AN1108

Microchip TCP/IP Stack with BSD Socket API

Author: *Abdul Rafiq*
Microchip Technology Inc.

INTRODUCTION

The Microchip TCP/IP Stack with BSD (Berkley Socket Distribution) Socket API provides the socket library for Internet TCP/IP communications. The generic socket programming interface was originally developed by University of California at Berkeley. Many popular operating systems such as Microsoft® Windows®, UNIX®, Linux®, eCOS™, and many commercial TCP/IP stacks support BSD socket API. With a common programming interface, applications can now be ported easily across completely different platforms. For example, network applications written for a PC environment can also be compiled in an embedded environment, provided the embedded platform supplies the BSD library API.

This application note describes the Microchip TCP/IP stack with BSD socket API. It is intended to serve as a programmer's reference guide. Topics discussed in this application note include:

- Creating client/server applications in an embedded environment
- TCP/IP stack components and design
- Building the stack
- Socket functions included in the API

ASSUMPTION

The author assumes that the reader is familiar with the Microchip MPLAB® IDE, MPLAB® REAL ICE™ in-circuit emulator, C programming language, and socket programming. Terminology from these technologies is used in this document, and only brief overviews of the concepts are provided. Advanced users are encouraged to read the associated specifications.

FEATURES

The TCP/IP Stack with BSD socket API incorporates these main features:

- Concurrent server support
- Application can be a server or a client, or both
- Optimized for embedded applications
- Full duplex communication
- Stream and datagram socket support
- IP address resolution done in background
- Can be used with or without a kernel/RTOS

LIMITATIONS

The stack is designed for the embedded PIC®-based platform, so there are some inherent limitations associated with the embedded environment. The limitations include:

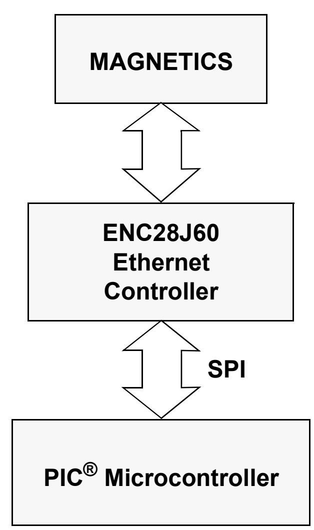
- The socket API implements a subset of the original BSD socket library.
- The behavior of the API function may differ slightly from the BSD library.
- All API functions are non-blocking.

SYSTEM HARDWARE

The Microchip TCP/IP stack with BSD socket is developed on the Microchip Explorer 16 platform. The network chip is a Microchip ENC28J60, a 10 Mbps integrated MAC/PHY Ethernet controller. The stack can easily be ported to other PIC microcontrollers. Microchip will release updates of the stack as new PIC microcontrollers are released.

A block diagram of the Microchip TCP/IP stack with BSD socket API is presented in Figure 1.

FIGURE 1: MICROCHIP TCP/IP STACK HARDWARE BLOCK DIAGRAM



PIC® Memory Resource Requirements

With a minimal application program, i.e., the associated demo application included with this application note, the stack currently consumes Flash and RAM memory as follows:

Flash: 23649 bytes
RAM: 2944 bytes

These numbers are derived using the Microchip MPLAB® C32 C Compiler, version 1.0, with the default compiler and linker settings. For further optimization, please refer to Optimizations section of the MPLAB C32 C Compiler.

PIC® Hardware Resource requirements:

The following table lists the I/O pins that interface between the Microchip TCP/IP stack and the ENC28J60 Ethernet controller. These ports are defined in `eTCP.def`, which can be modified for user customization.

TABLE 1: TCP/IP AND ENC28J60 INTERFACE PINS

PIC® I/O Pin	Ethernet Controller Pin
RD0 (Output)	Chip Select
RE8 (Input)	WOL (not used)
RE9 (Input)	Interrupt
RD15 (Output)	Reset
RF6 (Output)	SCK
RF7 (Input)	SDO
RF8 (Output)	SDI

INSTALLING SOURCE FILES

The complete source code for the Microchip TCP/IP stack is available for download from the Microchip web site (see **"Source Code"** on page 40). The source code is distributed in a single Windows installation file, `pic32mx_bsd_tcp_ip_v1_00_00.zip`.

Perform the following steps to complete the installation:

1. Unzip the compressed file and extract the installation `.exe` file.
2. Execute the file. (A Windows installation wizard will guide you through the installation process.)
3. Click **I Accept** to consent to the software license agreement.
4. After the installation process is completed, the **TCP/IP Stack with BSD Socket API** item is available under the **Microchip** program group. The complete source code will be copied into the `pic32_solutions` directory in the root drive of your computer.
5. Refer to the file `version.log` for the latest version-specific features and limitations.

The following table lists the source and header files for the stack.

TABLE 2: TCP/IP STACK HEADER AND SOURCE FILES

Source Files	Header Files	Directory	Description
earp.c	earp.h	pic32_solutions\microchip\bsd_tcp-ip\source	ARP packet handler functions
block_mgr.c	block_mgr.h	pic32_solutions\microchip\bsd_tcp-ip\source	Memory block manager functions
ENC28J60.c	ENC28J60.h MAC.h	pic32_solutions\microchip\bsd_tcp-ip\source	Driver for Microchip Ethernet controller
ether.c	ether.h	pic32_solutions\microchip\bsd_tcp-ip\source	Ethernet (MAC) layer handler
gpfunc.c	gpfunc.h	pic32_solutions\microchip\bsd_tcp-ip\source	Utility functions
eicmp.c	—	pic32_solutions\microchip\bsd_tcp-ip\source	ICMP (Ping) message handlers
eip.c	eip.h	pic32_solutions\microchip\bsd_tcp-ip\source	IP layer functions
etcp.c	etcp.h	pic32_solutions\microchip\bsd_tcp-ip\source	TCP layer functions
eudp.c	eudp.h	pic32_solutions\microchip\bsd_tcp-ip\source	UDP layer functions
pkt_queue.c	pkt_queue.h	pic32_solutions\microchip\bsd_tcp-ip\source	Generic packet FIFO queue implementation
route.c	route.h	pic32_solutions\microchip\bsd_tcp-ip\source	MAC to IP (or IP to MAC) mapping functions
socket.c	sockAPI.h socket.h	pic32_solutions\microchip\bsd_tcp-ip\source	Socket API implementation functions
StackMgr.c		pic32_solutions\microchip\bsd_tcp-ip\source	Core stack process manager
Tick.c	Tick.h	pic32_solutions\microchip\bsd_tcp-ip\source	Timer Tick functions for generating time-out events
—	Compiler.h	pic32_solutions\microchip\bsd_tcp-ip\source	Compiler-dependent typedefs
—	eTCPcfg.h	pic32_solutions\microchip\bsd_tcp-ip\source	Stack-wide configuration to customize user application
—	NetPkt.h	pic32_solutions\microchip\bsd_tcp-ip\source	TCP/IP packet structure for stack.
eTcp.def		pic32_solutions\microchip\bsd_tcp-ip\source	TCP/IP stack user configurations

DEMO APPLICATION

The demo applications included with this application note provide example client and server applications that use stream socket. A datagram-socket example application is also included in the demo. Refer to **"System Hardware"** on page 1 for information on setting up the target hardware. In addition to the target platform, you will need a host computer running Microsoft Windows for executing the demo applications. The host-side demo applications use various APIs to interface with the target hardware. Make sure the host and the target board are on the same network. Before building the application, review the section **"Stack Configuration"** on page 9 and make appropriate changes to the #defines in the `eTCP.def` configuration file to customize the settings for your particular setup. At minimum, you will need to change `DEFAULT_IP_ADDR` so that your PC will be able to establish connection with the target hardware. If you are using DHCP, the IP address will be assigned by the DHCP server.

If you plan to exercise client connection from the PIC® microcontroller to the server application running on the PC, you will also need to update the IP address of your PC in the `bsd_socket_demo\source\main.c` source file. Look for the #define `PC_SERVER_IP_ADDR` and change it to the IP address of the PC on which you plan to run the PC-side application software. You can see the IP address of your PC by executing `ipconfig.exe` at the command prompt on the PC.

Building Demo Firmware

The demo applications included with this application note are built using the MPLAB C32 C Compiler. The following is a high-level procedure for building demo applications. This procedure assumes that you will be using MPLAB IDE to build the applications and are familiar with MPLAB IDE. If not, refer to online MPLAB IDE Help to create, open, and build a project.

1. Make sure that the source files for the Microchip stack are installed. If not, refer to **"Installing Source Files"** on page 3.
2. Launch MPLAB IDE and open the project file: `bsd_socket_demo\bsd_socket_demo.mcw`
3. Open `eTCP.def` and configure the SPI port and other defines to match your target board. If you are using the Explorer 16 with ENC28J60 board, you can leave the settings in their default state.
4. Configure the default IP address, IP Mask, MAC address, and Gateway address in `eTSP.def`. Make sure these defines match your network settings. If you plan to use a DHCP server, you can leave these settings unchanged.
5. Use MPLAB IDE menu commands to build the project.
6. The build process should finish successfully. If not, make sure that your MPLAB IDE and MPLAB C32 C Compiler are set up properly.

Refer to **"Installing Source Files"**, to see the list of all source files.

Programming and Running the Demo Firmware

To program a target board with the demo application, you must have access to a PIC microcontroller programmer. The following procedure assumes that you will be using MPLAB REAL ICE in-circuit emulator as a programmer. If not, refer to the instructions for your specific programmer.

1. Connect the MPLAB REAL ICE in-circuit emulator to the Explorer 16 board or to your target board.
2. Apply power to the target board.
3. Launch the MPLAB IDE.
4. Select the PIC device of your choice (this step is required only if you are importing a hex file that was previously built).
5. Enable the MPLAB REAL ICE in-circuit emulator as your programming tool.
6. If you want to use a previously built hex file, import the project hex file.
7. If you are rebuilding the hex file, open the appropriate demo project file and follow the build procedure to create the application hex file.
8. Select the Program menu option from the MPLAB REAL ICE in-circuit emulator menu to begin programming the target.
9. After a few seconds, you should see the message "Programming successful". If not, check your board and your MPLAB REAL ICE connection. Click Help on the menu bar for further assistance.
10. Remove power from the board and disconnect the MPLAB REAL ICE cable from the target board.
11. Reapply power to the board. Make sure the board is connected to your Ethernet network that also has a DHCP server running. The demo firmware should start executing. The yellow LED on the ENC28J50 daughter board should be blinking.

Testing the TCP/IP Network

Before executing the demo programs you can test the TCP/IP communication link between the host and the target board by using the "ping" utility. On a command prompt window of your host computer, type the word "ping" followed by the IP address of the target board, as shown in the example below:

```
C:\> ping 10.10.33.201
```

The above example will send ping requests to the target board at the IP address 10.10.33.201. If the network setting is correct and the target is on the same network, you should see reply messages printed on the command prompt. If the target board is not reachable, you will see timeout messages for each ping request. In the event you don't see the ping replies, make sure the LEDs are working as described in section **"Programming and Running the Demo Firmware"**. Also, check your network settings, including the IP addresses and the subnet mask. If your computer and the board are separated by a router, make sure the board is configured with correct router gateway address.

Executing the Demo Applications

On the host computer side, there are three applications that demonstrate the various connection interfaces to the demo application on the target board.

- TCPClientAPP
- TCPServerAPP
- UDPApp

The host software programs are saved under the folder named `PCTestCode` in your main installation folder. Launch the program executable file.

The following subsections describe each of the demo applications in more detail.

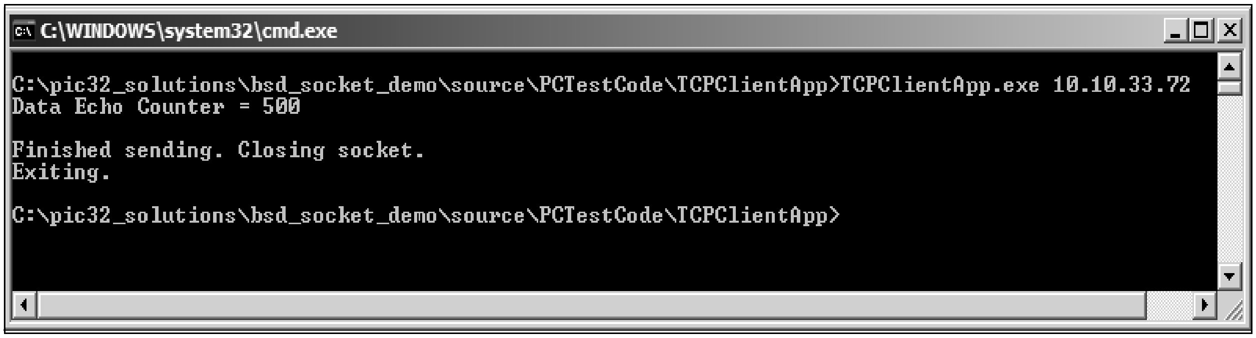
TCPCLIENTAPP

The TCPClientAPP is a Windows based TCP client application that connects to the demo TCP Server application running on the target board. To run this program, open a command prompt and type in the program name followed by the IP address of the target board.

```
C:\> TCPClientApp 10.10.33.201
```

The application will try to connect to the demo TCP server running in the target board. Once connected, the application will interchange 50 messages, using a TCP socket. The output of the program is shown in Figure 2.

FIGURE 2: TCPCLIENTAPP PROGRAM OUTPUT



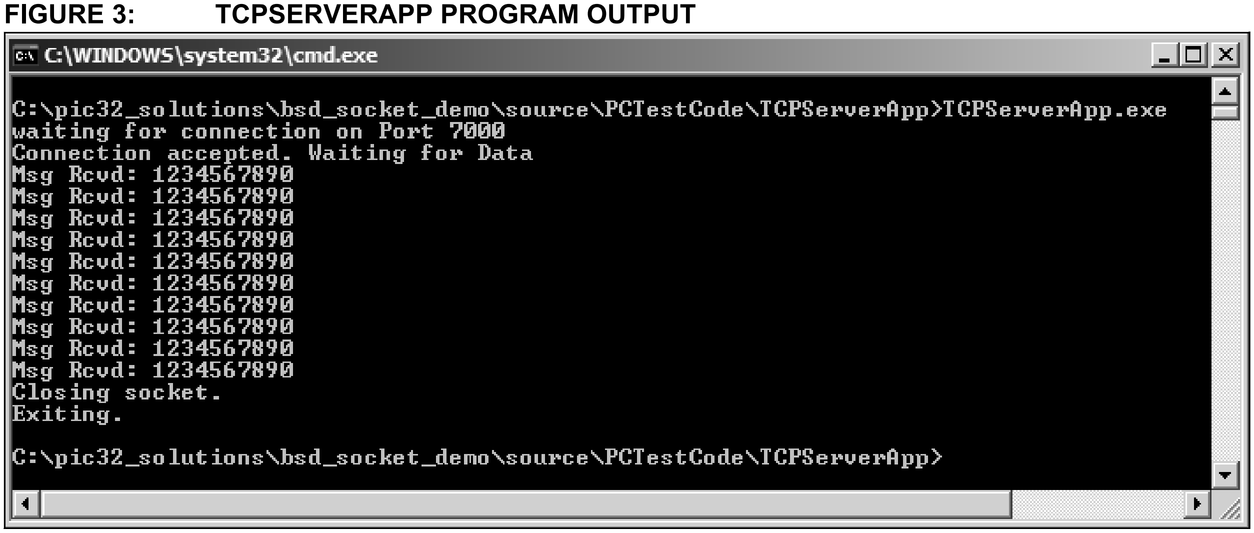
```
C:\WINDOWS\system32\cmd.exe

C:\pic32_solutions\bsd_socket_demo\source\PCTestCode\TCPClientApp>TCPClientApp.exe 10.10.33.72
Data Echo Counter = 500

Finished sending. Closing socket.
Exiting.

C:\pic32_solutions\bsd_socket_demo\source\PCTestCode\TCPClientApp>
```

TCPSERVERAPP

The TCPServerAPP is a Windows based TCP server application that accepts connection from the demo client application running on the target board. Before running this program, make sure you have copied the IP address of your computer to the `demo.c` source file.

The `#define PC_SERVER_IP_ADDR` declared in `demo.c` must be set to the IP address of your computer. When you launch the server application in a command prompt, the client application running inside the target board will connect to it and exchange data. The program output is shown in Figure 3:

FIGURE 3: TCPSERVERAPP PROGRAM OUTPUT

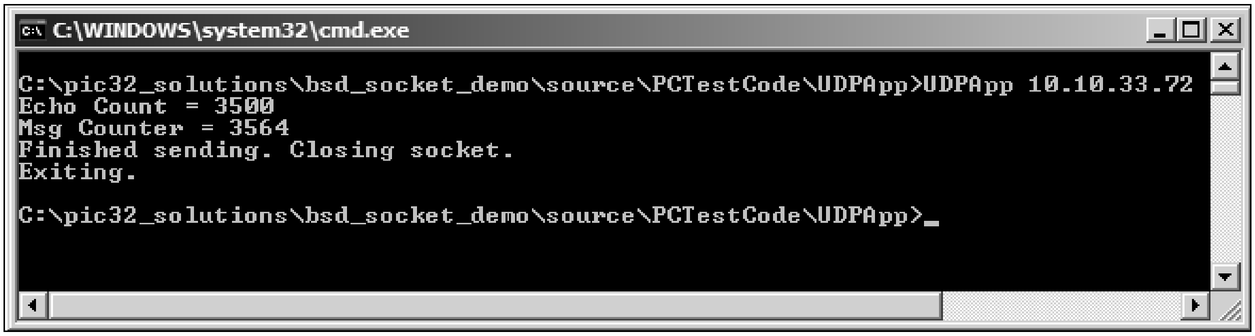


```
C:\WINDOWS\system32\cmd.exe

C:\pic32_solutions\bsd_socket_demo\source\PCTestCode\TCPServerApp>TCPServerApp.exe
waiting for connection on Port 7000
Connection accepted. Waiting for Data
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Msg Rcvd: 1234567890
Closing socket.
Exiting.

C:\pic32_solutions\bsd_socket_demo\source\PCTestCode\TCPServerApp>
```

UDPAPP

The UDPApp is a Windows based UDP application that exchanges messages over a Datagram-type socket with the demo firmware on the target board. With this application, the PC acts as a client, whereas the target board acts as the server. Figure 4 shows an example of running the program. The target board in this case has the IP address of 10.10.33.201.

FIGURE 4: UDPAPP PROGRAM OUTPUT



INTEGRATING YOUR APPLICATION

In order to integrate the stack into a user application, there are two methods that you can use. The easiest method is to start with the PIC32 demo application MPLAB project provided with this application note. You can modify the `main.c` file to add your application-specific code and include your other `.c` and `.lib` files in this project. If this method is not feasible, you can include the files for the BSD stack in your existing project.

You will modify the `main()` function to do the initialization and execution of the stack as follows:

```
#include <plib.h>
#include "bsd_dhcp_client\dhcp.h"
...
main()
{
    ...
    SetupDefault_IP_MAC();
    MSTimerInit(36000000);
    InitStackMgr();
    TickInit();
    DHCPInit();
    ...
    while(1)
    {
        StackMgrProcess();
        DHCPTask();
        ...
        ...
    }
    ...
}
```

For more information, see **"Using the Stack"** on page 16.

STACK CONFIGURATION

The stack provides several configuration options to customize it for your application. The configuration options are defined in the file `eTCP.def`. Once any option is changed, the stack must be clean built.

The following options can be used to customize the stack.

DEFAULT_IP_ADDR

Purpose: To set the IP address of the device at startup.

Precondition: None.

Valid Values: Must be a quoted ASCII string value with IP address in dotted-decimal format.

Example:

```
#define DEFAULT_IP_ADDR    "10.10.33.201"
```

DEFAULT_MAC_ADDR

Purpose: Set the MAC address of the device at startup (default value).

Precondition: None.

Valid Values: Must be a quoted ASCII string value with dash formatted MAC address.

Example:

```
#define DEFAULT_MAC_ADDR   "00-04-a3-00-00-00"
```

DEFAULT_IP_GATEWAY

Purpose: To set the IP address of the Router/Gateway.

Precondition: None.

Valid Values: Must be a quoted ASCII string value with IP address in dotted-decimal format.

Example:

```
#define DEFAULT_IP_GATEWAY "10.10.33.201"
```

DEFAULT_IP_MASK

Purpose: To set the IP address mask of the device at startup.

Precondition: None.

Valid Values: Must be a quoted ASCII string value with the mask in dotted-decimal format.

Example:

```
#define DEFAULT_IP_ADDR    "255.255.255.0"
```

CLOCK_FREQ

Purpose: To define peripheral clock frequency. This value is used by `Tick.c` and `Debug.c` files to calculate `TMR0` and `SPBRG` values. If required, you may also use this in your application.

Precondition: None.

Valid Values: Must be within the PIC32 peripheral clock specification.

Example:

```
#define CLOCK_FREQ  36000000 //define CLOCK_FREQ as 36 MHz,
```

MAX_SOCKET

Purpose: To setup the maximum number of sockets that can be opened by the application. Each socket takes up to 80 bytes of memory storage. When defining `MAX_SOCKET`, make sure your system can support the memory requirements for the entire socket array.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define MAX_SOCKET    8
```

TICKS_PER_SECOND

Purpose: To define the number of ticks in one second.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define TICKS_PER_SECOND   10
```

TX_COLLSION_TIMEOUT

Purpose: For a B1 errata for the ENC28J60, you will need to provide a transmit collision time out. It is recommended that it be somewhere between .5 and .20 times the `TICKS_PER_SECOND`. For more information about this errata, refer to the ENC28J60 errata documentation.

Precondition: None.

Valid Values: .5 and .20 times the `TICKS_PER_SECOND`

Example:

```
#define TX_COLLSION_TIMEOUT((WORD)(TICKS_PER_SECOND * .15))
```


TCP_DEFAULT_TX_BFR_SZ

Purpose: To define the default buffer size when sending data using stream socket. This number can be overridden by using `setsocketopt` API function.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define TCP_DEFAULT_TX_BFR_SZ     80   //Set the default Tx Buffer to 80 bytes.
```

NAGGLES_TX_BFR_TIMEOUT

Purpose: To define the number of ticks to detect the timeout condition when the stack is holding a transmit buffer. At the time-out expiration, the stack transmits the data packet. To maximize use of available bandwidth, the stack tries to fill the TX buffers as much as is possible. This time-out value lets you set how long the stack will wait before it sends the current TX packet for transmission.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define NAGGLES_TX_BFR_TIMEOUT(TICKS_PER_SECOND * 0.2)
```

ARP_TIMEOUT_TICKS

Purpose: When the stack requests the remote node MAC address, this time-out value controls how much time to wait for response from the remote node. Normally the time-out value is in the range of a few seconds to twenty or thirty seconds. Note that, the longer this delay is, the longer the application will have to wait in cases when the remote node does not respond. The actual delay in seconds is `ARP_TIMEOUT_TICKS / TICKS_PER_SECOND`.

Precondition: None.

Valid Values: An integer value from 1 through the user defined timeout value.

Example: For a 50 ms time out, given ticks are set at 1000 per second.

```
#define ARP_TIMEOUT_TICKS  50 // i.e. 50 Ticks / (1000Ticks/Sec) = .050sec
```

MAX_RETRY_COUNTS

Purpose: To define the maximum number of re-transmits before a packet is discarded. Re-transmissions will occur only in TCP sockets in cases when the peer node does not acknowledge the packets.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define MAX_RETRY_COUNTS   1
```

TCP_RETRY_TIMEOUT_VAL

Purpose: To define the number of ticks between re-transmissions.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define TCP_RETRY_TIMEOUT_VAL (TICKS_PER_SECOND * 5) // 5 sec interval between
resends.
```

MAX_TCP_TX_PKT_BFR

Purpose: The stack buffers the TCP transmit packets until acknowledged by the peer. This option defines the maximum number of packets that can be buffered. Once the packet count reaches this number, subsequent `SEND` API calls return the error code `SOCKET_TX_NOT_READY`..

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define MAX_TCP_TX_PKT_BFR  2
```

TCP_WAIT_SOCKET_DEL

Purpose: To define the number of ticks to wait before deleting the socket that is not properly closed by the peer. Only operates on the socket on which the application has already called `closesocket` API.

Precondition: None.

Valid Values: An integer value from 1 through 65535.

Example:

```
#define TCP_WAIT_SOCKET_DEL       (TICKS_PER_SECOND * 7)     // Set timeout to 7 sec
```

STACK_MGR_RX_PKT_CNT

Purpose: To define the maximum number of receive packets to process when `StackMgrProcess()` API is called. If you are expecting to process large incoming packet traffic, you can set it to a higher value, provided you have the buffer available to copy the packets from the Ethernet controller into RAM.

Precondition: None.

Valid Values: An integer value from 1 through 255.

Example:

```
#define STACK_MGR_RX_PKT_CNT       3
```

ENABLE_HEAP_PKT_ALLOCATION

Purpose: To enable stack software to allocate transmit and receive packet buffers on the heap, in addition to the static memory.

Precondition: None.

Valid Values: N/A.

Example:

```
#define ENABLE_HEAP_PKT_ALLOCATION
```

MAX_HEAP_PKTS

Purpose: To set the maximum number of packets that can be allocated on the heap.

Precondition: Must also define `ENABLE_HEAP_PKT_ALLOCATION`.

Valid Values: An integer value from 1 through the maximum number of packets the target memory can support.

Example:

```
#define MAX_HEAP_PKTS 10   //Allows max of 10 packets to be allocated on heap.
```

MGR_BLOCK_SIZE

Purpose: To define the memory block size to hold TCP/IP packets. Total memory allocated is MGR_BLOCK_SIZE times MGR_MAX_BLOCKS. This memory is allocated statically. The static memory manager creates a pool of memory blocks of MGR_BLOCK_SIZE size. The stack first tries to allocate the packet in one or more of these blocks. If this is not possible, the stack will try to malloc the packet memory if heap usage is enabled by the ENABLE_HEAP_PKT_ALLOCATION define.

Precondition: None.

Valid Values: An integer value from 25 through the maximum size of packet the target memory can support. As the majority of the packets are likely to be under 100 bytes, the stack will efficiently use these blocks as part of a memory pool to process the message stream.

Example:

```
#define MGR_BLOCK_SIZE 50  //defines the size of memory block to be 50 bytes.
```

MGR_MAX_BLOCKS

Purpose: To define the maximum number of blocks to allocate for the packet static memory pool. Total memory allocated is MGR_BLOCK_SIZE times MGR_MAX_BLOCKS. The static memory manager creates a pool of memory blocks of MGR_BLOCK_SIZE size. The stack first tries to allocate the packet in one or more of these blocks. If this is not possible, the stack will try to malloc the packet memory, if heap usage is enabled by the ENABLE_HEAP_PKT_ALLOCATION #define.

Precondition: None.

Valid Values: An integer value from 1 through the maximum size of packet the target memory can support.

Example:

```
#define MGR_MAX_BLOCKS16   //defines max of 16 blocks for static packet allocation
```

mSetUpCSDirection

Purpose: Sets up ChipSelect port pin for ENC28J60 as output.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetUpCSDirection()     {   mPORTDSetPinsDigitalOut(BIT_14);  } // RD14 as CS
```

mSetCSOn

Purpose: Turn on (assert) ChipSelect for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetCSOn()              {   mPORTDClearBits(BIT_14); }
```

mSetCSOff

Purpose: Turn off (deassert) Chip Select for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetCSOff()             {   mPORTDSetBits(BIT_14);   }
```

mSetUpResetDirection

Purpose: Sets up Reset port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetUpResetDirection()  {   mPORTDSetPinsDigitalOut(BIT_15);  } // RD15=Reset
```

mSetResetOn

Purpose: Turns on (Assert) Reset port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetResetOn()            {   mPORTDClearBits(BIT_15); }
```

mSetResetOff

Purpose: Turns off (deassert) Reset port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetResetOff()           {   mPORTDClearBits(BIT_15); }
```

mSetResetOn

Purpose: Turns on (Assert) Reset port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetResetOn()            {   mPORTDClearBits(BIT_15); }
```

mSetUpWOLDirection

Purpose: Sets up the Wake-On-LAN port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetUpWOLDirection()  {   mPORTESetPinsDigitalIn(BIT_8);  } // RE8=WOL
```

mSetUpINTDirection

Purpose: Sets up the Interrupt port pin for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetUpINTDirection()  {   mPORTESetPinsDigitalIn(BIT_9);  } // RE9=INT
```

mSetSPILatch

Purpose: Sets up the SPI signal routing on the Explorer16 board for ENC28J60.

Precondition: None.

Valid Values: N/A.

Example:

```
#define mSetSPILatch()          {   mPORTBSetPinsDigitalOut(BIT_12 | BIT_13); mPORT-
            BSetBits(BIT_12 | BIT_13);    }
```

USING THE STACK

Creating a UDP Application

The UDP datagram communication is the simplest way to transmit and receive messages. The following code segment shows the minimum code required for exchanging data:

```
#include <p32xxxx.h>
#include <plib.h>
...
main()
{
    ...
    SOCKET s;
    struct sockaddr_in addr;
    int addrlen = sizeof(struct sockaddr_in);
    int len;
    ...

    SetupDefault_IP_MAC();
    MSTimerInit(36000000);
    InitStackMgr();
    TickInit();
    // create datagram socket //////////////////
    if( (s = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP )) == SOCKET_ERROR )
        return -1;

    // bind to a unique local port
    addr.sin_port = 7000;
    addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
    if( bind( s, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
        return -1;
    ...
    ...
    while(1)
    {
        StackMgrProcess();
        ...
        ...
        addrlen = sizeof(struct sockaddr_in);
        len = recvfrom( s, bfr, sizeof(bfr), 0, (struct sockaddr*)&addr, &addrlen);
        if( len > 0 )
        {
            // process the datagram received in bfr of size len. Sender address is in addr.
            ...

            // send a datagram reply of size 25 bytes to the sender
            sendto( s, bfr, 25, 0, (struct sockaddr*)&addr, addrlen ); //echo back
        }
        ...
        ...
    }
    ...
}
```

Creating a TCP Client Application

The TCP-based application provides a connection-oriented reliable method to exchange data with one or more remote nodes. All TCP applications are programmed to as either a server or a client. The TCP client application initiates a connection to the remote server. Once connection is made, the normal send and receive routines are used to exchange data. Before you can create the connection, you will need the server IP address and port number.

```
#include <p32xxxx.h>
#include <plib.h>
...
main()
{
    ...
    ...
    SOCKET Sock;
    struct sockaddr_in addr;
    int addrlen = sizeof(struct sockaddr_in);
    int len;
    ...
    SetupDefault_IP_MAC();
    MSTimerInit(36000000);
    InitStackMgr();
    TickInit();

    //create tcp client socket
    if( (Sock = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
        return -1;

    //bind to a unique local port
    addr.sin_port = 0;                      // Let the stack pick a unique port for us
    addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
    if( bind( Sock, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
        return -1;

    //create the server address
    addr.sin_port = SERVER_PORT;
    addr.sin_addr.S_un.S_addr = SERVER_IP;
    addrlen = sizeof(struct sockaddr);
    ClientConnected = FALSE;

    while(1)
    {

        // execute the stack process once every iteration of the main loop
        StackMgrProcess();
        ...
        ...
        // The connect process requires multiple messages to be
        // sent across the link, so we need to keep on trying until successful
        while( !ClientConnected )
            if( connect( Sock, (struct sockaddr*)&addr, addrlen ) == 0 )
                ClientConnected = TRUE;

        // Generate data to send
        ...
```

Creating a TCP Client Application (Continued)

```
        //send the data to server
        while( (len = send( Sock, bfr, msg_len, 0 )) != msg_len )
        {
            if( len == SOCKET_ERROR )
            {
                closesocket( Sock);
                return(len);
            }
        }

        ...
        ...
        //Receive data from server
        len = recv( Sock, bfr, sizeof(bfr), 0 );
        if( len > 0 )
        {
            //process the data received in bfr
            ...
            ...
        }
        else if( len < 0 ) //handle error condition
        {
            closesocket( Sock );
            Sock = INVALID_SOCK;
            ClientConnected = FALSE;
            return(len);
        }
        ...
        ...
    }
    ...
    ...
}
```

Creating a TCP Server Application

In a TCP server application, the application listens for incoming connections from TCP client applications and accepts them in sequence. Once a connection is accepted and established, client and server applications can exchange data with each other. The Microchip stack is designed to support concurrent server connections, meaning that the server application can support multiple client requests simultaneously.

```
#include <p32xxxx.h>
#include <plib.h>
...
main()
{
    ...
    SOCKET srvr, NewClientSock;
    struct sockaddr_in addr;
    int addrlen = sizeof(struct sockaddr_in);
    int len;
    ...
    ...
    SetupDefault_IP_MAC();
    MSTimerInit(36000000);
    InitStackMgr();
    TickInit();
    // create tcp server socket
    if( (srvr = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
        return -1;

    // Bind to a unique known local port.
    addr.sin_port = 6653;
    addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
    if( bind( srvr, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
        return -1;

    // Start Listening for connections on this socket.  The max
    // connection queue size is 5 elements deep.
    listen( srvr, 5 );

    while(1)
    {

        // execute the stack process once every interation of the main loop
        StackMgrProcess();
        ....
        ....
        if( NewClientSock == INVALID_SOCK )
        {
            // Check if we have a new client connection waiting
            NewClientSock = accept( srvr, (struct sockaddr*)&addr, &addrlen );
        }
        else
        {
            //receive data from this client
            len = recvfrom( NewClientSock, bfr, sizeof(bfr), 0, NULL, NULL );
            if( len > 0 )
            {
                //process data receive in bfr
                ....
                ....
```

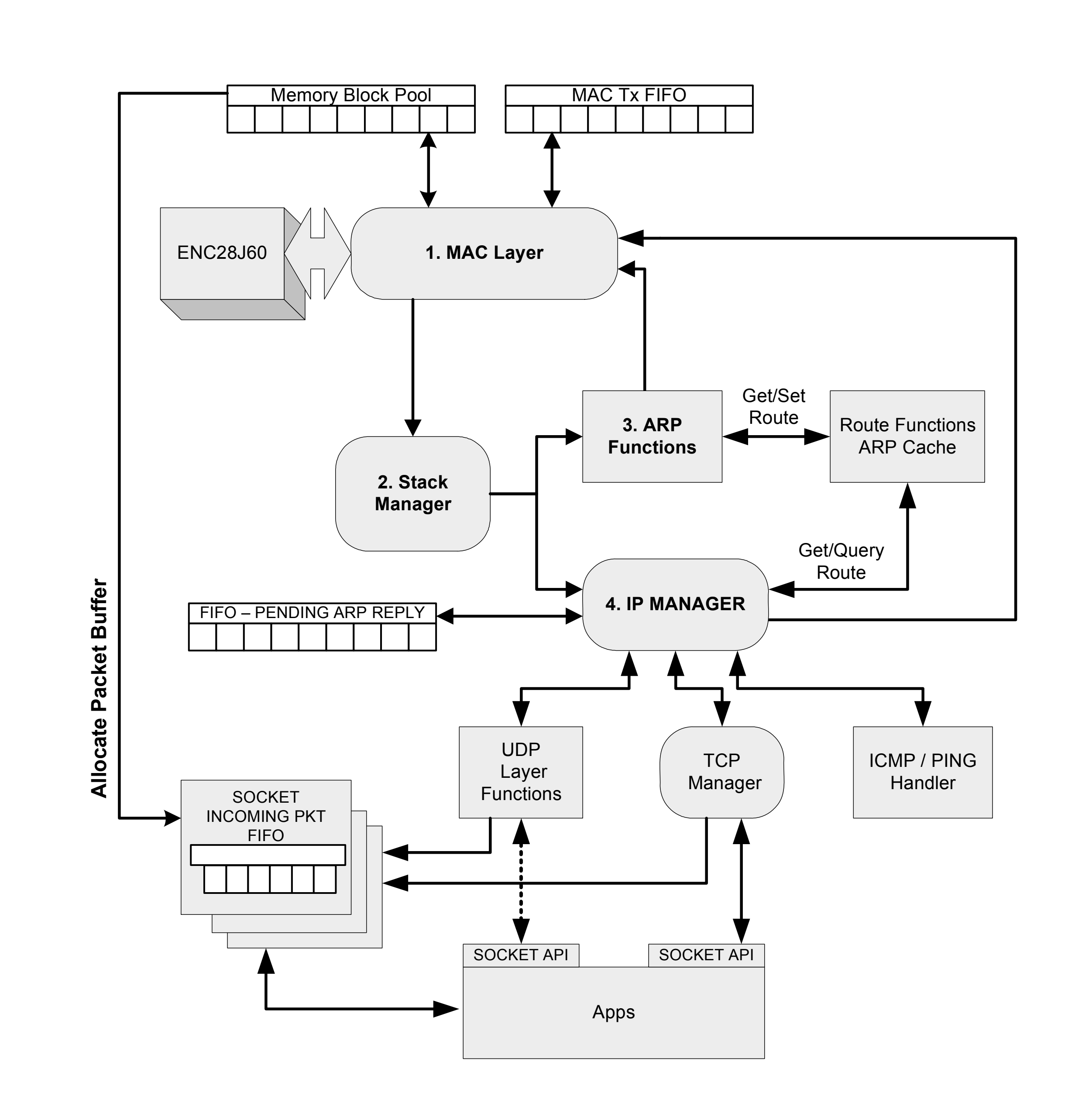
Creating a TCP Server Application (Continued)

```
            //send the reply to client
            send( NewClientSock, bfr, len, 0 );
            ....
            ....
         }
         else if( len < 0 )
         {
            closesocket( NewClientSock );
            NewClientSock = SOCKET_ERROR;
         }
      }
      ....
      ....
   }
   ....
   ....
}
```

STACK ARCHITECTURE

The Microchip BSD TCP/IP stack architecture is shown in Figure 5.

FIGURE 5: STACK ARCHITECTURE



Stack Modules and Objects

The stack is comprised of the following modules and objects:

1. Ethernet Controller Driver/MAC Layer:

   The Ethernet driver implements the low-level hardware interface for the Ethernet controller chip. It hides all the controller specific details and presents an abstract interface to higher level layers. The module retrieves messages from the Ethernet controller and packages them for high-level layers. The driver uses Memory Block Manager (MBM) to manage packet pool. A typical user application would not need to call or use MAC layer service directly.

2. Stack Manager:

   This module is responsible to execute state machines and process the message stream within the stack. The module interfaces with MAC layer to retrieve incoming packets. The manager then sends the packet to appropriate high level protocol handler. The use of stack manager allows the stack to operate independently of the application. The current version of stack manager is implemented in a cooperative multi-tasking manner. To keep the stack “alive” and execute stack-related logic, the main application must periodically call `StackMgrProcess` function. It is recommended that the main application be implemented in a cooperative multi-tasking manner and the `StackMgrProcess` be called from only main place such as main loop.

3. Address Resolution Protocol (ARP) Handler:

   The ARP handler function creates and processes packets for the ARP. The user application would not need to call any of the ARP layer functions. The IP manager automatically calls the ARP functions to initiate the address resolution process.

4. IP Manager:

   This module controls the IP layer of the packet. It receives the incoming packets from the stack manager and verifies/generates the IP checksum. All valid packets are delivered to the appropriate high-level IP protocol handlers. The current version of stack supports three IP level protocols:

   a) UDP
   b) TCP
   c) ICMP

   The IP Manager also generates ARP requests for outgoing packets. If the MAC ID of the destination node is not available, the IP Manager generates an ARP request and holds the packet in a pending ARP reply queue. When the ARP reply is received containing the MAC ID, the packet is sent to the MAC layer for immediate transmission.

MEMORY ALLOCATION SCHEME:

In order to speed up packet buffer allocation, the stack uses static memory allocation method. The MBM creates a pool of static memory blocks. One or more adjacent blocks are used to create the buffer for incoming and outgoing packets. As the blocks are allocated, they are marked as in-use. When the packet is sent or processed, the blocks are released for next packet allocation.

By way of example, with a configuration of a `MGR_BLOCK_SIZE` of 100 and a `MAX_MGR_BLOCK` of 16, the total static memory allocated will be 1600 bytes. This memory size is large enough for one maximum-sized Ethernet packet. As you have divided this 1600 bytes into 16 blocks, this means that you can buffer up to 16 packets of 100 bytes or less. The use of MBM, as opposed to byte manager, offers a very efficient method to allocate buffer for different size of packets.

API DESCRIPTION

This section describes the API functions implemented by the stack. These APIs are not a complete implementation of the BSD API. To see a complete list of BSD API functions, refer to http://en.wikipedia.org/wiki/Berkeley_sockets.

API - socket

socket creates a communication end point. The end point is called a socket. This API returns a descriptor of the socket created. The socket descriptor returned is used in all subsequent calls of the API.

The Microchip BSD socket API currently only supports UDP and TCP protocols.

Syntax

```
SOCKET socket( int af, int type, int protocol )
```

Parameters

af – Address family. Currently the only acceptable value is AF_INET

type – Defines the communication semantics. The following are the currently supported types:

SOCK_STREAM	Provides a reliable connection based data stream that is full-duplex.
SOCK_DGRAM	Support datagram. The communication is connection less, unreliable, and fixed maximum message size.

protocol – Each socket type must be used with a particular internet protocol. For stream sockets, the protocol must be IPPROTO_TCP. For datagram sockets, the protocol must be IPPROTO_UDP

Return Values

On success, the function returns a socket descriptor starting with a value of zero. On error SOCKET_ERROR is returned.

Precondition

InitStackMgr() is called

Side Effects

None

Example

```
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();

SOCKET sdesc;

sdesc = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
```


API - bind

bind assigns a name to an unnamed socket. The name represents the local address of the communication endpoint. For sockets of type SOCK_STREAM, the name of the remote endpoint is assigned when a connect or accept function is executed.

Syntax

```
int bind( SOCKET s, const struct sockaddr * name, int namelen )
```

Parameters

s – socket descriptor returned from a previous call to socket

name – pointer to the sockaddr structure containing the local address of the socket.

namelen – length of the sockaddr structure.

Return Values

If bind is successful, a value of 0 is returned. A return value of SOCKET_ERROR indicates an error.

Precondition

Socket is created by socket call

Side Effects

None

Example

```
SOCKET sdesc;

struct sockaddr_in addr;

int addrlen = sizeof(struct sockaddr_in);

SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();

sdesc = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );

addr.sin_port = 7300;  // Assign a unique known port number.

addr.sin_addr.S_un.S_addr = IP_ADDR_ANY; // accept packets from any MAC interface

if( bind( sdesc, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
       return -1;
```

API - listen

`listen` sets the specified socket in a `listen` mode. Calling the `listen` function indicates that the application is ready to accept connection requests arriving at a socket of type `SOCK_STREAM`. The connection request is queued (if possible) until accepted with an `accept` function.

The `backlog` parameter defines the maximum number of pending connections that may be queued.

Syntax

```
int listen( SOCKET s, int backlog )
```

Parameters

`s` – socket descriptor returned from a previous call to `socket`

`backlog` – maximum number of connection requests that can be queued

Return Values

If `listen` is successful, a value of `0` is returned. A return value of `SOCKET_ERROR` indicates an error.

Precondition

`bind()` API is called

Side Effects

None

Example

```
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();
// create tcp server socket
if( (srvr = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
   return -1;

// Bind to a unique known local port.
addr.sin_port = 6653;
addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
if( bind( srvr, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
   return -1;

// Start Listening for connections on this socket.  The max
// connection queue size is 5 elements deep.

if( listen( srvr, 5 ) < 0 )

    .... handle error condition
```

API - accept

`accept` is used to accept a connection request queued for a listening socket. If a connection request is pending, `accept` removes the request from the queue, and a new socket is created for the connection. The original listening socket remains open and continues to queue new connection requests. The socket `s` must be a `SOCK_STREAM` type socket.

Syntax

```
SOCKET accept( SOCKET s, struct sockaddr * addr, int * addrlen )
```

Parameters

`s` – socket descriptor returned from a previous call to socket. Must be bound to a local name and in listening mode.

`name` – pointer to the `sockaddr` structure that will receive the connecting node IP address and port number.

`namelen` – a vaue-result parameter. Should initially contain the amount of space pointed to by name; on return it contains the actual length (in bytes) of the name returned.

Return Values

If the `accept()` function succeeds, it returns a non-negative integer that is a descriptor for the accepted socket. Otherwise, the value `SOCKET_ERROR` is returned.

Precondition

`listen` API is called for the socket

Side Effects

None

Example

```
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();
// create tcp server socket
if( (srvr = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
    return -1;

// Bind to a unique known local port.
addr.sin_port = 6653;
addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
if( bind( srvr, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
    return -1;

// Start Listening for connections on this socket.  The max
// connection queue size is 5 elements deep.
listen( srvr, 5 );

while(1)
{
    // execute the stack process once every interation of the main loop
    StackMgrProcess();
    ....
    ....
    if( NewClientSock == INVALID_SOCK )
    {
        // Check if we have a new client connection waiting
        NewClientSock = accept( srvr, (struct sockaddr*)&addr, &addrlen );
    }
}
else
{
      //receive data from this client
```


API - accept (Continued)

```
SOCKET NewSock;
NewSock = accept( srvr, (struct sockaddr*)&addr, &addrlen );
if( NewSock != INVALID_SOCK )
{
     //new socket connected, send some data to this client
     send( NewSocket, ..... );
}
...
...
closesocket( NewSock );
...
```

API - connect

connect assigns the address of the peer communications endpoint. For stream sockets, connection is established between the endpoints. For datagram sockets, an address filter is established between the endpoints until changed with another connect() function.

Syntax

```
int connect( SOCKET s, struct sockaddr * name, int namelen );
```

Parameters

s – socket descriptor returned from a previous call to socket

name – pointer to the sockaddr structure containing the local address of the socket.

namelen – length of the sockaddr structure.

Return Values

If the connect() function succeeds, it returns 0. Otherwise, the value SOCKET_ERROR is returned to indicate an error condition. For stream based socket, if the connection is not established yet, connect returns SOCKET_CNXN_IN_PROGRESS.

Precondition

Socket s is created with the socket API call.

bind() is called in case of stream socket

Side Effects

None

Example

```
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();
//create tcp client socket
if( (Sock = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
    return -1;
//bind to a unique local port
addr.sin_port = 0;                          // Let the stack pick a unique port for us
addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
if( bind( Sock, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR )
    return -1;
//create the server address
addr.sin_port = SERVER_PORT;
addr.sin_addr.S_un.S_addr = SERVER_IP;
addrlen = sizeof(struct sockaddr);
ClientConnected = FALSE;

while(1)
{

    // execute the stack process once every iteration of the main loop
    StackMgrProcess();
    ...
    // The connect process requires multiple messages to be sent across the link,
    // so we need to keep on checking until successful
    while( !ClientConnected )
        if( connect( Sock, (struct sockaddr*)&addr, addrlen ) == 0 )
            ClientConnected = TRUE;
        ...
}
```

API - sendto

`sendto` is used to send outgoing data on a socket of type `datagram`. The destination address is given by `to` and `tolen`. If no memory block is available to create the datagram, the function returns an error code.

Syntax

```
int sendto( SOCKET s, const char * buf, int len, int flags, const struct sockaddr *
to, int tolen )
```

Parameters

`s` – socket descriptor returned from a previous call to socket

`buf` – application data buffer containing data to transmit

`len` – length of data in bytes

`flags` – message flags. Currently this field is not supported and must be `0`.

`to` – pointer to the `sockaddr` structure containing the destination address

`tolen` – length of the `sockaddr` structure

Return Values

On success, `sendto` returns number of bytes sent. In case of error, one of the following values is returned:

`SOCKET_BFR_ALLOC_ERROR`	No memory is available to allocate packet buffer.
`SOCKET_ERROR`	General error code. Check format of address structure and also make sure socket descriptor is correct.

Precondition

Socket `s` is created with the `socket` API call.

Connection is established in case of `stream` socket

Side Effects

None

Example

```
sendto( s, bfr, strlen(bfr)+1, 0, (struct sockaddr*)&addr, addrlen );
```

For a detailed example, please see the code in section **Creating a UDP Application.**

API - send

`send` is used to send outgoing data on an already connected socket. This function is normally used to send a reliable, ordered stream of data bytes on a socket of type `SOCK_STREAM`, but can also be used to send datagrams on a socket of type `SOCK_DGRAM`.

Syntax

```
int send( SOCKET s, const char* buf, int len, int flags );
```

Parameters

`s` – socket descriptor returned from a previous call to socket

`buf` – application data buffer containing data to transmit

`len` – length of data in bytes

`flags` – message flags. Currently this field is not supported and must be `0`.

Return Values

On success, send returns number of bytes sent. In case of error, one of the following values is returned:

`SOCKET_BFR_ALLOC_ERROR`	No memory is available to allocate packet buffer.
`SOCKET_ERROR`	General error code. Check format of address structure and also make sure socket descriptor is correct.
`SOCKET_TX_NOT_READY`	The TCP transmit functionality is temporarily disabled as the remote node sends acknowledgements for the transmitted packet by remote node.
`SOCKET_MAX_LEN_ERROR`	The maximum length of the data buffer must be less than the MTU value which for Ethernet is 1500 bytes.

Precondition

Socket `s` is created with the `socket` API call.

Connection is established in case of `stream` socket

Side Effects

None

Example

```
if( send( StreamSock, bfr, len, 0 ) < 0 )
    ... handle error condition and close socket
```

For a detailed example, please see the code in section **CREATING A TCP CLIENT APPLICATION**

API - recvfrom

recvfrom() is used to receive incoming data that has been queued for a socket. This function normally is used to receive messages on a datagrams socket, but can also be used to receive a reliable, ordered stream of data bytes on a connected socket of type SOCK_STREAM.

If from is not NULL, the source address of the datagram is filled in. The fromlen parameter is a value-result parameter, initialized to the size of the buffer associated with from and modified on return to indicate the actual size of the address stored there. This functionality is only valid for SOCK_DGRAM type sockets.

If a datagram is too long to fit in the supplied buffer buf, excess bytes are discarded in case of SOCK_DGRAM type sockets. For SOCK_STREAM types, the data is buffered internally so the application can retrieve all data in multiple calls of recvfrom.

If no data is available at the socket, recvfrom() returns 0

Syntax

```
int recvfrom( SOCKET s, char * buf, int len, int flags, struct sockaddr * from, int
* fromlen)
```

Parameters

s – socket descriptor returned from a previous call to socket

buf – application data receive buffer

len – buffer length in bytes

flags – message flags. Currently this field is not supported and must be 0.

from – pointer to the sockaddr structure that will be filled in with the destination address.

fromlen – size of buffer pointed by from.

Return Values

If recvfrom is successful, the number of bytes copied to application buffer buf is returned. A value of zero indicates no data is available. A return value of SOCKET_ERROR indicates an error condition.

Precondition

Connection is established in case of stream socket

Side Effects

None

Example

```
if( (len = recvfrom( s, bfr, sizeof(bfr), 0, (struct sockaddr*)&addr, &addrlen )) >=
0 )
    ... process data received
```

For a detailed example, please see the code in section **Creating a UDP Application.**

API - recv

`recv()` is used to receive incoming data that has been queued for a socket. This function can be used with both `datagram` and `stream` type sockets.

If the available data is too large to fit in the supplied application buffer `buf`, excess bytes are discarded in case of `SOCK_DGRAM` type sockets. For `SOCK_STREAM` types, the data is buffered internally so the application can retrieve all data by multiple calls of `recv`.

If no data is available at the socket, `recv()` returns `0`.

Syntax

```
int recv( SOCKET s, char * buf, int len, int flags )
```

Parameters

`s` – Socket descriptor returned from a previous call to socket

`buf` – application data receive buffer

`len` – buffer length in bytes

`flags` – message flags. Currently this field is not supported and must be `0`.

Return Values

If `recv` is successful, the number of bytes copied to application buffer `buf` is returned. A value of zero indicates no data available. A return value of `SOCKET_ERROR (-1)` indicates an error condition.

Precondition

Connection is established in case of `stream` socket

Side Effects

None

Example

```
if( (len = recv( s, bfr, sizeof(bfr), 0 ) >= 0 )
    .... process data received.
```

For a detailed example, please see the code in section **CREATING A TCP CLIENT APPLICATION**

API - closesocket

`closesocket` closes an existing socket. This function releases the socket descriptor `s`. Further references to `s` fail with `SOCKET_ERROR` code. Any data buffered at the socket is discarded. If the socket `s` is no longer needed, `closesocket()` must be called in order to release all resources associated with `s`.

Syntax

```
int closesocket( SOCKET s )
```

Parameters

`s` – socket descriptor returned from a previous call to socket

Return Values

If `closesocket` is successful, a value of `0` is returned. A return value of `SOCKET_ERROR (-1)` indicates an error.

Precondition

None

Side Effects

None

Example

```
len = recvfrom( clSock, bfr, sizeof(bfr), 0, NULL, NULL );
...
...
closesocket( clSock );
```

For a detailed example, please see the code in section **Creating a UDP Application.**

API - MPSocketConnected

MPSocketConnected is a custom (non-BSD) function to return the connection state of a stream type socket. For stream type sockets, the function returns 0 if the connection is established. If the connection establish process is not yet complete, the function returns SOCKET_CNXN_IN_PROGRESS. MpSocketConnected allows the application to check for connection status before assembling the packet. The equivalent BSD functions return connection status only after a packet is formed, which may be too late for many applications.

Syntax

```
int MPSocketConnected( SOCKET s )
```

Parameters

s - Socket descriptor returned from a previous call to socket

Return Values

For stream type sockets, the function returns 0 if the connection is established; and if connection establish process is not yet complete, the function returns SOCKET_CNXN_IN_PROGRESS. In all other cases, the function returns SOCKET_ERROR.

Precondition

None

Side Effects

None

Example

```
if( MPSocketConnected(s) == 0 )    //socket is connected
       send( s, bfr, len, 0 );
```

For a detailed example, please see the code in section **Creating a TCP Client Application.**

API - InitStackMgr

InitStackMgr performs the necessary initialization of all modules within the eTCPIP stack. The application must call this function once before making any other API call.

Syntax

```
void InitStackMgr()
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Example:

```
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();

TickInit();
```

API - StackMgrProcess

StackMgrProcess() executes all module tasks within the stack. The stack uses cooperative multitasking mechanism to execute multiple tasks. The application must call this function on every iteration of the main loop. Any delay in calling this function may result in loss of network packets received by the local NIC.

Syntax

```
void StackMgrProcess()
```

Parameters

None

Return Values

None

Precondition

None

Side Effects

None

Example

```
            main()
{
  ...
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();
DHCPInit();
  ..
  while(1)
  {
       StackMgrProcess();
       DHCPTask();
       ...
       ...
  }
  ...

}
```

API - setsockopt

`setsockopt` provides run-time option configuration of the stack. The supported options are `SO_SNDBUF` and `TCP_NODELAY`.

Syntax

```
int setsockopt( SOCKET s, int level, int optname,char * optval, int optlen );
```

Parameters

`s` – socket descriptor returned from a previous call to socket

`level` – must be SOL_SOCKET

`optname` – option to configure. The possible values are:

`SO_SNDBUF`	configures the `send` buffer size to use with `send` API for `tcp` sockets.
`TCP_NODELAY`	enable or disable `Naggles` algorithm for the socket. By default, `Naggles` algorithm is enabled. To turn it off, use a non-zero value for the `optval` data.

`optval` **–** pointer to the option data.

`optlen` – length of option data

Return Values

If `setsockopt` is successful, a value of `0` is returned. A return value of `SOCKET_ERROR (-1)` indicates an error condition.

Precondition

None

Side Effects

None

Example

```
...
//
SetupDefault_IP_MAC();
MSTimerInit(36000000);
InitStackMgr();
TickInit();
//create tcp client socket
if( (Sock = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP )) == SOCKET_ERROR )
   return -1;
int len = 1;
setsockopt( Sock, SOL_SOCKET, TCP_NODELAY, (char*)&len, sizeof(int) );
...
```

ANSWERS TO COMMON QUESTIONS

Q: Does the stack implement all BSD socket API functions?

A: No, the stack implements a subset of BSD API.

Q: Can I open UDP and Stream Sockets at the same time?

A: Yes, multiple sockets can be opened of any type, as long as `MAX_SOCKET` is not exceeded.

Q: Can I use the stack with an RTOS?

A: Yes, the stack can be used with RTOS or kernel. The stack is implemented as a single thread, so make sure the stack API is called from a single thread. As the API is not thread safe, the API function should be called inside a critical section if multiple threads need to call the API functions.

CONCLUSION

The stack provides a rich set of communication APIs for embedded applications using PIC micros. Through compatibility with BSD API, it creates endless possibilities for end-users to integrate their applications with the Internet, as well as with local networks. The BSD API works on buffer level, rather than byte level, and significantly improves the performance of the system, especially for 16- and 32-bit PIC microcontrollers.

REFERENCES

Jeremy Bentham, "*TCP/IP LEAN: Web Servers for Embedded Systems*", (Second Edition).
Manhasset. NY: CMP Books, 2002

W. Richard Stevens, "*TCP/IP Illustrated*". Indianapolis, IN: Addison-Wesley Professional, 1996.

N. Rajbharti, *AN833, "The Microchip TCP/IP Stack"* (DS00833). Microchip Technology Inc., 2002.

Socket API Manual Pages: Richard Verhoeven, enhanced by Michael Hamilton, "*VH-Man2html*", Computer Science @ Vassar College, http://www.cs.vassar.edu/cgi-bin/man2html?socket+2.

APPENDIX A: SOURCE CODE

The complete source code for the Microchip TCP/IP Stack with BSD Socket API, including the demo applications and necessary support files, is available under a no-cost license agreement. It is available for download as a single archive file from the Microchip corporate Web site at:

www.microchip.com

After downloading the archive, always check the `version.log` file for the current revision level and a history of changes to the software.

REVISION HISTORY

Rev. A Document (10/2007)

This is the initial released version of this document.

WORLDWIDE SALES AND SERVICE

AMERICAS

Corporate Office
2355 West Chandler Blvd.
Chandler, AZ 85224-6199
Tel: 480-792-7200
Fax: 480-792-7277
Technical Support:
http://support.microchip.com
Web Address:
www.microchip.com

Atlanta
Duluth, GA
Tel: 678-957-9614
Fax: 678-957-1455

Boston
Westborough, MA
Tel: 774-760-0087
Fax: 774-760-0088

Chicago
Itasca, IL
Tel: 630-285-0071
Fax: 630-285-0075

Dallas
Addison, TX
Tel: 972-818-7423
Fax: 972-818-2924

Detroit
Farmington Hills, MI
Tel: 248-538-2250
Fax: 248-538-2260

Kokomo
Kokomo, IN
Tel: 765-864-8360
Fax: 765-864-8387

Los Angeles
Mission Viejo, CA
Tel: 949-462-9523
Fax: 949-462-9608

Santa Clara
Santa Clara, CA
Tel: 408-961-6444
Fax: 408-961-6445

Toronto
Mississauga, Ontario,
Canada
Tel: 905-673-0699
Fax: 905-673-6509

ASIA/PACIFIC

Asia Pacific Office
Suites 3707-14, 37th Floor
Tower 6, The Gateway
Harbour City, Kowloon
Hong Kong
Tel: 852-2401-1200
Fax: 852-2401-3431

Australia - Sydney
Tel: 61-2-9868-6733
Fax: 61-2-9868-6755

China - Beijing
Tel: 86-10-8528-2100
Fax: 86-10-8528-2104

China - Chengdu
Tel: 86-28-8665-5511
Fax: 86-28-8665-7889

China - Fuzhou
Tel: 86-591-8750-3506
Fax: 86-591-8750-3521

China - Hong Kong SAR
Tel: 852-2401-1200
Fax: 852-2401-3431

China - Nanjing
Tel: 86-25-8473-2460
Fax: 86-25-8473-2470

China - Qingdao
Tel: 86-532-8502-7355
Fax: 86-532-8502-7205

China - Shanghai
Tel: 86-21-5407-5533
Fax: 86-21-5407-5066

China - Shenyang
Tel: 86-24-2334-2829
Fax: 86-24-2334-2393

China - Shenzhen
Tel: 86-755-8203-2660
Fax: 86-755-8203-1760

China - Shunde
Tel: 86-757-2839-5507
Fax: 86-757-2839-5571

China - Wuhan
Tel: 86-27-5980-5300
Fax: 86-27-5980-5118

China - Xian
Tel: 86-29-8833-7252
Fax: 86-29-8833-7256

ASIA/PACIFIC

India - Bangalore
Tel: 91-80-4182-8400
Fax: 91-80-4182-8422

India - New Delhi
Tel: 91-11-4160-8631
Fax: 91-11-4160-8632

India - Pune
Tel: 91-20-2566-1512
Fax: 91-20-2566-1513

Japan - Yokohama
Tel: 81-45-471- 6166
Fax: 81-45-471-6122

Korea - Daegu
Tel: 82-53-744-4301
Fax: 82-53-744-4302

Korea - Seoul
Tel: 82-2-554-7200
Fax: 82-2-558-5932 or
82-2-558-5934

Malaysia - Kuala Lumpur
Tel: 60-3-6201-9857
Fax: 60-3-6201-9859

Malaysia - Penang
Tel: 60-4-227-8870
Fax: 60-4-227-4068

Philippines - Manila
Tel: 63-2-634-9065
Fax: 63-2-634-9069

Singapore
Tel: 65-6334-8870
Fax: 65-6334-8850

Taiwan - Hsin Chu
Tel: 886-3-572-9526
Fax: 886-3-572-6459

Taiwan - Kaohsiung
Tel: 886-7-536-4818
Fax: 886-7-536-4803

Taiwan - Taipei
Tel: 886-2-2500-6610
Fax: 886-2-2508-0102

Thailand - Bangkok
Tel: 66-2-694-1351
Fax: 66-2-694-1350

EUROPE

Austria - Wels
Tel: 43-7242-2244-39
Fax: 43-7242-2244-393

Denmark - Copenhagen
Tel: 45-4450-2828
Fax: 45-4485-2829

France - Paris
Tel: 33-1-69-53-63-20
Fax: 33-1-69-30-90-79

Germany - Munich
Tel: 49-89-627-144-0
Fax: 49-89-627-144-44

Italy - Milan
Tel: 39-0331-742611
Fax: 39-0331-466781

Netherlands - Drunen
Tel: 31-416-690399
Fax: 31-416-690340

Spain - Madrid
Tel: 34-91-708-08-90
Fax: 34-91-708-08-91

UK - Wokingham
Tel: 44-118-921-5869
Fax: 44-118-921-5820

10/05/07