Application Note
QP™ and lwIP TCP/IP Stack

Document Revision L
January 2014

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

Table of Contents

1 Introduction........1
1.1 About lwIP........1
1.2 About QP™........2
1.3 Licensing QP™........3
1.4 About QP-lwIP Integration........3
1.5 Cortex Microcontroller Software Interface Standard (CMSIS)........3

2 Getting Started........4
2.1 What's Included in the QP-lwIP Example Code?........5
2.2 Software Installation........5
2.3 Defining Environment Variables........7
2.4 Building the ROM-Based File System........7
2.5 Building the Examples........8
2.6 Connecting the Board........9
2.7 Running the Examples........10
2.8 The lwIP Web-Server........12
2.9 The lwIP UDP Demo........16

3 Ethernet Device Driver for QP-lwIP........17
3.1 The Ethernet Device Driver Interface........18
3.2 Architecture-specific lwIP Header Files........19
3.3 The eth_driver_init() Function........19
3.4 The Ethernet ISR........22
3.5 The eth_driver_read() Function........23
3.6 The eth_driver_write() Function........24
3.7 The ethernetif_output() Callback Function........25

4 LwIPMgr Active Object........26
4.1 Launching and Configuring HTTP-Daemon and UDP/IP Applications........27
4.2 Implementing Server Side Include (SSI)........27
4.3 Implementing CGI........29
4.4 Implementing UDP........31
4.5 Assigning Priority to the LwIPMgr Active Object........32

5 Configuring and Customizing QP-lwIP........33

6 Related Documents and References........34

7 Contact Information........35

1 Introduction

This Application Note describes how to use the lightweight TCP/IP stack called lwIP with the QP™ state machine frameworks. This Application Note covers lwIP version **1.4.1** (the latest as of this writing) and QP/C and QP/C++ version **5.2.1** or higher.

1.1 About lwIP

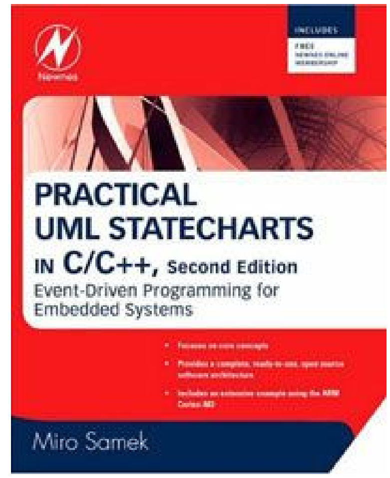
lwIP is a light-weight implementation of the TCP/IP protocol suite that was originally written by Adam Dunkels at the Computer and Networks Architectures (CNA) lab of the Swedish Institute of Computer Science but now is being actively developed by a team of developers distributed world-wide headed by Kieran Mansley.

lwIP is available under a BSD-style open source license in C source code format and can be downloaded from the development homepage at http://savannah.nongnu.org/projects/lwip. The focus of the lwIP is to reduce the RAM usage while still having a full scale TCP/IP implementation. This makes lwIP suitable for use in **embedded systems** with tens of kilobytes of RAM and around 40 KB of code ROM [Dunkels 01, Dunkels 07, lwIP-OS].

Since its release, lwIP has spurred a lot of interest and is today being used in many commercial products. lwIP has been ported to multiple platforms and operating systems and can be run either with or without an underlying OS. lwIP includes the following protocols and features [lwIP 1.3.0]:

- **IP** (Internet Protocol) including packet forwarding over multiple network interfaces
- **TCP** (Transmission Control Protocol) with congestion control, RTT estimation and fast recovery/fast retransmit
- **UDP** (User Datagram Protocol) including experimental UDP-lite extensions
- **ARP** (Address Resolution Protocol) for Ethernet
- **DHCP** (Dynamic Host Configuration Protocol)
- **AUTOIP** (for IPv4, conformant with RFC 3927)
- **ICMP** (Internet Control Message Protocol) for network maintenance and debugging
- **IGMP** (Internet Group Management Protocol) for multicast traffic management
- **Naive event-driven API** for enhanced performance
- Optional Berkeley-like **socket API**
- **DNS** (Domain names resolver
- **SNMP** (Simple Network Management Protocol)
- **PPP** (Point-to-Point Protocol)

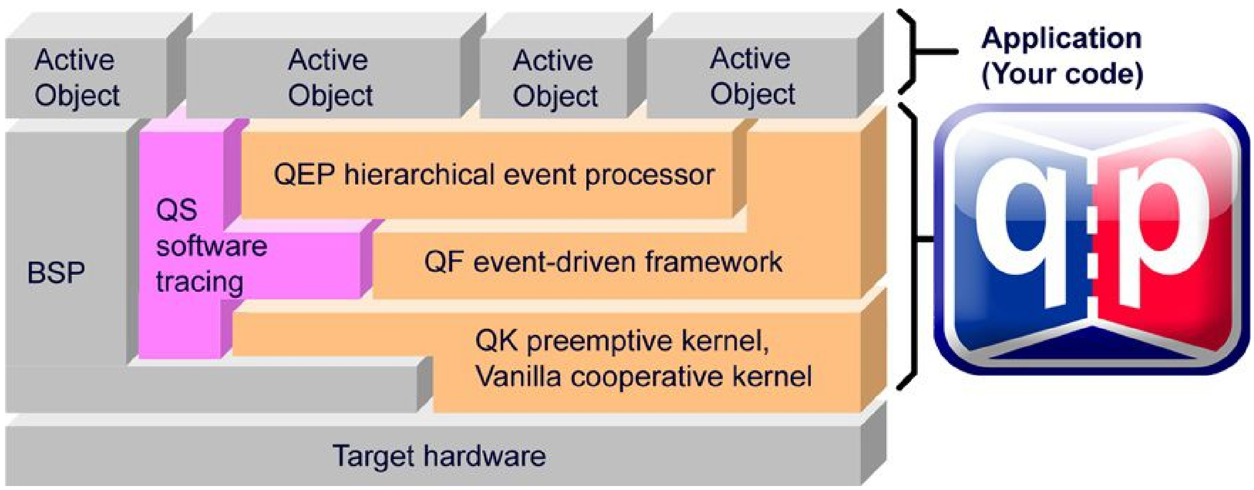
1.2 About QP™



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).

As shown in Figure 1, 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. The QP-lwIP integration described in this Application Note pertains to QP/C and QP/C++.

Figure 1: 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, and other popular OS/RTOS.

1.3 Licensing QP™

The **Generally Available (GA)** distributions of QP available for download from the www.state-machine.com/downloads are available under the following licensing terms:

- 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.

1.4 About QP-lwIP Integration

The QP-lwIP integration has been carefully designed for **hard real-time** control-type applications, in which the TCP/IP stack is used to monitor and configure the device as well as to provide remote user interface (e.g., by means of a web browser). In particular, The lwIP stack, which is **not reentrant**, is strictly encapsulated inside a dedicated active object (lwIP-Manager), so interrupt locking is unnecessary, which is critical for low interrupt latency. Also, the Ethernet interrupt service routine (ISR) runs very fast without performing any lengthy copy operations. This means that hard-real-time processing can be done at the task level, especially when you use the preemptive QK™ kernel built into QP for executing your application. **No external RTOS component is needed** to achieve fully deterministic real-time response of active object tasks prioritized above the lwiP task.

The QP-lwIP integration uses exclusively the event-driven lwIP API. The heavyweight Berkeley-like socket API requiring a blocking RTOS and is **not** used, which results in much better performance of the lwIP stack and less memory consumption.

NOTE: The lwIP source code has **not** been modified in any way to match the event-driven, run-to-completion execution model underlying QP. In other words, QP works with the standard lwIP code, as distributed from the lwIP homepage.

The QP-lwIP integration has been also carefully designed for **portability**. All hardware-specific code is clearly separated in the Ethernet/lwIP device driver with the clean interface to the lwIP stack and the QP application.

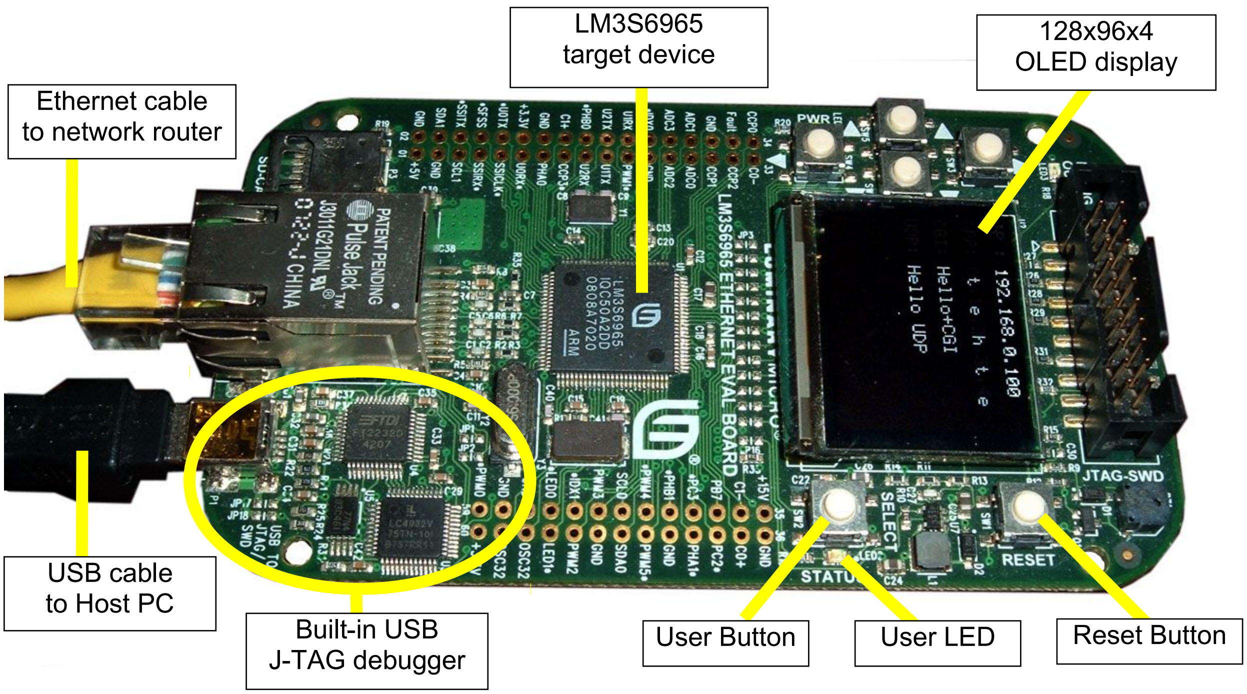
1.5 Cortex Microcontroller Software Interface Standard (CMSIS)

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

2 Getting Started

To focus the discussion, this Application Note uses the inexpensive EK-LM3S6965 Evaluation Kit based on the LM3S6965 Cortex-M3 MCU from Texas Instruments (see Figure 2). The example code has been compiled with the IAR EWARM KickStart™ edition, which is available for a *free download* from the IAR website www.iar.com. However, except for the Ethernet device driver that is specific to the Texas Instruments MCU, the rest of the code is generic and should be directly applicable to other CPUs and compilers without modifications.

Figure 2: Texas Instruments EK-LM3S6965 board with Ethernet and OLED display



The actual hardware/software used to test QP-lwIP integration is described below (see Figure 2):

1. Texas Instruments EK-LM3S6965 Evaluation Kit
2. IAR Embedded Workbench for ARM (EWARM) KickStart edition version **6.70**; or
3. Sourcery CodeBench (GNU + Eclipse IDE)
4. lwIP TCP/IP stack version **1.4.1**
5. QP/C or QP/C++ version **5.2.1** or higher

NOTE: The QP-lwIP examples assume that you are using the EK-LM3S6965 board **Revision C** or higher (please check the back of your board). At board Revision C Texas Instruments changed the graphical OLED display from OSRAM 128x64x4 to RITEK 128x96x4. If you happen to have the earlier board (Revision A or B), you can still use the examples by commenting out the macro `RITEK_OLED` and defining the macro `OSRAM_OLED` in `bsp.h`.

As shown in Figure 2, the EK-LM3S6965 board includes the built-in USB J-tag debugger and the target LM3S6965 target device with 64 KB single-cycle SRAM and 256 KB single-cycle flash ROM. However, the described QP-lwIP port should be applicable to smaller devices starting from some 20 KB of RAM and around 100 KB or ROM for code and data (such as web pages served over HTTP).

2.1 What's Included in the QP-lwIP Example Code?

This Application Note provides all you need to develop professional TCP/IP applications with lwIP, including embedded code and host-based utilities. The example code is based on the Dining Philosopher Problem (DPP) sample application described in Chapter 7 of [PSiCC2] as well as in the Application Note "Dining Philosopher Problem" [QL AN-DPP 08] (included in the example code distribution). The goal is to demonstrate lwIP running alongside an existing real-time application, as opposed to lwIP running all by itself that fails to show how lwIP can share the CPU and cooperate with other software components. The QP-lwIP example code includes the following components:

- The DPP example with lwIP for the cooperative "vanilla" kernel described in Chapter 7 of [PSiCC2]
- The DPP example with lwIP for the **preemptive** QK kernel described in Chapter 10 of [PSiCC2]
- lwIP source code version 1.4.1 (available also from http://savannah.nongnu.org/projects/lwIP)
- lwIP Ethernet device driver for the Texas Instruments Stellaris MCUs
- The web server (HTTP-Daemon) with Server-Side Includes (SSI) and rudimentary Common Gateway Interface (CGI) capabilities
- Example website consisting of multiple HTTP pages, graphics, and examples of SSI and CGI
- lwIP Example of using UDP communication to and from the embedded target

NOTE: Additionally, the **Qtools** collection of open source tools contains the `qfsgen.exe` utility for generating ROM-based file system data for the web pages as well as the `qudp.exe` host utility for generating and receiving UDP packets to and from the target. The **Qtools** collection is available for a separate download from www.state-machine.com/downloads.

2.2 Software Installation

The example code is distributed in a ZIP archive (qpc-`lwip_iar_ek-lm3s6965_<ver>.zip` for QP/C and `qpcpp-lwip_iar_lm3s6965_<ver>.zip` for QP/C, where `<ver>` stands for a specific QP version, such as 5.2.1). You can uncompress the archive into any directory. The installation directory you choose will be referred henceforth as `<root>`. The following Listing 1 shows the directory structure and selected files included in the QP distribution.

NOTE: The QP-lwIP example code does not include the platform-independent baseline code of QP™, which is available for a separate download from www.state-machine.com/downloads.
NOTE: This Application Note 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.

Listing 1: Selected directories and files after installing the QP-lwIP example code

```
<root>\                    - Root Directory you chose to install the software
  |
  +-doc\
  | +-QP_datasheet.pdf   - QP state machine frameworks datasheet
  | +-AN_DPP.pdf         - Application Note "Dining Philosopher Problem Example"
  | +-AN_QP_and_lwIP.pdf - This Application Note
```

```
| +-LwIP-STABLE-1.3.0.pdf             - lwIP Manual (the latest available)
| +-Using_lwIP_with_or_without_OS.pdf - lwIP Documentation
|
+-lwip-1.4.1\              - lwIP 1.4.1 source code
| +-doc\                   - lwIP documentation
| +-src\                   - lwIP source code
| | +-api\                 - lwIP high-level API (not used in the QP-lwIP)
| | +-core\                - lwIP core functionality
| | +-include\             - lwIP public include files
| | +-netif\               - lwIP generic network interface
|
+-qpc\                     - QP/C
| +-ports\                 - QP ports
| | +-arm-cm\              - ARM Cortex-M ports
| | | +-qk\                - QK ports (preemptive kernel)
| | | | +-gnu\             - GNU compiler
| | | | | +-dbg\           - Debug build
| | | | | | +-libqp_cortex-m3.a - QP library
| | | | | +- . . .
| | | | +-iar\             - IAR compiler
| | | | | +-dbg\           - Debug build
| | | | | | +-libqp_cortex-m3.a - QP library
| | | | | +-rel\           - Release build
| | | | | +-spy\           - Spy build
| | | | | +-make_cortex-m0.bat - Batch to build QP libraries for Cortex-M0 cores
| | | | | +-make_cortex-m3.bat - Batch to build QP libraries for Cortex-M3 cores
| | | | | +-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      - QK port to Cortex in assembly
| | | | | +-qs_port.h      - QS platform-dependent public include
| | | +-vanilla\           - Ports to the non-preemptive “vanilla” kernel
| | | | +-gnu\             - GNU compiler
| | | | | +- . . .
| | | | +-iar\             - IAR compiler
| | | | | +- . . .
| |
| +-examples\              - QP examples
| | +-arm-cm\              - ARM Cortex-M ports
| | | +-qk\                - Ports to the preemptive QK kernel
| | | | +-iar\             - IAR compiler
| | | | | +-lwip-qk_ek-lm3s6965\ - lwIP application for the EK-LM3S6965 board
| | | | | | +-dbg\             - Debug build
| | | | | | +-rel\             - Release build
| | | | | | +-spy\             - Spy build (instrumented with Q-SPY  tracing)
| | | | | | +-httpserver\      - HTTP server code
| | | | | | | +-fs\          - ROM-based file-system for web pages and graphics
| | | | | | | +-qfsgen.bat   - script to invoke QFSGEN utility for file-system
| | | | | | +-lwip_port \      - lwIP port and Ethernet device driver for LM3S6965
| | | | | | | +-arch\        - Architecture-dependent files
| | | | | | | +-netif\       - Network interface
| | | | | | +-bsp.c            - Board Support Package implementation
| | | | | | +-bsp.h            - Board Support Package header file
| | | | | | +-lm3s6965.icf     - Linker command file for LM3S6965 MCU
| | | | | | +-lwip.c           - consolidated lwIP implementation
| | | | | | +-lwip.h           - consolidated lwIP header file
```

```
| | | | | | +-lwipopts.h      - configuration options for lwIP
| | | | | | +-lwipmgr.c       - lwIP Manager active object
| | | | | | +-main.c          - main() function
| | | | | | +-rit128x96x4.c   - Driver for the OLED display of EK-LM3S6965
| | | | | | +-rit128x96x4.h   - Driver for the OLED display of EK-LM3S6965
| | | | | | +-philo.c         - Philosopher active objects
| | | | | | +-table.c         - Table active object
| | | | |
| | | +-vanilla\          - Ports to the non-preemptive "vanilla" kernel
| | | | +-iar\            - IAR compiler
| | | | | +-lwip_ek-lm3s6965\ - lwIP application for the EK-LM3S6965 board
| | | | | | +- ...   |
```

2.3 Defining Environment Variables

The QP-LwIP projects files provided with this Application Note assume that the environment variable **`QPC`** (for QP/C) and **`QPCPP`** (for QP/C++) are defined and that they point to the location of the QP/C framework or the QP/C++ framework, respectively. For example, assuming that you have installed QP/C into the directory `C:\qp\qpc`, you should define the environment variable `QPC` to `C:\qp\qpc`.

The QP-LwIP projects files provided with this Application Note also assume that the environment variable **`LWIP`** is defined and it points to the location of the LwIP software. For example, assuming that you have installed **`LWIP`** into the directory `C:\software\lwip-1.4.1`, you should define the environment variable LWIP to `C:\software\lwip-1.4.1`.

2.4 Building the ROM-Based File System

The **Qtools collection**, which is available for a separate download from www.state-machine.com/-downlaods, contains the `qfsgen` utility to generate ROM-based file system for your web page.

Figure 3: The `qfsgen` utility to generate the ROM-based file system

Administrator: Command Prompt

```
D:\software\qpc\examples\arm-cortex\vanilla\iar\lwip-ev-lm3s6965\webserver>qfsge
n.exe
qfsgen 4.2.00 (c) Quantum Leaps, www.state-machine.com
Usage: qfsgen [root-dir [output-file]] [-h]
          -h suppresses generation of the HTTP headers

    Adding: /404.htm...
    Adding: /bg_footer.gif...
    Adding: /cgi_demo.htm...
    Adding: /img/AN_QP_and_lwIP.jpg...
    Adding: /img/arrow.gif...
    Adding: /img/favicon.ico...
    Adding: /img/footer.jpg...
    Adding: /img/logo_lwip_qp.jpg...
    Adding: /img/logo_ql.jpg...
    Adding: /img/logo_sics.gif...
    Adding: /img/PSiCC2.gif...
    Adding: /img/QP_datasheet.gif...
    Adding: /index.htm...
    Adding: /ssi_demo.shtm...
    Adding: /style.css...
    Adding: /thank_you.htm...
    Adding: /udp_demo.htm...

Files processed: 17

D:\software\qpc\examples\arm-cortex\vanilla\iar\lwip-ev-lm3s6965\webserver>
```

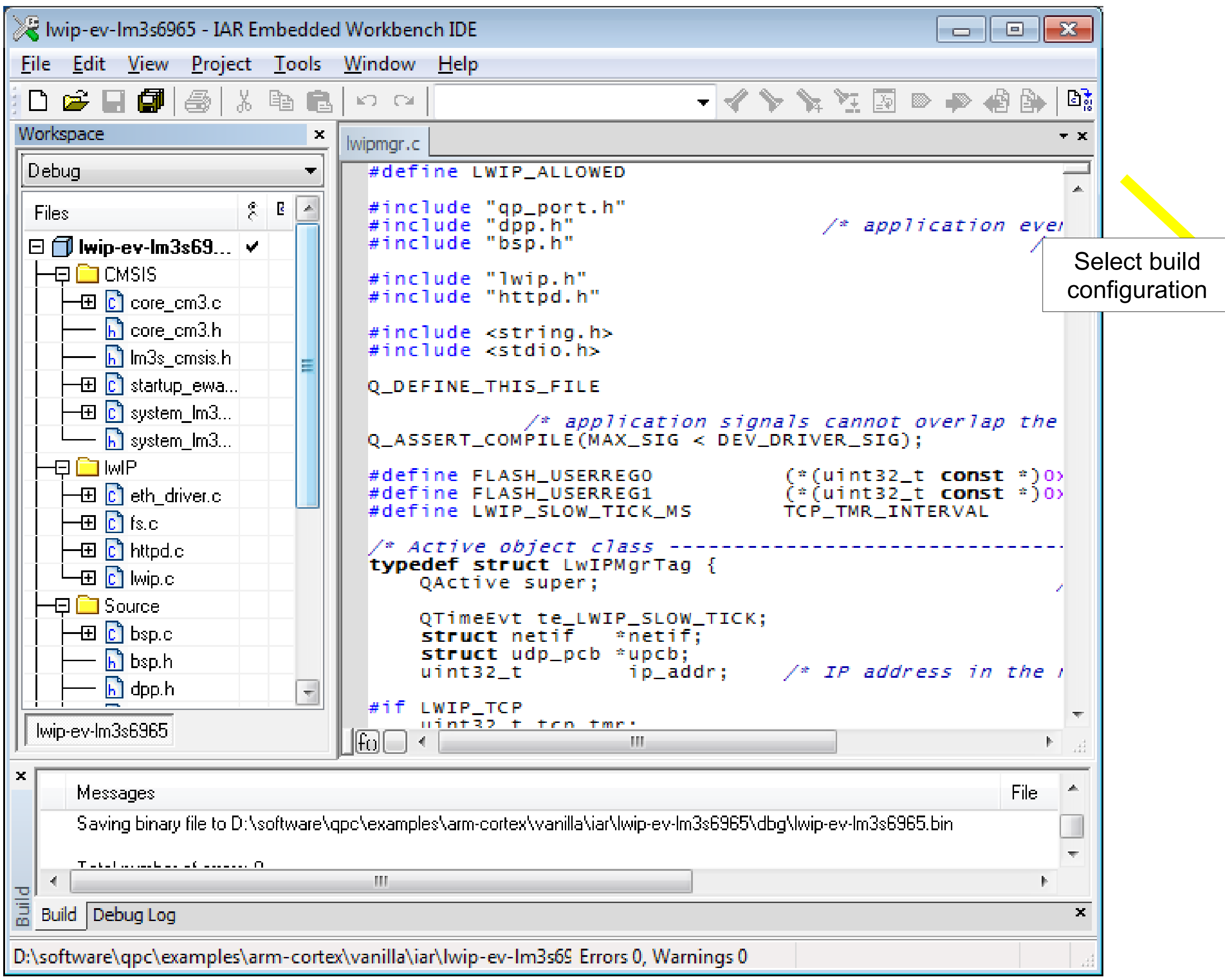
The read-only file system is generated as a set of constant byte arrays that can be included in the C source code, so that the compiler stores them in ROM. Figure 3 shows an example of output generated by the `qfsgen` utility.

> **NOTE:** The `qfsgen` utility is now included in the **Qtools collection**, which is available for a separate download from www.state-machine.com/downlaods. The following discussion assumes that you have downloaded and installed Qtools, including adding the Qtools directory to the PATH variable on your system.

2.5 Building the Examples

The examples accompanying this Application Note are based on the 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 `<root>\qpc\examples\cortex-m3\qk\iar\lwip_ek-lm3s6965\` contains the IAR workspaces and project that you can load into the IAR EWARM IDE, as shown in Figure 4.

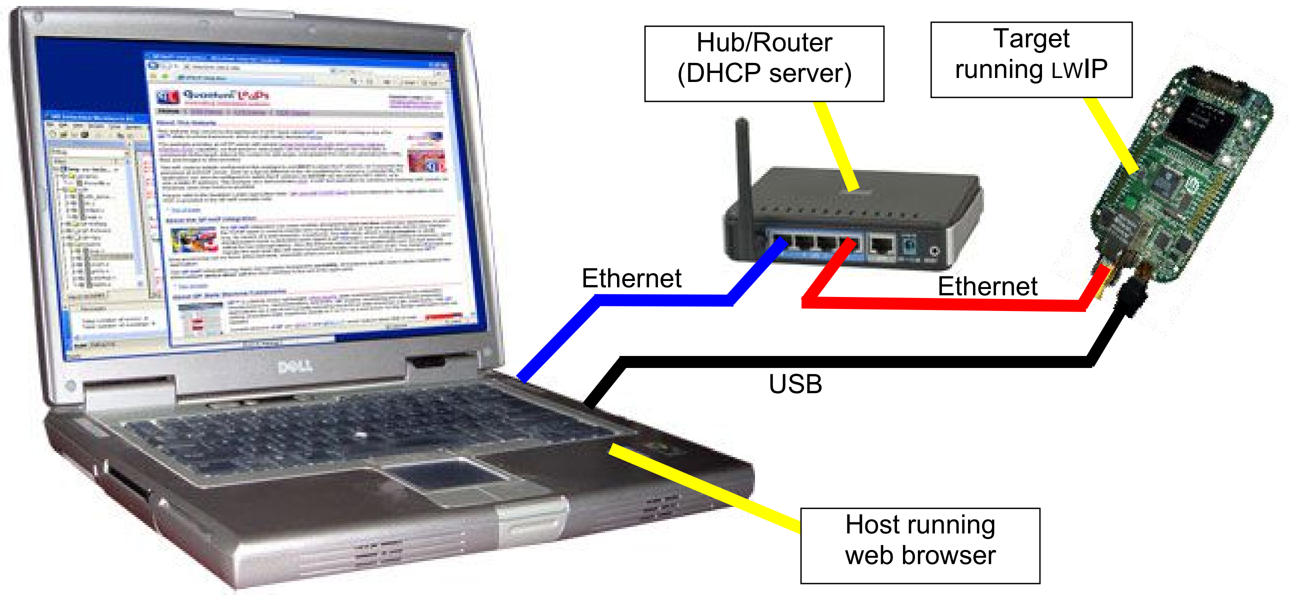
Figure 4: IAR EWARM IDE with the lwIP example



2.6 Connecting the Board

As shown in Figure 5, you connect the target board to the network hub/router using the regular Ethernet cable. You also connect the USB cable to the host machine to power up the board and to provide debugger connection for downloading and debugging the embedded code.

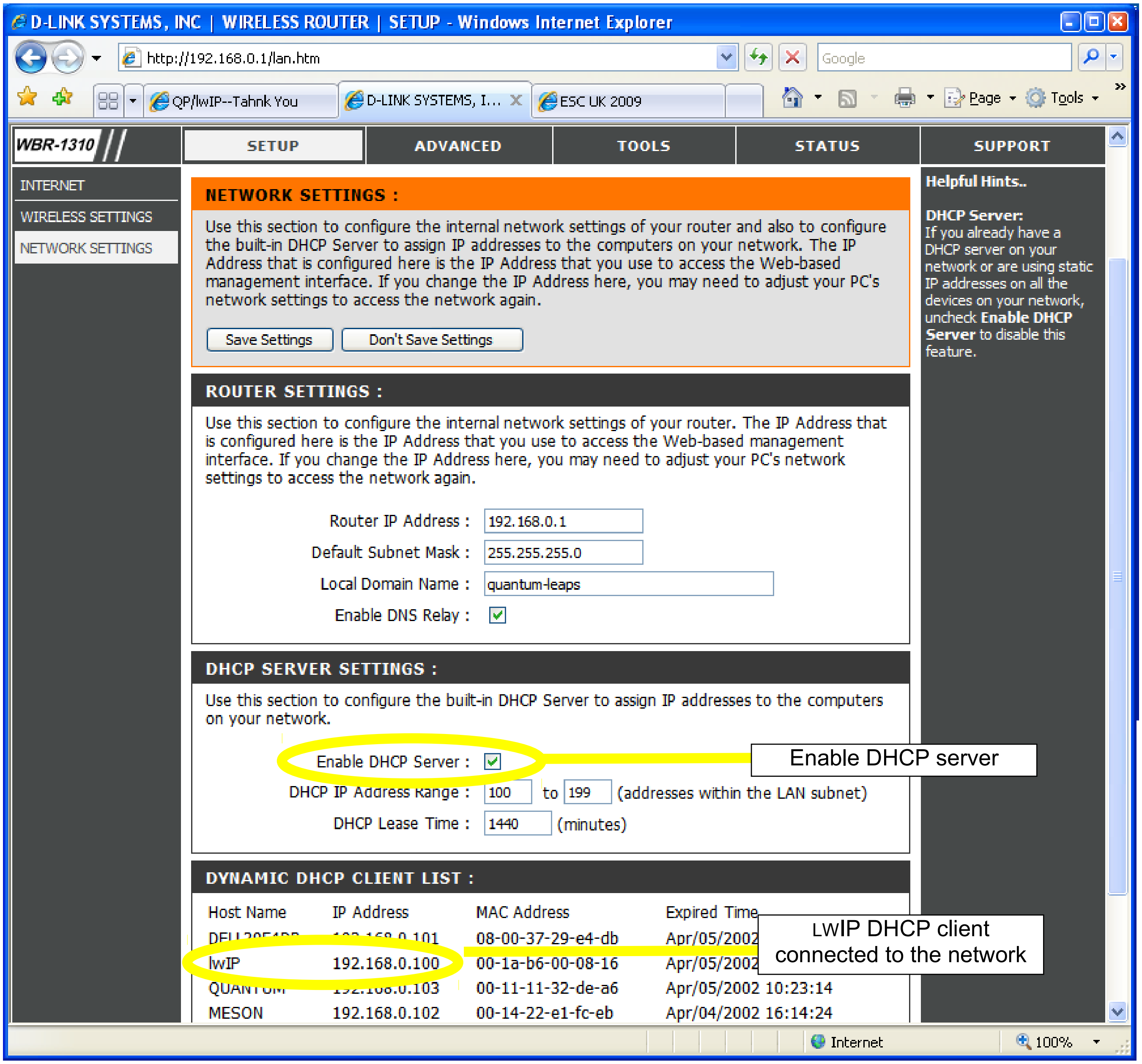
Figure 5: Connecting the target board to the network



The QP-lwIP examples assume that target board obtains its IP address from a **DHCP server** running on the network. Most internet hubs/routers provide DHCP server, but please make sure that the DHCP is actually enabled in your router. For example, Figure 6 shows how to enable DHCP server in the D-Link WBR-1310 router via the web-server user interface. Please refer to the manual of your router, but most hubs/routers work similarly.

NOTE: Because the OLED display of the EK-LM3S6965 board has burn-in characteristics similar to a CRT, the QP-lwIP example application turns of the screen after 30 seconds. You can always turn the screen back on by pressing the User Button (see Figure 2).

Figure 6: Accessing the D-Link router via the built-in web-server interface



2.7 Running the Examples

You program the code into the flash memory of the MCU through the IAR EWARM IDE by selecting Project | Download and Debug option. You run the program by selecting the Debug | Run menu (F5), or by clicking on the Run button. The OLED display should show the initial IP address of 0.0.0.0 as well as changing status of the Dining Philosophers. After several seconds, the IP address should change to something like 192.168.0.xxx, which means that the target board has obtained the IP address from the DHCP server and is ready to communicate via TCP/IP or UDP/IP.

2.7.1 *Testing the TCP/IP Connection to the Target*

The lwIP stack contains the rudimentary implementation of ICMP, so once your target system obtains the IP address you can ping it. For example the following screen shot shows the `ping` utility running on Windows. (Please note that the first very long timeout of over 11s is not caused by the latency of the lwIP stack, but rather the firewall of the host PC.)

```
Command Prompt
C:\tmp>ping 192.168.0.100
PING 192.168.0.100 (192.168.0.100): 56 data bytes
64 bytes from 192.168.0.100: icmp_seq=0 ttl=255 time=11316 ms
64 bytes from 192.168.0.100: icmp_seq=1 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=2 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=3 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=4 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=5 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=6 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=7 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=8 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=9 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=10 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=11 ttl=255 time=0 ms
64 bytes from 192.168.0.100: icmp_seq=12 ttl=255 time=0 ms

----192.168.0.100 PING Statistics----
13 packets transmitted, 13 packets received, 0.0% packet loss
round-trip (ms)  min/avg/max/med = 0/870/11316/0

C:\tmp>
```

The following screen shots show also the results of `pathping` and `tracert` utilities:

```
Command Prompt
C:\tmp>pathping 192.168.0.100

Tracing route to 192.168.0.100 over a maximum of 30 hops

  0  QUANTUM.quantum-leaps [192.168.0.103]
  1  192.168.0.100

Computing statistics for 25 seconds...
            Source to Here   This Node/Link
Hop  RTT    Lost/Sent = Pct  Lost/Sent = Pct  Address
  0                                           QUANTUM.quantum-leaps [192.168.0.1
03]
                                0/ 100 =  0%   |
  1    0ms     0/ 100 =  0%     0/ 100 =  0%  192.168.0.100

Trace complete.

C:\tmp>tracert 192.168.0.100

Tracing route to 192.168.0.100 over a maximum of 30 hops

  1    <1 ms    <1 ms    <1 ms  192.168.0.100

Trace complete.

C:\tmp>
```

2.8 The lwIP Web-Server

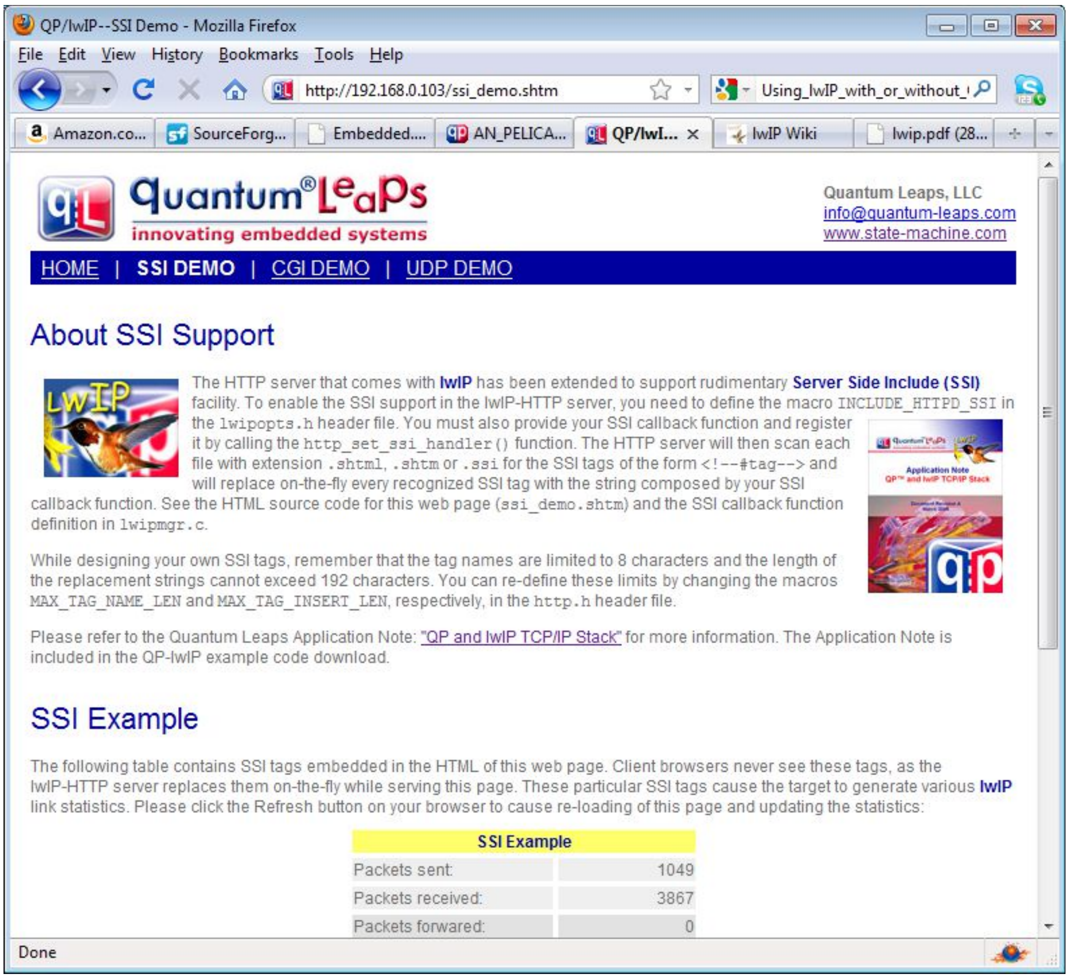
The QP-lwIP example provides a web-server (HTTP **1.0**), which you can access by pointing any standard web browser to the URL: `http://<IP address>`, where `<IP address>` is the IP address shown in the OLED display of the target board. The lwIP web server example demonstrates that you can use HTML pages, graphics (GIF, PNG, JPEG), Cascaded Style Sheets (CSS), and plain text.

Figure 7: Home page served by the QP-lwIP example application

2.8.1 Server-Side Includes (SSI) Demo

The lwIP web server has been extended to support rudimentary **Server Side Include (SSI)** facility. The lwIP web server implements SSI by replacing any HTML tag of the form `<!--#tag-->` with the dynamically generated string that corresponds to that tag. The server scans the served HTML for SSI flags only in files with the extensions `.shtm`, `.shtml`, or `.ssi`.

Figure 8: Server-Side Includes (SSI) demo web page



For example, Figure 8 shows a web page `ssi_demo.shtm` with several SSI tags embedded in the table that shows the lwIP link statistics. Each of these tags causes invocation of a callback function in the target, which dynamically synthesizes a string representing a certain lwIP statistics in this case. Because each of the SSI tags is sent in a separate TCP/IP session. For example, serving the entire SSI Demo web page with 16 SSI tags takes noticeably longer (some 3-4 seconds) than web pages without SSI tags.

While designing your own SSI tags, remember that the tag names are limited to 8 characters and the length of the replacement strings cannot exceed 192 characters. You can re-define these limits by changing the macros `MAX_TAG_NAME_LEN` and `MAX_TAG_INSERT_LEN`, respectively, in the `httpd.h` header file.

2.8.2 *Common Gateway Interface (CGI) Demo*

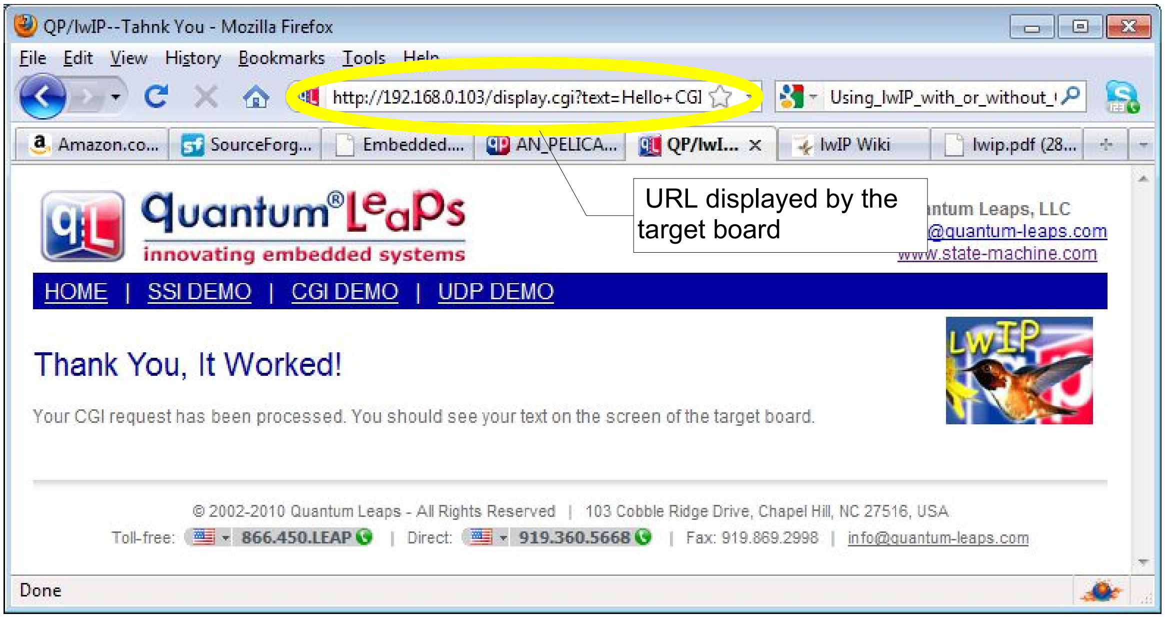
The lwIP web server has been extended to support rudimentary **Common Gateway Interface (CGI)** facility. CGI enables you to send commands with parameters to the embedded target via the lwIP web server. For example, you can place an HTML form on your webpage. When the user submits the form using the GET method, the lwIP web server will recognize a CGI request and will invoke a registered callback function in the target. The lwIP web server will then serve another webpage returned by the CGI callback.

Figure 9: Common Gateway Interface (CGI) demo web page

For example, Figure 9 shows a web page `cgi_demo.htm` with an HTML form that allows the user to enter a short text. When the user presses the Submit button, the text is embedded in the CGI request and will be displayed on the OLED display of the target board. The lwIP web server will then serve the `thank_you.htm` page, as shown in Figure 10.

As mentioned before, the current CGI implementation works only with the GET method, which encodes all parameters in the URI (e.g., "`display.cgi?text=Hello+CGI&submit=Submit`", see Figure 10). The lwIP web server parses the URI and breaks it up into separate parameters. Currently the maximum number of CGI parameters cannot exceed 16, but you can re-define this limit by changing the macro MAX_CGI_PARAMS in the `httpd.h` header file.

Figure 10: Thank you page served by the example after processing the CGI request



NOTE: The web browser adds some embellishments to the parameters combined into the URI before submitting the URI to the HTTP server. For example the original text "Hello CGI" has been changed to "Hello+CGI". You should be aware of such changes.

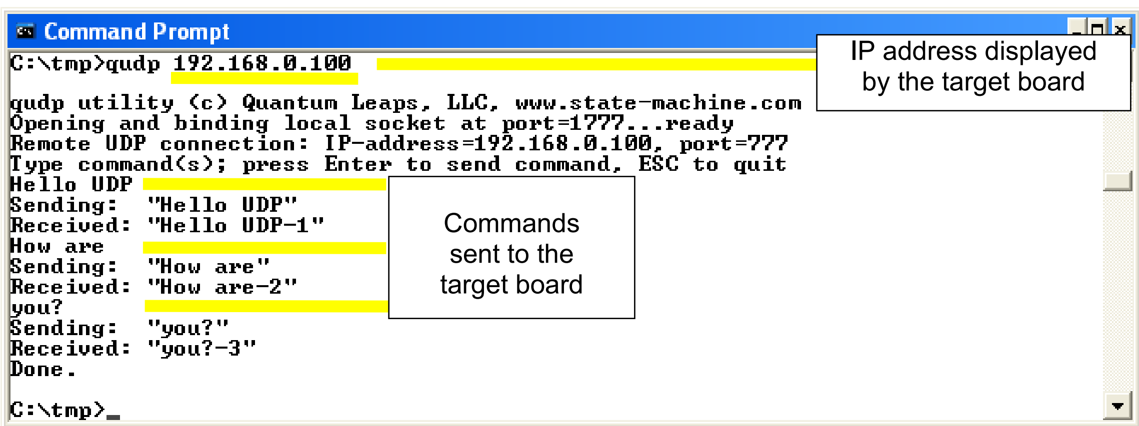
2.9 The lwIP UDP Demo

The QP-lwIP example application demonstrates also UDP communication to and from the embedded target. The embedded target opens a UDP connection and binds it the local port 777. Every UDP packet sent to this connection is interpreted as text to be displayed on the screen of the target board. The target board then adds a sequence number to the original text and sends it back to the same remote IP address and port number that has sent the original packet.

To facilitate testing of UDP connectivity, a simple console application called `qudp` for Windows or Linux hosts is provided in the **Qtools collection**, which is available for a separate download from www.state-machine.com/downlaods. Figure 11 shows an example output generated from the `qudp` utility. Each submitted command should be seen as text displayed on the target board. The `qudp` application provides command-line parameters, which let you override the default port numbers. If launched without any parameters, `qudp` will print the usage help.

NOTE: The `qudp` utility is now included in the **Qtools collection**, which is available for a separate download from www.state-machine.com/downlaods. The following discussion assumes that you have downloaded and installed Qtools, including adding the Qtools directory to the PATH variable on your system.

Figure 11: Example output generated by the qudp.exe utility



3 Ethernet Device Driver for QP-lwIP

The Ethernet device driver for QP-lwIP provides interface to the physical network hardware. The general rule in the design of the device driver is that the lwIP code must be strictly called from one thread of execution only, because the lwIP code is **not reentrant**. In the context of QP, the only thread allowed to execute any lwIP code is the thread context of the `LwIPMgr` active object.

NOTE: The UML term **active object** stands for an autonomous state machine executing in its own thread of control and communicating with other active object by asynchronous event exchange. In QP each active object has a private event queue and a unique priority. Please refer to the book "Practical UML Statecharts in C/C++, Second Edition" [PSiCC2] for more information.

The other principle is that the Ethernet Interrupt Service Routine (ISR), which is part of the Ethernet device driver, should not perform any lengthy copy operations, but instead should only post events to the `LwIPMgr` active object whenever a new packet has arrived or when a packet has been transmitted.

The Ethernet device driver for the LM3S6965 MCU is located in the directory `<root>\qpc\examples\arm-cm\`**`vanilla`**`\iar\lwip_ek-lm3s6965\lwip_port\netif\` for the non-preemptive "vanilla" kernel built into QP and in the directory `<root>\qpc\examples\arm-cm\`**`qk`**`\iar\lwip_ek-lm3s6965\lwip_port\netif\` for the preemptive QK kernel [LM3S6965 08]. In both cases the Ethernet device driver consists of two files `eth_driver.h` and `eth_driver.c`.

Figure 12: General structure of the QP-lwIP integration.
The elements of the Ethernet device driver are shown in grey.

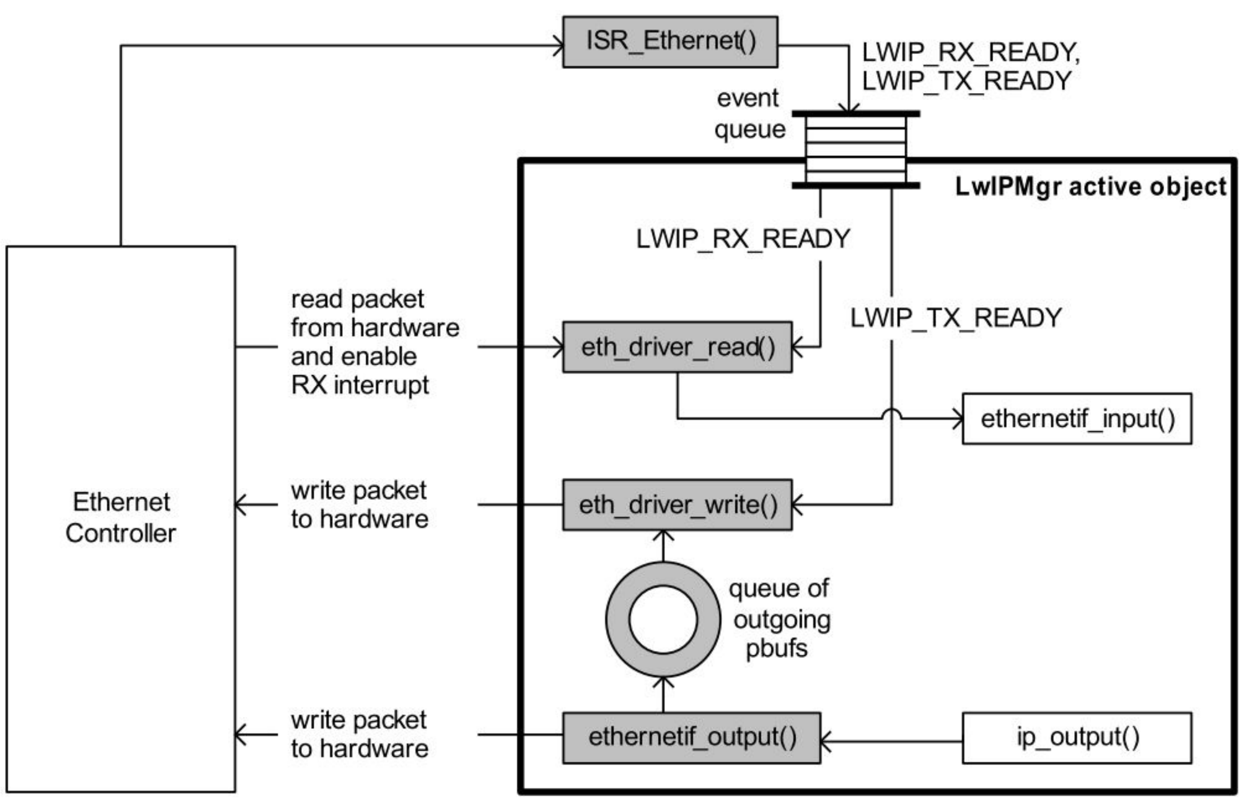


Figure 12 shows in general terms how lwIP integrates with the QP event-driven framework. The Ethernet ISR (`ISR_Ethernet()`) posts events LWIP_RX_READY and LW_TX_READY to the `LwIPMgr` active object when the Ethernet Controller receives or transmits a packet, respectively. These events don't carry any payload and the Ethernet ISR does **not** read or write any data to or from the Ethernet Controller. Data copying would take too much time in the interrupt context and would extend the task-level response. (It would also require accessing lwIP code from the interrupt context, which is not allowed due to the non-reentrant nature of the lwIP code.) Instead, whenever the Ethernet ISR posts the LWIP_RX_READY event, it disables further RX interrupts to prevent flooding the `LwIPMgr` active object with events.

All actual data reading or writing to and from the Ethernet Controller occurs in the context of the `LwIPMgr` active object. Specifically, `LwIPMgr` state machine calls the device driver function `eth_read()` as the action for the event LWIP_RX_READY and `eth_write()` as the action for the event LWIP_TX_READY, respectively.

The hardware-specific functions `eth_driver_read()` and `eth_driver_write()`, which belong to the Ethernet device driver, perform the actual reading and writing of packets to and from the Ethernet Controller, respectively. The function `eth_driver_read()` reads the Ethernet packet into the allocated lwIP packet buffer (pbuf structure) and then it calls the raw lwIP API function `ethernetif_intput()`, which passes the pbuf up the TCP/IP stack for processing.

Eventually, the call chain initiated from `eth_driver_read()` can produce new packets for transmission. New packets can also be produced as result of some external events posted to the `LwIPMgr` active object. For example, an event can trigger sending a UDP packet.

Regardless how a transmit packet (pbuf structure) is generated, it always funnels through the lwIP function `ip_output()`, which calls a registered callback `(*netif->linkoutput)()` to output the pbuf to the hardware. The lwIP Ethernet device driver registers the function `ethernetif_output()` (see Figure 12) as the callback function stored in the `netif->linkoutput` pointer-to-function.

The `ethernetif_output()` function tries to write the pbuf directly to hardware, but when the hardware cannot accept more data at this moment, the `ethernetif_output()` function stores the pbuf in the ring buffer for transmission at a later time. This buffering of pbufs **avoids blocking** the caller thread when the Ethernet Controller runs out of space for transmit packets. The ring buffer of pbufs stored for transmission is emptied by the function `eth_driver_write()`, which the `LwIPMgr` state machine calls as the action for the event LWIP_TX_READY. After writing each pbuf to the hardware, the `eth_driver_write()` function frees the pbuf.

The following sub-sections explain the elements of the Ethernet device driver in more detail.

3.1 The Ethernet Device Driver Interface

The lwIP Ethernet device driver presents a very simple event-driven interface to the QP application, as shown in Listing 2.

Listing 2: lwIP Ethernet Device Driver for QP (file <root>\qpc\examples\arm-cm\qk\iar\lwip-qk-ek-lm3s6965\lwip_port\netif\eth_driver.h)

```
(1) struct netif *eth_driver_init(QActive *active,
                                  enum_t base_sig,
                                  u8_t macaddr[NETIF_MAX_HWADDR_LEN]);
(2) void eth_driver_read(void);
(3) void eth_driver_write(void);

    enum EthDriverSignals {
```

```
(4)       LWIP_RX_READY_OFFSET,
(5)       LWIP_TX_READY_OFFSET,
(6)       LWIP_RX_OVERRUN_OFFSET,
(7)       LWIP_MAX_OFFSET
      };
```

(1) The function `eth_driver_init()` initializes the Ethernet Controller hardware and the lwIP stack. The function takes the pointer to the active object that encapsulates the lwIP stack as well as the base signal for the LwIP events generated by this driver. The function returns the pointer to the network interface. The `eth_driver_init()`, as all other code that calls lwIP facilities can be called only from the dedicated lwIP active object, typically from its top-most initial transition.

(2) The function `eth_driver_read()` reads one packet from the Ethernet Controller and passes it for to the lwIP stack for processing. This function can be called only from the dedicated lwIP active object, typically as action for the LWIP_RX_READY event.

(3) The function `eth_driver_write()` writes one packet to the Ethernet Controller, if a packet is available in the queue of outgoing pbufs. This function can be called only from the dedicated lwIP active object, typically as action for the LWIP_TX_READY event.

(4-6) The device driver interface defines the private signals offsets from the signal based provided in the function `eth_driver_init()` (see step (1)). These signals will be posted from the Ethernet ISR to the dedicated lwIP active object, whose pointer is also provided in `eth_driver_init()`.

(7) The last signal LWIP_MAX_OFFSET is defined so that the application can easily assert that the LWIP signals don't overlap any other signal group.

3.2 Architecture-specific lwIP Header Files

The lwIP code expects several architecture-specific header files in the `lwip_port\arch\` subdirectory (see Listing 1). These header files provide the fixed-size integer types used in lwIP as well as the non-standard, compiler-specific directives for packing structures. The architecture-specific header files could also define the interrupt locking policy for protecting certain parts of the lwIP code. However, in the QP-lwIP integration, no such protection is needed, because the lwIP stack is strictly encapsulated in a single thread of execution of the dedicated active object.

NOTE: The QP-lwIP integration does not need to use interrupt locking to protect integrity of the lwIP stack. This has very beneficial effects for the low interrupt latency and enables running the lwIP stack in the context of a hard-real-time QP application.

3.3 The eth_driver_init() Function

The Ethernet device driver is initialized by the function `eth_driver_init()` shown in Listing 3. This function, as all other lwIP code is invoked from the QP active object dedicated to lwIP (`LwIPMgr` in this example).

Listing 3: Ethernet device driver initialization (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwip_port\netif\eth_driver.c)

```
(1) struct netif *eth_driver_init(QActive *active
                                  enum_t base_sig,
                                  u8_t macaddr[NETIF_MAX_HWADDR_LEN])
```

```
     {
             struct ip_addr ipaddr;
             struct ip_addr netmask;
             struct ip_addr gw;

 (2)         lwip_init();                                  /* initialize the lwIP stack */

 (3)         l_active = active; /*save the active object associated with this driver */

                                   /* set up the static events issed by this driver... */
 (4)         l_lwipEvt[LWIP_RX_READY_OFFSET]  .sig = base_sig + LWIP_RX_READY_OFFSET;
 (5)         l_lwipEvt[LWIP_TX_READY_OFFSET]  .sig = base_sig + LWIP_TX_READY_OFFSET;
 (6)         l_lwipEvt[LWIP_RX_OVERRUN_OFFSET].sig = base_sig + LWIP_RX_OVERRUN_OFFSET;

         #if LWIP_NETIF_HOSTNAME
             l_netif.hostname = "lwIP";              /* initialize interface hostname */
         #endif
             l_netif.name[0] = 'Q';
             l_netif.name[1] = 'P';

             NETIF_INIT_SNMP(&l_netif, snmp_ifType_ethernet_csmacd, 1000000);

 (7)         l_netif.output     = &etharp_output;

 (8)         l_netif.linkoutput = &ethernetif_output;

 (9)         PbufQueue_ctor(&l_txq);                  /* initialize the TX pbuf queue */

         #if (LWIP_DHCP == 0) && (LWIP_AUTOIP == 0)
                       /* No mechanism of obtaining IP address specified, use static IP: */
(10)         IP4_ADDR(&ipaddr,  STATIC_IPADDR0,    STATIC_IPADDR1,
                                STATIC_IPADDR2,    STATIC_IPADDR3);
(11)         IP4_ADDR(&netmask, STATIC_NET_MASK0,  STATIC_NET_MASK1,
                                STATIC_NET_MASK2,  STATIC_NET_MASK3);
(12)         IP4_ADDR(&gwaddr,  STATIC_GW_IPADDR0, STATIC_GW_IPADDR1,
                                STATIC_GW_IPADDR2, STATIC_GW_IPADDR3);
         #else
              /* either DHCP or AUTOIP are configured, start with zero IP addresses: */
(13)         IP4_ADDR(&ipaddr,  0, 0, 0, 0);
(14)         IP4_ADDR(&netmask, 0, 0, 0, 0);
(15)         IP4_ADDR(&gw,      0, 0, 0, 0);
         #endif
                       /* add and configure the Ethernet interface with default settings */
(16)         netif_add(&l_netif,
                       &ipaddr, &netmask, &gw,              /* configured IP addresses */
                       active,                       /* use this active object as the state */
                       &ethernetif_init,           /* Ethernet interface initialization */
                       &ip_input);                      /* standard IP input processing */

(17)         netif_set_default(&l_netif);

(18)         netif_set_up(&l_netif);                          /* bring the interface up */

         #if (LWIP_DHCP != 0)
(19)         dhcp_start(&l_netif);          /* start DHCP if configured in lwipopts.h */
             /* NOTE: If LWIP_AUTOIP is configured in lwipopts.h and
```

```
         * LWIP_DHCP_AUTOIP_COOP is set as well, the DHCP process will start
         * AutoIP after DHCP fails for 59 seconds.
         */
     #elif (LWIP_AUTOIP != 0)
(20)     autoip_start(&l_netif);     /* start AutoIP if configured in lwipopts.h */
     #endif

         /* Enable Ethernet TX and RX Packet Interrupts. */
(21)     ETH->IM |= (ETH_INT_RX | ETH_INT_TX);
     #if LINK_STATS
(22)     ETH->IM |= ETH_INT_RXOF;
     #endif

(23)     return &l_netif;
     }
```

(1) The QP-lwIP Ethernet device driver initialization function is designed to be called from the top-most initial transition of the active object dedicated to executing lwIP code. The function takes the pointer to the lwIP active object and returns the network interface pointer (`struct netif`), so the active object can refer to the network interface.

NOTE: This device driver is designed to handle just one network interface.

(2) The function `lwip_init()` initializes the lwIP stack.

(3) The pointer to the dedicated lwIP active object is stored in the local variable, so that the device driver can post events directly to the active object.

(4-6) The signals of the static events posted from this device driver are initialized from the provided base.

(7) The callback function for IP output is set directly to lwIP function `etharp_output()`.

(8) The callback function for link output is set to the device driver function `ethernetif_output()` (see Listing 7)

(9) The queue of the outgoing pbufs is initialized.

(10-12) If DHCP or AUTOIP are not configured in `lwipopts.h`, the target has no means of acquiring the IP address. In this case the static IP is configured, whereas the constant IP addresses are also defined in the `lwipopts.h` header file.

(13-15) If DHCP or AUTOIP are configured in `lwipopts.h`, the initial IP address is configured to 0.0.0.0.

(16) The new network interface is added to lwIP and configured. The `l_netif` variable is local to the Ethernet device driver and is encapsulated.

(17-18) The network interface is set as default and is also set up.

(19) The DHCP processing is started, if configured. Also, if the macro `LWIP_DHCP_AUTOIP_COOP` is setup as well in `lwipopts.h`, DHCP will automatically launch AUTOIP if it fails to acquire IP address within a minute.

(20) Alternatively, if only AUTOIP is configured (and DHCP isn't), the AUTOIP processing is started right away.

(21) The RX and TX interrupts are enabled in the Stellaris Ethernet Controller.

(22) When the lwIP link statistics are enabled, the RX FIFO overrun interrupts are enabled in the Stellaris Ethernet Controller.

(23) Pointer to the network interface is returned to the caller, which is the `LwIPMgr` active object.

3.4 The Ethernet ISR

The Ethernet ISR (`ISR_Ethernet()`), shown in Listing 4, is simple and deterministic. In particular, it does not perform any lengthy copy operations to or from the Ethernet Controller.

Listing 4: Ethernet ISR (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwip_port\netif\eth_driver.c)

```
 (1) void ISR_Ethernet(void) {
         unsigned long eth_stat;

     #ifdef QK_ISR_ENTRY
 (2)     QK_ISR_ENTRY();                              /* inform QK about ISR entry */
     #endif


 (3)     eth_stat = ETH->RIS;

 (4)     ETH->IACK = eth_stat;                      /* clear the interrupt sources */

 (5)     eth_stat &= ETH->IM;                     /* mask only the enabled sources */

 (6)     if ((eth_stat & ETH_INT_RX) != 0) {
 (7)         QACTIVE_POST(l_active, &l_lwipEvt[LWIP_RX_READY_OFFSET],
                          &l_Ethernet_IRQHandler);
 (8)         ETH->IM &= ~ETH_INT_RX;                          /* disable further RX */
         }
 (9)     if ((eth_stat & ETH_INT_TX) != 0) {
(10)         QACTIVE_POST(l_active, &l_lwipEvt[LWIP_TX_READY_OFFSET],
                          &l_Ethernet_IRQHandler);
         }
     #if LINK_STATS
(11)     if ((eth_stat & ETH_INT_RXOF) != 0) {
(12)         QACTIVE_POST(l_active, &l_lwipEvt[LWIP_RX_OVERRUN_OFFSET],
                          &l_Ethernet_IRQHandler);
         }
     #endif

     #ifdef QK_ISR_EXIT
(13)     QK_ISR_EXIT();                                /* inform QK about ISR exit */
     #endif
     }
```

(1) In ARM Cortex the ISRs are just regular C functions. In most other CPU architectures ISRs require special prologue and epilogue code synthesized by the C compiler.

(2) The QK preemptive kernel is informed about entering the ISR.

> **NOTE**: This step is absolutely essential, but is only necessary when the QK kernel is used. The cooperative "vanilla" kernel does not need to be informed about entering an ISR.

(3) The interrupt status of the Ethernet Controller is read.

(4) All interrupt sources are explicitly cleared.

(5) The disabled interrupt sources are masked off.

NOTE: In the Stellaris Ethernet Controller, the interrupt status reports all possible interrupt sources, including sources that are actually disabled for generating interrupts. Therefore it is important to mask off the disabled interrupt sources before generating events based on the current Ethernet Controller interrupt status.

---

(6) If the interrupt status indicates reception of a packet…

(7) The event LWIP_RX_READY is posted directly to the `l_active` active object. The `l_active` pointer is initialized to point to `LwIPMgr` active object upon the initialization of the Ethernet device driver. Please also note that the posted event `evt_eth_rx` is allocated statically, because it has no payload.

(8) The further RX interrupts are **disabled** to prevent flooding the `LwIPMgr` active object with the LWIP_RX_READY events. The `LwIPMgr` active object will re-enable RX interrupt after it reads the actual packets from the Ethernet Controller hardware.

NOTE: In the Stellaris Ethernet Controller, as most Ethernet Controllers, provides buffer space (2KB in case of Stellaris), which stores arriving packets while the software is not ready to immediately read the data. This buffering of packets works independently from the interrupt status. In other words, even though RX interrupt are disabled, the Ethernet Controller keeps receiving packets as long as it has free buffer space.

---

(9) If the interrupt status indicates transmission of a packet…

(10) The event LWIP_TX_READY is posted directly to the `l_active` active object to trigger output of the next packet, if available. Please note that the posted event `evt_eth_tx` is allocated statically, because it has no payload.

(11) If the Ethernet Controller reports RX FIFO overrun this indicates that the software didn't read the incoming packets fast enough (see explanation to line (8))

(12) The event LWIP_RX_OVERRUN is posted directly to the `l_active` active object to update the error statistics. (Please note that only the dedicated active object can access the lwIP code.)

(13) The QK preemptive kernel is informed about entering the ISR.

NOTE: This step is absolutely essential, but is only necessary when the QK kernel is used. The cooperative "vanilla" kernel does not need to be informed about exiting an ISR.

3.5 The eth_driver_read() Function

Listing 5 shows the implementation of the `eth_driver_read()` function, which the `LwIPMgr` active object calls upon reception of the LWIP_RX_READY event.

Listing 5: The eth_driver_read() Ethernet device driver function (file qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwip_port\netif\eth_driver.c)

```
     void eth_driver_read(void) {
(1)      struct pbuf *p = low_level_receive();
         if (p != NULL) {                     /* new packet received into the pbuf? */
(2)          if (ethernet_input(p, &l_netif) != ERR_OK) {    /* packet processed? */
                 LWIP_DEBUGF(NETIF_DEBUG, ("eth_driver_input: input error\n"));
(3)              pbuf_free(p);
             }

             /* try to output a packet if TX fifo is empty and pbuf is available */
```

```
(4)          if ((ETH->TR & MAC_TR_NEWTX) == 0) {
                 p = PbufQueue_get(&l_txq);
                 if (p != NULL) {
                     low_level_transmit(p);
(5)                  pbuf_free(p);     /* free the pbuf, lwIP knows nothing of it */
                 }
             }
         }
(6)      ETH->IM |= ETH_INT_RX;                        /* re-enable the RX interrupt */
     }
```

(1) The function `low_level_receive()` allocates a pbuf of the right size and reads the received data from the hardware into the pbuf. Obviously, this function is dependent on the Ethernet Controller used.

(2) The lwIP function `ethernetif_input()` completely processes the incoming pbuf.

(3) If `ethernetif_input()` reports an error, the pbuf is explicitly freed to avoid memory leak.

(4) After processing of each incoming packet, the function `eth_read()` attempts also to output any accumulated pbufs. This is because the lwIP processing can take significant time, during which the transmitter can become ready to output the next packet.

(5) A pbuf coming from the queue of outgoing pbufs must be explicitly freed, because the pbuf is now owned by the driver code and the lwIP stack knows nothing about this pbuf.

(6) The function `eth_driver_read()` always re-enables RX interrupts in the Ethernet Controller.

3.6 The eth_driver_write() Function

Listing 6 shows the implementation of the `eth_driver_write()` function, which the `LwIPMgr` active object calls upon reception of the LWIP_TX_READY event.

Listing 6: The eth_driver_write() Ethernet device driver function (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwip_port\netif\eth_driver.c)

```
     void eth_driver_write(void) {
(1)      if ((ETH->TR & MAC_TR_NEWTX) == 0) {                    /* TX fifo empty? */
(2)          struct pbuf *p = PbufQueue_get(&l_txq);
(3)          if (p != NULL) {                          /* pbuf found in the queue? */
(4)              low_level_transmit(p);
(5)              pbuf_free(p);         /* free the pbuf, lwIP knows nothing of it */
             }
         }
     }
```

(1) If the Ethernet Controller's TX FIFO is empty...

(2) A pbuf is obtained from the TX queue.

(3) A pbuf of NULL indicates that the TX queue is empty.

(4) `low_level_transmit()` writes the data from the provided pbuf to the hardware, triggers the transition of the packet, and finally frees the pbuf. Obviously, this function is dependent on the Ethernet Controller used.

(5) A pbuf coming from the queue of outgoing pbufs must be explicitly freed, because the pbuf is now owned by the driver code and the lwIP stack knows nothing about this pbuf.

3.7 The ethernetif_output() Callback Function

Listing 7 shows the implementation of the `ethernetif_output()` function, which the Ethernet device driver registers as the lwIP callback function for output of the pbufs generated internally by the lwIP stack.

Listing 7: The ethernetif_output() Ethernet device driver function (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwip_port\netif\eth_driver.c)

```
(1) static err_t ethernetif_output(struct netif *netif, struct pbuf *p) {
(2)     if (PbufQueue_isEmpty(&l_txq) &&            /* nothing in the TX queue? */
(3)         ((ETH->TR & MAC_TR_NEWTX) == 0))                     /* TX empty? */
        {
(4)         low_level_transmit(p);                  /* send the pbuf right away */
            /* the pbuf will be freed by the lwIP code */
        }
        else {                 /* otherwise post the pbuf to the transmit queue */
(5)         if (PbufQueue_put(&l_txq, p)) { /*could the TX queue take the pbuf? */
(6)             pbuf_ref(p);     /* reference the pbuf to spare it from freeing */
            }
            else {                                      /* no room in the queue */
                /* the pbuf will be freed by the lwIP code */
(7)             return ERR_MEM;
            }
        }
(8)     return ERR_OK;
    }
```

(1) `ethernetif_output()` must match the signature of the lwIP link-output callback function.

(2) If the queue of outgoing pbufs is empty

(3) And the hardware TX FIFO is empty as well

(4) The pbuf is written to the TX FIFO right away.

NOTE: The lwIP code that calls the `ethernetif_output()` callback always frees the pbuf after the callback returns.

---

(5) Otherwise, the pbuf is placed in the queue of outgoing pbufs for transmission at a later time.

(6) If the queue accepted the pbuf, the reference count of the pbuf is incremented to spare it from recycling in the lwIP code. From that time on the pbuf is referenced by the queue of outgoing pbufs and will be freed explicitly by the device driver code (rather than the core lwIP code)

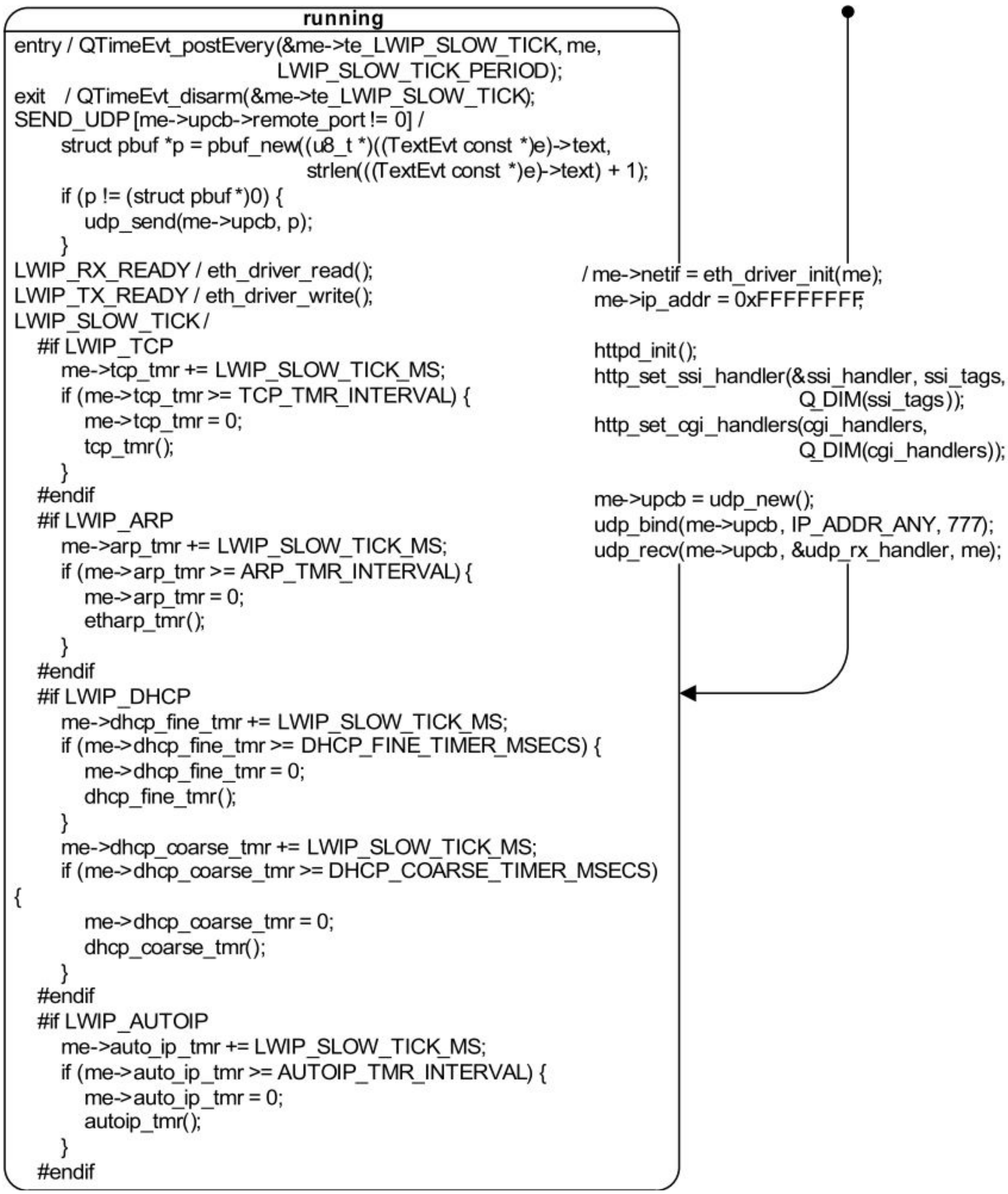
(7) When the pbuf cannot be sent out or stored in the TX queue, the `ethernetif_output()` callback returns the memory error (ERR_MEM) to the lwIP code. The lwIP code frees the pbuf.

(8) When the pbuf is written to the hardware or stored in the queue, the `ethernetif_output()` callback returns success (ERR_OK) to the lwIP code. The lwIP code frees the pbuf, but the pbuf is actually recycled only if it reference counter is 1. The incrementing of the pbuf reference counter in step (6) prevents recycling the pbuf that is held in the queue of outgoing pbufs.

4 LwIPMgr Active Object

The `LwIPMgr` (lwIP-Manager) active object provides **strict encapsulation** of the lwIP code and it the only thread of execution allowed to call any lwIP function or access lwIP data. The other active objects in the system as well as any ISRs can only use lwIP indirectly, by posting events to the `LwIPMgr` active object.

Figure 13: LwIPMgr state machine (<root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwipmgr.c)



The LwIPMgr code is located in the file lwipmgr.c and is independent on the underling kernel you choose (so lwipmgr.c is identical in the cooperative “vanilla” kernel and the preemptive QK versions.)

Figure 13 shows the state machine of the LwIPMgr active object. In this QP-lwIP example, the state machine is trivial (i.e., it consists of only one state “running”), because all state-like behavior is handled inside the lwIP stack. However, in more specialized applications, LwIPMgr can have more interesting state topology, typically added as sub-machine of the “running” state.

NOTE: Obviously, every TCP connection or DHCP processing are state machines, but they are not coded explicitly as state machines in lwIP. Instead, lwIP implementation uses the traditional if-s and else-s to capture the state behavior. You can view lwIP as a set of “orthogonal components” executed from the context of the LwIPMgr active object container.

The most important LwIPMgr’s behavior is processing the events generated from the Ethernet device driver, like LWIP_RX_READY and LWIP_TX_READY. Also, LwIPMgr handles the timeouts for all configured lwIP components, such as TCP, ARP, DHCP, or AUTOIP. Typically, you should have no need to change the handling of any device-driver generated events.

4.1 Launching and Configuring HTTP-Daemon and UDP/IP Applications

The LwIPMgr active object can run various TCP/IP or applications simultaneously, but this of course cannot be programmed generically and requires modifications of the boilerplate LwIPMgr code. The ideal place for launching various applications is the top-most initial transition of the LwIPMgr state machine.

As shown in Figure 13, in the QP-lwIP example application the LwIPMgr active object starts the HTTP-Daemon (web server) and configures the SSI and CGI handlers, which are specific to the web pages served by the web server. Additionally, LwIPMgr sets up a UDP connection and binds it to port 777.

4.2 Implementing Server Side Include (SSI)

As described in Section 2.8.1, the lwIP web server has been extended to support rudimentary **Server Side Include (SSI)** facility [TI-lwIP 08]. The lwIP web server implements SSI by replacing any HTML tag of the form <!--#tag--> with the dynamically generated string that corresponds to that tag. The server scans the served HTML for SSI flags only in files with the extensions .shtm, .shtml, or .ssi.

For example, Listing 8 shows fragments of the ssi_demo.shtm HTML code, which implements the web-page shown in Figure 8. The SSI tags are shown in boldface.

Listing 8: Fragments of the “SSI Example” web page with SSI tags (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\webserver\fs\ssi_demo.shtm)

```
<TABLE summary="cgi example" cellspacing=4 cellpadding=1 border=0
       align="center" valign="middle">
  <TR align="left">
    <TD colspan="2" bgcolor="#ffff66" align="center"><b>SSI Example</b>
    </TD>
  </TR>
  <TR bgColor=#eeeeee><TD>Packets sent:</TD>
    <TD align="right" width="100px"><!--#s_xmit--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Packets retransmitted:</TD>
    <TD align="right"><!--#s_rexmit--></TD></TR>
  <TR bgColor=#eeeeee><TD>Packets received:</TD>
```

```
    <TD align="right"><!--#s_recv--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Packets forwarded:</TD>
    <TD align="right"><!--#s_fw--></TD></TR>
  <TR bgColor=#eeeeee><TD>Packets dropped:</TD>
    <TD align="right"><!--#s_drop--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Checksum errors:</TD>
    <TD align="right"><!--#s_chkerr--></TD></TR>
  <TR bgColor=#eeeeee><TD>Packets with invalid length:</TD>
    <TD align="right"><!--#s_lenerr--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Memory errors:</TD>
    <TD align="right"><!--#s_memerr--></TD></TR>
  <TR bgColor=#eeeeee><TD>Routing errors:</TD>
    <TD align="right"><!--#s_rterr--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Protocol errors:</TD>
    <TD align="right"><!--#s_proerr--></TD></TR>
  <TR bgColor=#eeeeee><TD>Option errors:</TD>
    <TD align="right"><!--#s_opterr--></TD></TR>
  <TR bgColor=#e0e0e0><TD>Miscallaneous errors:</TD>
    <TD align="right"><!--#s_err--></TD></TR>
</TABLE>
```

Listing 9 shows the target-side implementation of SSI in the `LwIPMgr` active object. Each SSI tag is placed in the array `ssi_tags[]` and a callback function `ssi_handler()` is provided for processing these tags. The array of tags and the callback function are registered with the HTTP-Deamon by the call to `http_set_ssi_handler()` (see the initial transition in the state diagram in Figure 13).

Listing 9: Implementing SSI in the target code (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwipmgr.c)

```
 static char const * const ssi_tags[] = {
    "s_xmit",
    "s_rexmit",
    "s_recv",
    "s_fw",
    "s_drop",
    "s_chkerr",
    "s_lenerr",
    "s_memerr",
    "s_rterr",
    "s_proerr",
    "s_opterr",
    "s_err",
};

static int ssi_handler(int iIndex, char *pcInsert, int iInsertLen) {
    struct stats_proto *stats = &lwip_stats.link;
    STAT_COUNTER value;

    switch (iIndex) {
        case 0:                                              /* s_xmit   */
            value = stats->xmit;
            break;
        case 1:                                              /* s_rexmit */
            value = stats->rexmit;
            break;
```

```
            case 2:                                                    /* s_recv   */
                value = stats->recv;
                break;
            case 3:                                                    /* s_fw     */
                value = stats->fw;
                break;
            case 4:                                                    /* s_drop   */
                value = stats->drop;
                break;
            case 5:                                                    /* s_chkerr */
                value = stats->chkerr;
                break;
            case 6:                                                    /* s_lenerr */
                value = stats->lenerr;
                break;
            case 7:                                                    /* s_memerr */
                value = stats->memerr;
                break;
            case 8:                                                    /* s_rterr  */
                value = stats->rterr;
                break;
            case 9:                                                    /* s_proerr */
                value = stats->proterr;
                break;
            case 10:                                                   /* s_opterr */
                value = stats->opterr;
                break;
            case 11:                                                   /* s_err    */
                value = stats->err;
                break;
        }

        return snprintf(pcInsert, MAX_TAG_INSERT_LEN, "%d", value);
    }
```

In this particular case of the SSI implementation, the `http_set_ssi_handler()` returns a string that corresponds to each of the recognized tags. Please note that the returned string cannot exceed the allocated space, which is set by the `MAX_TAG_INSERT_LEN` macro (currently 192 characters).

4.3 Implementing CGI

As described in Section 2.8.2, the lwIP web server has been extended to support rudimentary **Common Gateway Interface (CGI)** facility [TI-lwIP 08]. CGI enables you to send commands with parameters to the embedded target via the lwIP web server. For example, you can place an HTML form on your webpage. When the user submits the form using the GET method, the lwIP web server will recognize a CGI request and will invoke a registered callback function in the target. The lwIP web server will then serve another webpage returned by the CGI callback.

For example, Listing 10 shows fragments of the `cgi_demo.htm` HTML code, which implements the HTML form shown in Figure 9. The CGI method and URI are shown in boldface.

Listing 10: HTML form that generates CGI GET request (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\webserver\fs\cgi_demo.htm)

```
<FORM name="CGI Example" method="GET" action="display.cgi">
  <TABLE summary="cgi example" width="95%" cellspacing=4
                 cellpadding=1 border=0 align="center" valign="middle">
    <TR bgcolor="#e0e0e0" align="left">
      <TD width="150px">Screen text:
      </TD>
      <TD>
          <input type="text" name="text" size="10" maxlength="10">
          <input type="submit" name="submit" value="Submit">
      </TD>
    </TR>
  </TABLE>
</FORM>
```

Listing 11 shows the target-side implementation of CGI in the `LwIPMgr` active object. The array `cgi_handlers[]` provides the mapping between the CGI URIs and the corresponding handler functions. For example, the URI "/display.cgi" (see Listing 10) is handled by the `cgi_display()` handler.

The CGI handler obtains the CGI request parameters as strings (currently up to 32 parameters can be handled). The handler function can execute arbitrary code in the target. For example, the `cgi_display()` handler publishes an event to display the text received in the CGI parameter on the OLED display of the target board.

NOTE: The OLED display is another resource, which should not be shared. In the QP-lwIP example, the OLED display is encapsulated in the `Table` active object. To preserve this encapsulation, the `LwIPMgr` active object sends an event rather than accessing the OLED display directly.

The CGI handler must then return a file name of the web page to be served in response. By returning a NULL pointer, a CGI handler signals to the HTTP-Daemon that the GCI request has not been processed correctly. In this case, HTTP-Daemon will serve the 404-error page.

Listing 11: Implementing CGI in the target code (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwipmgr.c)

```
static tCGI const cgi_handlers[] = {       /* URI to CGI-handler mappings */
    { "/display.cgi", &cgi_display },
};
static char const *cgi_display(int index, int numParams,
                   char const *param[], char const *value[])
{
    int i;
    for (i = 0; i < numParams; ++i) {
        if (strstr(param[i], "text") != (char *)0) {/* param screen found? */
            TextEvt *te = Q_NEW(TextEvt, DISPLAY_CGI_SIG);
            strncpy(te->text, value[i], Q_DIM(te->text));
            QF_publish((QEvent *)te);
            return "/thank_you.htm";
        }
    }
    return (char *)0;/*no URI, HTTPD will send 404 error page to the browser*/
}
```

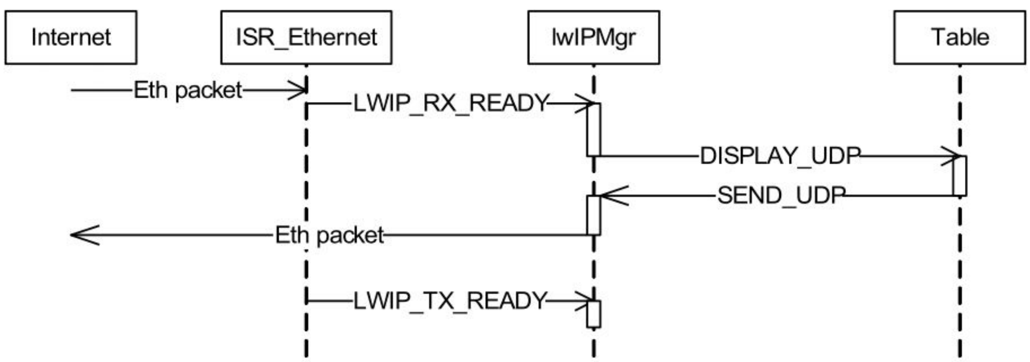

4.4 Implementing UDP

The QP-lwIP example application demonstrates also UDP communication to and from the embedded target. As shown in Figure 13, the `LwIPMgr` active object creates a UDP Protocol Control Block (PCB) in the top-most initial transition. Also, the UDP PCB is bound to the port 777 and the UDP receive handler is registered for the PCB:

```
me->upcb = udp_new();
udp_bind(me->upcb, IP_ADDR_ANY, 777);            /* use port 777 for UDP */
udp_recv(me->upcb, &udp_rx_handler, me);
```

The sequence diagram in Figure 14 shows how the QP-lwIP example application handles UDP. An Ethernet packet with the UDP payload triggers the Ethernet interrupt, which posts the LWIP_RX_READY event to the `LwIPMgr` active object. The LwIPMgr calls `eth_read()` driver function which recognizes the UDP payload and calls the registered UDP receive callback. The UDP receive callback in this application assumes that the payload of the packet is the text to be displayed on the OLED screen of the target board.

Figure 14: Sequence diagram of the UDP processing in the QP-lwIP example



To show sending UDP packets, the `Table` active object posts the SEND_UDP event to the `LwIPMgr` active object. The parameters of the SEND_UDP event carry the whole payload of the packet. Upon reception of this event, the `LwIPMgr` allocates a pbuf, copies the payload from the event parameters to the pbuf, and sends the UDP packet.

4.4.1 Receiving UDP Packets

The UDP receive handler is defined in Listing 12.

Listing 12: UDP handler example (file <root>\qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwipmgr.c)

```
    static void udp_rx_handler(void *arg, struct udp_pcb *upcb,
                               struct pbuf *p, struct ip_addr *addr, u16_t port)
    {
        TextEvt *te = Q_NEW(TextEvt, DISPLAY_UDP_SIG);
(1)     strncpy(te->text, (char *)p->payload, Q_DIM(te->text));
        QF_publish((QEvent *)te);
(2)     udp_connect(upcb, addr, port);          /* connect to the remote host */
(3)     pbuf_free(p);                                    /* don't leak the pbuf! */
    }
```

(1) This simplistic UDP handler expects that the arriving UDP packet contains text to be displayed on the OLED screen of the target board. You can see how the payload is accessed and copied into the event.

(2) The UDP handler also connects the UDP PCB to the remote host that sent has sent the UDP packet to the target. This allows the UDP connection to respond to the sender.

(3) The UDP must explicitly free the received pbuf, because the lwIP code assumes that now the UDP handler owns the pbuf.

4.4.2 Sending UDP Packets

The `LwIPMgr` active object reacts to the SEND_UDP event by preparing and sending a UDP packet, as shown in Listing 13.

Listing 13: Sending a UDP packet from the LwIPMgr active object

```
        . . .
        case SEND_UDP_SIG: {
(1)         if (me->upcb->remote_port != (uint16_t)0) {
(2)             struct pbuf *p = pbuf_new((u8_t *)((TextEvt const *)e)->text,
                                  strlen(((TextEvt const *)e)->text) + 1);
(3)             if (p != (struct pbuf *)0) {
(4)                 udp_send(me->upcb, p);
                }
            }
            return Q_HANDLED();
        }
        . . .
```

(1) If the remote port of the connection is set (i.e., the recipient of the packet is known)

(2) A new pbuf of the desired length is allocated and the payload is filled with the desired data.

> **NOTE**: The function `pbuf_new()` has been added by Quantum Leaps. It allocates a transport-layer pbuf and copies the provided data buffer 'data' of length 'len' bytes into the payload(s) of the pbuf. The function takes care of splitting the data into successive pbuf payloads, if necessary.

(3) If the new packet is created correctly…

(4) The packet is sent via the standard lwIP call `udp_send()`.

4.5 Assigning Priority to the LwIPMgr Active Object

As described in Section 1.4, the QP-lwIP integration allows you to use the lwIP TCP/IP stack inside hard-real-time applications. To achieve truly deterministic real-time response of high-priority active objects, you need to use the **preemptive** QK kernel (see example code in `<root>\qpc\examples\arm-cm\`**`qk`**`\iar\lwip_ek-lm3s6965\`) and you also need to prioritize the `LwIPMgr` active object **lower** than any hard-real-time active object (assign lower QP priority value to `LwIPMgr`).

> **NOTE**: Under the preemptive QK kernel you must be extremely careful about sharing any resources among tasks. Please refer to Chapter 10 in [PSiCC2].

5 Configuring and Customizing QP-lwIP

The lwIP stack is configured and customized by means of the `lwipopts.h` header file, which is collocated with the QP application (see Listing 1). As mentioned earlier in Section 1.4, QP works with the standard, unmodified lwIP source code, so the standard lwIP documentation fully applies to the lwIP configuration in the context of QP. In fact, the best available documentation for configuring lwIP are the comments embedded in the file `lwipopts.h` as well as in `lwip-1.4.0.rc2\include\lwip\opt.h` header file.

The `lwipopts.h` included with the QP-lwIP example application contains a few new sections, which pertain to configuring the static IP address, configuration of the lwIP Ethernet driver, and the HTTP-Daemon.

From the standard lwIP configuration options, perhaps the most important are the options that control the lwIP integration with an external operating system. For QP, these options should be set as follows:

Listing 14: Fragments of lwIP configuration
(file `qpc\examples\arm-cm\qk\iar\lwip_ek-lm3s6965\lwipopts.h`)

```
. . .
//*****************************************************************************
//
// ---------- Platform specific locking ----------
//
//*****************************************************************************
//#define SYS_LIGHTWEIGHT_PROT            0
#define NO_SYS                          1           // default is 0
//#define MEMCPY(dst,src,len)             memcpy(dst,src,len)
//#define SMEMCPY(dst,src,len)            memcpy(dst,src,len)
. . .
```

6 Related Documents and References

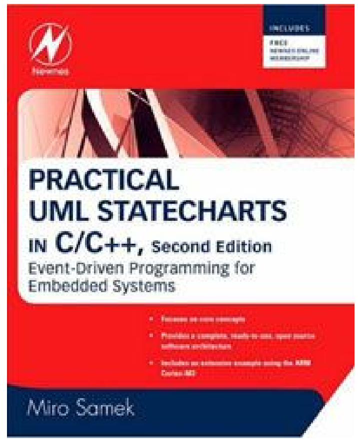
Document	Location
[PSiCC2] "Practical UML Statecharts in C/C++, Second Edition", Miro Samek, Newnes, 2008	Available from most online book retailers, such as Amazon.com. See also: http://www.state-machine.com/psicc2.htm
[QP/C 08] "QP/C Reference Manual", Quantum Leaps, LLC, 2010	http://www.state-machine.com/doxygen/qpc/
[QP/C++ 08] "QP/C++ Reference Manual", Quantum Leaps, LLC, 2010	http://www.state-machine.com/doxygen/qpcpp/
[QL AN-Directory 07] "Application Note: QP Directory Structure", Quantum Leaps, LLC, 2007	http://www.state-machine.com/doc/AN_QP_Directory_Structure.pdf
[QL AN-DPP 08] "Application Note: Dining Philosopher Problem Application", Quantum Leaps, LLC, 2008	http://www.state-machine.com/doc/AN_DPP.pdf
[LwIP 1.3.0] "LwIP 1.3.0 Documentation", 2008	Document `LwIP-STABLE-1.3.0.pdf` (included in the `doc\` directory)
[LwIP-OS] "Using lwIP with or without an operating system", Savannah, 2004	Document `Using_lwIP_with_or_without_OS.pdf` (included in the `doc\` directory); also available from http://savannah.nongnu.org/projects/lwIP/
[Dunkels 01] "Design and Implementation of the lwIP TCP/IP Stack", Adam Dunkels, 2001	Document `Design_and_Implementation_of_lwIP.pdf` (included in the `doc\` directory)
[Dunkels 07] "Programming Memory-Constrained Networked Embedded Systems", Adam Dunkels Ph.D. Thesis, 2007	Document `Dunkels_PhD_Thesis_07.pdf` (included in the `doc\` directory)
[LM3S6965 08] "LM3S6965 Microcontroller Data Sheet", Luminary Micro, 2008.	Luminary Micro literature number DS-LM3S6965-4660 available from http://www.luminarymicro.com/products/lm3s6965.html
[TI-lwIP 09] Texas Instruments Enet-lwIP example, Texas Instruments, 2009.	Example code included in the IAR EWARM 5.40 distribution in the directory `<IAR>\arm\examples\TexasInstruments\Stellaris\ boards\ek-lm3s6965_revc\enet_lwip`

7 Contact Information

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

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

Email: info@state-machine.com
Web : http://www.state-machine.com



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

lwIP homepage:
http://savannah.nongnu.org/projects/lwip