NetBurner Network Development Kit

Programmers Guide

Document Number: 350080-003

Revision History

Revision	Date	Description
3.1	10/27/2007	Added HTML Variable tag section
3.2	2/8/2008	Updated SRAM section 18, with description of 2.1 RC6 implemenation of SRAM usage
3.3	4/18/2008	Added OSFlags and updated examples in RTOS.
3.31	3/8/2010	Fixed example code in section 12 so that read() function using RX_BUFSIZE would read a maximum of RX_BUFSIZE-1 so that a NULL can be appended without overflowing the buffer.
3.32	10/29/2010	Added text in intro paragraph explaining this document applies only to the NetBurner Standard TCP/IP Stack. Added Dynamic Memory section and mallinfo example.
3.33	2/9/2011	Corrected Error in File Descriptor section, 5-37 changed to 5-36: • 5 – 36 for TCP (32 in total) • 37 – 63 for expansion (additional UARTs, TCP sockets, or custom)
3.34	10/21/2014	Added Multiple Interface section

1 Introduction

1.1 How to Use This Guide

The NetBurner Programming Guide is intended to provide an overview of the features and capabilities of the NetBurner Network Development Kits that use the **NetBurner Standard TCP/IP Stack**. If you have a development kit that uses the SBL2e hardware platform and NetBurner Single Chip TCP/IP Stack, please refer to the SBL2e Programmers Guide. The primary goal of this guide is to provide a brief explanation of common network applications and illustrate how you can implement these applications using NetBurner hardware, software and development tools.

The approach of this guide is to learn by example. The first program example, called Template, can be used as a starting point for most applications, and each application in this guide uses it as a base.

This guide should be useful to those new to embedded networking, and to experienced network professionals who are unfamiliar with the NetBurner tools.

1.2 Source Code for Example Programs

Over 100 example programs, including the examples in this manual, are located in the \nburn\examles\standardstack directory. The latest revision of this manual can be downloaded from http://www.netburner.com.

1.3 How to Use the NetBurner Reference Documents

All documentation is located in the "doc" directory of your tools installation. The default location is c:\nburn\docs. These documents include:

NBEclipse Getting Started Guide	Installation instructions and users guide for NBEclipse. This is required reading before using NBEclipse.
Network Programmers Guide	Programming tutorial for network platforms
NetBurner Runtime Libraries	Library reference guide for network and non-network platforms.
uC/OS Reference Manual	Library reference for uC/OS Real-time operating system.
Mod5213 Programmers Guide	Programming tutorial for Mod5213 devices
EFFS Programmers Guide	Embedded Flash File System programming tutorial
Freescale Processor Manual	Freescale detailed manuals for ColdFire microprocessors.
GNU Manuals	Manuals for GNU C/C++ libraries, compiler an linker. This includes the C/C++ language API functions.
NetBurner PC Tools	Reference manual for NetBurner tools that run on the PC, such as IPSetup, Autoupdate and MTTTY.
Platform Manuals for NetBurner Hardware	These are the NetBurner hardware manuals that include schematic information, memory maps and design guides.

There are a number of very useful resources available:

- Your support account at http://support.netburner.com
- The NetBurner open customer forum can be located by going to www.netburner.com and looking under the Support section.

1.4 NetBurner Network Development Kit Contents

You development kit contains everything you need to immediately begin writing network applications:

- NetBurner Hardware Platform
- uC/OS Real-Time Operating System
- NetBurner TCP/IP Stack
- NetBurner Web Server
- NBEclipse Integrated Development Environment (IDE)
- Command Line Tools for those who prefer not to use Eclipse
- GNU C/C++ Compiler and Linker
- NetBurner Configuration Utilities including IPSetup and AutoUpdate
- Power Supply
- Serial Cable, Standard Network Cable (blue) and Cross-wired Network Cable (red)

1.5 Getting Started

This guide will provide an overview of how to install and configure your NetBurner tools and devices, but please refer to the Quick Start Guide and User Manual (from Windows: Start→ Programs → Netburner NNDK → NNDK Users Manual) that came with your development kit for additional details.

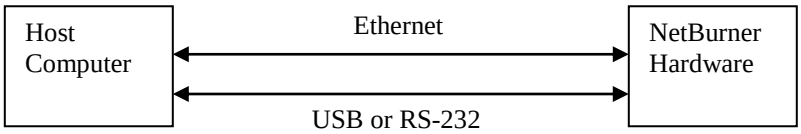
1.5.1 Software Installation

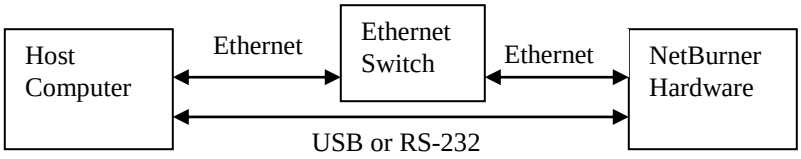
NetBurner software and tools run on Windows XP through Windows 8. You can download the software by registering the keycode on the large red card that came with your kit at support.netburner.com.

In addition, you will need the 32-bit version of the Java JRE, revision 1.5 or later.

1.5.2 Hardware Installation

Your Network Development Kit includes one of several possible NetBurner hardware platforms. Each platform will require a power connection (USB by default), an Ethernet network connection, and an optional serial connection through USB or the DB9 serial port available on most development boards. Please refer to the Quick Start Guide that came with your hardware platform for detailed instructions. Once the hardware installation is complete, you should have the equivalent of one of the two block diagrams below:



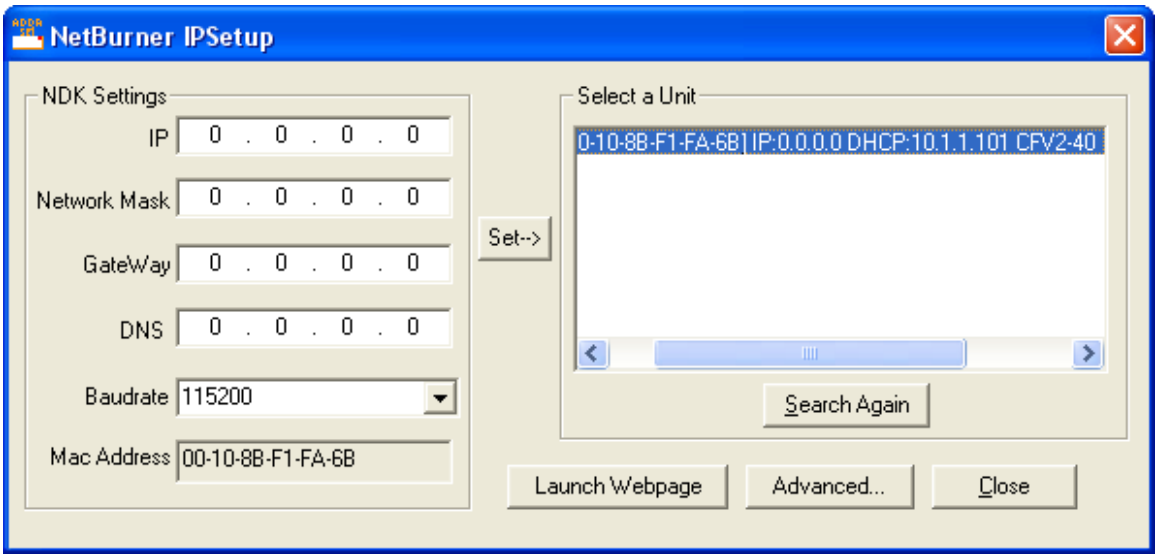


1.5.3 Network Configuration

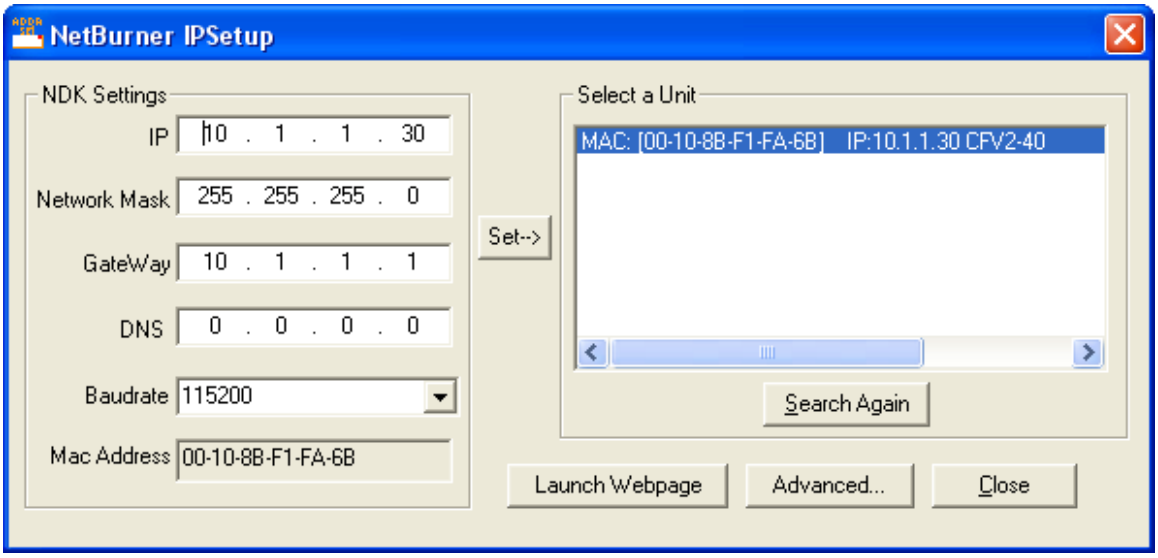
Once the hardware and software installations are complete, you will need to either verify automatic network settings, or assign static network settings of the NetBurner device.

1. Verify the hardware is connected correctly. A link light is located near the RJ-45 Ethernet connector on your NetBurner board. The link light will be lit if the network cable is connected correctly to both the host computer and the NetBurner device. **Note:** If the link light is not lit, network communication will not be possible.

2. Run the NetBurner IPSetup program. (From Windows: Start → Programs → Netburner NNDK →IP Setup tool.) IPSetup will allow you to view your NetBurner device's current settings, or modify the settings. If you are using DHCP, then the values in the left pane in the IP Setup window will be zero, and the IP address assigned by the DHCP server will appear in the "Select a Unit" pane as shown below. The name following the IP address indicates the NetBurner platform name. If your running application supports a web interface, the "Launch Webpage" button will open up (when clicked) your default web browser to display your NetBurner device's home page.



3. If you wish to assign a static IP address, enter the information into the "NDK Settings" pane (as shown below) and click on the "Set" button. There will be a short pause while parameters are updated. If you do not see your device in the "Select a Unit" window, click on the "Search Again" button. If your running application supports a web interface, the "Launch Webpage" button (when clicked) will open up your default web browser to display your NetBurner board's home page.



1.5.4 Debug Port

Throughout this guide, we will refer to the "debug port". The debug port is one of the RS-232 ports that can be used to interact with your NetBurner device in the example programs. By default stdout, stdin and stderr are mapped to the debug port, so when you use functions like printf( ), scanf( ), gets( ), etc. they read and write to the debug port. All of this is configurable. You can also disable the debug port and use the port as a general purpose UART, or you can reassign the stdio file descriptors to use other serial or network interfaces.

2 Networking Device Configuration

2.1 Obtaining an IP Address

To get your NetBurner Network Development Kit (NNDK) up and running as quickly as possible, you must have an IP Address for both your host computer and your NetBurner device. The NetBurner factory application supports both static and DHCP assigned IP addresses.

2.2 Static IP Address

If you are part of an existing network and want to use a static IP address, you must get the address from your network administrator. If you connecting your NetBurner hardware to a single computer, or are on an isolated network, you should select one of the reserved addresses described in the "How do I select an IP address" section of this guide.

2.3 Auto IP Address

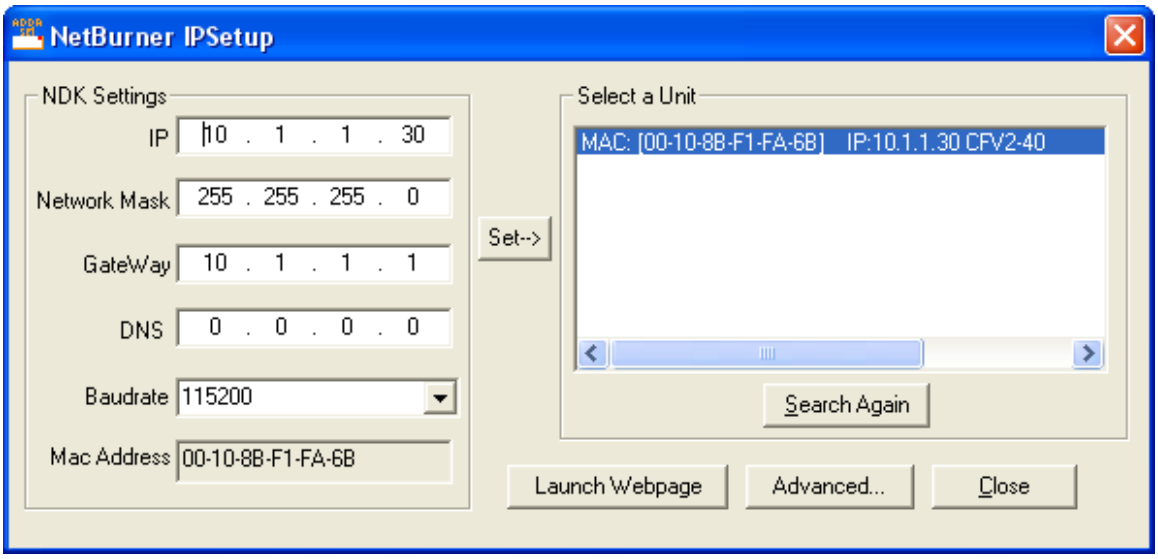
If you have the static IP address set to 0.0.0.0 our example applications will attempt to use DHCP to get an address. If a DHCP server is not available on your network a class B Auto IP address will be assigned to your device.

2.4 Dynamic IP Address (DHCP)

When the factory program boots, it will first attempt to obtain an address from a DHCP server. If you are connected to a network with a DHCP server, everything should get configured automatically.

2.5 Network Configuration Step by Step Instructions

1. Get a static IP address, or use DHCP.
2. Install the NetBurner Development Tools on your host computer.
3. Connect the Ethernet cable from the RJ-45 connector to your switch, network or computer.
4. Connect the USB cable or an optional external 9V to 12V power supply to the board. Please refer to your Quick Start Guide for additional information.
5. Execute the IPSetup.exe program. From the Windows: Start→ NetBurner→ IPSetup tool. (If you used the default installation settings, the program is located in the C:\nburn\pctools\ipsetup directory).
6. The IPSetup program will automatically locate all NetBurner devices on your network. If more than one device appears, select your device by matching the MAC address displayed in IPSetup with the MAC address sticker on your NetBurner board.
7. Click on the "Launch Webpage" page button in the IPSetup window to view the factory application.



For additional information, please refer to your User Manual. From the Windows Start Menu: Start→ Programs→ NetBurner NNDK→ NNDK Users Manual.

3 How Do I Select an IP Address?

If you are part of an existing network and are not using DHCP, stop reading now and go get an IP address and network mask from your network administrator. If you follow the advice in this paragraph on an existing network without an assigned IP address, the Administrator will hunt you down.....

IP addresses are used to route packets from place-to-place on Intranets/Internet. If you are not part of an established network, and your Ethernet segment is isolated, you can choose just about any IP address you desire.

The powers that be have actually set aside some addresses for isolated networks. They are documented in RFC1918. The reserved ranges are:

10.0.0.0 to 10.255.255.255 Class A
172.16.0.0 to 172.31.255.255 Class B
192.168.0.0 to 192.168.255.255 Class C

If you are doing development on an isolated network, you can use the following addresses (they will be used for all of the examples in the documentation):

- Set Your PC's Network Adapter Card IP Address to 10.1.1.10 (only change the Network Adapter Card, **do not** change your Dial-Up Adapter settings)
- Set the IP address of the NetBurner board to 10.1.1.11
- Set the network mask for both the PC network adapter and the NetBurner board to 255.255.255.0

4 Web Browsers and Proxy Servers

If you are working on a corporate LAN that uses a proxy server for Internet web browsing, you will need to **exclude** the IP address of your NetBurner hardware in your web browser's proxy server settings/preferences. Otherwise, an attempt to connect to a web page on the LAN will fail because the proxy server will attempt to route the request outside the LAN. For most web browsers, this can be accomplished in the advanced settings for the proxy server configuration. If you are using IE 6.x, click on Tools → Internet Options → Connections → LAN Settings → Proxy server. For additional information, please contact your IT Department.

5 Using the NBEclipse IDE to Create the Template Program

The NBEclipse Application Wizard can quickly create a project and generate source code for a fully functional network-enabled application. This application can be used as a starting point and modified to suit your product requirements. The tutorial on the NBEclipse Application Wizard is covered in detail in the NBEclipse Getting Started Guide, and is not repeated in this document.

If you prefer command line tools, you can do all of your development using the make utility. Please see the section entitled "Using the NetBurner Command Line Tools to Create the Template Program" for details.

Traditionally called the "Hello World" program, the "Template Program" will specify a minimal code base from which you can write your future applications. The objective of this template program is to print the characters "Hello World" out the debug port of your NetBurner device. In addition, this template program will enable network services so that it can be downloaded over a network connection instead of through a serial port or a BDM (Background Debug Mode) port.

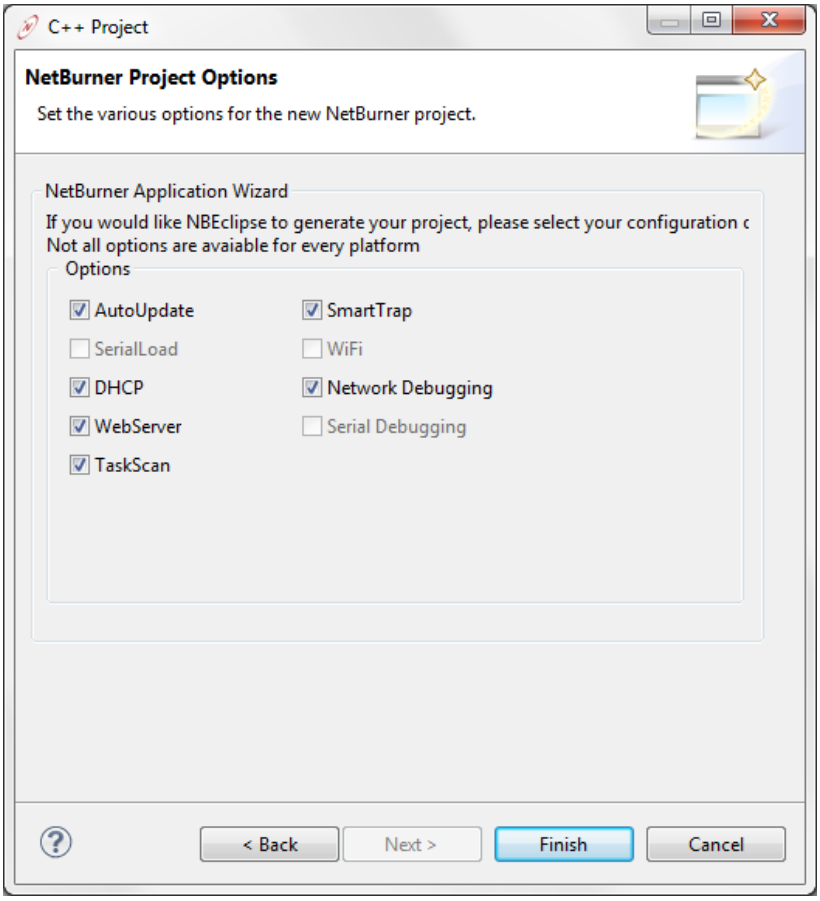
The Debug Monitor

The NetBurner device contains a flash memory boot sector loaded with a boot program called the "Debug Monitor". This program is designed to be very small and takes up less than 16 Kbytes of memory space. The Debug Monitor is not designed to provide full TCP network communications, although it does support the TFTP protocol.

The full TCP/IP Stack functionality is compiled as part of your application. If you download an application that immediately crashes when it boots, full network services will not be available. In this case, the NetBurner Debug Monitor comes to the rescue. Once in the Debug Monitor (at the NB> prompt), you can download a working application through the serial debug port. See the section on serial downloads using the Debug Monitor for more information.

5.1 Create a New Project with the AppWizard

The NetBurner Application Wizard will create a project and C/C++ source code for your application. This is a great way to start a new project since you can add functions like DHCP and HTML processing by selecting the appropriate checkbox items in the Application Wizard dialog box. The screenshot below shows the features selected in the NBEclipse Application Wizard to produce the code for the Template Program.



The features we selected are:

AutoUpdate	Enables application updates through a network connection.
DHCP	Ability to get a DHCP address from a DHCP server
Web Server	Enables HTTP web services
Task Scan	Task Scan enables you to get a snapshot of what RTOS tasks are running in the Release build of your application, including the call stack and source code line numbers for each task. To use Task Scan you run the Windows TaskScan.exe program on a PC, which communicates with your NetBurner device.
Smart Trap	Provides additional debug information if your application crashes. For example, a null pointer assignment, stack overflow, or bus error. The information is displayed through the debug serial port and can be viewed with a utility such as MTTTY. The default serial port is UART 0.
Network Debugging	Adds function call to enable network debugging if a Debug build of the code is compiled and downloaded to the target device.

5.2 Template Program Source Code

Edit the top of the file to add the Template header description as shown below. Comments have been added to the source code to explain the function of important functions.

```
/*-------------------------------------------------------------------
 Description: The Template Program
 Filename: main.cpp
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <taskmon.h>
#include <NetworkDebug.h>

extern "C" {
   void UserMain(void * pd);
}

const char * AppName="Template Program"; // Name for IPSetup

void UserMain(void * pd) {
    InitializeStack();      // Initialize TCP Stack
    if (EthernetIP == 0)    // Enable DHCP if no static IP
       GetDHCPAddress();
    OSChangePrio(MAIN_PRIO);// Set UserMain() task priority
    EnableAutoUpdate();     // Enable network updates
    StartHTTP();            // Start Web Server
    EnableSmartTraps();     // Enable Smart Traps
    EnableTaskMonitor();    // Enable Task Scan

    #ifdef _DEBUG           // Enable GDB stub if Debug build
    InitializeNetworkGDB_and_Wait();
    #endif

    iprintf("Application started\n");
    while (1) {
        OSTimeDly( 20 );
    }
}
```

The above program is a fully functional network application in just a few lines of code! The only application specific code is inside the `while(1)` loop; the remainder of the program is what we will refer to as the "Template Program". Although the purpose of our application is to print "Hello World" out the debug serial port, adding the network support will allow fast code development using NBEclipse, and also allow network configuration using the NetBurner IPSetup utility (i.e. IPSetup tool). For additional information on IPSetup and AutoUpdate please refer to your NNDK User Manual. From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.

The “extern C” declaration is used to so that UserMain( ) is compatible with both C and C++ applications; it prevents the name mangling associated with C++.

```
extern "C" {
   void UserMain(void * pd);
}
```

The line: `const char * AppName="Template Program";` will enable the IPSetup utility to display the application name. You can change “Template Program” to any string you wish. If this variable is not set, IPSetup will not display a value for this field. The maximum string length is 40 characters.

The UserMain( ) function is a thread created by the system to be your application’s main entry point for taking control over the function of the device . The parameter passed to UserMain( ) is a void pointer to some type of data. This is a feature of the uC/OS RTOS, but it is not needed for the NetBurner tool set.

The next group of function calls handle system initialization:

```
InitializeStack();      // Initialize TCP Stack
if (EthernetIP == 0)    // Enable DHCP if no static IP
   GetDHCPAddress();
OSChangePrio(MAIN_PRIO);// Set UserMain() task priority
EnableAutoUpdate();     // Enable network updates
StartHTTP();            // Start Web Server
EnableSmartTraps();     // Enable Smart Traps
EnableTaskMonitor();    // Enable Task Scan
```

`InitializeStack()` initializes the TCP/IP Stack. This is required for any network communications to take place.

The line: if (EthernetIP = = 0) GetDHCPAddress(); checks the NetBurner device’s IP address setting, and if the IP address is 0.0.0.0, the device will attempt to contact a DHCP Server and obtain a dynamic IP configuration, including the IP address, mask, gateway and DNS Server.

What if DHCP fails?

If you run IPSetup and notice that the IP address of your device is 0.0.0.1, this is an indication that a DHCP Server could not be found on your network. The DHCP Client runs as a separate task, so it will keep trying to get a DHCP address until it succeeds, or the application explicitly calls StopDHCP();.

The function call OSChangePrio(MAIN_PRIO) sets the UserMain task priority to the default, which is 50. The RTOS has a total of 63 priority levels. Level 1 is the highest, and level 63 is the lowest priority. In a preemptive RTOS, the highest priority task ready to run will execute. For example, the TCP/IP stack task is a higher priority task (lower priority number), and will interrupt UserMain() when necessary to process network data.

Some of the tasks are reserved. For example, task 63 is the system idle task, which runs when no other tasks are ready to run. The system defines the following tasks in \nburn\include\constants.h:

```
#define MAIN_PRIO (50)
#define HTTP_PRIO (45)
#define PPP_PRIO (44)
#define TCP_PRIO (40)
#define IP_PRIO (39)
#define ETHER_SEND_PRIO (38)
```

In addition, task 0 and task 63 are reserved. Tasks available for user applications are any of the remaining task priority numbers. Some network modules such as FTP and Telnet Command will also use task priorities, but the priority will be an option passed to the function that starts the task.

Always check the return value when creating a task!

If you call a function that creates a new task and you specify a task priority that is already in use, the function will return an error.

An easy way to keep track of priority levels for user applications is to use the MAIN_PRIO definition and add or subtract a number from it. For example, if you create a new task and want it to be of higher **priority** than UserMain, then use (MAIN_PRIO – 1). If you want it to be of lower **priority**, the add 1, etc.

The function `EnableAutoUpdate();` will enable the network flash memory update capability of the device. The Autoupdate utility is used both during development to quickly download code to flash memory, and also as an update mechanism once the device is deployed.

The `StartHTTP()` function starts the HTTP Web Server task. When you build your project, the web page data in your project's "html" directory will be processed and made available to be served up as a web page. For example, you will probably have a web page called "index.htm". The `StartHTTP()` function will start a task that listens on port 80 for incoming HTTP requests such as those from a web browser. An optional parameter may be passed to the function to select a different port number. For example, `StartHTTP(81);` will start the Web Server and tell it to listen on port 81. `StartHTTP( )` may only be called once.

The `EnableTaskMonitor()` enables TaskScan, a network debugging tool that is used to view tasks and their status in a running application. To use TaskScan you must add #include <taskmon.h> in your application's main.cpp file and EnableTaskMonitor(); in user main. When the TaskScan utility is run on your Windows computer you can view the running tasks, their status and current source code line number. There are no performance hits if you include TaskScan in your application; the only time it will be invoked is when you run the TaskScan utility and connect to your NetBurner device. TaskScan is covered later in this manual.

The NetBurner system will catch programming errors that cause traps, such as null pointers, and display debug information such as the program counter, address registers and data registers. The `EnableSmartTraps()` function call provides more detailed debugging information when a trap occurs, such as the RTOS task information. Note that if you are using GDB/Insight Debugging, EnableSmartTraps( ) must called before the GDB stub function.

The section of code for the debugger with check for the definition "_DEBUG". If defined, the build is a Debug build (as opposed to a Release build), and the network debug stub will be initialized. The "wait" in the function call name refers to the fact that the application will wait until the debugger connects, then continue execution.

```
#ifdef _DEBUG             // Enable GDB stub if Debug build
InitializeNetworkGDB_and_Wait();
#endif
```

The inside of the while loop is where you would place your application code. Modify main.cpp to add the iprintf( ) shown below:

```
while (1)
{
    iprintf("Hello World\r\n");  // integer version of printf()
    OSTimeDly( TICKS_PER_SECOND );  // Default is 20 ticks per second
}
```

The `iprintf()` and `OSTimeDly(20)` are just there for the example; you would replace those lines with whatever you want your application to do. Note that you should never return from this while loop; if you did, then your application would lose control of the hardware.

There are 20 ticks per second by default. The definition "TICKS_PER_SECOND" is defined as "20", and can be used in place of a numeric value. For example, a delay of 2 seconds is: "OSTimeDly(TICKS_PER_SECOND * 2)" .

5.3 Template Program Setup

Before running our program please verify that your hardware is set up as described in the previous Hardware Installation section.

5.3.1 Testing the RS-232 Debug Connection

You can determine if you are properly connected to the debug port with the following test:

1. Start the dumb terminal program MTTTY, which is included in your NetBurner tools. You can start it from within NBEclipse from the NBEclipe menu item or MTTTY icon, or from the Windows start menu: Start → Programs → Netburner NNDK → Mttty Serial Terminal. Set the com port to whichever port you are using on your computer (usually com1), and set the baud rate to 115,200. Make sure to click on the "Connect" button in the MTTTY window to establish the connection.
2. Power on or reset your NetBurner device. The MTTTY screen should display a sign-on message similar to: "Waiting to boot……". If you see this message, then you are connected correctly.

For additional information, please refer to your User Manual. From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.

5.4 Compiling and Running the Application

Now that we have the application source code file, we need to compile it into a code image and download it to your NetBurner device. There are four methods to download your applications:

1. Through the serial port
2. Through a network connection using AutoUpdate. This is the preferred method, and can be run from within the IDE, or as a stand-alone application.
3. Through a network connection using TFTP
4. Through a network connection using FTP

In order to execute your application on your NetBurner device you will need to do the following:

1. Compile the source code into an application image
2. Download the application image to the flash memory of the NetBurner device
3. Reboot the NetBurner device so the application can begin execution

The NBEclipse IDE uses the AutoUpdate functionality to download code to your NetBurner device by using a "Run Configuration". The device will then reboot and begin execution of the new application. If you are unfamiliar with this process, please reference the NBEclipse Getting Started Guide. Once the download is complete, you will see the debug messages appear in MTTTY, along with "Hello World".

Flash and RAM application files

When your code compiles correctly, two files are created: template.s19 and template_APP.s19. The template.s19 file is memory mapped to run from RAM, while template_APP.s19 is memory mapped to Flash memory. **Note:** All compiled images will be located in your projects Release directory (\nburn\pcbin for command line builds). This guide will focus on Flash downloads. Please refer to the section on Downloading to RAM in your User Manual for more information on downloading applications to RAM.

6 DHCP - Dynamic Host Configuration Protocol

DHCP is used to provide host configuration parameters on a TCP/IP network. DHCP is built on a client-server architecture in which one or more designated DHCP Servers allocate network addresses and other configuration information to hosts (DHCP Clients). **Note:** All NetBurner devices can function as a DCHP Client. For additional information about DHCP, please refer to your NNDK User Manual. From Windows: Start → Programs → NetBurner NNDK → NNDK Users Manual.

To enable DHPC Client services, your application code must contain DHCP Client API function calls to enable the service. DHCP can dynamically configure many parameters, including:

1. IP Address
2. Subnet Mask
3. Gateway Address
4. DNS Server Address

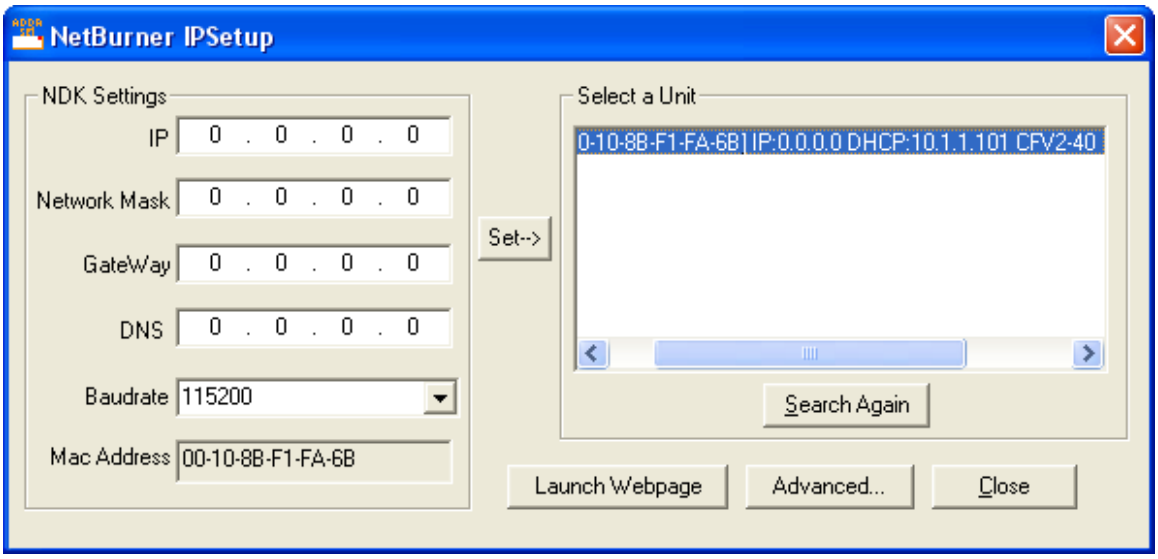
The example code below checks the first interface, but you can modify it to check for the second or third interfaces. Typically the first interface will be your primary Ethernet interface. To use the network interface functions, you must include netinterface.h.

```
// Get first interface identifier. Use GetNextInterface(<last interface>) to
// obtain additional interface numbers if necessary.
int FirstInterface = GetFirstInterface();
InterfaceBlock *ib = GetInterFaceBlock(FirstInterface);  // Get interface data

if (ib->netIP == 0)  // Check IP address for 0.0.0.0, and use DHCP if necessary
{
   iprintf("\r\nNo static IP address set, attempting DHCP\r\n");
   if ( GetDHCPAddress( FirstInterface ) == DHCP_OK )
   {
      iprintf("DHCP address: "); ShowIP(ib->netIP); iprintf("\r\n");
   else
   {
      iprintf("Error: could not obtain a DHCP address\r\n");
   }
}
```

The code checks the interface block variable ib->netIP to determine if the host IP address is 0. If the IP address is 0, then the DHCP Client should be invoked to obtain a dynamic IP parameters. If EthernetIP is not 0, then the system assumes a static IP address has been assigned, and DHCP is not used.

In previous tools releases that did not include multiple network interface support, a global variable named EthernetIP was used to access the Ethernet IP address of the device. While this variable will still work for backward compatibility, it is recommended that the interface functions be used for all new applications.

An easy way to check your NetBurner board to determine if it has a DHCP assigned IP address is to use the IPSetup program (from Windows: Start→ Programs → Netburner NNDK → IP Setup tool). In the screenshot below, the IP address field representing the static IP address is shown as 0.0.0.0, followed by the DHCP-assigned IP address (i.e. 10.1.1.101). Other parameters such as the network mask, gateway and DNS are also assigned and can be accessed as parameters in your application. The NDK Settings section (left pane of the IPSetup window), represents the static settings.



For additional information about IP Setup, please refer to your NNDK User Manual. From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.

7 Changing IP Addresses at Run-Time

System configuration parameters such as IP address, mask, gateway and DNS are stored in two places:

1. Configuration Records used by the system to store configuration parameters in flash memory.
2. Interface Blocks used by the system at run time.

There are 3 Configuration Records, numbered 0, 1 and 2, as defined in \nburn\include\netinterface.h:

```
#define CONFIG_IF_ID_ETHERNET  (0)
#define CONFIG_IF_ID_WIFI      (1)
#define CONFIG_IF_ID_ETHERNET2 (2)
```

The Configuration Record is a structure that contains all the system configuration parameters. When your NetBurner device boots, it copies these parameters to run-time variables that are used during normal system operation, called Interface Blocks. This application illustrated how to read and modify both runtime and stored flash configuration parameters.

NOTE: This application note was written for tools release 1.95. If you are using a later revision, please check the system file references to verify specific information on function calls and structures. This application note does not apply to prior releases.

7.1 THE CONFIGURATION RECORD

Configuration Records are stored in an 8k bytes sector of flash memory. There is one ConfigRecord structure for each network interface in your NetBurner device. In the 1.95 release, the each ConfigRecord occupies 256 bytes. To enable an application to modify and save the network settings you must retrieve the ConfigRecord for a specific interface, modify it, and save it to flash. The ConfigRecord structure is defined in \nburn\include\system.h:

```
typedef struct
{
   unsigned long recordsize;       /* The stored size of the struct */
   unsigned long ip_Addr;          /* The device IP Address */
   unsigned long ip_Mask;          /* The IP Address Mask */
   unsigned long ip_GateWay;       /* The address of the P gateway */
   unsigned long ip_TftpServer;    /* The address of the TFTP server to load
                                      data from */
   unsigned long baud_rate;        /* The initial system Baud rate */
   unsigned char wait_seconds;     /* The number of seconds to wait before
                                      booting */
   unsigned char bBoot_To_Application;  /* True = app., False = monitor */
   unsigned char bException_Action;     /* What should we do when we  have an
                                           exception? */
   unsigned char m_FileName[80];    /* The file name of the TFTP file to load */
   unsigned char mac_address[6];    /* The Ethernet MAC address */
   unsigned char ser_boot;
   unsigned long ip_DNS_server;
   unsigned char core_mac_address[6];  /* The Base unit MAC address */
   unsigned char typeof_if;
   unsigned char direct_Tx;
   unsigned long m_ExtraData[4];
   unsigned char m_bUnused[3];
   unsigned char m_q_boot;            /* True to boot without messages */
   unsigned short checksum;           /* A Checksum for this structure */
   }__attribute__(( packed) ) ConfigRecord;

   /* The read-only system config record  */
```

```
extern const ConfigRecord gConfigRec;
```

7.2 Reading the Configuration Record

The functions to retrieve and save a ConfigRecord are:

```
ConfigRecord *GetIfConfig( int num );
void SaveIfConfig( ConfigRecord *cr, int num );
```

where num represents the interface number: 0, 1 or 2. The *cr pointer would point to the new ConfigRecord you wish to save in flash memory.

Although the gConfigRec is available as the run-time copy for the first interface, it is recommended you use the GetIfconfig( ) function to get a fresh copy for the specific interface you want to modify.

THE NETWORK INTRFACE BLOCK

While the ConfigRecord described in the previous section is used to store parameters in flash, the InterfaceBlock structure is used during system run-time to store the network configuration settings for each network interface. At boot time, the network settings from the ConfigRecord are read and copied to the InterfaceBlock.

As of tools revision 1.95, the values of the device IP address, mask, gateway, DNS, etc. are controlled by a structure located in \nburn\include\netinterface.h. The netinterface method was created to enable devices to have multiple network interfaces, such as multiple Ethernet ports and WiFi. The configuration information is kept in a linked list of structures with the following format:

```
struct InterfaceBlock
{
      MACADR theMac;
      IPADDR netIP;
      IPADDR netIpMask;
      IPADDR netIpGate;
      IPADDR netDNS;
      SendNetBuffer *send_func;
      KillInterface *kill_if;
      fEnableMulticast *enab_multicast;
      const char *InterfaceName;
      int config_num;
};
```

The functions to get an existing structure or save a modified structure are:

```
int GetFirstInterface();
int GetnextInterface( int last );
```

Functions to read InterfaceBlock parameters are:

```
IPADDR InterfaceIP( int InterfaceNumber );
IPADDR InterfaceDNS( int InterfaceNumber );
IPADDR InterfaceMASK( int InterfaceNumber );
IPADDR InterfaceGate( int InterfaceNumber );
MACADR InterfaceMAC( int InterfaceNumber );
```

Where InterfaceNumber is the interface number: 1, 2, 3, etc.

If all you wish to do is to change a network parameter at run-time, then you only need to change the InterfaceBlock value.

EXAMPLE: MODIFY AND CHANGE NETWORK SETTINGS

The following is an example program illustrating how modify and save a network setting:

1. Read ConfigRecord
2. Read InterfaceBlock
3. Modify InterfaceBlock to affect run-time values
4. Save ConfigRecord to store new parameters

7.3 Static and DHCP IP Address Modification Example

```
/*******************************************************************
 Example program to illustrate how an application can change the
 run-time and stored Flash values for Ethernet IP, mask, Gateway
 and DNS.

 THIS EXAMPLE APPLIES TO TOOLS RELEASE 1.95 OR LATER

 INTRODUCTION FOR CONFIGURATION RECORDS AND INTERFACE BLOCKS
 - The NetBurner device contains runtime and stored (Flash) system
   configuration parameters. Interface Blocks are used at runtime,
   and Configuration Records are stored in Flash. At boot time,
   data from the Configuration Records are copied to Interface
   Blocks.

 - There are 3 Configuration Records as defined in netinterface.h:
     #define CONFIG_IF_ID_ETHERNET  (0)
     #define CONFIG_IF_ID_WIFI      (1)
     #define CONFIG_IF_ID_ETHERNET2 (2)

   The order of these configuration records is fixed, regardless of
   the Interface Block number. For example, if you have only a WiFi
   interface, you still use Configuration Record 1.

 - Configuration Records are numbered 0, 1 and 2. Interface Blocks
   are numbered 1, 2, 3. There is no correlation between this
   numbering. Interface Block 3 could reference Configuration Record
   1, depending on the order of interface registration calls by your
   application.

 - This example will provide a menu through the serial debug port
   that enables you to set/clear static IP settings, and start/stop
   the DHCP Client service.

*******************************************************************/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
```

```
#include <dhcpclient.h>

// Make sure to include these header files
#include <bsp.h>
#include <string.h>
#include <utils.h>
#include <system.h>
#include <netinterface.h>

extern "C" {
   void UserMain(void * pd);
}

// Application name
const char * AppName = "Change IP Example";


// Variable to indicate if appliction tried to obtain runtime IP settings
// from a DHCP server.
bool AssignedDHCP = FALSE;


// Since we want to be able to start and stop DHCP, we need to create out
// own DHCP Object instead of using the GetDHCPAdderss() function which
// handles this automatically.
DhcpObject *pDhcpObj;

// Add a device name for DNS
const char *DeviceName = "NetBurner";
extern const char *pDHCPOfferName;    // point this at above name in UserMain



/*-------------------------------------------------------------------
 Display runtime IP values
 This function demonstrates two methods to read the runtime IP
 values:
 1. Using the read-only Interface function calls
 2. Using the GetInterfaceBlock() function to get a pointer to an
    Interface Record.
 -------------------------------------------------------------------*/
void DisplayRuntimeIPSettings(int InterfaceNumber)
{
   iprintf("\r\n\r\n--- RUNTIME IP SETTINGS ---\r\n");
   if ( AssignedDHCP )
      iprintf("Values assigned by DHCP Server\r\n");

   // Display current runtime values using Interface read-only functions
   iprintf("IP runtime settings using Interface Functions for interface %d:\r\n",
            InterfaceNumber);
   iprintf("IP:   "); ShowIP(InterfaceIP(InterfaceNumber)); iprintf("\r\n");
   iprintf("Mask: "); ShowIP(InterfaceMASK(InterfaceNumber)); iprintf("\r\n");
   iprintf("Gate: "); ShowIP(InterfaceGate(InterfaceNumber)); iprintf("\r\n");
   iprintf("DNS:  "); ShowIP(InterfaceDNS(InterfaceNumber)); iprintf("\r\n");

   // Display current runtime values by getting a pointer to the Interface Block
   // and accessing it's variables.
   iprintf("\r\nIP runtime settings using GetInterfaceBlock() for interface %d:\r\n",
           InterfaceNumber);
   InterfaceBlock *ib = GetInterFaceBlock(InterfaceNumber);
   iprintf("IP:   "); ShowIP(ib->netIP); iprintf("\r\n");
   iprintf("Mask: "); ShowIP(ib->netIpMask); iprintf("\r\n");
   iprintf("Gate: "); ShowIP(ib->netIpGate); iprintf("\r\n");
```

```
   iprintf("DNS:  "); ShowIP(ib->netDNS); iprintf("\r\n");
   iprintf("Interface Name: %s\r\n", ib->InterfaceName);
   iprintf("\r\n");
}


/*-------------------------------------------------------------------
 Display Flash IP values
 This function demonstrates two methods to read the IP settings
 stored in the Flash Configuration Record:
 1. Using the gConfigRec global read-only structure
 2. Using the RawGetConfig() functoin to obtain a pointer to a
    specific Configuration Record.
 -------------------------------------------------------------------*/
void DisplayFlashIPSettings(int RecordNumber)
{
   // Note that gConfigRec only applies to ConfigRecord 0
   iprintf("\r\n--- FLASH IP SETTINGS ---\r\n");
   iprintf("These values will be 0 if you are using DHCP\r\n");
   iprintf("IP Flash settings using gConfigRec for Record 0:\r\n");
   iprintf("IP:   "); ShowIP(gConfigRec.ip_Addr); iprintf("\r\n");
   iprintf("Mask: "); ShowIP(gConfigRec.ip_Mask); iprintf("\r\n");
   iprintf("Gate: "); ShowIP(gConfigRec.ip_GateWay); iprintf("\r\n");
   iprintf("DNS:  "); ShowIP(gConfigRec.ip_DNS_server); iprintf("\r\n");

   // Note that ConfigRecord structures start at 0 for the first interface
   ConfigRecord *cr = RawGetConfig(RecordNumber);
   iprintf("\r\nIP Flash settings using RawGetConfig() for Record %d:\r\n",
           RecordNumber);
   iprintf("IP:   "); ShowIP(cr->ip_Addr); iprintf("\r\n");
   iprintf("Mask: "); ShowIP(cr->ip_Mask); iprintf("\r\n");
   iprintf("Gate: "); ShowIP(cr->ip_GateWay); iprintf("\r\n");
   iprintf("DNS:  "); ShowIP(cr->ip_DNS_server); iprintf("\r\n");
}



/*-------------------------------------------------------------------
 Change the runtime IP settings.
 This function will display the current IP address and mask,
 change the runtime variables to new values, then display the
 new values.
 -------------------------------------------------------------------*/
void ChangeRuntimeIPSettings(int InterfaceNumber, IPADDR IpAddr, IPADDR IpMask, IPADDR
IpGate, IPADDR IpDNS)
{
   iprintf("\r\nChanging IP runtime settings for interface %d:\r\n",
           InterfaceNumber);

   // Display current values
   InterfaceBlock *ib = GetInterFaceBlock(InterfaceNumber);
   iprintf("Old Settings:\r\n");
   iprintf("   IP:   "); ShowIP(ib->netIP); iprintf("\r\n");
   iprintf("   Mask: "); ShowIP(ib->netIpMask); iprintf("\r\n");
   iprintf("   Gway: "); ShowIP(ib->netIpGate); iprintf("\r\n");
   iprintf("   DNS:  "); ShowIP(ib->netDNS); iprintf("\r\n");
   iprintf("   Interface Name: %s\r\n", ib->InterfaceName);

   // Change to new values
   ib->netIP = IpAddr;
   ib->netIpMask = IpMask;
   ib->netIpGate = IpGate;
   ib->netDNS = IpDNS;
```

```
   // Display new values. At this point, you can communicate with the
   // device using the new ip address and mask.
   iprintf("New Settings:\r\n");
   iprintf("   IP:   "); ShowIP(ib->netIP); iprintf("\r\n");
   iprintf("   Mask: "); ShowIP(ib->netIpMask); iprintf("\r\n");
   iprintf("   Gway: "); ShowIP(ib->netIpGate); iprintf("\r\n");
   iprintf("   DNS:  "); ShowIP(ib->netDNS); iprintf("\r\n");
   iprintf("   Interface Name: %s\r\n", ib->InterfaceName);
}


/*-------------------------------------------------------------------
 Store new IP address and mask settings to the configuration record
 in Flash. In most cases you would have already changed the runtime
 values to the same settings.
 -------------------------------------------------------------------*/
void ChangeFlashIPSettings(int RecordNumber, IPADDR IpAddr, IPADDR IpMask, IPADDR
IpGate, IPADDR IpDNS)
{
   iprintf("\r\nChanging Flash settings using RawGetConfig() for Record %d:\r\n",
           RecordNumber);

   // Get pointer to Configuration Record
   ConfigRecord *cr = RawGetConfig(RecordNumber);

   // Display current Flash values
   iprintf("Old Settings:\r\n");
   iprintf("   IP:   "); ShowIP(cr->ip_Addr); iprintf("\r\n");
   iprintf("   Mask: "); ShowIP(cr->ip_Mask); iprintf("\r\n");
   iprintf("   Gate: "); ShowIP(cr->ip_GateWay); iprintf("\r\n");
   iprintf("   DNS : "); ShowIP(cr->ip_DNS_server); iprintf("\r\n");

  // create new config record and copy data
   ConfigRecord NewRec;
   memcpy( &NewRec, cr, sizeof( NewRec ) );

   // Change parameters
   NewRec.ip_Addr = IpAddr;
   NewRec.ip_Mask = IpMask;
   NewRec.ip_GateWay = IpGate;
   NewRec.ip_DNS_server = IpDNS;

   // Write new values to Flash system configuration sector
   UpdateConfigRecord_Num(&NewRec, RecordNumber);

   // Display current Flash values
   iprintf("New Settings:\r\n");
   iprintf("   IP:   "); ShowIP(cr->ip_Addr); iprintf("\r\n");
   iprintf("   Mask: "); ShowIP(cr->ip_Mask); iprintf("\r\n");
   iprintf("   Gate: "); ShowIP(cr->ip_GateWay); iprintf("\r\n");
   iprintf("   DNS : "); ShowIP(cr->ip_DNS_server); iprintf("\r\n");

   // You do not need to reboot if you change both the runtime
   // and flash values. This function call requires #include <bsp.h>
   //   ForceReboot();
}


/*-------------------------------------------------------------------
 DisplayUserMenu
 Displays menu through the serial port to interact with program.
 -------------------------------------------------------------------*/
```

```
void DisplayUserMenu()
{
   iprintf("\r\n--- Main Menu ---\r\n");
   iprintf("1. Display Runtime Settings\r\n");
   iprintf("2. Display Flash Settings\r\n");
   iprintf("3. Change Runtime Static Settings\r\n");
   iprintf("4. Change Flash Static Settings\r\n");
   iprintf("5. Change Flash & Runtime Static Settings to 0.0.0.0\r\n");
   iprintf("6. Start DHCP Client Servce\r\n");
   iprintf("   and attempt to get a DHCP address\r\n");
   iprintf("7. Stop DHCP Client Service\r\n");
}


/*-------------------------------------------------------------------
 MyStartDHCP
 Most applications can just use the GetDHCPAddress() function at boot
 time to enable the DHCP Client. If you require more control over the
 DHCP service, such as starting and stopping, then a few lines of
 code are required to create a DhcpObject and check a semaphore to
 determine if the DHCP request was successful. This is essentially
 what the GetDHCPAddress() function does.
 -------------------------------------------------------------------*/
void MyStartDHCP( InterfaceBlock *ib )
{
   int FirstInterface = GetFirstInterface();  // Get first interface identifier

   // The following lines of code are essentially what the GetDHCPAddress() function
   // does to make the DHCP process easier.
   pDhcpObj = new  DhcpObject( FirstInterface );
   pDhcpObj->StartDHCP();    // Start DHCP

   // Pend on semaphore to verify an address was obtained
   if ( OSSemPend( &( pDhcpObj->NotifySem ), 10 * TICKS_PER_SECOND ) == OS_TIMEOUT )
//Wait 10 seconds
   {
      iprintf("\r\n\r\n*** WARNING ***\r\n");
      iprintf("IP Address was set to 0.0.0.0, and a DHCP server could not be
found.\r\n");
      iprintf("Device does not have a valid IP address.\r\n\r\n");
   }
   else
   {
      iprintf("DHCP assigned the following values:\r\n");
      iprintf("IP:   "); ShowIP(ib->netIP); iprintf("\r\n");
      iprintf("Mask: "); ShowIP(ib->netIpMask); iprintf("\r\n");
      iprintf("Gate: "); ShowIP(ib->netIpGate); iprintf("\r\n");
      iprintf("DNS:  "); ShowIP(ib->netDNS); iprintf("\r\n");
      iprintf("Interface Name: %s\r\n", ib->InterfaceName);
      iprintf("\r\n");
      AssignedDHCP = TRUE;
  }
}


/*-------------------------------------------------------------------
 UserMain task
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
   InitializeStack();
   pDHCPOfferName = DeviceName;   // Host name for DNS
```

```
int FirstInterface = GetFirstInterface();  // Get first interface identifier
InterfaceBlock *ib = GetInterFaceBlock(FirstInterface);  // Get interface data

if (ib->netIP == 0)  // Check IP address for 0.0.0.0, and use DHCP if necessary
{
   iprintf("\r\nNo static IP address set, attempting DHCP\r\n");
   MyStartDHCP( ib );
}

OSChangePrio(MAIN_PRIO);
EnableAutoUpdate();
StartHTTP();

DisplayUserMenu();
while (1)
{
    char c = getchar();
    switch (c)
    {
       case '1': // Display runtime IP values
          DisplayRuntimeIPSettings(FirstInterface);
          break;

       case '2': // Display Flash IP values
          // This example uses only 1 Configuration Record, 0
          DisplayFlashIPSettings(0);
          break;

       case '3':    // Change the runtime IP address and mask
          // After this function, the device will respond to the new IP settings
          ChangeRuntimeIPSettings(FirstInterface,              // Interface Block
                                  AsciiToIp("10.1.1.24"),      // New IP address
                                  AsciiToIp("255.255.255.0"),  // New IP mask
                                  AsciiToIp("10.1.1.1"),       // New IP gateway
                                  AsciiToIp("10.1.1.2"));      // New IP dns
          break;

       case '4':    // Change the Flash IP address and mask
          // If the device reboots, the new values will be in effect
          ChangeFlashIPSettings(0,               // Use first Config Record, 0
                                AsciiToIp("10.1.1.24"),      // New IP address
                                AsciiToIp("255.255.255.0"),  // New IP mask
                                AsciiToIp("10.1.1.1"),       // New IP gateway
                                AsciiToIp("10.1.1.2"));      // New IP dns
          break;

       case '5':    // Set all flash values to 0.0.0.0
          ChangeFlashIPSettings(0,               // Use first Config Record, 0
                                AsciiToIp("0.0.0.0"),      // New IP address
                                AsciiToIp("0.0.0.0"),      // New IP mask
                                AsciiToIp("0.0.0.0"),      // New IP gateway
                                AsciiToIp("0.0.0.0"));     // New IP dns

          ChangeRuntimeIPSettings(FirstInterface,              // Interface Block
                                  AsciiToIp("0.0.0.0"),      // New IP address
                                  AsciiToIp("0.0.0.0"),  // New IP mask
                                  AsciiToIp("0.0.0.0"),       // New IP gateway
                                  AsciiToIp("0.0.0.0"));      // New IP dns
          break;

      case '6':  // Start DHCP Client service
          if ( !AssignedDHCP )
          {
```

```
               iprintf("Contacting DHCP Server ...\r\n");
               MyStartDHCP( ib );
            }
            else
            {
               iprintf("\r\n*** Error: DHCP Client service is alredy running\r\n");
            }
            break;

         case '7':  // Stop DHCP Client service
            if ( AssignedDHCP )
            {
               pDhcpObj->StopDHCP();    // Stop DHCP
               AssignedDHCP = FALSE;    // Flag for this application
               iprintf("\r\nDHCP Release sent and DHCP Client service has been
stopped\r\n");
               DisplayRuntimeIPSettings(FirstInterface);
            }
            else
            {
               iprintf("\r\n*** Error: DHCP Client service is not running\r\n");
            }
            break;

          default:  DisplayUserMenu();
       }
   }
}
```

8 Basic Web Server Functions

8.1 Introduction

The NetBurner tools handle HTML pages, JAVA applets, Flash and images automatically. Any project that makes use of the Web Server features must have a subdirectory immediately under the project directory named "html". Just put all HTML files, JAVA applets, images, etc. in this html subdirectory and the NetBurner tools will automatically compile and link them into the application image that you download into your NetBurner device.

The Template program is now at the point where it will boot up, display its IP address (static or acquired by DHCP), and loop forever printing messages to stdout (the debug serial port). In this chapter, we will use the web server to display some very simple static content. The NetBurner Web Server excels at providing dynamic content as well, which will be discussed in the next chapter.

A web server is a specialized case of a generic TCP server that listens on the "well known port number" 80. The web server operates as a task that waits for incoming TCP connections on port 80, then delivers the requested content to the client - which is usually a web browser.

To initiate the transfer, the web browser sends a GET request. If no file name is specified in the GET request, a default file named index.htm or index.html is returned. The NetBurner Web Server assumes a default of index.htm (you can change this to html if you desire). Once the web server sends the requested data, it terminates the TCP connection.

To enable the web server and serve up pages to a web browser an application needs the following:

1. Add the `StartHTTP()` function call to start the Web Server
2. A directory named "html" in the project directory
3. Create a web page called index.htm.

All of the above steps are done automatically by the AppWizard if you select the appropriate checkbox items. Since we did this with the initial template program, all we need to do now is edit the HTML content in the index.htm file.

8.2 Edit the index.htm Web Page

Next we will edit the index.htm file created by the AppWizard. The page will contain some text and an image.

> **Using HTML Tools**
>
> You can use HTML tools such as Dreamweaver to create your web content. However, remember that EVERYTHING in the html directory is included in the application image. Some HTML tools can leave large project files in the html directory that will take up a large portion of your application space. You should remove any such files before building your project.

Edit the index.htm file as shown below:

```
<HTML>
<BODY>
<IMG SRC="Logo.jpg" BORDER="0">
<IMG SRC="SB72IO.jpg" BORDER="0">
<H1>Thank you for NetBurning!</H1><BR>
</BODY>
</HTML>
```

The <HTML> and <BODY> tags define the file as containing HTML content and provide delimiters for the body of the web page. The <IMG> tags are used to display the logo and board pictures. The text message “Thank you for NetBurning” is then displayed. The <H1> tags specify that the text be displayed as a header in larger bold font.

Note: The images are available with the project files as a download from www.netburner.com. If you do not have these files, you can simply delete the two lines with the <IMG> tags and display only the text.

Now compile and download the application. When you view the web page the output should look like the screen shot below:

9 Interactive Web Form Example

9.1 Introduction

Dynamic HTML is generated at run-time on your NetBurner device; it is not a static web page. For example, if you wanted to create a front panel for an instrument in which the HTML page would change depending on the current operating parameters and measurements - that would be Dynamic HTML. This can be as simple as putting values into predefined slots, or generating whole pages from scratch at run time. For more information about Dynamic HTML, see the flashform and tictactoe example projects (in C:\Nburn\examples).

The short version of Dynamic HTML: Dynamic HTML works just like normal HTML. It just requires a special TAG within the HTML files. Anywhere you want to include Dynamic HTML in your HTML files, you would add the TAG:

```
<!--FUNCTIONCALL YourFunctionName -->
```

Then, in your project source code, you must include this C function:

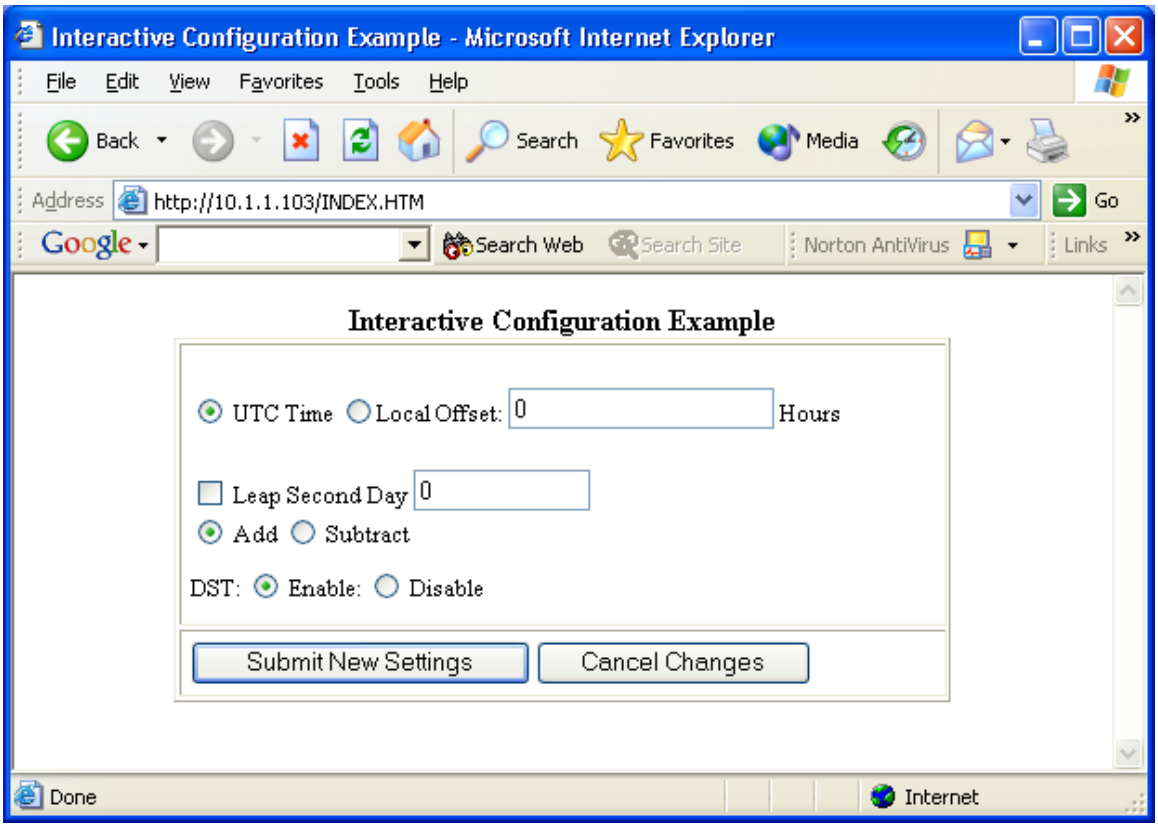
```
void YourFunctionName( int sock, PCSTR url )
{
   writestring( sock, buffer   // Data to output goes here );
}
```

Your function name is assumed to be a C function by the linker. Therefore, if you are using C++, you must include the following before your function definition:

```
extern "C"
{
  void YourFunctionName( int sock, PCSTR url );
}
```

This directive tells the C++ compiler to leave the function names intact by disabling C++“name mangling”.

In the previous chapter, our basic web page example demonstrated static web content. Your NetBurner platform can easily do dynamic content as well. In this example we will create a configuration web page interface that will provide submission and recall of changeable data by using HTML forms. Below is a screen shot of this application:



9.2 How to Use HTML Forms

You have probably encountered forms many times on the web, especially for ecommerce and feedback forms. The format is typically some number of text fields, checkboxes, radio buttons, combo boxes and a *submit* button. When you click on the *submit* button, the data from the form is sent to the web server as a HTTP POST. The web server then parses the data and takes appropriate action. If you have ever purchased anything on the web, filled out the order information, and clicked on a button like "confirm order" or "buy", you most likely submitted form information and were then redirected to a page confirming the order.

A form is defined in HTML by the <FORM></FORM> tags. User input is accomplished using the <INPUT> tag, representing text fields, checkboxes, radio buttons, etc.. For example,

```
<FORM ACTION="name.htm" METHOD="POST">
<INPUT TYPE="radio" VALUE="UTC" NAME="RadioGroup1" CHECKED="1"> UTC Time
<INPUT TYPE="radio" VALUE="Local" NAME="RadioGroup1">Local Offset:
<INPUT NAME="tfHours" VALUE="9" TYPE="text" SIZE="20"> Hours
<INPUT TYPE="submit" VALUE="Submit">
</FORM>
```

The above HTML source code shows the first few form items of the example explained in this chapter. The above form has two mutually exclusive radio buttons, one text field and the form submit button. The web browser will identify the input types and create the respective graphics on the web page. **Note:** The items in bold text will be

created dynamically by the application source code in our example program. The user will select a radio button, enter text in the text field, and finally click on the submit button. The user form values will then be sent to the web server as a POST.

9.3 Collecting User Input: Web Forms vs. URL's

There are two common methods for moving data from the client web browser to the web server on an embedded platform: HTML Forms using POST, and storing the data in the URL. The previous section described form operation. You have probably seen the URL method many times in e-commerce applications.

For example, the URL - *http://www.store.com/orderform?type=order123* is storing the data *type=order123* in the URL. Basically, everything following the '**?**' character is ignored by the browser, so your application can store whatever data it needs after the character. A big advantage of this method is that the application is stateful, meaning multiple users can access the same application and each user's session maintains its specific data in the URL.

9.4 Application Objectives

We will create a web page interface using forms that can:

- Modify and recall text fields, checkboxes and radio buttons
- Store and recall settings in flash memory using the Get/Set API function calls
- Use the Submit web page feature to modify settings
- Parse web form data submissions
- Use the FUNCTIONCALL HTML tag to link HTML I/O to C/C++ code

9.5 Application Files

Our application is divided up into the following files:

File Name	Description
main.cpp	This is the same as in the basic web example with one modification: the IP Address only prints to the debug port one time.
iadweb.cpp	Source code module for the interactive web functions.
iadweb.h	Header file for iadweb.cpp
Html\index.htm	HTML source file.

9.6 Interactive Web Functions

The source code file iadweb.cpp handles the web functions and interface for the application. It has the following functions:

- Definition and initialization of non-volatile flash storage
- Handles FUNCTIONCALL tags from the web page (index.htm)
- Handles web page POSTs, and extracts form data

The source code for iadweb.cpp is shown below:

```
#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <startnet.h>
#include <iosys.h>
#include <utils.h>
#include <ip.h>
#include <tcp.h>
#include <htmlfiles.h>
#include <http.h>
#include <string.h>
#include <basictypes.h>
#include "iadweb.h"


#define VERIFY_KEY (0x10220002)  // NV Settings key code


typedef struct NV_SettingsStruct // non-volatile storage structure
{
   DWORD  VerifyKey;
   BOOL   bUTC;
   int    nLocalOffset;
   BOOL   bCkboxLeapSecDay;
   int    nLeapSecDay;
   BOOL   bLeapSecDayAdd;
   BOOL   bEnableDST;
};


//----- global vars -----
char   gPostBuf[1000];
NV_SettingsStruct NV_Settings;
const char FirmwareVersion[] = "1.0, 12/17/2002";



/*--------------------------------------------------------------------
 Check NV Settings
 Assign default values if VerifyKey is not valid, otherwise load
 stored flash values.
 --------------------------------------------------------------------*/
void CheckNVSettings()
{
   NV_SettingsStruct *pData   = (NV_SettingsStruct *)GetUserParameters();
   NV_Settings.VerifyKey      = pData->VerifyKey;
   NV_Settings.bUTC       = pData->bUTC;
   NV_Settings.nLocalOffset   = pData->nLocalOffset;
```

```
49      NV_Settings.bCkboxLeapSecDay  = pData->bCkboxLeapSecDay;
50      NV_Settings.nLeapSecDay    = pData->nLeapSecDay;
51      NV_Settings.bLeapSecDayAdd = pData->bLeapSecDayAdd;
52      NV_Settings.bEnableDST     = pData->bEnableDST;
53
54      if (NV_Settings.VerifyKey != VERIFY_KEY)
55      {
56           iprintf("Reset NV Memory Defaults\n");
57
58           NV_Settings.VerifyKey    = VERIFY_KEY;
59           NV_Settings.bUTC         = TRUE;
60           NV_Settings.nLocalOffset = 0;
61           NV_Settings.bCkboxLeapSecDay = FALSE;
62           NV_Settings.nLeapSecDay      = 0;
63           NV_Settings.bLeapSecDayAdd   = TRUE;
64           NV_Settings.bEnableDST       = TRUE;
65
66           SaveUserParameters(&NV_Settings, sizeof(NV_Settings));
67      }
68   }
69
70
71
72   /*-------------------------------------------------------------------
73    -------------------------------------------------------------------*/
74   void WebUTC(int sock, PCSTR url)
75   {
76      iprintf("Entered WebUTC\r\n");
77      if (NV_Settings.bUTC)
78         writestring(sock, "CHECKED=\"1\" ");
79   }
80
81   /*-------------------------------------------------------------------
82    -------------------------------------------------------------------*/
83   void WebLocal(int sock, PCSTR url)
84   {
85      if (! NV_Settings.bUTC)
86         writestring(sock, "CHECKED=\"1\" ");
87   }
88
89
90   /*-------------------------------------------------------------------
91    -------------------------------------------------------------------*/
92   void WebLocalOffset(int sock, PCSTR url)
93   {
94      char buf[80];
95
96      sprintf(buf, "VALUE=\"%d\" ", NV_Settings.nLocalOffset);
97      writestring(sock, buf);
98   }
99
100
101
102  /*-------------------------------------------------------------------
103   -------------------------------------------------------------------*/
104  void WebLeapCkbox(int sock, PCSTR url)
105  {
106     if (NV_Settings.bCkboxLeapSecDay)
107     writestring(sock, "checked");
108  }
109
110  /*-------------------------------------------------------------------
111   -------------------------------------------------------------------*/
```

```
112  void WebLeapSecDay(int sock, PCSTR url)
113  {
114     char buf[80];
115
116     sprintf(buf, "VALUE=\"%d\" ", NV_Settings.nLeapSecDay);
117     writestring(sock, buf);
118  }
119
120  /*-------------------------------------------------------------------
121   -------------------------------------------------------------------*/
122  void WebLeapSecDayAdd(int sock, PCSTR url)
123  {
124     if (NV_Settings.bLeapSecDayAdd)
125        writestring(sock, "CHECKED=\"1\" ");
126  }
127
128  /*-------------------------------------------------------------------
129   -------------------------------------------------------------------*/
130  void WebLeapSecDaySub(int sock, PCSTR url)
131  {
132     if (! NV_Settings.bLeapSecDayAdd)
133        writestring(sock, "CHECKED=\"1\" ");
134  }
135
136  /*-------------------------------------------------------------------
137   -------------------------------------------------------------------*/
138  void WebEnableDST(int sock, PCSTR url)
139  {
140     if (NV_Settings.bEnableDST)
141        writestring(sock, "CHECKED=\"1\" ");
142  }
143
144  /*-------------------------------------------------------------------
145   -------------------------------------------------------------------*/
146  void WebDisableDST(int sock, PCSTR url)
147  {
148     if (! NV_Settings.bEnableDST)
149        writestring(sock, "CHECKED=\"1\" ");
150  }
151
152
153  /*-------------------------------------------------------------------
154   Handle HTTP Post
155
156   WARNING WARNING WARNING     CAUTION CAUTION CAUTION
157   The User data space is 8K in size.  The object we are storing is
158   less than 200 bytes long, thus it is probably ok to make an automatic
159   variable out of it.  It would NOT BE OK to make an 8K automatic
160   variable.  The choices for doing this with an 8K object...
161
162   First choice: make a global variable, this way the linker will
163   allocate space for it. All errors will be at link time not run time.
164
165   Second choice: Increase the HTTP stack size in constants.h and
166   recompile the whole system directory.
167
168   Third choice: use malloc and free.  The only problem is what do
169   you do if malloc fails?
170   -------------------------------------------------------------------*/
171  int MyDoPost(int sock, char * url, char * pData, char * rxBuffer)
172  {
173     int max_chars = 40;
174     char *buf = gPostBuf;   // post buffer is global
```

```
175
176       iprintf(pData);  // print all data sent from app
177
178
179       // Process UTC/Local radio buttons
180       if (ExtractPostData("radioUTC-Local", pData, buf, max_chars) == -1)
181          iprintf("Error reading post data: radioUTC\r\n");
182       else
183       {
184          if (strcmp(buf, "UTC") == 0)
185             NV_Settings.bUTC = TRUE;
186          else
187             NV_Settings.bUTC = FALSE;
188       }
189
190       // Process Local Offset Hours text field
191       if (ExtractPostData("tfLocalOffHours", pData, buf, max_chars) == -1)
192          iprintf( "Error reading post data: tfLocalOffHours\r\n");
193       else
194          NV_Settings.nLocalOffset = atoi(buf);
195
196       // Process Leap Second Day checkbox
197       if (ExtractPostData("ckboxLeapSecDay", pData, buf, max_chars) == -1)
198          NV_Settings.bCkboxLeapSecDay = FALSE;
199       else
200          NV_Settings.bCkboxLeapSecDay = TRUE;
201
202       // Process Leap Second Day text field
203       if (ExtractPostData("tfLeapSecDay", pData, buf, max_chars) == -1)
204          iprintf( "Error reading post data: tfLeapSecDay\r\n");
205       else
206          NV_Settings.nLeapSecDay = atoi(buf);
207
208       // Process Leap Sec Day Add/Sub radio buttons
209       if (ExtractPostData("radioLeapAddSub", pData, buf, max_chars) == -1)
210          iprintf("Error reading post data: radioLeapAddSub\r\n");
211       else
212       {
213          if (strcmp(buf, "Add") == 0)
214             NV_Settings.bLeapSecDayAdd = TRUE;
215           else
216             NV_Settings.bLeapSecDayAdd = FALSE;
217       }
218
219       // Process DST radio buttons
220       if (ExtractPostData("radioDST", pData, buf, max_chars) == -1)
221          iprintf("Error reading post data: radioDST\r\n");
222      else
223      {
224       if (strcmp(buf, "Enable") == 0)
225          NV_Settings.bEnableDST = TRUE;
226         else
227          NV_Settings.bEnableDST = FALSE;
228       }
229
230
231       // Now to store it in flash.
232       // WARNING: If new settings are added, remember to add them to
233       //          NV Settings default initialization!
234      NV_Settings.VerifyKey = VERIFY_KEY;
235       if ( SaveUserParameters(&NV_Settings, sizeof(NV_Settings)) != 0 )
236          iprintf( "New Settings Saved\r\n");
237       else
```

```
238         iprintf( "ERROR: Could not save new settings\r\n");
239
240
241       // We have to respond to the post with a new HTML page. In
242       // this case we will redirect so the browser will go to
243       // that URL for the response.
244       RedirectResponse(sock, "index.htm");
245
246       return 0;
247    }
248
249
250    /*-------------------------------------------------------------------
251     Register Post
252     This function sets the HTTP POST handler to point to our
253     function.
254     -------------------------------------------------------------------*/
255    void RegisterPost()
256    {
257       SetNewPostHandler(MyDoPost);
258    }
```

Line 16: VERIFY_KEY is used to determine if the flash memory space contains valid data. If the key is not found, then the flash memory is initialized with default values. This is useful to handle two conditions: when the hardware is used for the first time, and when a software modification changes the non-volatile data structure (the key would be incremented to a new value to force initialization again).

Lines 19-28: This is the non-volatile memory structure. The user parameter flash storage area is an 8k flash sector (8k by default, but can be expanded). Applications store and recall data to this sector by writing the entire structure with the API calls SaveUserParameters( ) and GetUserParameters( ).

Lines 72 – 150: Each FUNCTIONCALL tag in the index.htm file has a corresponding function defined in the format: `void foo(int sock, PCSTR url),` where *foo* is the name of the function, *sock* is a handle to the network socket connection, and *url* is the actual URL that was sent to the web server from the web browser. Some or all of these parameters will be used depending on your application. If the function needs to write data to the web client, then the write( ) function can be used with the *sock* parameter. This can be useful for filling in data fields or generating graphs.

The important thing to remember is that the client web browser will not see the FUNCTIONCALL tag. As far as the browser is concerned, all the data is just an input stream from a TCP connection. To verify this, you can view the HTML source in your web browser. All the FUNCTIONCALL tags should be replaced with HTML data. In this example application, most of the functions fill in a piece of a HTML tag to show text field data or remember which radio button was selected in a group.

Your application does not need to explicitly call any of these functions. As the web sever is delivering the web content to the web browser, it will automatically call each function with a FUNCTIONCALL tag.

Line 171: The MyDoPost( ) function will be called whenever a HTML form is submitted, which occurs when a user selects the “submit” button in a HTML form. The web server has a standard function to handle POSTs; the function RegisterPost( ) is used to intercept a POST and send it to MyDoPost( ). The return value of MyDoPost( ) is not currently used, but currently should return a value of 0 to be compatible with possible future expansin.

The parameters passed in MyDoPost( ) are:

Parameter Type	Description
sock	The file descriptor for the socket that this function should send a response to.
url	A string containing the URL of the POST destination. This field is used to determine which form the POST came from.
pData	A pointer to a string containing the form data (i.e. text fields, radio button and check box selections, etc)
rxbuffer	A pointer to the entire HTTP request. This is not normally needed.

Data is extracted from the form using the function:

```
int ExtractPostData(PCSTR name,PCSTR data, PSTR dest_buffer,int maxlen)
```

This function takes the HTML post data sent to the POST function, extracts the data associated with a specific *name* and returns it in *dest_buffer*. For example,

```
ExtractPostData("radioUTC-Local", pData, buf, max_chars)
```

looks for the name *radioUTC-Local* in the string *pData* and returns the data portion in *buf*. The *max_chars* parameter specifies the maximum length of characters to store in *buf* to avoid an overflow.

Once the data is stored in *buf*, the example application stores the value in an array and the array is written to the user flash storage area with the function:

```
SaveUserParameters(&NV_Settings, sizeof(NV_Settings))
```

SaveUserParameters( ) is an API function that will take a pointer to the data to be stored, in this case the address of *NV_Settings*, and a second parameter representing the size of the data. This is easily done by using a structure and the *sizeof( )* function.

10 Dynamic Web Content using the VARIABLE Tag

Most applications using the web server have a need to display dynamic content for users who connect to the web server. Dynamic content can be presented in a web page by embedding VARIABLE and FUNCTIONCALL HTML tags in the web page source code. The tags are embedded in and HTML comments so they do not affect the HTML presentation. As the NetBurner web server delivers code to a client, these tags are processed in real-time so that the dynamic content is transmitted in place of the tag. The tags are as follows:

FUNCTIONCALL	Calls a 'C' function in your application
VARIABLE <var>	Displays the specified variable
VARIABLE <func(param,…)>	Calls a C++ function with parameters

10.1 The FUNCTIONCALL Tag

The HTML FUNCTIONCALL tag will cause the web server to call the 'C' function associated with the tag. The 'C' function is passed the socket file descriptor and URL, and can use these parameters to do whatever the application needs to do such as displaying real time data, text and graphics. The previous section covered the FUNCTIONCALL tag. It is mentioned in this section because of its similarity with dynamic content. If you need specify a function call with parameters, please refer to the following section on using the VARIABLE tag to create a function call with parameters.

10.2 The CPPCALL Tag

The HTML CPPCALL tag behaves exactly the same as the FUNCTIONCALL tag, except that it will cause the webserver to call the 'C++' function associated with the tag, instead of a 'C' function.

10.3 The VARIABLE Tag

Variables in an application can be displayed on a web page using the VARIABLE tag. This can be useful for displaying the dynamic information used in the application, such as time, IP address, temperature, etc. The format of the tag is:

```
<!--VARIABLE <name> -->
```

Where "name" is the name of the application variable or an expression. For example, the system time tick variable, TimeTick can be displayed with:

```
<!--VARIABLE TimeTick -->
```

Or you can display the time in seconds with the equation:

```
<!--VARIABLE TimeTick/TICKS_PER_SECOND -->
```

The VARIABLE tag is processed during the compilation of the application by parsing the text between `<!--VARIABLE and the trailing -->` and converting it into a function call like:

WriteHtmlVariable(fd, TimeTick/TICKS_PER_SECOND);

The variable types are handled with C++, but you do not need to know anything about C++ to use this feature. The parameter types are defined by the function definitions located in c:\nburn\include\htmlfiles.h:

```
void WriteHtmlVariable(int fd, char c);
void WriteHtmlVariable(int fd, int i);
void WriteHtmlVariable(int fd, short i);
void WriteHtmlVariable(int fd, long i);
void WriteHtmlVariable(int fd, BYTE b);
void WriteHtmlVariable(int fd, WORD w);
void WriteHtmlVariable(int fd, unsigned long dw);
void WriteHtmlVariable(int fd, const char *);
void WriteHtmlVariable(int fd, MACADR ip);
```

In addition, we have included a class named IPCAST( ) that takes a 32-bit value and converts it into an IP address format (e.g. 192.168.1.2). The example below will display the IP address in dotted notation, rather than a 32-bit integer.

```
IP address: <!--VARIABLE IPCAST(gConfigRec.ip_Addr) !-->
```

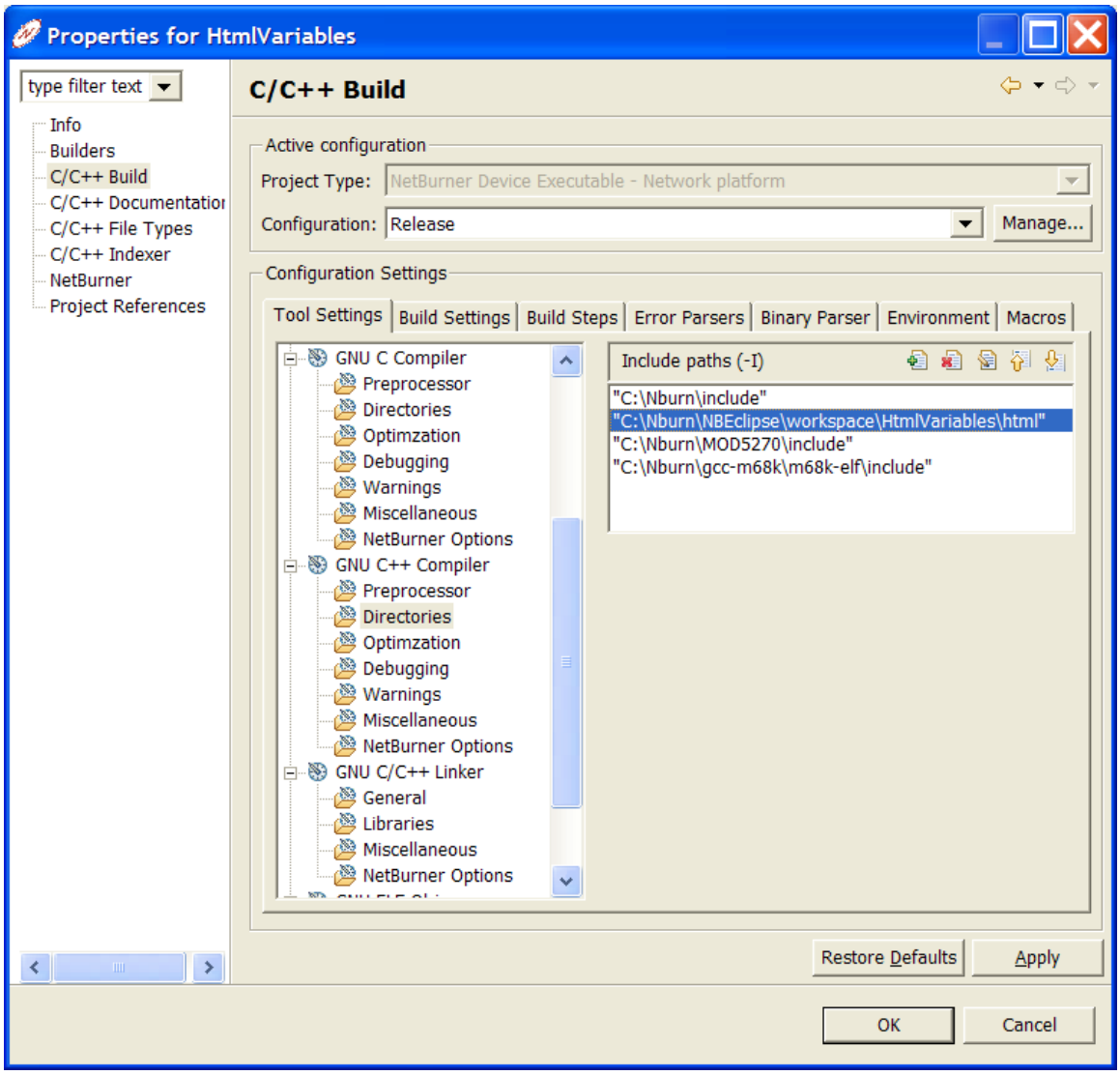
10.3.1 The INCLUDE Tag and htmlvar.h Header File

Now that we understand the resultant code is a function with the variable name, it should be apparent that the resultant htmldata.cpp file created by the HTML source code needs to be able to link to the variable name. For example, to display TimeTick the application would need to include utils.h, otherwise a linker error will occur.

Include files can be handled two ways: you can use an INCLUDE tag in the HTML code, or you can create a header file with the name “htmlvar.h” that will be automatically included if no INCLUDE tags are detected in the HTML source code. If you choose to use the htmlvar.h header file, it must be located in the same directory as the htmldata.cpp file, or if you are using NBEclipse, it must be in the include file path. To add the include file path using NBEclipse:

- Right-click on your project and select properties
- Select "C/C++ Build" options
- Select "GNU C++ Compiler" ➢ Directories
- Use the "+" icon in the include path box to add the project's "HtmlVariables\html" folder to the list of include paths.

A screen capture of NBEclipse with the path added is shown below:



Example htmlvar.h file:

```
#ifndef HTMLVARS_H_
#define HTMLVARS_H_

#include <constants.h>
#include <system.h>
#include <startnet.h>

const char * FooWithParameters(int fd, int v);

#endif /*HTMLVARS_H_*/
```

Example HTML file using the INCLUDE tag:

```
<HTML>
<BODY>
<!--INCLUDE foobar.h -->
Value = <!--VARIABLE MyVar --><BR>
</BODY>
</HTML>
```

10.3.2 Calling a Function with Parameters

If you need to specify a function call, but need to pass a parameter, the FUNCTIONCALL tag will not work because the parameters are fixed as the socket fd and URL. In this case we can use the VARIABLE tag to achieve the functionality of calling a function with a variable.

The include file (e.g. htmlvar.h) must specify the function definition in the format below. In this case we are passing an integer value 'v'. The first parameter must always be the socket fd.

```
const char * MyFunction(int fd,int v);
```

The HTML source code then used the VARIABLE tag with the function definition below. In this example we are passing the integer value of 1.

```
<!--VARIABLE MyFunction(fd,1) -->
```

When the application is compiled the resultant function call will be:

```
WriteHtmlVariable( fd, MyFunction(fd,1) );
```

This function returns an empty string, which will have no effect on the web page. An example of what a function might do is shown below:

```
const char * MyFunction(int fd, int v)
{
   char buffer[255];
   siprintf( buffer, "MyFunction() was called with v = %d\r\n", v );
   writestring( fd, buffer );

   return "\0"; /* Return a const char * here of zero length so it wont
                   print anything to the HTML page. */
}
```

10.3.3 Creating Custom Structures or Classes

The VARIABLE functionality can be extended to support user defined types. This would most commonly be used to display a use defines structure. Lets say you have a Class you want to display on a web page called MyClass:

```
struct  my_struct {
  int i;
  char buf[80];
  DWORD  dVal;
} MY_STRUCT;

MY_STRUCT MyStruct;
```

In your include file add the function definition:

```
void WriteHtmlVariable(int fd, MY_STRUCT MyStruct);
```

You can display it on the web page with the VARIABLE tag:

```
<!--VARIABLE MyStruct -->
```

What this look like behind the scenes is:

```
WriteHtmlVariable( fd, MyStruct );
```

Note that you still have to write the implementation of the above function. The function below is the source code for the MAC address type already defined:

```
void WriteHtmlVariable(int fd, MACADR ma)
{
   PBYTE lpb = ( PBYTE ) &ma;
   for (int I = 0; I < 5; i++)
   {
      fd_writehex( fd, lpb[i] );
      write(fd,":",1);
   }
   fd_writehex( fd, lpb[6] );
}
```

11 TCP vs UDP

A very common question that arises when designing a network application is whether to use TCP (Transmission Control Protocol) or UDP (User Datagram Protocol). There are a few guidelines and features that can determine which would be the preferred protocol. Some of the issues discussed may not mean much before your read the TCP and UDP sections later in this guide, but discussing these issues now may make you aware of certain issues as you read those sections.

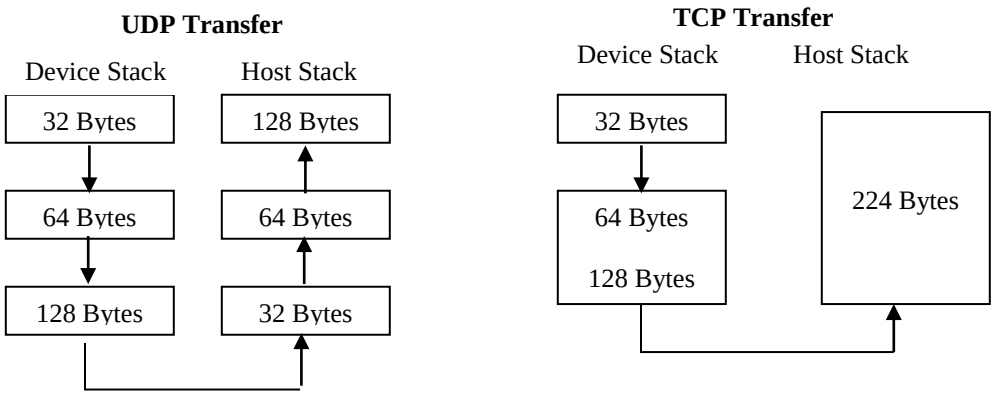
In general, TCP is a *connection oriented byte stream* used for point-to-point communications where reliability and sequencing of data is required. TCP handles retransmission of data, acknowledgements, error detection and will guarantee the data received will be sequenced in the same order as it was sent.

UDP can be thought of as a scaled down protocol as compared with TCP. It is a *connectionless* protocol that does not guarantee delivery and does not sequence the segments (although each segment is numbered). UDP is a "send and forget" protocol that does not use acknowledgements. A common comparison is that TCP is similar to a phone call and UDP is similar to a post card (although the time difference between a phone call and the mail must be ignored). With TCP, you connect to a specific destination phone number. When that person answers they say "hello", you say "Hi, my name is Bob", and then the conversation continues with each side speaking and responding (in a well-behaved conversation!). With UDP you essentially transmit a datagram, like writing on a post card, and send it without verifying it was received or undamaged along the way.

When choosing between TCP and UDP, some major concerns are the overhead it takes to establish a TCP connection, speed and the reliability of data transmission. For example, SNMP uses UDP. SNMP is used to monitor a network and sends a lot of messages for status updates and alerts. If TCP were used, the overhead of establishing, maintaining and closing a connection for each message would bog down the network leaving little room for other traffic. A second example of when UDP is a better choice is when an application handles its own reliability at the application layer. Using TCP in this instance would be redundant. The NetBurner UDP Class implementation also has a speed advantage over TCP. Data stored in a UDP packet is dispatched by providing the stack functions with a pointer to the data – it does not copy the data to separate buffer locations. This factor alone should provide a 30% speed increase over TCP, which must be buffered.

In addition, TCP is a stream based protocol and UDP is a datagram based protocol. Let's take an example of an application in which a NetBurner device takes A/D readings and sends them to another network device or host PC. Using UDP, each output operation (i.e. creating and sending a UDP packet) results in exactly one IP packet being created and sent. The result of taking and sending 10 A/D readings is that the host will receive 10 individual packets, each containing one reading. The host PC can then easily identify each reading, although each reading will have to be sent with a sequence number so that the reading order can be recreated. If TCP is used with a single continuous network connection (i.e. the connection is not closed and reestablished for each reading), you do not have control over how many readings are send with each IP packet. You would need to add start message and stop message identifiers to separate the data from each reading.

An example of sending data of different sizes using TCP vs UDP is illustrated below:



Applications that use TCP are: HTTP, FTP, Telnet and SMTP. Applications that use UDP are: DHCP, BOOTP, SNMP and DNS.

12 TCP - Transmission Control Protocol

TCP is used to create a reliable byte stream connection between two devices. Usually one device is called a *server* and the other a *client.* The first sentence contains important descriptive words that you need to consider when choosing between TCP and UDP:

Connection-oriented: The devices must establish a connection before they can exchange data

Reliable: TCP uses acknowledgements, retransmissions, checksums and flow control

Stream based: Although TCP and UDP both use IP, TCP sends information as a byte stream. There are no record markers to delimit the data. For example, if a server device is sending analog-to-digital (A/D) readings to a client device, the client will see a stream of digits; TCP will not automatically insert delimiters to allow the client to determine where one measurement ends and the next begins. To the client, the stream may look like: "98273129323424". Even if the client knew each reading was 4 digits, it would not know where one ended and the next began.

In contrast to TCP, UDP (covered in the next chapter) is an unreliable, datagram-oriented connectionless protocol. Delivery is not guaranteed, but each output operation creates and sends one UDP datagram. In the above A/D example, each reading (or some number of multiple readings) could be sent as a single datagram and the client could then process one datagram at a time.

> **Network Bandwidth and Packet Size**
>
> Network bandwidth and packet size are significant considerations when writing an application. In practice, an application such as the A/D example would probably create packets with a "number of reading" field, or send a fixed multiple number of readings in each packet.

12.1 TCP Server Introduction

A TCP server is basically an application that creates a listening socket, and then listens on the socket for incoming connections. When an incoming connection is detected, the connection is then accepted. A web server is an example of a familiar TCP server. A web server listens on "well-known" port number 80 for incoming connections. Once a connection is established, the web browser will send a GET request to the web server, which will then send the requested information and terminate the connection. A web server is just a specific case of a TCP server.

To connect to a TCP server you must specify a *port number*. A port number is a 16-bit value. Since you must know the port number before connecting, many port numbers have been defined for common protocols, and are called well-known port numbers. Some of these values are shown below:

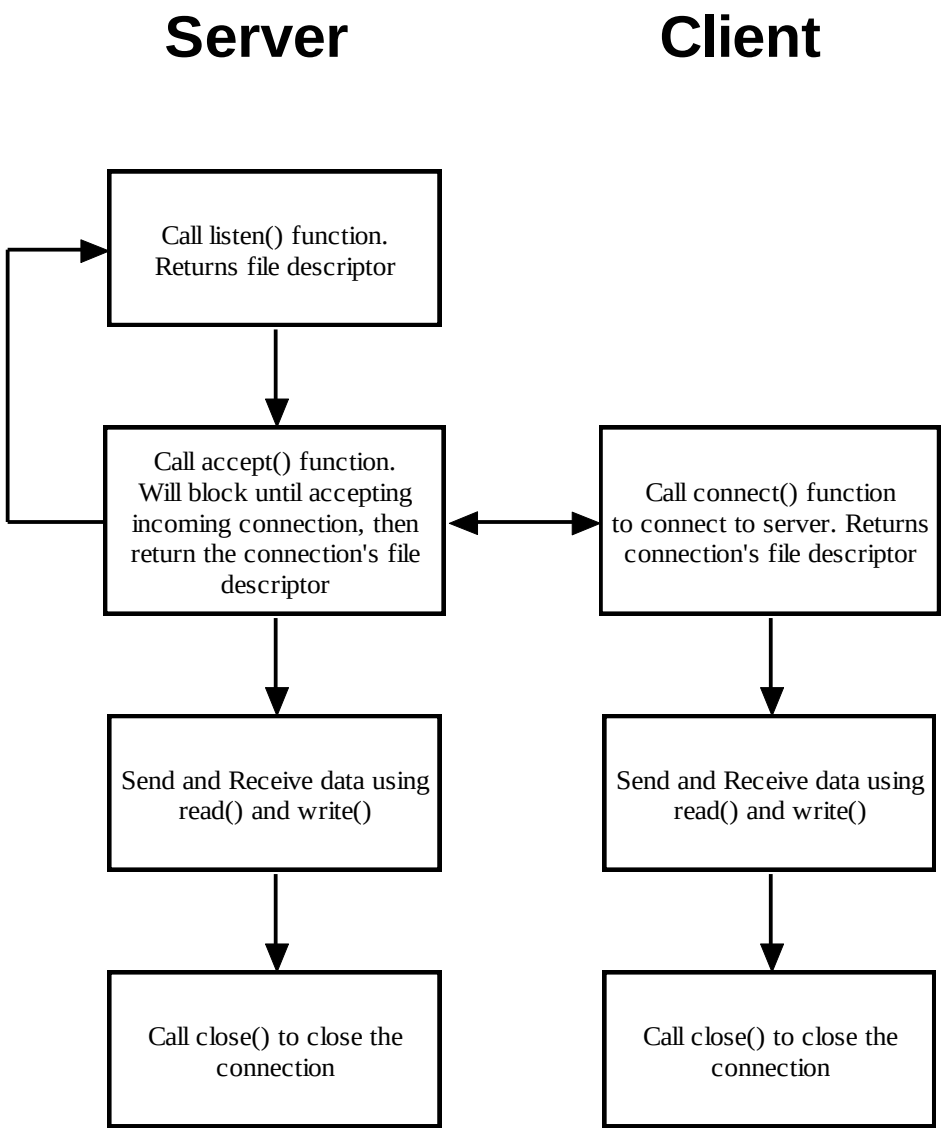
FTP	21
Telnet	23
SMTP	25
DNS	53
TFTP	69
HTTP	80
POP3	110
NTP	123

An important thing to remember is that it takes 4 parameters to define any TCP connection: Source IP address, Source port number, Destination IP address and Destination port number.

12.2 Writing a NetBurner TCP Server

Writing a TCP server on the NetBurner platform is very straightforward:

1. Create a listening socket with the `listen()` function
2. Wait for an incoming connection with the blocking call `accept()`



12.2.1 Simple TCP Server Application Source Code

This example program will listen on port 23 for incoming connections, send a sign on message to the client when a connection is made, and display all received data to the debug serial port. A telnet program on a host PC will be used to connect to the server as a client.

```
/*-------------------------------------------------------------------
 Description: NetBurner Simple TCP/IP Server Example Program
 This program will create a TCP server task, which listens on port
 23 by default. To test the application you can use Telnet. For example,
 from a windows commpan prompt, type "telent <ip address of netburner>.

 This example uses a simple read() function to receive data from a
 TCP Client.
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <startnet.h>
#include <autoupdate.h>
#include <startnet.h>
#include <dhcpclient.h>
#include <tcp.h>

#define TCP_LISTEN_PORT 23   // Telent port number
#define RX_BUFSIZE (4096)


const char *AppName = "Simple TCP Server Example";


//----- Global Vars -----
char RXBuffer[RX_BUFSIZE];

extern "C" {
   void UserMain(void * pd);
}


/*-------------------------------------------------------------------
 Convert IP address to a string
 -------------------------------------------------------------------*/
void IPtoString(IPADDR  ia, char *s)
{
    PBYTE ipb= (PBYTE)&ia;
    siprintf(s, "%d.%d.%d.%d",(int)ipb[0],(int)ipb[1],(int)ipb[2],(int)ipb[3]);
}


// Allocate task stack for UDP listen task
DWORD   TcpServerTaskStack[USER_TASK_STK_SIZE];

/*-------------------------------------------------------------------
 TCP Server Task
 -------------------------------------------------------------------*/
void TcpServerTask(void * pd)
{
    int ListenPort = (int) pd;

    // Set up the listening TCP socket
    int fdListen = listen(INADDR_ANY, ListenPort, 5);

    if (fdListen > 0)
```

```
   {
      IPADDR client_addr;
      WORD   port;

      while(1)
      {
           // The accept() function will block until a TCP client requests
           // a connection. Once a client connection is accepting, the
           // file descriptor fdnet is used to read/write to it.
         iprintf( "Wainting for connection on port %d...\n", ListenPort );
         int fdnet = accept(fdListen, &client_addr, &port, 0);

         iprintf("Connected to: "); ShowIP(client_addr);
         iprintf(":%d\n", port);

         writestring(fdnet, "Welcome to the NetBurner TCP Server\r\n");
         char s[20];
         IPtoString(EthernetIP, s);
         siprintf(RXBuffer, "You are connected to IP Address %s, port %d\r\n",
                             s, TCP_LISTEN_PORT);
         writestring(fdnet, RXBuffer);

         while (fdnet > 0)
         {
            /* Loop while connection is valid. The read() function will return
               0 or a negative number if the client closes the connection, so
               we test the return value in the loop. Note: you can also use
               ReadWithTimout() in place of read to enable the connection to
               terminate after a period of inactivity.
            */
               int n = 0;
            do {
               n = read( fdnet, RXBuffer, RX_BUFSIZE - 1 );
               RXBuffer[n] = '\0';
               iprintf( "Read %d bytes: %s\n", n, RXBuffer );
            } while ( n > 0 );

            // Don't foreget to close !
            iprintf("Closing client connection: ");
            ShowIP(client_addr);
               iprintf(":%d\n", port);
            close(fdnet);
            fdnet = 0;
         }
      } // while(1)
   } // while listen
}


/*-------------------------------------------------------------------
 User Main
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
   InitializeStack();      // Initialize the TCP/IP Stack

    if ( EthernetIP == 0 )
    {
       iprintf( "Trying DHCP\r\n" );
       GetDHCPAddress();
       iprintf( "DHCP assigned the IP address of :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
```

```
    }
    else
    {
       iprintf( "Static IP address set to :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }

   EnableAutoUpdate();     // Enable network code updates
   OSChangePrio(MAIN_PRIO);   // Set this task priority

    #ifdef _DEBUG
    InitializeNetworkGDB_and_Wait();
    #endif


   // Create TCP Server task
   OSTaskCreate( TcpServerTask,
               (void  *)TCP_LISTEN_PORT,
                  &TcpServerTaskStack[USER_TASK_STK_SIZE] ,
                  TcpServerTaskStack,
                  MAIN_PRIO - 1); // higher priority than UserMain

   while (1)
   {
      OSTimeDly( TICKS_PER_SECOND * 5 );
   }
}
```

This is an extremely simple example designed to illustrate how the `accept()` and `listen()` calls operate. It only listens to a single port number, and processes a single connection at a time. Any information sent from the Client will be displayed in the MTTTY window. The application does not have the capability to terminate the incoming connection.

We have added the #defines for the TCP listen port number, and the incoming TCP buffer storage array size. RXBuffer[] is then declared and will hold the received data. The `listen()` function call sets up a socket to listen for an incoming connection from any IP address on port number 23, the Telnet port number.

If the `listen()` succeeds , we then enter a second while loop. The application will then block at the `accept()` function call until an incoming connection is detected, such as when we run the Telnet program on the PC. When this connection is established, the `accept()` function returns and the sign-on message is sent to the Telnet application.

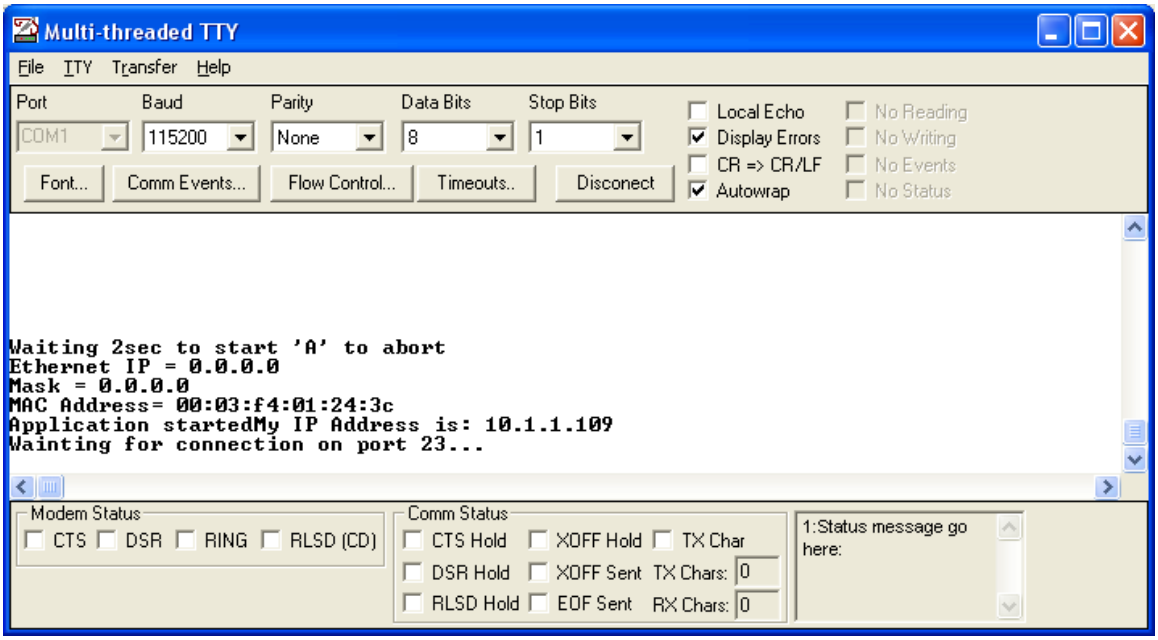
We now enter the do loop: `while(n > 0)`. The `read()` function will block until data is received or an error occurs such as the client terminating the connection. When data is sent from the Telnet application, the `read()` function will return with the data in the RXBuffer[] array. The application will stay in this while loop until the connection is terminated by the Telnet client (or you reset the NetBurner device). If the connection is broken by the Telnet client, the application will then loop back to the `accept()` function call and wait for another incoming connection.

Once application initialization is compete and we are running inside the while loop, the application will block on the `accept()` function call.

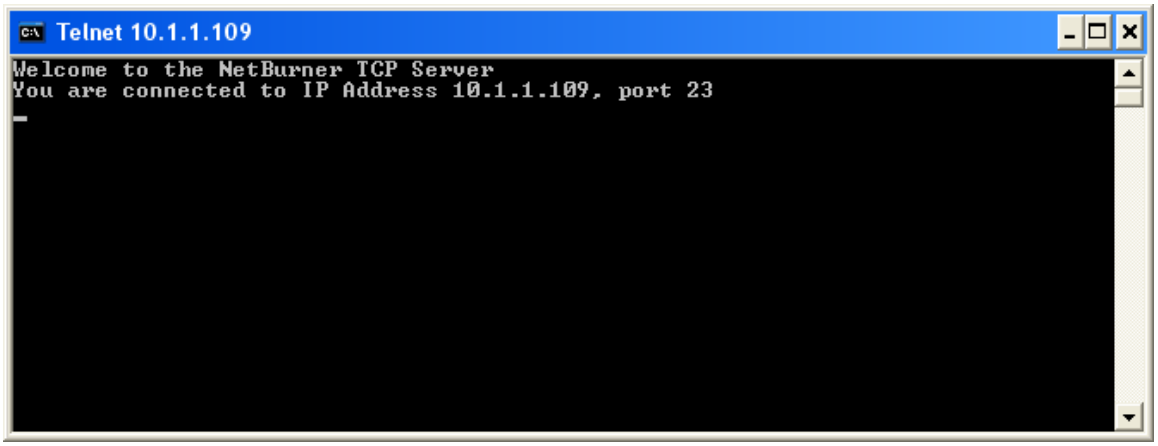
The IPtoString( ) function is used to convert a numerical IP address of type IPADDR to an ASCII string so it can be sent to the client as a sign-on message.

12.2.2 Running the NetBurner TCP Server Application

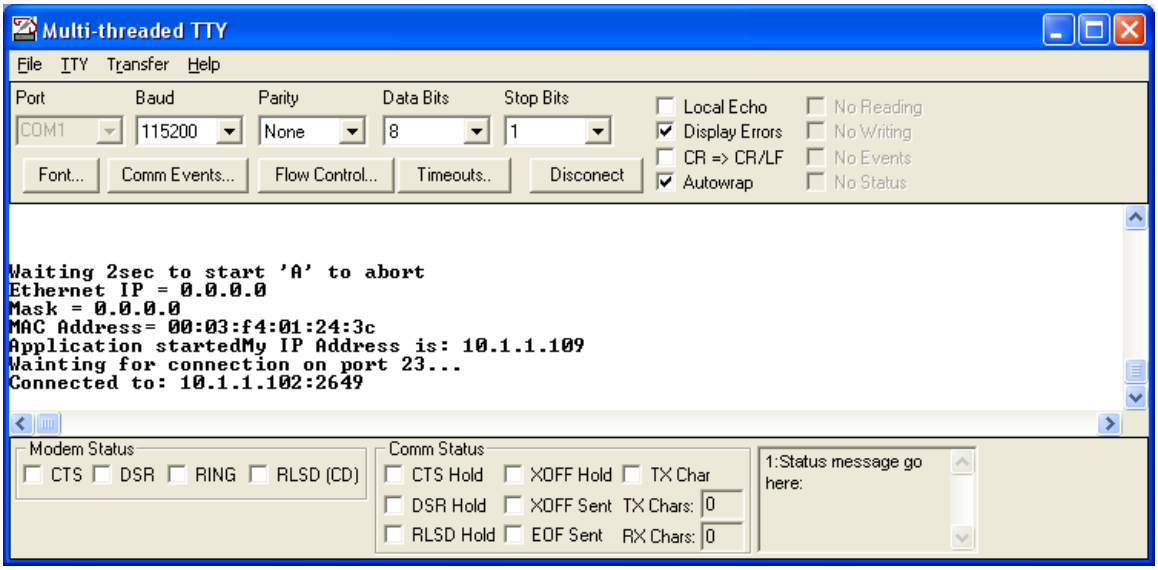
1. (If you have not already done so) Connect a serial cable from your host computer's serial port to your NetBurner board's debug serial port.
2. Start MTTTY and connect
3. Compile and download your application to your NetBurner board.
4. Verify that you can see the debug messages for the TCP Server. Note the IP address:



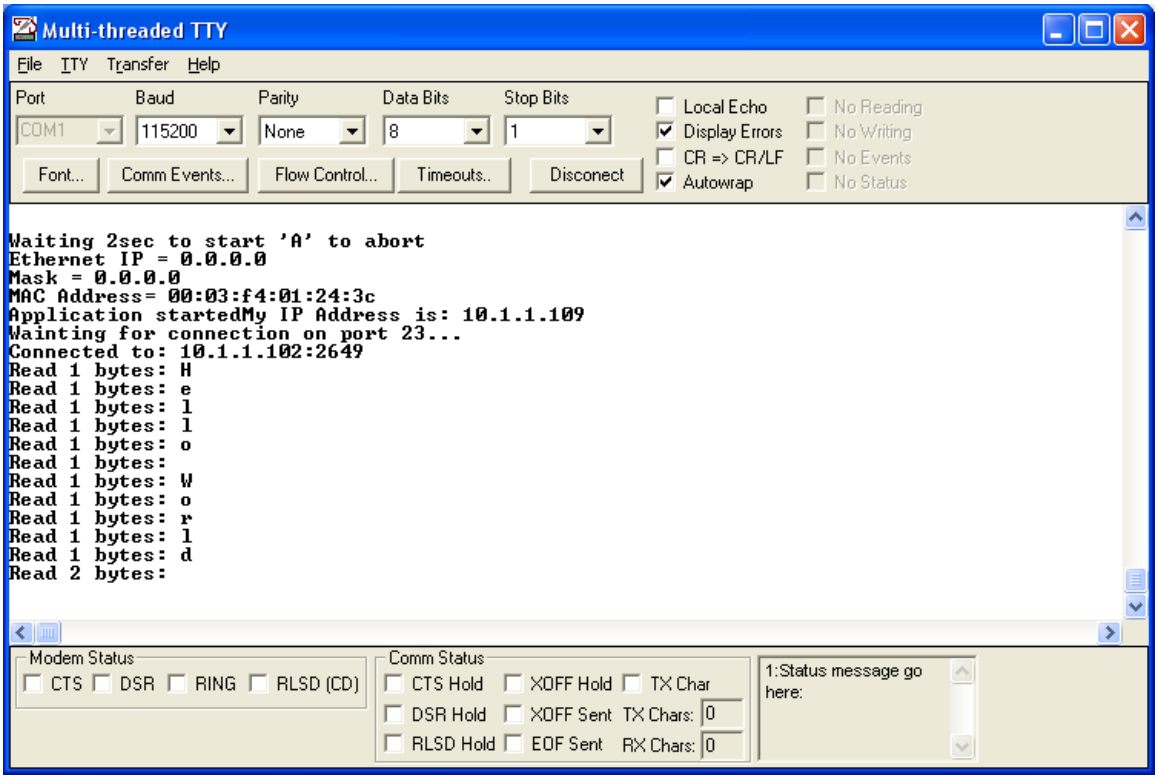
5. Run the telnet program on your host computer. This can be done from the Windows Start menu by selecting Start →Run or by opening a DOS window and simply typing “telnet <ip_addr>”, where <ip_adder> is replaced by the IP address of your NetBurner device; when finished press the "Enter" key. For example, “`telnet 10.1.1.117`”. **Note:** You do not need to specify the port number, since the telnet default port number is 23. If you use a different port number, then specify it in the telnet command after the IP address, for example: “`telnet 10.1.1.117 4332`”. You should see the following screen on a successful connection:



The MTTTY window should confirm the connection:



6. Now type some characters in the Telnet window. You should see those characters displayed in the MTTTY window as shown below:



12.2.3 Simple TCP Server Using the select( ) Function

In the previous example the TCP server processed only a single incoming connection. The select( ) function has the ability to pend on multiple file descriptors, which are used for TCP or serial connections. The example below demonstrates how the TCP Server can be written using select( ).

```
/*-------------------------------------------------------------------
 Description: NetBurner TCP/IP Server Example Program
 This program will create a TCP server task, which listens on port
 23 by default. To test the application you can use Telnet. For example,
 from a windows commpan prompt, type "telent <ip address of netburner>.

 This example uses the select() function to process the TCP connection.
 The real value of select() is that it can pend on multiple file
 descriptors, not just one as in this example. We are using one just
 to illustrate how it can be used.

 A timeout is used to close the connection if no data is received
 in 10 timeout periods.
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <startnet.h>
#include <autoupdate.h>
#include <startnet.h>
#include <dhcpclient.h>
#include <tcp.h>

#define TCP_LISTEN_PORT 23   // Telent port number
#define RX_BUFSIZE (4096)

const char *AppName = "TCP Server with Select() Example";

//----- Global Vars -----
char RXBuffer[RX_BUFSIZE];

extern "C" {
   void UserMain(void * pd);
}


/*-------------------------------------------------------------------
 Convert IP address to a string
 -------------------------------------------------------------------*/
void IPtoString(IPADDR  ia, char *s)
{
    PBYTE ipb= (PBYTE)&ia;
    siprintf(s, "%d.%d.%d.%d",(int)ipb[0],(int)ipb[1],(int)ipb[2],(int)ipb[3]);
}


// Allocate task stack for UDP listen task
DWORD   TcpServerTaskStack[USER_TASK_STK_SIZE];

/*-------------------------------------------------------------------
 TCP Server Task
 -------------------------------------------------------------------*/
void TcpServerTask(void * pd)
{
    int ListenPort = (int) pd;
```

```
// Set up the listening TCP socket
int fdListen = listen(INADDR_ANY, ListenPort, 5);

if (fdListen > 0)
{
    IPADDR client_addr;
    WORD   port;

    while(1)
    {
          // The accept() function will block until a TCP client requests
          // a connection. Once a client connection is accepting, the
          // file descriptor fdnet is used to read/write to it.
       iprintf( "Wainting for connection on port %d...\n", ListenPort );
       int fdnet = accept(fdListen, &client_addr, &port, 0);

       iprintf("Connected to: "); ShowIP(client_addr);
       iprintf(":%d\n", port);

       writestring(fdnet, "Welcome to the NetBurner TCP Server\r\n");
       char s[20];
       IPtoString(EthernetIP, s);
       siprintf(RXBuffer, "You are connected to IP Address %s, port %d\r\n",
                          s, TCP_LISTEN_PORT);
       writestring(fdnet, RXBuffer);

       int tcp_timeout = 0;
       while (fdnet > 0)
       {
             // declare file descriptor sets
          fd_set read_fds;
          fd_set error_fds;

             // Clear all bits in fd sets
          FD_ZERO(&read_fds);
          FD_ZERO(&error_fds);

             // Set bits in fd sets for our fd, fdnet
          FD_SET(fdnet, &read_fds);
          FD_SET(fdnet, &error_fds);

             // Select will block until an event occurs and a fd set bit is
             // set, or a timeout occurs. A timeout is ok in this situation,
             // we just use it to illustrate the task is still running.
          if (select(FD_SETSIZE, &read_fds,(fd_set *)0, &error_fds,
                     30 * TICKS_PER_SECOND))
          {
              if (FD_ISSET(fdnet,&read_fds))    // Network data available
              {
                 int n = read(fdnet, RXBuffer, RX_BUFSIZE - 1);
                 RXBuffer[n] = '\0';
                 iprintf("Read %d bytes: %s\n", n, RXBuffer);
                 //writestring(fdnet, RXBuffer);
              }

              if (FD_ISSET(fdnet, &error_fds) && !FD_ISSET(fdnet, &read_fds))
              // Network error condition
              {
                 iprintf("Network Error, closing socket\n");
                 close(fdnet);
                 fdnet = 0;
              }
          }
```

```
            else
            {   // Select timed out. If timeout exceeds 10 timeout
                // periods, the connection will be closed.
                iprintf("select() timeout\n");
                tcp_timeout++;
                if (tcp_timeout > 10)
                {
                    iprintf("Connection timed out, closing socket\n");
                    close(fdnet);
                    fdnet = 0;
                }
            } // Select
        } // While fdnet is valid
    } // while(1)
} // while listen
}


/*-------------------------------------------------------------------
 User Main
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
    InitializeStack();      // Initialize the TCP/IP Stack

    if ( EthernetIP == 0 )
    {
       iprintf( "Trying DHCP\r\n" );
       GetDHCPAddress();
       iprintf( "DHCP assigned the IP address of :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }
    else
    {
       iprintf( "Static IP address set to :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }

    EnableAutoUpdate();     // Enable network code updates
    OSChangePrio(MAIN_PRIO);   // Set this task priority

    #ifdef _DEBUG
    InitializeNetworkGDB_and_Wait();
    #endif


    // Create TCP Server task
    OSTaskCreate( TcpServerTask,
                (void  *)TCP_LISTEN_PORT,
                   &TcpServerTaskStack[USER_TASK_STK_SIZE] ,
                   TcpServerTaskStack,
                   MAIN_PRIO - 1);  // higher priority than UserMain

    while (1)
    {
       OSTimeDly( TICKS_PER_SECOND * 5 );
    }
}
```

12.2.4 Advanced TCP Server Using the select( ) Function

The following example will use the select() function to process up to 10 simultaneous TCP connections.

```
/*-------------------------------------------------------------------
 Description: NetBurner TCP/IP Server Multiple Socket Example Program
 Filename: main.cpp

 This example creates a TCP server that listens on the specified
 TCP port number and can handle multiple TCP connections simultaneously
 (10 in this example).

 An easy way to test the example is to use multiple Telnet sessions
 to create simultaneous connections to the NetBurner device. Status
 messages are sent out stdio to the debug serial port, and to the
 client TCP connections.
   -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <taskmon.h>
#include <tcp.h>
#include <networkdebug.h>

extern "C"
{
   void UserMain( void* pd );
}

const char* AppName = "TCP Multiple Sockets Example";

#define PORT (1000) // TCP port number to listen on
#define NFDS (10)   // Max number of file descriptors/connections

int fd_array[NFDS]; // Array of TCP file descriptors

void UserMain( void* pd )
{
   InitializeStack();

   if ( EthernetIP == 0 )
   {
      iprintf( "Trying DHCP\r\n" );
      GetDHCPAddress();
      iprintf( "DHCP assigned the IP address of :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }

   OSChangePrio( MAIN_PRIO );
   EnableAutoUpdate();
   StartHTTP();
   EnableSmartTraps();
   EnableTaskMonitor();

   #ifdef _DEBUG
   //InitializeNetworkGDB();
   InitializeNetworkGDB_and_Wait();
   #endif
```

```
/*
  Listen for incoming TCP connections. You only need to call
  listen() one time.
  - Any IP address
  - Port number = PORT
  - Queue up to 5 incoming connection requests
 */
int fdl = listen( 0, PORT, 5 );
iprintf( "Listening for incoming connections on port %d\r\n", PORT );

while ( 1 )
{
   // Declare file descriptor sets for select()
   fd_set read_fds;
   fd_set error_fds;

   // Init the fd sets
   FD_ZERO( &read_fds );
   FD_ZERO( &error_fds );

   // Configure the fd sets so select() knows what to process. In
   // this case any fd data to be read, or an error.
   for ( int i = 0; i < NFDS; i++ )
   {
      if ( fd_array[i] )
      {
         FD_SET( fd_array[i], &read_fds );
         FD_SET( fd_array[i], &error_fds );
      }
   }

   // select() should also process the listen fd
   FD_SET( fdl, &read_fds );
   FD_SET( fdl, &error_fds );

   /*
     select() will block until any fd has data to be read, or
     has an error. When select() returns, read_fds and/or
     error_fds variables will have been modified to reflect
     the events.
    */
   select( FD_SETSIZE, &read_fds, ( fd_set * )0, &error_fds, 0 );

   // If the listen fd has a connection request, accept it.
   if ( FD_ISSET( fdl, &read_fds ) )
   {
      IPADDR client_ip;
      WORD   client_port;

      int fda = accept( fdl, &client_ip, &client_port, 0 );

      // If accept() succeeded, find an open fd array slot
      if ( fda > 0 )
      {
         for ( int i = 0; i < NFDS; i++ )
         {
            if ( fd_array[i] == 0 )
            {
               fd_array[i] = fda;
               writestring( fda, "Welcome! 'Q' to quit." );
               iprintf( "Added connection on fd[%d] = %d, ", i, fda );
               ShowIP(client_ip);
```

```
            iprintf(":%d\n", client_port);
            fda = 0;
            break;
         }
      }
   }

   // If no array positions are open, close the connection
   if ( fda )
   {
      writestring( fda, "Server Full" );
    iprintf("Server Full\r\n");
      close( fda );
   }
}

// If the listen fd has an error, close it and reopen
if ( FD_ISSET( fdl, &error_fds ) )
{
   close( fdl );
   fdl = listen( 0, PORT, 5 );
}

// Process each fd array element and check it against read_fds
// and error_fds.
for ( int i = 0; i < NFDS; i++ )
{
   if ( fd_array[i] )
   {
      // Check for data to be read
      if ( FD_ISSET( fd_array[i], &read_fds ) )
      {
         char buffer[21];
         int rv = read( fd_array[i], buffer, 20 );
         if ( rv > 0 )
         {
            buffer[rv] = 0;
            if ( buffer[0] == 'Q' )
            {
               iprintf( "Closing connection fd[%d]\r\n", i );
               writestring( fd_array[i], "Bye\r\n" );
               close( fd_array[i] );
               fd_array[i] = 0;
            }
            else
            {
               iprintf( "Read \"%s\" from fd[%d]\r\n", buffer, i );
               siprintf( buffer, "Server read %d byte(s)\r\n", rv );
               writestring( fd_array[i], buffer );
            }
         }
         else
         {
            iprintf( "ReadError on fd[%d]\r\n", fd_array[i] );
            FD_SET( fd_array[i], &error_fds );
         }
      }

      // Check for errors
      if ( FD_ISSET( fd_array[i], &error_fds ) )
      {
         iprintf( "Error on fd[%d], closing connection\r\n", i );
         close( fd_array[i] );
```

```
                fd_array[i] = 0;
            }
        }
    }
}
```

12.3 Writing a Windows TCP Server

Writing a TCP server on the Windows platform involves a few more steps. As with the NetBurner TCP Server, the windows TCP server application will create a thread that listens for incoming connections on a specific port number. The basic steps are as follows:

1. Create a socket structure variable with socket( )
2. Initialize the socket structure
3. Bind the socket to a listen address with bind( )
4. Listen for incoming connections with listen( )
5. Accept an incoming connection with accept( )

Note: The TcpServerWin.exe and source code are located in the \nburn\pctools directory.

12.3.1 Windows TCP Server Application Source Code

This example program allows you to specify the listen port. Any client connecting to your server will need to know this port number in order to make a connection.

```
1   /*********************************************************************
2   TCP/IP Server Application
3
4   DESCRIPTION
5   This Win32 console application is a TCP/IP server that will listen
6   for connections on the specified port number.  The application
7   is written for Winsock 1.1 compliance.
8
9
10  COMMAND LINE PARAMETERS
11
12  Usage: tcpsrvWin <port> <echo>
13      port = port number to monitor for incoming connections
14      echo = yes to echo request to client, no is default
15
16
17  COMPILILATION INSTRUCTIONS
18      Compile as Windows console application with MS Visual C++ 5.
19      Include WS2_32.lib in linker configuration
20
21
22  REVISION HISTORY
23  1/12/2003  Initial Release
24
25
26  *********************************************************************/
27
28  #include <winsock.h>
29  #include <stdio.h>
30  #include <stdlib.h>
31  #include <time.h>
32  #include <memory.h>
33
34  #define RX_BUFSIZE (10000)
35
36  //----- functions -----
37  void ProcessCmdLine(int argc, char **argv);
38  DWORD WINAPI ProcessClient(LPVOID lpParam);
39
40
41  //----- global vars -----
42  WORD dListenPort;
43  char szIPAddress[80];
```

```
BOOL bEcho = FALSE;
DWORD dwStartTick, dwEndTick;
DWORD dwTotalBytesRead;

char RxBuf[RX_BUFSIZE];


/*-------------------------------------------------------------------
 FUNCTION: ProcessCmdLine

 DESCRIPTION:
 Process command line arguments

 RETURNS: Nothing
 -------------------------------------------------------------------*/
void ProcessCmdLine(int argc, char **argv)
{
    if (argc < 2)
    {
        printf("Usage: TCPServer <port> <echo>\n");
        exit(1);
    }

    dListenPort = (WORD) atoi(argv[1]);

    if (argc > 2)
    {
        if (strcmp(argv[2], "echo") == 0)
            bEcho = TRUE;
    }
}



/*-------------------------------------------------------------------
 FUNCTION: ProcessClient

 DESCRIPTION:
 This function is called as a thread to handle client connections.

 PARAMETERS
 lpParam is a pointer to the client socket handle.

 RETURNS: 32-bit thread exit code
 -------------------------------------------------------------------*/
DWORD WINAPI ProcessClient(LPVOID lpParam)
{
    SOCKET s = (SOCKET)lpParam;
    int SampleCtr = 0;

    dwTotalBytesRead = 0;
    while (1)
    {
        int bytes_read = recv(s, RxBuf, RX_BUFSIZE - 1, 0);
        RxBuf[bytes_read] = '\0';
        printf("Received %d bytes: %s\n", bytes_read, RxBuf);
        dwTotalBytesRead += bytes_read;

        if ( (bytes_read > 0) && bEcho )
        {
            if (send(s, RxBuf, strlen(RxBuf), 0) == SOCKET_ERROR)
            {
                printf("Error sending echo data: %d\n", WSAGetLastError());
            }
        }
        else if (bytes_read == 0)
        {
            closesocket(s);
            exit(0);
        }
        else if (bytes_read == SOCKET_ERROR)
```

```
115             {
116                 printf("recv() error in ProcessClient(): %d\n", WSAGetLastError());
117                 break;
118             }
119         }
120         return 0;
121     }
122
123
124
125
126
127     /*-------------------------------------------------------------------
128      FUNCTION: main
129
130      DESCRIPTION:
131      Main routine to start network services and listen for connections.
132
133      RETURNS: integer exit code
134      -------------------------------------------------------------------*/
135     int main(int argc, char **argv)
136     {
137         SOCKET  sListen, sClient;
138         WSADATA WSAData;
139         struct sockaddr_in sinListen, sinClient;
140
141
142         ProcessCmdLine(argc, argv);
143
144         if (WSAStartup( MAKEWORD(1,1), &WSAData) != 0)
145         {
146             printf("Error loading Winsock\n");
147             return 1;
148         }
149
150         // Create socket to listen on
151         sListen = socket(AF_INET, SOCK_STREAM, 0);
152         if (sListen == SOCKET_ERROR)
153         {
154             printf("Error creating sListen socket: %d\n", WSAGetLastError());
155             return 1;
156         }
157
158         // Setup listen address structure
159         sinListen.sin_addr.s_addr = htonl(INADDR_ANY);
160         sinListen.sin_family = AF_INET;
161         sinListen.sin_port = htons(dListenPort);
162
163         // Bind socket to listen address
164         if (bind(sListen, (struct sockaddr *) &sinListen, sizeof(sinListen)) == SOCKET_ERROR)
165         {
166             printf("bind() error for sListen: %d\n", WSAGetLastError());
167             return 1;
168         }
169
170         // Listen to interface and port
171         if (listen(sListen, 5) == SOCKET_ERROR)
172         {
173             printf("listen() error for sListen: %d\n", WSAGetLastError());
174             return 1;
175         }
176
177         printf("Listening on port: %d....\n", dListenPort);
178         while (1)
179         {
180             int len = sizeof(sinClient);
181             sClient = accept(sListen, (struct sockaddr *) &sinClient, &len);
182             if (sClient == INVALID_SOCKET)
183             {
184                 printf("accept() error: %d\n", WSAGetLastError());
185                 break;
```

```
186         }
187
188         printf("Accepted connection from: %s:%d\n",
189             inet_ntoa(sinClient.sin_addr), ntohs(sinClient.sin_port));
190
191
192
193         /*
194         HANDLE CreateThread(
195                 LPSECURITY_ATTRIBUTES lpThreadAttributes, // pointer to thread security
196 attributes
197                 DWORD dwStackSize,                  // initial thread stack size, in
198 bytes
199                 LPTHREAD_START_ROUTINE lpStartAddress,  // pointer to thread function
200                 LPVOID lpParameter,                      // argument for new thread
201                 DWORD dwCreationFlags,              // creation flags
202                 LPDWORD lpThreadId                  // pointer to returned thread
203 identifier
204                 );
205         */
206         HANDLE hThread;
207         DWORD  dwThreadId;
208         hThread = CreateThread(NULL, 0, ProcessClient, (LPVOID)sClient, 0, &dwThreadId);
209         if (hThread == NULL)
210         {
211             printf("Error creating Client thread: %d\n", GetLastError());
212             break;
213         }
214
215     }
216
217     closesocket(sListen);
218     WSACleanup();
219     return 0;
220 }
```

Line 59: The void ProcessCmdLine(int argc, char **argv) function is called at the beginning of main( ) to do what you would expect: process any command line parameters. Specifically, you will pass the listening port number. Whether or not to echo the data sent from the client is optional.

Line 89: The DWORD WINAPI ProcessClient(LPVOID lpParam) function is run as a separate thread spawned for the incoming connection. The recv( ) function is used to obtain the data and number of characters. The data and amount is then displayed to the console window with a printf( ).

If data has been read and the echo option enabled, the data is sent back to the client. If the number of bytes read is zero, or an error has occurred, the socket is closed.

Line 137: Declaration of the listen and client sockets. The client socket will be passed to the thread when it is created.

Line 138: Before you can call any winsock functions, you must load a winsock library. The call to initialize winsock is WSAStartup( ); when this function returns, the WSADATA structure contains the information about the version of winsock that was loaded. The MAKEWORD(1,1) macro specifies winsock version 1.1; you could specify (2,2) for version 2.2.

Line 139: Winsock applications specify IP addresses and ports using the SOCKADDR_IN structure. This line declares structures for the listen and client sockets.

Line 151: Create the listen socket. AF_INET specifies the Internet Protocol. The socket type is set to SOCK_STREAM; for UDP you would use SOCK_DGRAM. The last parameter specifies protocol, and a value of 0 tells the system to pick one based on the other two parameters.

Lines 159-161: Assign values to socket address structure. The functions htonl( ) and htons( ) stand for "host to network long" and "host to network short", and convert the byte order of long and short variables from the host platform (little endian for a typical pc running Windows) to standard network byte order (big endian). The INADDR_ANY parameter specifies that any incoming IP address will be accepted, the AF_INET specifies the Internet Protocol family, and the dListenPort parameter contains the value of the port the server will listen to.

Line 164: The bind( ) function associates the listen socket with the listen port number.

Line 171: The listen( ) function puts the socket into "listen mode" so it will accept incoming connections. The second parameter, 5, is the number of connections that can be queued (to handle simultaneous incoming connection requests).

Line 181: The accept( ) call accepts an incoming connection. Any pending connections will be serviced first. The second parameter is a pointer to an address structure that will contain the connection information such as IP address and port number. In this case, we pass a pointer to sinClient. The third parameter is a pointer to a variable that will contain the length of the address structure. The accept( ) function returns a socket descriptor that corresponds to the new client connection, which can then be used for communication with the client.

Line 208: This example creates a thread to handle an incoming client connection. The CreateThread( ) API call takes the following parameters:

- ProcessClient: A pointer to the thread function to execute
- (LPVOID)sClient: When creating a thread it is possible to pass a parameter to the function as a pointer. In this case, we are passing the client socket, sClient, as pointer. It is later cast back to a socket in line 91.
- &dwThreadId: A pointer to a variable that will hold the thread ID once it is created.

When CreateThread( ) is called successfully, the thread will start and the client connection will be processed. When the client connection is terminated, the thread will return and be deleted. The program will then return to the top of the while loop and block at the accept( ) call.

12.3.2 Running the Windows TCP Server Application

The application was built as a Win32 console application. To run it, open a command prompt and type "`tcpsrvWin <port>`", where <port> is the listening port number (e.g. 2000). The port number is a 16-bit value, and numbers less than 1024 are reserved.

12.4 Writing a NetBurner TCP Client

While a TCP Server waits for incoming connections, a TCP Client handles the opposite end – it initiates a connection. In order to initiate a connection, the TCP Client must know the destination IP address and port number of the server to which is wishes to make a connection. Writing a TCP Client application on the NetBurner platform only requires one function call to connect( ):

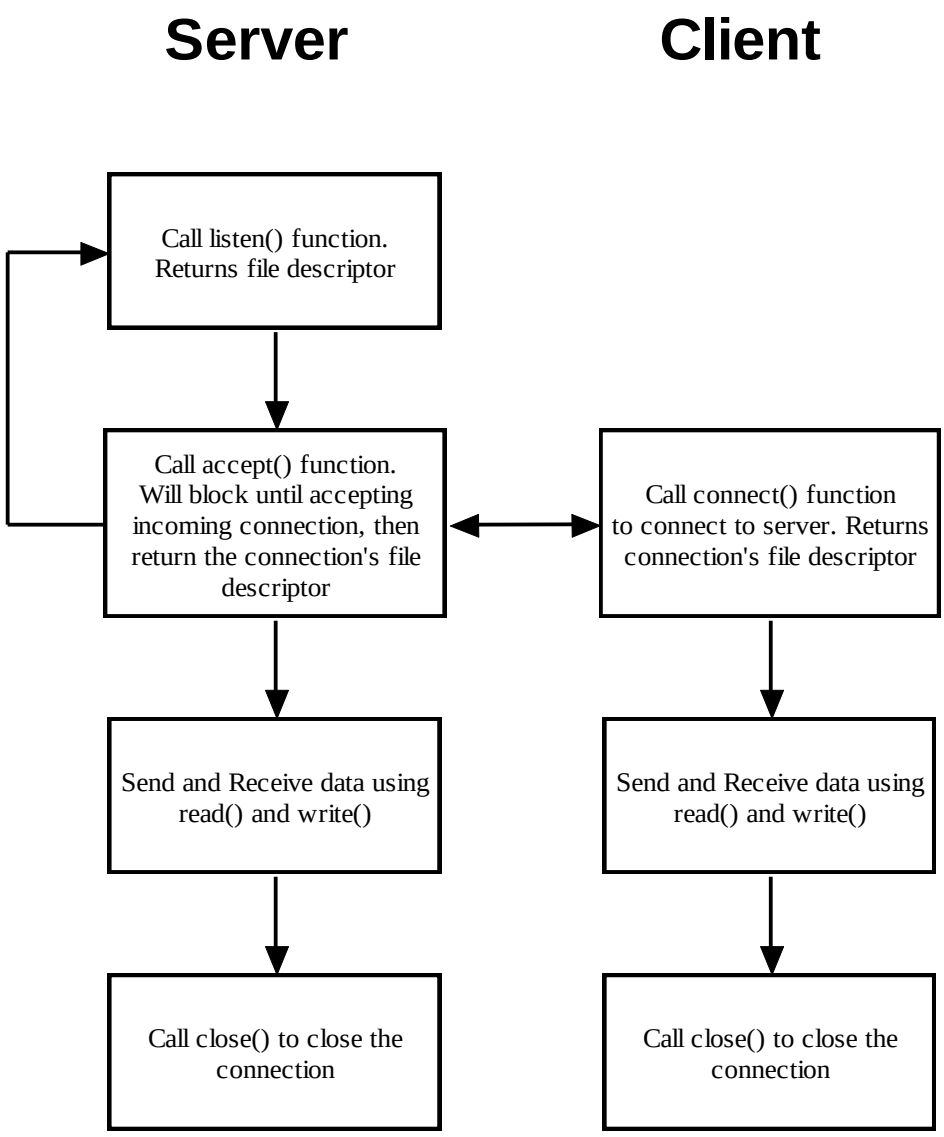
```
int connect(IPADDR dest_ipaddr, WORD localport, WORD destport, DWORD timeout);
```

where:

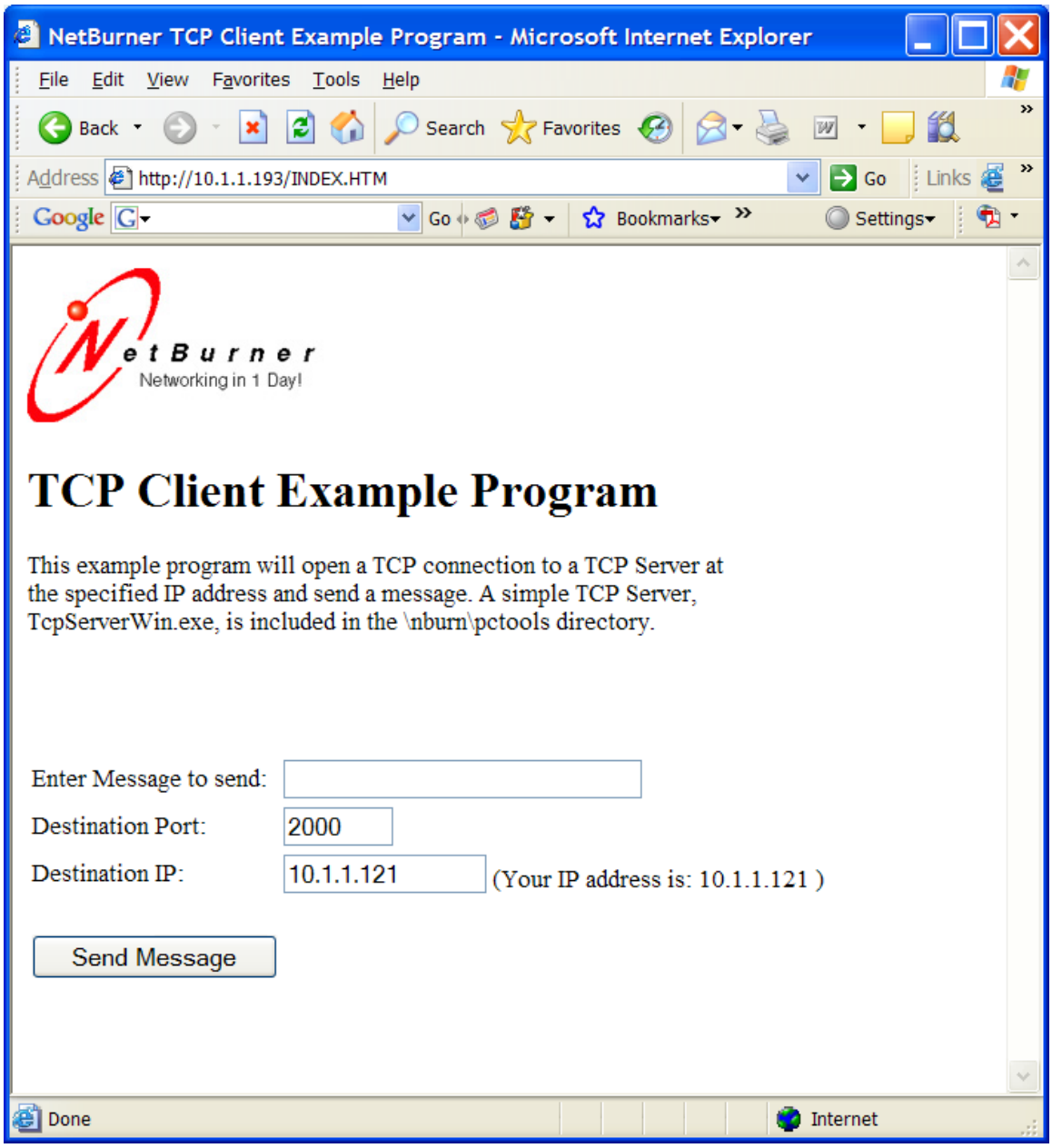
- dest_ipaddr is the destination IP address (the TCP server in this example)
- localport is the local port number of the TCP Client. Remember, 4 items are needed for a connection, and the TCP Client needs to tell the Server what port number it is using.
- destport is the destination port number – the one the server is listening on.
- timeout allows the connect( ) call to exit if a connection is not made within a specified amount of time. A timeout value of 0 will wait forever.

This example program will combine a number of features covered to this point:

- The application will create a web page interface with input fields for a message to send to the server, the server IP address and the server port number.
- HTML forms will be used to submit user data
- The web page interface will invoke the connect( ) function to create an outgoing connection to a TCP Server and send the message.



A screen shot of the web interface is shown below:



12.4.1 NetBurner TCP Client Application Source Code

The TCP Client example code is organized in the following code modules:

File	Description
main.cpp	Application initialization, DHCP and check of flash memory stored parameters.
clientweb.cpp	Handles the dynamic content for the web server interface.
index.htm	HTML code for the web server interface.

Main.cpp

Since all of the action occurs through the web server interface, main.cpp is very simple. It initializes the system, and does a check for DHCP.

```
/*-------------------------------------------------------------------
 This program demonstrates how to create a TCP Client. All interaction
 is through the web page interface, which allows a user to type
 in a message and send it to a TCP Server.
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <startnet.h>
#include <autoupdate.h>
#include <startnet.h>
#include <dhcpclient.h>
#include "clientweb.h"

//----- Function Prototypes -----
extern "C"
{
   void UserMain(void * pd);
}


/*-------------------------------------------------------------------
 User Main
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
   InitializeStack();      // Initialize the TCP/IP Stack

    if ( EthernetIP == 0 )
    {
       iprintf( "Trying DHCP\r\n" );
       GetDHCPAddress();
       iprintf( "DHCP assigned the IP address of :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }
    else
    {
       iprintf( "Static IP address set to :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }

   StartHTTP();         // Start the Web Server
   EnableAutoUpdate();     // Enable network code updates
   OSChangePrio(MAIN_PRIO);   // Set this task priority
   RegisterPost();         // Register HTML Post function

   while (1)
   {
       OSTimeDly( TICKS_PER_SECOND );
   }
}
```

ClientWeb.h

This is the header file for clientweb.cpp, and contains the declarations for functions called by main.cpp.

```
#ifndef _NB_CLIENTWEB_H
#define _NB_CLIENTWEB_H


//----- Function Prototypes -----
void RegisterPost();

#endif
```

ClientWeb.cpp

This code module handles the dynamic content and web server interface. When the web page is displayed, current values for the port numbers and IP addresses must be displayed. The user must be able to change these values, and also be able to send a message with the submit button.

```
/*-------------------------------------------------------------------
 * This code implements the web page entries for the message,
 * desination IP address and destination port number. When the
 * web page first loads it will automatically fill in the IP
 * address from the source requesting the web page, because in
 * most cases it will also be the address of the TCP Server.
 * The web page is a form, and when a user presses the submit
 * button the SendMsg() function will open a TCP connection to
 * the server, send the message, and close the connection.
 * Any error messages will be sent to stdout and can be viewed
 * with MTTTY.
 *
 * A TCP server program must already be listening at the specified
 * IP address and port number for the message to be sent. A simple
 * TCP Server called TcpServerWin.exe is located in the
 * \nburn\pctools directory of your NetBurner installation.
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <startnet.h>
#include <iosys.h>
#include <utils.h>
#include <ip.h>
#include <tcp.h>
#include <string.h>
#include "clientweb.h"

#define APP_VERSION "Version 1.1 4/23/2008"
#define POST_BUFSIZE (4096)          // form post buffer size

//----- function prototypes -----
extern "C"
{
    void WebDestPort(int sock, PCSTR url);
    void WebDestIp(int sock, PCSTR url);
   void WebShowClientIp(int sock, PCSTR url);
}
```

```
int    gDestPort;
IPADDR gDestIp;


/*-------------------------------------------------------------------
 Convert IP address to a string
 -------------------------------------------------------------------*/
void IPtoString(IPADDR  ia, char *s)
{
    PBYTE ipb= (PBYTE)&ia;
    siprintf(s, "%d.%d.%d.%d",(int)ipb[0],(int)ipb[1],(int)ipb[2],(int)ipb[3]);
}



/*-------------------------------------------------------------------
 Sends a message to the specified host.
 -------------------------------------------------------------------*/
void SendMsg( IPADDR DestIp, int DestPort, char *msg )
{
    iprintf( "Connecting to: " );
    ShowIP( DestIp );
    iprintf( " : %d\r\n", DestPort );
    iprintf( "Attempting to write \"%s\" \r\n", msg );
   int fd = connect( DestIp, 0, DestPort, TICKS_PER_SECOND * 5);
   if (fd < 0)
   {
      iprintf("Error: Connection failed\r\n");
   }
   else
   {
      int n = write( fd, msg, strlen(msg) );
      iprintf( "Wrote %d bytes\r\n", n );
       close( fd );
   }
}


/*-------------------------------------------------------------------
 Show destination port number on web page
 -------------------------------------------------------------------*/
void WebDestPort(int sock, PCSTR url)
{
   char buf[80];

   // If no dest port is specified, use a default
   if ( gDestPort == 0 )
      gDestPort = 2000;

   siprintf( buf, "VALUE=\"%d\" ", gDestPort );
   writestring( sock, buf );
}

/*-------------------------------------------------------------------
 Show destination ip address on web page
 -------------------------------------------------------------------*/
void WebDestIp(int sock, PCSTR url)
{
   char buf[80];
   char szDestIp[40];

   // If no dest ip address has been entered, use the one that
```

```
   // requested the web page.
   if ( gDestIp == 0 )
      IPtoString( GetSocketRemoteAddr( sock ), szDestIp );
   else
      IPtoString( gDestIp, szDestIp );

   siprintf( buf, "VALUE=\"%s\" ", szDestIp );
   writestring( sock, buf );
}


/*-------------------------------------------------------------------
 Show destination ip address on web page
 -------------------------------------------------------------------*/
void WebShowClientIp(int sock, PCSTR url)
{
    char buf[80];

     IPtoString( GetSocketRemoteAddr( sock ), buf );
    writestring( sock, buf );
}

/*-------------------------------------------------------------------
 Handle HTTP Post

 WARNING The User data space is 8K in size.  The object we are storing is
 less than 200 bytes long, thus it is probably ok to make an automatic
 variable out of it.  It would NOT BE OK to make an 8K automatic
 variable.  The coices for doing this with an 8K object...

 First choice: make a global variable, this way the linker will
 allocate space for it. All errors will be at link time not run time.

 Second choice: Increase the HTTP stack size in constants.h and
 recompile the whole system directory.

 Third choice: use malloc and free.  The only problem is what do
 you do if malloc fails?
 -------------------------------------------------------------------*/
int MyDoPost(int sock, char * url, char * pData, char * rxBuffer)
{
   int max_chars = 40;
   // This buffer is 4096 bytes, so make it a static so that it does
   // not take up task stack space, but instead uses global space.
   static char buf[POST_BUFSIZE];


   iprintf( "Post Data: %s\r\n", pData );  // print all data sent from app

   // The SendMessage button on the web page initiates a form POST
   // to send the message to the TCP server.
   if ( ExtractPostData( "SendMessage", pData, buf, max_chars) > 0 )
   {
      iprintf( "Processing SendMessage Post\r\n" );

      if (ExtractPostData("tfDestPort", pData, buf, max_chars) == -1)
         iprintf("Error reading post data: tfDestPort\r\n");
      else
         gDestPort = (int)atoi(buf);

      if (ExtractPostData("tfDestIpAddr", pData, buf, max_chars) == -1)
         iprintf( "Error reading post data: tfDestIpAddr\r\n");
      else
```

```
        gDestIp = AsciiToIp( buf );

      if ( ExtractPostData( "tfMessage", pData, buf, max_chars ) < 0 )
         iprintf("Error reading post data: tfDestPort\r\n");
      else
         SendMsg( gDestIp, gDestPort, buf );
   }


    // We have to respond to the post with a new HTML page. In
    // this case we will redirect so the browser will go to
    // that URL for the response.
   RedirectResponse( sock, "index.htm" );

   return 0;
}


/*-------------------------------------------------------------------
 Specify the function to call when a user clicks Submit on the
 web page form.
 -------------------------------------------------------------------*/
void RegisterPost()
{
   SetNewPostHandler( MyDoPost );
}
```

Index.htm

```
<HTML>
<HEAD>
<TITLE>NetBurner TCP Client Example Program</TITLE>
</HEAD>
<BODY BGCOLOR="#ffffff">

<img src="logo.jpg">
<br>
<H1>TCP Client Example Program</H1>
This example program will open a TCP connection to a TCP Server at <br>
the specified IP address and send a message. A simple TCP Server, <br>
TcpServerWin.exe, is included in the \nburn\pctools directory.

<FORM ACTION="bogus.htm" METHOD=POST>
<br>

<TABLE>
  <TR>
    <TD>Enter Message to send: </TD>
    <TD><INPUT NAME="tfMessage" TYPE="text" SIZE="30"></TD>
  </TR>

  <TR>
    <TD>Destination Port: </TD>
    <TD><INPUT NAME="tfDestPortNum" <!--FUNCTIONCALL WebDestPort -->
TYPE="text" SIZE="6"></TD>
  </TR>

  <TR>
    <TD>Destination IP: </TD>
    <TD><INPUT NAME="tfDestIpAddr" <!--FUNCTIONCALL WebDestIp --> TYPE="text"
SIZE="15">
```

```
    (Your IP address is: <!--FUNCTIONCALL WebShowClientIp --> )
    </TD>
  </TR>

  <TR>
    <TD> <br></TD>
  </TR>


  <TR>
    <TD><INPUT NAME="SendMessage" TYPE="submit" VALUE="Send Message"></TD>
  </TR>
  <BR><BR>

</TABLE>

</FORM>
</BODY>
</HTML>
```

12.4.2 NetBurner TCP Client Application Operation

The application boots and executes the initialization functions in main.cpp. From that point forward any information sent will occur through the web server interface. When a user connects to the device using a web server, the information retrieved is defined by the index.htm file.

Examining index.htm, we can see that the WebDestPort() and WebDestIp() functions are used to supply the current information into the web page text fields. Looking at the functions - WebDestPort( ) and WebDestIp( ) in clientweb.cpp, we can see the current settings are stored in two global variables:

```
int    gDestPort;
IPADDR gDestIp;
```

The function WebShowClientIp() extracts the IP address from the TCP connector of the web page request, and fills it in as a default value for the destination IP address to send the message.

The message to send is entered as a text field. The message is not stored in the NetBurner device, it is just a temporary string displayed in the web browser. To send a message a user clicks on the Send Message submit button. This is a special HTML command that sends a POST message to the web server containing the message to send. This is where the MyDoPost( ) function in clientweb.cpp comes into play. It intercepts the post and takes the appropriate action. In the case of sending a message, it extracts the message data and calls the SendMsg( ) function. SendMsg( ) in clientweb.cpp makes a connect( ) call using the passed message and stored values of destination port number and IP address. Once the connection is made, SendMsg( ) calls write( ) to send the message followed by close( ) to terminate the connection.

13 UDP - User Datagram Protocol

UDP is a datagram-oriented protocol that does not guarantee delivery. TCP is a stream-oriented protocol with guaranteed delivery. With TCP, you do not have control over what data will be sent with each IP datagram, but with UDP datagram will produce one IP datagram. The advantage with UDP is that each datagram can contain one standard piece of data, such as a measurement or command. With TCP, you would have to implement a protocol to identify the beginning, end and type of each piece of data and parse the data stream. UDP is connectionless: the devices do not establish a connection or establish a data stream. Data is sent as individual datagrams. The NetBurner API provides for two mechanisms to implement UDP: sockets and a C++ class. The choice of which to use is depenant of which method you find most comfortable using. There is not a performance difference.

13.1 UDP Client/Server Application Using the UDP Class

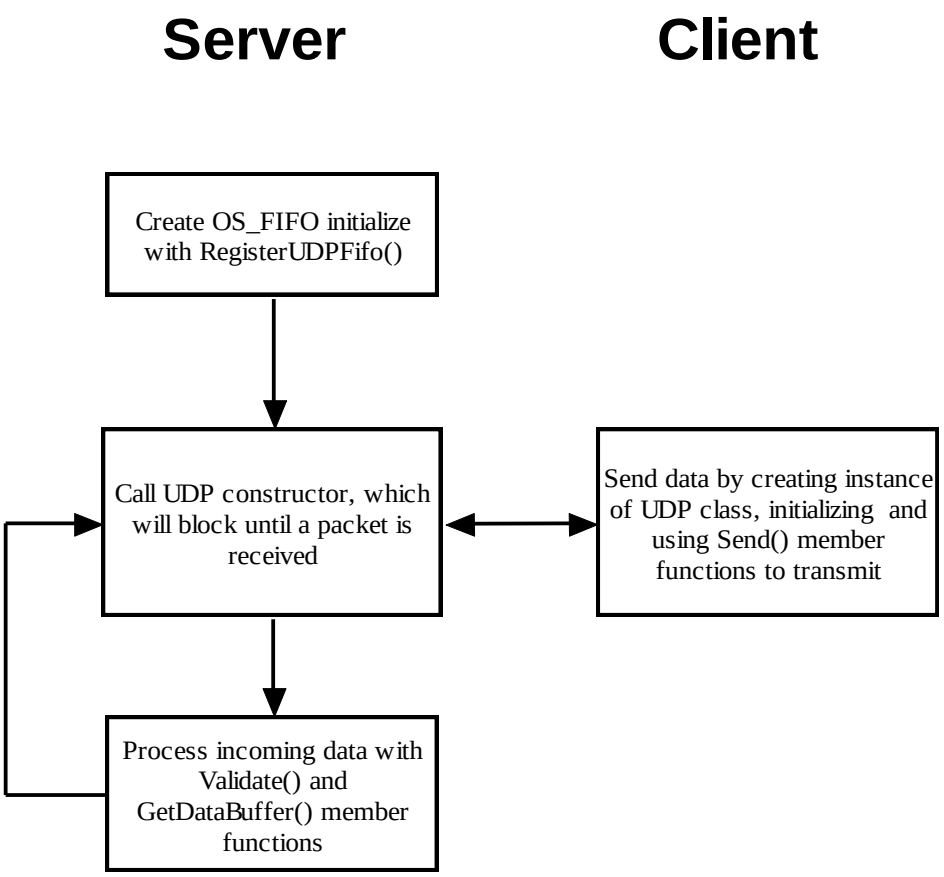
The NetBurner UDP API is a C++ Class. To create a **UDP Server** on the NetBurner platform we need to:

1. Create an OS_FIFO to hold incoming packets
2. Register the UDP FIFO with RegisterUDPFifo(port, &fifo) so it listens to a specific port number. This is where incoming UDP packets will be stored. If you are listening on multiple UDP ports, you can use one FIFO to store them all by calling RegisterUDPFifo( ) for each port number. You can also have multiple OS_FIFOs for each port if you wish.
3. Call a UDP constructor such as: UDPPacket upkt(&fifo, timeout), which will block until a packet is received, then accept and store the packet.

To send a UDP datagram on the NetBurner platform we need to:

1. Declare an instance of a UDP packet
2. Specify the source and destination port numbers, and add data to the packet using the class member functions.
3. Send to a specified IP address using the Send( ) member function.

Looking at the above description and the flow chart below, you will notice an important difference from TCP communication; since there is not a connection between the client and sever, you cannot simply send and receive data using read() and write() functions. You will need to create a UDP packet for each transmission, specifying the port numbers and destination IP address.



13.1.1 NetBurner UDP Class Application Source Code

```
/*****************************************************************************
 NetBurner simple UDP send/receive example.

 This example program will allow you to send and receive UDP packets to
 another device or host computer. To run the example, connect a serial
 port to the debug serial port on your NetBurner device and run a dumb
 terminal program such as MTTTY. On the PC, run the NetBurner UDP Terminal
 (be sure to set the IP address and port numbers to match). You will then
 be able to type characters in the UDP Termainal and see them in MTTTY,
 and vice versa.

 You will be prompted for the port number to send/receive data and the
 destination IP address of the other device or host. Note that the application
 uses the same port number to send and receive data, but you can change this
 to any other port number you wish.

 The application will create a thread to recive packets and display them
 on the debug port, while the main task will take any data you type in
 to the MTTTY dumb terminal and send it as a udp packet to the destination
 IP address.

 *****************************************************************************/
```

```
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <ucos.h>
#include <udp.h>
#include <autoupdate.h>
#include <dhcpclient.h>

extern "C" {
    void UserMain(void * pd);
}


const char *AppName = "UDP Send/Receive Example";


// Allocate task stack for UDP listen task
DWORD   UdpTestStk[USER_TASK_STK_SIZE];



/*-------------------------------------------------------------------
 UDP Server task will wait for incoming packets on the
 designated port number, which is passed as a OSTaskCreate()
 void * parameter.
 -------------------------------------------------------------------*/
void UdpReaderMain(void * pd)
{
    int port = (int)pd;   // cast void * param as int port number
    iprintf("Listening on UDP port: %d\r\n", port);

    // Create a FIFO for the UDP packet and initialize it
    OS_FIFO fifo;
    OSFifoInit(&fifo);

    // Register to listen for UDP packets on port number 'port'
    RegisterUDPFifo(port, &fifo);

    while (1)
    {
        // Construct a UDP packet object using the previously
        // declared FIFO. The UDP constructor will only return
        // when a packet has been received. The second parameter
        // is a timeout value (timeticks). A value of 0 will
        // wait forever. The TICKS_PER_SECOND definition can
        // be used for code readability.
        UDPPacket upkt(&fifo, 0 * TICKS_PER_SECOND);

        // Did we get a valid packet, or just time out?
        if (upkt.Validate())
        {
            WORD len = upkt.GetDataSize();
            iprintf("Received UDP packet with %d bytes from: ", (int)len);
            ShowIP(upkt.GetSourceAddress()); // show ip address
            iprintf("\r\n");

            ShowData(upkt.GetDataBuffer(), len); // hex dump function
            iprintf("\r\n");
        }
    }
}
```

```
/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
   int    portnum;
   IPADDR    ipaddr;
   char   buffer[80];

   InitializeStack();
    if ( EthernetIP == 0 )
    {
       iprintf( "Trying DHCP\r\n" );
       GetDHCPAddress();
       iprintf( "DHCP assigned the IP address of :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }
    else
    {
       iprintf( "Static IP address set to :" );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }

   EnableAutoUpdate();
   OSChangePrio(MAIN_PRIO);

   iprintf("Starting NetBurner UDP Example \r\n");
   iprintf("Enter the UDP port number (will be used for send & receive): ");
   iscanf("%d", &portnum);

   iprintf("\r\nEnter the destination IP Address: ");
   buffer[0] = 0;
   while(buffer[0] == 0)
   {
      gets(buffer);
   }
   ipaddr = AsciiToIp(buffer);

   iprintf("Listening/Sending on UDP Port %d, Sending to IP address: ",
portnum);
   ShowIP(ipaddr);
   iprintf("\r\n");

   // Create UDP listen task
   OSTaskCreate(UdpReaderMain,
             (void  *)portnum,
             &UdpTestStk[USER_TASK_STK_SIZE] ,
             UdpTestStk,
             MAIN_PRIO - 1);  // higher priority than send task


   // Main while loop will take any user input and send it as a
   // UDP datagram.
   while(1)
   {
      gets(buffer);
      iprintf("Sent \"%s\" to ", buffer);
      ShowIP(ipaddr);
      iprintf(":%d\r\n", portnum);
```

```
        {
            UDPPacket pkt;
            pkt.SetSourcePort(portnum);
            pkt.SetDestinationPort(portnum);
            pkt.AddData(buffer);
            pkt.AddDataByte(0);
            pkt.Send(ipaddr);
        }
        iprintf("\r\n");
    }
}
```

13.1.2 Running the Application

1. (If you have not already done so) Connect a serial cable from your host computer's serial port to your NetBurner board's debug serial port.
2. Start MTTTY (from Windows: Start →Programs → Netburner NNDK → Mttty Serial Terminal)
3. Compile and download the application to your NetBurner board.
4. Verify that you see the debug messages for the UDP Client/Server. Note the IP address.
5. Answer the prompts asking for the listening port number and destination IP address and port number of the host PC you will communicate with.
6. Run the NetBurner UDP Terminal Tool, available in the NetBurner NNDK program group in the windows start menu. Enter the values for the IP address, local and remote ports (ports will be the same). Alternatively, you can run the windows UDP application described in the section.
7. If everything is working properly, anything you type in the MTTTY window will be sent as UDP datagrams to the host PC, and vice versa.

13.2 UDP Client/Server Application Using UDP Sockets

To listen for incoming UDP packets, use the function:`int UdpFd = CreateRxUdpSocket( port )`, then use `recvFrom()` to block and wait for incoming packets. Note that the standard Berkley Sockets implementation will block forever on the receive function. If you want to allow the task to have control, you can wrap the `recvFrom()` function inside a `select()` function using the UDP file descriptor and a timeout.

To send outgoing UDP packets, use the `sendto()` function.

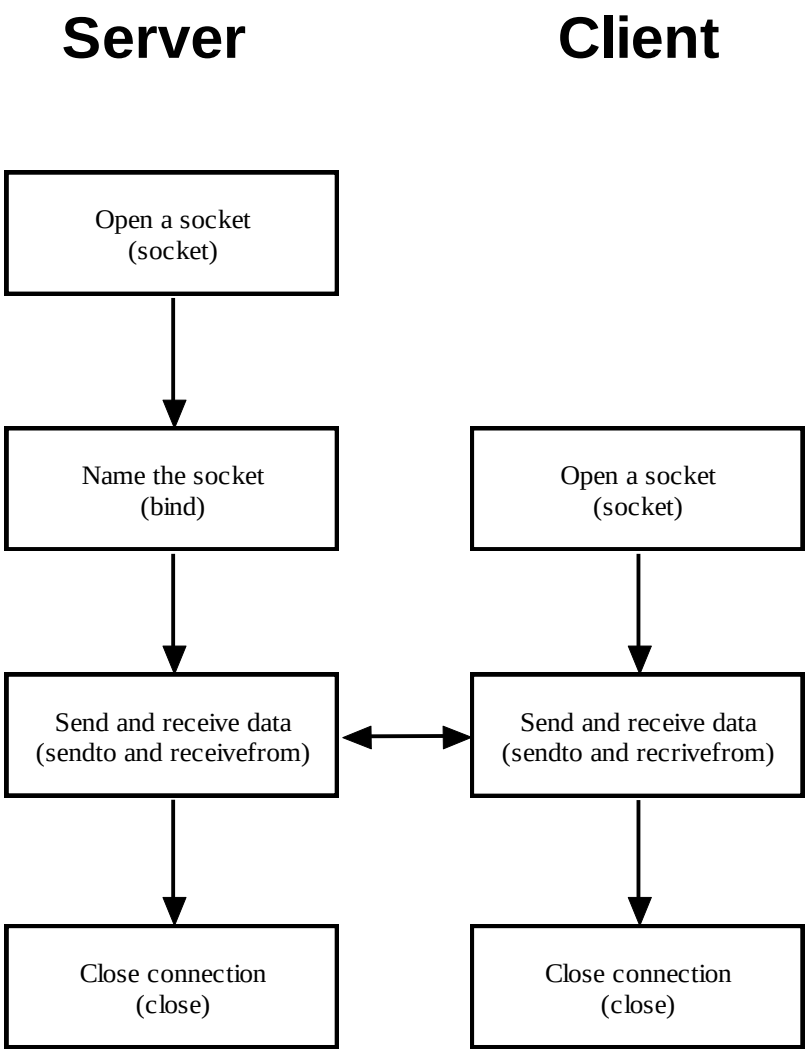
```
/*****************************************************************************
 UDP Sockets Example
 This application will send/receive UDP packets with another host on a network,
 such as a PC. Use the MTTTY serial port program to access the menu and
 prompts to specify the destination IP address and port number.

 NetBurner supplies an API for handling UDP as a C++ Class using UDPPacket, or
 you can use a UDP socets API (see UDP socet example).

 For an external UDP host you can use the NetBurner java example, or the
 NetBurner UDP terminal program.
*****************************************************************************/
#include "predef.h"
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <startnet.h>
#include <ucos.h>
#include <udp.h>
#include <autoupdate.h>
#include <string.h>
#include <taskmon.h>
#include <dhcpclient.h>
#include <networkdebug.h>

const char *AppName = "UDP Sockets Example";

extern "C"
{
   void UserMain( void *pd );
}


// Declare a task stack for the UDP Reader task
DWORD UdpReaderStack[USER_TASK_STK_SIZE];


/*-------------------------------------------------------------------
 * This task will wait for incoming UDP packets and process them.
 -------------------------------------------------------------------*/
void UdpReaderTask( void *pd )
{
   int port = ( int ) pd;
   iprintf( "UdpReaderTask monitoring port %d\r\n", port );

   // Create a UDP socket for receiving
   int UdpFd = CreateRxUdpSocket( port );
   if ( UdpFd <= 0 )
   {
```

```
      iprintf("Error Creating UDP Listen Socket: %d\r\n", UdpFd);
      while (1)
         OSTimeDly(TICKS_PER_SECOND);
   }
   else
   {
      iprintf( "Listening for UDP packets on port %d\r\n", port );
   }

   while (1)
   {
      IPADDR SrcIpAddr;  // UDP packet source IP address
      WORD   LocalPort;  // Port number UDP packet was sent to
      WORD   SrcPort;    // UDP packet source port number
      char   buffer[80];

      int len = recvfrom( UdpFd, (BYTE *)buffer, 80, &SrcIpAddr, &LocalPort,
                          &SrcPort );
      buffer[len] = '\0';

      iprintf( "\r\nReceived a UDP packet with %d bytes from :", len );
      ShowIP( SrcIpAddr );
      iprintf( "\r\n%s\r\n", buffer );
   }
}




/*-------------------------------------------------------------------------
 * UserMain Task
 * This is the first task to be executed and will create the UDP
 * Reader Task.
 -------------------------------------------------------------------------*/
void UserMain( void *pd )
{
   int       portnum;
   IPADDR    ipaddr;
   char   buffer[80];

   InitializeStack();
   EnableAutoUpdate();
   EnableTaskMonitor();

   if ( EthernetIP == 0 )
   {
      iprintf( "Trying DHCP\r\n" );
      GetDHCPAddress();
      iprintf( "DHCP assigned the IP address of :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }

   OSChangePrio( MAIN_PRIO );

   #ifdef _DEBUG
   InitializeNetworkGDB();
   #endif

   iprintf( "Starting UDP Sockets Example\r\n" );

   // Get desination IP address
```

```
   iprintf( "Enter the UDP Server destination IP address: " );
   buffer[0] = '\0';
   while ( buffer[0] == '\0' )  // Keep looping until something is entered
   {
      gets( buffer );
   }
   ipaddr = AsciiToIp( buffer );
   iprintf("\r\n");

   // Get the port number. This application uses the same
   // port number for send and receive.
   iprintf( "Enter the source/destination port number: " );
   gets( buffer );
   portnum = atoi( buffer );
   iprintf("\r\n");

   // Create a UDP socket for sending/receiving
   int UdpFd = CreateTxUdpSocket( ipaddr, portnum, portnum );
   if ( UdpFd <= 0 )
   {
      iprintf("Error Creating UDP Socket: %d\r\n", UdpFd);
      while (1)
         OSTimeDly(TICKS_PER_SECOND);
   }
   else
   {
      iprintf( "Sending/Recieving with host ");
      ShowIP( ipaddr );
      iprintf( ": %d\r\n", portnum );
   }

   // Create task to recieve UDP packets. We will pass the destination
   // port number in the optional second parameter field, and set the
   // priority to 1 less than the UserMain priority so packets get
   // processed as they are received.
   OSTaskCreate( UdpReaderTask,
                 ( void * ) portnum,
                 &UdpReaderStack[USER_TASK_STK_SIZE],
                 UdpReaderStack,
                 MAIN_PRIO - 1 );

   // Loop forever displaying UDP data
   while ( 1 )
   {
      iprintf( "Enter a string to send: " );
      gets(buffer);
      iprintf("\r\n");
      iprintf( "Sending \"%s\" using UDP to ", buffer );
      ShowIP( ipaddr );
      iprintf(" : %d\r\n", portnum );

      sendto( UdpFd, (BYTE *)buffer, strlen(buffer), ipaddr, portnum );
      iprintf( "\r\n" );
   }
}
```

13.3 Writing a Windows UDP Client/Server Application

While Windows does support a UDP class, it is more difficult to use than the NetBurner class. This example will use the sendto() and receivefrom() functions to send and receive UDP datagrams.



13.3.1 Windows UDP Application Source Code

```
/*********************************************************************
UDP Windows Application
File Name: UDPWin.cpp

DESCRIPTION
This Win32 console application will send and receive UDP packets


COMMAND LINE PARAMETERS

Usage: UDPWin <listen port> <dest ip> <dest port>

where
    listen port = port number to monitor for incoming datagrams
    dest ip = destination IP address
    dest port = destination port number


COMPILILATION INSTRUCTIONS
    Compile as Windows console application with MS Visual C++ 5.0
    Include WS2_32.lib in linker configuration


REVISION HISTORY
2/3/2003  Initial Release

*********************************************************************/

#include <winsock.h>
#include <stdio.h>
#include <stdlib.h>

#define RX_BUFSIZE (10000)

//----- functions -----
void ProcessCmdLine(int argc, char **argv);
DWORD WINAPI ReceiveUDP(LPVOID lpParam);


//----- global vars -----
WORD dListenPort, dDestPort;
char szDestIPAddr[80];
char RxBuf[RX_BUFSIZE];


/*-------------------------------------------------------------------
 FUNCTION: ProcessCmdLine

 DESCRIPTION:
 Process command line arguments

 RETURNS: Nothing
 -------------------------------------------------------------------*/
void ProcessCmdLine(int argc, char **argv)
{
          if (argc < 4)
          {
                    printf("Usage: UDPWin <listen port> <dest ip> <dest port>\n");
                    exit(1);
          }

          dListenPort = (WORD) atoi(argv[1]);
          strncpy(szDestIPAddr, argv[2], 80);
          dDestPort = (WORD) atoi(argv[3]);
}

```

```
67
68
69  /*-------------------------------------------------------------------
70   FUNCTION: ProcessClient
71
72   DESCRIPTION:
73   This function is called as a thread to handle client connections.
74
75   PARAMETERS
76   lpParam is a pointer to the client socket handle.
77
78   RETURNS: 32-bit thread exit code
79   -------------------------------------------------------------------*/
80  DWORD WINAPI ReceiveUDP(LPVOID lpParam)
81  {
82            SOCKET s = (SOCKET)lpParam;
83            struct sockaddr_in sinFrom;
84
85            while (1)
86            {
87                      printf("Listening on port: %d....\n", dListenPort);
88
89                      /*
90                        The recvfrom( ) API function has the following parameters:
91
92                        recvfrom(SOCKET s,         // socket to receive from
93                                char *buf,         // point to buffer to store rec'd data
94                                size_t len, // length of buffer or # of bytes to read
95                                int flags,         // flags (normally use 0)
96                                struct sockaddr *from,       // client data returned
97                                int *fromlen       // size of client data struct
98                                )
99                      */
100                     int len = sizeof(sinFrom);
101                     int bytes = recvfrom(s, RxBuf, RX_BUFSIZE - 1, 0,
102                                          (struct sockaddr *)&sinFrom, &len );
103                     if (bytes == SOCKET_ERROR)
104                     {
105                               printf("recvfrom() error: %d\n", WSAGetLastError());
106                               break;
107                     }
108
109                     printf("Received packet from: %s:%d\n",
110                               inet_ntoa(sinFrom.sin_addr), ntohs(sinFrom.sin_port));
111                     printf("Bytes read: %d: \"", bytes);
112                     for (int i=0; i < bytes; i++)
113                               printf("%c", RxBuf[i]);
114                     printf("\"\n");
115
116           }
117           return 0;
118 }
119
120
121
122
123 /*-------------------------------------------------------------------
124  FUNCTION: main
125
126  DESCRIPTION:
127  Main routine to start network services and listen for connections.
128
129  RETURNS: integer exit code
130  -------------------------------------------------------------------*/
131 int main(int argc, char **argv)
132 {
133           SOCKET    sListen, sSend;
134           WSADATA   WSAData;
135           struct sockaddr_in sinListen, sinSend;
136
137           ProcessCmdLine(argc, argv);
```

```
138
139             if (WSAStartup( MAKEWORD(1,1), &WSAData) != 0)
140             {
141                      printf("Error loading Winsock\n");
142                      return 1;
143             }
144
145             // Create socket to listen for UDP datagrams
146             sListen = socket(AF_INET, SOCK_DGRAM, 0);
147             if (sListen == SOCKET_ERROR)
148             {
149                      printf("Error creating sListen socket: %d\n", WSAGetLastError());
150                      return 1;
151             }
152
153             // Setup listen address structure
154             sinListen.sin_addr.s_addr = htonl(INADDR_ANY);
155             sinListen.sin_family = AF_INET;
156             sinListen.sin_port = htons(dListenPort);
157
158             // Bind socket to listen address
159             if (bind(sListen, (struct sockaddr *) &sinListen,
160                      sizeof(sinListen)) == SOCKET_ERROR)
161             {
162                      printf("bind() error for sListen: %d\n", WSAGetLastError());
163                      return 1;
164             }
165
166             /*
167             Create and start thread to receive UDP packets.
168
169             HANDLE CreateThread(
170                      LPSECURITY_ATTRIBUTES lpThreadAttributes,  // ptr to attributes
171                      DWORD dwStackSize,  // initial thread stack size, in bytes
172                      LPTHREAD_START_ROUTINE lpStartAddress, // pointer to thread function
173                               LPVOID lpParameter,          // argument for new thread
174                               DWORD dwCreationFlags,       // creation flags
175                               LPDWORD lpThreadId // pointer to returned thread identifier
176                               );
177             */
178             DWORD  dwThreadId;
179             HANDLE hThread;
180             hThread = CreateThread(NULL, 0, ReceiveUDP, (LPVOID) sListen, 0, &dwThreadId);
181
182             // Create socket to send UDP datagrams
183             sSend = socket(AF_INET, SOCK_DGRAM, 0);
184             if (sSend == INVALID_SOCKET)
185             {
186                      printf("Error creating sSend socket: %d\n", WSAGetLastError());
187                      return 1;
188             }
189
190             // name the socket
191             sinSend.sin_family = AF_INET;
192             sinSend.sin_addr.s_addr = inet_addr(szDestIPAddr);
193             sinSend.sin_port = htons(dDestPort);
194
195             printf("Enter data to send to %s:%d\n", szDestIPAddr, dListenPort);
196
197             while (1)
198             {
199                      /*
200                        The sendto function is used to write outgoing data on a socket.
201                        For message-oriented sockets, care must be taken not to exceed
202                        the maximum packet size of the underlying subnets. You can get
203                        the size information from the getsockopt() API call. See the
204                        microsoft documentation for more information.
205
206                        The sendto( ) API function has the following parameters:
207
208                      int sendto(
```

```
209                       SOCKET s,
210                       const char* buf, // buffer containing the data to be transmitted
211                       int len,         // length of data in buf
212                       int flags,// indicator specifying the way in which the call is made
213                       const struct sockaddr* to, // optional ptr to dest address structure
214                       int tolen                  // size of address structure in bytes
215                               );
216                     */
217                     char TxBuf[512];
218                     gets(TxBuf);
219                     if (TxBuf[0] == '\0')
220                               break;
221                     else
222                     {
223                               int len = sizeof(sinSend);
224                               sendto(sSend, TxBuf, strlen(TxBuf), 0,
225                                         (struct sockaddr *)&sinSend, len );
226                               if (sSend == INVALID_SOCKET)
227                               {
228                                         printf("sendto() error: %d\n", WSAGetLastError());
229                                         break;
230                               }
231
232                               printf("Sent packet to: %s:%d\n",
233                               inet_ntoa(sinSend.sin_addr), ntohs(sinSend.sin_port));
234                     }
235           }
236
237           closesocket(sListen);
238           closesocket(sSend);
239           WSACleanup();
240           return 0;
241 }
```

Starting in main( ), the ProcessCmdLine(int argc, char **argv) function is called at the beginning of main( ) to do what you would expect: process any command line parameters. Specifically, you will pass the listening port number, destination port number and destination IP address.

Before you can call any winsock functions, you must load a winsock library. The call to initialize winsock is WSAStartup( ); when this function returns, the WSADATA structure contains the information about the version of winsock that was loaded. The MAKEWORD(1,1) macro specifies winsock version 1.1; you could specify (2,2) for version 2.2.

The listen socket, sListen, is then created with:

```
sListen = socket(AF_INET, SOCK_DGRAM, 0);
```

AF_INET specifies the Internet Protocol. The socket type is set to SOCK_DGRAM; for TCP you would use SOCK_STREAM. The last parameter specifies protocol, and a value of 0 tells the system to pick one based on the other two parameters.

Winsock applications specify IP addresses and ports using the SOCKADDR_IN structure, which is declared as sinListen in our program. The functions htonl( ) and htons( ) stand for “host to network long” and “host to network short”, and convert the byte order of long and short variables from the host platform (little endian for a typical pc running Windows) to standard network byte order (big endian). The INADDR_ANY parameter specifies that any incoming IP address will be accepted, the AF_INET specifies the Internet Protocol family, and the dListenPort parameter contains the value of the port the server will listen to.

The bind( ) function associates the listen socket with the listen port number.

A separate thread is created to handle an incoming datagrams. The CreateThread( ) API call takes the following parameters:

- ReceiveUDP: A pointer to the thread function to execute
- (LPVOID)sListen: When creating a thread it is possible to pass a parameter to the function as a pointer. In this case we are passing the listening socket, sListen, as pointer. It is later cast back to a socket in the ReceiveUDP( ) thread.
- &dwThreadId: A pointer to a variable that will hold the thread ID once it is created.

When CreateThread( ) is called successfully, the thread will start and begin listening for incoming datagrams. Each time a datagram arrives the sender information and data will be displayed.

Now that the application can receive incoming datagrams, we need to have a method of sending outgoing datagrams. An outgoing socket is created with the line:

```
sSend = socket(AF_INET, SOCK_DGRAM, 0);
```

As with sListen, the sSend socket needs a `SOCKADDR_IN` structure. The sinSend is declared and initialized to allow datagrams to be sent to the host destination IP address and destination port number specified on the command line.

The application then enters a while(1) loop that waits for user input strings and sends those strings using the sendto( ) function.

13.3.2 Running the Windows UDP Application

The application was built as a Win32 console application. To run it, open a command prompt and type:

"`UDPWin <listen port> <dest IP> <dest port>`" (and then press the "Enter" Key)

where <listen port> is the local port number to listen for incoming datagrams, <dest IP> and <dest port> are the IP address and port number of the host or device you wish to send datagrams to. For example,

```
UDPWin 2500 10.1.1.15 2501
```

will listen on port 2500, and send any data you type into the console to the host or device at 10.1.1.15:2501.

13.4 Writing a Windows JAVA UDP Client/Server Application

This example is a UDP Client/Server written in Java that can be run on any host computer running the Java Virtual Machine (JVM). This includes Windows, Unix, Linux, etc. **Note:** The NetBurner platform does not run the JVM.

To create a UDP Server on the NetBurner platform:

1. Create a DatagramSocket
2. Create a DatagramPacket
3. Use the receive( ) member function to block for incoming packets and store the packet.

To send a UDP datagram on the NetBurner platform:

1. Create a DatagramPacket
2. Use the send( ) member function to send the packet

13.4.1 Windows Java UDP Application Source Code

```
/*************************************************************************
 This is a UDP example Java application to send and receive UDP datagrams.
 It will listen on a port for incoming UDP packets, and allow the user to
 send strings as UDP datagrams to another host.

 This application can run on Windows, Unix or Linux hosts with the Java
 Virtual Machine installed. It is not designed to run on the NetBurner
 platform.

 Revision 1.0, May 31, 2003

*************************************************************************/

import java.net.*;
import java.io.*;

/*-------------------------------------------------------------
 UDP Server Class
 This class will:
 1. Parse command line parameters for the local port number,
    destination ip address, and destination port number.

 2. Obtain the local host name and ip address, then display all
    parameters to the console.

 3. Declare an instance of the client UDP class, which will
    handle local host user input and allow the user to send
     strings.

 4. Start the UDP server thread that will loop forever listening
    for incoming UDP packets on the designated port. All
     incoming messages will be displayed to the console window
     along with the sender's ip address and port number.
-------------------------------------------------------------*/
public class udpserv {
        protected DatagramSocket dgSocket;    // declare socket

        // Constructor
        public udpserv (int port) throws IOException {
            dgSocket = new DatagramSocket(port);    // allocate socket
        }

```

```
43      // execute() loop to wait for incoming datagram packets
44      public void execute()throws IOException {
45          while (true)
46          {
47              byte buffer[] = new byte[65536];
48              DatagramPacket dgPacket = new DatagramPacket(buffer, buffer.length);
49              dgSocket.receive(dgPacket);
50
51              System.out.print("Rec'd From: " + dgPacket.getAddress());
52              System.out.print(":" + dgPacket.getPort());
53              System.out.print(", " + dgPacket.getLength() + " bytes: \"");
54              String msg = new String(buffer, 0, dgPacket.getLength());
55              System.out.println(msg + "\"\n");
56          }
57      }
58
59
60      public static void main(String args[]) throws IOException {
61          int ListenPort, DestPort;
62          String DestIP;
63
64          if (args.length != 3)
65             throw new RuntimeException("Syntax:  udpserv <listen port> <dest IP> <dest
66  port>");
67          ListenPort = Integer.parseInt(args[0]);
68          DestIP     = args[1];
69          DestPort   = Integer.parseInt(args[2]);
70
71          // Display command line parameters
72          System.out.println("Listen Port: " + ListenPort);
73          System.out.println("Dest IP    : " + DestIP);
74          System.out.println("Dest Port  : " + DestPort);
75
76          InetAddress LocalHost = InetAddress.getLocalHost();
77          System.out.println("Local Host Name: " + LocalHost.getHostName());
78          System.out.println("Local Host IP Addr: " + LocalHost.getHostAddress());
79
80          // Create instance of udpserv class
81          udpserv serverUDP = new udpserv(ListenPort);
82          System.out.println("Starting UDP echo server, listening on port " + ListenPort);
83
84
85          // Convert destination IP string to InetAddress type
86          InetAddress DestAddr = InetAddress.getByName(DestIP);
87
88          // Start thread to accept user input strings and send them
89          // as UDP datagrams. This is the client side of this example
90          clientUDPThread ct = new clientUDPThread(DestAddr, DestPort);
91          ct.start();
92
93          // Start server execution to receive UDP datagrams
94          serverUDP.execute();
95      }
96  }
97
98
99
100 /*--------------------------------------------------------------------
101  UDP Client Class
102  This class simply waits for user input from the console, creates
103  a UDP datagram with the input data, and sends the datagram to
104  the designated destination host.
105  --------------------------------------------------------------------*/
106 class clientUDPThread extends Thread {
107
108     int DestPort;
109     InetAddress DestAddr;
110     protected DatagramSocket dgSocket;    // declare socket
111
112     public clientUDPThread(InetAddress DestinationAddr, int DestinationPort) throws
113 IOException
```

```
114         {
115             dgSocket = new DatagramSocket(DestPort);    // allocate socket
116             DestAddr = DestinationAddr;
117             DestPort = DestinationPort;
118         }
119
120         public String GetUserInput()
121         {
122             String s = "";
123
124             try {
125                 while (true) {
126                     char c = (char)System.in.read();
127                     if (c == '\r') {
128                         c = (char)System.in.read();
129                         if (c == '\n') {
130                             break;
131                         } else {
132                             continue;
133                         }
134                     } else if (c == '\n') {
135                         break;
136                     } else {
137                         s += c;    // store char
138                     }
139                 } // while
140
141             }
142             catch (Exception e) {
143                 System.err.println("Exception: " + e.toString());
144             }
145             return s;
146         }
147
148
149         public void run()
150         {
151             while (true)
152             {
153                 String msg = GetUserInput();
154                 byte[] buffer = msg.getBytes();
155                 System.out.println("Sending Message: \"" + msg + "\"");
156                 DatagramPacket packet = new DatagramPacket(buffer,
157                                                    buffer.length, DestAddr, DestPort);
158                 try {
159                     dgSocket.send(packet);
160                 } catch (Exception e) {
161                     System.err.println("Exception: " + e.toString());
162                 }
163             }
164         }
165
```

13.4.2 Running the Windows Java UDP Application

To run the application you will need to have the Java SDK installed. If you need to obtain the software package, go to http://www.javasoft.com and download it free of charge. Once installed, verify you have the correct path by opening a DOS window and typing "java" at the prompt and press the "Enter" key. If your path is correct, you will see the Java usage information message. If your path is incorrect, you will see an error message indicating the command is not recognized. Follow the Java installation documents to perform a correct installation.

To compile the application, open a command prompt and go to the directory containing the source code. The default directory is java\JavaWin. To compile the example, type "**javac udpserv.java**". The compilation will produce a file called udpserv.class.

To run the application, now type:

"`java udpserv  <listen port> <dest IP> <dest port>`"

where <listen port> is the local port number to listen for incoming datagrams, <dest IP> and <dest port> are the IP address and port number of the host or device you wish to send datagrams to. For example,

"`udpserv 2500 10.1.1.15 2501`"

will listen on port 2500, and send any data you type into the console to the host or device at 10.1.1.15:2501.

As a quick test you can run two instances of the application on your host pc. In the following example you will need to replace the IP address with the correct IP address for your host computer.

1. Open two command windows
2. In the first window, type "java udpserv 3000 10.1.1.100 4000"
3. In the second window, type "java udpserv 4000 10.1.1.100 3000".
4. Now anything you type in one window will appear in the other. Note that user input will only be sent once the enter key is pressed.

For additional information on UDP, please refer to your User Manual. From Windows: Start → Programs → Netburner NNDK → NNKD Users Manual.

14 uC/OS Real-Time Operating System

14.1 Overview

The uC/OS Operating system is a full featured pre-emptive multitasking Real-Time Operating System (RTOS). You can easily create tasks, semaphores, mail boxes and queues just to name a few features. As part of the NetBurner development package, the RTOS is pre-configured, integrated and running a default user task named UserMain( ) that can be used as your main program task. Detailed information can be found in the uC/OS Reference Library document in \nburn\docs\NetBurnerRuntimeLibrary directory.

The RTOS provides 63 priority levels numbered from 1 to 63. The lower the number, the higher the priority. Some of these tasks are reserved by the system, such as the idle task at level 63. You can specify a priority when the task is created, and change the priority later if you wish. A priority level can only be used by one task at a time. Be sure to check the return values when creating a task or changing a task priority to verify that the operation was successful.

The uC/OS system files are located in the \nburn\system directory. They are:

ucos.c	uC/OS function source code
ucosmain.c	uC/OS helper, debug and start-up functions. Also has the stack definition for the idle task.
ucosmcfc.c	ColdFire specific uC/OS functions. Specifically, this code module contains the OSTaskCreate( ) function, initializes the Task Control Blocks, and contains the stack checking code.
ucosmfca.s	Contains the assembler functions for the ColdFire port of uC/OS, including functions for task switching and the timer. Note that the file extension of ".s" designates an assembly language file.
main.c	Contains the main( ) function that does some system initialization and creates UserMain( ).

14.2 Pre-emptive Operation and Blocking

In a pre-emptive RTOS, the highest priority task ready to run will always run. This is an extremely important point, so I will mention it again: the highest priority task ready to run will always run. This is different than a Windows or Unix system that employs a round-robin approach in which each task will run based on its time slice. If you create a high priority task that can always run, then no lower priority tasks will ever run. Lower priority tasks can only run when a higher priority task blocks on a resource or time delay.

uC/OS functions that can cause a task to block:

```
OSTimeDly()
OSSemPend()
OSMBoxPend()
OSQPend()
OSFlagPendAny()
OSFlagPendAll()
```

I/O system calls that can cause a task to block

```
select()
read()   // including all variants with timeouts
write()  // unitl at least one char can be written
gets()
getchar()
fgets()
```

Network calls can cause a task to block:

```
Accept()
Creation of a UDP packet when initialized with received data
```

Functions that can be used to enable a task to be "ready to run":

```
OSMBoxPost()
OSQPost()
OSSemPost()
OSTimeTick()
```

Lets say you have two tasks: A and B. Task A has priority 50 and Task B has priority 51. Since Task A is of higher priority, it will always run (and Task B will never run) unless it calls a blocking function. Task B will then run for as long as Task A blocks; this could be 1 second due to a call to OSTimeDly(TICKS_PER_SECOND), until a shared resource is available due to a call to OSSemPend(), or until data is available from a network connection due to a select( ) call. It both tasks were in a blocking state, then the idle task (63) would run.

14.3 Default Configuration and Resources

The number and types of system tasks that will be running depends on which options and features you are using in your specific application. For example, if your application called StartHTTP( ) to enable web services, then a system task will be created that handles web server requests. The following is a list of system tasks. The default values are specified in \nburn\include\constants.h. The table below shows the information for the SB72 platform.

Name	Default Priority	Description
Ethernet Driver	38	Handles packets sent to the Ethernet hardware
IP Protocol	39	Handles the IP layer of the TCP/IP protocol stack
TCP Protocol	40	Handles the TCP layer of the TCP/IP protocol stack.
PPP Protocol	44	Handles the PPP layer of the TCP/IP protocol stack
HTTP Protocol	45	Handles the HTTP application for the web server.
Main Priority	50	This is a macro used for the default UserMain( ) priority.
Idle	63	What does the RTOS do when it has nothing important to do? It runs the Idle Task, which just loops and does nothing.

14.4 Creating Tasks

Whether you use the AppWizard in the IDE to create a new application, or start with one of the example programs, you will notice that the point at which you take control of the device is at the function called UserMain( ). The UserMain( ) task is a uC/OS task created by the system for you to use in your application. Normally, the first few lines will consist of system initialization functions such as the functions described in the Template example.

To create additional tasks, you use the OSTaskCreate( ) function. The following is an example program that allocates the task stack, implements a new task function, and launches the new task.

```
/*****************************************************************************
 Multiple task example
 This program creates 2 tasks and send prints messages from each.

 *****************************************************************************/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <dhcpclient.h>


//----- function prototypes -----
extern "C"
{
   void UserMain( void *pd );
}

DWORD MyTaskStack[USER_TASK_STK_SIZE];


/*-------------------------------------------------------------------
 MyTask
 -------------------------------------------------------------------*/
void MyTask( void *pdata )
{
   WORD data = *( WORD * ) pdata;  // cast passed parameter

   iprintf( "Data passed to MyTask(): %d\r\n", data );

   while ( 1 )
   {
      iprintf( "     Message from MyTask()\r\n" );
      OSTimeDly( TICKS_PER_SECOND );  // this is a blocking call
   }
}


/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   InitializeStack();
   OSChangePrio( MAIN_PRIO );

   if ( EthernetIP == 0 )
   {
      iprintf( "Attempting DHCP\r\n" );
      if ( GetDHCPAddress() != DHCP_OK )
         iprintf("*** DHCP Failed\r\n");
      else
         iprintf( "DHCP IP Address: ");
   }
   else
      iprintf( "Static IP Address: ");
   ShowIP(EthernetIP);iprintf( "\r\n" );
```

```
   EnableAutoUpdate();

   // This example passes data to the task, but this is optional
   WORD  MyTaskData = 1234;

   /* Create and launch the task. Note that we selected a priority of
      MAIN_PRIO+1, which means the new task is of lower priority than UserMain.
      The implication is that if UserMain does not block, MyTask will never
      run.
   */
   if ( OSTaskCreate( MyTask,
                      ( void * ) &MyTaskData,
                      ( void * ) &MyTaskStack[USER_TASK_STK_SIZE],
                      ( void * ) MyTaskStack,
                     MAIN_PRIO + 1 ) != OS_NO_ERR )
   {
      iprintf( "*** Error creating task\r\n" );
   }

   while ( 1 )
   {
      OSTimeDly( TICKS_PER_SECOND * 2 );
      iprintf( "Message from UserMain()\r\n" );
   }
}
```

14.5 Interrupts

To obtain specific information on interrupts used in your specific NetBurner device, please reference the Platform Documents that came with the development kit. Some common values and their functions are shown below:

Name	Interrupt Level	Description
Serial ports (UARTS)	3	Interrupts when a character is received or transmitted. Note: The NetBurner monitor operates in polled mode. To enable interrupts your application must close the serial ports with SerialClose( ), then open them with OpenSerial( ).
Ethernet	4	Ethernet receive and transmit.
System Timer	5	This is the RTOS system timer. By default it generates an interrupt 20 times per second. If necessary, this rate can be changed in \nburn\include\constants.h with: #define TICKS_PER_SECOND. This value should stay within the range of 10 to 200. For high resolution timing, we recommend using a different timer.

15 Protecting Shared Data Structures

15.1.1 Overview

The following RTOS mechanisms can be used to protect shared data resources. They are listed in a decreasing order of severity as regarding system latency (all pend and post functions are at the same level).

OSSemPend( ) OSSemPost( )	Protects an area of memory or resource. A task calls OSSemPend, which will block until the resource is available. OSSemPost releases the resource.
OSMboxPend( ) OSMboxPost( )	Same as semaphore, except a pointer variable is passed as the "message". A task or ISR can post a message, but only a task can pend on a message. Both the posting task and pending task must agree on what the pointer points to.
OSQPend( ) OSQPost( )	A Queue is basically an array of mailboxes. It is used to post one or more messages. When you initialize a Queue, you must specify the maximum number of messages. The first message posted to the queue will be the first message extracted from the queue (FIFO).
OSFifoPend( ) OSFifoPost( ) OSFifoPostFirst( ) OSFifoPendNoWait( )	A FIFO is similar to a queue, but is specifically designed to pass pointers to OS_FIFO structures. The first parameter of the structure must be a (void *) element, which is used by the operating system to create a linked list of FIFOs. When initializing a FIFO, you do not specify the maximum number of entries as with a queue. Instead, your application has the ability (and responsibility) to allocate memory (static or dynamic) in which to store the structures. This can be done statically by declaring global variables, or dynamically by allocating memory from the heap. As with a queue, the first message posted to the FIFO will be the first message extracted from the queue.
OSCritEnter OSCritExit OSCritObj	This is a counted critical section that restricts access to resources to one task at a time, sometimes called a "mutex". For example, you have a linked list that is maintained by 3 separate tasks. If one task is manipulating the list, you could first call OSCirtEnter for that object (the list). If any other task tries to manipulate the list, it will block at the OSCritEnter until the task that previously called OSCritEnter, calls OSCritExit. Note that the number of enter calls must match number of exit calls. OSCritObj is a C++ implementation that uses scoping to automatically call the enter and exit functions. See example below.
OSLock( ) OSUnlock( ) OSLockObj	Disables other tasks, but not interrupts. Increments for each OSLock, decrements for each OSUnlock. The C++ object OSLockObj was created to assist in making sure that an unlock is called for each lock. When an OSLockObj is created, the constructor calls OSLock( ). When the object goes out of scope, OSUnlock( ) is automatically called by the destructor.
USER_ENTER_CRITICAL USER_EXIT_CRITICAL	Macro that disables other tasks and interrupts. Increments count on enter, decrements on exit.

How do you decide which type of mechanism to use? Some guidelines are:

- If you need some type of signal, but do not need to pass any data, use a Semaphore. A semaphore is a single 32-bit integer that increments and decrements for each pend or post.
- If you want to pass a single 32-bit number, you can use a Mailbox or Queue. Most applications use the 32-bit number as the data, but it could also be a pointer to a structure or object. A queue is like an array of mailboxes. You declare the number of queue entrys a compile time.
- If you want to pass a structure or object, then use a FIFO. You may be wondering how a FIFO differs from a Queue. The difference is that a Queue has a predefined number of entries. The FIFO implementation uses a linked list, so the only limit to the number of entries is available memory. Using a FIFO is not as simple as any of the other mechanisms, because your application must implement some type of memory management to allocate and deallocate the FIFO objects. This is usually done by managing a predeclared array of objects, or through dynamic memory allocation. We encourage all embedded designers to avoid dynamic memory allocation if at all possible, since in any embedded system memory fragmentation could eventually occur and the call to allocate a new object could fail. If you create an array of objects at compile time you will always be guaranteed the maximum number can exist.

15.1.2 Semaphore Example

A semaphore is a protected variable that is used to control access to shared system resources (such as memory or serial ports), to signal the occurrence of events and task synchronization. A task can request a semaphore and wait until the resource or event takes place (called pending). A task can also post to a semaphore to indicate it no longer needs a resource, or to signal an event has taken place.

To create a semaphore you declare one of type OS_SEM and initialize with OSSemInit( ):

```
OS_SEM MySemaphore;
OSSemInit( &MySemaphore, 0 );  // set initial value to 0
```

Your application tasks can now use the post and pend functions on the semaphore:

```
OSSemPost( &MySemaphore );     // post to a semaphore
OSSemPend( &MySemaphore, 0 );  // pend on a semaphore
```

The second parameter in the OSSemPend( ) function specifies the number of time ticks to wait. A value of 0 waits forever. A good way to express a wait value is to use the TICKS_PER_SECOND definition provided by the RTOS: OSSemPend(&MySemaphore, TICKS_PER_SECOND * 5) to wait 5 seconds.

```
/*-------------------------------------------------------------------
 Semaphore Example Program
 This program will create 2 tasks and a semaphore. The UserMain task will
 block for a semaphore posted from MyTask.

 The output for this example is displayed through the serial debug port, which
 can be viewed with the MTTTY program.

 -------------------------------------------------------------------*/
#include <predef.h>
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <serial.h>
#include <dhcpclient.h>

//----- function prototypes -----
extern "C"
{
   void UserMain( void *pd );
}

//----- Global Variables
const char * AppName = "Semaphore Example";
DWORD MyTaskStack[USER_TASK_STK_SIZE];
OS_SEM MySemaphore;


/*-------------------------------------------------------------------
 MyTask
 This task will post to a semaphore every 3 seconds
 -------------------------------------------------------------------*/
void MyTask( void *pdata )
{
   while ( 1 )
   {
```

```
      OSTimeDly( TICKS_PER_SECOND * 3 );
      iprintf( "      MyTask: Posted to Semaphore\r\n" );
      OSSemPost( &MySemaphore );
   }
}


/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   InitializeStack();
   if ( EthernetIP == 0 )
   {
      iprintf( "Attempting DHCP\r\n" );
      if ( GetDHCPAddress() != DHCP_OK )
         iprintf("*** DHCP Failed\r\n");
      else
         iprintf( "DHCP IP Address: ");
   }
   else
      iprintf( "Static IP Address: ");
   ShowIP(EthernetIP);iprintf( "\r\n" );

   OSChangePrio( MAIN_PRIO );
   EnableAutoUpdate();

   OSSemInit( &MySemaphore, 0 );

   if ( OSTaskCreate( MyTask,
                      NULL,
                      ( void * ) &MyTaskStack[USER_TASK_STK_SIZE],
                      ( void * ) MyTaskStack,
                      MAIN_PRIO + 1 ) != OS_NO_ERR )
   {
      iprintf( "*** Error creating task\r\n" );
   }

   // The while loop will pend on MySemaphore and display a message
   // when a post is detected.
   while ( 1 )
   {
      iprintf( ">>> UserMain: Pending on Semaphore from MyTask\r\n" );
      OSSemPend( &MySemaphore, 0 );
      iprintf( "<<< UserMain: Semaphore was posted from MyTask!\r\n\r\n" );
   }
}
```

15.1.3 Mailbox Example

A mailbox is similar to a semaphore, except a pointer variable is passed as the “message”. A task or ISR can post a message, but only a task can pend on a message. Both the posting task and pending task must agree on what the pointer points to.

```
/*-------------------------------------------------------------------
 Multiple task user input example.
 This program illustrates how to create a task and use a mailbox to accept
 user input from a serial port.

 1. UserMain task will wait pend on a mailbox for user input
 2. UserInput task will block waiting for a user to type in data from the
    serial port
 3. This example also illustrates how to terminate a task

 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <serial.h>
#include <string.h>
#include <dhcpclient.h>

//----- function prototypes -----
extern "C"
{
   void UserMain( void *pd );
}


//----- global vars -----
DWORD   UserInputTaskStack[USER_TASK_STK_SIZE];
OS_MBOX UserInputMbox;
BOOL    bExitUserInputTask;


/*-------------------------------------------------------------------
 UserInputTask
 Once created, this task will loop unitl bExitUserInputTask = FALSE.
 The fgets will block until the user enters a string in mttty. This
 task will have a higher priority than UserMain.
 -------------------------------------------------------------------*/
void UserInputTask( void *pdata )
{
   static char strInput[1024];

   while ( !bExitUserInputTask )
   {
      fgets(strInput, 80, stdin);  // block until user enters a string

      int len = strlen(strInput);
      iprintf("Received %d bytes\r\n", len);

      // Remove any trailing \r or \n
      if ( len >= 2 )
      {
         for (int i=0; i < 2; i++)
         {
```

```
            if ( ( strInput[len-i] == '\r' ) || ( strInput[len-i] == '\n' ) )
               strInput[len-i] = '\0';
         }
      }

      // Post pointer to input string in mailbox
      OSMboxPost( &UserInputMbox, ( void * ) strInput );
   }

   // Not much to do in this example once the task exits
   iprintf("*** UserInputTask() terminated. Please reset program ***\r\n");
}


/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   putleds( 0 );
   InitializeStack();
   if ( EthernetIP == 0 )
   {
      iprintf( "Attempting DHCP\r\n" );
      if ( GetDHCPAddress() != DHCP_OK )
         iprintf("*** DHCP Failed\r\n");
      else
         iprintf( "DHCP IP Address: ");
   }
   else
      iprintf( "Static IP Address: ");
   ShowIP(EthernetIP);iprintf( "\r\n" );

   OSChangePrio( MAIN_PRIO );
   EnableAutoUpdate();

   /* We want to hve the UserInputTask block on fgets(), so
    * we need to be in serial interrupt mode, rather than the
    * default polled mode. To enable interrupts, close the
    * serial port and call OpenSerial()
    */
   OSTimeDly( TICKS_PER_SECOND );  // delay to allow previous prints
   SerialClose( 0 );
   int fd = OpenSerial( 0, 115200, 1, 8, eParityNone );
   ReplaceStdio( 0, fd );
   ReplaceStdio( 1, fd );
   ReplaceStdio( 2, fd );

   // Initialize the mailbox
   OSMboxInit( &UserInputMbox, NULL );

   bExitUserInputTask = FALSE;
   if ( OSTaskCreate( UserInputTask,
                      NULL,
                      ( void * ) &UserInputTaskStack[USER_TASK_STK_SIZE],
                      ( void * ) UserInputTaskStack,
                      MAIN_PRIO - 1 ) != OS_NO_ERR )
   {
      iprintf( "*** Error creating task\r\n" );
   }

   BYTE i = 0;
   iprintf("Enter String: ");
```

```
   while ( 1 )
   {
      BYTE err;

      /* We can use the Wait or No Wait version of OSMboxPend.
       * In this example we will use NoWait so that we can
       * show how multiple tasks can operate and count on the
       * LEDs while we are waiting for user input. The Wait
       * version of the function is:
       *   void *pmsg = OSMboxPend( &UserInputMbox, 0, &err );
       */
      void *pmsg = OSMboxPendNoWait( &UserInputMbox, &err );
      if ( pmsg != NULL )
      {
         iprintf( "\r\nUserMain Received: \"%s\"\r\n\r\n", ( const char * ) pmsg );
         iprintf("Enter String: ");

         // If user input was "exit", then send exit signal to UserInputTask()
         if ( strcmp((const char *)pmsg, "exit") == 0 )
            bExitUserInputTask = TRUE;
      }
      else
      {
        // Count on 8 LEDs on dev board while wainting for input
        putleds( i++ );
        OSTimeDly( 1 );
      }
   }
}
```

15.1.4 FIFO Example

A FIFO is similar to a queue, but is specifically designed to pass pointers to OS_FIFO structures. The first parameter of the structure must be a (void *) element, which is used by the operating system to create a linked list of FIFOs. When initializing a FIFO, you do not specify the maximum number of entries as with a queue. Instead, your application has the ability (and responsibility) to allocate memory (static or dynamic) in which to store the structures. This can be done statically by declaring global variables, or dynamically by allocating memory from the heap. As with a queue, the first message posted to the FIFO will be the first message extracted from the queue.

```
/*************************************************************************
 FIFO Example Program

 This program creates two tasks and a FIFO. Messages are sent  from one task
 to another using the FIFO. A timeout value for the pend function is used to
 illustrate the timeout feature of the FIFO.

 *************************************************************************/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <ucos.h>
#include <serial.h>
#include <dhcpclient.h>
#include <netinterface.h>


#define NUM_POSTS (5)

//----- function prototypes -----
extern "C"
{
   void UserMain( void *pd );
}


//----- structs -----
typedef struct // Pointers to this structure will be passed in the FIFO
{
      void * pUsedByFifo;  // Don't modify this value, and keep it first
      int    x;            // Any other members come after the pointer
      BOOL   free;         // free = TRUE if structure is free to be used.
} MyStructure;


//----- global vars -----
DWORD FifoPostTaskStack[USER_TASK_STK_SIZE];
OS_FIFO MyFIFO;
MyStructure StructArray[5];


/*-------------------------------------------------------------------
 Init FIFO members
 -------------------------------------------------------------------*/
void InitFifoMembers()
{
   for ( int i = 0; i < NUM_POSTS; i++ )
   {
      StructArray[i].x = 0;
```

```
      StructArray[i].free = TRUE;
   }
}


/*-------------------------------------------------------------------
 Find free FIFO structure.
 Rather than use dynamic memory allocation, this example uses a
 declared array of structures with a structure element that indicates
 if a particular structure is being used. This function will return
 the index of the first free structure array member, or -1 if all
 members are being used.
 -------------------------------------------------------------------*/
int FindFreeFifoStruct( )
{
   int index = -1;
   int i = 0;

   do {
      if ( StructArray[i].free )
         index = i;
      else
         i++;
   } while ( ( i < NUM_POSTS ) && ( index == -1 ) );

   return index;
}


/*-------------------------------------------------------------------
 FIFO Post Task
 -------------------------------------------------------------------*/
void FifoPostTask( void *pdata )
{
   while ( 1 )
   {
      iprintf("\r\n     FIFOPostTask():\r\n");
      for ( int count = 0; count < NUM_POSTS; count++ )
      {
         int i;
         while ( (i = FindFreeFifoStruct() ) < 0 )
         {
            iprintf("Waiting for free FIFO structure\r\n");
            OSTimeDly( TICKS_PER_SECOND / 2 );
         }

         StructArray[i].x = count;
         StructArray[i].free = FALSE;
         // Put a message in the Fifo
         OSFifoPost( &MyFIFO, ( OS_FIFO_EL * ) &StructArray[i] );
         iprintf( "     Posted FIFO StructArray[%d].x = %d\r\n", i, count );
      }
      // This delay simulates a resource taking time to pend.
      iprintf("     Delaying 9 seconds before next FIFO post\r\n\r\n");
      OSTimeDly( TICKS_PER_SECOND * 9 );
   }
}


/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
```

```
{
   DWORD FifoCnt = 0;

   InitializeStack();

   if ( EthernetIP == 0 )
   {
      iprintf( "Trying DHCP\r\n" );
      GetDHCPAddress();
      iprintf( "DHCP assigned the IP address of :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }
   else
   {
      iprintf( "Static IP address set to :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }

   OSChangePrio( MAIN_PRIO );
   EnableAutoUpdate();

   OSFifoInit( &MyFIFO );
   InitFifoMembers();

   if ( OSTaskCreate( FifoPostTask,
                      NULL,
                      ( void * ) &FifoPostTaskStack[USER_TASK_STK_SIZE],
                      ( void * ) FifoPostTaskStack,
                      MAIN_PRIO - 1 ) != OS_NO_ERR )
   {
      iprintf( "*** Error creating FIFO task\r\n" );
   }

   while ( 1 )
   {
      iprintf( "\r\n>>> Calling OSFifoPend()...\r\n" );
      MyStructure *pData = ( MyStructure * ) OSFifoPend( &MyFIFO, TICKS_PER_SECOND * 5
);
      if ( pData == NULL )
      {
         // Because of the OSTimeDly() in the post task, we will timeout
         // once per sequence on purpose.
         iprintf( "    Timeout in OSFifoPend(), waiting for next FIFO Post\r\n" );
      }
      else
      {
         iprintf( "    FIFO Read #%d, pData->x = %d\r\n", FifoCnt++, pData->x );
         pData->free = TRUE;
      }
   }
}
```

15.1.5 OSCritObject Example

`OSCritEnter(), OSCritExit()` and an `OSCritObj` are enably an application to use counted critical sections that restrict access to resources to one task at a time (also called a "mutex"). For example, you have a linked list that is maintained by 3 separate tasks. If one task is manipulating the list, you could first call `OSCritEnter()` for that object (the list). If any other task tries to manipulate the list, it will block at the `OSCritEnter()` call in that task until the task that previously called `OSCritEnter()`, calls `OSCritExit()`. Since this is a counting critical section implementation, the number of enter calls must match number of exit calls for each task.

OSCritObj is a C++ implementation that uses scoping to automatically call the enter and exit functions so you do not need to manually match each enter with an ext.

In comparison with OsLock, OsCritEnter does not restrict task swapping unless two tasks want to access the same resource. If you used OsLock, then ALL task swapping would be prevented.

```
/*-------------------------------------------------------------------
 This example will show how to create critical sections with the standard
 C type function calls OSCritEnter() and OSCritLeave(), and compare
 them to the the C++ object type which eliminates the problem of matching
 each OSCritEnter() to a corresponsing OSCritLeave().

 While this example is trivial with only a simple global variable, critical
 sections need to be used for objects, such as linked lists, that could
 be changed by multiple tasks. In such cases, each task would use a
 critical section to ensure the modifications could be completed before
 a task switch occurred.
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <ucos.h>


extern "C" {
   void UserMain(void * pd);
}

const char * AppName="OsCritObj Example";

int GlobalVariable = 0; // A global variable

/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain(void * pd)
{
   InitializeStack();

   if ( EthernetIP == 0 )
   {
      iprintf( "Trying DHCP\r\n" );
      GetDHCPAddress();
      iprintf( "DHCP assigned the IP address of :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }
   else
   {
```

```
      iprintf( "Static IP address set to :" );
      ShowIP( EthernetIP );
      iprintf( "\r\n" );
   }

   OSChangePrio(MAIN_PRIO);
   EnableAutoUpdate();


   //===== Implement critical section with OSCritEnter() & OSCritLeave() =====
   OS_CRIT cs;            // The critical section identifer
   OSCritInit( &cs );

   /* The second parameter is a timeout value; the number of time
    * ticks to wait for "cs" to be available if another task
    * currently ownes it. A value of 0 waits forever.
    */
   OSCritEnter( &cs, 0 );

   GlobalVariable = 1;

   /* You need to call OSCritLeave() for each call to OSCritEnter().
    * While easy to see in this case, in larger applications it becomes
    * easier to get out of sync.
    */
   OSCritLeave( &cs );
   iprintf("GlobalVariable = %d\r\n", GlobalVariable);


   //===== Implement critical section with critical C++ object =====
   OS_CRIT cs2;           // The critical section identifer
   OSCritInit( &cs2 );

   {  /* opening scope operator
       * Now create an OSCriticalSectionObj object. When the object gets
       * created, the constructor will be called, which will call OSCritEnter()
       * This call always sets the timeout to "wait forever".
       */
      OSCriticalSectionObj UserMainCriticalSection( cs2 );

      GlobalVariable = 2;

      // Note that the OSCritLeave() call is not required, so you can never
      //  get out of sync with OSCritEnter()

   }  // destructor gets call when object goes out of scope, which calls OSCritLeave()
   iprintf("GlobalVariable = %d\r\n", GlobalVariable);

   while(1);
   {
      OSTimeDly( TICKS_PER_SECOND*5 );  // this is a blocking call
   }

}
```

15.1.6 OSFlags Example

OSFlags enables a function or task to pend on multiple flags/events, in contrast to a OSSemaphore which which can pend on only a single event. The OSFlag implementation is basically a 32-bit bitmap in which each bit position represents a "flag". You create a OSFlag object with OSFlagCreate(), then set, clean and read the flags with the appropriate functions. There are a number of functions used to monitor or pend on the flags, and provide the ability to pend on any one or more of the flags being set, or pending on all of flags being set at one time.

Flag Functions

OSFlagCreate ---Creates and initializes an OS_FLAGS object
OSFlagSet --- Sets the bits asserted bits_to_set
OSFlagState ---Returns the current value of flags
OSFlagClear --- Clears the bits asserted in bits_to_clr

OSFlagPendAll ---- Waits until all of the flags indicated by mask are set
OSFlagPendNoWait --- Checks (but does not wait) if all of the flags indicated by the mask are set
OSFlagPendAny --- Waits until any of the flags indicated by the mask are set
OSFlagPendAnyNoWait ---Checks (but does not wait) if any of the flags indicated by the mask are set

```
/*-------------------------------------------------------------------
 This illustrates how OSFlags can be used to pend on multiple events.
 UserMain creats 3 tasks, each of which will set a OSFlag after a
 time delay of some number of seconds. UserMain will then pend and
 block until ANY of the 3 flags are set. You can also modify the example
 to pend until ALL of the 3 flags are set.
 -------------------------------------------------------------------*/

/* NB Library Definitions */
#include "predef.h"

/* C Standard Library */
#include <ctype.h>
#include <stdio.h>
#include <string.h>

/* Portability & uCos Definitions */
#include <basictypes.h>
#include <ucos.h>
#include <ucosmcfc.h>
#include <utils.h>

/* NB Runtime Libraries */
#include <constants.h>
#include <system.h>
#include <iosys.h>
#include <http.h>

/* NB Network Stack */
#include <nettypes.h>
#include <ip.h>
#include <tcp.h>

/* Features */
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <taskmon.h>
```

```
#ifdef _DEBUG
   #include <NetworkDebug.h>
#endif


/*
 * OSFlag ( gOsFlag ) assigned bits
 */
#define TASK_FLAG_1      ( ( DWORD )0x00000001 )
#define TASK_FLAG_2      ( ( DWORD )0x00000002 )
#define TASK_FLAG_3      ( ( DWORD )0x00000004 )


extern "C"
{
   void UserMain( void* notUsed );
}


const char* AppName="OSFlags Example";


/*
 * Declare OSFlag
 */
OS_FLAGS gOsFlag;


void OsFlagTask1( void* notUsed )
{
   while ( TRUE )
   {
      OSTimeDly( ( 2 * TICKS_PER_SECOND ) );
      OSFlagSet( &gOsFlag, TASK_FLAG_1 );
   }
}

void OsFlagTask2( void* notUsed )
{
   while ( TRUE )
   {
      OSTimeDly( ( 4 * TICKS_PER_SECOND ) );
      OSFlagSet( &gOsFlag, TASK_FLAG_2 );
   }
}

void OsFlagTask3( void* notUsed )
{
   while ( TRUE )
   {
      OSTimeDly( ( 6 * TICKS_PER_SECOND ) );
      OSFlagSet( &gOsFlag, TASK_FLAG_3 );
   }
}


/*-------------------------------------------------------------------
 * UserMain
 * ------------------------------------------------------------------*/
void UserMain( void* notUsed )
{
   DWORD gOsFlagState = 0;
```

```
InitializeStack();
if ( EthernetIP == 0 )
{
   iprintf( "Attempting DHCP\r\n" );
   if ( GetDHCPAddress() != DHCP_OK )
      iprintf("*** DHCP Failed\r\n");
   else
      iprintf( "DHCP IP Address: ");
}
else
   iprintf( "Static IP Address: ");
ShowIP(EthernetIP);iprintf( "\r\n" );

OSChangePrio( MAIN_PRIO );
EnableAutoUpdate();
EnableTaskMonitor();

/* Create and initialize flags */
OSFlagCreate( &gOsFlag );

/* Create example tasks. With release 2.2 and later, you can create
 * tasks with names. Otherwise, create without names. MAIN_PRIO is the
 * priority of UserMain, so each of the tasks are of lower priority,
 * and UserMain must block to allow them to run.
 */
/*
OSSimpleTaskCreatewName( OsFlagTask1, ( MAIN_PRIO + 1), "OsFlagTask1");
OSSimpleTaskCreatewName( OsFlagTask2, ( MAIN_PRIO + 2), "OsFlagTask2");
OSSimpleTaskCreatewName( OsFlagTask3, ( MAIN_PRIO + 3), "OsFlagTask3");
*/
OSSimpleTaskCreate( OsFlagTask1, ( MAIN_PRIO + 1) );
OSSimpleTaskCreate( OsFlagTask2, ( MAIN_PRIO + 2) );
OSSimpleTaskCreate( OsFlagTask3, ( MAIN_PRIO + 3) );

#ifndef _DEBUG
EnableSmartTraps();
#endif

#ifdef _DEBUG
InitializeNetworkGDB_and_Wait();
#endif

while ( TRUE )
{
   /* There are two pending functions we can use:
    *   OSFlagPendAny(), returns if one or more of the flags are set
    *   OSFlagPendAll(), returns only if all flags are set
    *
    * This example will use the Any version, which is useful for
    * a task that is reporting on multiple events. It will return
    * if any of the flags in TASK1, TASK2 or TASK3 are set.
    */
   OSFlagPendAny( &gOsFlag, ( TASK_FLAG_1 | TASK_FLAG_2 | TASK_FLAG_3 ), 0 );
   // Only ONE of the OSFlagPend functions can be used at a time
   //OSFlagPendAll( &gOsFlag, ( TASK_FLAG_1 | TASK_FLAG_2 | TASK_FLAG_3 ), 0 );
   iprintf("OSFlagPendAny() detected the following flag(s) are set: ");

   /* Get the state of the flags so we can determine which ones are set */
   gOsFlagState = OSFlagState( &gOsFlag );

   /* Now test each flag (bit position) */
   if ( ( gOsFlagState & TASK_FLAG_1 ) == TASK_FLAG_1 )
      iprintf( "TASK1 " );
```

```
      if ( ( gOsFlagState & TASK_FLAG_2 ) == TASK_FLAG_2 )
         iprintf( "TASK2 " );

      if ( ( gOsFlagState & TASK_FLAG_3 ) == TASK_FLAG_3 )
         iprintf( "TASK3 " );

      iprintf("\r\n");

      /* Clear the Flag mask. You might need to signal the setting
       * task when done in a real application
       */
      OSFlagClear( &gOsFlag, gOsFlagState );
   }
}
```

16 File Descriptors

16.1 Overview

The NetBurner development package integrates the uC/OS operating system, TCP/IP stack and other system peripherals with a file I/O system and file descriptors. A **file descriptor** is a handle to a network socket, serial port or other system peripheral. May of the API functions pass a file descriptor to your application functions, such as the web server MyGet function example in this guide.

There are a maximum of 64 file descriptors:

- 0 – 2 for stdin, stdout and stderr
- 3 – 4 for the first two UART serial ports, 0 and 1.
- 5 – 36 for TCP (32 in total)
- 37 – 63 for expansion (additional UARTs, TCP sockets, or custom)

The expansion file descriptor positions can be used for many things, including additional serial ports, such as an external UART, or TCP ports.

16.2 Creating Custom I/O Drivers Using File Descriptors

The header file in \nburn\include\iointernal.h defines the programming interface functions to create a custom file descriptor. The header file content for tools release 1.98 is shown below, but be sure to check the header file for your particular tools installation for the latest function definitions.

```
void SetDataAvail( int fd );
void ClrDataAvail( int fd );

void SetWriteAvail( int fd );
void ClrWriteAvail( int fd );

void SetHaveError( int fd );
void ClrHaveError( int fd );

struct IoExpandStruct
{
      int ( * read )  ( int fd, char *buf, int nbytes );
      int ( * write ) ( int fd, const char *buf, int nbytes );
      int ( * close ) ( int fd );
      void *extra;
};

int GetExtraFD( void *extra_data, struct IoExpandStruct *pFuncs );
void *GetExtraData( int fd );
void FreeExtraFd( int fd );

// The TCP state call back, fd = socket has new data, fd< 0 means error
typedef void  ( tcp_read_notify_handler )( int tcp_fd );

// When data comes in or the TCP connection enters an error state,
// register a callback to handle the event
void RegisterTCPReadNotify( int tcp_fd, tcp_read_notify_handler *newhandler );
```

The Set and Clr functions are used to update the state of your fd device for file I/O functions such as select( ), read( ) and write( ). The IoExpandStruct is used to declare function pointers for the read, write and close functions that will be implemented in your application, as well as an “extra” void pointer that you can use for whatever you wish.

GetExtraFD( ) is the function that will return a fd for the object passed as the IoExpandStruct. The extra_data void pointer is optional, and can be used to pass data into your fd. You can read the extra_data value at any time with the GetExtraData( ) function. FreeExtraFd( ) will release the fd back to the pool of available fds.

The tcp_read_notify_handler and RegisterTCPReadNotify are callback functions. These functions will get called by the system if you define them for the corresponding TCP event.

16.3 Using File Descriptors to Pend on Multiple Events

Once you have created a file descriptor you can use the select( ) function call just as you would for network or serial file descriptors. In fact, you can pend on a mixture of them all at the same time.

16.4 Example: Circular Buffer Implementation Using File Descriptors

The following example uses file descriptors to implement a circular buffer.

```
/*******************************************************************
   The NetBurner sytem software has support for creating your own
   custom I/O device as a "file descriptor". It is encapsulated in
   the nburn\include\iointernal.h header file.

   This is a trivial example of creating a circular buffer file
   descriptor.  It will create a 256 byte circular buffer that
   you can write() to and read() from. The 256 byte size makes the
   circular buffer wrap around easy - just use bytes and the
   rollover math is simple

   In any real world example of a new I/O dvice the device would
   probably be interrupt driven. For interrupts, you should
   use USER_ENTER_CRITICAL and USER_EXIT_CRITICAL to protect the
   internal data structures.

   The 6 fd status functions are interrupt safe:
     void SetDataAvail( int fd );
     void ClrDataAvail( int fd );

     void SetWriteAvail( int fd );
     void ClrWriteAvail( int fd );

     void SetHaveError( int fd );
     void ClrHaveError( int fd );
 *******************************************************************/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <taskmon.h>
#include <iointernal.h>
```

```
const char * AppName = "ExtraFD Example";

static IoExpandStruct cb_ioexpand;

extern "C"
{
   void UserMain( void *pd );
}


// A circular buffer object, referred to as a "cbo"
struct CircularBufferObject
{
      OS_CRIT critical_section;     // Critical section to make it task safe.
      BYTE get_pointer;             // Where we get chars in the buffer
      BYTE put_pointer;             // Where we put chars in the buffer
      char data_buffer[256];        // Data storage space
};


/*--------------------------------------------------------------------
  Function to read from the circular buffer
 --------------------------------------------------------------------*/
int cbo_read( int fd, char *buf, int nbytes )
{
   // Get the cbo object associated with this fd
   CircularBufferObject * pCbo = ( CircularBufferObject * ) GetExtraData( fd );

   int number_read = 0;

   // Set up the critical section so we have exclusive access
   OSCritEnter( &pCbo->critical_section, 0 );

   // If both get and put pointers point at each other, there is nothing in
   // the buffer, otherwise we can read the data.
   if ( pCbo->get_pointer != pCbo->put_pointer )
   {
      // Read as long as data is availible and we have not read the max allowed
      while ( ( pCbo->get_pointer != pCbo->put_pointer ) && ( number_read < nbytes ) )
      {
         *buf++ = pCbo->data_buffer[pCbo->get_pointer++];
         number_read++;
      }

      if ( pCbo->get_pointer == pCbo->put_pointer )
      {
         // We have read everything. Now update the status so fd functions like
         // select() will work properly
         ClrDataAvail( fd );
      }
   }// We had some data to read


   if ( number_read > 0 )
   {
      // We have read something so there should now be space in the write buffer
      // update status for select() function
      SetWriteAvail( fd );
   }

   OSCritLeave( &pCbo->critical_section );
```

```
   return number_read;
}


/*-------------------------------------------------------------------
  Function to write to the circular buffer
 -------------------------------------------------------------------*/
int cbo_write( int fd, const char *buf, int nbytes )
{
   // Get the cbo object associated with this fd
   CircularBufferObject * pCbo = ( CircularBufferObject * ) GetExtraData( fd );

   int number_written = 0;

   // Enter the critical section
   OSCritEnter( &pCbo->critical_section, 0 );

   if ( ( BYTE ) ( pCbo->put_pointer + 1 ) != pCbo->get_pointer )
   {
      // We can write data while there is space and we have bytes to write
      while ( ( ( BYTE ) ( pCbo->put_pointer + 1 ) != pCbo->get_pointer ) &&
              ( number_written < nbytes ) )
      {
         pCbo->data_buffer[pCbo->put_pointer++] = *buf++;
         number_written++;
      }

      if ( ( BYTE ) ( pCbo->put_pointer + 1 ) == pCbo->get_pointer )
      {
         // Buffer is full, so mark the fd as full
         ClrWriteAvail( fd );
      }

      if ( number_written )
      {
         // We wrote something so we have data ready to be read
         SetDataAvail( fd );
      }
   }

   OSCritLeave( &pCbo->critical_section );
   return number_written;
}


/*-------------------------------------------------------------------
  Function to close a fd
 -------------------------------------------------------------------*/
int cbo_close( int fd )
{
   // Normally you would have to clean up your I/O device before closing it.
   // Here we just close the fd.
   FreeExtraFd( fd );
   return 0;
}


/*-------------------------------------------------------------------
  Function to open the circular buffer, which returns the file
  descriptor that will be associated with the circular buffer object
  passed as the function parameter.
 -------------------------------------------------------------------*/
int OpenCircularBuffer( CircularBufferObject *pCbo )
```

```
{
   // First initialize my circular buffer
   pCbo->get_pointer = 0;
   pCbo->put_pointer = 0;
   OSCritInit( &pCbo->critical_section );

   // Set up the I/O expand function to point at the cbo functions
   cb_ioexpand.read = cbo_read;
   cb_ioexpand.write = cbo_write;
   cb_ioexpand.close = cbo_close;

   // Get the fd for the cbo
   int fde = GetExtraFD( ( void * ) pCbo, &cb_ioexpand );

   // Now set up the file descriptor state for our fd
   ClrDataAvail( fde );   // no data to be read
   SetWriteAvail( fde );  // space available for writes
   ClrHaveError( fde );   // no errors yet

   return fde;
}


/*-------------------------------------------------------------------
  UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   InitializeStack();
   if ( EthernetIP == 0 )
   {
      GetDHCPAddress();
   }
   OSChangePrio( MAIN_PRIO );
   EnableAutoUpdate();
   EnableSmartTraps();
   EnableTaskMonitor();

   // Create the circular buffer objects. In this example we will create 2
   static struct CircularBufferObject cb1, cb2;

   // Open the buffers
   int fdcb1 = OpenCircularBuffer( &cb1 );
   int fdcb2 = OpenCircularBuffer( &cb2 );

   iprintf( "Application started\r\n" );
   while ( 1 )
   {
      // Write to the circular buffers
      writestring( fdcb1, "Testing circular buffer #1" );
      writestring( fdcb2, "Testing circular buffer #2" );

      OSTimeDly( TICKS_PER_SECOND );  // Delay so loop runs slow

      // Read the data out of buffer 1
      while ( dataavail( fdcb1 ) )
      {
         char buf[2];
         int rv = read( fdcb1, buf, 1 );
         if ( rv == 1 )
         {
            iprintf( "[%c]", buf[0] );
         }
```

```
      }
      iprintf( "\r\n" );

      // Read the data out of buffer 2
      while ( dataavail( fdcb2 ) )
      {
         char buf[2];
         int rv = read( fdcb2, buf, 1 );
         if ( rv == 1 )
         {
            iprintf( "(%c)", buf[0] );
         }
      }
      iprintf( "\r\n" );
   }
}
```

17 Multiple Network Interfaces

NetBurner network devices can have multiple interfaces, both in software and in hardware.

Hardware interfaces:

- Primary Ethernet interface
- Secondary Ethernet interface. A second Ethernet interface will be automatically initialized by the system.
- Wifi interface. A wifi interface needs to be added using the AddWifiInterface( ) function.
- PPP interface.

On platforms such as the MOD54417 there are 2 Ethernet interfaces that can be configured as two independent ports, each with its own MAC address, or as a network switch. In switch mode only one logical interface is presented to the TCP/IP stack described below.

Additional software interfaces:

- Multi-home. A single physical Ethenet port can be operate on multiple IP addresses. For example, 10.1.1.10 and 192.168.1.10.
- Auto IP. Each wired Ethernet port will automatically be assigned a class B address (169.168.x.x) so network communication is possible even if the static IP address is 0.0.0.0 and a DHCP server is not available on the network.

The TCP/IP stack assigns logical interfaces to the hardware interfaces. The tables below show the logical interface numbers for the most common configurations. **Note the Auto IP numbering for the dual Ethernt platform occurs with Ethernet hardware interface 1 first. A wifi interface is optional.**

Single Ethernet Hardware	
Hardware Interface	**Logical Interface**
Ethernet 0	1
Auto IP	2
Wifi	3

Dual Ethernet Hardware, Independent Mode	
Hardware Interface	**Logical Interface**
Ethernet 0	1
Ethernet 1	2
Auto IP 1	3
Auto IP 0	4
Wifi	5

The maximum number of interfaces is configured in netinterfaces.cpp by MAX_INTERFACES. The default value is 4, plus an additial 10 if multihome or is enabled.

17.1 Wifi

Example programs for adding a Wifi interface are located in the \nburn\examples directory.

17.2 Multi-Home

Multi-home means that your device can operate on more then one IP address. However, your device has only one MAC address per Ethernet hardware interface, so there are some limitations such as DHCP can only be used for one address per Ethernet hardware interface since a separate MAC address is required for each DHCP client.

To add Multi-Home functionality to your application:

1. Uncomment the "#define MULTIHOME" in \nburn\include\predef.h. Multi-home is disabled by default at a system level so that non-multi-home applications do not incur the additional overhead.

2. Add the following includes in your application:
   #include <multihome.h>
   #include <netinterface.h>

3. Use the API function:
   int AddInterface(IPADDR addr, IPADDR mask, IPADDR gateway, int root_if = 0);
   to add additional interfaces. You must specify the first 3 parameters.

4. Select your project in NBEclipse, then from the main menu select NBEclipse ➢ "Rebuild System Files" to rebuild the the system library.

An example program is shown below. Please refer to the example programs for adding a Multi-home interface in the \nburn\examples directory for the latest changes to this code.

```
/*****************************************************************
 Multihome application example.

 This program will demonstrate how to implement both a DHCP address
 and static IP addres using the Multihome functionality of the
 NetBurner TCP/IP Stack. The NetBurner device will try to obtain
 a dynamic IP address from a DHCP Server for the first Network
 Interface, and set a static IP address for the second Network Interface.
 The end result is that the NetBurner device will respond to either
 IP address. The example will print debug information out the debug
 serial port, and display the IP address information on a web page
 that can be accessed from either IP address.

 To enable multihome capability, you must uncomment the MULTIHOME
 definition in the include file \nburn\include\predef.h, and rebuild
 the system files. If using the IDE, select Build->Rebuild All, If
 using the command line, go to \nburn\system and run "make clean".

*****************************************************************/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <tcp.h>
#include <taskmon.h>
#include <multihome.h>
#include <netinterface.h>

// Check for Multihome options and display compiler warnings if not enabled
#ifndef MULTIHOME
```

```
#error This example requires that MULTIHOME be uncommented in the header      \
         file nburn\include\predef.h, and that the system files be rebuilt.
#endif


// App name for IPSetup
const char * AppName = "Multihome Example";

extern "C"
{
   void UserMain( void *pd );
   void WebDisplayIpSettings( int sock, PCSTR url );
}


// Global variables
InterfaceBlock *DhcpIb, *StaticIb, *MultiHome1Ib;

// External function
void ShowIPOnFd( int fd, IPADDR ia );

/*-------------------------------------------------------------------
 Display IP setting on web page
 -------------------------------------------------------------------*/
void WebDisplayIpSettings( int sock, PCSTR url )
{
   // Show header message indicating source and destination
   writestring( sock, "Received request from " );
   ShowIPOnFd( sock, GetSocketRemoteAddr( sock ) );
   writestring( sock, " on IP address: " );
   ShowIPOnFd( sock, GetSocketLocalAddr( sock ) );
   writestring( sock, "\r\n<br><br>" );

   // Show IP address for multihome
   writestring( sock, "Valid IP Address for this device:<br>\r\n" );
   writestring( sock, "DHCP:");
   ShowIPOnFd( sock, DhcpIb->netIP );
   writestring( sock, "<br>\r\n" );

   writestring( sock, "AutoIP:");
   ShowIPOnFd( sock, StaticIb->netIP );
   writestring( sock, "<br>\r\n" );

   writestring( sock, "MultiHome 1:");
   ShowIPOnFd( sock, MultiHome1Ib->netIP );
   writestring( sock, "<br>\r\n" );
}


/*-------------------------------------------------------------------
 UserMain
 -------------------------------------------------------------------*/
void UserMain( void *pd )
{
   InitializeStack();   // Init TCP/IP Stack

   /* Get a DHCP address if the stored IP Address in flash memory is
      set to 0.0.0.0 (set via IPSetup or serial interface).
      You may want to add a check for the return value from this function
      See the function definition in  \nburn\include\dhcpclient.h
   */
   GetDHCPAddressIfNecessary();
```

```
   OSChangePrio( MAIN_PRIO ); // Change priority from highest to something in the
middle
   EnableAutoUpdate();       // Enable the ability to update code over the network
   StartHTTP();              // Start the web serrver
   EnableTaskMonitor();      // Enable the Task scan utility

   // Add the multihome interface at fixed ip address, IP, Mask, Gateway
   AddInterface( AsciiToIp( "10.1.1.240" ), AsciiToIp( "255.255.255.0" ), 0 );

   int DhcpInterface = GetFirstInterface();  // Get first interface identifier
   DhcpIb = GetInterFaceBlock( DhcpInterface );  // Get interface data
   iprintf( "DHCP IP Address: " ); ShowIP( DhcpIb->netIP ); iprintf( "\r\n" );

   // Get next interface identifier
   int StaticInterface = GetnextInterface( DhcpInterface );
   StaticIb = GetInterFaceBlock( StaticInterface );  // Get interface data
   iprintf( "AutoIP Address: " ); ShowIP( StaticIb->netIP ); iprintf( "\r\n" );

   // Get next interface identifier
   int MultiHome1Interface = GetnextInterface( StaticInterface );
   MultiHome1Ib = GetInterFaceBlock( MultiHome1Interface );  // Get interface data
   iprintf( "Multihome 1 IP Address: " ); ShowIP( MultiHome1Ib->netIP );
   iprintf( "\r\n" );

   while ( 1 )
   {
      OSTimeDly( TICKS_PER_SECOND * 5 );
   }
}
```

18 Using the ColdFire Processor On-Chip SRAM

18.1 Introduction

In addition to the SDRAM on NetBurner modules, the ColdFire processor on your NetBurner device may have on-chip SRAM available for your application to use. For example, the 5272 has 4k, and the 5234, 5270 and 5282 have 64k bytes of SRAM. The primary reason to use the on-chip SRAM is that it has single-cycle access time. The SRAM is initialized by the NetBurner system software, and the first 0x400 bytes of SRAM is reserved for the vector table.

IMPORTANT: The NetBurner system uses the first 400 bytes of the SRAM for the application vector table. Your application must not write to that space.

Starting with tools release 2.1 RC6, the NetBurner system software uses the on-chip SRAM by default for fast network buffering and OS task switching. This provides a system speed increase of up to 50%, and is the recommended use for the SRAM. Using this SRAM in this way will provide a faster overall system performance increase than if you use the SRAM for a specific application function. If you prefer, the system use of SRAM can be disabled.

In addition to the fast buffering and task switching, you can specify that certain variable be located in the SRAM with the FAST_USER_VAR and FAST_USER_STK macros as described in the following section.

18.2 Using Fast SRAM in Tools Release 2.2 and Later

The default setting for SRAM is to use it for the critical system variables, stacks and network buffers. In this default mode of operation, the critical system variables will be always be stored in SRAM, and the network buffers will use SRAM if at all possible. If network buffering requirements increase beyond the size of the SRAM, then the system will automatically begin using SDRAM for additional storage. System use of SRAM is only enabled for release builds. Debug builds will automatically disable system SRAM use.

You can specify that your application's variables and task stacks be located in SRAM by declaring them with with the FAST_USER_STK and FAST_USER_VAR macros. If you exceed the amount of SRAM on your system a linker warning will be generated.

To declare a variable in SRAM:

```
BYTE MyByte FAST_USER_VAR;      // MyByte will be located in SRAM
```

To declare a task stack in SRAM:

```
DWORD MyTaskStack[MY_TASK_STACK_SIZE] __attribute__((aligned( 4 ))) FAST_USER_STK;
```

The use of SRAM by the system variables and stacks is defined in \nburn\include\constans.h. By modifying the defnitions in constants.h you can disable the system use of SRAM completely, or control which system tasks are able to use the SRAM. Any stacks or variables that do not use SRAM will be located in SDRAM. If you make any changes to constants.h, you must rebuild the system library for the change to take effect.

```
#ifndef _DEBUG
#define ENABLE_SRAM_SYS // SRAM use is only enabled in release builds
#endif

#ifdef ENABLE_SRAM_SYS

#define FAST_SYSTEM_VARIABLES   // locate critical system variables in SRAM

// Uncommented system tasks will be stored in SRAM, otherwise SDRAM will be used.
//#define FAST_IDLE_STACK
#define FAST_MAIN_STACK
#define FAST_ETHERNET_VARIABLES
#define FAST_ETHERNET_STACK
#define FAST_BUFFERS_VARIABLES
#define FAST_BUFFERS
#define FAST_IP_VARIABLES
#define FAST_IP_STACK
#define FAST_TCP_VARIABLES
#define FAST_TCP_STACK
//#define FAST_HTTP_STACK
//#define FAST_FTP_STACK
//#define FAST_WIFI_STACK
//#define FAST_PPP_STACK
//#define FAST_COMMAND_STACK

/* If these defines are enabled, any user variables or tasks declared with
    FAST_USR_STK or FAST_USR_VAR will be stored in SRAM.
*/
#define FAST_USER_VARIABLES
#define FAST_USER_STACK

#endif
```

18.3 Using SRAM in Tools Release Prior to 2.1 RC6

Declare a structure that contains the variables you want to locate in SRAM, and a pointer to the SRAM base address plus 0x400. The default SRAM base address for NetBurner platforms is 0x20000400. This can be verified by checking the memory map in the platform documents for your NetBurner platform, located in \nburn\docs\platform. An example program for this method is shown below:

```
/*-------------------------------------------------------------------
 SRAM Pointer Example
 -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <string.h>

extern "C"
{
   void UserMain( void * pd);
}

const char * AppName="SRAMPointer";

// SDRAM Base Address, starting after the 0x400 vector table
#define SRAM_BASE (0x20000400)

typedef struct
{
   int I;
   long l;
   char str[100];
} sram_struct ;

sram_struct *pSram = (sram_struct *)SRAM_BASE;  // Set pointer address

void UserMain( void * pd)
{
    InitializeStack();
    if (EthernetIP==0)GetDHCPAddress();
    OSChangePrio( MAIN_PRIO );
    EnableAutoUpdate();

    iprintf( "Application started\r\n" );

    // It is the Programmer's responsibility to ensure the structure fits
    // within the available physical SRAM space.
    pSram->i = 1234;
    strcpy(pSram->str, "Hello World");
    iprintf("pSram->str: %s\r\n", pSram->str);
    iprintf("pSram->i: %d\r\n", pSram->i);

    while ( 1 )
    {
        OSTimeDly( TICKS_PER_SECOND );
    }
}
```

19 Time Functions

Embedded systems have a number of options to set and access time. Your NetBurner kit will enable you to set the time manually, through a real-time clock (if available), and using a network connection and the Network Time Protocol (NTP).

19.1 Standard C Time Functions

The following standard C time structures and functions are supported by the development tools and libraries. To use these functions you will need to include both time.h and nbtime.h in your application.

The `tm` structure is used to store time and date information:

```
typedef struct tm {
   int tm_hour;   // hour (0 - 23)
   int tm_isdst;  // daylight saving time enabled/disabled
   int tm_mday;   // day of month (1 - 31)
   int tm_min;    // minutes (0 - 59)
   int tm_mon;    // month (0 - 11 : 0 = January)
   int tm_sec;    // seconds (0 - 59)
   int tm_wday;   // Day of week (0 - 6 : 0 = Sunday)
   int tm_yday;   // Day of year (0 - 365)
   int tm_year;   // Year less 1900
}
```

The `time_t` variable type is a DWORD value.

The time functions in nbtime.h are:

time_t time(time_t *time)	Get current calendar time as single number of type time_t
set_time(time_t time)	Sets the system time to values in time_t structure

The functions in time.h are:

char *asctime(const struct tm *time)	Convert tm structure to string

ctime( )	Convert time_t value to string
difftime( )	Return difference between two times
gmtime( )	Convert time_t value to tm structure as UTC time
localtime( )	Convert time_t value to tm structure as local time
mktime( )	Convert tm structure to time_t value
strftime( )	Flexible calendar time formatter

20 Dynamic Memroy Allocation and Free Space

The 3 main types of RAM memory available to an embedded system are:

- Statically allocated variables at compile-time.
- Task and system stack space.
- Dynamic memory (used by malloc and new).

Statically allocated memory at compile time is the safest to use, since the system does not need to notify the user or recover from a failure if a dynamic memory allocation call fails. For those cases when dynamic memory allocation must be used, the `spaceleft()` and `mallinfo()` functions can be used to determine the amount of free memory available. The `spaceleft()` function returns the amount of free space between the amount of memory allocated to the heap (the heap pointer) and the top of the system stack. However, there may be free space available inside the heap as well from previous malloc calls that have been freed. The `mallinfo()` function provides information on heap memory blocks that are not in use.

An example program demonstrating the use of these functions is shown below.

```
/* $Revision: 1.0 $ */
/* Copyright $Date: 2009/10/05 22:53:41 $ */
/*-------------------------------------------------------------------
 * This example demonstrates the usage of malloc and free. It
 * identifies way's to track your heap space used by calling
 * spaceleft() and mallinfo().  Use these function to help
 * ensure that applications do not run out of heap space.
 *
 * It is important to use both spaceleft() and mallinfo() to
 * calculate the size of your heap. As this application
 * demonstrates, spaceleft() alone will not always give the
 * total space available to malloc.
 *
 * In this example, the application will allocate 3 chunks
 * of space. The first is 1MB, then a 3MB, then a 512KB
 * chunk. It will then free the data in a different order.
 *
 * After every malloc and free, a heapinfo print will occur.
 * This shows the current heapspace used, heapspace free, and
 * space reported by spaceleft().
 * -------------------------------------------------------------------*/
#include "predef.h"
#include <stdio.h>
#include <ctype.h>
#include <startnet.h>
#include <autoupdate.h>
#include <dhcpclient.h>
#include <smarttrap.h>
#include <NetworkDebug.h>
#include <stdlib.h>
#include <string.h>
#include <bsp.h>
#include <malloc.h>

extern "C" {
    void UserMain(void * pd);
}

const char * AppName = "Malloc Example";

/*-------------------------------------------------------------------
  showHeapSpace
```

```
  Uses the mallinfo() funcation call to return the following structure:

   struct mallinfo {
     int arena;    // total space allocated from system
     int ordblks;  // number of non-inuse chunks
     int smblks;   // unused -- always zero
     int hblks;    // number of mmapped regions
     int hblkhd;   // total space in mmapped regions
     int usmblks;  // unused -- always zero
     int fsmblks;  // unused -- always zero
     int uordblks; // total allocated space
     int fordblks; // total non-inuse space
     int keepcost; // top-most, releasable (via malloc_trim) space
   };

 ------------------------------------------------------------------*/
void showHeapSpace()
{
    struct mallinfo mi;
    mi = mallinfo();

    iprintf("-heapinfo---------------\r\n");
    iprintf("used      : %d\r\n", mi.uordblks);
    iprintf("spaceleft : %d\r\n", spaceleft());
    iprintf("free      : %d\r\n\r\n", spaceleft() + mi.fordblks);
}


/*-------------------------------------------------------------------
 * UserMain
 *-------------------------------------------------------------------*/
void UserMain(void * pd)
{
    InitializeStack();

    if ( EthernetIP == 0 )
    {
       iprintf( "Trying DHCP...\r\n" );
       GetDHCPAddress();
       iprintf( "DHCP assigned the IP address of:  " );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }
    else
    {
       iprintf( "Static IP address:  " );
       ShowIP( EthernetIP );
       iprintf( "\r\n" );
    }

    OSChangePrio(MAIN_PRIO);
    EnableAutoUpdate();

    while (1)
    {
       iprintf("=========================================\r\n\r\n");

       BYTE *pA,*pB,*pC = NULL;  // Pointers to allocated memory blocks

       iprintf("Allocating 1,000,000 bytes (pA)\r\n");
       pA = (BYTE *)malloc(sizeof(BYTE) * 1000000);
       if( pA == NULL )
```

```
            iprintf("malloc failed\r\n");
        showHeapSpace();

        iprintf("Allocating 3,000,000 bytes (pB)\r\n");
        pB = (BYTE *)malloc( sizeof(BYTE) * 3000000 );
        if( pB == NULL )
            iprintf("malloc failed\r\n");
        showHeapSpace();

        iprintf("Allocating 512,000 bytes (pC)\r\n");
        pC = (BYTE *)malloc( sizeof(BYTE) * 512000 );
        if( pC == NULL )
            iprintf("malloc failed\r\n");
        showHeapSpace();

        // Now freeing data in order B,A,C. This illustrates that the spaceleft
        // value can not increase until the last block allocated (pC) is
        // freed.
        if ( pB != NULL )
        {
            iprintf("Free (pB) \r\n");
            free(pB);
            showHeapSpace();
        }

        if ( pA != NULL )
        {
            iprintf("Free (pA) \r\n");
            free(pA);
            showHeapSpace();
        }

        if ( pC != NULL )
        {
            iprintf("Free (pC) \r\n");
            free(pC);
            showHeapSpace();
        }

        OSTimeDly( TICKS_PER_SECOND * 10 );
    }
}
```

21 The Template Program – Command Line Mode

You have two options for code development using the NetBurner Tools. The first is the IDE discussed earlier. The second is using the make utility from a command prompt. Typical reasons for choosing the command prompt mode are that some developers simple prefer this method, or you may have a favorite editor/IDE that you prefer over the NetBurner IDE. Most editors can invoke an external compiler to build the project.

This section will detail how to write, compile, and download a program using the command line tools. The details of the make utility "makefile" will also be covered. Traditionally called the "Hello World" program, our "Template Program" will specify a minimal code base from which you can write your future applications. The objective of this template program is to print the characters "Hello World" out the debug port of your NetBurner device. In addition, this template program will enable network services so that it can be downloaded over a network connection instead of through a serial port or a BDM (Background Debug Mode) port.

The Debug Monitor

The NetBurner device contains a flash memory boot sector loaded with a boot program called the "Debug Monitor". This program is designed to be very small and takes up less than 16 Kbytes of memory space. **Note:** The Debug Monitor is not designed to provide full TCP network communications, although it does support the TFTP protocol.

The full TCP/IP Stack functionality is compiled as part of your application. If you download an application that immediately crashes when it boots, full network services will not be available. In this case, the NetBurner Debug Monitor comes to the rescue. Once in the Debug Monitor (at the NB> prompt), you can download a working application through the serial debug port. See the section on serial downloads using the Debug Monitor for more information.

Before starting to write your application, you must create a project directory. You project directory can be located anywhere on your hard drive. For this example:

1. Create a directory with a path of c:\netburner
2. Create a subdirectory with the path of c:\netburner\template

Now, look at the template program. First, you will need to create a file (using your editor of choice) called "main.cpp" in the c:\netburner\template directory. Simply copy and paste the code below (in section 5.1), and save this file (as main.cpp) in the c:\netburner\template directory.

Using C++ File Extensions

Throughout this manual, we will always use the .cpp extension to indicate the source files are C++. This is done to take advantage of the benefits of the C++ compiler as well as support those programmers using C++. **Note:** You do not need to know any C++ to use the NetBurner Development Kit, and the majority of examples in this guide do not use C++.

21.1 Template Program Source Code

```
1   /*-------------------------------------------------------------------
2    Description: The Template Program
3    Filename: main.cpp
4    -------------------------------------------------------------------*/
5      #include "predef.h"
6      #include <stdio.h>
7      #include <startnet.h>
8      #include <autoupdate.h>
9
10     extern "C" {
11     void UserMain(void * pd);
12     }
13
14     void UserMain(void * pd)
15     {
16        InitializeStack();
17        EnableAutoupdate();
18        OSChangePrio(MAIN_PRIO);
19
20        while (1)
21        {
22           iprintf("Hello World\n");
23           OSTimeDly(TICKS_PER_SECOND * 1);
24        }
25     }
```

The above program is a fully functional network application in just a few lines of code! The only application specific code is on lines 20 and 21; the remainder of the program is what we will refer to as the "Template Program". Although the purpose of our application is to print "Hello World" out the debug serial port, adding the network support will allow fast code development using the NetBurner "make load" build command, and also allow network configuration using the NetBurner IPSetup utility (i.e. IPSetup tool). For additional information on "make load" and AutoUpdate, please refer to your NNDK User Manual. From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.

Lines 5 – 9 specify the include files:

- predef.h defines constants for debugging (covered in the Debugging chapter) and version info
- stdio.h defines standard input and output functions
- startnet.h defines the function calls necessary to start the TCP/IP Stack and HTTP
- autoupdate.h defines the functions necessary to download firmware updates over a network connection

Line 10 tells the C++ compiler to declare the UserMain( ) function as a "C" type function call. **Note:** This is done to allow straight C programming as opposed to C++.

Line 14 declares the UserMain( ) function. The parameter passed to UserMain( ) is a void pointer to some type of data. This is a feature of the uC/OS RTOS. **Note:** This example does not use the passed parameter.

Line 16 initializes the TCP/IP stack.

Line 17 enables the code development AutoUpdate feature using "make load"

Line 18 sets the task priority to the defined value of MAIN_PRIO, which defaults to 50. More information on the RTOS will be covered in the RTOS section of your NNDK User Manual .(From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.)

Lines 20 - 24 create an infinite loop that prints the message "Hello World" to stdout (which defaults to the debug port of your NetBurner device) once per second.

Line 21 is a while loop that loops forever. **Note:** Your application should never return from UserMain( ); that would mean your application has lost control of the system.

21.2 Compiling and Running the Application - Overview

Now that we have the application source code file, we need to compile it into a code image and download it to your NetBurner device. There are four methods to download your applications:

1. Through the serial port
2. Through a network connection using AutoUpdate (preferred method)
3. Through a network connection using TFTP
4. Through a network connection using FTP

In this example, we will use the AutoUpdate method. In order to run your application on your NetBurner device you will need to do the following:

1. Create a "makefile", which is used to compile and link your source code
2. Download your code to your NetBurner device
3. Burn this code into the Flash memory
4. Reboot your NetBurner device

Thankfully, this is a very simple process that can be accomplished with a single build command. The first step is to create a makefile, which is used to tell the compiler how to compile the source code. Once we create the first makefile, it can be used as a template for your future projects. For additional information, please refer to your User Manual. From Windows: Start → Programs → Netburner NNDK → NNDK Users Manual.

21.3 Creating a makefile

Below is the makefile for the template example application. The file name is "makefile" with no extension. Simply copy and paste the code below (using your editor of choice), and save this file (as makefile) in the c:\netburner\template directory. **Note:** The makefile should always be located in the same directory as your application source code. The makefile below will work for applications with or without HTML support as described in the comments. **Note:** A comment in C++ is signified by the '#' character at the beginning of a line.

```
1   #Build NAME.x and save it as $(NBROOT)/bin/NAME.x
2   NAME  =  template
3   CXXSRCS := main.cpp
4
5   #Uncomment and modify these lines if you have C or S files.
6   #CSRCS := foo.c
7   #ASRCS := foo.s
8   #CREATEDTARGS := htmldata.cpp
9
10  #Include the file that does all of the automagic work!
11  include $(NBROOT)/make/main.mak
12
13  #htmldata.cpp : $(wildcard html/*.*)
14  #comphtml html -ohtmldata.cpp
```

Once you have created a single makefile, you can copy it to the project directories of any applications you create in the future. **Note:** If you application does not use HTML, the only lines you will need to change will be the name of the application (line 2) and the list of source code files (line 3). If your application uses the Web Server and HTML, simply uncomment lines 8, 13 and 14 - you do not need to modify these lines.

Line 2 specifies the name of the application image file that will be built. Ours is called “template”. **Note:** there is not a semicolon after the ‘=’ here.

Line 3 specifies the C++ source code files that need to be compiled. We only have one file in this example. If your project has more than one source code file, you would just add them to this line separated by spaces.

Lines 6 and 7 specify C and S (assembly language) source files.

Line 8 is used for applications that use HTML and the web server. You do not need to change the file name.

Line 11 calls additional make files that are part of your NetBurner tool set.

Lines 13 and 14 are used to process code used by the Web Server such as HTML. **Note:** Do not uncomment these lines unless you have a directory named “html” under your project directory.

21.4 Compiling the Application

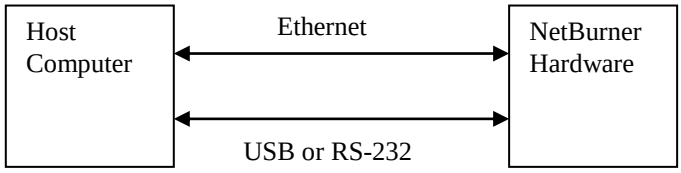
At this point you should have a directory called c:\netburner\template that contains two files: “main.cpp” and “makefile”. First, we will run “make” to verify the application compiles correctly. Once all errors have been corrected, we will download this application to your NetBurner device. To compile the source code and create the application image:

1. If you have not done so already, open a command prompt and move to the c:\netburner\template directory (cd c:\netburner\template). If you did not select the automatic variable configuration during the installation process, you must run c:\nburn\setenv.bat manually. A simple method to determine if your system is configured correctly is to type “set” at the command prompt (then press the "Enter" key). Look for an environment variable called “NBROOT”. If NBROOT is defined, then you should be ready to go. If NBROOT is not defined, just type "setenv" at the command prompt (in the c:\nburn directory) and press the "Enter" key. Now, if you type “set” at the command prompt (and press the "Enter" key), you will see that NBROOT is defined (i.e. NBROOT=/nburn).
2. At the command prompt (in the c:\netburner\template directory), type “make” and press the "Enter" key. This will invoke the compiler to compile and link your code, and build the code image. If you have any coding or syntax errors, correct them now and continue to run make until your code compiles correctly.

When your code compiles correctly, two files are created: template.s19 and template_APP.s19. The template.s19 file is memory mapped to run from RAM, while template_APP.s19 is memory mapped to run from Flash memory. **Note:** All compiled images will be located in the c:\nburn\bin directory. This guide will focus on Flash downloads. Please refer to the section on Downloading to RAM in your User Manual for more information on downloading applications to RAM. (From Windows: Start → Programs → NetBurner NNDK → NNDK Users Manual.)

21.5 Template Program Setup

Before running our program, verify your hardware is set up correctly. To run the Template program, you will need your hardware to be set up as shown below:



The Ethernet connection should be between your host computer and your NetBurner device's RJ-45 connector. The USB or RS-232 connection should be made between your host computer's Serial port and the Debug Serial port of your NetBurner device. The Serial port connection on the NetBurner device will vary with each hardware platform, but it should be a DB9 connector on the processor board itself, or on a separate Adapter board or Carrier board supplied with your kit. Please refer to your Quick Start Guide for additional details on how to "make" and download files to your NetBurner device.

21.5.1 Testing the RS-232 Debug Connection

You can determine if you are properly connected to the debug port with the following test:

1. Start the dumb terminal program MTTTY, which is included in your NetBurner tools. (From Windows: Start → Programs → NetBurner NNDK → Mttty Serial Terminal. From the command line type: \Nburn\pcbin\mttty and then press the "Enter" key.) Set the baud rate to 115,200 and make sure to click on the "Connect" button in the MTTTY window.
2. Power on or reset your NetBurner device. The MTTTY screen should display a sign-on message similar to "Waiting to boot……". If you see this message, then you are connected correctly.

For additional information, please refer to your User Manual. From Windows: Start → Programs → NetBurner NNDK → NNDK Users Manual.

22 Software Licensing

The Software included in this development kit is licensed to run only on NetBurner provided hardware. If your application involves manufacturing your own hardware, please contact NetBurner Sales for details on a royalty-free software license. The following sections describe the agreements presented during installation of the development kit.

22.1 NetBurner License Agreement

Notice to Developer: This is a contract. By installing the NetBurner Tools you accept all the terms and conditions of the "NetBurner Tools Software" and "NetBurner Embedded Software" Agreements. If you do not agree with the terms and conditions of these agreements, return this development kit and all components to NetBurner.

1. All embedded software and source code provided in this Network development kit is subject to one of four possible licenses:

    - The NetBurner Tools License (most restrictive)
    - The NetBurner Embedded Software License
    - The GNU Public License
    - The Newlib License (least restrictive)

2. The GNU development executables provided in the nburn\GCC-M68k directory branch are subject to the GNU public license. This license can be found in nburn\docs\GNULicense.txt file.

3. The runtime libraries and include files provided in the nburn\GCC-M68k directory branch are subject to the Newlib license. This license can be found in nburn\docs\NetlibLicense.txt file.

4. The compcode application provided in the nburn\pctools\compcode directory is subject to the GNU public license. This license can be found in the nburn\docs\GNULicense.txt file.

5. The other programs in the \nburn\pctools directory are subject to the NetBurner Tools License provided below.

6. All other provided source code and libraries are subject to the NetBurner Embedded Software License provided below.

22.2 The NetBurner Tools Software License

Permission is hereby granted to purchasers of the NetBurner Network Development Kit to use these programs on one computer, and only to support the development of embedded applications that will run on **NetBurner provided hardware**. No other rights to use this program or its derivatives, in part or in whole, are granted. It may be possible to license this or other NetBurner software for use on non-NetBurner hardware. Please contact sales@netburner.com for more information.

NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

22.3 The NetBurner Embedded Software License

Permission is hereby granted to purchasers of NetBurner hardware to use or modify this computer program for any use as long as the resultant program is **only executed on NetBurner provided hardware**. No other rights to use this program or its derivatives, in part or in whole, are granted. It may be possible to license this or other NetBurner software for use on non-NetBurner hardware. Please contact sales@netnurner.com for more information.

NetBurner makes no representation or warranties with respect to the performance of this computer program, and specifically disclaims any responsibility for any damages, special or consequential, connected with the use of this program.

23 NetBurner Contact Information

Main Web Site:
www.netburner.com

E-Mail:
sales@netburner.com
support@netburner.com (requires active support account)

Support and Product Registration Web Site:
http://support.netburner.com

Free customer forum:
Please refer to the Support section of www.netburner.com.

24 NetBurner Support Information

Each NetBurner Network Development kit includes 6 months of email support. To access support resources and to register your product, please go to http://support.netburner.com, or you can send an E-Mail to support@netburner.com. **Note:** You must register your product before you can request support. You will need to include the software release number, the hardware Ethernet address and the hardware revision number of your NetBurner board. The NetBurner web site is a good source for answers to frequently asked questions and design examples. Annual support and software maintenance agreements can also be purchased on the NetBurner web site at http://www.netburner.com.