## Contents

Preface .................................................................................................................................................. 7

1 Getting Started ................................................................................................................................ 9
   1.1 Introduction ..................................................................................................................................... 10
   1.2 What This Document Covers ........................................................................................................ 10

2 Installation and Test of the TCP/IP Stack .......................................................................................... 11
   2.1 Setting Up the TCP/IP Stack ............................................................................................................. 12
      2.1.1 TCP/IP Stack Contents ............................................................................................................... 12
      2.1.2 TCP/IP Stack Software CD ........................................................................................................ 12
   2.2 Using a SMSC LAN91C111 Based Ethernet Adapter ......................................................................... 12
      2.2.1 Using the LogicIO ETH6000F (Macronix) Ethernet Adapter ......................................................... 12
      2.2.2 Using the D.SignT DSK-91C111 (SMSC) Ethernet Adapter ......................................................... 13
      2.2.3 Using the Spectrum Digital DSK Comm (SMSC) Ethernet Adapter .............................................. 13
   2.3 Ethernet Hardware Testing .............................................................................................................. 13
      2.3.1 Ethernet Hardware Confidence Test ............................................................................................. 13
      2.3.2 Ethernet EEPROM Utility .......................................................................................................... 14
   2.4 Rebuilding TCP/IP Stack Libraries .................................................................................................. 14

3 Example Applications .......................................................................................................................... 17
   3.1 The Network Client Example Application ......................................................................................... 18
      3.1.1 Introduction ............................................................................................................................... 18
      3.1.2 Building the Application ............................................................................................................. 18
      3.1.3 Loading the Application .............................................................................................................. 18
      3.1.4 Testing the Application ............................................................................................................... 18
   3.2 The Network Configuration Example Application .............................................................................. 19
      3.2.1 Introduction ............................................................................................................................... 19
      3.2.2 Building the Application ............................................................................................................. 19
      3.2.3 Loading the Application .............................................................................................................. 19
      3.2.4 Configuring the Application ....................................................................................................... 19
      3.2.5 Testing the Application ............................................................................................................... 20
   3.3 The Serial Client Example Application ............................................................................................ 21
      3.3.1 Introduction ............................................................................................................................... 21
      3.3.2 Setting Up the Network ............................................................................................................... 21
      3.3.3 Building the Application ............................................................................................................. 21
      3.3.4 Loading the Application .............................................................................................................. 21
      3.3.5 Testing the Application ............................................................................................................... 21
   3.4 The Serial Router Example Application ............................................................................................ 22
      3.4.1 Introduction ............................................................................................................................... 22
      3.4.2 Setting Up the Network ............................................................................................................... 22
      3.4.3 Building the Application ............................................................................................................. 23
      3.4.4 Loading the Application .............................................................................................................. 23
      3.4.5 Testing the Application ............................................................................................................... 23

4 TCP/IP Stack Library Overview .......................................................................................................... 25
   4.1 Stack Library Design ....................................................................................................................... 26
      4.1.1 Design Philosophy ...................................................................................................................... 26
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1.2</td>
<td>Organization</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Programming API</td>
<td>27</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Operating System Abstraction</td>
<td>27</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Sockets and Stream IO API</td>
<td>27</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Initialization and Configuration</td>
<td>27</td>
</tr>
<tr>
<td>4.2.4</td>
<td>NETTOOL Support Functions</td>
<td>28</td>
</tr>
<tr>
<td>4.2.5</td>
<td>NETTOOL Services</td>
<td>28</td>
</tr>
<tr>
<td>4.2.6</td>
<td>Internal Stack API</td>
<td>28</td>
</tr>
<tr>
<td>4.2.7</td>
<td>Hardware Adaptation Layer API</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>TCP/IP Stack Software Directory</td>
<td>28</td>
</tr>
<tr>
<td>4.3.1</td>
<td>TCP/IP Stack Documentation</td>
<td>28</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Ethernet MAC Hardware Utilities</td>
<td>28</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Example Programs</td>
<td>29</td>
</tr>
<tr>
<td>4.3.4</td>
<td>TCP/IP Stack Include File Directory</td>
<td>29</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Linkable Libraries Directory</td>
<td>30</td>
</tr>
<tr>
<td>4.3.6</td>
<td>Library Source Directory</td>
<td>31</td>
</tr>
<tr>
<td>4.3.7</td>
<td>Windows Test Utilities</td>
<td>31</td>
</tr>
<tr>
<td>5</td>
<td>Network Application Development</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Using Code Composer Studio</td>
<td>34</td>
</tr>
<tr>
<td>5.1.1</td>
<td>Required Configuration Entries</td>
<td>34</td>
</tr>
<tr>
<td>5.1.2</td>
<td>Include Files and Library Files</td>
<td>34</td>
</tr>
<tr>
<td>5.1.3</td>
<td>CCS Project Link Order</td>
<td>34</td>
</tr>
<tr>
<td>5.1.4</td>
<td>TCP/IP Stack Memory Sections</td>
<td>34</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Using Cache</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Developing Socket Applications with DSP/BIOS</td>
<td>35</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Default Environment API Restrictions</td>
<td>35</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Creating a Task</td>
<td>36</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Memory Allocation</td>
<td>37</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Example Code</td>
<td>37</td>
</tr>
<tr>
<td>5.3</td>
<td>TCP/IP Stack Initialization and Configuration</td>
<td>38</td>
</tr>
<tr>
<td>5.3.1</td>
<td>TCP/IP Stack Initialization Using NETCTRL</td>
<td>38</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Adding Standard Services</td>
<td>42</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Initialization Examples</td>
<td>43</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Controlling TCP/IP Stack and OS Options via the Configuration</td>
<td>47</td>
</tr>
<tr>
<td>5.3.5</td>
<td>Saving and Loading a Configuration</td>
<td>47</td>
</tr>
<tr>
<td>5.4</td>
<td>Application Debug and Troubleshooting</td>
<td>49</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Most Common Problems</td>
<td>49</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Controlling Debug Messages</td>
<td>50</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Interpreting Debug Messages</td>
<td>50</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Memory Corruption</td>
<td>51</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Program Lockups</td>
<td>51</td>
</tr>
<tr>
<td>5.4.6</td>
<td>Memory Management Reports</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>Network Control Functions</td>
<td>55</td>
</tr>
<tr>
<td>6.1</td>
<td>Introduction to NETCTRL Source</td>
<td>56</td>
</tr>
<tr>
<td>6.1.1</td>
<td>History</td>
<td>56</td>
</tr>
<tr>
<td>6.1.2</td>
<td>NETCTRL Source Files</td>
<td>56</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Main Functions</td>
<td>56</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Additional Functions</td>
<td>56</td>
</tr>
</tbody>
</table>
6.1.5 Booting and Scheduling ................................................................. 57
6.2 NETCTRL Scheduler ................................................................. 57
  6.2.1 Scheduler Overview ............................................................. 57
  6.2.2 Scheduling Options ............................................................. 58
  6.2.3 Scheduler Thread Priority ...................................................... 59
  6.2.4 Tracking Events with STKEVENT ............................................. 59
  6.2.5 Scheduler Loop Source Code .................................................. 59
6.3 Disabling “On-Demand” Services .................................................. 62

7 OS Adaptation Layer : OS.LIB and MiniPrintf.LIB .......................... 63
  7.1 Introduction to OS Source .......................................................... 64
    7.1.1 History .............................................................................. 64
    7.1.2 Source Files ...................................................................... 64
  7.2 Task Thread Abstraction - TASK.C ............................................. 64
    7.2.1 TaskSetEnv() and TaskGetEnv() ......................................... 65
    7.2.2 TaskCreate(), TaskExit(), and TaskDestroy() ......................... 65
    7.2.3 Choosing the IIEnter()/IIExit() Exclusion Method .................... 65
  7.3 Packer Buffer Manager - PBM.C ............................................... 66
    7.3.1 Packet Buffer Pool .............................................................. 66
    7.3.2 Packet Buffer Allocation Method .......................................... 66
    7.3.3 Referenced Route Handles ................................................... 67
  7.4 Memory Allocation System - MEM.C ........................................ 67
    7.4.1 mmBulkAllocSeg – Set the DSP/BIOS Heap Segment for Bulk Allocation Functions .............. 68
  7.5 Embedded File System - EFS.C .................................................. 68
  7.6 General OS Support - OSSYS.C ............................................... 68
  7.7 Print Functions - MINIPRINTF.C ................................................ 68
List of Figures

4-1 Stack Control Flow ................................................................. 26
About This Manual

This guide introduces the TMS320C6000 TCP/IP Stack and provides a brief overview about the purpose and construction of the TCP/IP stack, along with hardware and software environment specifics in the context of TCP/IP Stack deployment.

How to Use This Manual

The information presented in this document is divided into the following chapters:

- **Chapter 1 - Introduction**, introduces the stack and developing network applications.
- **Chapter 2 - Installation and Test of the TCP/IP Stack**, provides help with getting started with TCP/IP networking hardware and software.
- **Chapter 3 - Example Applications**, provides examples that are good for platform test and demonstration, and also serve as a good starting point for developing your own network applications.
- **Chapter 4 - TCP/IP Stack Library Overview**, describes the TCP/IP stack library architecture, features, and functions of the stack software package.
- **Chapter 5 - Network Application Development**, describes the TCP/IP stack software, and how to start developing network applications now.
- **Chapter 6 - Network Control Functions**, describes the internal workings of the network control layer (NETCTRL).
- **Chapter 7 - OS Adaptation Layer**, describes the OS adaptation layer, which controls how the TCP/IP stack uses DSP/BIOS resources. This includes tasks, semaphores, memory and printing. Anything that is related to OS can be adjusted here.

Notational Conventions

This document uses the following conventions:

- Program listings, program examples, and interactive displays are shown in a special typeface.
- In syntax descriptions, the function or macro appears in a bold typeface and the parameters appear in plainface within parentheses. Portions of a syntax that are in bold should be entered as shown; portions of a syntax that are within parentheses describe the type of information that should be entered.
- Macro names are written in uppercase text; function names are written in lowercase.
- The TMS320C62x™ DSP is also referred to in this reference guide as the C62x™ DSP.

Related Documentation From Texas Instruments

The following books describe the TMS320C6x™ devices and related support tools. To obtain a copy of any of these TI documents, call the Texas Instruments Literature Response Center at (800) 477–8924. When ordering, please identify the book by its title and literature number. Many of these documents can be found on the Internet at [http://www.ti.com](http://www.ti.com).

**SPRU189** — TMS320C6000 DSP CPU and Instruction Set Reference Guide. Describes the CPU architecture, pipeline, instruction set, and interrupts for the TMS320C6000™ digital signal processors (DSPs).

**SPRU190** — TMS320C6000 DSP Peripherals Overview Reference Guide. Provides an overview and briefly describes the peripherals available on the TMS320C6000™ family of digital signal processors (DSPs).


**SPRU197 — TMS320C6000 Technical Brief.** Provides an introduction to the TMS320C62x™ and TMS320C67x™ digital signal processors (DSPs) of the TMS320C6000™ DSP family. Describes the CPU architecture, peripherals, development tools and third-party support for the C62x™ and C67x™ DSPs.

**SPRU198 — TMS320C6000 Programmer’s Guide.** Reference for programming the TMS320C6000™ digital signal processors (DSPs). Before you use this manual, you should install your code generation and debugging tools. Includes a brief description of the C6000 DSP architecture and code development flow, includes C code examples and discusses optimization methods for the C code, describes the structure of assembly code and includes examples and discusses optimizations for the assembly code, and describes programming considerations for the C64x™ DSP.

**SPRU509 — TMS320C6000 Code Composer Studio Development Tools v3.1 Getting Started Guide.**
Introduces some of the basic features and functionalities in Code Composer Studio™ to enable you to create and build simple projects.

**SPRU524 — TMS320C6000 TCP/IP Stack Programmer’s Reference Guide.** Describes the various API functions provided by the stack libraries, including the low level hardware APIs.

**SPRU030 — TMS320C6000 TCP/IP Stack Software Platform Porting Kit User’s Guide** Describes the source code and libraries necessary for porting the TCP/IP Stack Software to alternate hardware platforms.

**Trademarks**

TMS320C62x, C62x, TMS320C6x, TMS320C6000, TMS320C67x, C67x, C6x, Code Composer Studio, DSP/BIOS are trademarks of Texas Instruments.
This chapter introduces the TMS320C6000 TCP/IP Stack to the user by providing a brief overview of the purpose and construction of the TCP/IP Stack, along with hardware and software environment specifics in the context of TCP/IP Stack deployment.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Introduction</td>
<td>10</td>
</tr>
<tr>
<td>1.2 What This Document Covers</td>
<td>10</td>
</tr>
</tbody>
</table>
1.1 Introduction

The TMS320C6000™ TCP/IP Stack has been designed as a platform for development and demonstration of network enabled applications on the TMS320C6000 DSP family.

The TCP/IP Stack includes demonstration software showcasing C6000 DSP capabilities across a range of network enabled applications. In addition, the TCP/IP Stack serves as a rapid prototyping platform for the development of network and packet processing applications, or to add network connectivity to existing DSP applications for communications, configuration, and control.

Using the software and hardware components provided with the TCP/IP Stack, developers can quickly move from development concepts to working implementations attached to the network.

1.2 What This Document Covers

This TCP/IP Stack User's Guide serves as an introduction to both the TMS320C6000 TCP/IP Stack and to developing network applications. The document covers programming as it applies to the TMS320C6000 programming environment, including Code Composer Studio. It is not intended as an API reference. This manual also provides necessary information regarding how to effectively install, build, and use the TCP/IP Stack in user systems and applications.
This chapter provides information about the installation, setup, and testing of the TCP/IP Stack software. It is a good place to begin when getting started with the TCP/IP Stack software and with the C6000-based networking hardware platforms.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Setting Up the TCP/IP Stack</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Using a SMSC LAN91C111 Based Ethernet Adapter</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Ethernet Hardware Testing</td>
<td>13</td>
</tr>
<tr>
<td>2.4 Rebuilding TCP/IP Stack Libraries</td>
<td>14</td>
</tr>
</tbody>
</table>
2.1 Setting Up the TCP/IP Stack

This section defines the TCP/IP stack and provides instructions on its setup and use.

2.1.1 TCP/IP Stack Contents

The TCP/IP Stack is a networking stack that operates on top of DSP/BIOS RTOS. The stack can be ported to any TMS320C6000 based hardware through the Porting Kit. The base TCP/IP Stack includes support for the following hardware platforms:

- TMS320C6711 DSK
- TMS320C6713 DSK
- TMS320C6416 DSK
- TMS320DM642 EVM
- TMS320C6455 DSK/EVM

The C6455 and DM642 DSPs have Ethernet on chip; for the other platforms listed above, one of the following Ethernet daughtercards may be used:

- LogicIO ETH6000F 10/100 Ethernet Adapter (based on Macronix MX98728EC)
- ATEME NDK-NG 3034E1 Ethernet Adapter (based on SMSC LAN91C111) *

In addition to this hardware, a compatible emulator connection to CCS is required. This varies by platform. The TCP/IP Stack software CD contains support files for any combination of the above hardware and Ethernet devices.

* Either the Spectrum Digital DSK Comm Adapter or the D.SignT DSK-91C111 Ethernet Adapter can be used as a substitute for the ATEME NDK-NG 3034E1. See Section 2.2 for more information.

2.1.2 TCP/IP Stack Software CD

The TCP/IP Stack software image is contained on the TCP/IP Stack CD. Additional instructions are available on this CD. To view the installation instructions, insert the CD into a Windows based PC. If the AUTORUN feature is not enabled on your system, run the program NDKSTART.EXE located in the software root directory. When the TCP/IP Stack Startup dialog appears, click on the "Getting Started" button for more installation information, or click on "Software Install" to jump right to the installation program.

2.2 Using a SMSC LAN91C111 Based Ethernet Adapter

On networking platforms designed around a baseboard (like the DSKs), there is an option to use either the LogicIO (Macronix) or a SMSC LAN91C111 Ethernet adapter daughtercard. In order to decrease the number of project files in the stack, the TCP/IP Stack is shipped with all example programs and utilities built for use with the SMSC Ethernet adapter. When using the LogicIO Ethernet daughtercard, the project OUT files must be re-built using the correct MX based driver. The process varies slightly according to the Ethernet hardware used.

2.2.1 Using the LogicIO ETH6000F (Macronix) Ethernet Adapter

The Macronix Ethernet libraries for the TCP/IP Stack are written for use with the LogicIO Ethernet Adapter. Therefore, when using the LogicIO hardware, the project OUT files just need to be re-linked using the correct MX based driver. No other project or hardware modifications are required.

For example, to re-link the Ethernet confidence test running on the 6711 DSK, you would perform the following steps:

1. Open the project file "emactest.pjt" in the directory "C:\<Stack Install Directory>\ndk\emacutil\emactest\ds6k6711".
2. Under the Libraries section of the project files, remove the library "hal_eth_smccf.lib"
3. Use the "Add Files to Project" option under "Project" to add the library file "hal_eth_mxf.lib" from the directory "C:\Stack Install Directory\NDK\lib\hal\dsk6711".

4. Rebuild the project using the new library

The resulting OUT file is for use with the LogicIO/Macronix Ethernet adapter. All example program projects supplied for the DSK boards are compatible with the LogicIO/Macronix daughtercard and can be re-linked using this procedure.

2.2.2 Using the D.SignT DSK-91C111 (SMSC) Ethernet Adapter

The D.SignT DSK 91C111 Ethernet Adapter is nearly hardware compatible with its ATEM E counterpart. All that needs to be done is to swap the interrupt polarity from its default setting to that of the ATEM E NDK-NG 3034E1. This is done by moving the JPINTPOL jumper on the D.SignT board from the "a" position to the "b" position. After this modification has been performed, the D.SignT board can be treated the same as the ATEM E NDK-NG 3034E1.

2.2.3 Using the Spectrum Digital DSK Comm (SMSC) Ethernet Adapter

The Spectrum Digital DSK Comm Adapter is based on the same SMSC Ethernet MAC as its ATEM E counterpart. Therefore, it can be used with the same SMSC Ethernet libraries shipped with the TCP/IP Stack. However, the DSK Comm Adapter contains configuration registers that must be programmed to enable the SMSC Ethernet MAC, and invert the SMSC interrupt signal. Therefore, an extra step is required to initialize the hardware.

For example, to re-build the Ethernet confidence test running on the 6711 DSK, you would perform the following steps:

1. Open the project file "emactest.pjt" in the directory "C:\Stack Install Directory\NDK\emactest\dsk6711".
2. Open the project TCF file, and under System/Global Settings, replace the User Init Function "_emactest_init" with "_sd_dskcomm_init".
3. Use the "Add Files to Project" option under "Project" to add the source file "sd_dskcomm_init.c" from the directory "C:\Stack Install Directory\NDK\example\tools\common\sd_dskcomm".
4. Rebuild the project using the new hardware initialization routine.

2.3 Ethernet Hardware Testing

After installing the TCP/IP Stack software image, the operation of your hardware plus the Ethernet card should be verified. Note that on the TMS320DM642 EVM, on the TMS320C6455 DSK, and TMS320C6455 EVM boards, this step is not necessary as the Ethernet test is part of your hardware diagnostics.

2.3.1 Ethernet Hardware Confidence Test

For those platforms that require the addition of an Ethernet daughtercard, an EMACTEST is provided to test the functionality of the daughtercard, and the communications between the daughtercard and the platform.

The EMACTEST test application will test the Ethernet board's ability to communicate with the base platform. This is the first test that should be run on any networking platform that includes a daughtercard. The test project is located in the 'NDK\emactest\dsk6711" directory. Source code is provided and the test can be rebuilt via Code Composer Studio. Code Composer is also used to load the final executable file.

Before running this test, disconnect any Ethernet cable from the Ethernet board. The test will use an internal loopback feature of the card to send and receive packets. Once the platform is ready, load and execute the test by performing the following steps:

1. Run Code Composer, and verify it is able to reset the DSP. (The DSP reset is necessary to run the test on some platforms.)
2. Use the "Load Program ..." command from the File menu to load the executable file "emactest.out" from the project directory that corresponds to your hardware.

3. After loading, execute the test by selecting the "Run" command from the Debug menu. As the test runs, text messages will be printed to the debugger output window. There will be natural pauses after some of the text messages.

4. Verify the MAC address printed out by the test application matches the MAC address printed on the label affixed to the Ethernet board.

5. After the MAC test, the board will be placed in internal loopback mode, and the application will echo packets to itself. After several trials, the test should complete without error.

If this test fails, the platform may be unstable, and should not be used for development until the cause is discovered.

2.3.2 Ethernet EEPROM Utility

An EEPROM utility is included in the rare event that the MAC address programmed on the LogicIO or ATEME Ethernet daughtercards does not match the MAC address supplied on the affixed label, or needs to be altered for some other reason.

The test utility programs are located in the \NDK\emacutil\eeprom directory. Source code is provided with each and can be rebuilt with Code Composer Studio. Code Composer is also used to load the executable file EEPROM.OUT.

Once run, this test will print out the MAC address that is currently programmed on the device. The user will then be prompted as to if the address is correct. If user responds with a negative ("n"), the program will prompt for a new MAC address to be programmed. The new address should be entered as:

xx-xx-xx-xx-xx-xx

where "xx" represents a two digit HEX value. Note that the first 2 digit hex number must be even for the address to be legal.

Once the new address is entered, the program will attempt to write the address to EEPROM and then reset the device. The utility will print out an error code if it was unable to program the EEPROM.

2.4 Rebuilding TCP/IP Stack Libraries

The TCP/IP Stack root directory contains a batch file used to build the three user serviceable stack libraries: OS.LIB, MiniPrint.LIB and NETCTRL.LIB. This batch file is called MAKELIB.BAT.

Before using MAKELIB from a command prompt, the CCS batch file DOSRUN_BIOS.BAT must be run from the root TCP/IP Stack install directory in order to setup the proper environment for running the TI code generation tools from a command prompt.

The form of the MAKELIB command is:

makelib [platform] [library] (noclean)

In this command, the platform and library names are required.

The platform argument determines the CPU environment for which the library is built. The value of platform can be any of the following:

- c62  TMS320C6200
- c64  TMS320C6400
- c64+ TMS320C64Plus
The value of `library` determines what device library to build. The value of `library` can be any of the following:

- **os**: OS Adaptation Layer
- **netctrl**: Network Control layer
- **miniPrintf**: Small Printf Functions

The final parameter “noclean” can be added to the command line to suppress cleaning old object files from the target directory. This is only useful when rebuilding the same driver for the same platform.
This section describes the main example applications included with the TCP/IP Stack software release. The example applications are designed to provide a small sample of potential applications that can be developed with the TCP/IP Stack.

These sample applications can be run as is for a quick demonstration, but the user will benefit most by using these samples as sample source code in developing new applications. For this, a working knowledge of how the Code Composer Studio environment interacts with the TCP/IP Stack is helpful. Chapter 5 of this User’s Guide is dedicated to the development of networking applications using Code Composer.

---

**Note:** Some of the example applications described in this section require a network with support for DHCP. If DHCP is not available, only the Configuration example can be run as-is. The remaining examples can be rebuilt to use a fixed IP configuration using Code Composer Studio. See Chapter 5 for details on network application initialization.

---

**Note:** On some platforms, it is necessary to reset the DSP before loading an OUT file. If the example file fails to initialize properly, it can be stopped or sent off into the weeds. This is caused by cache and interrupts being in non-default state when loading.

---

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 The Network Client Example Application</td>
<td>18</td>
</tr>
<tr>
<td>3.2 The Network Configuration Example Application</td>
<td>19</td>
</tr>
<tr>
<td>3.3 The Serial Client Example Application</td>
<td>21</td>
</tr>
<tr>
<td>3.4 The Serial Router Example Application</td>
<td>22</td>
</tr>
</tbody>
</table>
3.1 The Network Client Example Application

3.1.1 Introduction
The client example is the most important of all the example programs since it includes the most components of an actual network application. The client example can use either DHCP or a statically configured IP address. It will launch a console application accessible via Telnet, an HTTP server with a couple of example WEB pages, plus several data servers that can be tested by running client test applications on a Windows PC.

3.1.2 Building the Application
The client example is located in the EXAMPLE\NETWORK\CLIENT directory off the TCP/IP Stack root. The application can be rebuilt directly from its project file using Code Composer Studio™.

3.1.3 Loading the Application
The application is loaded and executed via Code Composer Studio. It is a good idea to reset the board before loading Code Composer, but this should not be required. The application will print out status messages via Code Composer’s standard IO output window (Stdout).

On a successful execution, one of the status lines printed by the application will display the client's IP address (either through DHCP or static configuration). Once this address is displayed, the DSP will respond to requests made to its IP address. When using DHCP, it is possible that the application will be unable to obtain an IP address from a DHCP server. In this case, the application will eventually print a DHCP status message with the "fault" condition. Note that all the messages are generated by the main client module in CLIENT.C.

3.1.4 Testing the Application
Once the application is executing and has printed out its IP address, several tests can be performed.

3.1.4.1 HTTP Server
To see the HTTP server in action, run an Internet browser, and point it to the IP address displayed by the application. If the client application's IP address is "196.12.1.14", the URL would be:

http://196.12.1.14

Be sure and disable any proxy settings on the browser if your network is behind a firewall.

The browser will display a small WEB page describing the example application. There are server status screens that can be accessed off this page. The source code used to generate these pages is further described in the HTTP appendix of the TCP/IP Stack Programmer's Reference Guide.

3.1.4.2 Telnet Server
The client example application also includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application's IP address. Note that the console program will timeout and disconnect after a period of inactivity.

To get a list of console commands, type "help" or simply "?". A list of console commands will be printed to the telnet terminal. The console program is important as a programming demonstration as much as a run time demonstration. There are many functions in the console program that display or test features particular to the TCP/IP Stack. When an application developer wants to use these features in their application, the console example source code can be very useful as a guide.
3.1.4.3 Data Servers

To try out the data servers, use the Windows test applications found in the WINAPPS directory off the TCP/IP Stack root. The applications are command line driven and require a target IP address. For example, type:

```
send 196.12.1.14
```

to start the data receiver. This will request data from the server running on the DSP. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
recv 196.12.1.14 100
```

starts the data send test (receive from the DSP’s point of view) with a display update interval of 100 iterations.

```
echoc 196.12.1.14 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP.

All the Windows test clients will run until a key is pressed, or Control-C in the event of an error (e.g., trying to connect to a bad IP address).

3.2 The Network Configuration Example Application

3.2.1 Introduction

The Configuration Demo (CFGDEMO) example illustrates how the stack running in an embedded environment can be easily configured by the end user without relying on DHCP. The demo boots up the DSP in an idle state with no IP address. A temporary IP address is assigned by the user, and then an HTTP client browser is used to complete the configuration.

3.2.2 Building the Application

The client example is located in the `EXAMPLE\NETWORK\CFGDEMO` directory off the TCP/IP Stack root. The application can be rebuilt directly from its project file using Code Composer Studio.

3.2.3 Loading the Application

The application is loaded and executed via Code Composer Studio. The application will print out status messages via Code Composer's standard I/O output window (Stdout).

On a successful execution, one of the status lines printed by the application will display "GetIP Ready". This is an indication that the DSP board is ready to have an IP address assigned by the user. Note that all the messages are generated by the main module in `CFGDEMO.C`.

3.2.4 Configuring the Application

Once the application is executing and has printed out its "GetIP Ready" message, it is ready for configuration.
3.2.4.1 Setting the Initial IP Address

The first step in configuring the device is to assign it a temporary (or permanent) IP address. The CFGDEMO application uses the ICMP ping message to initially detect its IP address.

Once a free IP address is chosen by the user (say "192.63.10.5"), the user can assign the IP address to the DSP by using the ping command from another machine. Note that the DSK will not reply to ARP requests when not configured, therefore, the MAC address for the chosen IP must be manually entered by the user.

For those devices requiring a daughtercard, the MAC address of the DSK is usually found on the white label affixed to the Ethernet daughtercard. For example, if the MAC address were "08-00-28-32-08-26", to assign this MAC address to the selected IP address, on a Windows command line, type:

```
arp -s 192.63.10.5 08-00-28-32-08-26
```

Next, to assign the IP address to the CFGDEMO application on the DSK, type

```
ping 192.63.10.5
```

The DSK board should start replying to the ping command. Since the demo application prints some additional status messages to Code Composer, it may miss a ping request during this time.

Once the application is responding to ping requests, it is ready for full configuration.

3.2.4.2 Full System Configuration

To complete the system configuration, an Internet browser is used. Run the browser and point it to the IP address assigned to the DSP in the previous step. If the IP address is "192.63.10.5", the URL would be:

```
http://192.63.10.5
```

Be sure and disable any proxy settings on the browser if your network is behind a firewall.

The browser will display a small WEB page describing the example application. There is a button on this page that will take you to the configuration page. The password required to enter the configuration page is printed on the screen.

Once in the configuration page, simply fill out the form and press the "Submit" button.

If DHCP was selected on the configuration form, the application will attempt to get an IP address from a DHCP server as with the Client example described in the previous section.

3.2.5 Testing the Application

Once the application is configured, has restarted and printed out its IP address, several tests can be performed. These tests are identical to those in the previous Client example.

3.2.5.1 Telnet Server

The example application includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application's IP address. Note that the console program will timeout and disconnect after a period of inactivity. Note also that the Telnet console can be disabled from the configuration WEB page.

3.2.5.2 Data Servers

To try out the data servers, use the Windows test applications found in the \WINAPPS directory. The applications are command line driven and require a target IP address. For example, type:

```
recv 192.63.10.5
```

to start the data receiver. This will request data from the server running on the DSP. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
send 192.63.10.5 100
```
starts the data send test (receive from the DSP's point of view) with a display update interval of 100 iterations.

```
echo 192.63.10.5 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP. All the Windows test clients will run until a key is pressed, or Control-C in the event of an error (e.g.: trying to connect to a bad IP address).

### 3.3 The Serial Client Example Application

#### 3.3.1 Introduction

The serial client example includes the same application components as the network example, with the communication link as Point-to-Point, over a UART cable. Before running the example, an serial connection needs to be set up between the DSP board and a PC. The TMS320DM642 EVM has integrated UART connections. For the other baseboards supported, the 6211 DSK Memory UART Card may be used.

#### 3.3.2 Setting Up the Network

To test the PPP connection when the DSP is acting as a client and a PC as a server, set up an incoming connection on the PC. To do this, go to Start => Accessories => Communications => New Connection Wizard. Click ‘Next’, and then select ‘Set up an advanced connection’. On the next screen, select ‘Accept incoming connections’. When ‘Devices for Incoming Connections’ screen appears, select ‘Communications cable between two computers (COM1)’. Set the properties for this connection at 115200 for port speed and no flow control. On the Advanced tab of the Properties, select 8 Data bits, no parity, and 1 stop bit. On the next screen, select ‘Do not allow virtual private connections’.

After you set up the Incoming connections, open the Network Connections window and select Properties from the right click menu of the Incoming Connections. Create a new user on the Users tab, using the password ‘password’. On the Users tab, also make sure that ‘Require all users to secure their passwords and data’ is unchecked. Otherwise the DSP TCP/IP Stack will not be able to establish a serial connection with the PC.

#### 3.3.3 Building the Application

The serial client example is located in the EXAMPLE\SERIAL\CLIENT directory off the TCP/IP Stack root. The application can be rebuilt directly from its project file using Code Composer Studio.

#### 3.3.4 Loading the Application

The application is loaded and executed via Code Composer Studio. It is a good idea to reset the board before loading Code Composer, but this should not be required. The application will print out status messages via Code Composer's standard IO output window (Stdout).

On a successful execution, one of the status lines printed by the application will display the client's IP address. Once this address is displayed, the DSP will respond to requests made to its IP address. You will also notice that a new networking icon appears on the PC status bar, with the name of the incoming connection you set up.

#### 3.3.5 Testing the Application

Once the application is executing and has printed out its IP address, several tests can be performed.
3.3.5.1 Telnet Server

The client example application also includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application’s IP address. Note that the console program will timeout and disconnect after a period of inactivity.

To get a list of console commands, type “help” or simply “?”. A list of console commands will be printed to the telnet terminal. The console program is important as a programming demonstration as much as a run time demonstration. There are many functions in the console program that display or test features particular to the TCP/IP Stack. When an application developer wants to use these features in their application, the console example source code can be very useful as a guide.

3.3.5.2 Data Servers

To try out the data servers, use the Windows test applications found in the WINAPPS directory off the TCP/IP Stack root. The applications are command line driven and require a target IP address. For example, type:

```
send 196.12.1.14
```

to start the data receiver. This will request data from the server running on the DSP. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
recv 196.12.1.14 100
```

starts the data send test (receive from the DSP’s point of view) with a display update interval of 100 iterations.

```
echoc 196.12.1.14 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP.

All the Windows test clients will run until a key is pressed, or Control-C in the event of an error (e.g.: trying to connect to a bad IP address).

3.4 The Serial Router Example Application

3.4.1 Introduction

The serial router example illustrates how to build a router serving both PPP/serial and Ethernet interfaces. A simple UART connection is used for the serial interface. Before running the example, a client serial connection needs to be set up between the DSP board and a PC. The TMS320DM642 EVM has integrated UART connections. For the other baseboards supported, the 6211 DSK Memory/UART Card may be used.

3.4.2 Setting Up the Network

To test the PPP connection when the DSP is acting as a server and a PC as a client, set up a client connection on the PC. To do this, go to Start => Accessories => Communications => New Connection Wizard. Click ‘Next’, and then select ‘Set up an advanced connection’. On the next screen, select ‘Connect directly to another computer’. On the next screen select ‘Guest’, so the PC can act as a client for the PPP connection. Next, give the connection a name, such as ‘TIDSP’. Next, select ‘Communications cable between two computers (COM1)’, and complete the setup with the default settings.

Set the properties for this connection at 115200 for port speed and no flow control. On the Advanced tab of the Properties screen, select 8 Data bits, no parity, and 1 stop bit. On the next screen, select ‘Do not allow virtual private connections’.
After you set up the client connection, open the 'Network Connections' window and select Properties from the right click menu of the direct connection you just setup. Set the properties for the COM1 device at 115200 for port speed and no flow control. On the Networking tab, click the ‘Settings’ button and make sure that 'Enable LCP extensions' is unchecked.

### 3.4.3 Building the Application

The serial router example is located in the EXAMPLE\SERIAL\ROUTER directory off the TCP/IP Stack root. The application can be rebuilt directly from its project file using Code Composer Studio.

### 3.4.4 Loading the Application

The application is loaded and executed via Code Composer Studio. It is a good idea to reset the board before loading Code Composer, but this should not be required. The application will print out status messages via Code Composer's standard IO output window (Stdout).

If an Ethernet connection is available, one of the status lines printed by the application will display the Ethernet interface IP address on a successful execution (either through DHCP or static configuration). When using DHCP, it is possible that the application will be unable to obtain an IP address from a DHCP server. In this case, the application will eventually print a DHCP status message with the "fault" condition.

Also, one of the printed lines will indicate that the serial connection is ready to accept connections. Once ‘Sctrl: Ready’ is displayed, you can double click the icon of the client connection you have setup earlier, and log in with the User name ‘username’ and password ‘password’. Once the connection is completed, CCS will print the IP address assigned to the serial interface. You will also notice that a new networking icon appears on the PC status bar, with the name of the client connection you set up.

The DSP will now respond to requests made to its both its serial and its Ethernet IP addresses.

### 3.4.5 Testing the Application

Once the application is executing and has printed out its IP address, several tests can be performed.

#### 3.4.5.1 HTTP Server

To see the HTTP server in action, run an Internet browser, and point it to one of the IP addresses displayed by the application. For an IP address of "196.12.1.14", the URL would be:

http://196.12.1.14

Be sure and disable any proxy settings on the browser if your network is behind a firewall.

The browser will display a small WEB page describing the example application. There are server status screens that can be accessed off this page. The source code used to generate these pages is further described in the HTTP appendix of the TCP/IP Stack Programmer's Reference Guide (SPRU524).

#### 3.4.5.2 Telnet Server

The client example application also includes a console application with several tests and status query functions available. In order to get to the console, simply telnet into the application's IP address. Note that the console program will timeout and disconnect after a period of inactivity.

To get a list of console commands, type “help” or simply “?”. A list of console commands will be printed to the telnet terminal. The console program is important as a programming demonstration as much as a run time demonstration. There are many functions in the console program that display or test features particular to the TCP/IP Stack. When an application developer wants to use these features in their application, the console example source code can be very useful as a guide.

#### 3.4.5.3 Data Servers

To try out the data servers, use the Windows test applications found in the WINAPPS directory off the TCP/IP Stack root. The applications are command line driven and require a target IP address. For example, type:

send 196.12.1.14
The Serial Router Example Application

to start the data receiver. This will request data from the server running on the DSK. To get more accurate benchmark numbers, the number of display updates can be reduced by typing an update period. For example:

```
recv 196.12.1.14 100
```

starts the data send test (receive from the DSP’s point of view) with a display update interval of 100 iterations.

```
echoc 196.12.1.14 100
```

starts the TCP data echo test (echoes back the characters it receives from the DSP) with a display update interval of 100 iterations.

```
testudp 196.12.1.14
```

starts an UDP data server that tests the UDP client running on the DSP.

All the Windows test clients will run until a key is pressed, or Control-C in the event of an error (e.g.: trying to connect to a bad IP address).
TCP/IP Stack Library Overview

The TCP/IP Stack software package is designed to be a transparent add-on to DSP/BIOS and Code Composer Studio. This chapter provides information concerning the design, philosophy and organization of the TCP/IP Stack library.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Stack Library Design</td>
<td>26</td>
</tr>
<tr>
<td>4.2 Programming API</td>
<td>27</td>
</tr>
<tr>
<td>4.3 TCP/IP Stack Software Directory</td>
<td>28</td>
</tr>
</tbody>
</table>
4.1 Stack Library Design

The TCP/IP Stack was designed to provide a full TCP/IP functional environment, with or without routing, in a small memory footprint.

4.1.1 Design Philosophy

The TCP/IP Stack is isolated from both the native OS and the low-level hardware by abstracted programming interfaces. The native OS is abstracted by an operating system adaptation layer (OS.LIB), and custom hardware is supported via a hardware adaptation layer (HAL.LIB) library. These libraries are used to interface the stack to DSP/BIOS and to the system peripherals.

4.1.2 Organization

Figure 1 shows a conceptual diagram of how the stack package is organized in terms of function call control flow. The five main libraries that make up the TCP/IP Stack are shown. These are STACK.LIB, NETTOOL.LIB, OS.LIB and MiniPrintf.LIB, HAL.LIB, and NETCTRL.LIB. All these libraries are summarized below.

![Figure 4-1. Stack Control Flow](image)

4.1.2.1 STACK.LIB Library

The STACK library is the main TCP/IP networking stack. It contains everything from the sockets layer at the top to the Ethernet and PPP layers at the bottom. The library is compiled to make use of the DSP/BIOS operating system, and does not need to be ported when moved from one platform to another. Several builds of the library are included in the TCP/IP Stack. Different versions of the library either include or exclude features like PPP, PPPoE, and NAT.
4.1.2.2 NETTOOL.LIB Library

The NETTOOL library contains all the sockets based network services supplied with the TCP/IP Stack, plus a few additional tools designed to aid in the development of network applications. The most frequently used component in the NETTOOL library is the tag based configuration system. The configuration system is used to control nearly every facet of the stack and its services. Configurations can be stored in non-volatile RAM for auto-loading at BOOT time.

4.1.2.3 OS.LIB and MiniPrintf Libraries

These libraries form a thin adaptation layer that maps some abstracted OS function calls to DSP/BIOS function calls. This adaptation layer allows the DSP/BIOS system programmer to tune the TCP/IP Stack system to any OS based on DSP/BIOS. This includes task thread management, memory allocation, packet buffer management, printing, logging, critical sectioning, and cache coherency.

MiniPrintf.LIB provides slim printing functions for the user who needs to keep the footprint small. They are packaged into a separate library, to allow the user the flexibility to use either the full-fledged RTS printing functions provided with the Code Composer Studio, or the small functions included with the MiniPrintf.LIB library.

4.1.2.4 HAL.LIB Library

The HAL.LIB contains files that interface the hardware peripherals to the TCP/IP stack. These include timers, LED indicators, Ethernet devices, and serial ports.

4.1.2.5 NETCTRL.LIB Library

The NETCTRL or Network Control library is in a sense the heart of the stack. It controls the interaction between the TCP/IP and the outside world. Of all the stack modules, it is the most important to the operation of the TCP/IP stack. Its responsibilities include:

- Initializing the TCP/IP stack and low level device drivers
- Booting and maintaining system configuration via configuration service provider callback functions
- Interfacing to the low level device drivers and scheduling driver events to call into the TCP/IP stack
- Unloading the system configuration and driver cleanup on exit

4.2 Programming API

As previously stated, the stack has been designed for optimal isolation, and so that it may seamlessly “plug-in” to varying run-time environments. The applications programmer will be exposed to potentially several different programming interfaces. They are listed here in decreasing order of relevance. All of the following are described in detail in the TCP/IP Stack Programmer's Reference Guide.

4.2.1 Operating System Abstraction

The OS abstraction is comprised of a custom task and semaphore API contained in the OS adaptation layer. The STACK and NETTOOL libraries use these abstractions so that their OS use can be adjusted by adjusting the implementation of the abstraction in OS.LIB. Note that task and semaphore handles created by these APIs are physically DSP/BIOS TSK and SEM objects.

4.2.2 Sockets and Stream IO API

The sockets API is primarily comprised of the standard "BSD" socket layer API, but contains a few other useful calls. These functions are reentrant and thread safe. They appear as an extension of the standard IO supplied with the operating system, and should not conflict with any native file support functions.

4.2.3 Initialization and Configuration

Stack initialization and configuration, and the configuration and initialization of the services that execute on the stack can be a tedious task. For this reason, and to support the ability to save and restore configurations using non-volatile RAM, a configuration manager is supplied which maintains a
configuration database. Service providers can register with the configuration manager and define what action is performed based on information that is written to the configuration. The TCP/IP Stack includes source code to a network "control" module (NETCTRL) that initializes the stack and provides service functions for the standard configuration entities (network services, network addresses, address pools, etc.). This network control module can be used as-is, or modified to suit a custom environment.

4.2.4 **NETTOOL Support Functions**

The NETTOOL library includes both network services and basic network support functions. The API to the support functions is standardized to that of Berkeley Unix where it makes sense, with some additional functions provided for custom features.

4.2.5 **NETTOOL Services**

The NETTOOL services include most network protocol servers required to operate the stack as a network server or router. The API to the services is standardized and uniform across all supported services, plus services may also be invoked by using the configuration system, bypassing the NETTOOLS API entirely.

4.2.6 **Internal Stack API**

The applications programmer will most likely never use the internal stack API (can be thought of as "kernel" level API). However, it is required for some types of stack maintenance, and it is called by some of the sample source code.

4.2.7 **Hardware Adaptation Layer API**

The applications programmer will most likely never call the HAL API directly, but it is required when moving the stack to an alternate hardware platform. The HAL is described in more detail in the **TCP/IP Stack Programmer’s Reference Guide**, and the **TCP/IP Stack Platform Porting Guide**.

4.3 **TCP/IP Stack Software Directory**

Once the TCP/IP Stack is installed from the CD, all the necessary files for using the stack software are contained in the TCP/IP Stack base directory (defaults to C:\CCStudio_v3.2\c6000\NDK). This base directory contains the following subdirectories:

- `<DOCS>`: TCP/IP Stack documentation
- `<EMACUTIL>`: Ethernet MAC hardware utilities (LogicIO & ATEME)
- `<EXAMPLE>`: Example applications
- `<INC>`: TCP/IP Stack include file directory
- `<LIB>`: TCP/IP Stack linkable library directory
- `<SRC>`: TCP/IP Stack source code
- `<WINAPPS>`: Windows "DOS Box" text utilities

4.3.1 **TCP/IP Stack Documentation**

The DOCS directory contains the same documentation that is included on the TCP/IP Stack CD.

4.3.2 **Ethernet MAC Hardware Utilities**

The EMACUTIL directory contains two test applications for verifying the operation and configuring either the LogicIO ETHC6000F or ATEME NDK-NG 3034E1 Ethernet adapters. These tests should be run prior to using the stack to verify the hardware is operating properly. See **Section 2.3** for more details.
4.3.3 Example Programs

The example directory is broken down into several subdirectories. These are as follows:

<TOOLS> Example application source code
<TOOLS \COMMON> Common source code used by multiple examples
<TOOLS \COMMON\BINSRC> Utility for creating embedded WEB page data
<TOOLS \COMMON\CGI> Functions for use when creating embedded HTTP CGI files
<TOOLS \COMMON\CONSOLE> Command line based console program
<TOOLS \COMMON\HDLC> HDLC Serial Interface (SI) module for PPP over serial
<TOOLS \COMMON\SERVERS> Test servers used for testing and benchmarks

<NETWORK> Demos of the software in COMMON for different platforms
<NETWORK\CFGDEMO> Examples showing embedded system configuration via HTTP
<NETWORK\CLIENT> Standard IP client demonstration
<SERIAL> Example programs using the TL16C750 serial daughter card
<SERIAL\CLIENT> HDLC/PPP client over serial (no Ethernet)
<SERIAL\ROUTER> Router with HDLC/PPP Server over serial and Ethernet

In each of the NETWORK and SERIAL examples, the directories are further broken down by platform and common code. For example, under example\network\client are the following directories:

<COMMON> Example source code that is common to all platforms
<DSK6416> Project files for the 6416 DSK using internal memory
<DSK6416_EXT> Project files for the 6416 DSK using external memory
<DSK6711> Project files for the 6711 DSK
<DSK6713> Project files for the 6713 DSK
<EVMDM642> Project files for the DM642 EVM
<DSK6455> Project files for the 6455 DSK/EVM using internal memory
<DSK6455_EXT> Project files for the 6455 DSK/EVM using external memory

When choosing an example to run, select the directory that matches your hardware platform.

4.3.4 TCP/IP Stack Include File Directory

The include file directory (INC) contains all the include files that can be referenced by a network application. It will be necessary to include this directory in the software tools default search path, or in the search path of the Code Composer project file. The latter method is used in the example programs. The major include files are as follows:

<INC> Main include file directory

NETMAIN.H Master include file for applications (STACKSYS.H, _NETTOOL.H, _NETCTRL.H)
STACKSYS.H Main include file (minus the end-application oriented include files) (USERTYPE.H, SERRNO.H, SOCKET.H, OSIF.H, HAL.H)
_NETCTRL.H Includes references for the NETCTRL scheduler library
_NETTOOL.H Includes references for all the services in the NETTOOL library
_OSKERN.H  Includes kernel level OS functions declarations
_STACK.H  Includes all low level STACK interface functions
SERRNO.H  Standard error values
SOCKET.H  Prototypes for all file descriptor based functions
STKMAIN.H  Include file used by low-level modules (not for use by applications)
USERTYPE.H  Standard types used by the stack

4.3.5  Linkable Libraries Directory
The LIB directory contains two sets of libraries, the TCP/IP stack libraries which are platform independent, and the HAL libraries which are platform dependent. The top level structure of the LIB directory is as follows:

<LIB>  Linkable Library Files
   <LIB\C6200>  TCP/IP Stack libraries for TMS320C62x DSP
   <LIB\C6400>  TCP/IP Stack libraries for TMS320C64x DSP
   <LIB\C64PLUS>  TCP/IP Stack libraries for TMS320C64Plus DSP
   <LIB\HAL>  TCP/IP Stack HAL libraries

4.3.5.1  TCP/IP Stack Libraries
The TCP/IP Stack library files are those that do not change when moving from platform to platform. Several builds of the main stack library are provided with PPP, PPPoE, and NAT either enabled or disabled. For example, the directory structure for TMS320C62x DSP is:

   <C6200>  TCP/IP Stack libraries for 320C62x DSP
      NETCTRL.LIB  Network Initialization and Control library
      NETTOOL.LIB  Network Tools function library
      OS.LIB  OS Adaptation Layer library (with priority exclusion)
      OS_SEM.LIB  OS Adaptation Layer library (with semaphore exclusion)
      MiniPrintf.LIB  Small code size printf library
      STACK.LIB  TCP/IP Stack library
   <C6200\ALL_STK>  Alternate builds of the stack library
      STK.LIB  Stack without PPP, PPPoE, and NAT
      STK_NAT.LIB  Stack with NAT, but without PPP and PPPoE
      STK_NAT_PPP.LIB  Stack with PPP and NAT, but without PPPoE
      STK_NAT_PPP_PPPOE.LIB  Stack with PPP, PPPoE, and NAT
      STK_PPP.LIB  Stack with PPP, but without PPPoE and NAT
      STK_PPP_PPPOE.LIB  Stack with PPP and PPPoE, but without NAT (default library)

4.3.5.2  HAL Libraries
The TCP/IP Stack HAL libraries are arranged by platform and device. The basic structure of the HAL directory is as follows:
Inside each HAL directory is a set of driver files. The driver files use the naming convention of HAL_CLASSDEVICE.LIB. For example, HAL_ETH_MXF.LIB is an Ethernet driver based on the Macronix MX98728. The “F” in “MXF” is a flavor of driver meaning full background EDMA is used.

As an example of what is contained in the HAL for a specific platform, the HAL files for the 6416 DSK are:

```
<DSK6416>
HAL_ETH_MXF.LIB  Ethernet MX98728 (LogicIO) full background EDMA
HAL_ETH_MXI.LIB  Ethernet MX98728 (LogicIO) interrupt driven (no EDMA)
HAL_ETH_MXP.LIB  Ethernet MX98728 (LogicIO) polling (no EDMA)
HAL_ETH_SMSCF.LIB  Ethernet SMSC LAN91C111 (ATEME) full background EDMA
HAL_ETH_SMSCFLIB  Ethernet SMSC LAN91C111 (ATEME) interrupt (no EDMA)
HAL_ETH_STUB.LIB  Ethernet stub driver (when not using Ethernet)
HAL_SER_STUB.LIB  Serial port stub driver (when not using serial port)
HAL_SER_TI750.LIB  Serial port driver for TI TL16C750
HAL_TIMER.LIB  Timer driver
HAL_USERLED.LIB  User LED indicator driver
```

Note that some devices are not available with all platforms. The contents of any one HAL platform directory may vary.

### 4.3.6 Library Source Directory

The SRC directory contains source code to the serviceable libraries in the TCP/IP Stack. Source code includes the following directories:

```
<SRC>
SRC\HAL>  Source Code to Library Files
SRC\NETCTRL>  Source to the Network Control Module
SRC\OS>  Source to the OS Adaptation Layer
SRC\MiniPrintf>  Source to the small footprint printing routines
```

The NETCTRL, OS and MiniPrintf source modules are discussed in detail at the end of this document. The HAL is discussed in the *TCP/IP Stack Platform Porting Guide*. 
4.3.7 **Windows Test Utilities**

The WINAPPS directory contains four very simple test applications that can be run from a Windows DOS box to verify the operation of the Console example program. These test applications act as network clients for TCP send, receive, and echo, and for UDP echo operations. Most of the TCP/IP Stack example programs contain network data servers that can communicate with these test applications. The SEND, RECV, ECHO, and TESTUDP applications are referenced in the description of these examples which can be found in Chapter 3.
Network Application Development

Developing a network application with the C6000 TCP/IP Stack software is as easy as programming with a standard sockets API. However, integrating with Code Composer Studio, DSP/BIOS™, and system initialization will be new to application developers. This chapter shows how to start developing network applications now, as it discusses the issues and guidelines involved in the development of network applications using the TCP/IP Stack libraries.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Using Code Composer Studio</td>
<td>34</td>
</tr>
<tr>
<td>5.2 Developing Socket Applications with DSP/BIOS</td>
<td>35</td>
</tr>
<tr>
<td>5.3 TCP/IP Stack Initialization and Configuration</td>
<td>38</td>
</tr>
<tr>
<td>5.4 Application Debug and Troubleshooting</td>
<td>49</td>
</tr>
</tbody>
</table>
5.1 Using Code Composer Studio

All network application development is performed in Code Composer Studio (CCS). The stack libraries are designed to work with Code Composer Studio. This section provides some guidelines for developing TCP/IP Stack applications under CCS.

5.1.1 Required Configuration Entries

The TCP/IP stack does not have any special requirements, but is bolted to DSP/BIOS and the hardware via the OS adaptation layer and the HAL layer. These libraries do require DSP/BIOS objects to be created in order for them to work properly. This requirement can be altered by altering the OS and HAL layers.

5.1.1.1 PRD Object

The timer driver in the HAL requires that a PRD function be created to drive its main timer. The PRD must be configured to fire every 100mS, and call the timer driver function IlTimerTick().

5.1.1.2 HOOK Object

The task adaptation module in the OS library requires a hook to be able to save and load private environment pointers for the TCP/IP stack. This is done by creating a DSP/BIOS “hook”. A hook module must be created to call the OS hook functions NDK_hookInit() and NDK_hookCreate().

5.1.2 Include Files and Library Files

The base release package is organized as a set of library files and an include file directory. The directory structure is shown in the previous section. When developing an application, it is best to include the base TCP/IP Stack include directory in the project build options of the CCS project. For example, with the default installation, the project should be set to include the include file path C:\CCStudio_v3.2\C6000\NDK\inc.

The selection of available library files is described in the previous section. It is easiest to add desired library files directly into the project. This way, the linker will know where to find them. This can become an issue for link ordering however, so be sure to take note of the following section on link order.

5.1.3 CCS Project Link Order

CCS has the ability to link object files and libraries in a specific order. This “link order” is a very important step in getting a TCP/IP Stack application to work correctly. By default, CCS projects do not have a link order, and sometimes the “unordered” link order causes problems; the linker may report multiply defined symbols, or the program may link without errors and simply fail to operate correctly.

To ensure a smooth and correct project build, be sure to enable order on any CCS project. To do this, select the “Link Order” tab from the project “build options” dialog. Then, add all project files to the link order and manipulate the order so that all .LIB files are at the end of the link order list. Alternately, you can use the ‘-priority’ linker switch to establish a link order.

The best order for the TCP/IP Stack libraries (at the end of the list) is: NETCTRL.LIB, HAL_xxx.LIB, NETTOOLS.LIB, STACK.LIB, OS.LIB, and MiniPrintf.LIB. When the small footprint printing library is not added to the project, the standard RTS functions will be used for printing.

5.1.4 TCP/IP Stack Memory Sections

The TCP/IP Stack defines some special memory segments via the pragma:

```
#pragma DATA_SECTION( memory_label, "SECTIONNAME")
```

The TCP/IP Stack sections are defined by default as subsections of the far memory segment. External memory is usually used for the far section. The additional section names are shown below.
.far:NDK_PACKETMEM — This section is defined in the HAL and OS adaptation layers for packet buffer memory. The size required is normally 32kb to 48kb bytes. This is set by the packet buffer manager (pbm.c) in the OS adaptation layer.

.far:NDK_MMBUFFER — This section is defined by the memory allocation system for use as a scratchpad memory resource. The size of the memory declared in this section is adjustable, but the default is less than 48k bytes.

.far:NDK_OBJMEM — This section is a catch-all for other large data buffer declarations. It is used by the example application code and the OS adaptation layer (for print buffers).

Instead of using the default project CMD file, the example networking code includes an alternate CMD file that is used to specify the Chip Support Library (CSL) when required by the project. The alternate linker command file includes the project's default command file. If the project is changed, the include line must also be altered.

All these memory sections are defined in either the network application, HAL, or OS adaptation layer. The user has full control over the placement of the TCP/IP Stack sections in memory. If you want to place these special segments into a user-defined section called MYSDRAM, instead of the default far memory section, just add the following lines at the end of the alternate CMD file for your project:

```
SECTIONS
{
 .far:PACKETMEM: {} > MYSDRAM
 .far:NMBUFFER: {} > MYSDRAM
 .far:OBJMEM: {} > MYSDRAM
}
```

5.1.5 Using Cache

The example program for each individual platform is pre-set with the preferred cache configuration. When internal memory is not required, 4-way cache is used. Otherwise, cache is selected to meet internal memory requirements. Any system that requires a specific memory/cache map should be clearly documented. Also, the HAL drivers all assume that there is at least some L2 cache. The drivers may not behave properly when L2 is mapped entirely to SRAM.

5.2 Developing Socket Applications with DSP/BIOS

Chapter 3 of the TCP/IP Stack Programmer’s Reference Guide describes the API for using file descriptors in a DSP/BIOS task thread. This section discusses some of the issues when using this task and file descriptor table environment, but does not discuss the entire API.

5.2.1 Default Environment API Restrictions

The stack is designed to be flexible, and has a OS adaptation layer that can be adjusted to support any system software environment that is built on top of DSP/BIOS. Although the environment can be adjusted to suit any need by adjusting the HAL, NETCTRL and OS modules, the following restrictions should be noted for the most common environments:

1. The Network Control Module (NETCTRL) contains a network scheduler thread that schedules the processing of network events. The scheduler thread can run at any priority with the proper adjustment. Typically, the scheduler priority will be low (lower than any network task), or high (higher than any network task). Running the scheduler thread at a low priority places certain restrictions on how a task can operate at the socket layer. For example:
   - If a task polls for data using the recv() function in a non-block mode, no data will ever be received. This is because the application never blocks to allow the scheduler to process incoming packets.
   - If a task calls send() in a loop using UDP, and the destination IP address is not in the ARP table, the UDP packets will not be sent because the scheduler thread is never allowed to run to process the ARP reply. These cases are seen more in UDP operation than in TCP. In order to make the TCP/IP behave more like a standard socket environment for UDP, the priority of the scheduler thread can be set to high priority. See Chapter 6 for more details on network event scheduling.
2. The TCP/IP stack requires a re-entrance exclusion methodology to call into internal stack functions. This is called "kernel mode" by the TCP/IP Stack, and is entered by calling the function ilEnter() and exited via ilExit(). Application programmers will typically not call these functions, but they must be aware of how the functions work.

By default, priority inversion is used to implement the kernel exclusion methods. When in kernel mode, a task's priority is raised to "OS_TASKPRIKERN". Application programmers need to be careful not to call stack functions from threads with a priority equal to or above that of OS_TASKPRIKERN, as this could cause illegal reentrancy into the stack's kernel functions. For systems that cannot tolerate priority restrictions, the TCP/IP Stack can be adjusted to use semaphores for kernel exclusion. This can be done by altering the OS adaptation layer discussed in Section 7.2.3, or by using the semaphore-based version of the OS library: OS_SEM.LIB.

5.2.2 Creating a Task

The process of creating a sockets application begins with the creation of the task thread. With the supplied stack library, tasks can be created using the standard DSP/BIOS API or the provided task abstraction. For example, the following call creates a basic task:

```c
struct TSK_Attrs ta;
ta = TSK_ATTRS;
ta.priority = OS_TASKPRINORM;
ta.stack = 0;
ta.stacksize = stacksize;
ta.stackseg = 0;
ta.environ = 0;
ta.name = "TaskName";
ta.exitflag = 0;
hMyTask = TSK_create( (Fxns)entrypoint, &ta, arg1, arg2, arg3 );
```

The same task can be created via the TaskCreate() function in the task abstraction API. The abstracted function is a little more restrictive. It creates a task thread with exactly 3 parameters (they do not all have to be used) with the same basic task attributes as DSP/BIOS. For example, the following call would create an identical task to that shown above:

```c
hMyTask = TaskCreate( entrypoint, "TaskName", OS_TASKPRINORM,
                        stacksize, arg1, arg2, arg3 );
```

In both cases, hMyTask is a handle to a DSP/BIOS "TSK" task thread.

5.2.2.1 Stack Size

Care should be taken when choosing a stack size. Due to its somewhat recursive nature, the stack tends to consume a significant amount of stack. A stack size of 3072 is OK for UDP based communications. For TCP, 4096 should be used as a minimum, with 5120 being chosen for protocol servers. The thread that calls the NETCTRL library functions should have a stack size of at least 4096 bytes. If lesser values are used the applications programmer should be on the lookout for stack overflow conditions.

5.2.2.2 Choosing Task Priorities

In general, tasks that make use of functions in the network stack should be of a priority no less than OS_TASKPRILOW, and no higher than OS_TASKPRIHIGH. For a typical task, use a priority of OS_TASKPRINORM. The values for these #define variables can be altered by adjusting the OSENVCFG structure as described in the Programmer's Guide, however this is strongly discouraged. When altering the priority band, care must be taken so that the network scheduler thread and the kernel priority are both accounted for.

5.2.2.3 Initializing the File Descriptor Table

Each task thread that needs to make use of the sockets or file API included in the stack must allocate a file descriptor table and associate the table with the task handle. This process is described fully in the TCP/IP Stack Programmer's Reference Guide. The basic idea is that a call to fdOpenSession() must be performed before any file descriptor oriented functions are used, and then fdCloseSession() is called when they are no longer required.
5.2.3 Memory Allocation

Section 2.4 of the TCP/IP Stack Programmer’s Reference Guide describes the memory allocation API provided by the OS library for use by the various stack libraries. Although the stack’s memory allocation API has a couple benefits (it is portable, bucket based to prevent fragmentation, and tracks memory leaks), it is expected that application code use the standard malloc()/free() or equivalent “MEM” allocation routines provided by DSP/BIOS.

5.2.4 Example Code

Sometimes its best to jump right into it. The following is an echo sockets application for DSP/BIOS. It creates a socket, connects to port 7, sends some data and then tries to receive it back.

The lines of code in **boldface** represent new functions required to provide sockets functionality to DSP/BIOS. The functions in **bold italic** are standard, but their names have been adjusted to avoid naming conflicts with Code Composer Studio’s runtime support library. The remainder of the functions should be very familiar to Berkeley sockets programmers. All of these functions are described in detail in the TCP/IP Stack Programmer’s Reference Guide.

```c
void EchoTcp ( IPN IPAddr )
{
    SOCKET s = INVALID_SOCKET;
    struct sockaddr_in sin1;
    int i;
    char *pBuf = 0;
    struct timeval timeout;

    // Allocate the file descriptor environment for this task
    fdOpenSession( (HANDLE)TSK_self() );
    printf("\n== Start TCP Echo Client Test ==\n");

    // Create test socket
    s = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    if( s == INVALID_SOCKET )
    {
        printf("failed socket create (%d)\n", fdError());
        goto leave;
    }

    // Prepare address for connect
    bzero( &sin1, sizeof(struct sockaddr_in) );
    sin1.sin_family = AF_INET;
    sin1.sin_len = sizeof( sin1 );
    sin1.sin_addr.s_addr = IPAddr;
    sin1.sin_port = htons(7);

    // Configure our Tx and Rx timeout to be 5 seconds
    timeout.tv_sec = 5;
    timeout.tv_usec = 0;
    setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof( timeout ) );
    setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof( timeout ) );

    // Connect socket
    if( connect( s, (PSA)&sin1, sizeof(sin1) ) < 0 )
    {
        printf("failed connect (%d)\n", fdError());
        goto leave;
    }

    // Allocate a working buffer
    if( !(pBuf = malloc( 4096 )) )
    {
        printf("failed temp buffer allocation\n");
        goto leave;
    }

    // Fill buffer with a test pattern
    for(i=0; i<4096; i++)
        *(pBuf+i) = (char)i;

    // Send the buffer
```
5.3 TCP/IP Stack Initialization and Configuration

Before a sockets application like the example shown in Section 5.2.4 can be executed, the stack must be properly configured and initialized. In order to facilitate a standard initialization process, and yet allow customization, source code to the network control module (NETCTRL.LIB) is included in the TCP/IP Stack. The NETCTRL module is the heart of the stack's initialization, configuration, and event scheduling. A solid comprehension of NETCTRL's operation is essential for building a solid networking application.

This section describes how to use NETCTRL in an networking application. An explanation of how NETCTRL works and how it can be tuned is provided in Chapter 6.

5.3.1 TCP/IP Stack Initialization Using NETCTRL

The process of initialization and configuration of the TCP/IP stack is described in detail in Chapter 4 of the TCP/IP Stack Programmer's Reference Guide. This section closely mirrors the initialization procedure described in Section 4.3 of that document. Here we describe the information with a more practical slant. Programmers concerned with the exact API of the functions mentioned here should refer to the TCP/IP Stack Programmer's Reference Guide for a more precise description.

5.3.1.1 The NETCTRL Task Thread

The NETCTRL task thread (called scheduler thread) is the task thread on which nearly all the NETCTRL activity takes place. This thread is created by the programmer, either in the DSP/BIOS .tcf file or via the DSP/BIOS API. In all the example applications, there is one main application thread created in the

```c
if( send( s, pBuf, 4096, 0 ) < 0 )
{
    printf("send failed (%d)\n", fdError());
    goto leave;
}
// Try and receive the test pattern back
i = recv( s, pBuf, 4096, MSG_WAITALL );
if( i < 0 )
{
    printf("recv failed (%d)\n", fdError());
    goto leave;
}
// Verify reception size and pattern
if( i != test )
{
    printf("received %d (not %d) bytes\n", i, test);
    goto leave;
} for(i=0; i<test; i++)
    if( *(pBuf+i) != (char)i )
    {
        printf("verify failed at byte %d\n", i);
        break;
    }
// If here, the test passed
if( i==test )
    printf("passed\n");
leave:
    if( pBuf )
        free( pBuf );
    if( s != INVALID_SOCKET )
        fdClose( s );
    printf("== End TCP Echo Client Test ==\n");
    // Free the file descriptor environment for this task
    fdCloseSession( (HANDLE)TSK_self() );
    TSK_exit();
}````
5.3.1.2 Pre-Initialization

Before calling any other of the stack API functions, the primary initialization function \texttt{NC\_SystemOpen()} must be called. This initializes the stack and the memory environment used by all the stack components. Two calling arguments, \texttt{Priority} and \texttt{OpMode}, indicate how the scheduler should execute.

\texttt{Priority} is set to either \texttt{NC\_PRIORITY\_LOW} or \texttt{NC\_PRIORITY\_HIGH}, and determines the scheduler task’s priority relative to other networking tasks in the system.

\texttt{OpMode} is set to either \texttt{NC\_OPMODE\_POLLING} or \texttt{NC\_OPMODE\_INTERRUPT}, and determines when the scheduler attempts to execute. The “interrupt” mode is used in the vast majority of applications. Note that “polling” mode attempts to run continuously, so when polling is used, \texttt{Priority} must be set to \texttt{NC\_PRIORITY\_LOW}.

For example, all the example applications included in the TCP/IP Stack contain the following:

```c
//
// THIS IS THE FIRST THING DONE IN AN APPLICATION!!
//
rc = NC\_SystemOpen( NC\_PRIORITY\_LOW, NC\_OPMODE\_INTERRUPT );
if( rc )
{
    printf("NC\_SystemOpen Failed (%d)\n",rc);
    for(;;);
}
```

5.3.1.3 System Configuration

In order to use the NETCTRL API, a system configuration must be created. The configuration is a handle based object that holds a multitude of system parameters. It is these parameters that control the operation of the stack. Typical configuration parameters include:

- Network Hostname
- IP Address and Subnet Mask
- IP Address of Default Routes
- Services to be Executed (DHCP, DNS, HTTP, etc)
- IP Address of name servers
- Stack Properties (IP routing, socket buffer size, ARP timeouts, etc.)

The process of creating a configuration always starts out with a call to \texttt{CfgNew()} to create a configuration handle. Once the configuration handle is created, configuration information can be loaded into the handle in bulk or constructed into it one entry at a time.

Loading a configuration in bulk requires that a previously constructed configuration has been saved to non-volatile storage. Once the configuration is in memory, the information can be loaded into the configuration handle by calling \texttt{CfgLoad()}.

Another option is to manually add individual items to the configuration for the various desired properties. This is done by calling \texttt{CfgAddEntry()} for each individual entry to add.

The exact specification of the stack’s configuration API appears in Chapter 4 of the \textit{TCP/IP Stack Programmer’s Reference Guide}. Some additional programming examples are provided in Section 5.3.3 of this document, and in the TCP/IP Stack example programs.
5.3.1.4 Network Startup

Once the configuration handle is loaded with the proper configuration, the network (and the network event scheduler) is invoked by calling the NETCTRL function `NC_NetStart()`. Besides the handle to the configuration, this function takes three additional callback pointer parameters; a pointer to a "Start" callback function, a "Stop" function, and a "IP Address Event" function.

The first two callback functions are called only once. The "Start" callback is called when the system is initialized and ready to execute network applications (note there may not be a local IP network address installed yet). The "Stop" callback is called when the system is shutting down and signifies that the stack will soon not be able to execute network applications. The third callback can be called multiple times. It is called when a local IP address is either added or removed from the system. This can be useful in detecting new DHCP or PPP address events, or just to record the local IP address for use by local network applications.

The call to `NC_NetStart()` will not return until the system has shutdown, and then it returns a shutdown code as its return value. How the system was shutdown may be important to determine if the stack should be rebooted. For example, a reboot may be desired in order to load a new configuration. The return code from `NC_NetStart()` can be used to determine if `NC_NetStart()` should be called again (and hence perform the reboot).

For a simple example, the following code continuously reboots the stack using the current configuration handle if the stack shuts down with a return code greater than zero. The return code is set when the stack is shutdown via a call to `NC_NetStop()`.

```c
//
// Boot the system using our configuration
//
// We keep booting until the function returns 0. This allows
// us to have a "reboot" command.
//
// do
//  rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
// while( rc > 0 );
```

5.3.1.5 Invoking New Network Tasks and Services

Some standard network services can be specified in the system configuration, and these are loaded and unloaded automatically by the NETCTRL module. Other services, including those written by an applications programmer should be launched from the "Start" callback function that was supplied to `NC_NetStart()`.

As an example of a network start callback, the `NetworkStart()` function below will open a user SMTP server application by calling an “open” function to create the main application thread.

```c
static SMTP_Handle hSMTP;

//
// NetworkStart
//
// This function is called after the configuration has booted
//
static void NetworkStart( )
{
    // Create an SMTP server
    task hSMTP = SMTP_open( );
}
```

The above code launches a self contained application that needs no further monitoring, but we need to shutdown the application when the system shuts down. This is done via the `NetworkStop()` callback function. Therefore, our `NetworkStop()` function must “undo” what we did in `NetworkStart()`.

```c
//
// NetworkStop
//
```
5.3.1.6 Shutdown

There are two ways the stack can be shutdown. The first is a manual shutdown which occurs when an application calls `NC_NetStop()`. Here, the calling argument to the function is returned to the NETCTRL thread as the return value from `NC_NetStart()`. Therefore, for our example code, calling `NC_NetStop(1)` reboots the network stack, while calling `NC_NetStop(0)` shuts down the network stack.

The second way the stack can be shutdown is when the stack code detects a fatal error. A fatal error is an error above the fatal threshold set in the configuration. This type of error generally indicates that it is not safe for the stack to continue. When this occurs, the stack code will call `NC_NetStop(-1)`. It is then up to the programmer to determine what should be done next. The way the `NC_NetStart()` loop is coded determines if the system will shutdown (as in our example), or simply reboot.

Note that the critical threshold to shutdown can also be disabled. The following code can be added to the configuration to disable error related shutdown:

```c
// We do not want the stack to abort on any error
uint rc = DBG_NONE;
CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGABORTLEVEL,
    CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)(&rc, 0 ));
```

The above example assumes that the network task can be launched whether or not the stack has a local IP address. This is true for servers that “listen” on a wildcard address of 0.0.0.0. In some rare cases, an IP address may be required for task initialization, or perhaps an IP address on a certain device type is required. In these circumstances, the `NetworkIPAddr()` callback function is used to signal the application that it is safe to start.

The following example illustrates the calling parameters to the `NetworkIPAddr()` callback. Note that the `IFIndexGetHandle()` and `IFGetType()` functions can be called to get the type of device (HTYPE_ETH or HTYPE_PPP) on which the new IP address is being added or removed. This example just prints a message. The most common use of this callback function is to synchronize network tasks that require a local IP address to be installed before executing.

```c
//
// NetworkIPAddr
//
// This function is called whenever an IP address binding is
// added or removed from the system.
//
static void NetworkIPAddr( IPN IPAddr, uint IfIdx, uint fAdd )
{
    IPN IPTmp;
    if( fAdd )
        printf("Network Added: ");
    else
        printf("Network Removed: ");

    // Print a message
    IPTmp = ntohl( IPAddr );
    printf("If-%d:%d.%d.%d\n", IfIdx,
        (UINT8)(IPTmp>>24)&0xFF, (UINT8)(IPTmp>>16)&0xFF,
        (UINT8)(IPTmp>>8)&0xFF, (UINT8)IPTmp&0xFF );
}
```
5.3.2 Adding Standard Services

The configuration system can also be used to invoke the standard network services found in the NETTOOLS library. The services available to network applications using the TCP/IP Stack are discussed in detail in Chapter 6 of the TCP/IP Stack Programmer's Reference Guide. This section gives a short summary of using the services described in that chapter.

When using the NETTOOLS library, the NETTOOLS status callback function is introduced. This callback function is used to track the state of services that are enabled through the configuration. There are actually two levels to the status callback function. The first callback is made by the NETTOOLS service. It calls the configuration service provider when the status of the service changes. The configuration service provider then adds its own status to the information and calls back to the application's callback function. A pointer to the application's callback is provided when the application adds the service to the system configuration.

Here, we show the basic status callback function that is used in all our examples:

```c
//
// Service Status Reports
//
static char *TaskName[] = { "Telnet","HTTP","NAT","DHCP","DNS" };
static char *ReportStr[] = { ":","Running","Updated","Complete","Fault" };
static char *StatusStr[] = { "Disabled", "Waiting", "IPTerm",
                           "Failed", "Enabled" }
static void ServiceReport( uint Item, uint Status, uint Report, HANDLE h )
{
    printf( "Service Status: %-9s: %-9s: %-9s
",
            TaskName[Item-1], StatusStr[Status],
            ReportStr[Report/256], Report&0xFF );
}
```

Note that the names of the individual services are listed in the TaskName[] array. This order is specified by the definition of the service "items" in the configuration system and is constant. See the file INC\NETTOOLS\NETCFG.H for the physical declarations.

Note that the strings defining the master report code are listed in the ReportStr[] array. This order is specified by the NETTOOLS standard reporting mechanism and is constant. See the file INC\NETTOOLS\NETTOOLS.H for the physical declarations.

Note that the strings defining the task state are defined in the StatusStr[] array. This order is specified by the definition of the standard service structure in the configuration system. See the file INC\NETTOOLS\NETCFG.H for the physical declarations.

The last value this callback function prints is the least significant 8 bits of the value passed in Report. This value is specific to the service in question. For most services this value is redundant. Usually, if the service succeeds it will report "Complete", and if the service fails it will report "Fault". For services that never complete (for example DHCP client continues to run while the IP lease is active), the upper byte of Report will signify "Running" and the service specific lower byte must be used to determine the current state.

For example, the status codes returned in the 8 least significant bits of Report when using the DHCP client service are:

```c
DHPCPCODE_IPADD       Client has added an IP address
DHPCPCODE_IPREMOVE     IP address removed and CFG erased
DHPCPCODE_IPRENEW      IP renewed, DHCP config space reset
```

These DHCP client specific report codes are defined in INC\NETTOOLS\INC\DHCPIF.H. In most cases the application developer will not have to examine state report codes down to this level of detail. However, there is one case that deserves mention.

When using the DHCP client to configure the stack, the DHCP client controls the first 256 entries of the CFGETAG_SYSINFO tag space. These entries correspond to the 256 DHCP option tags. An application may check for DHPCPCODE_IPADD or DHPCPCODE_IPRENEW return codes so that is can read or alter information obtained by DHCP client. This is discussed further in Section 5.3.3.2.
5.3.3 Initialization Examples

This section contains some sample code for constructing configurations. These examples use the same initialization, configuration, and callback functions discussed in the previous sections.

5.3.3.1 Construction a Configuration for a Static IP and Gateway

The NetworkTest() function in this example consists of the main initialization thread for the stack. It creates a new configuration, adds a static IP address, subnet, and default gateway, and then boots up the stack.

In this case, it is assumed that the addressing and name information is stored in some non-volatile fashion. Here, we have defined some strings to hold the information. For example:

```c
char *LocalIPAddr = "194.16.11.12";
char *LocalIPMask = "255.255.255.0";
char *GatewayIP = "194.16.10.1";
char *HostName = "testhost";
char *DomainName = "demo.net";
```

The code below will perform the following operations:

1. Call NC_SystemOpen() and Create a new configuration.
2. Create and add a configuration entry for the local IP address and subnet using the supplied LocalIPAddr, LocalIPMask, and DomainName strings.
3. Create and add a configuration entry for the local hostname using the Hostname string.
4. Create and add a default route to the router supplied in the GatewayIP string.
5. Boot the system using this configuration by calling NC_NetStart().
6. Free the configuration on system shutdown (when NC_NetStart() returns) and call NC_SystemClose().

```c
int NetworkTest()
{
    int rc;
    CI_IPNET NA;
    CI_ROUTE RT;
    HANDLE hCfg;

    // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    //
    rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if( rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }

    // Create and build the system configuration from scratch.
    //
    // Create a new configuration
    hCfg = CfgNew();
    if( !hCfg )
    {
        printf("Unable to create configuration\n");
        goto main_exit;
    }

    // We better validate the length of the supplied names
    if( strlen( DomainName ) >= CFG_DOMAIN_MAX ||
        strlen( HostName ) >= CFG_HOSTNAME_MAX )
    {
        printf("Names too long\n");
    }
```


5.3.3.2 Constructing a Configuration using the DHCP Client Service

In this section we take the initialization example of the previous section and alter it to instruct the stack to use the DHCP (Dynamic Host Configuration Protocol) client service to perform its IP address configuration.

Since DHCP provides the IP address, route, domain, and domain name servers, we only need to provide the hostname. Here is how the NetworkTest() function would look (please refer to the TCP/IP Stack Programmer's Reference Guide for more details on using DHCP).

The code below will perform the following operations:
1. Call NC_SystemOpen() and create a new configuration.
2. Create and add a configuration entry specifying the DHCP client service to be used.
3. Create and add a configuration entry for the local hostname using the Hostname string.
4. Boot the system using this configuration by calling `NC_NetStart()`.
5. Free the configuration on system shutdown (when `NC_NetStart()` returns) and call `NC_SystemClose()`.

```c
char *HostName = "testhost";

int NetworkTest()
{
    int rc;
    CI_SERVICE_DHCPC dhcpc;
    HANDLE hCfg;

    // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    // rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if (rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }

    // Create and build the system configuration from scratch.

    // Create a new configuration
    hCfg = CfgNew();
    if ( !hCfg )
    {
        printf("Unable to create configuration\n");
        goto main_exit;
    }

    // We better validate the length of the supplied names
    if( strlen( HostName ) >= CFG_HOSTNAME_MAX )
    {
        printf("Names too long\n");
        goto main_exit;
    }

    // Specify DHCP Service on interface 1
    bzero( &dhcpc, sizeof(dhcpc) );
    dhcpc.cisargs.Mode = CIS_FLG_IFIDXVALID;
    dhcpc.cisargs.IfIdx = 1;
    dhcpc.cisargs.pCbSrv = &ServiceReport;
    CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
                 sizeof(dhcpc), (UINT8 *)&dhcpc, 0 );

    // Add our hostname
    CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_HOSTNAME, 0,
                 strlen(HostName), (UINT8 *)HostName, 0 );

    // Boot the system using this configuration
    // We keep booting until the function returns less than 1. This allows
    // us to have a "reboot" command.
    //
    do
    {
        rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
    } while( rc > 0 );
}
5.3.3.3 Using a Statically Defined DNS Server

One area of the configuration system that can get a little tricky is that used by the DHCP client. When the DHCP client is in use, it has full control over the first 256 entries in the system information portion of the configuration system. In some rare instances, it may be useful to "share" this space with DHCP.

For example, assume a network application needs to manually add the IP address of a Domain Name System (DNS) server to the system configuration. When DHCP is not being used, this code is very straightforward. To add a DNS server of "128.114.12.2" the following code would be added to the configuration build process (before calling NC_NetStart()).

```c
IPN IPTmp;

// Manually add the DNS server "128.114.12.2"
IPTmp = inet_addr("128.114.12.2");
CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAME, 
  0, _inet_addr("128.114.12.2"), (UINT8 *)&IPTmp, 0,0);
```

Note that the CLIENT example program in the example applications uses a form of this code. Now, when a DHCP client is used, it will clear and reset the contents of the part of the configuration it controls. This includes the DNS server addresses. Therefore, if the above code was added to an application that used DHCP, the entry would be cleared whenever DHCP executed a status update.

In order to share this configuration space with DHCP (or to read the results of a DHCP configuration), the DHCP status callback report codes must be used. The status callback function was introduced in Section 5.3.3. When DHCP reports a status change, the application knows that the DHCP portion of the system configuration has been reset.

The following code also appears in the CLIENT example program. This code is used to manually add a DNS server address when the DHCP client is in use. Note that this code is part of the standard service callback function that is supplied to the configuration when the DHCP client service is specified.

```c
// Service Status Reports
//
static char *TaskName[] = { "Telnet", "HTTP", "NAT", "DHCPSE", "DNS" };
static char *ReportStr[] = { "", "Running", "Updated", "Complete", "Fault" };
static char *StatusStr[] = { "Disabled", "Waiting", "IPTerm", 
  "Failed", "Enabled" };
static void ServiceReport( uint Item, uint Status, uint Report, HANDLE h )
{
  printf( "Service Status: %-9s: %-9s: %-9s: %03d\n", 
    TaskName[Item-1], StatusStr[Status],
    ReportStr[Report/256], Report&0xFF );

  // Example of adding to the DHCP configuration space
  //
  // When using the DHCP client, the client has full control over access
  // to the first 256 entries in the CFGTAG_SYSINFO space. Here, we want
  // to manually add a DNS server to the configuration, but we can only
  // do it once DHCP has finished its programming.
  //
  if( Item == CFGITEM_SERVICE_DHCPCLIENT &&
    Status == CIS_SRV_STATUS_ENABLED &&
    (Report == (NETTOOLS_STAT_RUNNING|DHCPPCODE_IPADD) ||
     Report == (NETTOOLS_STAT_RUNNING|DHCPPCODE_IPRENEW)) )
  {
    IPN IPTmp;
  }
```
5.3.4 Controlling TCP/IP Stack and OS Options via the Configuration

Along with specifying IP addresses, routes, and services, the configuration system allows the programmer to directly manipulate the configuration structures of the OS adaptation layer and the TCP/IP stack. The OS configuration structure is discussed in Chapter 2 of the TCP/IP Stack Programmer's Reference Guide, and the TCP/IP configuration structure is discussed in Chapter 7. The configuration interface to these internal structures is consolidated into a single configuration API specified in Chapter 4.

Although the values in these two configuration structures can be modified directly, adding the parameters to the system configuration is useful for two reasons. First, it provides a consistent API for all network configuration, and second, if the configuration load and save feature is used, these configuration parameters are saved along with the rest of the system configuration.

As a quick example of setting an OS configuration option, the following code makes a change to the debug reporting mechanism. By default, all debug messages generated by the TCP/IP stack are output to the Code Composer output window. However the OS configuration can be adjusted to print only messages of a higher severity level, or to disable the debug messages entirely.

Most of the example applications included with the TCP/IP Stack will raise the threshold of printing debug messages from the "INFO" level to the "WARNING" level. Here is how it appears in the source code:

```c
// We do not want to see debug messages less than WARNINGS
rc = DBG_WARN;
CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGPRINTLEVEL,
            CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&rc, 0 );
```

5.3.5 Saving and Loading a Configuration

Once a configuration is constructed, the application may wish to save it off into non-volatile RAM so that it can be reloaded on the next cold boot. This is especially useful in an embedded system where the configuration can be modified by a user at runtime using a serial cable, telnet, or an HTTP browser.

5.3.5.1 Saving the Configuration

The method of saving the configuration is to convert it to a linear buffer, and then save the linear buffer off to storage. Here is a quick example of a configuration save operation. Note the MyMemorySave() function is assumed to save off the linear buffer into non-volatile storage.

```c
int SaveConfig( HANDLE hCfg )
{
    UINT8  *pBuf;
    int     size;
    // Get the required size to save the configuration
    CfgSave( hCfg, &size, 0 );
    if( size && (pBuf = malloc(size)) )
    {
        CfgSave( hCfg, &size, pBuf );
        MyMemorySave( pBuf, size );
        Free( pBuf );
        return(1);
    }
    return(0);
}
```
5.3.5.2 Loading the Configuration

Once a configuration is saved, it can be loaded from non-volatile memory on startup. For this final NetworkTest() example, assume that another task has created/edited/saved a valid configuration to some storage medium on a previous execution. In our network initialization routine, all that is required is to load the configuration from storage and boot the TCP/IP stack using the current configuration.

For this example, assume that the function MyMemorySize() returns the size of the configuration in a stored linear buffer and that MyMemoryLoad() loads the linear buffer from non-volatile storage.

```c
int NetworkTest()
{
    int rc;
    HANDLE hCfg;
    UINT8 *pBuf;
    Int size;
    // // THIS MUST BE THE ABSOLUTE FIRST THING DONE IN AN APPLICATION!!
    rc = NC_SystemOpen( NC_PRIORITY_LOW, NC_OPMODE_INTERRUPT );
    if( rc )
    {
        printf("NC_SystemOpen Failed (%d)\n",rc);
        for(;;);
    }
    // // First load the linear memory block holding the configuration
    // // Allocate a buffer to hold the information
    size = MyMemorySize();
    if( !size )
        goto main_exit;
    pBuf = malloc( size );
    if( !pBuf )
        goto main_exit;
    // Load from non-volatile storage
    MyMemoryLoad( pBuf, size );
    // // Now create the configuration and load it
    // // Create a new configuration
    hCfg = CfgNew();
    if( !hCfg )
    {
        printf("Unable to create configuration\n");
        free( pBuf );
        goto main_exit;
    }
    // Load the configuration (and then we can free the buffer)
    CfgLoad( hCfg, size, pBuf );
    mmFree( pBuf );
    // // Boot the system using this configuration
    // // We keep booting until the function returns less than 1. This allows
    // // us to have a "reboot" command.
    // do
    { rc = NC_NetStart( hCfg, NetworkStart, NetworkStop, NetworkIPAddr );
      while( rc > 0 );
    // Delete Configuration
    CfgFree( hCfg );
    // Close the OS
    main_exit:
    NC_SystemClose();
}
```
return(0);
}

5.4 Application Debug and Troubleshooting

Although there is certainly no magic formula for debugging a TCP/IP Stack application, a quick run down of some of the potential problem areas using the TCP/IP Stack may help.

Some of these topics are discussed elsewhere in the documentation, but having a quick checklist is useful:

5.4.1 Most Common Problems

One of the most common support requests for the TCP/IP Stack deals with the inability to either send or receive network packets. This may also take the form of dropping packets or general poor performance. There are many causes for this type of behavior. For potential “scheduling issues”, see Section 5.2.1. It is also recommended that application programmers fully understand the workings of the NETCTRL module. For this, see Chapter 6.

Here is a quick list.

All socket calls return “error” (-1)

• Make sure there is a call to \texttt{tcpOpenSession()} in the task before it uses sockets, and a call to \texttt{tcpCloseSession()} when the task terminates.

No link indication, or will not re-link when cable is disconnected and reconnected.

• Make sure there is a PRD function in your DSP/BIOS configuration that is calling the driver function \texttt{llTimerTick()} every 100 ms.

Not receiving any packets – ever

• When “polling” for data by making \texttt{recv()}, \texttt{fdPoll()}, or \texttt{fdSelect()} calls in a non-blocking fashion, make sure you do not have any scheduling issues. When the NETCTRL scheduler is running in low priority, network applications are not allowed to “poll” without blocking. Try running the scheduler in high priority (via \texttt{NC\_SystemOpen()}).
• The TCP/IP Stack assumes there is some L2 cache. If the DSP is configured to “all internal memory” with nothing left for L2 cache, the TCP/IP Stack drivers will not function properly.
• Try running the Ethernet driver confidence test as described in Section 2.3.1.

Performance is “sluggish”. Very slow ping response.

• Make sure there is a PRD function in your DSP/BIOS configuration that is calling the driver function \texttt{llTimerTick()} every 100 ms.
• If porting an Ethernet driver and running NETCTRL in “interrupt” mode, make sure your device is correctly detecting interrupts. Make sure the interrupt polarity is correct.

UDP application drops packets on send() calls.

• If sending to a new IP address, the very first send may be held up in the ARP layer while the stack determines the MAC address for the packet destination. While in this mode, subsequent sends will be discarded.
• When using UDP and sending multiple packets at once, make sure you have plenty of packet buffers available (see Section 7.3.1).
• Verify you do not have any scheduling issues. Try running the scheduler in high priority (via \texttt{NC\_SystemOpen()}).

UDP application drops packets on recv() calls.

• Make sure you have plenty of packet buffers available (see Section 7.3.1).
• Make sure the packet threshold for UDP is high enough to hold all UDP data received in between calls to \texttt{recv()} (see CFGITEM\_IP\_SOCKUDPRXLIMITER in the TCP/IP Stack Programmer’s Reference Guide).
• Verify you do not have any scheduling issues. Try running the scheduler in high priority (via \texttt{NC\_SystemOpen()}).
• It is possible that packets are being dropped by the Ethernet device driver. Some device drivers have...
adjustable RX queue depths, while others do not. Refer to the source code of your Ethernet device driver for more details (device driver source code is provided in TCP/IP Stack Porting Kit).

**In General**

- Do not try to “tune” the PRD function frequency. Make sure it calls *llTimerTick()* every 100 ms.
- Watch for “out of memory” conditions. These can be detected by the return from some functions, but will also print out “warning” messages when the messages are enabled. These messages will contain the acronym “OOM” for “out of memory”. (Out of memory conditions can be caused by many things, but the most common cause in the TCP/IP Stack is when TCP sockets are created and closed very quickly without using the SO_LINGER socket option. This puts many sockets in the TCP “timewait” state, exhausting scratchpad memory. The solution is the use the SO_LINGER socket option.)

### 5.4.2 Controlling Debug Messages

Most of the text messages generated by a network application will come from the application. However, it is possible that the network stack will generate debug messages.

The TCP/IP Stack includes its own debug message system. This system can be ported to behave in any manner desired, by using the TCP/IP Stack Porting Kit, but by default, debug messages are printed to the debugger using an internal *printf()* function.

Debug messages also include an associated severity level. These levels are DBG_INFO, DBG_WARN, and DBG_ERROR. The severity level is used for two purposes. First, it determines whether or not the debug message will be printed, and second, it determines whether or not the debug message will cause the TCP/IP stack to shutdown.

By default, all debug messages are printed, and messages with a level of DBG_ERROR will cause a stack shutdown. This behavior can be modified by using the OS configuration structure or through the system configuration. Although this information is contained in the *TCP/IP Stack Programmer's Reference Guide*, example code to change the printing and system shutdown behavior of the debug system is supplied in this User's Guide in *Section 5.3.4* and *Section 5.3.1.6* respectively.

### 5.4.3 Interpreting Debug Messages

The following is a list of some of the debug messages that may occur during stack operation, along with the most common associated cause.

#### 5.4.3.1 TCP: Retransmit Timeout - Level DBG_INFO

This message is generated by TCP when it has sent a packet of data to a network peer, and the peer has not replied in the expected amount of time. This can be just about anything; the peer has gone down, the network is busy, the network packet was dropped or corrupted, etc..

#### 5.4.3.2 FunctionName: Buffer OOM - Level DBG_WARN

This message will be generated by some modules when unexpected out of memory conditions occur. The stack has an internal resource recovery routine to help deal with these situations; however, a significant number of these messages may also indicate that there is not enough large block memory available, or that there is a memory leak. See the notes on the memory manager reports in this section for more details.

#### 5.4.3.3 mmFree: Double Free - Level DBG_WARN

A double free message occurs when the mmFree() function is called on a block of memory that was not marked as allocated. This can be caused by physically calling mmFree() twice for the same memory, but more commonly is caused by memory corruption. See the notes on memory corruption in this section for possible causes.
5.4.3.4 FunctionName: HTYPE nnnn - Level DBG_ERROR

This message will be generated only by the strong checking version of the stack. It is caused when a handle is passed to a function that is not of the proper handle type. Since the object oriented nature of the stack is hidden from the network applications writer, this error should never occur. If it is not caused by the attempt to call internal stack functions, then it is most likely the result of memory corruption. See the notes on memory corruption in this section for possible causes.

5.4.3.5 mmAlloc: PIT ???? Sync - Level DBG_ERROR

This message is generated by the scratch memory allocation system. PIT is an acronym for "page information table". Table synchronization errors can only be caused by memory corruption. See the notes on memory corruption in this section for possible causes.

5.4.3.6 PBM_enq: Invalid Packet - Level DBG_ERROR

This message is generated by the packet buffer manager (PBM) module driver in the OS adaptation layer. When the PBM module initially allocates its packet buffer pool, it marks each packet buffer with a magic number. During normal operation, packets are "pushed" and "popped" to and from various queues. On each push operation, the packet's magic number is checked. When the magic number is invalid, this message results. It is possible for an invalid packet to be introduced into the system when using the "non copy" sockets API extensions, but the vastly more common cause is memory corruption. See the notes on memory corruption in this section for possible causes.

5.4.4 Memory Corruption

The words "memory corruption" come up frequently when diagnosing TCP/IP Stack debug messages. This is because it is very easy to corrupt memory on cache devices. Most of the example programs included in the TCP/IP Stack run using full L2 cache. In this mode, any read or write access to the internal memory range of the CPU can cause cache corruption and hence cause memory corruption. Since the internal memory range starts at address 0x00000000, a NULL pointer can cause havoc when using full cache.

The easiest way to check to see if corruption is being caused by a NULL pointer is to change the cache mode to use less cache. When there is some internal memory available, reads or writes to address 0x0 will not cause cache corruption (of course the application still may not work, but the error messages should go away).

Another way to track down any kind of cache corruption is to break on CPU reads or writes to the entire cache range. Code Composer Studio has the ability to trap reads or writes to a range of memory, but both can not be checked simultaneously. Therefore, a couple of trials may be necessary.

Of course, it is possible that the memory corruption has nothing to do with the stack. It could be a wild pointer. However, since corrupting the cache can corrupt memory throughout the system, the cache is the first place to start.

5.4.5 Program Lockups

Just a quick note on program lockups. Most lockup conditions are caused by insufficient task stack sizes. For example, when writing an HTTP CGI function, the CGI function task thread has only about 5000 bytes of total task stack. Therefore, using large amounts of stack is a very bad idea. In general, do not do:

myTask()
{
    char TempBuffer[2000];
    myFun( TempBuffer );
}

but rather do:

myTask()
{
    char *pTempBuf;
    pTempBuf = MEM_alloc( 0, 2000, 0 );
    if( pTempBuf != MEM_ILLEGAL )
}
mmCheck — Generate Memory Manager Report

```c
{
    myFun ( pTempBuf );
    MEM_free( pTempBuf, 2000 );
}
```

If course calling a memory allocation function is too much of a speed overhead, consider using an external buffer. Of course this is just an example. A little forethought should eliminate all possible stack overflow conditions, and eliminate the possibility of program lockups from this condition.

5.4.6 Memory Management Reports

The memory manager used to manage scratch memory in the TCP/IP Stack has a built in reporting system. It tracks the use of scratch memory very closely (calls to `mmAlloc()` and `mmFree()`), and also tracks calls to the large block memory allocated (calls to `mmBulkAlloc()` and `mmBulkFree()`). Note that the "bulk" allocation functions simply call `malloc()` and `free()`. This behavior can be altered by adjusting the memory manager included in the TCP/IP Stack Porting Kit.

The memory report is shown below. It lists the max number of blocks allocated per size bucket, the number of calls to malloc and free, and a list of allocated memory. An example report is shown below:

```
48:48 ( 75%) 18:96 ( 56%)  8:128 ( 33%)  28:256 ( 77%)
1:512 ( 16%)  0:1536  0:3072
(21504/46080 mmAlloc: 61347036/0/61346947, mmBulk: 25/0/17)
```

1 blocks allocated in 512 byte page
38 blocks allocated in 48 byte page
18 blocks allocated in 96 byte page
8 blocks allocated in 128 byte page
12 blocks allocated in 256 byte page
12 blocks allocated in 256 byte page

Here, the entry "18:96 (56%)" means that at most, 18 blocks were allocated in the 96 byte bucket. The page size on the memory manager is 3072, so 56% of a page was used.

The entry "21504/46080" means that at most 21,504 bytes were allocated, with a total of 46,080 bytes available.

The entry "mmAlloc: 61347036/0/61346947" means that 61,347,036 calls were made to `mmAlloc()`, of which 0 failed, and 61,346,947 calls were made to `mmFree()`. Note that at any time, the call to mmAlloc plus the failures must equal the calls to mmFree plus any outstanding allocations. Therefore, on a final report were the report is "mmAlloc: n1/n2/n3", n1+n2 should equal n3. If not, there is a memory leak.

There are a couple of ways to obtain a memory report when using the telnet console program included with most of the example applications. The console 'mem' command will print out a current report, but more importantly, the console 'shutdown' command will shutdown the stack and print out a final report. If all network applications are created and destroyed according to the specifications in this document, there should be no memory leaks detected in the final report.

The function called to get a memory report is defined below.

5.4.6.1 mmCheck — Generate Memory Manager Report

<table>
<thead>
<tr>
<th>mmCheck</th>
<th>Generate Memory Manager Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntax</td>
<td>void _mmCheck( uint CallMode, int (*pPrn)(const char *,...) );</td>
</tr>
<tr>
<td>Parameters</td>
<td>CallMode</td>
</tr>
<tr>
<td></td>
<td>pPrn</td>
</tr>
</tbody>
</table>
# mmCheck — Generate Memory Manager Report

## Description

Prints out a memory report to the printf() compatible function pointed to by pPrn. The type of report printed is determined by the value of CallMode. The reporting function has the option of printing out memory block "ID"s. This means that the first uint sized field in the memory block of each allocated block is printed in the report. This is a useful option when the first field of allocated memory is used to store an object handle type, or some other unique identifier.

## Call Mode

Can be set to one of the following:

<table>
<thead>
<tr>
<th>Call Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MMCHECK_MAP</td>
<td>Map out allocated memory, but do not dump ID's</td>
</tr>
<tr>
<td>MMCHECK_DUMP</td>
<td>Dump allocated block IDs</td>
</tr>
<tr>
<td>MMCHECK_SHUTDOWN</td>
<td>Dump allocated block IDs &amp; free scratchpad memory</td>
</tr>
</tbody>
</table>

Note: Do not attempt to use any mmAlloc() functions after requesting a MMCHECK_SHUTDOWN report!

## Returns

None
This chapter describes the network control functions.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1 Introduction to NETCTRL Source</td>
<td>56</td>
</tr>
<tr>
<td>6.2 NETCTRL Scheduler</td>
<td>57</td>
</tr>
<tr>
<td>6.3 Disabling &quot;On-Demand&quot; Services</td>
<td>62</td>
</tr>
</tbody>
</table>
6.1 Introduction to NETCTRL Source

6.1.1 History

The origin of the NETCTRL module was a recommended initialization and scheduling method to execute the TCP/IP stack. Although mostly straightforward, this code became somewhat standard. Eventually, it was separated out into what is today’s NETCTRL library.

The NETCTRL module is the heart of the TCP/IP stack because it connects the HAL and the OS adaptation layer to the TCP/IP stack. It controls both initialization and how events are scheduled for execution within the stack. Understanding how the NETCTRL module works is important to being able to tune your DSP networking application for ideal performance.

6.1.2 NETCTRL Source Files

Source code to the NETCTRL library is comprised of two C files located in the \SRC\NETCTRL directory:

- **NETCTRL.C**  Network Control (Initialization and Scheduling) Module
- **NETSRV.C**  Configuration service module (system configuration service provider)

There are two include files associated with NETCTRL in the \INC\NETCTRL directory:

- **NETCTRL.H**  Interface specification to NETCTRL
- **NETSRV.H**  Interface specification to NETSRV

6.1.3 Main Functions

The NETCTRL.C source module contains source code for all the functions with the "NC_" prefix. The function of the NETCTRL module has three basic parts.

The first function of NETCTRL.C is to perform the system initialization and shutdown that is necessary before calling any other stack functions. These functions are declared as NC_SystemOpen() and NC_SystemClose().

The second function of NETCTRL.C is to perform the driver environment initialization and configuration bootstrap necessary to start the stack functionality. This startup function and its shutdown counterpart are declared as NC_NetStart() and NC_NetStop().

The final function of NETCTRL.C, which is hidden from the caller, is to implement the stack’s event scheduling, which is the heart of the stack's operation.

The NETSRV.C module contains the code that is used to boot all the services on the stack. It is this code that takes what is stored in the stack’s configuration and implements the necessary stack functions to keep the configuration current. When an active item in the configuration is changed, there is code in the NETSRV module to execute that change in the TCP/IP stack.

6.1.4 Additional Functions

There are some additional NETCTRL functions that are not documented in the Programmer’s Guide. These functions are NC_BootComplete() and NC_IPUpdate(). They are both called from the NETSRV module.

The NC_NetStart() function initiates the configuration boot process by creating a boot thread with an entry
point of \textit{NS\_BootTask()} (from NETSRV.C). The \textit{NC\_BootComplete()} function is called by the configuration boot thread when the configuration boot is complete. It is used to signal to NETCTRL that it can now call the \textit{NetworkStart()} application callback that was passed to NC\_NetStart() by the application. On return from \textit{NC\_BootComplete()}, the boot thread is terminated. Therefore, the application programmer is allowed to (but not encouraged to) take control of the \textit{NetworkStart()} callback thread.

The IP address update function is called by NETSRV when an address is added to or removed from the system. It is this function that will then call the NetworkIPAddr() application callback that was originally passed to \textit{NC\_NetStart()}.

### 6.1.5 Booting and Scheduling

Section 5.3 discussed using the network control \textit{(NETCTRL.LIB)} module. This section examines the internal source code of the main NETCTRL module and the operation of the event scheduler.

The stack event scheduler is the routine that actually calls the stack to process packet and timer events. The scheduler is called from within \textit{NC\_NetStart()} and does not return until the stack is being shut down. This is why the \textit{NC\_NetStart()} function does not return to the application until the system is shutdown and the scheduler terminates.

The basic flow of \textit{NC\_NetStart()} is as follows:

\begin{verbatim}
NC\_NetStart()
{
   Initialize\_Devices();
   CreateConfigurationBootThread();NetScheduler();
   CloseConfiguration();
   CloseDevices();
}
\end{verbatim}

Out of the functional stages for \textit{NC\_NetStart()} listed above, the two that will be of the most concern to applications programmers are the creation of the boot thread, and the implementation of the network event scheduler.

The boot thread is handled by a second C module in the NETCTRL library named NETSRV.C. This name is an abbreviation for “Network Service Manager”. The NETSRV module hooks into the configuration system as a configuration service provider. The configuration system module does nothing in itself, it is just an active database. It is the network service module that does the work of turning configuration entries into actual TCP/IP stack objects. The service module can be altered to fit a particular need. This would most likely involved the creation of custom configuration tags for the configuration system. However, a full understanding of the code in NETSRV would require a basic understanding of nearly all the API functions discussed in the \textit{TCP/IP Stack Programmer's Reference Guide}.

An application programmer should be most concerned about the \textit{NetScheduler()} function. It is this scheduler that “runs” the TCP/IP stack. It looks for events that need to be processed by the TCP/IP stack, and it performs the work necessary to do so.

### 6.2 NETCTRL Scheduler

#### 6.2.1 Scheduler Overview

The NETCTRL scheduler code is an infinite loop function named \textit{NetScheduler()} and appears at the end of the source file NETCTRL.C. It looks for activity events from the low level device drivers, and acts when events are detected. The loop terminates when a static variable is set through an outside call to \textit{NC\_NetStop()}.

Although the TCP/IP stack provides a reentrant environment, the core of the stack is not reentrant. Portions of the code must be protected from access by reentrant calls. Instead of using critical sections that block out all other task execution, the software defines an operating mode called “kernel mode”. Kernel mode is defined such that only one task may be in kernel mode at any given time. It does nothing
to prevent tasks from running that do not use the TCP/IP stack. This provides protection for the stack, without affecting the execution of unrelated code. There are two functions defined to enter and exit kernel mode, \texttt{llEnter()} and \texttt{llExit()}. They are part of the OS adaptation layer, and are discussed in more detail in Section 7.2.3. In short, \texttt{llEnter()} must be called before calling into the stack, and \texttt{llExit()} must be called when done calling stack functions.

The basic flow of the scheduler loop can be summarized by this pseudo code:

```c
static void NetScheduler()
{
    SetSchedulingPriority();
    while( !NetHaltFlag )
    {
        WaitOrPollForEvents();
        ServiceDeviceDrivers();

        // Process current events in Kernel Mode
        if( StackEvents )
        {
            // Enter Kernel Mode
            llEnter();
            ServiceStackEvents();

            // Exit Kernel Mode
            llExit();
        }
    }
}
```

The sections that follow address each of the highlighted functions in turn. Note that the code continues to run until the NetHaltFlag is set. This flag is set when an application calls the \texttt{NC\_NetStop()} function.

### 6.2.2 Scheduling Options

There are three basic ways to run the scheduler. They can be viewed as three operating modes:

1. Scheduler runs at low priority and only when there are network events to process
2. Scheduler runs continuously at low priority, "polling" the device drivers for events
3. Scheduler runs a high priority, but only when there are network events to process

The best way to run the scheduler will depend on the application and system architecture.

Mode 1 is the most efficient way to run the TCP/IP stack. Here, the scheduler loop runs at a low priority. This allows applications that potentially have real-time requirements to have priority over networking where the real-time restrictions are more lax. In addition, the scheduling loop only runs when there is network related activity, therefore, a standard DSP/BIOS idle loop can also be used.

Mode 2 is used when the device drivers are prevented from using interrupts. This is best for real-time tasks, but worst for network performance. Since the scheduler thread runs continuously, it also prevents the use of a DSP/BIOS idle loop. This is the mode that NETCTRL must use when using a device driver that requires polling.

Mode 3 is the most “Unix-like” environment. Here, the network scheduler task runs at a higher priority than any other networking task in the system. The stack will run whenever new network related events are detected, pre-empting other tasks from potentially using the stack. This is the best method for keeping the networking environment up to date without placing restrictions on how network applications are written.

Setting priority and polling or interrupt driven scheduling is done when the application first calls \texttt{NC\_SystemOpen()}. This is discussed further in Section 5.3.1.2 and in the TCP/IP Stack Programmer's Reference Guide.
6.2.3 Scheduler Thread Priority

The first lines of the actual implementation of NetScheduler() include the following code:

```c
// Set the scheduler priority
TKS_setpri(TSK_self(), SchedulerPriority);
```

This code changes the priority of the task thread that calls into NC_NetStart(). This is done so that there is a single control point to set the scheduler priority. The priority used is that which was passed to the NC_SystemOpen() function. This is discussed further in Section 5.3.1.2 and in the TCP/IP Stack Programmer’s Reference Guide.

The scheduler priority (relative to network application thread priority) greatly affects how network applications can be programmed. For example, when running the scheduler in low priority, a network application can not poll for data by continuously calling recv() in a non-blocking fashion. This is because if the application thread never blocks, the network scheduler thread never runs, and incoming packets are never processed by the TCP/IP stack.

6.2.4 Tracking Events with STKEVENT

As previously mentioned, the NETCTRL module is the interface between the stack and the device drivers in the HAL layer. In older versions of the TCP/IP Stack, device drivers signaled the NETCTRL module through a global semaphore. In order to improve this process slightly, the simple semaphore has been encapsulated into an object called STKEVENT.

From the device driver’s point of view, this event object is a handle that is passed to a function called STKEVENT_signal(). In reality, this function is only a MACRO that operates on a structure of type STKEVENT. The NETCTRL module operates directly on this structure. The STKEVENT structure is defined as follows:

```c
// Stack Event Object
typedef struct _stkevent {
    SEM_Handle hSemEvent;
    uint EventCodes[3];
} STKEVENT;
```

```c
#define STKEVENT_TIMER 0
#define STKEVENTETHERNET 1
#define STKEVENT_SERIAL 2
```

There are two parts to the structure, a semaphore handle and an array of events. Each driver signals an event by setting a flag in the EventCode[] array for its event type, and then optionally signaling the event semaphore. The semaphore is only signaled when the driver detects an interrupt condition. If the event is detected during driver polling (either periodic polling or constant in the case of a polling only driver), the event is set, but the semaphore is not signaled.

Note that in a polling environment, the semaphore handle hSemEvent is NULL.

The NETCTRL module creates a private instance of the STKEVENT structure that it passes to device drivers as a handle of type STKEVENT_Handle. The private instance that is operated on directly by NETCTRL is declared as:

```c
// Static Event Object
static STKEVENT stkEvent;
```

In the full source to NetScheduler() that follows, the STKEVENT structure is referred to by its instance stkEvent.

6.2.5 Scheduler Loop Source Code

The code for the example scheduler implementation included in the TCP/IP Stack is shown below. This implementation fleshes out the pseudo code shown in Section 6.2.1, using the methods and objects described in this section. In this code, the number of serial port devices and Ethernet devices is passed in as calling arguments. This device count is obtained from the device drivers when they are asked to enumerate their physical devices.
```c
#define FLAG_EVENT_TIMER   1
#define FLAG_EVENT_ETHERNET 2
#define FLAG_EVENT_SERIAL  4

static void NetScheduler( uint const SerialCnt, uint const EtherCnt )
{
    register int fEvents;

    // Set the scheduler priority
    TSK_setpri( TSK_self(), SchedulerPriority );

    while( !NetHaltFlag )
    {
        if( stkEvent.hSemEvent )
        {
            SEM_pend( stkEvent.hSemEvent, SYS_FOREVER );
            SEM_reset( stkEvent.hSemEvent, 0 );
        }

        // Clear our event flags
        fEvents = 0;
        // First we do driver polling. This is done from outside
        // kernel mode since pure "polling" drivers can not spend
        // 100% of their time in kernel mode.
        // Check for a timer event and flag it
        if( stkEvent.EventCodes[STKEVENT_TIMER] )
        {
            stkEvent.EventCodes[STKEVENT_TIMER] = 0;
            fEvents |= FLAG_EVENT_TIMER;
        }

        // Poll only once every timer event for ISR based drivers,
        // and continuously for polling drivers. Note that "fEvents"
        // can only be set to FLAG_EVENT_TIMER at this point.
        if( fEvents || !stkEvent.hSemEvent )
        {
            // Poll Ethernet Packet Devices
            if( EtherCnt )
                _llPacketServiceCheck( fEvents );
            // Poll Serial Port Devices
            if( SerialCnt )
                _llSerialServiceCheck( fEvents );
        }

        // Note we check for Ethernet and Serial events after
        // polling since the ServiceCheck() functions may
        // have passively set them.
        //
        // Check for a Ethernet event and flag it
        if(EtherCnt && stkEvent.EventCodes[STKEVENT_ETHERNET] )
        {
            // We call service check on an event to allow the
            // driver to do any processing outside of kernel
            // mode that it requires, but don't call it if we
            // already called it due to a timer event.
            if( !(fEvents & FLAG_EVENT_TIMER) )
                _llPacketServiceCheck( 0 );

            // Clear the event and record it in our flags
```
stkEvent.EventCodes[STKEVENT ETHERNET] = 0;
fEvents |= FLAG EVENT ETHERNET;
}

// Check for a Serial event and flag it
if(SerialCnt & stkEvent EventCodes[STKEVENT SERIAL]) {
    // We call service check on an event to allow the
    // driver to do any processing outside of kernel
    // mode that it requires, but don’t call it if we
    // already called it due to a timer event.
    if( !(fEvents & FLAG EVENT TIMER) )
        llSerialServiceCheck( 0 );

    // Clear the event and record it in our flags
    stkEvent.EventCodes[STKEVENT SERIAL] = 0;
    fEvents &= FLAG EVENT SERIAL;
}

// Process current events in Kernel Mode
if( fEvents )
{
    // Enter Kernel Mode
    llEnter();

    // Check for timer event
    if( fEvents & FLAG EVENT TIMER )
        ExecTimer();

    // Check for packet event
    if( fEvents & FLAG EVENT ETHERNET )
        llPacketService();

    // Check for serial port event
    if( fEvents & FLAG EVENT SERIAL )
        llSerialService();

    // Exit Kernel Mode
    llExit();
}
Disabling "On-Demand" Services

Services are specified via the configuration system at runtime, and can be executed "on-demand". This is a nice feature for allowing users to alter the configuration without rebuilding the network application, but has a liability in that any service that can be invoked via the configuration is always linked into the system executable. This increases the footprint of the system software in order to support services that may never be used.

In order to cope with this problem, some #define declarations have been included in the source file `\INC\NETCTRL\NETSRV.H`. These statements are as follows:

```c
#elif
// The following #define statements are used to determine if certain service
// entry-points are linked into the executable. So if a service is not
// going to be used, set the corresponding #define to zero. When set
// to zero, the service is unavailable.
#endif
#define NETSRV_ENABLE_TELNET 1
#define NETSRV_ENABLE_HTTP 1
#define NETSRV_ENABLE_NAT 0
#define NETSRV_ENABLE_DHCPCLIENT 1
#define NETSRV_ENABLE_DHCPSERVER 1
#define NETSRV_ENABLE_DNSSERVER 1
```

By setting any of the above to "0" and rebuilding the NETCTRL library (which consists of two files), the individual services can be purged from the executable. These services will then be unavailable via the configuration system.
The OS adaptation layer controls how the TCP/IP Stack uses DSP/BIOS resources. This includes tasks, semaphores, memory and printing. Anything OS related can be adjusted here. This chapter also includes a history of the OS adaptation layer source, and describes the files which comprise the source code.

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 Introduction to OS Source</td>
<td>64</td>
</tr>
<tr>
<td>7.2 Task Thread Abstraction - TASK.C</td>
<td>64</td>
</tr>
<tr>
<td>7.3 Packer Buffer Manager - PBM.C</td>
<td>66</td>
</tr>
<tr>
<td>7.4 Memory Allocation System - MEM.C</td>
<td>67</td>
</tr>
<tr>
<td>7.5 Embedded File System - EFS.C</td>
<td>68</td>
</tr>
<tr>
<td>7.6 General OS Support - OSSYS.C</td>
<td>68</td>
</tr>
<tr>
<td>7.7 Print Functions - MINIPRINTF.C</td>
<td>68</td>
</tr>
</tbody>
</table>
7.1 Introduction to OS Source

7.1.1 History

One reason the TCP/IP Stack contains an OS adaptation layer is so that applications that are coded to the abstraction can be executed in any environment to which the abstraction is ported. For DSP centric applications, cross-platform portability is not usually practical nor required. DSP programmers prefer to use DSP/BIOS and take advantage of the support features provided by Code Composer Studio.

For most of the OS adaptation API, the abstraction functions are converted to direct DSP/BIOS calls through the use of #define macros. However, there are some additions and refinements made at the OS layer that tend to vary slightly from one DSP/BIOS based system to the next. For these refinements, external OS abstraction functions are required. This allows the system programmer to adapt the OS layer to meet the particular system requirements of their DSP/BIOS based environment.

This section covers the OS functions which may need to be adjusted. The OS source code referenced in this section is found in the SRC\OS directory. Printing functions have been separated into a different directory and library; this way, a user who needs to use the bigger size, more feature rich, printing APIs provided with the standard RTS library, can easily replace the slim APIs in MiniPrintf.LIB with the RTS API.

7.1.2 Source Files

Source code to the OS library is comprised of several files, located in the SRC\OS directory:

- TASK.C: Task thread abstraction
- PBM.C: Packet Buffer Manager
- MEM.C: Memory allocation and memory copy functions
- EFS.C: Embedded (RAM based) File System
- OSSYS.C: Additional OS support (debug logging, stricmp() function)
- OSCRIT.S62: DSP critical section and cache control

The printing functionality is contained in the following file, located in the SRC\MiniPrintf directory:

- MINIPRINTF.C: Basic printf() functions

Two additional include files are located in the INC\OS directory:

- OSIF.H: Interface specifications to the adaptation library
- OSKERN.H: Semi-private declarations for use by functions like NETCTRL

7.2 Task Thread Abstraction - TASK.C

The TASK.C module contains a subset of the task abstraction API documented in the TCP/IP Stack Programmer's Reference Guide. It also contains the source code to the stack's exclusion method functions: ilEnter() and ilExit(). The latter are discussed in Section 7.2.3 of this document.

Most of the task and semaphore functions defined in the Programmer's Guide are in actuality macros which call DSP/BIOS. These macros are defined in INC\OS\OSIF.H. The functions that do not directly map to DSP/BIOS are listed here.
7.2.1 TaskSetEnv() and TaskGetEnv()

The set environment and get environment functions are supplied in TASK.C so that they can be ported to the DSP/BiOS based system in such a way that they will not conflict with other system use of the TSK_setenv() and TSK_getenv() functions.

The ability to associate a data structure with a task thread is essential for the stack library. The problem with the implementation in DSP/BiOS is that it only allows a single entity to assign this environment pointer. The result is that any use of TSK_setenv() or TSK_getenv() by a third party conflicts with the stack software.

The implementation of the TASK.C supplied in the TCP/IP Stack gets around this limitation by using the DSP/BiOS task HOOK object. The HOOK object allows multiple entities to hook into DSP/BiOS task creation – including the ability to expand the environment. For more information, see the DSP/BiOS documentation.

**Note:** In the TASK.C module provided, the TaskSetEnv() and TaskGetEnv() functions do not implement the slot calling parameter. All internal stack functions use slot zero. The additional slots were originally intended to be used by applications, but under DSP/BiOS, applications must use TSK_setenv() and TSK_getenv() functions. Therefore, the DSP/BiOS based implementation of TaskSetEnv() and TaskGetEnv() is simplified.

7.2.2 TaskCreate(), TaskExit(), and TaskDestroy()

The create, exit and destroy functions all call their DSP/BiOS equivalents. However, they are provided in this module as slightly tuned hook functions.

The main reason that these functions are not defined as macros is so that the TaskExit() function could be tuned to perform its own cleanup. The TSK_exit() function provided in DSP/BIOS does not delete the task that has exited. In fact, a task can only be deleted by another task - it cannot delete itself. Some system software copies this in a garbage collection thread, but the TCP/IP stack library does not require nor implement such a thread. This was done intentionally so that the TCP/IP stack would not conflict with any IDLE thread provided by the system programmer.

In order to allow TaskExit() to clean up after itself, some additional code is called to track the last thread to call the TaskExit() function. In that cleanup function, the thread that previously called TaskExit() is deleted. Therefore, thread deletion is always "1 behind" thread exit, and no thread deletes itself.

7.2.3 Choosing the llEnter()/llExit() Exclusion Method

Although the TCP/IP stack provides a reentrant environment, the core of the stack is not reentrant. Portions of the code can be protected from access by reentrant calls. Instead of using critical sections that block out all other task execution, the software defines an operating mode called "kernel mode". Kernel mode is defined such that only one task may be in kernel mode at any given time. It does nothing to prevent tasks from running that do not use the TCP/IP stack. This provides protection for the stack software, without affecting the execution of unrelated code.

The llEnter() and llExit() functions are used throughout the stack code to enter and exit kernel mode, and provide code exclusion without using critical sectioning. They are equivalent to the splhigh()/splx() Unix functions and their multiple cousins.

There are two example implementations of the llEnter() and llExit() functions included in the TCP/IP Stack. The example implementations provide exclusion through task priority or by using semaphores. Source code to both implementations is included in the task abstraction source file: SRC\OS\TASK.C

One method of exclusion is the priority method. Here, the task that calls llEnter() is boosted to a priority level of OS_TASKPRIKERN, which guarantees that it will not be pre-empted since it is impossible for another task to be running (all tasks that can possibly call into the stack have a lower priority level). The stack is coded so that a task at the kernel mode priority level will never block. When llExit() is called, the task's original priority is restored. Note that time critical tasks can be assigned a priority higher than OS_TASKPRIKERN, but they are not allowed to call into the TCP/IP stack.
An alternate implementation of the enter and exit functions uses a semaphore with an initial count of 1. When \texttt{IIEnter()} is called, the task calls a \texttt{pend} operation on the semaphore. If some other task is currently executing in kernel mode, the new task will \texttt{pend} until \texttt{IILookup()} is called by the original task. A call to \texttt{IILookup()} results in a post operation which frees up one task to enter the stack. This form of the function pair is safer than the priority method, but may also be slower. In general, semaphore operations are a little slower than task priority changes. However this method also has its advantages. The main advantage with the semaphore method is that tasks can be assigned priority levels more freely. There is no need to restrict task priority or be concerned if a high priority task is going to call into the TCP/IP stack.

By altering the \#if statements around the two implementations, the system developer can choose to use either implementation.

### 7.3 Packer Buffer Manager - PBM.C

The Packet Buffer Manager (PBM) is charged with managing all the packet buffers in the system. Packet buffers are used by the TCP/IP stack and device drivers to carry networking packet data. The PBM programming abstraction is discussed in the \textit{TCP/IP Stack Programmer’s Reference Guide}. This section discusses the implementation provided in the TCP/IP Stack.

#### 7.3.1 Packet Buffer Pool

There are two \#define statements at the top of the PBM.C module:

```
#define PKT_NUM_FRAMEBUF 48
#define PKT_SIZE_FRAMEBUF 1664
```

These determine the number of packet buffers in the main buffer pool, and the size of the buffer. Note that when the memory is declared, it is placed on a cache aligned boundary. Also, each packet buffer must be an even number of cache lines in size so that it can be reliably flushed without the risk of conflicting with other buffers.

Why use a 1664 byte packet buffer? In a very simple Ethernet system, the max size of the frame buffer would be 1518, but a few things happen in the TCP/IP Stack environment to change that.

First, all devices use a standard packet header size from the start of the packet buffer to the IP header. The result is that any packet can be routed to any device without altering the location of the IP header. The standard size used is 22 bytes, which is the size of a PPPoE header plus the standard Ethernet header.

Next, the Macronix Ethernet supported by the TCP/IP Stack MAC transfers data in 16 byte bursts. However, the first four bytes of the first transfer is 4 bytes of status, leaving only 12 bytes of data. A little math reveals the Macronix writes 1532 bytes into the packet buffer on a 1518 byte frame.

Taking the standard 1532 bytes required by Macronix, and adding an additional 8 byte pad for the standard header (22 – standard 14 byte Ethernet) gives 1540 bytes which rounds to 1664 when expanded to fill a full L2 cache line.

When not using the Macronix (LogicIO) Ethernet, \texttt{PKT_SIZE_FRAMEBUF} can be set to 1536.

#### 7.3.2 Packet Buffer Allocation Method

The basic method of buffer allocation is the buffer pool. Buffers are allocated when the \texttt{PBM_alloc()} function is called. This function can be called at interrupt time, so care must be taken to ensure only non-blocking calls are made as a result. However, only device drivers can make calls from an ISR and device drivers will never ask for a buffer larger than \texttt{PKT_SIZE_FRAMEBUF}. Therefore, the fallback method for allocating larger buffers can technically make blocking calls, although the implementation included in the TCP/IP Stack does not make blocking calls under any circumstance.

The basic method of allocation is to check the size. When the size is less than or equal to
PKT_SIZE_FRAMEBUF, then the packet buffer is obtained off the free queue. If there are no free packet buffers on the queue, the function returns NULL. Note that the PBM module could be modified to grow the free pool or use memory allocation as a fallback, but any buffer supplied as a result of a request with the size less than or equal to PKT_SIZE_FRAMEBUF, must adhere to the cache line restrictions outlined in the previous section.

For packet buffers larger than PKT_SIZE_FRAMEBUF, standard memory can be used. These allocation requests are only made for re-assembling large IP packets. The resulting packet can not be submitted to a hardware device without being fragmented. Therefore, the packet buffer does not need to be compatible for hardware transmission.

7.3.3 Referenced Route Handles

One of the fields in the PBM structure is a referenced handle to a route used to route a packet to its final destination. The PBM module must be aware of this handle in two circumstances: freeing a packet buffer and copying a packet buffer.

When packet buffer is freed by calling PBM_free(), the PBM module must check for a route handle held by the packet buffer, and dereference the handle if it exists. For example:

```c
if( pPkt->hRoute )
{
    RtDeRef( pPkt->hRoute );
    pPkt->hRoute = 0;
}
```

As noted in the source code to PBM.C, the function RtDeRef() can only be called from kernel mode. However, instead of defining two versions of the PBM_free() function, the PBM module relies on the fact that device drivers are never given packet buffers containing routes. Therefore, any call to PBM_free() where the buffer contains a route, must have been called from within kernel mode. It is, therefore, safe to call RtDeRef().

When a packet buffer is copied with PBM_copy(), all the information about the packet is also copied. This information may include a referenced route handle. If the handle to a route is copied in the process of copying the packet buffer, then a reference to that handle must also be added by calling the RtRef() function. The PBM module does not need to worry about kernel mode for the same reason as it did not with PBM_free().

7.4 Memory Allocation System - MEM.C

The memory allocation system consists of allocation functions for small blocks of memory, large blocks, and for initializing and copying memory blocks. The API definitions for the files contained in this module is defined in the TCP/IP Stack Programmer's Reference Guide. These functions are used throughout the stack. The source code is provided so the systems programmer can adapt the memory system to fit a particular need.

The allocation functions for the small memory blocks (mmAlloc() and mmFree()) should not be altered. These functions are used by the TCP/IP stack to allocate and free scratchpad type memory. They can be called at interrupt time and are not allowed to block. The memory is currently allocated out of a static array. The size and placement of this array can be altered by changing the declarations at the top of the source file. The page size (RAW_PAGE_SIZE) is depended upon by various stack entities and should not be altered. The number of pages used (RAW_PAGE_COUNT) can be adjusted up or down to increase or decrease the scratchpad memory size.

The memory manipulation functions mmZeroInit() and mmCopy() are both coded in C. A system programmer may wish to recode these functions in assembly, or to use an EDMA channel to move memory.

The allocation functions for the large memory blocks (mmBulkAlloc() and mmBulkFree()) are currently defined to use MEM_alloc() and MEM_free() on heap ID zero. These functions can be altered to use any memory allocation system of choice. They are not called at interrupt time and are allowed to block.
mmBulkAllocSeg — Set the DSP/BIOS Heap Segment for Bulk Allocation Functions

The heap ID can be altered using a custom function. This function is not documented in the TCP/IP Stack Programmer’s Reference Guide as it is dependent on this particular implementation of mmBulkAlloc()/mmBulkFree().

7.4.1 mmBulkAllocSeg – Set the DSP/BIOS Heap Segment for Bulk Allocation Functions

Syntax

void _mmBulkAllocSeg( uint segId )

Parameters

segId DSP/BIOS heap segment

Description

Sets the DSP/BIOS segment to use in mmBulkAlloc() and mmBulkFree() function calls. This function can only be called prior to any use of the bulk allocation functions. The default segment value is zero.

Returns

None

7.5 Embedded File System - EFS.C

The EFS file system is used to provide RAM based file support for the HTTP server and any CGI functions provided by the applications programmer. This API is defined in the TCP/IP Stack Programmer’s Reference Guide. The source code is provided for adapting the functions to support a physical storage media. This will allow the HTTP server to work on the physical device without porting the server.

7.6 General OS Support - OSSYS.C

The OSSYS file is a generic catch-all for functions that do not have a home elsewhere. Currently, this module contains DbgPrintf() - a debug logging function and stricmp(), which is not contained in the RTS.

7.7 Print Functions - MINIPRINTF.C

The MINIPRINTF.C module under \\SRC\MiniPrintf directory contains an implementation of printf(), sprintf(), vprintf(), and vsprintf(). These are basic implementations that do not support floating point. The function at the top of the module, printstr() can be altered to redirect standard output from the debugger to a buffer or some external device.
IMPORTANT NOTICE

Texas Instruments Incorporated and its subsidiaries (TI) reserve the right to make corrections, modifications, enhancements, improvements, and other changes to its products and services at any time and to discontinue any product or service without notice. Customers should obtain the latest relevant information before placing orders and should verify that such information is current and complete. All products are sold subject to TI’s terms and conditions of sale supplied at the time of order acknowledgment.

TI warrants performance of its hardware products to the specifications applicable at the time of sale in accordance with TI's standard warranty. Testing and other quality control techniques are used to the extent TI deems necessary to support this warranty. Except where mandated by government requirements, testing of all parameters of each product is not necessarily performed.

TI assumes no liability for applications assistance or customer product design. Customers are responsible for their products and applications using TI components. To minimize the risks associated with customer products and applications, customers should provide adequate design and operating safeguards.

TI does not warrant or represent that any license, either express or implied, is granted under any TI patent right, copyright, mask work right, or other TI intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information published by TI regarding third-party products or services does not constitute a license from TI to use such products or services or a warranty or endorsement thereof. Use of such information may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

Reproduction of information in TI data books or data sheets is permissible only if reproduction is without alteration and is accompanied by all associated warranties, conditions, limitations, and notices. Reproduction of this information with alteration is an unfair and deceptive business practice. TI is not responsible or liable for such altered documentation.

Resale of TI products or services with statements different from or beyond the parameters stated by TI for that product or service voids all express and any implied warranties for the associated TI product or service and is an unfair and deceptive business practice. TI is not responsible or liable for any such statements.

Following are URLs where you can obtain information on other Texas Instruments products and application solutions:

<table>
<thead>
<tr>
<th>Products</th>
<th>Applications</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amplifiers</td>
<td>Audio</td>
</tr>
<tr>
<td>Data Converters</td>
<td>Automotive</td>
</tr>
<tr>
<td>DSP</td>
<td>Broadband</td>
</tr>
<tr>
<td>Interface</td>
<td>Digital Control</td>
</tr>
<tr>
<td>Logic</td>
<td>Military</td>
</tr>
<tr>
<td>Power Mgmt</td>
<td>Optical Networking</td>
</tr>
<tr>
<td>Microcontrollers</td>
<td>Security</td>
</tr>
<tr>
<td></td>
<td>Telephony</td>
</tr>
<tr>
<td></td>
<td>Wireless</td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/audio">www.ti.com/audio</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/automotive">www.ti.com/automotive</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/broadband">www.ti.com/broadband</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/digitalcontrol">www.ti.com/digitalcontrol</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/military">www.ti.com/military</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/opticalnetwork">www.ti.com/opticalnetwork</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/security">www.ti.com/security</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/telephony">www.ti.com/telephony</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/video">www.ti.com/video</a></td>
</tr>
<tr>
<td></td>
<td><a href="http://www.ti.com/wireless">www.ti.com/wireless</a></td>
</tr>
</tbody>
</table>

Mailing Address: Texas Instruments
                Post Office Box 655303 Dallas, Texas 75265

Copyright © 2006, Texas Instruments Incorporated