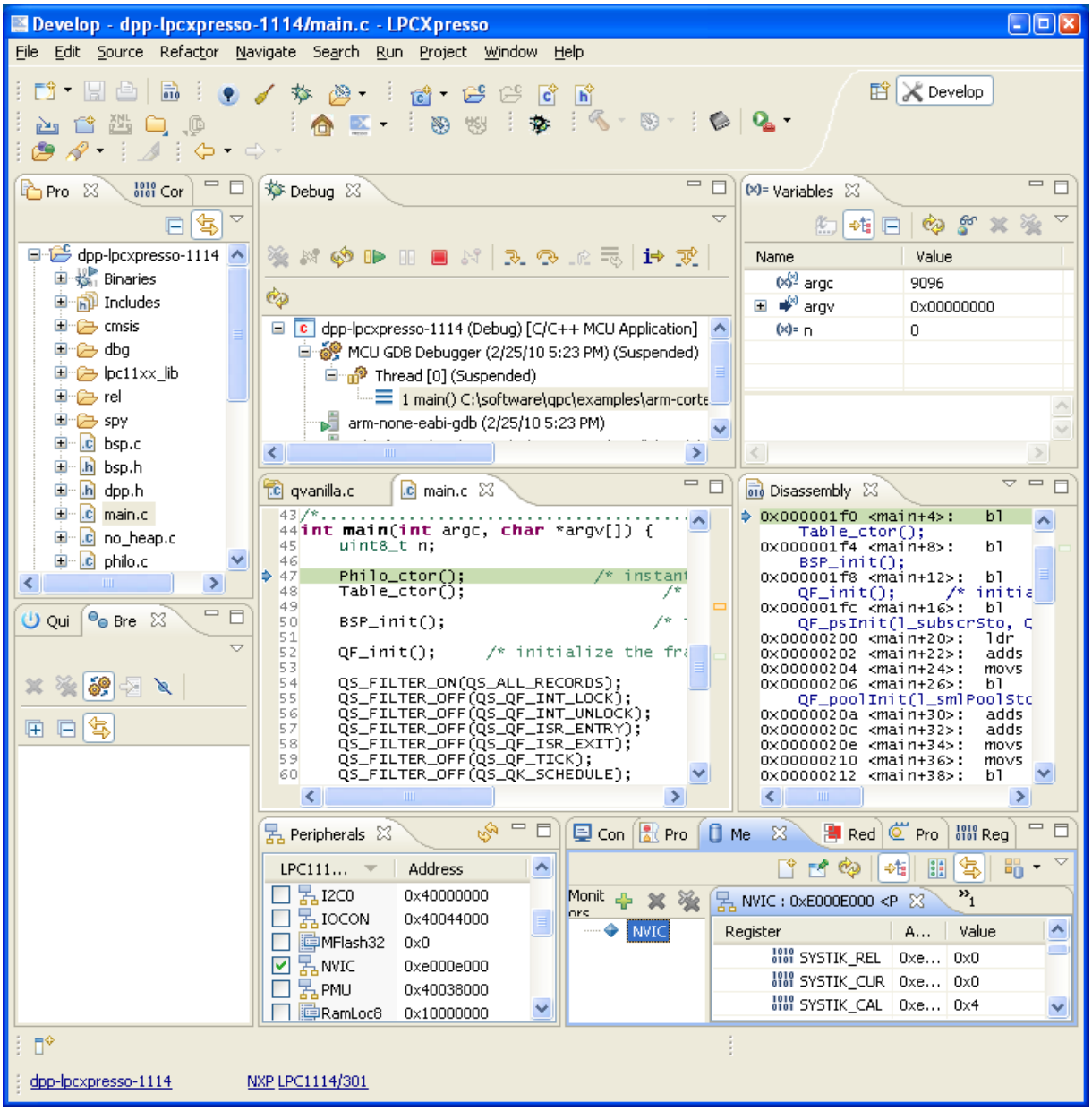
The example directory `<qp>\examples\arm-cortex\vanilla\gnu\dpp-lpcxpresso-1114\` contains the "launch" files for the LPCXpresso IDE that contain all information required for flash-downloading and debugging all the build configurations of each project. Unlike other Eclipse-based IDEs, LPCXpresso takes care automatically for launching the GDB server application for the LPC-Link hardware debugger. Figure 6 shows the steps required to start debugging one of the provided projects with LPCXpresso IDE.

Figure 6 Debugging the provided projects with LPCXpresso IDE



The following screen shot in Figure 7 shows a debugging session in Eclipse with various views.

Figure 7 Debugging with LPCXpresso IDE



2.3.1 *Software Tracing with Q-SPY*

For the QS (Q-SPY) software tracing output, you need to connect a TTL to RS-232 transceiver to the LPCXpresso board, as shown in Figure 8. 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.

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. Figure 8 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).
The LPCXpresso board brings out all the MCU pins to the edges of the board. To connect the RS-232 transceiver you need to fit the pins into the following 3 positions: GND, 3V3, and TXD. (Figure 8 shows additionally RXD connection).

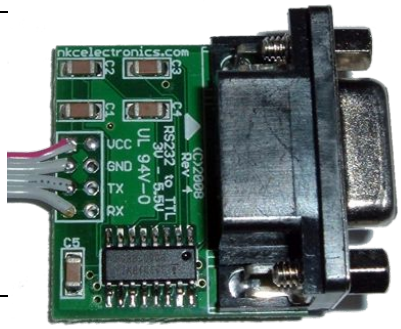
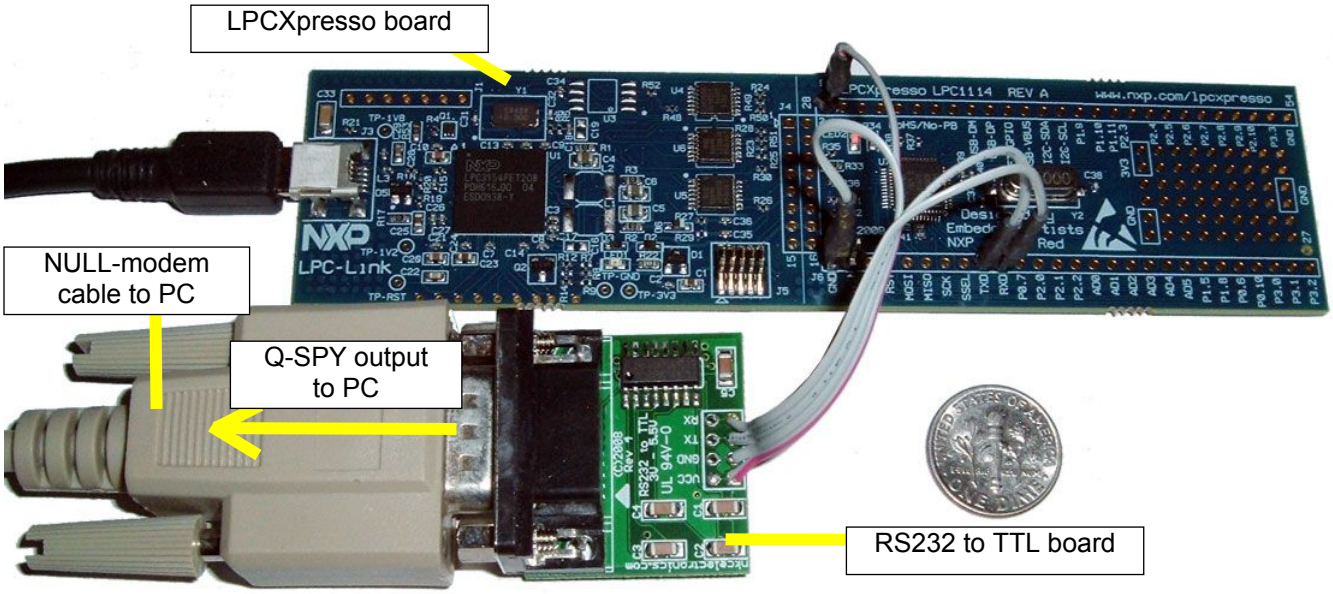


Figure 8 Connecting RS232-TTL board to the LPCXpresso board.



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 `<qp>\tools\qspy\win32\mingw\rel` and execute:

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

Figure 9 Screen shot from the QSPY output

```
qs.txt @ C:\tmp\
File  Edit  Search  AutoText  View  Help
QSPY host application 4.1.00
Copyright (c) Quantum Leaps, LLC.
Sun Oct 25 20:12:29 2009

-T 4
-O 4
-F 4
-S 1
-E 2
-Q 1
-P 2
-B 2
-C 2

           Obj Dic: 00200A80->l_smlPoolSto
           . . .
           Sig Dic: 00000005,Obj=00000000 ->DONE_SIG
           Sig Dic: 00000004,Obj=00000000 ->EAT_SIG
           Sig Dic: 00000006,Obj=00000000 ->TERMINATE_SIG
           Sig Dic: 00000008,Obj=00201460 ->HUNGRY_SIG
           Q_INIT : Obj=l_table Source=QHsm_top Target=Table_serving
0000139702 ==>Init: Obj=l_table New=Table_serving
           TICK   : Ctr=         1
           TICK   : Ctr=         2
           TICK   : Ctr=         3
           TICK   : Ctr=         4
           TICK   : Ctr=         5
           TICK   : Ctr=         6
           TICK   : Ctr=         7
0000224711 Disp==>: Obj=l_philo[4] Sig=TIMEOUT_SIG Active=Philo_thinking
           Q_ENTRY: Obj=l_philo[4] State=Philo_hungry
0000224829 ==>Tran: Obj=l_philo[4] Sig=TIMEOUT_SIG Source=Philo_thinking
New=Philo_hungry
0000224903 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_serving
0000224960 User000: 4 hungry
0000225076 User000: 4 eating
0000225130 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000225197 Disp==>: Obj=l_philo[4] Sig=EAT_SIG Active=Philo_hungry
           Q_ENTRY: Obj=l_philo[4] State=Philo_eating
0000225307 ==>Tran: Obj=l_philo[4] Sig=EAT_SIG Source=Philo_hungry
New=Philo_eating
0000225381 Disp==>: Obj=l_philo[3] Sig=TIMEOUT_SIG Active=Philo_thinking
           Q_ENTRY: Obj=l_philo[3] State=Philo_hungry
0000225499 ==>Tran: Obj=l_philo[3] Sig=TIMEOUT_SIG Source=Philo_thinking
New=Philo_hungry
0000225573 Disp==>: Obj=l_table Sig=HUNGRY_SIG Active=Table_serving
0000225630 User000: 3 hungry
0000225686 Intern : Obj=l_table Sig=HUNGRY_SIG Source=Table_serving
0000225753 Disp==>: Obj=l_philo[3] Sig=EAT_SIG Active=Philo_hungry
Ready                                                  Ln 16, Col 17
```

3 Interrupt Vector Table and Startup Code

ARM-Cortex requires you to place the initial Main Stack pointer and the addresses of all exception handlers and ISRs into the Interrupt Vector Table (IVT) placed in ROM. By the Cortex Microcontroller Software Interface Standard (CMSIS), the IVT and the startup code is located in the CMSIS directory in the `startup_LPC11.c` file. The following listing shows the beginning of the IVT for the DPP example, which uses two interrupts: SysTick and EXTI:

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

The IVT for the STM32F10x family of the connectivity line devices, is located in the file `<qp>\examples\arm-cortex\vanilla\gnu\dpp-qk-lpcxpresso-1114\cmsis\startup_LPC11.c`. This IVT can be easily adapted to other ARM-Cortex microcontrollers by modifying the IRQ handlers according to the datasheet of the specific ARM Cortex device.

Listing 2 Startup code and IVT for ARM-Cortex (cmsis\`startup_LPC11.c`)

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

                                                  /* external interrupts... */
    void __attribute__ ((weak)) I2C_IRQHandler(void);
    void __attribute__ ((weak)) TIMER16_0_IRQHandler(void);
    void __attribute__ ((weak)) TIMER16_1_IRQHandler(void);
    . . .
(2) void __attribute__ ((weak)) Spurious_Handler(void);

    /*----------------------------------------------------------------------------
    * weak aliases for each Exception handler to the Spurious_Handler.
    * Any function with the same name will override these definitions.
    */
(3) #pragma weak NMI_Handler                = Spurious_Handler
    #pragma weak MemManage_Handler          = Spurious_Handler
    #pragma weak BusFault_Handler           = Spurious_Handler
    #pragma weak UsageFault_Handler         = Spurious_Handler
    #pragma weak SVC_Handler                = Spurious_Handler
    . . .
    /* exception and interrupt vector table -------------------------------------*/
(4) typedef void (*ExceptionHandler)(void);
(5) typedef union {
        ExceptionHandler handler;
        void            *pointer;
    } VectorTableEntry;
```

```
 (6) extern unsigned __c_stack_top__;

     /*..........................................................................*/
 (7) __attribute__ ((section(".isr_vector")))
 (8) VectorTableEntry const g_pfnVectors[] = {
 (9)     { .pointer = &__c_stack_top__           }, /* initial stack pointer     */
(10)     { .handler = &Reset_Handler             }, /* Reset Handler             */
(11)     { .handler = &NMI_Handler               }, /* NMI Handler               */
         { .handler = &HardFault_Handler         }, /* Hard Fault Handler        */
         { .handler = &MemManage_Handler         }, /* MPU Fault Handler         */
         { .handler = &BusFault_Handler          }, /* Bus Fault Handler         */
         { .handler = &UsageFault_Handler        }, /* Usage Fault Handler       */
         { .handler = &Spurious_Handler          }, /* Reserved                  */
         { .handler = &Spurious_Handler          }, /* Reserved                  */
         { .handler = &Spurious_Handler          }, /* Reserved                  */
         { .handler = &Spurious_Handler          }, /* Reserved                  */
         { .handler = &SVC_Handler               }, /* SVCall Handler            */
         { .handler = &DebugMon_Handler          }, /* Debug Monitor Handler     */
         { .handler = &Spurious_Handler          }, /* Reserved                  */
         { .handler = &PendSV_Handler            }, /* PendSV Handler            */
         { .handler = &SysTick_Handler           }, /* SysTick Handler           */
         /* external interrupts (IRQs) ... */
         { .handler = WAKEUP_IRQHandler          }, /* PIO0_0  Wakeup            */
         . . .
     };
     . . .
     /*..........................................................................*/
(12) void Reset_Handler(void) __attribute__((__interrupt__));
     void Reset_Handler(void) {
         extern int main(void);
         extern int __libc_init_array(void);
(13)     extern unsigned __data_start;    /* start of .data in the linker script */
(14)     extern unsigned __data_end__;      /* end of .data in the linker script */
(15)     extern unsigned const __data_load;  /* initialization values for .data  */
(16)     extern unsigned __bss_start__;    /* start of .bss in the linker script */
(17)     extern unsigned __bss_end__;        /* end of .bss in the linker script */
         unsigned const *src;
         unsigned *dst;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(12) The `Reset_Hanlder()` exception handler performs the low-level initialization required by the C/C++ standard and eventually calls `main()`. Even though ARM Cortex is designed to use regular C functions as exception and interrupt handlers, functions that are used directly as interrupt handlers should be annotated with `__attribute__((__interrupt__))`. This tells the GNU compiler to add special stack alignment code to the function prologue.

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

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

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

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

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

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

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

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

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

4 Linker Script

The linker script must match the startup code for all the section names and other linker symbols. The linker script cannot be quite generic, because it must define the specific memory map of the target device. The linker script for the LPC1114 devices is located in the application directory in the file `lpc1114.ld`, which corresponds to the DPP example application.

Listing 3 Linker script for LPC1114 devices

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

(4) MEMORY {                                        /* memory map of LPC1114/301  */
```

```
 (5)     ROM (rx)  : ORIGIN = 0x00000000, LENGTH = 32K
 (6)     RAM (xrw) : ORIGIN = 0x10000000, LENGTH = 8K
     }

     /* The size of the stack used by the application. NOTE: you need to adjust  */
 (7) STACK_SIZE = 600;

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

     SECTIONS {

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

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

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

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

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

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

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

(22)     .data : {
             __data_load = LOADADDR (.data);
             __data_start = .;
             *(.data)                                            /* .data sections */
             *(.data*)                                          /* .data* sections */
```

```
            . = ALIGN(4);
            __data_end__ = .;
            _edata = __data_end__;
(23)      } >RAM AT>ROM

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

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

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

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

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

Listing 3 shows the linker script for the LPC1114 MCU. The script is identical for C and C++ versions. The highlights of the linker script are as follows:

(1) The OUTPUT_FORMAT directive specifies the format of the output image (elf32, little-endian, ARM)

(2) OUTPUT_ARCH specifies the target machine architecture.

(3) ENTRY explicitly specifies the first instruction to execute in a program

(4) The MEMORY command describes the location and size of blocks of memory in the target.

(5) The region ROM corresponds to the on-chip flash of the LPC1114 device. It can contain read-only and executable sections (rx), it starts at 0x00000000 and is 32KB in size.

(6) The region RAM corresponds to the on-chip SRAM of the LPC1114 device. It can contain read-only, read-write and executable sections (rwx), it starts at 0x10000000 and is 8KB in size.

(7) The `STACK_SIZE` symbol determines the sizes of the ARM Main stack. You need to adjust the size for your particular application. The stack size cannot be zero.

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

(8) The `HEAP_SIZE` symbol determines the sizes of the heap. You need to adjust the sizes for your particular application. The heap size can be zero.

(9) The `.isr_vector` section contains the ARM Cortex IVT and must be located as the first section in ROM.

(10) This line locates all `.isr_vector` section.

(11) The section size is aligned to the 4-byte boundary

(12) This section is loaded directly to the ROM region defined in the MEMORY command.

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

(14) The `.text` section groups all individual `.text` and `.text*` sections from all modules.

(15) The section `.rodata` is used for read-only (constant) data, such as look-up tables.

(16-17) The `.init` and `.fini` sections are synthesized by the GNU C++ compiler and are used for static constructors and destructors. These sections are empty in C programs.

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

(19,20) The `.preinint_array` and `.inint_array` sections hold arrays of function pointers that are called by the startup code to initialize the program. In C++ programs these hold pointers to the static constructors that are called by `__libc_init_array()` before `main()`.

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

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

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

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

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

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

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

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

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

4.1 Linker Options

The linker options for C and C++ are the same and are defined in the `Makefile` located in the DPP directory. The most

Linker options for C and C++ builds.

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

(1) `-T` option specifies the name of the linker script (`dpp.ld` in this case).

(2) `-o` option specifies the name of image file (`dpp.elf` in this case).

(3) `--gc-sections` enable garbage collection of unused input sections..

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

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

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

5.1 Compiler Options for C

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

```
       ARM_CORE = cortex-m0

CCFLAGS = -g -c \
(1a)   -mcpu=$(ARM_CORE) \
(2a)   -mthumb \
(3a)   -O \
(4a)   -DARM_ARCH_V6M
       -Wall

CCFLAGS = -c \
(1b)   -mcpu=$(ARM_CORE) \
(2a)   -mthumb \
(3b)   -Os \
(4b)   -DARM_ARCH_V6M
(5b)   -DNDEBUG \
       -Wall
```

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

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

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

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

(4) `-DARM_ARCH-V6M` selects the ARMv6M profile, which does not support certain features, such as the CLZ (Count Leading Zeros) instruction

NOTE: The option `-DARM_ARCH-V6M` **must** be defined for Cortex-M0 / M0+ / M1, because otherwise the macro `QF_LOG2()` will attempt to use the CLZ instruction, which is not supported on ARMv6M cores, so the code won't compile.

At the same time, the option `-DARM_ARCH-V6M` should not be defined for ARMv7M cores, such as Cortex-M3 / M4 / M4F, because these cores support the CLZ instruction and take advantage of this.

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

5.2 Compiler Options for C++

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

Listing 5 Compiler options used for C++ project.

```
CPPFLAGS = -g -c -mcpu=$(ARM_CPU) -DARM_ARCH_V6M -mthumb \
(1)    -fno-rtti \
(2)    -fno-exceptions \
       -Wall
```

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

(1) `-fno-rtti` disables generation of information about every class with virtual functions for use by the C++ runtime type identification features (`dynamic_cast` and `typeid`). Disabling RTTI eliminates several KB of support code from the C++ runtime library (assuming that you don't link with code that uses RTTI). Note that the `dynamic_cast` operator can still be used for casts that do not require runtime type information, i.e. casts to `void *` or to unambiguous base classes.

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

5.3 Reducing the Overhead of C++

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

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

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

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

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

     }                                                          // extern "C"
```

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

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

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

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

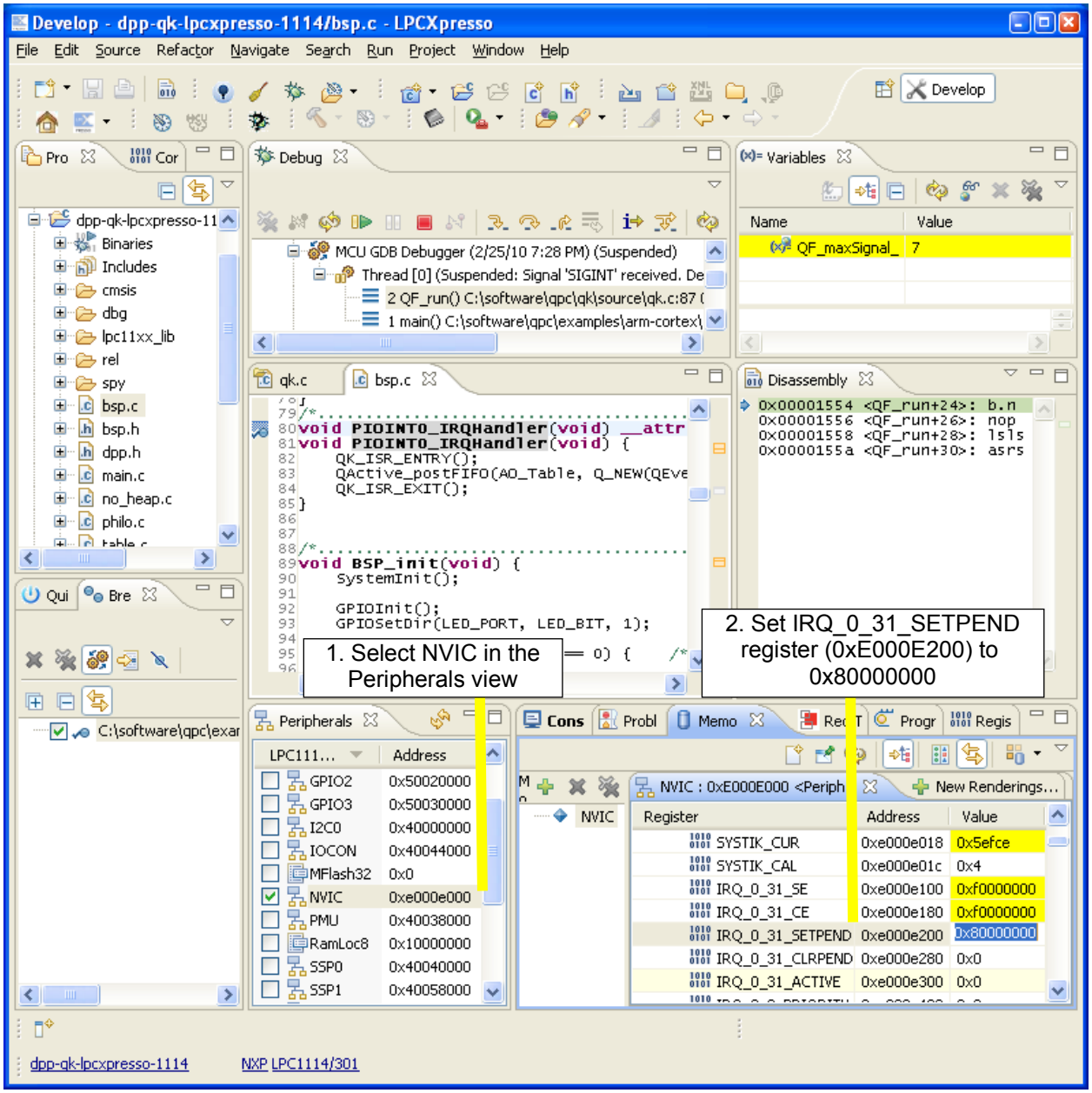
Finally, if you don't use the heap, which you shouldn't in robust, deterministic applications, you can reduce the C++ overhead even further (by about 2.8KB). The module `no_heap.cpp` provides dummy empty definitions of `malloc()` and `free()`:

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

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

6 Testing QK Preemption Scenarios

The DPP example application includes special instrumentation for convenient testing of various preemption scenarios, such as those illustrated in Figure 10. 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 PIOINT0 interrupt (`INTID` == 31).

Figure 10 Triggering the PIOINT0 interrupt from the Eclipse debugger.



The ISR for this interrupt is shown below:

```
void PIOINT0_IRQHandler(void) __attribute__((__interrupt__));
void PIOINT0_IRQHandler(void) {
    QK_ISR_ENTRY();                        /* inform QK-nano about ISR entry */
    QActive_postFIFO(AO_Table, Q_NEW(QEvent, MAX_PUB_SIG));  /* for testing */
    QK_ISR_EXIT();                          /* inform QK-nano 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 10 shows how to trigger the PIOINT0 interrupt from the Eclipse debugger. From the debugger you need to first open the register window and select NVIC registers from the drop-down list (see right-bottom corner of Figure 10). You scroll to the STIR register, which denotes the Software Trigger Interrupt Register in the NVIC. This write-only register is useful for software-triggering various interrupts by writing the INTID to it. To trigger the PIOINT0 interrupt (INTID == 31) you need to write 0x80000000 to the IRQ_0_31_SETPEND field by clicking on this field, entering the value, and pressing the Enter key.

The general testing strategy is to break into the application at an interesting place for preemption, set breakpoints to verify which path through the code is taken, and trigger the PIOINT0 interrupt. Next, you need to free-run the code (don't use single stepping) so that the NVIC can perform prioritization. You observe the order in which the breakpoints are hit. This procedure will become clearer after a few examples.

6.1.1 *Interrupt Nesting Test*

The first interesting test is verifying the correct tail-chaining to the PendSV exception after the interrupt nesting occurs. To test this scenario, you place a breakpoint inside the `PIOINT0_IRQHandler()` and also inside the `SysTick_Handler()` ISR. When the breakpoint is hit, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window) and also another breakpoint on the first instruction of the `QK_PendSV` handler. Next you trigger the PIOINT0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

1. The first breakpoint hit is the one inside the `PIOINT0_IRQHandler()` function, which means that PIOINT0 ISR preempted the SysTick ISR.
2. The second breakpoint hit is the one in the `SysTick_Handler()`, which means that the SysTick ISR continues after the PIOINT0 ISR completes.
3. The last breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is tail-chained only after all interrupts are processed.

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

6.1.2 *Task Preemption Test*

The next interesting test is verifying that tasks can preempt each other. You set a breakpoint anywhere in the `Philosopher` state machine code. You run the application until the breakpoint is hit. After this happens, you remove the original breakpoint and place another breakpoint at the very next machine instruction (use the Disassembly window). You also place a breakpoint inside the `PIOINT0_IRQHandler()` interrupt handler and on the first instruction of the `PendSV_Handler()` handler. Next you trigger the PIOINT0 interrupt per the instructions given in the previous section. You hit the Run button.

The pass criteria of this test are as follows:

4. The first breakpoint hit is the one inside the `PIOINT0_IRQHandler()` function, which means that PIOINT0 ISR preempted the `Philospher` task.
5. The second breakpoint hit is the one in `PendSV_Handler()` exception handler, which means that the PendSV exception is activated before the control returns to the preempted `Philosopher` task.
6. After hitting the breakpoint in QK `PendSV_Handler` handler, you single step into the `QK_scheduler_()`. You verify that the scheduler invokes a state handler from the PED state machine. This proves that the `Table` task preempts the `Philosopher` task.
7. After this you free-run the application and verify that the next breakpoint hit is the one inside the `Philosopher` state machine. This validates that the preempted task continues executing only after the preempting task (the `Table` state machine) completes.

6.1.3 Other Tests

Other interesting tests that you can perform include changing priority of the PIOINT0 interrupt to be lower than the priority of SysTick to verify that the PendSV is still activated only after all interrupts complete.

In yet another test you could post an event to `Philosopher` active object rather than `Table` active object from the `PIOINT0_IRQHandler()` function to verify that the QK scheduler will not preempt the `Philosopher` task by itself. Rather the next event will be queued and the `Philosopher` task will process the queued event only after completing the current event processing.

7 Related Documents and References

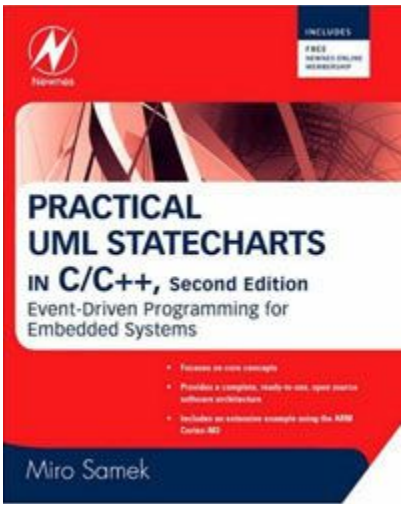
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP-Cortex] "Application Note: QP and ARM-Cortex with GNU", Quantum Leaps, 2013.	http://www.state-machine.com/arm/AN_QP_and_ARM-Cortex-GNU.pdf
[Samek+ 06b] "Build a Super Simple Tasker", Miro Samek and Robert Ward, Embedded Systems Design, July 2006.	http://www.embedded.com/showArticle.jhtml?articleID=190302110
[ARM 08a] "ARM v7-M Architecture Application Level Reference Manual", ARM Limited	Available from http://infocenter.arm.com/help/.
[ARM 08b] "Cortex™-M3 Technical Reference Manual", ARM Limited	Available from http://infocenter.arm.com/help/.
[CodeSourcery] Sourcery G++ Lite ARM EABI Sourcery G++ Lite Getting Started	Ihttp://www.codesourcery.com/sgpp/lite/arm/portal/doc2861/getting-started.pdf.
[LPC111x] "LPC111x Preliminary user manual Rev. 00.10" — 11 January 2010	Available in PDF from NXP at: http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc1111.lpc1112.lpc1113.lpc1114.pdf
[LPC1343] "LPC1311/13/42/43 User manual Rev. 01.01" — 11 January 2010	Available in PDF from NXP at: http://ics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc13xx.pdf

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