SBL2e Network Programming Guide

Table of Contents

1. Introduction

This document is a reference manual for the NetBurner SBL2e network libraries. The SBL2e network libraries are designed to run in a very small amount of RAM, as low as 32k bytes. The most efficient use of memory is achieved through the use of function "callbacks". A callback function is a function that is called through a function pointer. These are useful in small memory environments because it enables the TCP/IP stack to call your network function directly and avoid having copies of the data stored in additional memory buffers.

All NetBurner documents are located in the documents directory created during installation. The default location is c:\nburn\docs.

Hardware-specific software functions and information are provided in the c:\nburn\docs\SBL2e directory. The platform documents contain schematics, memory maps, and any software features that are specific to the hardware platform you are using.

The software included in your NetBurner Development Kit is licensed to run only on processor hardware manufactured by NetBurner, such as the modules and serial to Ethernet devices. If your application involves manufacturing your own processor based hardware (ie you are not going to purchase NetBurner modules for production), please contact NetBurner Sales for details on a Royalty-Free Software License.

1.1 Additional Documentation

All NetBurner documentation is located by default in your C:\nburn\docs directory.

- Eclipse Getting Started Guide
- NetBurner PC Tools Guide
- uC/OS Library Reference manual
- Freescale microprocessor manuals
- Platform Documents – The hardware specific documents for your device

1.2 Example Programs

There are many example programs located in the c:\nburn\examples directory. Some of these examples are referenced in this guide. Please refer to the examples directory for the lasted version of any source code you wish to use in your application.

2. System Overview

2.1 Network Protocols

The SBL2e supports the following protocols:

- DHCP
- DNS
- HTTP
- ICMP
- Ping
- SMTP
- TCP
- UDP

2.2 uC/OS Real Time Operating System

The SBL2e uses the uC/OS RTOS. Please refer to the c:\nburn\docs\NetBurnerRuntimeLibriary directory for the uC/OS reference documentation. There are also example programs located in the c:\nburn\examples\rtos directory.

2.3 Tasks

The maximum number of tasks is 32, numbered from 0 to 31 with 31 being the lowest priority task which is reserved as the Idle task. Because each task takes up RAM space, the maximum number of tasks is set to a lower number in the \nburn\include_sc\constants.h file. This total includes the system tasks, of which there are typically 4.

#define OS_MAX_TASKS 10

When you create a task, be sure to check the return value for error conditions such as if the priority is already in use, or if you have exceeded the maximum number of tasks.

The system creates the following tasks:

Task	Priority	Description
Idle	31	The Idle task runs when all higher priority tasks are blocking
UserMain	5/25	The first task created for your application. You may create additional tasks from UserMain if they are needed. The UserMain priority is set at 5 at boot, then moved to 25.
HTTP	15	The web server task
IP	10	Handles all IP, UDP, TCP and DHCP processing. If an application uses network callback functions, they are executed at the IP task level.

The definition `#define MAIN_PRIO (25)` can be use as a reference in your application when creating additional tasks. For example, a task that is of a higher priority can be MAIN_PRIO-1.

The idle task stack size is 128 bytes. All other tasks are 786 unsigned longs (3072 bytes) and use the standard task stack size definition:

```
#define USER_TASK_STK_SIZE (786) // Bytes = 4 times this number
```

If you create your own user tasks you can specify whatever task size you wish as long as there is enough ram. A minimum size of 512 is recommended. Among other things, the task stack is used for allocation of local variables – those variables declared inside a function call. The total number of all local variable bytes must be able to fit within the task stack or stack corruption will occur. The total runtime number can be difficult to determine because functions may call other functions resulting in a larger stack space requirement. The "static" keyword can be used when declaring a large variable like a buffer inside a function, which will use global memory space rather than task stack memory space.

2.4 System Timer

The system timer runs at 20 ticks per second. The following definition can be used in applications to specify delay or wait times:

```
#define TICKS_PER_SECOND (20)
```

For example

```
OSTimeDly( TICKS_PER_SECOND * 5 );  // delay 5 seconds
```

2.5 TCP Sockets

The maximum number of TCP sockets is 10, divided into 6 possible active TCP connections and up to 4 listening sockets.

2.6 Interrupts

Processor interrupts can be shared and have both an IRQ number and priority level. Please refer to the Freescale manual for further details on processor interrupts. The higher the IRQ number, the higher the priority. Level 7 is the highest priority and is non-maskable.

2.6.1 SBL2e Reserved Interrupts

Use	Number	Priorities
GDB	7	
Serail/UART	3	1, 2 and 3.
Ethernet (FEC)	2	3, 4, 5, and 6.
System Timer	1	3. The system timer uses PIT0.

3. DHCP

The Dynamic Host Configuration Protocol (DHCP) support is automatically invoked by the stack initialization function `InitializeStack()` when the static IP address setting is 0.0.0.0. Setting the static IP address to any value other than 0.0.0.0 will disable DHCP.

The global status variables are commonly used with DHCP:

```
BOOL    bEtherLink
IPADDR MyIpAddr;
IPADDR MyIpMask;
IPADDR MyIpGateway;
IPADDR MyIpDNS;
```

The DHCP address cannot be assigned before the bEtherLink variable is TRUE. Most examples have a short test loop to ensure link is established before checking for DHCP assigned values.

The IPADDR variables are valid for DHCP or static assigned values, and can be read at any time. IPADDR is an unsigned long 32-bit value. To display the information in the common dotted decimal notation you can use iprintf( ) or siprintf( ) with the "%I" option. For example:

```
/*-------------------------------------------------------------------
 * Display System information
 *-------------------------------------------------------------------*/
void DisplaySystemInformation()
{
   iprintf("\r\n--- System Information ---\r\n");
   iprintf("IP Address: %I\r\n", MyIpAddr );
   iprintf("IP Mask:    %I\r\n", MyIpMask );
   iprintf("IP Gateway: %I\r\n", MyIpGateway );
   iprintf("IP DNS:     %I\r\n", MyIpDNS );
   iprintf("\r\n\r\n");
}
```

Please see the SBL2e example folder in your NetBurner tools distribution for the latest example code.

4. DNS

4.1 Function Summary

Include Files
#include <dns.h>

Functions

`GetHostByName()`	Returns the IP address for the specified host name

4.2 GetHostByName

Synopsis:

```
int GetHostByName( const char *name, IPADDR *pIPaddr,
                   IPADDR DnsServer, WORD timeout,
                   WORD type = DNS_A );
```

Description:
Uses the Dynamic Name Service protocol to obtain the IP address associated with a given DNS name (eg www.yahoo.com).

Parameters:

Type	Name	Description
const char *	name	Pointer to a null terminated ASCII string holding the DNS name, such as “www.yahoo.com”.
IPADDR *	pIPaddr	Point to a variable of type IPADDR. If successful, the IP address of the DNS name will be written to this location.
IPADDR	DnsServer	IP address of the DNS server to contact for the request. The network configuration settings for your device must have a valid DNS IP address and Internet Gateway IP address for DNS to succeed. These values are normally assigned if you are using DHCP. If you are specify a static IP address you will need to set these values manually as well. A DnsServer value of 0 will use the active system DNS IP address.
WORD	timeout	Number of system time ticks to wait for a DNS reply.
WORD	type	DNS record type per RFC 1035. Default value is DNS_A. Possible values: DNS_A, DNS_CNAME, DNS_MB, DNS_MG, DNS_MX

Returns:

DNS_OK (0)
DNS_TIMEOUT (1)
DNS_ERR (3)

Example:

```
IPADDR IPaddr = 0;

int rv = GetHostByName( “www.yahoo.com”, &IPaddr, 0, TICKS_PER_SECOND * 10 );
if ( rv == DNS_OK )
   iprintf("IP address: %I\r\n", IPaddr );
else
   iprintf("DNS error: %d\r\n", rv );
```

5. HTTP

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.

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:

- Add a directory named "html" in your project directory.
- Create a web page called index.htm in the html directory.
- Add the StartHTTP() function call to start the Web Server. This call is usually located after the InitializeStack() function in the UserMain() task.

5.1 Function Summary

Include Files
#include <http.h>

Functions

`StartHTTP()`	Start HTTP Server
`RedirectResponse()`	Redirect client web browser to a new page
`NotFoundResponse()`	Send HTTP Page Not Found response

5.2 StartHTTP

Synopsis:

```
void StartHTTP( WORD port = 80 )
```

Description:

Starts the HTTP Web Server. Must be called after InitializeStack(). The default port of 80 will be used unless a different port number is specified. Once the HTTP server has been started it cannot be stopped.

Parameters:

Type	Name	Description
WORD	port	The listen port number of the HTTP server. The default is 80.

Returns:

Nothing

5.3 RedirectResponse

Synopsis:

```
void RedirectResponse( int sock, PCSTR new_page );
```

Description:
Send a response to the active socket to redirect the client web browser to a new page.

Parameters:

Type	Name	Description
int	sock	The active TCP socket.
PCSTR	new_page	Pointer to a string that represents the new URL.

Returns:

Nothing

5.4 NotFoundResponse

Synopsis:

```
void NotFoundResponse( int sock );
```

Description:
Send a response that indicates the page can't be found.

Parameters:

Type	Name	Description
int	sock	The active TCP socket.

Returns:

Nothing

5.5 HTTP/TCP Write Functions

The functions in this section are designed to be used in dynamic web content applications to write formatted data. During the web server processing the these functions are called from the web server task; unpredictable results can occur if your application code calls them from any other task.

5.5.1.1 tcp_printf

Synopsis:

```
BOOL tcp_printf(int sock, const char * format,...)
```

Description:
Writes to the specified TCP socket using printf() style formatting. This version supports floating point numbers.

Parameters:

Type	Name	Description
int	sock	Active TCP socket.
const char *	format	Variable parameter printf() format stirng.

Returns:
TRUE on success
FALSE if socket does not exist

Example:

```
tcp_printf( TcpClientSocket, "Floating point value: %f\r\n", fValue );
```

5.5.1.2 tcp_iprintf

Synopsis:

```
BOOL tcp_iprintf(int sock, const char * format,...)
```

Description:
Writes to the specified TCP socket using printf() style formatting. This version does not support floating point numbers, but uses less memory.

Parameters:

Type	Name	Description
int	sock	Active TCP socket.
const char *	format	Variable parameter printf() format stirng.

Returns:
TRUE on success
FALSE if socket does not exist

Example:

```
tcp_iprintf( TcpClientSocket, "Anything but floats!, %ld\r\n", TimeTicks );
```

6. 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. As each tag is processed it will substitue the associated variable, or call the associated function which will pass control to the application so it can write to the socket directly.

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

An example web page from the HTMLVariables example program is shown on the next page. Although this is quite a bit of code it can be helpful to see how the tags are used before going into the details. Please see the example source code in the \nburn\examples\SBL2e\HTMLVariables for the latest updates.

6.1 Web Browser view of index.htm:

HTML Variable Example

This example program uses the FUNCTIONCALL and VARIABLE tags to display variables and use parameters in function calls.

Display System Configuration Variables:

Description	Value
IP Address	10.1.1.202
IP Mask	255.255.255.0
IP Gateway	10.1.1.1
IP DNS Server	66.75.160.15

```
Boot Baud Rate: 115200
Boot Delay: 2seconds
System tick count: 9seconds
```

HTML Source Code:
The first line of the table and an equation are shown below. The IPCAST() macro is used to convert a 32-bit value into an IP address format.

```
   Display a single variable:
   <td>IP Address</td>
   <td> <!--VARIABLE IPCAST(MyIpAddr) --> </td>

   Display the result of an equation:
   Tick count is : <!--VARIABLE TimeTick/TICKS_PER_SECOND --> seconds
```

Function Call

The FUNCTIONCALL tag passes the socket fd and URL to a 'C' function in your application code:
Message from FooFunction(): No parameters were supplied after '?' in the URL

HTML Source Code:

```
<!--FUNCTIONCALL FooFunction -->
```

Function Call with Parameters

The VARIABLE tag can be used to call a 'C++' function with multiple parameters.
The example below shows 2 parameters:
Message from FooWithParameters(): fd = 8, v = 123

HTML Source Code:

```
<!--VARIABLE FooWithParameters( fd, 123 ) -->
```

6.2 The HTML source code for index.htm:

```
<HTML>
<BODY>
<h1>HTML Variable Example</h1>
This example program uses the FUNCTIONCALL and VARIABLE tags to display variables
and use parameters in function calls.
<br>
<h2>Display System Configuration Variables:</h2>
<pre>
<table border=2 cellpadding=5>
<tr>
   <td>Description</td>
   <td>Value</td>
</tr>
<tr>
   <td>IP Address</td>
   <td> <!--VARIABLE IPCAST(MyIpAddr) --> </td>
</tr>
<tr>
   <td>IP Mask</td>
   <td> <!--VARIABLE IPCAST(MyIpMask) --> </td>
</tr>
<tr>
   <td>IP Gateway</td>
   <td> <!--VARIABLE IPCAST(MyIpGateway) --> </td>
</tr>
<tr>
   <td>IP DNS Server</td>
   <td> <!--VARIABLE IPCAST(MyIpDNS) --> </td>
</tr>
</table>

Boot Baud Rate: <!--VARIABLE gConfigRec.baud_rate -->
Boot Delay: <!--VARIABLE gConfigRec.wait_seconds --> seconds
System tick count: <!--VARIABLE TimeTick/TICKS_PER_SECOND --> seconds
</pre>

HTML Source Code:<br>
The first line of the table and an equation are shown below. The
IPCAST() macro is used to convert a 32-bit value into an IP
address format.
<pre>
   Display a single variable:
   &lt;td>IP Address&lt;/td&gt;
   &lt;td&gt; &lt;!--VARIABLE IPCAST(MyIpAddr) --&gt; &lt;/td&gt;

   Display the result of an equation:
   Tick count is : &lt;!--VARIABLE TimeTick/TICKS_PER_SECOND --&gt; seconds
</pre>

<h2>Function Call</h2>
The FUNCTIONCALL tag passes the socket fd and URL to a 'C' function in your
application code:<br>
<!--FUNCTIONCALL FooFunction --> <br>
```

```
<br>
HTML Source Code:<br>
<pre>
&lt;!--FUNCTIONCALL FooFunction --&gt;
</pre>

<h2>Function Call with Parameters</h2>
The VARIABLE tag can be used to call a 'C++' function with multiple
parameters.<br>
The example below shows 2 parameters:<br>
<!--VARIABLE FooWithParameters(fd,123) --> <br>

<br>
HTML Source Code:<br>
<pre>
&lt;!--VARIABLE FooWithParameters( fd, 123 ) --&gt;
</pre>
</BODY>
</HTML>
```

6.3 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.

6.4 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.

6.5 Writing to a Web Browser From Within a Function: tcp_printf()

When the web server invokes your function, your function is passed a socket file desciptor and must quickly write whatever data you want to send because the web browser is waiting for the response. This is true for both the FUNCTIONCALL tag, and the VARIABLE tag when used to call a function with parameters.

The functions availale to write the data to the web broswer are:

```
BOOL tcp_printf(int sock, const char * format,...)
BOOL tcp_iprintf(int sock, const char * format,...)
```

These function work like a standard C printf() with formatting parameters. The ‘i' version does not have floating point capability. The functions are located in tcp.h.

6.6 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) !-->
```

6.7 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.

Example of 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 using the INCLUDE tag for file with a name other than htmlvar.h:

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

6.8 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 * FooWithParameters(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 FooWithParameters(fd,123) -->
```

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

```
WriteHtmlVariable( fd, FooWithParameters(fd,123) );
```

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 * FooWithParameters(int fd, int v)
{
    tcp_iprintf( fd, "Message from FooWithParameters(): fd = %d,
                 v = %d\r\n", fd, v );
    return "\0"; // Return a const char * here of zero length so it
                 // wont print anything.
}
```

6.9 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++)
      tcp_iprintf( fd, "%02X:", lpb[i] );
   tcp_iprintf( fd, "%02X", lpb[6] );
}
```

7. Web Form Posts

A HTML Forms are used to present a web page interface that contains "controls" (text boxes, check boxes, radio buttons, etc..) that can be modified by the user viewing the page. The user then clicks on a "submit" button that sends the form data to the NetBurner web server for processing by your application. The submission mechanism is referred to as a form POST. This section will assume you are familiar with HTML and HTML form programming tags.

To create an HTML Form:

1. Create a web page that includes a form. There are two very important tags involved in forms: The <form> tag must specify the name of your form with the **action** parameter, and the **method** parameter must be "post". It is convention to use a web page type name (i.e. formpost.htm) for the action, even though that web page does not exist as a file.
   For example: `<form action="formpost.htm" method=post>`

   One <input> tag must be specified to handle the form submission, with the **type** parameter set to "submit". The **value** parameter is the text label that shows up inside the submit button.
   For example: `<input type="submit" value="Submit Changes">`.

2. Create a function in your application to process the form data received from the web browser post. The function name can be any name of your choosing, but must match the name you use when instantiating the global C++ object. You can create a unique processing function for each form in your application, or you can use a single processing function to handle one or more forms by specifying the same processing function name when you instantiate the C++ objects.
   The format must be as follows:

```
void ProcessFormPost( process_post_action action, int fd,
                      const char * var_name, const char * varvalue )
```

3. Declare the C++ object for your post handler (this calls the C++ constructor). Don't worry, you will not need to know any C++ beyond this simple declaration. The declaration must be called with the form name and form processing function for each form in your application/web page. In this example the form name is called "formpost.htm", and the function we created to process the form data is `ProcessFormPost()`. The third parameter enables a password. The name of the object does not matter (in this case it is `MyFormPost()`) , but each object name must be unique. The examples below show two forms using the same processing function, and one form using a different processing function:

```
PostHandler MyFormPost1( "formpost1.htm", ProcessFormPost, 1 );
PostHandler MyFormPost2( "formpost2.htm", ProcessFormPost, 1 );
PostHandler MyFormPost3( "formpost3.htm", ProcessFormUserData, 1 );
```

7.1 Processing Form Post Data

The key to form post data processing is the function created to handle the incoming data. The function you create will be called in real time to process the form data one variable at a time. This method is used due to the limited amount of ram available to buffer incoming data. As described earlier, the format of the function is:

```
void ProcessFormPost( process_post_action action, int fd, const char * var_name,
                      const char * var_value )
```

where

action:	Post action: eStarting, eVariable or eFinished
fd:	File descriptor for the active TCP socket
var_name:	Form variable name
var_value:	Form variable value

You must have one form post processing function for each form in your application.

When a user clicks on the web page form submit button the following sequence occurs:

1. The client web browser requests a web page containing a HTML form
2. The SBL2e web server calls the appropriate form processing function matching the form name (e.g. “formpost.htm”) with post_action = eStarting. The eStarting phase occurs before any form data is sent from the client. This is an opportunity to initialize variables, check passwords, etc.
3. The processing function will then be called for each variable in the form, with post_action = eVariable. The processing function must parse the variable name and variable value, and take any appropriate action.
4. When all eVariable actions are complete, the processing function will be called with post_action = eFinished.

7.2 Form Post Example

The following example is located in the \nburn\examples\SBL2e directory. Please refer to the example source code for the most recent version of this code.

The example consists of three main parts:

1. The main.cpp file which initializes the SBL2e.
2. The web.cpp file that includes the form post processing function and other HTML function calls.
3. The index.htm web page containing the form.

7.2.1 The main.cpp source code file

The UserMain( ) function is shown for completeness, but there isn't anything specific to form posting in main.cpp.

```
void UserMain(void *pd)
{
    SimpleUart( SERIAL_DEBUG_PORT, SystemBaud ); // Initialize debug UART
    assign_stdio( SERIAL_DEBUG_PORT );           // Use UART 0 for STDIO

    InitializeStack();           // Initialize TCP stack
    EnableAutoUpdate();          // Enable network downloads to target
    OSChangePrio( MAIN_PRIO );   // set standard UserMain task priority

    WaitForDhcpAddressAssignment();
    StartHTTP();

    DisplaySystemInformation();
    DisplayMenu();
    while ( 1 )
    {
        if ( charavail( SERIAL_DEBUG_PORT ) )
        {
            ProcessSerialDebugCommand();
        }
    }
}
```

7.2.2 The index.htm source code file

The example program enables a user to post a name and message that will be stored in the on-chip flash of the SBL2e in the User Parameter area. The first section of the page displays the current values, and the form enables those values to be modified.

```
<html>
<body>
<img src="netburner-logo.gif">
<h1><font face="arial">HTML Form Post and Flash Storage Example</font></h1>
This example demonstrates web form post processing and user parameter flash memory
storage. <br><br>

<b><font face="arial">Current values stored in User Parameter Flash:</font></b>
<br>
Name: <!--FUNCTIONCALL WebMessageName -->
<br>
Message: <!--FUNCTIONCALL WebMessageBody -->
<br><br><hr>

<form action="formpost.htm" method=post>
<b><font face="arial">Modify Flash Values:</font></b><br><br>
Name: <input type="text" name="name" value="<!--FUNCTIONCALL WebMessageName --> "
size=30>
<br><br>
Message: <input type="text" name="message" value="<!--FUNCTIONCALL WebMessageBody
--> " size=80>
<br><br>

<input type="submit" value="Change The Stored Message">
</form>
</body>
</html>
```

7.2.3 The web.cpp source code file

The real work is done by the processing function in web.cpp, which is called when a user clicks on the submit button on the web page.

```
/*-------------------------------------------------------------------
 * This example demonstrates how to:
 * - Store and retrieve data to/from the User Parameter Flash memory
 * - Implement a HTML FORM POST handler
 *-------------------------------------------------------------------*/
#include "predef.h"
#include <basictypes.h>         // Include for variable types
#include <constants.h>          // Include for constants like MAIN_PRIO
#include <system.h>             // Include for system functions
#include <string.h>
#include <stdlib.h>
#include <tcp.h>
#include <http.h>

extern "C"
{
   void WebMessageName( int sock, PCSTR url );
   void WebMessageBody( int sock, PCSTR url );
}

// Structure for storing and retrieving from User Parameter Flash memory space
#define SIZEOF_NAME (40)              // max name length in bytes
#define SIZEOF_MSG (128)          // max message length in bytes
#define VERIFY_VALUE (0x48666012) // Random value that should change if the
structure changes

// Structure that will be read/written to user parameter flash
struct MyOwnDataStore
{
      DWORD verify_key;
      char name[SIZEOF_NAME];
      char msg[SIZEOF_MSG];
};

MyOwnDataStore *pData;



/*-------------------------------------------------------------------
 * Web function called to display the stored name
 *-------------------------------------------------------------------*/
void WebMessageName(int sock, PCSTR url)
{
      // Read the stored data
      pData = (MyOwnDataStore *) GetUserParameters();
      //iprintf( "pdata = %p, key = %08X\r\n", pData, pData->verify_key );


      // Verify it has the right key value.
      if ( pData->verify_key == VERIFY_VALUE )
            tcp_printf( sock, "%s", pData->name );
```

```
	else
		tcp_printf( sock, "No stored name" );
}

/*---------------------------------------------------------------------
 * Web function called to display the stored name
 *---------------------------------------------------------------------*/
void WebMessageBody(int sock, PCSTR url)
{
	//Read the stored data
	pData = (MyOwnDataStore *) GetUserParameters();

	if (pData->verify_key == VERIFY_VALUE)
		tcp_printf(sock, "%s", pData->msg);
	else
		tcp_printf(sock, "No stored message");
}


static MyOwnDataStore my_dataset;

/*---------------------------------------------------------------------------
 * ProcessFormPost
 * A function to process a FORM POST can be unique for each form, or shared
 * between forms.
 * Forms are processed as follows:
 * - When a web browser requests a web page containing a FORM, the
 *   web server will call this function with post_action = eStarting before
 *   any content is sent from the client. This is an opportunity to
 *   initialize variables, check passwords, etc.
 * - The function will then be called for each variable in the FORM,
 *   with post_action = eVariable. This part of the function must parse
 *   the variable name and variable value.
 * - When all eVariable actions are complete, the function will be called
 *   with post_action = eFinished.
 *---------------------------------------------------------------------------*/
void ProcessFormPost(process_post_action action, int fd, const char * var_name,
                     const char * varvalue)
{
	if (varvalue)
	{
	  // Converts a HTML POST URL parameter string to a regular C string
		ConvertPostStringInPlace( (char *) varvalue );
	}

	switch (action)
	{
		case eStarting:
		// Initialize variables
			my_dataset.name[0] = 0;
			my_dataset.msg[0] = 0;
			break;

		case eVariable:
			// The two variables we are looking for are "name" and "message"
			if ((var_name[0] == 'n') && (var_name[1] == 'a'))
			{
```

```
                        strncpy(my_dataset.name, varvalue, SIZEOF_NAME);
                    }
                    else if ((var_name[0] == 'm') && (var_name[1] == 'e'))
                    {
                        strncpy(my_dataset.msg, varvalue, SIZEOF_MSG);
                    }
                    break;

            case eFinished:
                    // Done reading variables, so do something with them
                    my_dataset.verify_key = VERIFY_VALUE;
                    SaveUserParameters((BYTE *) &my_dataset, sizeof(my_dataset));
                    RedirectResponse(fd, "INDEX.HTM");
                    break;
        }
}

/*---------------------------------------------------------------------
* C++ declaration to create the post handler object. The declaration
 * must be called with the form name and form processing function for
 * each form in your application, even if you are sharing the processing
 * function between forms. In this case the form name is called
 * formpost.htm, and the procesing function we created above is ProcessFormPost.
 * ---------------------------------------------------------------------*/
PostHandler MyFormPost("formpost.htm", ProcessFormPost, 1);
```

8. Web Page Passwords

The web pages of an application can be password protected with a single password, or with multiple passwords by using the password group feature. For example, the index.htm page could require no password, a status page could have a viewer password, and the configuration page could require an administrator password.

To enable web page password protection:

1. Include the PASSWORDGROUP tag in each of your HTML web pages. For example, `<!--PASSWORDGROUP 1 -->`. A group value of 0 means the page does not have a password, a value of 1 means it is in group 1, etc. All pages with the same group number will have the same password. The password group tag can appear anywhere in your HTML page, but it is usually the first line in the file. A password group tag is not required if you do not want to password protect the page.
2. Create a function in your application to verify the passwords. Whenever a web browser requests a web page that has a password group tag, the system will call your function. The function must have the format: `int MyCheckPassword(int group, const char *puser, const char *ppass),` where the function name (e.g. MyCheckPassword) can be any name you choose.
3. In the applications UserMain( ) function assign the pointer value `pHttpPassFunc` to your password checking function. For example, `pHttpPassFunc = MyCheckPassword;`

8.1 Example HTTP Password Program

8.1.1 HTML Web Pages

This example contains three web pages. The index.htm file has no password, page1.htm has a group 1 password, and page2.htm has a group 2 password. This example is located in the \nburn\examples\SBL2e\HttpPassword directory. The key element is the PASSWORD group tag.

Source code: index.htm

```
<!--PASSWORDGROUP 0 -->
<html>
<body>
<table>
  <tr>
  <td><img src="netburner-logo.gif"></td>
  <td width=30></td>
```

*** rest of HTML code follows ****

Source code: page1.htm

```
<!--PASSWORDGROUP 1 -->
<html>
<body>
<table>
  <tr>
  <td><img src="netburner-logo.gif"></td>
  <td width=30></td>
```

*** rest of HTML code follows ****

Source code: page2.htm

```
<!--PASSWORDGROUP 1 -->
<html>
<body>
<table>
  <tr>
  <td><img src="netburner-logo.gif"></td>
  <td width=30></td>
```

*** rest of HTML code follows ****

8.1.2 Password Check Function Example

The system will call your password check function with the parameters shown below, and your function must return an integer. The system passes in the HTML page PASSWORD group number and the user name and password entered by the user in the client web browser. The password check function then uses these three parameters to determine whether or not the username and password are valid.

In this simple example, the user name and passwords for two group levels are hard coded to user/pass for group 1 and top/secret for group 2.

```
/*-------------------------------------------------------------------
 * Function to process the password check. A return value of 0 means
 * the password/username are not valid. A value of 1 means they are
 * valid.
*-------------------------------------------------------------------*/
int MyCheckHttpPass( int group, const char *puser, const char *ppass )
{
   iprintf( "Testing %s, %s for group %d\r\n", puser, ppass, group );

   if ( group == 0 )    // Group 0 means there is no password
      return 1;

   if ( puser == NULL ) // Fail if a parameter is missing
      return 0;
   if ( ppass == NULL )
      return 0;

   // Now test each password group
   if ( ( group == 1 ) && ( ( strcmp( puser, "user" ) == 0 ) &&
        ( strcmp( ppass, "pass" ) == 0 ) ) )
      return 1;

   if ( ( group == 2 ) && ( ( strcmp( puser, "top" ) == 0 ) &&
        ( strcmp( ppass, "secret" ) == 0 ) ) )
      return 1;

   return 0;
}
```

8.1.3 Password Check Function Assignment

If the global system value of `pHttpPassFunc` is NULL, then no password checks are performed. The application must assign this function pointer value to point at the password check function. An example is shown below:

```
/*-------------------------------------------------------------------
 * UserMain
 *-------------------------------------------------------------------*/
void UserMain(void *pd)
{
   SimpleUart( SERIAL_DEBUG_PORT, SystemBaud ); // Initialize debug UART
   assign_stdio( SERIAL_DEBUG_PORT );           // Use UART 0 for STDIO

   InitializeStack();         // Initialize TCP stack
   EnableAutoUpdate();        // Enable network downloads to target
   OSChangePrio( MAIN_PRIO ); // set standard UserMain task priority

   EnableTaskMonitor();
   EnableSmartTraps();

   /* The default value of pHttpPassFunc is NULL. Passwords are enabled
    * by assigning this pointer to a valid function to do the password
    * check.
    */
   pHttpPassFunc = MyCheckHttpPass;

   WaitForDhcpAddressAssignment();
   StartHTTP();

   **** rest of UserMain() follows ****
```

9. Serial Ports

9.1 Polled vs. Interrupt-Driven

The NetBurner API provides two types of serial interfaces for the onchip UARTs: polled and interrupt-driven. You can switch between either mode easily just by changing the include file in your application; the application function calls are identical.

```
#include <serialpoll.h>          // UARTs use polling
```

or

```
#include <serialirq.h>          // UARTs use interrupts
```

Polling means that any time your application attempts a serial read or write, the underlying code will block until a character can be read or written. This is accomplished by polling a status bit in the UART registers. The advantage of polling is that it takes up less SRAM resources than an interrupt-driven scheme since the serial I/O is not buffered.

Interrupt-driven means that the serial I/O is buffered so your application does not have to wait for the actual I/O to occur. It also means the application will not miss any incoming characters when it is busy elsewhere. Unless you are constrained on SRAM space, interrupt-driven serial I/O is recommended.

9.2 Serial Polling and Interrupt-Driven Example Programs

In the AppWizard-generated application below, we can see that it uses interrupt-driven serial I/O, as indicated by the included serialirq.h header file:

```
/*****************************************************************************
   Interrupt driven serial I/O
*****************************************************************************/

#include <predef.h>
#include <basictypes.h>         // Include for variable types
#include <stdio.h>
#include <ucos.h>               // Include for RTOS functions
#include <ucosmcfc.h>           // Include for RTOS functions
#include <serialirq.h>          // UARTs use interrupts
#include <utils.h>              // Include for LED writes on carrier board
#include <constants.h>          // Include for constants like MAIN_PRIO
#include <system.h>             // Include for system functions
#include <netif.h>
#include <ip.h>
#include <autoupdate.h>
#include <string.h>

// Instruct the C++ compiler not to mangle the function name
extern "C"
{
   void UserMain( void *pd );
}

// Name for development tools to identify this application
const char *AppName = "Interrupt Driven Serial Example";

// Main task
void UserMain(void *pd)
{
   SimpleUart( 0, SystemBaud );    // initialize UART 0
   assign_stdio( 0 );              // use UART 0 for stdio
   InitializeStack();
   WaitForDhcpAddressAssignment();
   EnableAutoUpdate();
   OSChangePrio( MAIN_PRIO ); // set standard UserMain task priority

   iprintf( "this application uses interrupt driven serial I/O\r\n");

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

We can convert this to polled serial I/O just by changing the include file. However, if we want the serial flash update utility to work, we need to look (poll) for any incoming characters. The code to do this has been added to the `while` loop in `UserMain()` and highlighted in italics:

```
/*****************************************************************************
   Polled Serial
*****************************************************************************/
#include <predef.h>
#include <basictypes.h>         // Include for variable types
#include <stdio.h>
#include <ucos.h>               // Include for RTOS functions
#include <ucosmcfc.h>           // Include for RTOS functions
#include <serialpoll.h>       // UARTs use polling
#include <utils.h>              // Include for LED writes on carrier board
#include <constants.h>          // Include for constants like MAIN_PRIO
#include <system.h>             // Include for system functions
#include <netif.h>
#include <ip.h>
#include <autoupdate.h>
#include <string.h>


// Instruct the C++ compiler not to mangle the function name
extern "C"
{
   void UserMain( void *pd );
}

// Name for development tools to identify this application
const char *AppName = "Polled Serial Example";

// Main task
void UserMain(void *pd)
{
   SimpleUart( 0, SystemBaud );   // initialize UART 0
   assign_stdio( 0 );             // use UART 0 for stdio
   InitializeStack();
   WaitForDhcpAddressAssignment();
   EnableAutoUpdate();
   OSChangePrio( MAIN_PRIO ); // set standard UserMain task priority

   iprintf( "this application uses polled serial I/O\r\n");

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

9.3 Interrupt Serial Buffers

The serial I/O buffer sizes are located in \Nburn\include_sc\constants.h, and are shown below (in bytes):

```
// UART buffer sizes */
#define SERIAL0_RX_BUFFER_SIZE (16)
#define SERIAL0_TX_BUFFER_SIZE (16)

#define SERIAL1_RX_BUFFER_SIZE (16)
#define SERIAL1_TX_BUFFER_SIZE (16)

#define SERIAL2_RX_BUFFER_SIZE (5)
#define SERIAL2_TX_BUFFER_SIZE (5)
```

You can change these values to increase or decrease the overall memory usage. After making the modifications, the system library needs to be recompiled in order for the new changes to take effect. This is done in the NBEclipse IDE by going to the top main menu and selecting: NBEclipse → Rebuild system files, followed by recompiling your application so it uses the new library.

9.4 The NetBurner Serial API

The following sections describe the NetBurner serial API function calls. All the functions can be run in polled or interrupt-driven mode by changing the include file as described earlier in this chapter. Each API function call has the underlying polled and interrupt-driven functions defined.

Include Files
#include <serialirq.h>
#include <serialpoll.h>

Open and Close

`InitUart()`	Initialize the serial port, all options
`SimpleUart()`	Initialize serial port, specify baud rate
`close()`	Close serial port

Read and Write

`charavail()`	TRUE if character is available to be read
`sgetchar()`	Get a single character
`SerialGetLine( )`	Get a line of text terminated by a carrage return
`writechar()`	Write a single character
`writestring()`	Write an ASCII null terminated string

I/O Assignments

`assign_stdio()`	Assign serial port as stdin and stdout
`assign_sterr()`	Assign serial port as stderr
`create_file()`	Create a FILE type handle for the serial port

Serial Errors

```
SERIAL_ERR_NOSUCH_PORT (-1)
SERIAL_ERR_PORT_NOTOPEN (-2)
SERIAL_ERR_PORT_ALREADYOPEN (-3)
SERIAL_ERR_PARAM_ERROR  (-4)
```

9.4.1 Open and Close Functions

9.4.1.1 InitUart

Synopsis:

```
int InitUart( int portnum, unsigned int baudrate, int stop_bits,
              int data_bits, parity_mode parity );
```

Description:
Initializes the serial port with the specified parameters. When serialpoll.h is included: InitPolledUart() is called. When serialirq.h is included: InitIRQUart() is called.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1
DWORD	baud	Baud rate: 1200 to 115200
int	stop_bits	Stop bits: 1 or 2.
int	data_bits	Data bits: 5, 6, 7 or 8.
parity_mode	parity	eParityNone, eParityOdd, eParityEven or eParityMulti

Returns:
0 on success
Serial error on falure

Example:

```
InitUart( 0, 115200, 1, 8, eParityNone );   // Initialize UART 0 to 115,200 baud
assign_stdio( 0 );          // Assign UART 0 to stdio
```

9.4.1.2 SimpleUart

Synopsis:

```
#define SimpleUart( port, baud )
InitUart( port, baud, 1, 8, eParityNone )
```

Description:
SimpleUart() is a #define to make opening a serial port easier for the most common settings for stop bits, start bits and parity. You need to speify only the UART port number and baud rate.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1
DWORD	baud	Baud rate: 1200 to 115200
int	stop_bits	Always set to 1
int	data_bits	Always set to 8
parity_mode	parity	Always set to eParityNone

Returns:
0 on success
Serial error on falure

Example:

```
SimpleUart( 0, 115200 );   // Initialize UART 0 to 115,200 baud
assign_stdio( 0 );         // Assign UART 0 to stdio
```

9.4.1.3 close

Synopsis:

```
void close( int portnum );
```

Description:
Close a serial port.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
Nothing

Example:

```
close( 0 );   // Close UART 0
```

9.4.2 Read and Write Functions

9.4.2.1 charavail

Synopsis:

```
BOOL charavail( int portnum );
```

Description:
Returns TRUE if a characer is available to be read. This can be useful to avoid a serial read function from blocking. The UART must be opened before using this function.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
TRUE if at least one characer is available to be read, otherwise FALSE.

Example:

```
while ( 1 )
{
    if ( charavail( SERIAL_DEBUG_PORT ) )
    {
        ProcessSerialDebugCommand();
    }
}
```

9.4.2.2 sgetchar

Synopsis:

```
char sgetchar( int portnum );
```

Description:
Retruns a single character from the specified UART port number. This function will block until a character is available. The UART must be opened before using this function.

Note: The polled version does not yield to the RTOS if it blocks, so no lower priority tasks can run. The IRQ version will yield to the RTOS until a character is available to be read.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
One character

Example:

```
char c = sgetchar( SERIAL_DEBUG_PORT );
```

9.4.2.3 SerialGetLine

Synopsis:

```
void SerialGetLine(int portnum, char *buffer, int maxlen );
```

Description:
Read a line of text from the specified serial port and copy it to the specified buffer. This function is primarily used for user input, and the line of text must be terminated by a carrage return. The function will process backspaces and delete commands. This function is preferable to a standard I/O function such as gets() and fgets() because it uses just a few bytes of memory. The standard I/O function may dynamically allocte up to 2k of RAM.

Note: The polled version does not yield to the RTOS if it blocks, so no lower priority tasks can run. The IRQ version will yield to the RTOS until a character is available to be read.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1
char *	buffer	Point to the destincation buffer to copy received data
int	maxlne	Maximum numer of characters to read

Returns:
One character

Example:

//Read up to 80 characters from UART 0.

```
#define UART_0 0
#define BUFFER_SIZE 80

static char buffer[BUFFER_SIZE];
SerialGetLine( UART_0, buffer, BUFFER_SIZE );
```

9.4.2.4 writechar

Synopsis:

```
void writechar( int portnum, char c );
```

Description:
Write a single character to the specified UART port. This function will block until the character can be written. The UART must be opened before using this function.

Note: The polled version does not yield to the RTOS if it blocks, so no lower priority tasks can run. The IRQ version will yield to the RTOS until a character is available to be written.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
Nothing

Example:

```
char c = 'A';
int port = 0;
writechar( port, c );
```

9.4.2.5 writestring

Synopsis:

```
void writestring( int portnum, const char * s );
```

Description:
Writes an ASCII null terminated string to the specified UART port. This function will block until all the characters can be written. The UART must be opened before using this function.

Note: The polled version does not yield to the RTOS if it blocks, so no lower priority tasks can run. The IRQ version will yield to the RTOS until a character is available to be written.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1
char *	s	Pointer to an ASCII null terminated string.

Returns:
Nothing

Example:

```
int port = 0;
writestring( port, "Hello World\r\n");
```

9.4.3 I/O Assignments

9.4.3.1 assign_stdio

Synopsis:

```
void assign_stdio( int portnum );
```

Description:
Assigns specified UART port number to stdin and stdout, which enables it to be used for system I/O functions such as iprintf(), printf(), siprintf() and sprintf(). The ‘i' in these function names stands for “integer”, and fucntions such as iprintf() cannot print floating point numbers. The advantage is that iprintf() takes very little memory, so if you are not printing floating point numbers it is the best function to use. If even one printf() function is called, the floating point library will be linked in.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:

Nothing

Example:

```
assign_stdio( 0 );              // use UART 0 for stdio
```

9.4.3.2 assign_stderr

Synopsis:

```
void assign_stderr( int portnum );
```

Description:
Assigns specified UART port number to stderr.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
Nothing

Example:

```
assign_stderr( 0 );             // use UART 0 for stderr
```

9.4.3.3 assign_stderr

Synopsis:

```
FILE * create_file( int portnum );
```

Description:
Creates a pointer of type FILE, which can then be used in functions such as read() and write() that take FILE pointers as arguments.

Parameters:

Type	Name	Description
int	portnum	UART port number: 0 or 1

Returns:
Nothing

Example:

```
FILE *fp = create_file( 1 );                // Create a FILE pointer for
                                            //    UART 1
fprintf( fp, "This goes out port 1\r\n" );  // Write string
fclose( fp );
```

10. TCP

10.1 TCP Buffered Programming Interface

The Buffered TCP interface is implemented as a C++ Class called BufferedTcpObject, located in c:\nburn\include_sc\tcp_buffer.h. You do not need to know how to program in C++ to use this class once you have created the object. The constructor is show below:

```
BufferedTcpObject( BYTE * pBuf, int buf_len, OS_SEM *notify_sem = NULL )
```

To create this object you need to provide a buffer to store the received network data. This buffer should be declared global, or you can use the "static" keyword if it is declared in a function. The size of the buffer determines the received buffer size. The buf_len specifies the size of the buffer.

The optional semaphore nofity_sem enables an application to attach a semaphore to a TCP socket for so it can pend on the semaphore for event notifications. This method can be used in a task to make the task block, allowing other lower priority tasks to run. Notifications are generated for:

- New connections on a listening socket
- Closed connections
- Connection errors
- New data available to read

A single semaphore can be used for multiple objects. In this way a task could pend on the semaphore, and when any of the objects has an event change the task would wake up and use the status functions to determine which object needs to be serviced.

The following example creates a TCP object named "tcp" with a receive buffer of 256 bytes, and no semaphore.

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );
```

To use this object with any of the Buffered function calls, use the object name, "tcp", followed by a '.' and the function name. For example, to call the status function Connected():

```
int status;
status = tcp.Connected();
```

10.1.1 TCP Buffered Function Summary

Include Files
#include <tcp.h>
#include <tcp_buffer.h>

Status functions

`bool NewConnection()`	Returns TRUE one time when a new connection is made.
`bool Connected()`	Returns TRUE for active socket connection.
`bool Listening()`	Returns TRUE if socket is listening.
`int DataAvail()`	Returns number of bytes available to read.
`int ErrorState()`	Returns the error state. 0 = no error.

State Changing Activities

`void Close()`	Close an active connection.
`void CloseListen()`	Tells a listen socket to stop listening for connections.
`bool Listen()`	Listen will close if already open.
`bool Connect()`	Make an outgoing connection.

Active Socket Connection Information

`WORD RemotePort()`	Return remote client port number
`WORD LocalPort()`	Return local port number
`IPADDR RemoteIP()`	Return remote client IP address

Read Functions
Read functions block until at least one character is available or a socket error occurs.

`int Read()`	Read available data, up to the specified maximum bytes
`int ReadAtLeast()`	Read at least the specified number of bytes

Write functions
Write functions return the number of bytes written, and do not return until the client either acknowledges everything or the specified timeout value expires.

`int Write()`	Write the specified number of bytes
`int Printf()`	Write formatted output using the printf style

Error Codes

```
#define TCP_ERR_NORMAL (0)
#define TCP_ERR_TIMEOUT (-1)
#define TCP_ERR_NOCON  (-2)
#define TCP_ERR_CLOSING (-3)
#define TCP_ERR_NOSUCH_SOCKET (-4)
#define TCP_ERR_NONE_AVAIL (-5)
#define TCP_ERR_CON_RESET (-6)
#define TCP_ERR_CON_ABORT (-7)
```

10.1.2 TCP Buffered Status Functions

10.1.2.1 NewConnection

Synopsis:

```
bool NewConnection()
```

Description:
Function will return TRUE if a new TCP socket connection has been established. Can be used for outgoing client connections, or for incoming connections on listening sockets. The new connection status is cleared the first time the function is called and will not be TRUE again until a new connection is established.

Parameters:
None

Returns:
TRUE if a new connection has been established. Otherwise return FALSE.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.NewConnection() )
{
   iprintf( "Accepted connection from: %I : %d\r\n", tcp.RemoteIP(),
            tcp.RemotePort() );
   tcp.Printf( "Welcome [%I] to the new connection\r\n", tcp.RemoteIP() );
   // If you do not want to allow new connections to bump
   // existing one, comment out the line below.
   tcp.Listen( TCP_LISTEN_PORT );
}
```

10.1.2.2 Connected

Synopsis:

```
bool Connected()
```

Description:
Function returns TRUE if an active TCP connection exists.

Parameters:
None

Returns:
TRUE if an active connection exists. Otherwise return FALSE.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.Connected() )
   iprintf( "Status = connected\r\n" );
else
   iprintf( "Status = not connected\r\n" );
```

10.1.2.3 Listening

Synopsis:

```
bool Listening()
```

Description:
Function returns TRUE if the socket is listening for incoming TCP connections.

Parameters:
None

Returns:
TRUE if the socket is listening. Otherwise return FALSE.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.Listening() )
  iprintf( "Socket is listening\r\n" );
else
  iprintf( "Socket is not listening\r\n" );
```

10.1.2.4 DataAvail

Synopsis:

```
int DataAvail()
```

Description:
Function returns the number of bytes available to read in the applications TCP receive buffer. This function should be called before a Read() function to ensure there is data available to be read.

Parameters:
None

Returns:
Integer value of the number of bytes available to read.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

while ( tcp.DataAvail() )
{
   char rxbuffer[32];
   int rv = tcp.Read( rxbuffer, 31 );
   if ( rv > 0 )
   {
      rxbuffer[rv] = 0;
      iprintf( "Received%d:[%s]\r\n", rv, rxbuffer );
   }
}
```

10.1.2.5 ErrorState

Synopsis:

```
int ErrorState()
```

Description:
Returns the TCP socket error state. This is the only way to determine if there are errors on a TCP socket connection. This function should be placed in your socket servicing loop as a regular check for error conditions such as the remote host closing or resetting an active connection.

Parameters:
None

Returns:
Integer value of the number of bytes available to read.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

int last_error = 0;
if ( last_error != tcp.ErrorState() )
{
   last_error = tcp.ErrorState();
   iprintf( "Error State= %d\r\n", last_error );
}
```

10.1.3 TCP Buffered State Change Functions

10.1.3.1 Close

Synopsis:

```
void Close()
```

Description:
Close an active TCP connection. Can be used for outgoing client connections, or on listen sockets with active connections. If called for a listen socket, the active connection will be closed and the socket will continue to listen for new incoming connections. This function is safe to call even if the socket is already closed.

Parameters:
None

Returns:
Nothing

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.Connected() )
{
   tcp.Close();
   iprintf( "Closing socket \r\n" );
}
else
   iprintf( "Socket not connected\r\n" );
```

10.1.3.2 CloseListen

Synopsis:

```
void CloseListen()
```

Description:
Call to close a listening socket. Note that any time a listening socket accepts an incoming connection, listening is automatically disabled and must be re-enabled by calling the Listen( ) member function again. Therefore the use of CloseListen( ) is primarily to close a listening socket without an active connection. If you are unsure if a connection is active, you can safely call Close( ), followed by CloseListen( ).

Parameters:
None

Returns:
Nothing

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( !tcp.Listening() )
{
   iprintf( "We aren't listening\r\n" );
}
else
{
   tcp.Close();
   tcp.CloseListen();
   iprintf( "Listen closed\r\n" );
}
```

10.1.3.3 Listen

Synopsis:

```
bool Listen(WORD port)
```

Description:
Enable the socket to listen for incoming TCP connections on the specified TCP port number. If the function is called a second time with a different port number it will close the current listening port and start listing on the new port number.

Parameters:

Type	Name	Description
bool	port	TCP port number to listen on for incoming connections.

Returns:
TRUE on success, FALSE if a socket could not be opened.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

tcp.Listen( TCP_LISTEN_PORT );
```

10.1.3.4 Connect

Synopsis:

```
bool Connect( IPADDR dest_ip, WORD dest_port, WORD timeout )
```

Description:
Create a TCP connection to the specified destination IP address and port number. If called for an existing connection, that connection will be closed.

Parameters:

Type	Name	Description
IPADDR	dest_ip	Destination IP address
WORD	dest_port	Destination TCP port number
WORD	timeout	Maximum wait time in system clock ticks

Returns:
TRUE on success, FALSE if a socket could not be opened.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );
IPADDR DestIP = AsciiToIp( "192.168.1.10" );
int DestPort = 2323;

iprintf( "Attempting TCP connection to %I : %d\r\n", DestIP, DestPort );

if( tcp.Connect( DestIP, DestPort, TICKS_PER_SECOND * 5) )
{
   iprintf( "Connection Succeeded. \r\n");
}
else
{
   iprintf( "*** Connection Failed. \r\n");
}
```

10.1.4 TCP Buffered Connection Information Functions

10.1.4.1 LocalPort

Synopsis:

```
WORD   LocalPort()
```

Description:
Returns the local TCP port number of an active TCP connection.

Parameters:
None

Returns:
Local host port number.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.NewConnection() )
{
  iprintf( "New connection with local port number: %d\r\n", tcp.LocalPort() );
}
```

10.1.4.2 RemoteIP

Synopsis:

```
IPADDR   RemoteIP()
```

Description:
Returns the remote host IP address of an active TCP connection.

Parameters:
None

Returns:
Remote host IP address.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.NewConnection() )
{
   iprintf( "Accepted connection from: %I : %d\r\n", tcp.RemoteIP(),
            tcp.RemotePort() );
   tcp.Printf( "Welcome [%I] to the new connection\r\n", tcp.RemoteIP() );
   // If you do not want to allow new connections to bump
   // existing one, comment out the line below.
   tcp.Listen( TCP_LISTEN_PORT );
}
```

10.1.4.3 RemotePort

Synopsis:

```
WORD    RemotePort()
```

Description:
Returns the remote host TCP port number of an active TCP connection.

Parameters:
None

Returns:
Remote host port number.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

if ( tcp.NewConnection() )
{
   iprintf( "Accepted connection from: %I : %d\r\n", tcp.RemoteIP(),
            tcp.RemotePort() );
   tcp.Printf( "Welcome [%I] to the new connection\r\n", tcp.RemoteIP() );
   // If you do not want to allow new connections to bump
   // existing one, comment out the line below.
   tcp.Listen( TCP_LISTEN_PORT );
}
```

10.1.5 TCP Buffered Read Functions

10.1.5.1 Read

Synopsis:

```
int Read( char * buffer, int maxlen )
```

Description:
All Read functions will block until at least one byte can be read from the specified buffer, or an error in the connection occurs. Read() will return no more than the specified maximum number of bytes, but it can only read the number of bytes available; there is no guarantee the maximum number of bytes will be read. To prevent Read() from blocking, it is recommended to call the DataAvail() function first to determine if there are bytes available in the buffer.

Parameters:

Type	Name	Description
char *	buffer	Buffer allocated to store received network data
int	maxlen	Maximum number of characters to read. Note that function may return with less than the maximum number of characters.

Returns:
Number of bytes read on success
TCP error code if a connection failure occurs

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

while ( tcp.DataAvail() )
{
   char rxbuffer[32];
   int rv = tcp.Read( rxbuffer, 31 );
   if ( rv > 0 )
   {
      rxbuffer[rv] = 0;
      iprintf( "Received%d:[%s]\r\n", rv, rxbuffer );
   }
}
```

10.1.5.2 ReadAtLeast

Synopsis:

```
int ReadAtLeast( char * buffer, int maxlen, int minlen,
                 WORD timeout = 0 )
```

Description:
ReadAtLeast() will block until the specified minimum number of characters are read from the buffer, the optional timeout parameter expires, or an error in the connection occurs. A timeout value of 0 (the default) will wait forever. To prevent blocking it is recommended to call the DataAvail() function first to determine if there are bytes available in the buffer.

Parameters:

Type	Name	Description
char *	buffer	Buffer allocated to store received network data
int	maxlen	Maximum number of characters to read. Note that function may return with less than the maximum number of characters.
int	minlen	Minimum number of characters to read. A minlen of 0 will return immediately with whatever characters could be read.
WORD	timeout	Amount of time to wait for minimum number of characters, in system clock ticks. Default is 0, which waits forever.

Returns:
Number of bytes read on success
TCP error code if a connection failure occurs

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

while ( tcp.DataAvail() )
{
   char rxbuffer[32];
   int rv = tcp.ReadAtLeast( rxbuffer, 31, 5, TICKS_PER_SECOND * 5 );
   if ( rv > 0 )
   {
      rxbuffer[rv] = 0;
      iprintf( "Received%d:[%s]\r\n", rv, rxbuffer );
   }
}
```

10.1.6 TCP Buffered Write Functions

10.1.6.1 Write

Synopsis:

```
int Write( char * data, int len = 0, WORD timeout = 0 )
```

Description:
Writes "len" bytes from buffer pointed to by "data" and returns the number of bytes written. The function does not return until the remote client acknowledges all bytes written or the optional timeout value expires. The default timeout value of 0 waits forever. A len value of 0 will automatically do a strlen() to support writing ASCII null terminated strings.

Parameters:

Type	Name	Description
char *	data	Pointer to data buffer containing data to send.
int	len	Number of bytes to write. A value of 0 will invoke the strlen() function for ASCII null terminated strings.
WORD	timeout	Amount of time to wait for minimum number of characters, in system clock ticks. Default is 0, which waits forever.

Returns:
Number of bytes written on success
0 on timeout or if socket does not have a connection.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

// Write an ASCII string using default 0 for len and timeout
tcp.Write( "This message is from the TCP Server\r\n" );

// Write 10 data bytes with a timeout of 5 seconds
tcp.Write( data, 10, TICKS_PER_SECOND * 5 );
```

10.1.6.2 Printf

Synopsis:

```
int Printf( const char * format,... )
```

Description:
Writes all bytes and returns the number of bytes written. The function does not return until the remote client acknowledges all bytes written. The standard printf format characters are supported, as well at %I, which formats an IP address in dotted decimal, and %b for binary.

Parameters:

Standard printf variable format

Returns:
Number of bytes written on success.
Returns 0 if socket does not have a connection.

Example:

```
static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

static BYTE buffer[256];
BufferedTcpObject tcp( buffer, 256, NULL );

tcp.Printf("Hello World");
```

11. UDP

User Datagram Protocol (UDP) processing consists of a callback function to handle received UDP packets, and API functions to transmit outgoing UDP packets.

In order to send a UDP packet you need to know both the destination IP and Media Access Control (MAC) addresses. To perform well in small memory footprints the system does not maintain and Address Resolution Protocol (ARP) cache that pairs each IP address with its corresponding MAC address, and there are different methods to how the UDP data to be sent is buffered. These topics only apply to outgoing UDP packets, and each is explained in the following sections.

Please see the example code for UDP functions located in your \nburn\examples\SBL2e directory for the latest revisions.

11.1 UDP Function Summary

Include Files
#include <udpsend.h>

Send Immediate Functions

`SendScratchUdp()`	Send packet using system scratchpad buffer
`SendBufferedUdp()`	Send packet using private application buffer

Assemble and Send Functions

`PreSendBufferedUdp()`	Assemble UDP packet header
`AddBufferedUdpData()`	Add data to UDP packet
`PostSendBufferedUdp()`	Finish and send UDP packet

11.2 UDP Receive Function Example

```
#define UDP_LOCAL_PORT (1000)  // Source UDP port
#define UDP_REMOTE_PORT (1000) // Destination UDP port

pUserUdpProcessFunction = UdpReceiveFunction;  // Call from UserMain()

/*-------------------------------------------------------------------
 * Called for all received UDP packets not handled by other parts
 * of the system. Display packet information.
 *-------------------------------------------------------------------*/
int UdpReceiveFunction(PEFRAME pf)
{
    PUDPPKT pUdpPkt = GetUdpPkt( pf );
    PIPPKT pIp = GetIpPkt( pf );

    if ( pUdpPkt->dstPort == UDP_LOCAL_PORT )
    {
        iprintf( "Received UDP packet from: %I : %d, length = %d\r\n[",
              pIp->ipSrc, pUdpPkt->srcPort, GetUdpDataLen( pUdpPkt ));

        for ( int i = 0; i < GetUdpDataLen( pUdpPkt ); i++ )
        {
            iprintf( "%c", pUdpPkt->DATA[i] );
        }
        iprintf( "]\r\n" );

        return 1; //We handled the packet
    }
    return 0;//We did handle the packet
}
```

11.3 Determining the Destination MAC Address

All manufacturers of Ethernet devices are required to provide a unique MAC address that identifies their company and the network device. In order to send a UDP packet over Ethernet to another network device, the sender must resolve the destination IP address to the destination's MAC address. This is the function of the ARP. To conserve RAM space the system does not maintain an ARP cache as you would find on larger systems. There are 2 methods to determine the destination MAC address:

Method 1: Use the GetArp() Function

Call the GetArp() function before sending the UDP packet. If your application is sending to one IP address, you could call GetArp() once, store the MAC address, and use it for future transmissions to the same IP address. One item to be aware of is that if the hardware is changed at the destination, the MAC address for that IP address would change and you would need to run GetArp() again to update the value. It would be a good idea periodically call GetArp() to refresh this value. An ARP cache on a larger system will typically delete an ARP entry if it has not been used in 10 minutes.

An example code snippet is shown below:

```
IPADDR DestinationIp;
MACADR DestinationMac;

/*-------------------------------------------------------------------
 * Call GetArp() to resolve an IP address to a MAC address.
 *-------------------------------------------------------------------*/
void SendToSpecifiedIp()
{
    char buf[17];

    iprintf("Enter destination IP address: ");
    fgets( buf, 17, stdin );
    DestinationIp = AsciiToIp( buf );
    iprintf("\r\n");

    if ( !GetArp( DestinationIp, DestinationMac, 40 ) )
    {
        //We failed to get the MAC address
        iprintf( "ARP failed for IP Address: %I\r\n", DestinationIp );
    }
    else
    {
       iprintf("Destination IP address set to: %I\r\n",
                DestinationIp );
    }
}
```

Method 2: Send to the Address of the Last Received Packet

Each time a UDP packet is received the sender's MAC address is specified in the packet. In many applications a client is requesting information, and if you always want to send data back to that same client you can store the MAC address of the last received packet and use it as the destination MAC address when you transmit a UDP packet. In this way you do not need to call GetArp(), and it will not matter if the sender's IP address or MAC address changes, or even if you want to service multiple clients.

An example code snippet is shown below.

```
IPADDR DestinationIp;
MACADR DestinationMac;
IPADDR LastReceivedIp;   // IP address of last received packet
MACADR LastReceivedMac;  // MAC address of last received packet

/*---------------------------------------------------------------------
 * Use destination IP and MAC address from last received UDP packet.
 * If no UDP packets have been received, use the UDP broadcast address.
 *---------------------------------------------------------------------*/
void SendToLastReceivedIp()
{
    if ( LastReceivedIp == 0 )
    {
        // No received packets, so use broadcast
        LastReceivedMac = ENET_BCAST;
        LastReceivedIp = 0xFFFFFFFF;
    }

    DestinationIp = LastReceivedIp;
    DestinationMac = LastReceivedMac;
    bSendToLastIp = TRUE;
    iprintf("Destination IP address set to: %I\r\n", LastReceivedIp );
}
```

11.4 UDP Send Data Options

In order for an application to send data in a UDP packet, the system needs a memory buffer to assemble the UDP packet, including things like the UDP header and checksum. The SBL2e network stack provides 3 means for providing this buffer space:

11.4.1 Method 1: Allocate Data Buffer and Use System Scratchpad Buffer

The scratchpad buffer is a single buffer that is used by the system for multiple purposes. While it saves on memory usage by reusing a single buffer, the sending of the UDP packet can be delayed if a different part of the system is using the buffer when the send function is called.

Example:

```
void SendUdpWithScratchpadBuffer( char *data, int len )
{
   SendScratchUdp( data, len, UDP_LOCAL_PORT, UDP_REMOTE_PORT,
                   DestinationIp, DestinationMac );
}
```

Pros:

- Easy to use
- Saves memory by using scratchpad buffer to send UDP packets.

Cons:

- Must share system scratchpad, so if there is more than one task doing this they will have to wait for each other.
- System tasks like ipsetup, arp and autoupdate will also use the scratchpad buffer.
- The application must store all the data in a memory buffer before sending.

11.4.2 Method 2: Allocate Both System and Data Buffers

The second method allocates a buffer to be used by the system (instead of the scratchpad buffer) to hold the entire UDP packet including the application data and UDP packet overhead. **This buffer must be 42 bytes bigger than the maximum packet size and aligned on a 16 byte boundary.**

This method consumes the most RAM space because the application must create 2 buffers: one to hold the application data to be sent, and the system buffer that must be large enough to hold a copy of the application data plus the UDP packet overhead. The buffers only need to be large enough to handle the amount of data you want to send, up to a maximum of 1458 bytes. To assist in sizing the buffers the following definitions are provided in constants.h:

```
// Max number of data bytes in a single UDP packet
#define MAX_UDP_TX_BYTES 1458

// UDP packet overhead, add to data for total system buffer size
#define UDP_OVERHEAD_SIZE 42
```

A semaphore is used to determine when the sending buffer has been released by the system.

```
BYTE UdpPrivateBuffer[MY_UDP_DATA_SIZE + UDP_OVERHEAD_SIZE] __attribute__((aligned(4)));

int SendUdpWithPrivateBuffer( char *data, int len )
{
    OS_SEM sem;

    if ( len < MY_UDP_DATA_SIZE )
    {
       OSSemInit( &sem, 0 );
       SendBufferedUdp( data, len, UDP_LOCAL_PORT, UDP_REMOTE_PORT,
                        DestinationIp, DestinationMac, UdpPrivateBuffer, &sem );
       OSSemPend( &sem, 0 );  // Wait until the entire buffer has been sent
       iprintf("Sent %s, %d bytes to %I\r\n", data, len, DestinationIp );
       return 0;
    }
    else
    {
       iprintf(" Error: max data size = %d bytes\r\n", MY_UDP_DATA_SIZE );
       return -1;  // error
    }
}
```

Pros:

- Easy to use.
- UDP system transmit buffer is not shared with other tasks.

Cons:

- Allocation of both system and data buffers consume the most RAM space. The system buffer must be 42 bytes bigger than the data buffers size.

11.4.3 Method 3: Allocate System Buffer Only

This method consumes a single RAM buffer that is used for the UDP system transmit buffer. It is similar to the scratchpad example, without the requirement to share the buffer with the rest of the system. The second major difference is that the application does not need to allocate its own data buffer; The AdddBufferedUdpData() function is used to add application data to the system buffer directly.

As an example, lets say an application needs to take g 10 analog to digital readings and send them out in a single UDP packet along with a time stamp. As each A/D value is obtained, it is added to the UDP packet using AddBufferedUdpData(), which is being assembled in the system transmit buffer. When all 10 readings are complete the packet is sent. The memory benefit of this method is that you have a single buffer that is used to assemble the data and to send the UDP packet.

As with Method 2, the system transmit buffer must be 42 bytes bigger than the maximum packet size and aligned on a 16 byte boundary.

```
BYTE UdpPrivateBuffer[MY_UDP_DATA_SIZE + UDP_OVERHEAD_SIZE] __attribute__((aligned(4)));

OS_SEM sem;
OSSemInit( &sem, 0 );

PreSendBufferedUdp( UDP_LOCAL_PORT, UDP_REMOTE_PORT, DestinationIp,
                    DestinationMac, UdpPrivateBuffer );

for ( int i = 0; i < 3; i++ )
{
   iprintf("Enter message %d of 3: ", i );
   fgets( MsgBuffer, MSG_BUFFER_SIZE, stdin );
   AddBufferedUdpData( UdpPrivateBuffer, (PBYTE)MsgBuffer, strlen(MsgBuffer) );
   iprintf("\r\n");
}
PostSendBufferedUdp( UdpPrivateBuffer, &sem );
OSSemPend( &sem, 0 );
```

Pros:

- UDP system transmit buffer is not shared with other tasks.
- Only requires the system transmit buffer, does not require an application data buffer.
- Intermediate memory footprint

Cons:

- Most complex to use.

11.5 SendScratchUdp

Synopsis:

```
void SendScratchUdp(const char * data, int datalen, WORD local_port,
                    WORD remote_port, IPADDR ip_to, MACADR & mac);
```

Description:
The scratchpad buffer is a single buffer that is used by the system for multiple purposes. While it saves on memory usage by reusing a single buffer, the sending of the UDP packet can be delayed if a different part of the system is using the buffer when the send function is called.

Parameters:

Type	Name	Description
const char *	Data	Data to be transmitted in the UDP packet
int	datalen	Number of data bytes
WORD	local_port	Local/Source UDP port number
WORD	remote_port	Remote/Destination UDP port number
IPADDR	ip_to	Destination IP address
MACADR &	Mac	Destination MAC address

Returns:

Nothing

11.6 SendBufferedUdp

Synopsis:

```
void SendBufferedUdp(const char * data, int datalen, WORD local_port,
                     WORD remote_port, IPADDR ip_to, MACADR & mac,
                     BYTE * buffer, OS_SEM * pSem );
```

Description:
Sends a UDP packet using a private system transmit buffer for the UDP packet data and UDP packet overhead. A semaphore should be used to determine when the buffer has been released by the system.

Parameters:

Type	Name	Description
const char *	data	Pointer to data to be transmitted in the UDP packet
int	datalen	Number of data bytes
WORD	local_port	Local/Source UDP port number
WORD	remote_port	Remote/Destination UDP port number
IPADDR	ip_to	Destination IP address
MACADR &	mac	Destination MAC address
BYTE	buffer	UDP transmit buffer to be used by the system. This buffer must be 42 bytes larger than the data you want to transmit and aligned on a 16 byte boundary.
OS_SEM *	pSem	Pointer to a RTOS semaphore used to determine when the system has released the UDP transmit buffer.

Returns:

Nothing

11.7 PreSendBufferedUdp

Synopsis:

```
void PreSendBufferedUdp( WORD local_port, WORD remote_port, IPADDR ip_to,
                         MACADR & mac,BYTE * buffer );
```

Description:
Used in conjunction with `AddBufferedUdpData()` and `PostSendBufferedUdp()` to create a UDP packet by assembling it in a UDP system transmit buffer allocated by the application. This function configures the UDP packet header.

Parameters:

Type	Name	Description
WORD	local_port	Local/Source UDP port number
WORD	remote_port	Remote/Destination UDP port number
IPADDR	ip_to	Destination IP address
MACADR &	mac	Destination MAC address
BYTE	buffer	UDP transmit buffer to be used by the system

Returns:

Nothing

11.8 AddBufferedUdpData

Synopsis:

```
void AddBufferedUdpData(BYTE * buffer, const unsigned char * pData, WORD len);
```

Description:
Used in conjunction with `PreSendBufferedUdp()` and `PostSendBufferedUdp()` to create a UDP packet by assembling it in a UDP system transmit buffer allocated by the application. This function configures the UDP packet header.

Parameters:

Type	Name	Description
BYTE	buffer	UDP transmit buffer to be used by the system. This buffer must be 42 bytes larger than the data you want to transmit and aligned on a 16 byte boundary.
unsigned char *	pData	Pointer to data to be transmitted in the UDP packet. This data is concatenated to any existing data already added to the buffer.
WORD	len	Number of data bytes.

Returns:

Nothing

11.9 PostSendBufferedUdp

Synopsis:

```
void PostSendBufferedUdp( BYTE * buffer, OS_SEM * pSem );
```

Description:
Used in conjunction with `PreSendBufferedUdp()` and `AddBufferedUdpData()` to create a UDP packet by assembling it in a UDP system transmit buffer allocated by the application. This function finalizes and transmits the UDP packet.

Parameters:

Type	Name	Description
BYTE	buffer	UDP transmit buffer to be used by the system. This buffer must be 42 bytes larger than the data you want to transmit and aligned on a 16 byte boundary.
OS_SEM *	pSem	Pointer to a RTOS semaphore used to determine when the system has released the UDP transmit buffer.

Returns:

Nothing

Revision History

Revision	Date	Description
1.0	11/30/2009	Initial release
1.1	1/6/2010	Added HTTP Passwords section Added Web Form Post section
1.2	2/22/2010	Clarified description of packetization settings to indicate the settings only apply to the outgoing serial to Ethernet direction.