AN833

The Microchip TCP/IP Stack

Author: *Nilesh Rajbharti*
Microchip Technology Inc.

INTRODUCTION

Note: This application note was originally written for the Microchip TCP/IP Stack released back in 2002; the stack has been updated many times since. The latest API information is now provided as a Windows® Help file, `TCPIP Stack Help.chm`, which is distributed with the latest TCP/IP Stack that can be downloaded from http://www.microchip.com/tcpip. The stack now supports 8, 16 and 32-bit PIC® and dsPIC® devices. This application note is still useful as a reference material.

There is nothing new about implementing TCP/IP (Transmission Control Protocol/Internet Protocol) on Microchip microcontrollers. Interested developers can easily find many commercial and non-commercial implementations of TCP/IP for Microchip products. This application note details Microchip's own freely available implementation of the TCP/IP Stack.

The Microchip TCP/IP Stack is a suite of programs that provides services to standard TCP/IP-based applications (HTTP Server, Mail Client, etc.), or can be used in a custom TCP/IP-based application. To better illustrate this, a complete HTTP Server application is described at the end of this document and is included with the stack's source code archive.

The Microchip TCP/IP Stack is implemented in a modular fashion, with all of its services creating highly abstracted layers. Potential users do not need to know all the intricacies of the TCP/IP specifications to use it. In fact, those who are only interested in the accompanying HTTP Server application do not need any specific knowledge of TCP/IP. (Specific information on the HTTP Server starts on page 77.)

This application note does not discuss the TCP/IP protocols in depth. Those who are interested in the details of the protocols are encouraged to read the individual Request For Comment (RFC) documents. A partial list of the key RFC numbers can be found at the end of this document.

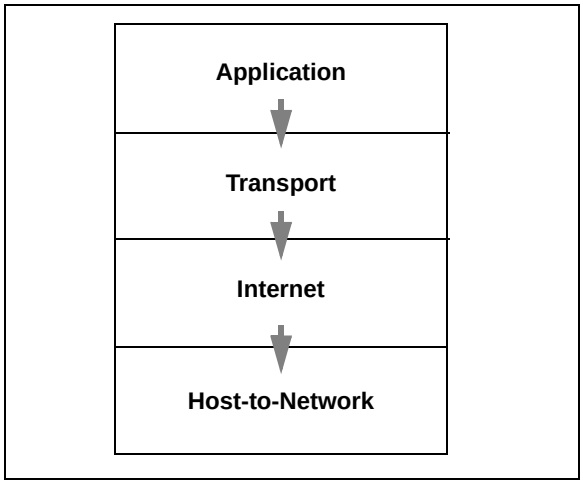
STACK ARCHITECTURE

Many TCP/IP implementations follow a software architecture referred to as the "TCP/IP Reference model". Software based on this model is divided into multiple layers, where layers are stacked on top of each other (thus the name "TCP/IP Stack") and each layer accesses services from one or more layers directly below it. A simple version of the TCP/IP Stack model is shown in Figure 1.

Per specifications, many of the TCP/IP layers are "live", in the sense that they do not only act when a service is requested but also when events like time-out or new packet arrival occurs. A system with plenty of data memory and program memory can easily incorporate these requirements. A multitasking operating system may provide extra facility and therefore, may make implementation modular. But the task becomes difficult when a system employing only an 8-bit microcontroller, with a few hundred bytes of RAM and limited program memory is used. In addition, without access to a multitasking operating system, the user must pay special attention to make the stack independent of the main application. A TCP/IP Stack that is tightly integrated with its main application is relatively easy to implement, and may even be very space efficient. But such a specialized stack may pose unique problems as more and more new applications are integrated.

The stack is written in the 'C' programming language, intended for both Microchip C18 and HI-TECH® PICC-18™ C compilers. Depending on which is used, the source files automatically make the required changes. The Microchip TCP/IP Stack is designed to run on Microchip's PIC18 family of microcontrollers only. In addition, this particular implementation is specifically targeted to run on Microchip's PICDEM.net™ Internet/Ethernet demonstration board. However, it can be easily retargeted to any hardware equipped with a PIC18 microcontroller.

FIGURE 1: LAYERS OF THE TCP/IP REFERENCE MODEL



Stack Layers

Like the TCP/IP reference model, the Microchip TCP/IP Stack divides the TCP/IP Stack into multiple layers (Figure 2). The code implementing each layer resides in a separate source file, while the services and APIs (Application Programming Interfaces) are defined through header/include files. Unlike the TCP/IP reference model, many of the layers in the Microchip TCP/IP Stack directly access one or more layers which are not directly below it. A decision as to when a layer would bypass its adjacent module for the services it needs, was made primarily on the amount of overhead and whether a given service needs intelligent processing before it can be passed to the next layer or not.

An additional major departure from traditional TCP/IP Stack implementation is the addition of two new modules: “StackTask” and “ARPTask”. StackTask manages the operations of the stack and all of its modules, while ARPTask manages the services of the Address Resolution Protocol (ARP) layer.

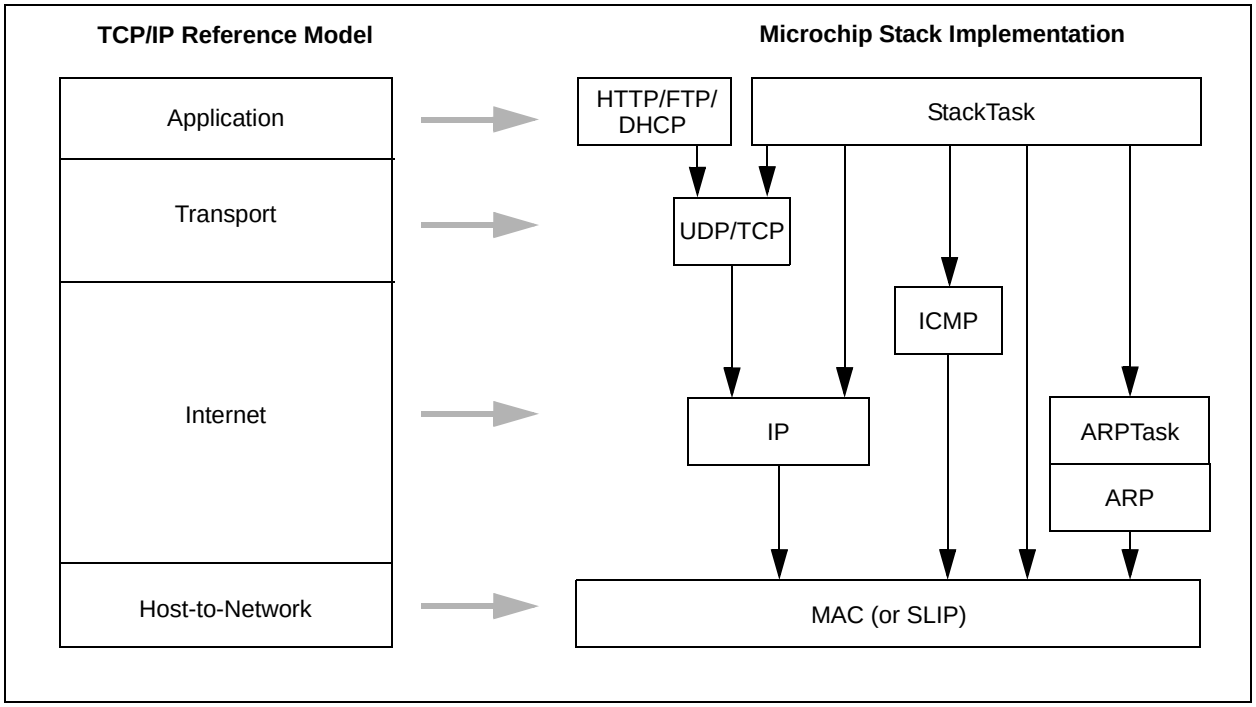
As mentioned earlier, the TCP/IP Stack is a “live” stack; some of its layers must be able to perform some timed operations asynchronously. To be able to meet this requirement and still stay relatively independent of the main application using its services, the Microchip TCP/IP Stack uses a widely known technique called *cooperative multitasking*. In a cooperative multitasking system, there is more than one task; each performs its job and returns its control so that the next task can perform its job. In this context, “StackTask” and “ARPTask” are cooperative tasks.

Usually cooperative multitasking (or any other type of multitasking, for that matter) is implemented by either the operating system, or the main application itself. The Microchip TCP/IP Stack is designed to be independent of any operating system and thus, implements its own cooperative multitasking system. As a result, it can be used in any system, regardless of whether it uses a multitasking operating system or not. However, an application utilizing the Microchip TCP/IP Stack must also use a cooperative multitasking method itself. This is done by either dividing its job into multiple tasks, or organizing its main job into a Finite State Machine (FSM) and dividing a long job into multiple smaller jobs. The HTTP Server, discussed later in this document, follows the latter paradigm, and illustrates how a cooperative application can be implemented.

Notice that the Microchip TCP/IP Stack does not implement all of the modules that are normally present in the TCP/IP Stack. Although they are not present, they can always be implemented as a separate task or module, if required.

Microchip will implement additional protocols based on this stack.

FIGURE 2: COMPARING THE MICROCHIP TCP/IP STACK STRUCTURE TO THE TCP/IP REFERENCE MODEL



STACK CONFIGURATION

Cooperative multitasking allows the user's main application to perform its own tasks without having to manage the TCP/IP Stack as well. As already noted, achieving this functionality means that all applications using the Microchip TCP/IP Stack must also be written in cooperative multitasking fashion. In addition to the cooperative multitasking design, the user must first understand some basic configuration details.

To ease the configuration process, the stack uses 'C' compiler "defines". To enable, disable or set a particular parameter, the user changes one or more of these defines. Most of these are defined in the header file, "StackTsk.h". Some defines that are defined in other files are shown with corresponding file name. Once these file are modified, the user must rebuild the application project to include the changes. The "defines" are listed in Table 1.

TABLE 1: STACK CONFIGURATION DEFINITIONS

Define	Values	Used By	Purpose
CLOCK_FREQ (compiler.h)	Oscillator Frequency (Hz)	Tick.c	Define system oscillator frequency to determine tick counter value
TICKS_PER_SECONDS	10-255	Tick.c	To calculate a second
TICK_PRESCALE_VALUE	2, 4, 8, 16, 32, 64, 128, 256	Tick.c	To determine tick counter value
MPFS_USE_PGRM	N/A	MP File System (MPFS.c)	Uncomment this if program memory will be used for MPFS storage
MPFS_USE_EEPROM	N/A	MPFS.c	Uncomment this if external serial EEPROM will be used for MPFS storage
MPFS_RESERVE_BLOCK	0-255	MPFS.c	Number of bytes to reserve before MPFS storage starts
EEPROM_CONTROL	External Data EEPROM Control Code	MPFS.c	To address external data EEPROM
STACK_USE_ICMP	N/A	StackTsk.c	Comment this if ICMP is not required
STACK_USE_SLIP	N/A	SLIP.c	Comment this if SLIP is not required
STACK_USE_IP_GLEANING	N/A	StackTsk.c	Comment this if IP Gleaning is not required
STACK_USE_DHCP	N/A	DHCP.c, StackTsk.c	Comment this if DHCP is not required
STACK_USE_FTP_SERVER	N/A	FTP.c	Comment this if FTP Server is not required
STACK_USE_TCP	N/A	TCP.c, StackTsk.c	Comment this if TCP module is not required. This module will be automatically enabled if there is at least one high-level module requiring TCP.
STACK_USE_UDP	N/A	UDP.c, StackTsk.c	Comment this if UDP module is not required. This module will be automatically enabled if there is at least one high-level module requiring UDP.
STACK_CLIENT_MODE	N/A	ARP.c, TCP.c	Client related code will be enabled
TCP_NO_WAIT_FOR_ACK	N/A	TCP.c	TCP will wait for ACK before transmitting next packet
MY_DEFAULT_IP_ADDR_BYTE? MY_DEFAULT_MASK_BYTE? MY_DEFAULT_GATE_BYTE? MY_DEFAULT_MAC_BYTE?	0-255	User Application	Define default IP, MAC, gateway and subnet mask values. Default values are: 10.10.5.15 for IP address 00:04:163:00:00:00 for MAC 10.10.5.15 for gateway 255.255.255.0 for subnet mask

TABLE 1: STACK CONFIGURATION DEFINITIONS (CONTINUED)

Define	Values	Used By	Purpose
MY_IP_BYTE? MY_MASK_BYTE? MY_GATE_BYTE? MY_MAC_BYTE?	0-255	MAC.c, ARP.c, DHCP.c, User Application	Actual IP, MAC, gateway and subnet mask values as saved/defined by application. If DHCP is enabled, these values reflect current DHCP server assigned configuration.
MAX_SOCKETS	1-253	TCP.c	To define the total number of sockets supported (limited by available RAM). Compile-time check is done to make sure that enough sockets are available for selected TCP applications.
MAX_UDP_SOCKETS	1-254	UDP.c	To define total number of sockets supported (limited by available RAM). Compile-time check is done to make sure that enough sockets are available for selected UDP applications.
MAC_TX_BUFFER_SIZE	201-1500	TCP.c, MAC.c	To define individual transmit buffer size
MAX_TX_BUFFER_COUNT	1-255	MAC.c	To define total number of transmit buffers. This number is limited by available MAC buffer size.
MAX_HTTP_CONNECTIONS	1-255	HTTP.c	To define maximum number of HTTP connections allowed at any time
MPFS_WRITE_PAGE_SIZE (MPFS.h)	1-255	MPFS.c	To define writable page size for current MPFS storage media
FTP_USER_NAME_LEN (FTP.h)	1-31	FTP.c	To define maximum length of FTP user name string
MAX_HTTP_ARGS (HTTP.c)	1-31	HTTP.c	To define maximum number of HTML form fields including HTML form name
MAX_HTML_CMD_LEN (HTTP.c)	1-128	HTTP.c	To define maximum length of HTML form URL string

USING THE STACK

The files accompanying this application note contain the full source for the Microchip TCP/IP Stack, HTTP and FTP servers, and DCHP and IP Gleaning modules. Also included is a sample application that utilizes the stack.

There are several MPLAB® project files designed to illustrate all of the different configurations in which the stack can be used. These are described in Table 2.

Since each of the modules comprising the stack resides in its own file, users must be certain to include all of the appropriate files in their project for correct compilation. A complete list of the modules and required files is presented in Table 3 (following page).

TABLE 2: STACK PROJECT FILES

Project Name	Purpose	"Defines" Used
HtNICEE.pjt	Microchip TCP/IP Stack using Network Interface Controller (NIC) and external serial EEPROM – HI-TECH® C compiler. Uses IP Gleaning, DHCP, FTP Server, ICMP and HTTP Server.	MPFS_USE_EEPROM, STACK_USE_IP_GLEANING, STACK_USE_DHCP, STACK_USE_FTP_SERVER, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
HtNICPG.pjt	Microchip TCP/IP Stack using NIC and internal program memory – HI-TECH C compiler. Uses IP Gleaning, DHCP, ICMP and HTTP Server.	MPFS_USE_PGRM, STACK_USE_IP_GLEANING, STACK_USE_DHCP, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
HtSlEE.pjt	Microchip TCP/IP Stack using SLIP and external serial EEPROM – HI-TECH Compiler. Uses FTP Server, ICMP and HTTP Server.	STACK_USE_SLIP, MPFS_USE_EEPROM, STACK_USE_FTP_SERVER, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
HtSlPG.pjt	Microchip TCP/IP Stack using SLIP and internal program memory – HI-TECH Compiler. Uses ICMP and HTTP Server.	STACK_USE_SLIP, MPFS_USE_PGRM, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
MPNICEE.pjt	Microchip TCP/IP Stack using NIC and external serial EEPROM – Microchip C18 Compiler. Uses ICMP and HTTP Server.	MPFS_USE_EEPROM, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
MPNICPG.pjt	Microchip TCP/IP Stack using NIC and internal program memory – Microchip C18 Compiler. Uses ICMP and HTTP Server.	MPFS_USE_PGRM, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
MPSlEE.pjt	Microchip TCP/IP Stack using SLIP and external serial EEPROM – Microchip C18 Compiler. Uses ICMP and HTTP Server.	STACK_USE_SLIP, MPFS_USE_EEPROM, STACK_USE_ICMP, STACK_USE_HTTP_SERVER
MPSlPG.pjt	Microchip TCP/IP Stack using SLIP and internal program memory – Microchip C18 Compiler. Uses ICMP and HTTP Server.	STACK_USE_SLIP, MPFS_USE_PGRM, STACK_USE_ICMP, STACK_USE_HTTP_SERVER

TABLE 3: MICROCHIP TCP/IP STACK MODULES AND FILE REQUIREMENTS

Module	Files Required	Purpose
MAC	MAC.c Delay.c	Media Access Layer
SLIP	SLIP.c	Media Access Layer for SLIP
ARP	ARP.c ARPTsk.c MAC.c **or** SLIP.c Helpers.c	Address Resolution Protocol
IP	IP.c MAC.c **or** SLIP.c Helpers.c	Internet Protocol
ICMP	ICMP.c StackTsk.c IP.c MAC.c **or** SLIP.c Helpers.c	Internet Control Message Protocol
TCP	StackTsk.c TCP.c IP.c MAC.c **or** SLIP.c Helpers.c Tick.c	Transmission Control Protocol
UDP	StackTsk.c UDP.c IP.c MAC.c **or** SLIP.c Helpers.c	User Datagram Protocol
Stack Manager	StackTsk.c TCP.c IP.c ICMP.c ARPTsk.c ARP.c MAC.c **or** SLIP.c Tick.c Helpers.c	Stack Manager (“StackTask”), which coordinates the other Microchip TCP/IP Stack modules
HTTP Server	HTTP.c TCP.c IP.c MAC.c **or** SLIP.c Helpers.c Tick.c MPFS.c XEEPROM.c**(1)**	HyperText Transfer Protocol Server
DHCP Client	DHCP.c UDP.c IP.c MAC.c Helpers.c Tick.c	Dynamic Host Configuration Protocol

Note 1: Required only if the external serial EEPROM for MPFS Storage option (MPFS_USE_EEPROM definition) is enabled. If selected, the corresponding MPFS image file must be included. (Refer to "MPFS Image Builder" (page 84) for additional information.)

TABLE 3: MICROCHIP TCP/IP STACK MODULES AND FILE REQUIREMENTS (CONTINUED)

Module	Files Required	Purpose
IP Gleaning	StackTsk.c ARP.c ARPTsk.c ICMP.c MAC.c or SLIP.c	To configure node IP address only.
FTP Server	FTP.c TCP.c IP.c MAC.c or SLIP.c	File Transfer Protocol Server.

Note 1: Required only if the external serial EEPROM for MPFS Storage option (MPFS_USE_EEPROM definition) is enabled. If selected, the corresponding MPFS image file must be included. (Refer to "MPFS Image Builder" (page 84) for additional information.)

Once a project is set up with the appropriate files included, the main application source file must be modified to include the programming sentences shown in Example 1. For a complete working example, refer to the source code for "Websrvr.c". This source file, along with other stack files, implements the HTTP Server.

To understand how an application specific logic is performed with cooperative multitasking, refer to the source code for HTTP.c (the HTTP Server task). Without cooperative multitasking, applications using the stack would need to divide their logic into several smaller tasks, with each being constrained from monopolizing the CPU for extended periods. The code for 'HTTP.c' demonstrates the state machine approach of dividing a long application into smaller state machine states, returning control to the main application whenever the logic must wait for a certain event.

EXAMPLE 1: CODE ADDITIONS TO THE MAIN APPLICATION

```
// Declare this file as main application file
#define THIS_IS_STACK_APPLICATION

#include "StackTsk.h"
#include "Tick.h"
#include "dhcp.h" // Only if DHCP is used.
#include "http.h" // Only if HTTP is used.
#include "ftp.h" // Only if FTP is used.
// Other application specific include files
...

// Main entry point
void main(void)
{
        // Perform application specific initialization
        ...

        // Initialize Stack components.
        // If StackApplication is used, initialize it too.
        TickInit();
        StackInit();
        HTTPInit(); // Only if HTTP is used.
        FTPInit(); // Only if FTP is used.

        // Enter into infinite program loop
        while(1)
        {
                // Update tick count. Can be done via interrupt.
                TickUpdate();

                // Let Stack Manager perform its task.
                StackTask();

                // Let any Stack application perform its task.
                HTTPServer();   // Only if HTTP is used.
                FTPServer();    // Only if FTP is used.

                // Application logic resides here.
                DoAppSpecificTask();
        }
}
```

STACK MODULES AND APIs

The Microchip TCP/IP Stack consists of many modules. In order to utilize a module, the user must review and understand its purpose and APIs. A general overview of each module, along with a description of their APIs, is provided in the following sections.

Media Access Control Layer (MAC)

The version of the Microchip TCP/IP Stack covered in this application note has been specifically written to utilize the Realtek RTL8019AS Network Interface Controller (NIC). The RTL8019AS is a NE2000 compatible NIC, that implements both the Ethernet physical (PHY) and MAC layers. If a different NIC is to be used, users will need to modify or create a new `MAC.c` file to implement access. As long as services provided by `MAC.c` are not changed, all other modules will remain unchanged.

The stack utilizes the on-chip SRAM available on the NIC as a holding buffer, until a higher level module reads it. It also performs the necessary IP checksum calculations in the NIC's SRAM buffer. In addition to the receive FIFO buffer managed by the NIC itself, the MAC layer manages its own transmit queue. Using this queue, the caller can transmit a message and request the MAC to reserve it so that the same message can be retransmitted, if required. The user can specify sizes for the transmit buffer, transmit queue and receive queue using 'C' defines (see Table 1, "Stack Configuration Definitions" for more information).

Serial Line Internet Protocol (SLIP)

SLIP layer utilizes serial cable as the main communication media, rather than ethernet cable. SLIP does not require a NIC, and thus offers very simple and inexpensive IP connection. SLIP is usually a one-to-one connection, where a host computer acts as the client. The SLIP module is designed to operate with a Windows®-based computer, though it may be modified to work with other operating systems with very few changes. With the modular design of the stack, it is only necessary to link the SLIP module (`SLIP.c`) to use it. The APIs provided by the SLIP module are essentially the same as those used by MAC (refer to the API descriptions for MAC on the following pages for details).

SLIP uses interrupt driven serial data transfer, as opposed to the polling method used by the NIC MAC. The main application must declare an interrupt handler and call the SLIP interrupt handler, `MACISR`. For additional details, refer to the source code for "`Websrvr.c`", the Web server example program included in the downloadable Zip file archive.

In order to connect to the Microchip TCP/IP Stack using the SLIP module, the host system must be configured for a SLIP connection. Refer to "Demo Applications" (page 87) for more information.

`MACInit`

This function initializes MAC layer. It initializes internal buffers and resets the NIC to a known state.

Syntax

```
void MACInit()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

All pending transmission and receptions are discarded.

Example

```
// Initialize MAC Module
MACInit();
```

MACIsTxReady

This function indicates whether at least one MAC transmit buffer is empty or not.

Syntax

```
BOOL MACIsTxReady()
```

Parameters

None

Return Values

TRUE: If at least one MAC transmit buffer is empty.

FALSE: If all MAC transmit buffers are full.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
// Check MAC transmit readiness...
if ( MACIsTxReady() )
{
       // Transmit buffer is empty, transmit a message.
...
```

MACGetHeader

This function checks the MAC receive buffer; if any packet is found, it returns the remote host and data packet information.

Syntax

```
BOOL MACGetHeader(MAC_ADDR *remote, BYTE *type)
```

Parameters

Remote [out]

Remote MAC address

type [out]

Data packet type

Possible values for this parameter are:

Value	Meaning
MAC_IP	An IP data packet is received
MAC_ARP	An ARP data packet is received
MAC_UNKNOWN	An unknown or unsupported data packet is received

Return Values

TRUE: If a data packet is received and found to be valid. All parameters are populated.

FALSE: If no data packet is received or found to be invalid.

Pre-Condition

None

Side Effects

None

Remarks

Once a data packet is fetched by calling MACGetHeader, the complete data packet must be fetched (using MACGet) and discarded (using MACDiscardRx). Users cannot call MACGetHeader multiple times to receive multiple packets and fetch data later on.

Example

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNodeMAC, &PacketType) )
{
        // A packet is received, process it.
        ...

        // Once done, discard it.
        MACDiscardRx();
...
```

MACGet

This function returns the next byte from an active transmit or receive buffer.

Syntax

```
BYTE MACGet()
```

Parameters

None

Return Values

Data byte

Pre-Condition

`MACGetHeader, MACPutHeader, MACSetRxBuffer` or `MACSetTxBuffer` must have been called.

Side Effects

None

Remarks

The active buffer (whether transmit or receive) is determined by previous calls to `MACGetHeader`, `MACPutHeader`, `MACSetRxBuffer`, or `MACSetTxBuffer` functions. For example, if `MACGetHeader` is followed by `MACGet`, the receive buffer becomes the active buffer. But if `MACPutHeader` is followed by `MACGet`, the transmit buffer becomes the active buffer.

Example 1

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
        // A packet is received, process it.
        data = MACGet();
        ...

        // When done, discard it.
        MACDiscardRx();
...
```

Example 2

```
// Load a packet for transmission
if ( MACIsTxReady() )
{
        // Load MAC header.
        MACPutHeader(&RemoteNode, MAC_ARP);

        // Get active transmit buffer.
        Buffer = MACGetTxBuffer();

        // Load all data.
        ...

        // We want to calculate checksum of whole packet.
        // Set transmit buffer access pointer to beginning of packet.
        MACSetTxBuffer(buffer, 0);

        // Read transmit buffer content to calculate checksum.
        checksum += MACGet();
        ...
...
```

MACGetArray

This function fetches an array of bytes from the active transmit or receive buffer.

Syntax

`WORD MACGetArray(BYTE *val, WORD len)`

Parameters

val [out]

Pointer to a byte array

len [in]

Number of bytes to fetch

Return Values

Total number of bytes fetched.

Pre-Condition

`MACGetHeader`, `MACPutHeader`, `MACSetRxBuffer` or `MACSetTxBuffer` must have been called.

Side Effects

None

Remarks

The caller must make sure that total number of data bytes fetched does not exceed available data in current buffer. This function does not check for buffer overrun condition.

Example

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
        // A packet is received, process it.
        actualCount = MACGetArray(data, count);
...
```

MACDiscardRx

This function discards the active receive buffer data and marks that buffer as free.

Syntax

```
void MACDiscardRx()
```

Parameters

None

Return Values

None

Pre-Condition

`MACGetHeader` must have been called and returned TRUE.

Side Effects

None

Remarks

Caller must make sure that there is at least one data packet received before calling this function. `MACGetHeader` should be used to determine whether a receive buffer should be discarded or not.

Example

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
        // A packet is received, process it.
        actualCount = MACGetArray(data, count);
        ...

        // Done processing it. Discard it.
        MACDiscardRx();
        ...
```

MACPutHeader

This function assembles the MAC header and loads it to an active transmit buffer.

Syntax

`void MACPutHeader(MAC_ADDR *remote, BYTE type, WORD dataLen)`

Parameters

remote [in]

Remote node MAC address

type [in]

Type of data packet being sent

Possible values for this parameter are:

Value	Meaning
MAC_IP	An IP data packet is to be transmitted
MAC_ARP	An ARP data packet is to be transmitted

data [in]

Number of bytes for this packet, including IP header

Return Values

None

Pre-Condition

`MACIsTxReady` must have been called and returned TRUE.

Side Effects

None

Remarks

This function simply loads the MAC header into the active transmit buffer. The caller still has to load more data and/or flush the buffer for transmission. Call `MACFlush` to initiate transmission of the current packet.

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
        // Assemble IP packet with total IP packet size of 100 bytes
        // including IP header.
        MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);
        ...
```

MACPut

This function loads the given data byte into an active transmit or receive buffer.

Syntax

`void MACPut(BYTE val)`

Parameters

val [in]

Data byte to be written

Return Values

None

Pre-Condition

`MACGetHeader, MACPutHeader, MACSetRxBuffer` or `MACSetTxBuffer` must have been called.

Side Effects

None

Remarks

This function can be used to modify either a transmit buffer or receive buffer – whichever is currently active.

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
   // Assemble IP packet with total IP packet size of 100 bytes
   // including IP header.
   MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

   // Now put the actual IP data bytes
   MACPut(0x55);
   ...
```

MACPutArray

This function writes an array of data bytes into an active transmit or receive buffer.

Syntax

`void MACPutArray(BYTE *val, WORD len)`

Parameters

val [in]

Data bytes to be written

len [in]

Total number of bytes to write

Return Values

None

Pre-Condition

`MACGetHeader, MACPutHeader, MACSetTxBuffer` or `MACSetRxBuffer` must have been called.

Side Effects

None

Remarks

None

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
   // Assemble IP packet with total IP packet size of 100 bytes
   // including IP header.
   MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

   // Now put the actual IP data bytes
   MACPut(0x55);
   MACPutArray(data, count);
   ...
```

MACFlush

This function marks active transmit buffer as ready for transmission.

Syntax

```
void MACFlush()
```

Parameters

None

Return Values

None

Pre-Condition

`MACPutHeader` or `MACSetTxBuffer` must have been called.

Side Effects

None

Remarks

None

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
   // Assemble IP packet with total IP packet size of 100 bytes
   // including IP header.
   MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

   // Put the actual IP data bytes
   MACPut(0x55);
   MACPutArray(data, count);
   ...

   // Now transmit it.
   MACFlush();
...
```

MACDiscardTx

This function discards given transmit buffer content and marks it as free.

Syntax

`void MACDiscardTx(BUFFER *buffer*)`

Parameters

buffer [in]

Buffer to be discarded

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

A transmit buffer identifier that was received by calling `MACGetTxBuffer` must be used.

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
        // Assemble IP packet with total IP packet size of 100 bytes
        // including IP header.
        MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

        // Get current transmit buffer
        buffer = MACGetTxBuffer();

        // Reserve it.
        MACReserveTxBuffer (Buffer);

        // Put the actual IP data bytes
        ...

        // Now transmit it.
        MACFlush();

        // No longer need this buffer
        MACDiscardTx(buffer);

        ...
```

MACSetRxBuffer

This function sets the access location for the active receive buffer.

Syntax

```
void MACSetRxBuffer(WORD offset)
```

Parameters

offset [in]

Location (with respect to beginning of buffer) where next access is to occur

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

Users must make sure that the supplied *offset* does not go beyond current receive buffer content. If *offset* overruns the current receive buffer, all subsequent access to that buffer would yield invalid data.

Example

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
        // A packet is received, process it.
        actualCount = MACGetArray(data, count);
        ...

        // Fetch data beginning at offset 20
        MACSetRxBuffer(20);
        data = MACGet();
...
```

MACSetTxBuffer

This function sets the access location for a given transmit buffer, and makes that transmit buffer active.

Syntax

`void MACSetTxBuffer(BUFFER *buffer*, WORD *offset*)`

Parameters

buffer [in]

A transmit buffer where this access offset be applied

offset [in]

Location (with respect to beginning of buffer) where next access is to occur

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

Users must make sure that the supplied *offset* does not go beyond current transmit buffer content. If *offset* overruns the current transmit buffer, all subsequent access to that buffer would yield invalid data.

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
        // Assemble IP packet with total IP packet size of 100 bytes
        // including IP header.
        MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

        // Get current transmit buffer
        buffer = MACGetTxBuffer();

        // Put the actual IP data bytes
        ...

        // Calculate the checksum of data packet that is being transmitted
        ...

        // Now update the checksum in this packet.
        // To update the checksum, set transmit buffer access to checksum
        MACSetTxBuffer(buffer, checksumLocation);
        ...

        // Now transmit it.
        MACFlush();

...
```

MACReserveTxBuffer

This function reserves a given transmit buffer and marks it as unavailable. This function is useful for TCP layer where a message would be queued until it is correctly acknowledged by remote host.

Syntax

`void MACReserveTxBuffer(BUFFER buffer)`

Parameters

buffer [in]

A transmit buffer to reserve

This value must be a valid transmit buffer identifier as returned by `MACGetTxBuffer` function

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
// Check to see if at least one transmit buffer is empty
if ( MACIsTxReady() )
{
        // Transmit IP packet with total IP packet size of 100 bytes
        // including IP header.
        MACPutHeader(&RemoteNodeMAC, MAC_IP, 100);

        // Get current transmit buffer
        buffer = MACGetTxBuffer();

        // Reserve it, to be discarded when ACK is received.
        MACReserveTxBuffer(buffer);

        // Put the actual IP data bytes
        ...

        // Calculate the checksum of data packet that is being transmitted
        ...

        // Now update the checksum in this packet.
        // To update the checksum, set transmit buffer access to checksum
        MACSetTxBuffer(buffer, checksumLocation);
        ...

        // Now transmit it.
        MACFlush();

        ...
```

MACGetFreeRxSize

This function returns total receive buffer size available for future data packets.

Syntax

```
WORD MACGetFreeRxSize()
```

Parameters

None

Return Values

Total number of bytes available for future data packets.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
// Get available receive buffer size
freeRxBuffer = MACGetFreeRxSize();
```

MACGetRxBuffer

This macro returns the current receive buffer identifier.

Syntax

```
BUFFER MACGetRxBuffer()
```

Parameters

None

Return Values

Active receive buffer identifier.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
// Get possible data packet info.
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
        // Get active receive buffer id.
        buffer = MACGetRxBuffer();

        // Fetch checksum directly without fetching other data.
        MACSetRxBuffer(checkLocation);
        checksum = MACGet();

...
```

MACGetTxBuffer

This macro returns the current transmit buffer identifier.

Syntax

```
BUFFER MACGetTxBuffer()
```

Parameters

None

Return Values

Active transmit buffer identifier.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
// If there is a room, load new message.
if ( MACIsTxReady() )
{
        // Load MAC header
        MACPutHeader(&RemoteNode, MAC_ARP, 100);

        // Get active transmit buffer id.
        buffer = MACGetTxBuffer();

        // Modify data byte #20 in transmit buffer
        MACSetTxBuffer(buffer, 20);
        MACPut(0x55);
...
```

Address Resolution Protocol (ARP and ARPTask)

The ARP layer of the Microchip TCP/IP Stack is actually implemented by two modules: ARP and ARPTask. ARP, implemented by the file "ARP.c", creates the ARP primitives. ARPTask, implemented by the file "ARPTsk.c", utilizes the primitives and provides complete ARP services.

ARPTask is implemented as a cooperative state machine, responding to ARP requests from the remote host. It also maintains a one-level cache to store the ARP reply and returns to a higher level when the appropriate calls are made. ARPTask does not implement a retry mechanism, so the upper level modules or applications must detect time-out conditions and respond accordingly.

ARPTask operates in two modes: *Server mode* and *Server/Client mode*. In Server/Client mode, a portion of code is enabled and compiled to generate ARP requests from the local host itself. In Server mode, the ARP request code is not compiled. Usually, if the stack is used with server applications (such as HTTP Server or FTP serve)r, ARPTask should be compiled in Server mode to reduce code size.

The compiler define STACK_CLIENT_MODE includes the client portion of code. In Server/Client mode, ARPTask maintains a one-level cache to store the ARP reply from the remote host. When Server/Client mode is not enabled, the cache is not defined and the corresponding RAM and program memory is not used.

ARP Functions:

ARPIsTxReady

This is a macro that calls MACIsTxReady in turn.

Syntax

```
BOOL ARPIsTxReady()
```

Parameters

None

Return Values

TRUE: If there is at least one transmit buffer empty.

FALSE: If there is no empty transmit buffer.

Pre-Condition

None

Side Effects

None

Remarks

This macro is provided to create an abstraction only. Rather than calling MACIsTxReady directly, the upper layer that uses ARP services should call this macro.

Example

```
// If ARP transmit buffer is ready, transmit ARP packet
if ( ARPIsTxReady() )
{
        // Transmit ARP packet.
        ARPPut(&RemoteNode, ARP_REPLY);

...
```

ARPGet

This function fetches complete ARP packet and returns necessary information.

Syntax

```
BOOL ARPGet(NODE_INFO *remote, BYTE *opCode)
```

Parameters

remote [out]

Remote node information such as MAC and IP addresses

opCode [out]

ARP code

Possible values for this parameter are:

Value	Meaning
ARP_REPLY	"ARP Reply" packet is received
ARP_REQUEST	"ARP Request" packet is received
ARP_UNKNOWN	An unknown ARP packet is received

Return Values

TRUE: If a valid ARP packet that was addressed to local host was fetched; remote and opCode contain valid values.

FALSE: Either unknown ARP code was fetched or this packet was not addressed to local host.

Pre-Condition

MACGetHeader is already called AND

Received MAC packet type == MAC_ARP

Side Effects

None

Remarks

This function assumes that the active receive buffer contains an ARP packet and access pointer to this buffer is at the beginning of ARP packet. Usually higher level layers would check MAC buffer and call this function only if an ARP packet was detected. This function fetches the complete ARP packet and releases the active receive buffer.

Example

```
// If MAC packet is received...
if ( MACGetHeader(&RemoteNode, &PacketType) )
{
   // If this is an ARP packet, fetch it.
   If ( PacketType == MAC_ARP )
   {
      // This is ARP packet.
      ARPGet(&RemoteNode, &ARPCode);

...
```

ARPPut

This function loads MAC buffer with valid ARP packet.

Syntax

```
void ARPPut(NODE_INFO *remote, BYTE opCode)
```

Parameters

remote [in]

Remote node information such as MAC and IP addresses

opCode [in]

ARP code

Possible values for this parameter are:

Value	Meaning
ARP_REPLY	Transmit this packet as “ARP Reply”
ARP_REQUEST	Transmit this packet as “ARP Request”

Return Values

None

Pre-Condition

```
ARPIsTxReady == TRUE
```

Side Effects

None

Remarks

This function assembles complete ARP packet and transmits it.

Example

```
// Check to see if transmit buffer is available
if ( ARPIsTxReady() )
{
        // Transmit it
        ARPPut(&RemoteNode, ARP_REQUEST);

...
```

ARPTask Functions

ARPInit

This function initializes the ARPTask state machine and prepares it to handle ARP requests and replies.

Syntax

```
void ARPInit()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

In Server/Client mode, this function also initializes an internal one-level cache.

Example

```
// Initialize ARPTask
ARPInit();
...
```

ARPResolve

This function sends out an ARP request to a remote host.

Syntax

`void ARPResolve(IP_ADDR *IPAddr)`

Parameters

IPAddr [in]

IP Address of remote host to be resolved

Return Values

None

Pre-Condition

`ARPIsTxReady == TRUE`

Side Effects

None

Remarks

This function is available when `STACK_CLIENT_MODE` is defined.

Example

```
// Check to see if transmit buffer is available
if ( ARPIsTxReady() )
{
        // Send ARP request
        ARPResolve(&RemoteNodeIP);

...
```

ARPIsResolved

This function checks the internal cache and returns matching host address information.

Syntax

`BOOL ARPIsResolved(IP_ADDR *IPAddr, MAC_ADDR *MACAddr)`

Parameters

IPAddr [in]

IP Address of remote host that is to be resolved

MACAddr [out]

Buffer to hold MAC Address of remote host that is to be resolved

Return Values

TRUE: If a matching IP Address was found in internal cache, corresponding MAC Address is copied to MACAddr.

FALSE: If there is no matching IP Address in internal cache; MACAddr is not populated.

Pre-Condition

None

Side Effects

An entry that matches with internal cache is removed from cache and declared as resolved.

Remarks

Since ARPTask maintains only one level of cache, higher level layer must make sure that second ARP request is not sent until previous request is resolved. This function is available when STACK_CLIENT_MODE is defined.

Example

```
// Check to see if transmit buffer is available
if ( ARPIsResolved(&RemoteIPAddr, &RemoteMACAddr) )
{
        // ARP is resolved. Continue with further connection...
        ...
}
else
{
        // Not resolved. Wait...
        ...
}
```

Internet Protocol (IP)

The IP layer of the Microchip TCP/IP Stack is implemented by the file "IP.c". The header file "IP.h" defines the services provided by the layer.

In this architecture, the IP layer is passive; it does not respond to IP data packets. Instead, higher level layers use IP primitives and fetch the IP packet, interpret it and take appropriate action.

The IP specification requires that the local host generate a unique packet identifier for each packet transmitted by it. The identifier allows remote host to identify duplicate packets and discard them. The Microchip TCP/IP Stack's IP layer maintains a private 16-bit variable to track packet identifiers.

IPIsTxReady

This is a macro that calls MACIsTxReady in turn.

Syntax

```
BOOL IPIsTxReady()
```

Parameters

None

Return Values

TRUE: If there is at least one transmit buffer empty.

FALSE: If there is no empty transmit buffer.

Pre-Condition

None

Side Effects

None

Remarks

This macro is provided to create an abstraction only. Rather than calling MACIsTxReady directly, the upper layer that uses IP services should call this macro.

Example

```
// If IP transmit buffer is ready, transmit IP packet
if ( IPIsTxReady() )
{
        // Assemble IP packet.
        IPPutHeader(&Remote, MAC_TCP, IPPacketLen);

...
```

IPSetTxBuffer

This is a macro to allow higher level layer set transmit buffer access pointer. This macro takes IP header into account before calling `MACSetTxBuffer`.

Syntax

`void IPSetTxBuffer(BUFFER *buffer*, WORD *offset*)`

Parameters

buffer [in]

Transmit buffer identifier whose access pointer is to be set

offset [in]

An offset with respect to IP Data

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

Layers that use IP services should call this macro to set access pointer for a given transmit buffer. This macro adjusts the given offset by the length of IP header.

Example

```
// If IP transmit buffer is ready, transmit IP packet
if ( IPIsTxReady() )
{
        // Assemble IP packet.
        IPPutHeader(&Remote, MAC_TCP, IPPacketLen);

        // Get current transmit buffer id.
        buffer = MACGetTxBuffer();

        // Load transmit data
        ...

        // Calculate checksum checkHi:checkLo
        ...

        // Update the checksum.
        IPSetTxBuffer(buffer, checkLocation);
        MACPut (checkHi);
        MACPut (checkLo);
        ...
```

IPPutHeader

This function assembles a valid IP header and loads it into active transmit buffer.

Syntax

WORD IPPutHeader(NODE_INFO *remote, BYTE protocol, WORD len)

Parameters

remote [in]

Remote node information such as MAC and IP addresses

protocol [in]

Protocol to use for this data packet

Possible values for this parameter are:

Value	Meaning
IP_PROT_ICMP	Assemble this packet as ICMP
IP_PROT_TCP	Assemble this packet as TCP segment
IP_PROT_UDP	Assemble this packet as UDP segment

len [in]

Total length of IP data bytes, excluding IP header

Return Values

None

Pre-Condition

IPIsTxReady == TRUE

Side Effects

None

Remarks

This function assembles an IP packet complete with header checksum and correct network byte order. After this function, the active transmit buffer access pointer points to the beginning of the IP data area.

This function does not initiate IP packet transmission. The caller must either load IP data by calling the MACPut function and/or calling MACFlush to mark the buffer as ready to transmit.

Example

```
// Check to see if transmit buffer is available
if ( IPIsTxReady() )
{
        // Load the header
        IPPutHeader(&RemoteNode, IP_PROT_ICMP, ICMP_HEADER_SIZE+dataLen);

        // Load ICMP data
        IPPutArray(ICMPData, dataLen);

        // Mark it as ready to be transmitted
        MACFlush();
...
```

IPPutArray

This macro loads an array of bytes into the active transmit buffer.

Syntax

`void IPPutArray(BYTE *buffer, WORD len)`

Parameters

buffer [in]

Data array that is to loaded

len [in]

Total number of items in data array

Return Values

None

Pre-Condition

`IPIsTxReady == TRUE`

Side Effects

None

Remarks

This macro calls `MACPutArray` function. This is provided for abstraction only.

Example

```
// Check to see if transmit buffer is available
if ( IPIsTxReady() )
{
        // Load the header
        IPPutHeader(&RemoteNode, IP_PROT_ICMP, ICMP_HEADER_SIZE+dataLen);

        // Load ICMP data
        IPPutArray(ICMPData, dataLen);

        // Mark it as ready to be transmitted
        MACFlush();
...
```

IPGetHeader

This function fetches the IP header from the active transmit buffer and validates it.

Syntax

BOOL IPGetHeader(IP_ADDR *localIP, NODE_INFO *remote, BYTE *protocol, WORD *len)

Parameters

localIP [out]

Local node information such as MAC and IP addresses

remote [out]

Remote node information such as MAC and IP addresses

protocol [out]

Protocol associated with this IP packet

Possible values for this parameter are:

Value	Meaning
IP_PROT_ICMP	This is an ICMP packet
IP_PROT_TCP	This is a TCP packet
IP_PROT_UDP	This is a UDP packet
All others	Unknown protocol

len [out]

Total length of IP data in this packet

Return Values

TRUE: A valid IP packet was received. Remote IP address, packet protocol and packet length parameters are populated.

FALSE: An invalid IP packet was received. Parameters are not populated.

Pre-Condition

MACGetHeader == TRUE

Side Effects

None

Remarks

This function assumes that the active receive buffer access pointer is positioned to the beginning of the MAC Data area. In order to satisfy this condition, the higher level layer must perform the following checks before calling this function:

```
If MACGetHeader == TRUE and PacketType == MAC_IP, call IPGetHeader
Else do not call IPGetHeader
```

Once the IP packet is processed and no longer needed, the caller must discard it from the MAC buffer by calling the MACDiscardRx function. Refer to the source code of the Stack Task Manager (StackTsk.c) for details of the actual implementation.

`IPGetHeader` (Continued)

Example

```
// Check to see if any packet is ready
if ( MACGetHeader(&RemoteMACAddr, &PacketType) )
{
        // Check what kind of protocol it is
        if ( PacketType == MAC_IP )
        {
                // This is IP packet. Fetch it.
                IPGetHeader(&Local, &Remote, &IPProtocol, &IPLen);

                // Process this IP packet.
                ...

                // When done processing this packet, discard it
                MACDiscardRx();
        }
        else
        {
                // This is not an IP packet. Handle it
...
```

IPGetArray

This macro fetches an array of bytes from an active transmit or receive buffer.

Syntax

`WORD IPGetArray(BYTE *val, WORD len)`

Parameters

val [out]

Pointer to a buffer to byte array

len [in]

Number of bytes to fetch

Return Values

Total number of bytes fetched

Pre-Condition

`IPGetHeader`, `IPPutHeader`, `IPSetRxBuffer` or `IPSetTxBuffer` must have been called.

Side Effects

None

Remarks

The caller must make sure that the total number of data bytes fetched does not exceed the available data in the active buffer. This macro does not check for buffer overrun conditions.

Example

```
// Check to see if any packet is ready
if ( MACGetHeader(&RemoteMACAddr, &PacketType) )
{
        // Check what kind of protocol it is
        if ( PacketType == MAC_IP )
        {
                // This is IP packet. Fetch it.
                IPGetHeader(&Remote, &IPProtocol, &IPLen);

                // Get 20 bytes of data
                IPGetArray(IPData, 20);
                ...

                // When done processing this packet, discard it
                MACDiscardRx();
        }
        else
        {
                // This is not an IP packet. Handle it
...
```

IPSetRxBuffer

This macro allows a higher level layer to set the receive buffer access pointer. It takes the IP header into account before calling `MACSetRxBuffer`.

Syntax

`void IPSetRxBuffer(WORD offset)`

Parameters

offset [in]

An offset with respect to IP Data

Return Values

None

Pre-Condition

None

Side Effects

None

Remark

Layers that use IP services should call this macro to set the access pointer for the current buffer. This macro adjusts the given offset by the length of IP header.

Example

```
// Check to see if any packet is ready
if ( MACGetHeader(&RemoteMACAddr, &PacketType) )
{
        // Check what kind of protocol it is
        if ( PacketType == MAC_IP )
        {
                // This is IP packet. Fetch it.
                IPGetHeader(&Remote, &IPProtocol, &IPLen);

                // Fetch 20th byte within IP data.
                IPSetRxBuffer(20);
                data = MACGet();
                ...

                // When done processing this packet, discard it
                MACDiscardRx();
        }
        else
        {
                // This is not an IP packet. Handle it
...
```

Internet Control Message Protocol (ICMP)

The ICMP layer is implemented by the file "ICMP.c". The header file "ICMP.h" defines the services provided by the layer.

In this architecture, the ICMP layer is a passive layer; it does not respond to the ICMP data packet. Instead, higher level layers use ICMP primitives and fetch the ICMP packet, interpret it and take appropriate action.

Normally, ICMP is used to send and receive IP error or diagnostic messages. In the Microchip TCP/IP Stack, ICMP implements ICMP primitives that can be used to generate any of the ICMP messages. In embedded applications, ICMP is useful for diagnostic purpose. When enabled, ICMP can respond to "ping" packets, thus allowing a remote host to determine local host presence.

The Microchip ICMP layer only responds to ping data packets of up to 32 bytes; larger packets will be ignored. If it is necessary to handle larger packets, modify the compiler define MAX_ICMP_DATA_LEN (in the header file "StackTsk.h").

ICMPIsTxReady

This macro determines if at least one transmit buffer is empty.

Syntax

```
BOOL ICMPIsTxReady()
```

Parameters

None

Return Values

TRUE: If there is at least one transmit buffer empty.

FALSE: If there is no empty transmit buffer.

Pre-Condition

None

Side Effects

None

Remarks

This macro is provided to create an abstraction only. Rather than calling MACIsTxReady directly, the upper layer that uses IP services should call this macro.

Example

```
// If IP transmit buffer is ready, transmit IP packet
if ( ICMPIsTxReady() )
{
        // Transmit ICMP packet.
...
```

ICMPPut

This function assembles a valid ICMP packet and transmits it.

Syntax

```
void ICMPPut(NODE_INFO *remote,
             ICMP_CODE code,
             BYTE *data,
             BYTE len,
             WORD id,
             WORD seq)
```

Parameters

remote [in]

Remote node information such as MAC and IP addresses

code [in]

ICMP code to be used for this ICMP packet

Possible values for this parameter are:

Value	Meaning
ICMP_ECHO_REPLY	This is an ICMP Echo reply packet
ICMP_ECHO_REQUEST	This is an ICMP Echo request packet

data [in]

ICMP data

len [in]

ICMP data length

id [in]

ICMP packet identifier

seq [in]

ICMP packet sequence number

Return Values

None

Pre-Condition

IPIsTxReady == TRUE

Side Effects

None

Remarks

This function asks IP layer to assemble IP header and assembles an ICMP packet, complete with header checksum and correct network byte order. One major difference between this and other layer "Put" functions is that this function assembles and transmits the packet in one function. Caller supplies complete data buffer and does not have to supply data as a separate function call.

Example

```
// Check to see if transmit buffer is available
if ( ICMPIsTxReady() )
{
   // Transmit ICMP packet.
   ICMPPut(&RemoteNode, ICMP_ECHO_REPLY, data, datalen, id, seq);

   // Done. ICMP is put into transmit queue.
...
```

ICMPGet

This function fetches the ICMP header from the active transmit buffer and validates it.

Syntax

```
void ICMPGet(NODE_INFO *remote,
             ICMP_CODE *code,
             BYTE *data,
             BYTE *len,
             WORD *id,
             WORD *seq)
```

Parameters

remote [out]

Remote node information such as MAC and IP addresses

code [out]

ICMP code for received ICMP packet

Possible values for this parameter are:

Value	Meaning
ICMP_ECHO_REPLY	An ICMP Echo reply packet is received
ICMP_ECHO_REQUEST	An ICMP Echo request packet is received
For all others	An unknown/unsupported packet is received

data [out]

ICMP data

len [out]

ICMP data length

id [out]

ICMP packet identifier

seq [out]

ICMP packet sequence number

Return Values

TRUE: A valid ICMP packet was received. All parameters are populated.

FALSE: An invalid ICMP packet was received. Parameters are not populated.

Pre-Condition

IPGetHeader == TRUE and PacketType == IP_PROT_ICMP

Side Effects

None

Remarks

This function assumes that the active receive buffer access pointer is positioned to the beginning of IP Data area. In order to satisfy this condition, the higher level layer must perform the following checks before calling this function:

```
If IPGetHeader == TRUE and PacketType == IP_PROT_ICMP, call ICMPGet
Else
       Do not call ICMPGet
```

Once the IP packet is processed and no longer needed, the caller must discard it from MAC buffer by calling the MACDiscardRx function.

`ICMPGet` (Continued)

Example

```
// Check to see if any packet is ready
if ( IPGetHeader(&Remote, &IPProtocol, &IPLen) )
{
        // Check what kind of protocol it is
        if ( IPProtocol == IP_PROT_ICMP )
        {
                // This is ICMPP packet. Fetch it.
                ICMPGet(&ICMPCode, data, &dataLen, &id, &seq);

                // Process this ICMP packet.
                ...

                // When done processing this packet, discard it
                MACDiscardRx();
        }
        else
        {
                // This is not an ICMP packet. Handle it
...
```

Transmission Control Protocol (TCP)

The TCP layer of the Microchip TCP/IP Stack is implemented by the file "TCP.c". The header file "TCP.h" defines the services provided by the layer. In this stack architecture, TCP is an active layer. It fetches TCP packets and responds to the remote host according to the TCP state machine. The TCP module is also implemented as a cooperative task, performing automatic operations without the knowledge of the main application.

"TCP.h" provides TCP socket services and hides all TCP packet handling from the caller. The layer allows from 2 to 253 TCP sockets, the number limited only by available memory and compiler used. With more than one socket, higher level applications can maintain multiple simultaneous TCP connections and there could be more than one application using this layer. This facility is useful when HTTP Server is used. It is important to know that each socket consumes approximately 36 bytes (check source file for actual consumption) and increases overall TCP processing time.

Unlike other TCP/IP implementations, all sockets in the Microchip TCP/IP Stack share one or more common transmit buffers. This approach reduces overall RAM requirements, but it may create a potential problem, where a few sockets reserve all available transmit buffers and do not release them on time for other sockets to use. Under these circumstances, remote hosts and/or local applications would not be able to contact the stack. To avoid this, users must make sure that there are enough transmit buffers for all sockets.

On the receive side, there is only one receive buffer. If a socket receives its data, the owner of that socket must fetch and discard the receive buffer in one task time in order for the other sockets to receive their data. This design mandates that once a task detects a packet it is interested in, it must consume the complete packet in one task time. A task cannot fetch part of a packet during one task time and expect to fetch the rest of the packet later.

As required by TCP specifications, each TCP segment contains a checksum that covers the entire TCP packet, including the data area. To reduce RAM requirements, the TCP layer uses the MAC buffer in the NIC as storage and performs the checksum calculation in the MAC buffer itself. If the NIC is used as a MAC, the NIC SRAM is used as a buffer space. But if SLIP is used as a MAC, the microcontroller's internal data RAM is used.

The TCP layer of the Microchip TCP/IP Stack implements most of the TCP state machine states proposed by RFC793. It also implements automatic retry and timed operations, which users can enable or disable by the compile time definition TCP_NO_WAIT_FOR_ACK. When automatic retry is enabled, each socket transmit buffer is reserved until an acknowledgement from the remote host is received. This design effectively creates a transmit window of one TCP segment. Thus, data throughput would be considerably lower than that in "No Retry" mode. If only the HTTP Server application is used, the user may disable automatic retry and effectively increase throughput. If the main application's logic requires that each packet be acknowledged before a new one can be transmitted, the user should enable the "Automatic Retry" mode. With Automatic Retry enabled, some opened connections may not get served on time, and the remote host may get time-out or Reset errors.

TCPInit

This function initializes the TCP state machine and prepares it for multiple TCP connections.

Syntax

```
void TCPInit()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

This function is called only once on application start-up.

Example

```
// Initialize TCP
TCPInit();
```

TCPListen

This function assigns one of the available sockets to listen on given TCP port.

Syntax

`TCP_SOCKET TCPListen(TCP_PORT port)`

Parameters

port [in]

TCP Port number on which to listen

Return Values

A valid socket identifier if there was at least one free socket.

`INVALID_SOCKET` if there is no socket available.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_LISTEN:
        // Listen for HTTP requests.
        httpSocket = TCPListen(80);
        If ( httpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
                ...
        }
        else
                smState = SM_LISTEN_WAIT;
        return;
case SM_LISTEN_WAIT:
        // Wait for connection...
...
```

TCPConnect

This function initiates a connection request to a remote host on a given remote port.

Syntax

`TCP_SOCKET TCPConnect(NODE_INFO *remote, TCP_PORT port)`

Parameters

remote [in]

Remote host that needs to be connected

port [in]

TCP Port number on remote host to connect to

Return Values

A valid socket identifier if there was at least one free socket and connection request was sent.

`INVALID_SOCKET` if there is no socket available.

Pre-Condition

None

Side Effects

None

Remarks

This function is available only when `STACK_CLIENT_MODE` is defined (see "Stack Configuration" (page 3)).

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
...
```

TCPIsConnected

This function determines whether a given socket is connected to remote host or not.

Syntax

BOOL TCPIsConnected(TCP_SOCKET *socket*)

Parameters

socket [in]

Socket identifier for which the connection is to be checked

Return Values

TRUE: If given socket is connected to remote host.

FALSE: If given socket is not connected to remote host.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(ftpSocket) )
                smState = SM_CONNECTED;
        return;
...
```

TCPDisconnect

This function requests remote host to disconnect.

Syntax

`void TCPDisconnect(TCP_SOCKET socket)`

Parameters

socket [in]

Socket identifier that needs to be disconnected

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(ftpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Send data
        ...
        // Disconnect
        TCPDisconnect(ftpSocket);
...
```

TCPIsPutReady

This function determines if a socket is ready to transmit. A socket is ready to transmit when it is connected to a remote host and its transmit buffer is empty.

Syntax

BOOL TCPIsPutReady(TCP_SOCKET *socket*)

Parameters

socket [in]

Socket identifier that needs to be checked

Return Values

TRUE: If given socket is ready to transmit.

FALSE: If given socket is not connected or there is no transmit buffer ready.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(ftpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Send data
        if ( TCPIsPutReady(ftpSocket) )
        {
                // Send data
...
```

TCPPut

This function loads a data byte into the transmit buffer for a given socket.

Syntax

`BOOL TCPPut(TCP_SOCKET socket, BYTE byte)`

Parameters

socket [in]

Socket identifier that needs to be checked

byte [in]

Data byte to be loaded

Return Values

`TRUE`: If a given data byte was successfully loaded into the transmit buffer and there is room for more data.

`FALSE`: If a given data byte was successfully loaded into the transmit buffer and there is no room for more data.

Pre-Condition

`TCPIsPutReady == TRUE`

Side Effects

None

Remarks

Once a socket is found to be ready to transmit, the user must load all data, or until there is no more room in the socket buffer. The user cannot load part of the data in one socket and load another socket buffer.

It is important to remember that when socket data is loaded using this function, the actual transmission may or may not start, depending on total number of data bytes loaded. If the number of bytes loaded is less than the available socket buffer size, the user must explicitly flush the transmit buffer using the `TCPFlush` function. If the user tries to load more bytes then the available buffer size, this function automatically starts the transmission and returns FALSE, so the user can try again. Usually, it is a good practice to flush the socket buffer after all known data is loaded into buffer, regardless of whether the buffer was full or not.

TCPPut (Continued)

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(ftpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Send data
        if ( TCPIsPutReady(ftpSocket) )
        {
                // Send data
                TCPPut(ftpSocket, dataByte);
...
```

TCPFlush

This function marks given socket transmit buffer as ready to be transmitted.

Syntax

`void TCPFlush(TCP_SOCKET *socket*)`

Parameters

socket [in]

Socket identifier that needs to transmitted

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

This function marks the current transmit buffer as ready to transmit; the actual transmission may not start immediately. User does not have to check the status of transmission. The NIC will retry transmitting a message up to 15 times (for the RTL8019AS; check the documentation for the specific NIC if the RTL8019AS is not being used). If the socket is already flushed, another flush would be ignored.

Example

```
...

switch(smState)
{
case SM_CONNECT:
        // Connect to a remote FTP server.
        ftpSocket = TCPConnect(&RemoteNode, 21);
        If ( ftpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_CONNECT_WAIT;
        return;
case SM_CONNECT_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(ftpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Send data
        if ( TCPIsPutReady(ftpSocket) )
        {
                // Send data
                TCPPut(ftpSocket, dataByte);
                ...

                // Now transmit it.
                TCPFlush(ftpSocket);
...
```

TCPIsGetReady

This function determines if the given socket contains receive data.

Syntax

BOOL TCPIsGetReady(TCP_SOCKET *socket*)

Parameters

socket [in]

Socket identifier that needs to transmitted

Return Values

TRUE: If given socket contains receive data.

FALSE: If given socket does not contain any data.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_LISTEN:
        // Listen to HTTP socket
        httpSocket = TCPListen(&RemoteNode, 80);
        If ( httpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_LISTEN_WAIT;
        return;
case SM_LISTEN_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(httpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Fetch data
        if ( TCPIsGetReady(httpSocket) )
        {
                // Fetch data
...
```

TCPGet

This function fetches one data byte from a given socket receive buffer.

Syntax

BOOL TCPGet(TCP_SOCKET *socket*, BYTE **byte*)

Parameters

socket [in]

Socket identifier that needs to be fetched

byte [out]

Data byte that was read

Return Values

TRUE: If a byte was read.

FALSE: If no byte was read.

Pre-Condition

TCPIsGetReady == TRUE

Side Effects

None

Remarks

When a socket is found to contain receive data, the user must fetch one or more data bytes (if required) in one task time and discard the socket buffer. Data cannot be fetched from another socket until the socket buffer contents for the first is discarded.

Example

```
...
switch(smState)
{
case SM_LISTEN:
        // Listen to HTTP socket
        httpSocket = TCPListen(&RemoteNode, 80);
        If ( httpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_LISTEN_WAIT;
        return;
case SM_LISTEN_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(httpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Fetch data
        if ( TCPIsGetReady(httpSocket) )
        {
                // Fetch data
                TCPGet(httpSocket, &dataByte);
...
```

TCPGetArray

This function fetches a data array from a given socket receive buffer.

Syntax

```
WORD TCPGetArray(TCP_SOCKET socket,
                 BYTE *byte,
                 WORD count)
```

Parameters

socket [in]

Socket identifier that needs to be fetched

byte [out]

Data array that was read

count [out]

Total number of bytes to read

Return Values

Total number of data bytes read.

Pre-Condition

`TCPIsGetReady == TRUE`

Side Effects

None

Remarks

When a socket is found to contain receive data, the user must fetch one or more data bytes (if required) and discard the socket buffer. Data cannot be fetched from another socket until the socket buffer contents for the first is discarded.

This function does not check the request against available data bytes in the receive buffer. The user must make sure that the current receive buffer is not overrun.

TCPGetArray (Continued)

Example

```
...

switch(smState)
{
case SM_LISTEN:
        // Listen to HTTP socket
        httpSocket = TCPListen(&RemoteNode, 80);
        If ( httpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_LISTEN_WAIT;
        return;
case SM_LISTEN_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(httpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Fetch data
        if ( TCPIsGetReady(httpSocket) )
        {
                // Fetch 20 bytes of data
                TCPGetArray(httpSocket, buffer, 20);
...
```

TCPDiscard

This function releases the receive buffer associated with a given socket.

Syntax

BOOL TCPDiscard(TCP_SOCKET *socket*)

Parameters

socket [in]

Socket identifier that needs to transmitted

Return Values

TRUE: If receive buffer for given was successfully discarded.

FALSE: If receive buffer for given buffer was already discarded.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_LISTEN:
        // Listen to HTTP socket
        httpSocket = TCPListen(&RemoteNode, 80);
        If ( httpSocket == INVALID_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                smState = SM_LISTEN_WAIT;
        return;
case SM_LISTEN_WAIT:
        // Wait for connection...
        if ( TCPIsConnected(httpSocket) )
                smState = SM_CONNECTED;
        return;
case SM_CONNECTED:
        // Fetch data
        if ( TCPIsGetReady(httpSocket) )
        {
                // Fetch 20 bytes of data
                TCPGetArray(httpSocket, buffer, 20);

                // Process data.
                ...

                // Release the buffer.
                TCPDiscard(httpSocket);
...
```

TCPProcess

This function acts as “TCPTask”. It fetches an already received TCP packet and executes the TCP State machine for matching sockets. This function must be called only when a TCP packet is received.

Syntax

`BOOL TCPProcess(NODE_INFO *remote, WORD len)`

Parameters

remote [in]

Remote node from which current TCP packet was received

len [out]

Total length of TCP packet length, including TCP header

Return Values

`TRUE`: If this function (task) has completely processed current packet.

`FALSE`: If this function (task) has partially processed current packet.

Pre-Condition

`IPGetHeader == TRUE` and `IPProtocol = IP_PRO_TCP`

Side Effects

None

Remarks

As mentioned above, this function implements the TCP state machine. Users must call this function when a valid TCP data packet is received. Once a packet is detected, this function fetches and handles the packet. The return value from this function indicates to the caller if the StackTask state machine state should be changed or not.

In its current implementation, this function always returns TRUE. Refer to the Stack Manager task source code (`StackTsk.c`) for details on the actual implementation.

Example

```
...

switch(smState)
{
case SM_STACK_IDLE:
        if ( MACGetHeader(&RemoveMAC, &MACFrameType) )
        {
                if ( MACFrameType == MAC_IP )
                        smState = SM_STACK_IP;
        ...
        return;
case SM_STACK_IP:
        if ( IPGetHeader(&RemoteNode, &IPFrameType, &IPDataCount) )
        {
                if ( IPFrameType == IP_PROT_TCP )
                        smState = SM_STACK_TCP;
        ...
        return;
case SM_STACK_TCP:
        if ( TCPProcess(&RemoteNode, IPDataCount) )
                smState = SM_STACK_IDLE;
        return;
...
```

TCPTick

This function acts as another “TCPTask” in addition to `TCPProcess`. This function checks for time-out conditions for all sockets and attempts to recover from them.

Syntax

```
void TCPTick()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

This function implements time-out handling. User must call this function periodically to ensure that time-out conditions are handled in a timely manner.

Example

```
TCPTick();
```

User Datagram Protocol (UDP)

The UDP layer of the Microchip TCP/IP Stack is implemented by the file "UDP.c". The header file "UDP.h" defines the services provided by the layer. In this stack architecture, UDP is an active layer. It fetches UDP packets and notifies corresponding UDP socket of data arrival or transmission. The UDP module is implemented as a cooperative task, performing automatic operations without the knowledge of the main application.

"UDP.h" provides UDP socket services and hides all UDP packet handling from the caller. The layer allows up to 254 UDP sockets (the number limited only by available memory and compiler used). With more than one socket, higher level applications can maintain multiple simultaneous UDP connections; more than one application could be using this layer. It is important to know that each socket consumes approximately 19 bytes (check "UDP.h" file for actual consumption) and increases overall UDP processing time.

Unlike other socket implementations, all sockets in the Microchip TCP/IP Stack share one or more common transmit buffers. This approach reduces overall RAM requirements, but it may create a potential problem, where a few sockets reserve all available transmit buffers and do not release them on time for other sockets to use. Under these circumstances, remote hosts and/or local applications would not be able to contact the stack. To avoid this, users must make sure that there are enough transmit buffers for all sockets.

On the receive side, there is only one receive buffer. If a socket receives its data, the owner of that socket must fetch and discard the receive buffer in one task time in order for the other sockets to receive their data. This design mandates that once a task detects a packet it is interested in, it must consume the complete packet in one task time. A task cannot fetch part of a packet during one task time and expect to fetch the rest of the packet later.

The UDP specifications do not mandate that checksum calculation be performed on UDP packets. To reduce overall program and data memory requirements, the Microchip TCP/IP Stack does not implement UDP checksum calculation; instead, it sets the checksum fields to zero to indicate that no checksum calculation is performed. This design decision requires that all modules utilizing the UDP module ensure their data integrity.

UDPInit

This function initializes the UDP module and prepares it for multiple UDP connections.

Syntax

```
void UDPInit()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

This function is called only once on application start-up.

Example

```
// Initialize UDP
UDPInit();
```

UDPOpen

This function prepares the next available UDP socket on a given port for possible data transfer. Either the local or remote node may initiate the data transfer.

Syntax

UDP_SOCKET UDPOpen(UDP_PORT *localPort*, NODE_INFO **remoteNode*, TCP_PORT *remotePort*)

Parameters

localPort [in]

Local UDP port number on which data transfer will occur

remoteNode [in]

Remote host that contains *remotePort*

remotePort [in]

UDP Port number on remote host to transfer the data to and from

Return Values

A valid socket identifier if there was at least one free socket.

INVALID_UDP_SOCKET if there is no socket available.

Pre-Condition

None

Side Effects

None

Remarks

This function may be used for both local host-initiated and remote host-initiated data transfers. There is no explicit connection step in UDP. Once a UDP socket is opened, it is ready to transmit or receive the data. When a socket is said to be opened, it simply means that the corresponding socket is setup to receive and transmit one or more data packets until that socket is closed.

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Broadcast DHCP Broadcast message.
        break;
...
```

UDPClose

This function closes a given UDP socket and declares it as a free socket.

Syntax

`void UDPDisconnect(UDP_SOCKET *socket*)`

Parameters

socket [in]

Identifier of socket that needs to be closed

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Send DHCP request...
                ...
                // Close the socket
                UDPClose(DHCPSocket);
        break;
...
```

UDPIsPutReady

This macro determines if a given socket is ready to transmit. A socket is ready to transmit when at least one of the MAC transmit buffers is empty. It also sets the given socket as an active UDP socket.

Syntax

BOOL UDPIsPutReady(UDP_SOCKET *socket*)

Parameters

socket [in]

Identifier of the socket that needs to be checked and set active

Return Values

TRUE: If a given socket is ready to transmit.

FALSE: If there is no transmit buffer ready.

Pre-Condition

None

Side Effects

None

Remarks

Since UDP is a connection less protocol, a socket is always ready to transmit as long as at least the MAC transmit buffer is empty. In addition to checking for transmitter readiness, UPDIsPutReady also sets the active UDP socket. Once a socket is set active, successive calls to other UDP functions (such as UDPGet, UDPPut, UDPFlush and UDPDiscard) use the active socket as the current socket. Therefore, it is not necessary for the user to pass the socket on every call.

See the UDPGet, UDPPut, UDPFlush and UDPDiscard functions for additional information.

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Broadcast DHCP Broadcast message.
                smState = SM_BROADCAST;
        break;
case SM_BROADCAST:
        if ( UDPIsPutReady(DHCPSocket) )
        {
                // Socket is ready to transmit. Transmit the data...
                ...
        }
        break;
...
```

UDPPut

This function loads a data byte into the transmit buffer for an active socket.

Syntax

`BOOL UDPPut(BYTE byte)`

Parameters

byte [in]

Data byte to be loaded

Return Values

`TRUE`: If a given data byte was successfully loaded into the transmit buffer and there is room for more data.

`FALSE`: If a given data byte was successfully loaded into the transmit buffer and there is no room for more data.

Pre-Condition

`UDPIsPutReady == TRUE`

Side Effects

None

Remarks

Once a socket is found to be ready to transmit, the user must load all data, or until there is no more room in the socket buffer. The user cannot load part of the data into one socket and part into another socket buffer.

It is important to remember that when socket data is loaded using this function, the actual transmission may or may not start, depending on total number of data bytes loaded. If the number of bytes loaded is less than the available socket buffer size, the user must explicitly flush the transmit buffer using the `UDPFlush` function. If the user tries to load more bytes then the available buffer size, this function automatically starts the transmission and returns FALSE, so the user can try again. Usually, it is a good practice to flush the socket buffer after all known data is loaded into buffer, regardless of whether the buffer was full or not.

UDPPut (Continued)

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Broadcast DHCP Broadcast message.
                smState = SM_BROADCAST;
        break;
case SM_BROADCAST:
        if ( UDPIsPutReady(DHCPSocket) )
        {
                // Socket is ready to transmit. Transmit the data...
                // Note that there is DHCPSocket parameter in UDPPut.
                // This UDPPut call will use active socket
                // as set by UDPIsPutReady() - that is DHCPSocket.
                UDPPut(0x55);
                ...
        }
        break;
...
```

UDPFlush

This function marks the active socket transmit buffer as ready to be transmitted.

Syntax

`void UDPFlush()`

Parameters

Return Values

None

Pre-Condition

`UDPPut()` is already called, and the desired UDP socket is set as the active socket by calling `UDPIsPutReady()`.

Side Effects

None

Remarks

This function marks the current transmit buffer as ready to transmit; the actual transmission may not start immediately. User does not have to check the status of transmission. The NIC will retry a transmission up to 15 times (for the RTL8019AS; check the documentation for the specific NIC if the RTL8019AS is not being used). If the socket is already flushed, another flush would be ignored.

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Broadcast DHCP Broadcast message.
                smState = SM_BROADCAST;
        break;
case SM_BROADCAST:
        if ( UDPIsPutReady(DHCPSocket) )
        {
                // Socket is ready to transmit. Transmit the data...
                // Note that there is DHCPSocket parameter in UDPPut.
                // This UDPPut call will use active socket
                // as set by UDPIsPutReady() - that is DHCPSocket.
                UDPPut(0x55);
                ...
                // Now transmit it.
                UDPFlush();
        }
        break;
...
```

UDPIsGetReady

This function determines if the given socket contains receive data. It also sets a given socket as an active socket.

Syntax

`BOOL UDPIsGetReady(UDP_SOCKET socket)`

Parameters

socket [in]

Identifier for the socket that needs to transmitted and set active

Return Values

TRUE: If a given socket contains receive data.

FALSE: If a given socket does not contain any data.

Pre-Condition

`UDPOpen()` is already called. The value of *socket* must be that returned by `UDPOpen()` call.

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Wait for response from DHCP server
                smState = SM_WAIT_FOR_DATA;
        break;
case SM_WAIT_FOR_DATA:
        if ( UDPIsGetReady(DHCPSocket) )
        {
                // Socket does contain some data. Fetch it and process it.
                ...
        }
        break;
...
```

UDPGet

This function fetches one data byte from an active socket receive buffer.

Syntax

BOOL UDPGet(BYTE *byte)

Parameters

byte [out]

Data byte that was read

Return Values

TRUE: If a byte was read.

FALSE: If no byte was read.

Pre-Condition

UDPIsGetReady == TRUE

Side Effects

None

Remarks

When a socket is found to contain receive data, the user must fetch one or more data bytes (if required) in one task time and discard the socket buffer. Data cannot be fetched from another socket until the socket buffer contents for the first is discarded.

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Wait for response from DHCP server
                smState = SM_WAIT_FOR_DATA;
        break;
case SM_WAIT_FOR_DATA:
        if ( UDPIsGetReady(DHCPSocket) )
        {
                // Socket does contain some data. Fetch it all.
                // buffer is a pointer to BYTE.
                while( UDPGet(buffer) )
                        buffer++;
                // Process it.
                ...
                // Discard the socket buffer.
                ...
        }
        break;
...
```

UDPDiscard

This function releases the receive buffer associated with an active socket.

Syntax

```
BOOL UDPDiscard()
```

Parameters

None

Return Values

TRUE: If the receive buffer was successfully discarded.

FALSE: If the receive buffer was already discarded.

Pre-Condition

None

Side Effects

None

Remarks

None

Example

```
...

switch(smState)
{
case SM_OPEN:
        // Talk to a remote DHCP server.
        DHCPSocket = UDPOpen(68, &DHCPServerNode, 67);
        If ( DHCPSocket == INVALID_UDP_SOCKET )
        {
                // Socket is not available
                // Return error.
        }
        else
                // Wait for response from DHCP server
                smState = SM_WAIT_FOR_DATA;
        break;
case SM_WAIT_FOR_DATA:
        if ( UDPIsGetReady(DHCPSocket) )
        {
                // Socket does contain some data. Fetch it all.
                // buffer is a pointer to BYTE.
                while( UDPGet(buffer) )
                        buffer++;
                // Process it..
                ...
                // Discard the socket buffer.
                UDPDiscard();
        }
        break;
...
```

UDPProcess

This function acts as “UDPTask”. It fetches an already received UDP packet and assigns it to a matching UDP socket. This function must be called only when a UDP packet is received.

Syntax

`BOOL UDPProcess(NODE_INFO *remote, WORD len)`

Parameters

remote [in]

Remote node from which the current UDP packet was received

len [in]

Total length of UDP packet length, including UDP header

Return Values

`TRUE`: If this function (task) has completely processed the current packet.

`FALSE`: If this function (task) has partially processed the current packet.

Pre-Condition

`IPGetHeader == TRUE` and `IPProtocol = IP_PRO_UDP`

Side Effects

None

Remarks

As mentioned above, this function implements the UDP task. Users must call this function when a valid UDP data packet is received. Once a packet is detected, this function fetches and handles the packet. The return value from this function indicates to the caller if the StackTask state machine state should be changed or not. In its current implementation, this function always returns TRUE.

Refer to the Stack Manager task source code (`StackTsk.c`) for details on the actual implementation.

Example

```
...

switch(smState)
{
case SM_STACK_IDLE:
        if ( MACGetHeader(&RemoveMAC, &MACFrameType) )
        {
                if ( MACFrameType == MAC_IP )
                        smState = SM_STACK_IP;
        ...
        return;
case SM_STACK_IP:
        if ( IPGetHeader(&RemoteNode, &IPFrameType, &IPDataCount) )
        {
                if ( IPFrameType == IP_PROT_UDP )
                        smState = SM_STACK_UDP;
        ...
        return;
case SM_STACK_UDP:
        if ( UDPProcess(&RemoteNode, IPDataCount) )
                smState = SM_STACK_IDLE;
        return;
...
```

Dynamic Host Configuration Protocol (DHCP)

The DHCP layer of the Microchip TCP/IP Stack is implemented by the file "dhcp.c". The header file "dhcp.h" defines the services provided by the layer. DHCP is an active layer that broadcasts DHCP requests and automatically receives and decodes DHCP responses. Its main features include:

- Configuration of the IP and gateway addresses and subnet mask
- Automatic DHCP lease time and lease renewal management
- Fully automatic operation without user application intervention

The DHCP module is implemented as a cooperative task, performing automatic operations without the knowledge of the main application. The actual DHCP integration and control is done by the Stack Manager; it handles all required operations as part of its standard task, using the DHCP APIs to control the module's behavior. The user does not need to know about DHCP in order to use it.

A user application may also choose to call some of the APIs to directly control DHCP operation, such as whether DHCP is configured or not, and whether to permanently stop DHCP. Normally, the user's application should not need to directly interact with DHCP at all.

To use the DHCP module, the user project files must be modified as follows:

1. Uncomment STACK_USE_DHCP in the header file "StackTsk.h".
2. Include the "dhcp.c" and "udp.c" files in the project.
3. Increase MAX_UDP_SOCKETS by one (at least one UDP socket must be available for DCHP; adjust the numbers of sockets based on UDP and DCHP needs).

When DHCP is implemented, the user application must not attempt network communications until DHCP is configured properly. Normally, if a user application contains one or more client applications that require communications on power-up or Reset, the application must check that DHCP is configured before transmitting any data using the lower layer modules. This can be done with the function DHCPIsBound (page 75).

The official DHCP specification (RFC1541) requires the DHCP client to renew its IP configuration before its lease time expires. To track lease time, the user application must make sure that TickUpdate() is called as required, and that reasonable time accuracy is maintained (refer to the source code file "websrvr.c" for a working example). The time resolution required is 15 minutes, plus or minus, which means that TickUpdate() may be called at a very low priority.

For the DHCP module to automatically configure the addresses and subnet mask, there must be at least one DHCP server on the network. It is the user's responsibility to implement some method for "publishing" the configurations back to potential users. Options range from manually reading the information from a display on each node, to storing the information in a central server. DHCP updates the values of MY_IP_BYTE?, MY_GATE_BYTE? and MY_MASK_BYTE? as a result of automatic configuration.

DHCPTask

This is the core DHCP task function which performs the necessary protocol actions.

Syntax

```
void DHCPTask()
```

Parameters

None

Return Values

None

Pre-Condition

None

Side Effects

None

Remarks

This function must be called periodically.

Example

```
// Invoke DHCP task
DHCPTask();
```

DHCPDisable

This macro disables the DHCP module, even before its task is invoked.

Syntax

```
void DHCPDisable()
```

Parameters

None

Return Values

None

Pre-Condition

Must be called before `DHCPTask` is called.

Side Effects

None

Remarks

This function is useful if an application requires user option to disable DHCP mode as part of its configuration. Normally, if DHCP is not required at all, it should not be included in the project. But in the event that an application requires run time selection of DHCP, this function can be used to disable DHCP. The Microchip TCP/IP Stack Demo application uses this function to demonstrate run time disabling of DHCP module.

This function must be called before `DHCPTask` function is called. Once `DHCPTask` is called, use `DHCPAbort` (page 74) to disable the DHCP. If `DHCPDisable` is called after `DHCPTask` was executed, the DHCP UDP socket will become an orphan. Any new DHCP responses received by this node will be stored in DHCP UDP socket, but it will not be retrieved by any application; eventually, no application or layer will be able to receive data at all.

Example

```
void main()
{
        // Initialize application and stack
        ...
        StackInit();

        // If required disable DHCP here.
        if ( DHCPIsDisabled )
                DHCPDisable();

        // Now enter into main loop.
        while(1)
        {
                // Perform necessary steps.
                ...
        }
}
```

DHCPAbort

This function aborts an already active DHCP task and disables it for the life of the application.

Syntax

```
void DHCPAbort()
```

Parameters

None

Return Values

None

Pre-Condition

Must be called after `DHCPTask` is called at least once.

Side Effects

None

Remarks

This function can be used to disable or abort DHCP after `DHCPTask` is called. It is important to note that, once DHCPTask is started, the node may have already obtained a DHCP configuration. If DHCP is aborted after obtaining an IP configuration, the configuration will not be automatically renewed. Failure to renew the configuration may result in the node being unable to communicate on the network. It may also result in the same IP address being assigned to another node, causing unreliable communications.

The Microchip Stack does not provide any feedback regarding the validity of the IP configuration.

Example

```
void main()
{
        // Initialize application and stack
        ...
        StackInit();

        // Now enter into main loop.
        while(1)
        {
                // Perform necessary steps.
                ...
                // After some iteration, application needs DHCP aborted.
                DHCPAbort();
        }
}
```

DHCPIsBound

This macro checks if the DHCP module has obtained an IP configuration (is “bound”).

Syntax

```
BOOL DHCPIsBound()
```

Parameters

None

Return Values

`TRUE:` If DHCP is bound to an IP configuration.

`FALSE:` If DHCP is not yet bound.

Pre-Condition

None

Side Effects

None

Remarks

Use this function to determine if DHCP is bound or not. This function is useful when a user application consists of one or more client applications that needs to communicate on power-up or Reset. Before attempting to communicate, the user application must wait until DHCP is bound to IP configuration.

When DHCP is bound to an IP configuration, the corresponding values `MY_IP_BYTE?`, `MY_GATE_BYTE?` and `MY_MASK_BYTE?` are updated.

Example

```
void main()
{
        // Initialize application and stack
        ...
        StackInit();

        // Now enter into main loop.
        while(1)
        {
                // Perform necessary steps.
                ...
                // Check to see if DHCP is bound. If yes, display the IP address
                if ( DHCPIsBound() )
                {
                        // Display MY_IP_BYTE? value.
                        DisplayIPAddress();
                }
        }
}
```

IP Gleaning for IP Address Configuration

As a lean alternative to DCHP, the Microchip TCP/IP Stack also implements a simple method, known as *IP Gleaning*, to remotely set the IP address of Microchip Stack node. This method is *not* an Internet Protocol standard, and there is no corresponding RFC document. IP Gleaning allows only the IP address to be set. For complete IP configuration, DCHP must be used.

IP Gleaning does not require any additional software modules. Instead, it uses the ARP and ICMP modules. To use it, the file "`icmp.c`" must be included in the project, and the compiler define `STACK_USE_IP_GLEANING` must be uncommented in the `StackTsh.h` header file.

With IP Gleaning enabled, the Microchip Stack enters into a special "IP Configuration" mode on Reset. During this mode, it accepts any ICMP (ping) message that is destined to this node, and sets the node's IP address to that of the ping message's destination. Once a valid ping packet is received, the stack exits from "Configuration" and enters into normal mode.

During configuration, the user application must not attempt to communicate; it may call the function `StackIsInConfigMode()` to determine whether or not the stack is in Configuration mode. `StackIsInConfigMode() == TRUE` indicates that the stack is in Configuration mode.

To remotely set the IP address of the stack node, it is necessary to issue a sequence of commands from a remote machine. For this example, assume that a node running the Microchip Stack needs to be assigned an IP address of 10.10.5.106. The MAC address of this node (hexadecimal) is `0a-0a-0a-0a-0a-0a`. After resetting the node, another system on the network (we will assume a computer running Microsoft® Windows®) issues the commands:

```
> arp -s 10.10.5.106 0a-0a-0a-0a-0a-0a
> ping 10.10.5.106
```

This should generate a standard series of ping responses from 10.10.5.106, indicating the node has been successfully assigned the IP address.

The Stack Manager

As already noted, the Microchip TCP/IP Stack is a collection of different modules. Some modules (such as IP, TCP, UDP and ICMP) must be called when a corresponding packet is received. Any application utilizing the Microchip TCP/IP Stack must perform certain steps to ensure that modules are called at the appropriate times. This task of managing stack modules remains the same, regardless of main application logic.

In order to relieve the main application from the burden of managing the individual modules, the Microchip TCP/IP Stack uses a special application layer module known as "StackTask", or the Stack Manager. This module is implemented by the source file "`StackTsk.c`". StackTask is implemented as a cooperative task; when given processing time, it polls the MAC layer for valid data packets. When one is received, it decodes it and routes it to the appropriate module for further processing.

It is important to note that Stack Manager is *not* part of the Microchip TCP/IP Stack. It is supplied with the stack so the main application does not have to manage stack modules, in addition to its own work. Before the Stack Manager task can be put to work, it must be initialized by calling the `StackInit()` function. This function initializes the Stack Manager variables and individual modules in the correct order. Once `StackInit()` is called, the main application must call the `StackTask()` function periodically, to ensure that all incoming packets are handled on time, along with any time-out and error condition handling.

The exact steps used by StackTask are shown in the algorithm in Example 1.

EXAMPLE 1: THE STACK MANAGER ALGORITHM

```
If a data packet received then
   Get data packet protocol type
      If packet type is IP then
         Fetch IP header of packet
         Get IP packet type
         if IP packet type is ICMP then
            Call ICMP module
         else if IP packet type is TCP then
            Call TCP module
         else if IP packet type is UDP then
            Call UDP module
         else
            Handle not supported protocol
         End If
      End If
   Else if packet type is ARP then
      Call ARP module
   End If
End If
```

THE MICROCHIP HTTP SERVER

The HTTP Server included with this application note is implemented as a cooperative task that co-exists with the Microchip TCP/IP Stack and the user's main application. The Server itself is implemented in the source file "HTTP.c", with a user application implementing two callback functions. The demo application source file "Websrvr.c" file should be used as a template application to create the necessary interfaces.

The HTTP Server provided here does not implement all HTTP functionality; it is a minimal server targeted for embedded system. The user can easily add new functionality as required.

The HTTP Server incorporates these main features:

- Supports multiple HTTP connections
- Contains a simple file system (MPFS)
- Supports Web pages located in either internal program memory or external serial EEPROM
- Includes a PC-based program to create MPFS images from a given directory
- Supports the HTTP method "GET" (other methods can be easily added)
- Supports a modified Common Gateway Interface (CGI) to invoke predefined functions from within the remote browser
- Supports dynamic web page content generation

The server consists of the following main components:

- MPFS Image Builder
- MPFS Access Library
- MPFS Download Routine (implemented by the main application)
- HTTP Server Task

In order to integrate the HTTP Server into a user application, do the following:

1. Uncomment STACK_USE_HTTP_SERVER in the header file "StackTsk.h" to enable HTTP Server related code.
2. Set the desired MAX_HTTP_CONNECTIONS value in the "StackTsk.h" header file.
3. Include the files "http.c" and "mpfs.c" in the project.
4. Depending where web pages are stored, uncomment either MPFS_USE_PGRM, or MPFS_USE_EEPROM. If external data EEPROM is used as a storage media, include the file "xeeprom.c" as well.
5. Modify the main() function of the application to include the HTTP server (see the code example in Example 1 (page 8)).

It will also be necessary to generate any Web pages in advance and convert them into a compatible format for storage. For the Microchip Stack and its HTTP Server, the particular format is MPFS (see the section on "Microchip File System (MPFS)" (page 83) for details). If the MPFS image is to be stored in an external data EEPROM, a programming method may need to be included in the application, especially if the content is expected to change. There are three main options for programming external EEPROMs:

1. Use a programmer application or device supplied by the data EEPROM vendor to program the MPFS image. Always make sure that MPFS image starts after the "Reserved Block".
2. Include a download routine in the main application that can accept data from an external data source (i.e., from a computer through a serial data connection) and program it into the EEPROM. An example routine is provided with the Microchip Stack source code (see "MPFS Download Demo Routine" (page 88)).
3. Include the FTP Server module into the project and program the MPFS image remotely across the network using FTP. See "Uploading an MPFS Image Using the FTP Client" (page 85) for more information.

The HTTP Server uses the file "index.htm" as its default Web page. If a remote client (browser) accesses the HTTP Server by its IP address or domain name only, "index.htm" is the default page served. This requires that all applications include a file named "index.htm" as part of their MPFS image. If necessary, the name of this default file can be changed by modifying the compiler definition HTTP_DEFAULT_FILE_STRING in the "http.c" file.

It is very important to make sure that none of the Web page file names contain any of the following non-alphanumeric characters:

- single or double quotes (' and ")
- left or right angle brackets (< and >)
- the pound sign (#)
- the percent sign (%)
- left or right brackets or braces ([,{,] and })
- the "pipe" symbol (|)
- the backslash (\ )
- the caret (^)
- the tilde (~)

If a file does contain any of these characters, the corresponding Web page will become inaccessible. No prior warning will be given.

The HTTP Server also maintains a list of file types that it supports. It uses this information to advise a remote browser on how to interpret a particular file, based on the file's three-letter extension. By default, the Microchip HTTP Server supports ".txt", ".htm", ".gif", ".cgi", ".jpg", ".cla" and ".wav" files. If an application uses file types that are not included in this list, the user may modify the table "httpFiles", along with corresponding "httpContents" enumerations in the file "http.c".

DYNAMIC HTTP PAGE GENERATION

The HTTP Server can dynamically alter pages and substitute real-time information, such as input/output status. To incorporate this real-time information, the corresponding CGI file (*.cgi) must contain a text string '%xx', where the '%' character serves as a control code and 'xx' represents a two-digit variable identifier. The variable value has a range of 00-99. When the HTTP Server encounters this text string, it removes the '%' character and calls the HTTPGetVar function. If the page requires '%' as a display character, it should be preceded by another '%' character. For example, to display "23%" in a page, put "23%%".

HTTPGetVar

This function is a callback from HTTP. When the HTTP server encounters a string '%xx' in a CGI page that it is serving, it calls this function. This function is implemented by the main user application and is used to transfer application specific variable status to HTTP.

Syntax

WORD HTTPGetVar(BYTE *var*, WORD *ref*, BYTE **val*)

Parameters

var [in]

Variable identifier whose status is to be returned

ref [in]

Call Reference. This reference value indicates if this is a very first call. After first call, this value is strictly maintained by the main application. HTTP uses the return value of this function to determine whether to call this function again for more data. Given that only one byte is transferred at a time with this callback, the reference value allows the main application to keep track of its data transfer. If a variable status requires more than bytes, the main application can use *ref* as an index to data array to be returned. Every time a byte is sent, the updated value of *ref* is returned as a return value; the same value is passed on next callback. In the end, when the last byte is sent, the application must return HTTP_END_OF_VAR as a return value. HTTP will keep calling this function until it receives HTTP_END_OF_VAR as a return value.

Possible values for this parameter are:

Value	Meaning
HTTP_START_OF_VAR	This is the very first callback for given variable for the current instance. If a multi-byte data transfer is required, this value should be used to conditionally initialize index to the multi-byte array that will be transferred for current variable.
For all others	Main application-specific value.

val [out]

One byte of data that is to be transferred

Return Values

New reference value as determined by main application. If this value is other than HTTP_END_OF_VAR, HTTP will call this function again with return value from previous call.

If HTTP_END_OF_VAR is returned, HTTP will not call this function and assumes that variable value transfer is finished.

Possible values for this parameter are:

Value	Meaning
HTTP_END_OF_VAR	This is a last data byte for given variable. HTTP will not call this function until another variable value is needed.
For all others	Main application specific value.

Pre-Condition

None

HTTPGetVar (Continued)

Side Effects

None

Remarks

Although this function requests a variable value from the main application, the application does not have to return a value. The actual variable value could be an array of bytes that may or may not be the variable value. What information to return is completely dependent on the main application and the associated Web page. For example, the variable '50' may mean a JPEG frame of 120 x 120 pixels. In that case, the main application can use the reference as an index to the JPEG frame and return one byte at a time to HTTP. HTTP will continue to call this function until it receives `HTTP_END_OF_VAR` as a return value of this function.

Given that this function has a return value of 16 bits, up to 64 Kbytes of data can be transferred as one variable value. If more length is needed, two or more variables can be placed side-by-side to create a larger data transfer array.

Example 1

Consider the page "`status.cgi`" that is being served by HTTP.

"`status.cgi`" contains following HTML line:

```
…
<td>S3=%04</td><td>D6=%01</td><td>D5=%00</td>
…
```

During processing of this file, HTTP encounters the '`%04`' string. After parsing it, HTTP makes a callback `HTTPGetVar(4, HTTP_START_OF_VAR, &value)`. The main user application implements `HTTPGetVar` as follows:

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *val)
{
        // Identify variable.
        // Is it RB5 ?
        if ( var == 4 )
        {
                // We will simply return '1' if RB5 is high,
                // or '0' if low.
                if ( PORTBbits.RB5 )
                        *val = '1';
                else
                        *val = '0;
                // Tell HTTP that this is last byte of
                // variable value.
                return HTTP_END_OF_VAR;
        }
        else
        // Check for other variables...
        ...
}
```

For more detail, refer to "`Webpages\*.cgi`" files and the corresponding callback in the "`Websrvr.c`" source file.

HTTPGetVar (Continued)

Example 2

Assume that the page "status.cgi" needs to display the serial number of the HTTP Web server device.

The page "status.cgi" being served by HTTP contains the following HTML line:

```
...
<td>Serial Number=%05</td>
...
```

While processing this file, HTTP encounters the '%05' string. After parsing it, HTTP makes a callback HTTPGetVar(4, HTTP_START_OF_VAR, &value). The main application implements HTTPGetVar as follows:

```
WORD HTTPGetVar(BYTE var, WORD ref, BYTE *val)
{
        // Identify variable.
        // Is it RB5 ?.
        // If yes, handle RB5 value - will be similar to Example 1.
        // Is it "SerialNumber" variable ?
        if ( var == 5 )
        {
                // Serial Number is a NULL terminated string.
                // First of all determine, if this is very first call.
                if ( ref == HTTP_START_OF_VAR )
                {
                        // This is the first call.  Initialize index to SerialNumber
                        // string. We are using ref as our index.
                        ref = (BYTE)0;
                }
                // Now access byte at current index and save it in buffer.
                *val = SerialNumberStr[(BYTE)ref];
                // Did we reach end of string?
                if ( *val == '\0' )
                {
                        // Yes, we are done transferring the string.
                        // Return with HTTP_END_OF_VAR to notify HTTP server that we
                        // are finished transferring the value.
                        return HTTP_END_OF_VAR;
                }
                // Or else, increment array index and return it to HTTP server.
                (BYTE)ref++;
                // Since value of ref is not HTTP_END_OF_VAR, HTTP server will call
                // us again for rest of the value.
                return ref;
        else
        // Check for other variables...
        ...
}
```

For more detail, refer to "Webpages*.cgi" files and the corresponding callback in the "Websrvr.c" source file.

HTTP CGI

The HTTP server implements a modified version of CGI. With this interface, the HTTP client can invoke a function within HTTP and receive results in the form of a Web page. A remote client invokes a function by HTML GET method with more than one parameter. Refer to RFC1866 (the HTML 2.0 language specification) for more information.

When a remote browser executes a GET method with more than one parameter, the HTTP server parses it and calls the main application with the actual method code and its parameter. In order to handle this method, the main application must implement a callback function with an appropriate code.

The Microchip HTTP Server does not perform “URL decoding”. This means that if any of the form field text contains certain special non-alphanumeric characters (such as <, >, ”, #, %, etc.), the actual parameter value would contain “`%xx`” (“`xx`” being the two-digit hexadecimal value of the ASCII character) instead of the actual character. For example, an entry of “`<Name>`” would return “`%3CName%3C`”. See "The Microchip HTTP Server" (page 77) for a complete list of characters.

A file that contains HTML form, must have “`.cgi`” as its file extension.

HTTPExecCmd

This function is a callback from HTTP. When the HTTP server receives a GET method with more than one parameter, it calls this function. This callback function is implemented by the main application. This function must decode the given method code and take appropriate actions. Such actions may include supplying new Web page name to be returned and/or performing an I/O task.

Syntax

`void HTTPExecCmd(BYTE **argv, BYTE argc)`

Parameters

argv [in]

List of command string arguments. The first string (*argv*[0]) represents the form action, while the rest (*argv*[1..n]) are command parameters.

argc [in]

Total number of parameters, including form action.

Return Values

Main application may need to modify `argv[0]` with a valid web page name to be used as command result.

Pre-Condition

None

Side Effects

None

Remarks

This is a callback from HTTP to the main application as a result of a remote invocation. There could be simultaneous (one after another) invocation of a given method. Main application must resolve these simultaneous calls and act accordingly.

By default, the number of arguments (or form fields) and total of argument string lengths (or form URL string) is limited to 5 and 80, respectively. The form fields limit includes the form action name. If an application requires a form with more than four fields and/or total URL string of more than 80 characters, the corresponding definitions of `MAX_HTTP_ARGS` and `MAX_HTML_CMD_LEN` (defined in “`http.c`”) must be increased.

HTTPExecCmd (Continued)

Example

Consider the HTML page "Power.cgi", as displayed by a remote browser:

```
<html>
<body><center>
<FORM METHOD=GET action=Power.cgi>
<table>
<tr><td>Power Level:</td>
<td><input type=text size=2 maxlength=1 name=P value=%07></td></tr>
<tr><td>Low Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=L value=%08></td></tr>
<tr><td>High Power Setting:</td>
<td><input type=text size=2 maxlength=1 name=H value=%09></td></tr>
<tr><td><input type=submit name=B value=Apply></td></tr>
</table>
</form>
</body></html>
```

This page displays a table with labels in the first column and text box values in the second column. The first row, first column cell contains the string "Power Level"; the second column is a text box to display and modify the power level value. The last row contains a button labelled "Apply". A user viewing this page has the ability to modify the value in the Power Level text box and click on "Apply" button to submit the new power level value to the Microchip Stack.

Assume that a user enters values of '5', '1' and '9' in Power Level, Low-Power Setting and High-Power Setting text boxes respectively, then clicks on the "Apply" button. The browser would create a HTTP request string "Power.cgi?P=5&L=1&H=9" and send it to the HTTP server. The server in turn calls HTTPExecCmd with the following parameters:

argv[0] = "Power.cgi", *argv[1]* = "P", *argv[2]* = "5", *argv[3]*="L", *argv[4]*="1", *argv[5]*="H", *argv[6]*="9"
argc = 7

The main application implements HTTPExecCmd as below:

```
void HTTPExecCmd(BYTE *argv, BYTE argc)
{
        BYTE i;
        // Go through each parameters for current form command.
        // We are skipping form action name, so i starts at 1...
        for (i = 1; i < argc; i++ )
        {
                // Identify parameter.
                if ( argv[i][0] == 'P' )        // Is this power level?
                {
                        // Next parameter is the Power level value.
                        PowerLevel = atoi(argv[++i]);
                }
                else if ( argv[i][0] == 'L' )   // Is this Low Power Setting?
                        LowPowerSetting = atoi(argv[++i]);
                else if ( argv[i][0] == 'H' )   // Is this High Power Setting?
                        HighPowerSetting = atoi(argv[++i]);
        }
        // If another page is to be displayed as a result of this command, copy
        // its upper case name into argv[0]
        // strcpy(argv[0], "RESULTS.CGI");
}
```

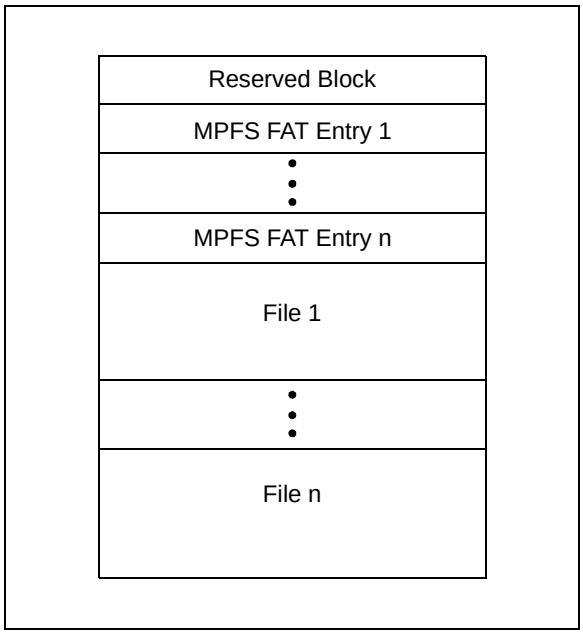
Note: For this example, the total number of arguments exceeds the default limit of 5. In order for this example to function properly, the value of MAX_HTTP_ARGS (located in "http.c") must be set to at least 7.

For more detail, refer to the "Webpages*.cgi" files and the corresponding callback in the "Websrvr.c" source file.

MICROCHIP FILE SYSTEM (MPFS)

The Microchip HTTP Server uses a simple file system (the Microchip File System, or “MPFS”) to store Web pages. The MPFS image can be stored in on-chip program memory or an external serial EEPROM. MPFS follows a special format to store multiple files in given storage media, which is summarized in Figure 3.

FIGURE 3: MPFS IMAGE FORMAT



The length of “Reserved Block” is defined by `MPFS_RESERVE_BLOCK`. The reserved block can be used by the main application to store simple configuration values. MPFS storage begins with one or more MPFS FAT (File Allocation Table) entries, followed by one or more file data. The FAT entry describes the file name, location and its status. The format for the FAT entry is shown in Figure 4.

FIGURE 4: MPFS FAT ENTRY FORMAT

Flag	Address	File Name
(8 bits)	(16 or 24 bits)	(8-byte + 3-byte format)

The *Flag* indicates whether the current entry is in use, deleted, or at the end of the FAT.

Each FAT entry contains either a 16-bit or 24-bit address value. The address length is determined by the type of memory used, as well as the memory size model. If internal program memory is used, and the Microchip TCP/IP Stack project is compiled with a small memory model, 16-bit addresses are used. If internal program memory and the large memory model are selected, a 24-bit address is used. The 16-bit addressing scheme is always used for external EEPROM devices, regardless of the memory size model. This implies a maximum MPFS image size of 64 Kbytes for these devices.

MPFS uses “short” file names of the “8 + 3” format (8 bytes for the actual file name and 3 bytes for the extension, or `NNNNNNNN.EEE`). The 16-bit address gives the start of the first file data block. All file names are stored in upper case to make file name comparisons easier.

The address in each FAT entry points in turn to a data block that contains the actual file data. The data block format is shown in Figure 5. The block is terminated with a special 8-bit flag called EOF (End Of File), followed by FFFFh (for 16-bit addressing), or FFFFFFh (24-bit addressing). If the data portion of the block contains an EOF character, it is stuffed with the special escape character, DLE (Data Link Escape). Any occurrence of DLE itself is also stuffed with DLE.

FIGURE 5: MPFS DATA BLOCK FORMAT

Data	EOF	FFFFh or FFFFFFh
(variable length)	(8 bits)	(16 or 24 bits)

MPFS Image Builder

This application note includes a special PC-based program (MPFS.exe) that can be used to build MPFS image from a set of files. Depending on where the MPFS will ultimately be stored, the user has the option to generate either a 'C' data file or binary file representing the MPFS image.

The complete command line syntax for MPFS.exe is

```
mpfs [/?] [/c] [/b] [/r<Block>]
<InputDir> <OutputFile>
```

where

/? displays command line help

/c generates a 'C' data file

/b generates a binary data file (default output)

/r reserves a block of memory at beginning of the file (valid only in Binary Output mode, with a default value of 32 bytes)

<InputDir> is the directory containing the files for creating the image

<OutputFile> is the output file name

For example, the command

```
mpfs /c <Your WebPage Dir> mypages.c
```

generates the MPFS image as a 'C' data file, mypages.c, from the contents of the directory "Your Web Pages". In contrast, the command

```
mpfs <Your WebPage Dir> mypages.bin
```

generates a binary file of the image with a 32-byte reserved block (both binary format and the 32-byte block are defaults), while

```
mpfs /r128 <Your WebPage Dir> mypages.bin
```

generates the same file with a 128-byte reserved block.

Note:	Using a reserve block size other than the default of 32 bytes requires a change to the compiler define MPFS_RESERVE_BLOCK in the header file "StackTsk.h".

Before the MPFS image is built, the user must create all of the Web pages and related files and save in a single directory. If the file extension is "htm", the Image Builder strips all carriage return and linefeed characters to reduce the overall size of the MPFS image.

If the MPFS image is to be stored in internal program memory, the generated 'C' data file must be linked with the "Websrvr' project. If the image is to be stored in an external serial EEPROM, the binary file must be downloaded there. For more information, refer to "The Microchip FTP Server" (page 85).

The MPFS Image Builder does not check for size limitations. If the binary data format is selected, verify that the total image size does not exceed the available MPFS storage space.

MPFS Access Library

The source file "MPFS.c" implements the routines required to access MPFS storage. Users do not need to understand the details of MPFS in order to use HTTP. All access to MPFS is performed by HTTP, without any help from the main application.

The current version of the MPFS library does not allow for the addition or deletion of individual files to an existing image; all of the files comprising a single MPFS image are added at one time. Any changes require the creation of a new image file.

If internal program memory is used for MPFS storage, MPFS_USE_PGRM must be defined. Similarly, if external data EEPROM is used for MPFS storage, MPFS_USE_EEPROM must be defined. Only one of these definitions can be present in "StackTsk.h"; a compile-time check makes certain that only one option is selected.

Depending on the type of memory device used, its page buffer size will vary. The default setting of the page buffer size (as defined by MPFS_WRITE_PAGE_SIZE in the header file ("MPFS.h")) is 64 bytes. If a different buffer size is required, change the value of MPFS_WRITE_PAGE_SIZE appropriately.

Note:	This version of the MPFS access library uses the file "xeeprom.c" for access to external data EEPROMs. When a file is being read or written, MPFS exclusively controls the I^2C bus and will not allow any other I^2C slave or master device to communicate. Users creating applications with multiple I^2C devices need to bear this in mind.

THE MICROCHIP FTP SERVER

The FTP Server included with this application note is implemented as a cooperative task that co-exists with the Microchip TCP/IP Stack and the user's main application. The Server itself is implemented in the source file "FTP.c".

The FTP Server provided here does not implement all of the functionality specified in RFC 959; it is a minimal server targeted for embedded system. It incorporates these main features:

- One FTP connection, authenticated by the user application
- Automatic interaction with the file system (MPFS)
- Remote programmability of entire MPFS image with one "put" command
- No individual file upload or retrieve support

The user can add new functionality as required.

The server actually consists of two components: the FTP server itself, and the FTP authentication callbacks implemented by a user application. The user must implement a callback with the FTPVerify function that verifies both the FTP user name and password. (See the "FTPVerify" function on the next page for more detail.) The demo application source file "Websrvr.c" should be used as a template application to create the necessary interfaces. Refer to "Demo Application" for a working example of how to integrate the FTP server into a user application.

To integrate the FTP Server into a user application, do the following:

1. Uncomment STACK_USE_FTP_SERVER in the "StackTsk.h" header file.
2. Increase the number of defined sockets by two (depending on the number already available).
3. Include the files "FTP.c" and "mpfs.c" in the project.
4. Uncomment either MPFS_USE_PGRM or MPFS_USE_EEPROM, depending on where the Web pages are to be stored. If an external data EEPROM is used as a storage media, include "xeeprom.c" in the project.
5. Modify the main() function of the application to include the HTTP server (see the code example in Example 1 (page 8)).

The Microchip FTP Server allows only one FTP connection at a time. Each FTP connection requires two TCP sockets.

FTP Server uses a default time-out of approximately 180 seconds for both uploads and downloads. If a remote FTP connection stays IDLE for more than 180 seconds, it is automatically disconnected. This ensures that an orphan connection or a problem during a file upload does not tie up the FTP server indefinitely.

Uploading an MPFS Image Using the FTP Client

The main purpose of the FTP Server in the Microchip Stack is to remotely upgrade the MPFS image. The current version is available only when using the external data EEPROM for MPFS storage; it will not work if the MPFS_USE_PGRM option is selected.

A typical exchange between a user and a node running the Microchip Stack with FTP Server is shown in Example 2. In this instance, an MPFS image is being uploaded from a computer to the node. For the sake of illustration, this is what a user would see using a command window from a computer running Microsoft Windows; other systems and terminal emulations may vary slightly. The actual FTP user name and password depends on the user application; the Web Server demo application (page 87) uses the values shown. FTP Client actions (i.e., manual input from the user) are shown in **bold**. System prompts and FTP server responses are in plainface.

EXAMPLE 2: UPLOADING AN MPFS IMAGE USING FTP

```
c:\ ftp 10.10.5.15
220 ready
User (10.10.5.15: (none)): ftp
331 Password required
Password: microchip
230 Logged in
ftp> put mpfsimg.bin
200 ok
150 Transferring data...
226 Transfer Complete
ftp> 16212 bytes transferred in 0.01Seconds 16212000.00Kbytes/sec.
ftp> quit
221 Bye
```

Note: The FTP server does NOT echo back the password as the user types it in. In the instance above, it is shown to illustrate what the user would enter.

FTPVerify

This function is a callback from the FTP Server. The server calls this function when it receives a connect request from a remote FTP client. This function is implemented by the main user application, and is used to authenticate the remote FTP user.

Syntax

`BOOL FTPVerify(char *login, char *password)`

Parameters

login [in]

Character string that contains the user name

password [in]

Character string that contains the password

Return Values

TRUE: If *login* and *password* match the login and password variable values defined in the user application

FALSE: If either the *login* or *password* do not match

FTP Server uses the return value from this function to allow or disallow access by the remote FTP user.

Pre-Condition

None

Side Effects

None

Remarks

The length user name may range from 0 through 9. If a longer user name is required, modify the value of FTP_USER_NAME_LEN in the header file "ftp.h".

The maximum password string length and total FTP command length is predefined to be 31. If a longer password and/or command is required, modify the value of MAX_FTP_CMD_STRING_LEN in "FTP.c".

Example

This example demonstrates how a FTP callback will be implemented.

```
ROM char FTPUserName[] = "ftp";
#define FTP_USER_NAME_LEN   (sizeof(FTP_USER_NAME)-1)
ROM char FTPPassword[] = "microchip";
#define FTP_USER_PASS_LEN   (sizeof(FTP_USER_PASS)-1)

BOOL FTPVerify(char *login, char *password)
{
        if ( !memcmppgm2ram(FTP_USER_NAME, login, FTP_USER_NAME_LEN) )
        {
                if ( !memcmppgm2ram(FTP_USER_PASS, password, FTP_USER_PASS_LEN) )
                        return TRUE;
        }
        return FALSE;
}
```

For more detail, refer to "Webpages*.cgi" files and the corresponding callback in the "Websrvr.c" source file.

DEMO APPLICATIONS

Included with the Microchip TCP/IP Stack is a complete working application that demonstrates all of the TCP/IP modules. This application (the "Web Server") is designed to run on Microchip's PICDEM.net™ Ethernet/Internet demonstration board. The main source file for this application is "`websrvr.c`". Users should refer to the source code as a starting point for creating their own applications, utilizing different aspects of the Microchip TCP/IP Stack. The sample projects also included with the Microchip Stack illustrate different configurations in which this demo application can be run.

Configuring the PICDEM.net Board and the Web Server

To run the demo application, it is necessary to have a HEX code file. One option is to build one of the sample projects based on the compiler and hardware configuration to being used; alternatively, use the already built HEX file for the corresponding project (see Table 2 on page 5 for a list of projects included with the Microchip Stack). Follow the standard procedure for your device programmer when programming the microcontroller. Make sure that the following configuration options are set:

- Oscillator: HS
- Watchdog Timer: Disabled
- Low-Voltage Programming: Disabled

When the programmed microcontroller is installed on the PICDEM.net demo board and powered up, the System LED should blink to indicate that the application is running. The LCD display will show

MCHPStack v2.0

on the first line (the version number may differ, depending on the release level of the application), and either a configuration message or an IP address on the second line.

Once programmed, the demo application may still need to be configured properly before it is put on a real network. The instructions below are specific to Microsoft Windows and the HyperTerminal terminal emulator; your procedure may vary if you use a different operating system or terminal software.

1. Program a PIC18 microcontroller as noted above, and install it on the PICDEM.net board.
2. Connect the PICDEM.net board to an available serial port on the computer, using a standard RS-232 cable.
3. Launch HyperTerminal (*Start > Programs > Accessories*).
4. At the "Connection Description" dialog box, enter any convenient name for the connection. Click "OK".
5. At the "Connect To" dialog box, select the COM port that the PICDEM.net board is connected to. Click "OK".
6. Configure the serial port connected to the PICDEM.net board:
   - 19200 bps,
   - 8 data bits, 1 STOP bit and no parity
   - no flow control

   Click "OK" to initiate the connection.
7. Apply power to the board while holding the S3 switch, or press and hold both the Reset and S3 switches; then, release the Reset switch. The LCD display shows the message

   MCHPStack v2.0

   Board Setup...

   (The version number may differ depending on the release level of the application.) Release S3.

   The Configuration menu appears in the terminal window:

   ```
   MCHPStack Demo Application v1.0
   (Microchip TCP/IP Stack 2.0)

   1: Change board Serial number.

   2: Change default IP address.

   3: Change default gateway address.

   4: Change default subnet mask.

   5: Enable DHCP and IP Gleaning.

   6: Disable DHCP and IP Gleaning.

   7: Download MPFS image.

   8: Save & Quit.

   Enter a menu choice (1-8):
   ```
8. Select each of the items that need to be configured and enter the new values (generally, this will only be items 2, 3, and 4). Select item 8 to save the changes and exit configuration; the new addresses are saved to the data EEPROM. The application exits Configuration mode and runs the Web Server.

Connecting to an Ethernet Network

When running the Web Server demo application, the PICDEM.net board can be directly connected to an Ethernet network with no other modifications. Of course, the IP configuration must be compatible with that of the network. By default, the Web Server application uses these values for configuration:

- IP Address: 10.10.5.15
- Gateway Address: 10.10.5.15
- Subnet Mask: 255.255.255.0

Even if the IP address is compatible, the gateway and mask may not be. If changes are required, there are several ways to go about it.

AUTOMATIC CONFIGURATION WITH DHCP

If the network uses DHCP configuration, no additional work is needed. When the board is connected to the network and powered up, it will be assigned an IP configuration by the DHCP server. During this process, the LCD display shows the message

DCHP/Gleaning...

After several seconds, the display shows the assigned IP address, for example:

100.100.100.1 1

The actual IP address displayed in the assigned address of the board. The number on the far right indicates the number of times the DHCP lease has been renewed. This is shown for informational purposes only.

Depending on how the network has been configured, the PICDEM.net board's IP address may change after being powered down for an extended period (i.e., the board's DHCP lease has expired and the old address has been taken by another device). Always use the IP address currently displayed to communicate with the board.

PRE-DEFINED NETWORK CONFIGURATIONS

Some networks may be "hard configured"; that is, each device has an address that has been manually assigned by the network administrator. In these cases, the PICDEM.net board should be configured manually before attaching it to the network, with the IP configuration provided by the administrator. Refer back to "Configuring the PICDEM.net Board and the Web Server" (page 87) for details.

SETTING THE IP ADDRESS WITH IP GLEANING

If the board is connected to the network and only requires a change of IP address, IP Gleaning (page 76) can be used. As already mentioned, this method can be used to configure the IP address, but not the gateway or subnet mask.

To use IP gleaning, the MAC address of the device must be known. This is always a 6-byte hexadecimal number of the format "xx-xx-xx-xx-xx-xx". For PICDEM.net boards, the MAC is always 00-04-A3-00-nn-nn, where "nn-nn" is the serial number of the board in hexadecimal format. Thus, a board with serial number 1234 (or 04D2h) has a MAC address 00-04-A3-00-04-D2.

Once the MAC address and new IP address of the device are determined, the address is determined by resetting the device, then issuing from a remote terminal the arp and ping commands. Continuing with the example above, if we wanted to assign the previously mentioned board the new IP address of 10.10.5.50, we would send the commands

```
> arp -s 10.10.5.50 00-04-a3-00-04-d2
> ping 10.10.5.50
```

A successful ping response indicates that the IP address has been changed.

MPFS Download Demo Routine

If an MPFS image is to be stored in an external serial EEPROM, it must either be pre-programmed with the MPFS image or downloaded from another application. The Web Server demo implements a simple MPFS download routine, which accepts an MPFS binary file from a terminal emulator using the Xmodem protocol.

To download a binary file in MPFS format:

1. If not already done, set up the PICDEM.net board for configuration (see "Configuring the PICDEM.net Board and the Web Server", steps 1 through 7).
2. At the Configuration menu, type '7' to start the MPFS download. You should see "Ready to download..." message. At this time, you should also see the left User LED (D6) blinking approximately twice per second.
3. From the HyperTerminal "Transfer" menu, select "Send File…". In the "Send File" dialog box, browse to the directory containing the file "mpfsimg.bin", and select. Select "Xmodem" as the protocol.
4. Click "Send". Data transfer should start automatically. The User LED will blink as fast as data is received from the computer.
5. When the file is completely transferred, press '8' to exit the Configuration mode.

The Web Server application is now running, and the HTTP server is ready to serve the pages just loaded.

Demo HTTP Server

When configured correctly and provided with an MPFS image, the demo HTTP Server (page 77) will serve Web pages. The sample pages included with the Microchip Stack source archive illustrate both a modified form of CGI (a remote method invocation) and dynamic page generation.

Demo FTP Server

This demo application is discussed in detail starting on page 85. Among other things, it allows for remotely updating the MPFS image across a network. The default configuration uses the FTP user name of "ftp" and password of "microchip". These are defined in the Web Server demo application.

SLIP CONFIGURATION FOR WINDOWS 95/98 SYSTEMS

Any personal computer running 32-bit versions of Microsoft Windows (that is, Windows 95 or higher) can be configured to use a SLIP connection for its network communication services. This section outlines the steps required to configure a desktop system for SLIP, and create a pre-defined SLIP connection.

Creating the connection is done in two steps: 1) creating a dummy modem on an available COM port, and 2) defining a dial-up connection type for that device.

The procedure described here applies to both Windows 95 (all versions except the original release) and 98; some steps may differ, depending on the operating system and revision level. For sake of brevity, the configuration process for Windows NT and Windows 2000 Professional Desktop have been omitted. Interested users in creating a SLIP connection with these operating systems should refer to Microsoft documentation for more details, using these steps as a guide.

To create the dummy modem:

1. From the Control Panel (*Start > Settings > Control Panel*), double click on the “Modems” applet.
2. If no modem is installed in the system, an introductory dialog box will appear; check the appropriate box to prevent Windows from searching for a device. Click “Next”.
   If a modem is already installed, a Modem Properties sheet will appear. Click the “Add” button. At any subsequent dialog boxes, opt to manually choose the device; do not let Windows search for the device or automatically select drivers.
3. At the device selection dialog box, select “Standard Modem Types” from the “Manufacturers” drop-down list; select any of the standard modems (preferably 19.2 Kbaud or faster) from the “Models” list. Click “Next”.
4. At the following dialog box, select the appropriate COM port, and speed (if requested). Click “Next”.
5. Following the installation message and a brief delay, a successful installation message will appear. Click “Finish”.

Once the device is created on the proper port, it can be configured for SLIP.

1. Launch the Dial-up Connection Wizard (*Start > Settings > Dial-up Connections > Add a Connection*)
2. At the first “Make New Connection” dialog box, select the newly created Standard Modem as the device. Click “Next”.
3. When asked for a phone number at the next dialog, enter any phone number (this will not be used). Use the default area code and location. Click “Next”.
4. Click “Finish” to create the next connection. It will appear as a new icon in the “Dial-up Networking” folder.
5. Right click on the new connection icon and select “Properties” from the menu.
6. At the Connection Properties sheet, select the “General” tab. Verify that the standard (dummy) modem is selected in the “Connect Using” drop-down menu. Click on the “Configure” button.
7. At the configuration property sheet, set the baud rate to 38,400. Select the “Connection” tab, and verify that the communications settings are 8 data bits, 1 STOP bit and no parity. Click on the “Advanced” button.
8. Deselect the “Use Flow Control” checkbox. Click “OK” to close the dialog box, then “OK” to close the configuration property sheet (you are now back at the Connection Properties sheet).
9. Select the “Server Types” tab. Select “SLIP – Unix Connection” from the “Dial-Up Server Type” drop-down list. Uncheck all check boxes except at the bottom check box, “TCP/IP”. Click on the “TCP/IP Settings” button.
10. Enter an IP address of “10.10.5.1”. Deselect the “IP header compression” and “Use default gateway on remote network” options. Click “OK” to return to the Properties sheet.

> **Note 1:** The address used in this configuration is for the host desktop system for this connection; it is NOT the address of the target system. The IP address of 10.10.5.1 is used specifically with the PICDEM.net Demonstration Board.
>
> **2:** The host system and target must be on the same subnet.

11. Select the “Scripting” tab, then click the “Browse” button to locate the directory where the file “`empty.scp`” resides (included in the Microchip TCP/IP Stack archive file). Select the file and click “OK”.
12. Click “OK” to finish.

To use the SLIP connection, double-click on its icon. Use the Web browser and any IP application to communicate over the connection.

MEMORY USAGE

The total amount of memory used for the Microchip TCP/IP Stack depends on the compiler and modules being used. Typical sizes for the various stack modules at the time of publication of this document (June 2002), using HI-TECH PICC-18™ V8.11PL1 compiler are shown in Table 4.

TABLE 4: MEMORY USAGE FOR THE VARIOUS STACK MODULES USING HI-TECH® PICC-18™ COMPILER

Module	Program Memory (words)	Data Memory (bytes)
MAC (Ethernet)	906	5(1)
SLIP	780	12(2)
ARP	392	0
ARPTask	181	11
IP	396	2
ICMP	318	0
TCP	3323	42
HTTP	1441	10
FTP Server	1063	35
DHCP Client	1228	26
IP Gleaning	20	1
MPFS(3)	304	0
Stack Manager	334(4)	12+ICMP Buffer

Note 1: As implemented with the RTL8019AS NIC.
2: Does not include the size of transmit and receive buffers, which are user defined.
3: Internal program memory storage.
4: Maximum size. Actual size may vary.

CONCLUSION

The Microchip TCP/IP Stack is not the first application of its type for the Microchip PIC18 family of microcontrollers. What it does provide is a space efficient and modular version of TCP/IP, implementing cooperative multitasking (without an operating system) in a small footprint. With its adaptability to many firmware applications and its availability as a no cost software solution, this stack should prove to be useful in developing applications where embedded control requires network connectivity.

APPENDIX A: SOURCE CODE

The complete source code for the Microchip TCP/IP Stack, 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 file "version.log" for the current revision level and a history of changes to the software.

APPENDIX B: PARTIAL LIST OF RFC DOCUMENTS

RFC Document	Description
RFC 826	Ethernet Address Resolution Protocol (ARP)
RFC 791	Internet Protocol (IP)
RFC 792	Internet Control Message Protocol (ICMP)
RFC 793	Transmission Control Protocol (TCP)
RFC 768	User Datagram Protocol (UDP)
RFC 821	Simple Mail Transfer Protocol (SMTP)
RFC 1055	Serial Line Internet Protocol (SLIP)
RFC 1866	Hypertext Markup Language (HTML 2.0)
RFC 2616	Hypertext Transfer Protocol (HTTP) 1.1
RFC 1541	Dynamic Host Configuration Protocol (DHCP)
RFC 1533	DHCP Options
RFC 959	File Transfer Protocol (FTP)

The complete list of Internet RFCs and the associated documents are available on many Internet web sites. Interested readers are referred to **www.faqs.org/rfcs** and **www.rfc-editor.org** as starting points.

NOTES:

Note the following details of the code protection feature on Microchip devices:

- Microchip products meet the specification contained in their particular Microchip Data Sheet.
- Microchip believes that its family of products is one of the most secure families of its kind on the market today, when used in the intended manner and under normal conditions.
- There are dishonest and possibly illegal methods used to breach the code protection feature. All of these methods, to our knowledge, require using the Microchip products in a manner outside the operating specifications contained in Microchip's Data Sheets. Most likely, the person doing so is engaged in theft of intellectual property.
- Microchip is willing to work with the customer who is concerned about the integrity of their code.
- Neither Microchip nor any other semiconductor manufacturer can guarantee the security of their code. Code protection does not mean that we are guaranteeing the product as "unbreakable."

Code protection is constantly evolving. We at Microchip are committed to continuously improving the code protection features of our products. Attempts to break Microchip's code protection feature may be a violation of the Digital Millennium Copyright Act. If such acts allow unauthorized access to your software or other copyrighted work, you may have a right to sue for relief under that Act.

Trademarks

QUALITY MANAGEMENT SYSTEM
CERTIFIED BY DNV
ISO/TS 16949:2002

Microchip received ISO/TS-16949:2002 certification for its worldwide headquarters, design and wafer fabrication facilities in Chandler and Tempe, Arizona; Gresham, Oregon and design centers in California and India. The Company's quality system processes and procedures are for its PIC® MCUs and dsPIC® DSCs, KEELOQ® code hopping devices, Serial EEPROMs, microperipherals, nonvolatile memory and analog products. In addition, Microchip's quality system for the design and manufacture of development systems is ISO 9001:2000 certified.

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

China - Xiamen
Tel: 86-592-2388138
Fax: 86-592-2388130

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

China - Zhuhai
Tel: 86-756-3210040
Fax: 86-756-3210049

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

01/02/08