TCP/IP LIBRARY PROGRAMMER'S GUIDE

Relevant Devices

This application note applies to the following devices:
C8051F120, C8051F121, C8051F122, C8051F123, C8051F124, C8051F125, C8051F126, C8051F127, C8051F130, C8051F131, C8051F132, C8051F133, C8051F020, C8051F021, C8051F022, C8051F023, C8051F340, C8051F341, C8051F342, C8051F343, C8051F344, C8051F345, C8051F346, C8051F347

1. Introduction

The Silicon Laboratories TCP/IP stack is designed to add network connectivity to the 'F12x, 'F13x, 'F02x, and 'F34x family of microcontrollers. It is highly configurable and has a small memory footprint. The TCP/IP stack is packaged with a Configuration Wizard that can generate the framework code required to develop a networked application and numerous examples to jump-start development and minimize time to market.

The TCP/IP stack includes the following features:

- HTTP web server with CGI scripting, SMTP e-mail client, FTP server, TFTP client, Netfinder, DNS client, and virtual file system.
- Up to 127 simultaneous TCP or UDP sockets. Direct access to sockets allows custom application development.
- Support for Ethernet with DHCP/BOOTP capability. Interfaces to a CP220x through the external memory parallel interface. Custom driver support allows any Ethernet controller to be used.
- Support for PPP and SLIP with customizable modem settings (C8051F12x only). Interfaces to an Si2457 modem through the serial port. Supports any standard "AT" serial modem.

The TCP/IP stack is freely available for use with a Silicon Laboratories MCU and can be downloaded from the Silicon Laboratories web site. It is also included in the Embedded Ethernet Development Kit (Ethernet-DK) and the Embedded Modem Development Kit (Modem-DK), which include:

- C8051F12x Target Board, USB Debug Adapter, and Universal Power Supply.
- AB4 Ethernet Development Board **or** the Si2457FT18-EVB Modem Board and AB3 Modem Adapter Board. **Note:** A direct telephone line or phone simulator is required to communicate with the modem.
- Evaluation version of the Keil C51 toolchain limited to 4 kB object code generated from application code. TCP/IP library code does not count towards the 4 kB limit with BL51 linker Version 5.15 or higher.
- TCP/IP Configuration Wizard to generate custom libraries and example projects that demonstrate how to set up an HTTP web server, send an e-mail, and send and receive TCP and UDP packets.

2. API Function Overview

The TCP/IP stack provides a set of functions that implement an application programming interface (API). These functions provide the microcontroller an Ethernet or dial-up network interface via the CP220x (Ethernet) or Si2457 (Modem). All low-level hardware details and protocols are handled by the API and do not require management by application code. The API is provided in the form of a library file pre-compiled under the Keil C51 tool chain. (Device firmware must be developed using the Keil C51 tool chain.) Some commonly-used API functions are listed below:

mn_init()	Initializes all sockets and stack variables
mn_send()	Sends data using a TCP or UDP socket
mn_recv()	Waits for data to arrive on a TCP or UDP socket
mn_server()	Starts an HTTP or FTP Server
mn_smtp_send_mail()	Sends an e-mail to an SMTP mail server

TABLE OF CONTENTS

3. Getting Started

Starting a new project that uses the TCP/IP stack is simple. There are five ways to get started.

- Modifying an HTTP Web Server Example.
- Modifying an SMTP Mail Client Example.
- Modifying a TCP Socket Example.
- Modifying a UDP Socket Example.
- Modifying a DHCP/BOOTP Example (Ethernet Only).
- Using the TCP/IP Configuration Wizard to generate a custom library and framework code.

3.1. Project Directory Structure

A typical TCP/IP project directory consists of the following files and sub-directories:

Group 1 - User Files:

`main.c`	Contains the main routine and callback functions.
`mn_callback.c`	Contains additional callback functions.
`mn_userconst.h`	Header file containing user settings.
`TCPIP_Project.wsp`	Project file that can be opened from the Silicon Labs IDE.

Group 2 - TCP/IP Stack Files:

`mn_defs.h`	Contains type definitions used by the TCP/IP stack.
`mn_errs.h`	Contains error code definitions.
`mn_funcs.h`	Contains function prototype information.
`mn_stackconst.h`	Contains constants required by the TCP/IP stack.
`mn_vars.c`	Contains global variables used by the TCP/IP stack.
`mn_stack_common_000.lib`	TCP/IP common library. Note the three digit library number.
`mn_stack_bank_000.lib`	TCP/IP banked library. Note the three digit library number.

Group 3 - Optional Files:

`VFILE_DIR`	Optional subdirectory containing HTML files, images, and other content.
`VFILE_DIR\html2c.exe`	Optional html2c.exe utility that converts content to file arrays.
`VFILE_DIR\update.bat`	Optional batch file to automate conversion to file arrays.
`VFILE_DIR\mn_defs.h`	Optional header file required only if using file arrays.
`VFILE_DIR\index.html`	Optional main HTML page for web server.
`VFILE_DIR\index.h`	Optional main HTML page converted to file array using *html2c.exe.*
`VFILE_DIR\index.c`	Optional main HTML page converted to file array using *html2c.exe.*
`custom_ethernet.h`	Optional header file for custom ethernet driver.
`custom_ethernet.c`	Optional source file for custom ethernet driver.
`netfinder.h`	Optional header file for Netfinder customization.
`netfinder.c`	Optional source file for Netfinder support.

3.2. TCP/IP Configuration Wizard Output

The TCP/IP Configuration Wizard generates two custom libraries with a unique three-digit library number that describes the selected protocol configuration. When the TCP/IP project is configured for code banking, the common library is placed in the common area, and the banked library is placed in a code bank. In non-banked TCP/IP projects, both libraries are included in the build and automatically located by the linker. The Wizard also generates the supporting directory structure and framework code required to start a new TCP/IP project. Note that the framework code generated will change based on the library number, and libraries with different three-digit library numbers cannot be interchanged between projects without regenerating the supporting code. To start using the code generated by the Wizard, open the *TCPIP_Project.wsp* file using the *Project→Open Project...* command from the Silicon Laboratories IDE.

3.3. Using the TCP/IP Examples

The TCP/IP code examples are a good starting point for new projects. If the protocols used in an example meet the needs of an end application, such as the HTTP Web Server, the example can be easily modified to include additional application code and the HTML files changed to suit the application. If a different combination of protocols is needed, a new library and supporting code can be generated using the TCP/IP Configuration Wizard. See the Embedded Ethernet Development Kit User's Guide or the Embedded Modem Development Kit User's Guide for step-by-step instructions on setting up the hardware and running the code examples.

3.4. Getting Additional Help

If you have any questions or run into any problems while using the TCP/IP Stack or the TCP/IP Configuration Wizard, contact MCU Applications by visiting www.silabs.com and clicking on the "Support" link. If you are designing an application that requires features or protocols not currently available in the TCP/IP Library, please contact us, and we will be glad to help you find a solution.

4. TCP/IP Stack API Reference

4.1. Function Groups

The TCP/IP stack functions are divided into the following groups:

`Socket Functions`	Open, close, and manage sockets.
`Ethernet Functions`	Initialize the Ethernet controller and provide direct register access.
`Modem Functions`	Manage the connection between the MCU and the modem.
`PPP Functions`	Open, close, and manage a PPP connection.
`DHCP/BOOTP Functions`	Used to obtain a dynamic IP address and boot file.
`Application Functions`	Start and use application layer services (HTTP Web Server, FTP Server, TFTP Client, SMTP Mail Client, and DNS client).
`Callback Functions`	Event handlers called by the stack to notify the application layer.
`VFILE Functions`	Add and remove files or CGI scripts to the virtual file system.
`Support Functions`	Convert between various data types.

4.2. Data Types

The following data types are used by the TCP/IP stack. See "Appendix B—TCP/IP Stack Data Structures" on page 47 for detailed data structure information.

`byte`	8-bit unsigned char
`SCHAR`	8-bit signed char
`word16`	16-bit unsigned integer
`word32`	32-bit unsigned integer
`PCONST_BYTE`	Pointer to a constant byte (*const unsigned char**)

`Socket Data Types:`

`PSOCKET_INFO`	Pointer to a *SOCKET_INFO_T* structure

`Virtual File System Data Types:`

`VF_PTR`	Pointer to a *VF* structure
`POST_FP`	Function pointer to a CGI content creation function
`PF_PTR`	Pointer to a *POST_FUNCS* structure

`SMTP Mail Client Data Types:`

`PSMTP_INFO`	Pointer to an *SMTP_INFO_T* structure

`DHCP/BOOTP Data Types and Global Variables:`

`ip_src_addr`	Global 4-byte array containing the IP address
`eth_src_hw_addr`	Global 6-byte array containing the Ethernet MAC address
`subnet_mask`	Global 4-byte array containing the Subnet Mask
`gateway_ip_addr`	Global 4-byte array containing the default Gateway IP address
`dhcp_info`	Global variable of type *DHCP_INFO_T*
`dhcp_lease`	Global variable of type *DHCP_LEASE_T*
`PBOOTP_INFO`	Pointer to a *BOOTP_INFO_T* structure

4.3. Important Notes

- **The *mn_init()* function must be called prior to calling any other stack function.**
- **To ensure the project builds correctly, the TCP/IP Libraries must be the last two items in the linker build list.** The linker build list order can be viewed from the *Project→Target Build Configuration →Customize Button →Files to Link* Tab. Projects generated by the TCP/IP Configuration Wizard automatically place the library files at the end of the linker build list; however, any additional files added to the project or any overloaded library functions may be inserted before or after the library. **The user should check the linker build list order after adding any new files or after overloading any library functions.**
- If the framework code generated by the TCP/IP Configuration Wizard is built without adding additional application code, warnings about uncalled callback functions should be expected. These warnings will not appear if application code contains calls into the application layer stack functions, such as `mn_server()`. If application code does not call into any application layer stack functions, the uncalled functions can be deleted to remove warnings.
- **If the total program code size is greater than 64 kB, the project must be set up for code banking.** In a code banked project, the total code size of the common area cannot exceed 32 kB. See application note “AN130: Code Banking Using the Keil 8051 Tools” for more information about code banking. The TCP/IP Configuration Wizard can automatically configure the project for code banking if the code banked project option is selected.
- In a code banked project, any code constants accessed by the virtual file system (e.g. HTML content, binary files, etc.) must reside in the common bank or in the same bank as mn_stack_bank_000.lib.
- To locate code constants in a code bank, the BANKx (x = 1,2,3) directive must be used. By default, the linker places all code constants in the common area. When the code banked project option is selected, the TCP/IP Configuration Wizard automatically places the file array for "index.html" in code bank 1 by adding the following to the linker command line:

  BANK1(8000H, ?CO?INDEX (8000H))

  See the linker manual for additional information about the BANKx directive.
- In order for the TCP/IP project to build correctly, the linker command line must include `NOOL XD(10h)`. The `NOOL` command line option may be removed if HTTP is not enabled.
- The default buffer sizes have been optimized for performance. To recover some of the memory used by the buffers, the buffer sizes can be reduced by changing the configuration constants in the *mn_userconst.h* header file.
- The TCP/IP stack uses interrupts and automatically enables interrupts. Functions in the TCP/IP stack should never be called from a high priority interrupt. It is also not recommended to call stack functions from a low priority interrupt.
- On devices that have SFR Paging, the SFR page is changed and restored by hardware when an interrupt occurs. This feature should not be disabled.
- If an MCU in the 'F34x family is used, and the *c8051F340_usb_fifo* bit in *mn_userconst.h* is set to 1, the transmit buffer is located in USB FIFO space starting at address 0x400. When this option is selected, the transmit buffer cannot be made larger than 1024 bytes. Note that if this option is selected, the system clock must be 24.5 MHz or less.

If using Ethernet as the physical layer:

- **The *mn_ether_init()* function must be called after *mn_init()*, before calling any other stack function.**
- The TCP/IP stack uses Timer0 and sets the prescaler shared by both Timer0 and Timer1 to SYSCLK/48 ('F12x, 'F13x, 'F34x) or SYSCLK/12 ('F02x). Timer1 is available for use by application code but is limited to using SYSCLK or the same prescaler as Timer0 for its time base. The SFRs affecting Timer0 should not be changed after *mn_init()* is called.
- If the system clock is changed from the default value, the **th0_flash:tl0_flash** variable in *mn_userconst.h* must be adjusted to allow the TCP/IP stack to accurately maintain time. The th0_flash:tl0_flash variable is used as the reload value for Timer0, which should overflow every 10 ms. On 'F12x, 'F13x, and 'F34x devices, this 16-bit value should be set to (-SYSCLK/48/100).
- On 'F02x devices, this 16-bit value should be set to (-SYSCLK/12/100).
- The TCP/IP stack contains device-specific code. The *device_id* constant in *mn_userconst.c* must specify the correct device for the TCP/IP stack to operate properly. The device_id may be set to C8051F120, C8051F340, or C8051F020 to specify a device in the 'F12x or 'F13x, 'F34x, and 'F02x family, respectively.
- If the CP220x is selected as the Ethernet controller, Interrupt 0 must be configured as a level-triggered interrupt. The TCP/IP stack automatically enables the interrupt (as low priority), but the initialization code should ensure that INT0 is enabled in the Crossbar ('F02x, 'F12x, 'F13x) or the IT01CF register is properly configured ('F34x).

If using a dial-up modem as the physical layer:

- Modem support is only available for the C8051F12x and C8051F13x devices.
- The TCP/IP stack uses Timer0, Timer1, UART1, and PCA0. SFR registers related to these peripherals should not be modified after calling `mn_init()`.
- The TCP/IP stack enables the PCA0 interrupt in normal priority and the UART1 interrupt in high priority. These interrupts are not available to application code.
- The TCP/IP Configuration Wizard automatically configures the UART baud rate and system time base based on the selected system clock frequency. If the system clock initialization routine is modified to change the system clock frequency, the following constants in *mn_userconst.h* must be manually changed:
  - **th0_flash:tl0_flash**—The Timer0 reload value in MODE1 (16-bit timer) with a time base of system clock divided by 48. This 16-bit value should be set so that the timer overflows in 10 ms (100 Hz). For example, if the system clock is 98 MHz, this value would be set to (–98000000/48/100) = –20416 = 0xB040.
  - **uart_reload**—The 8-bit UART1 reload value derived from Timer1 in 8-bit auto reload mode. The time base for UART1 is the system clock. The recommended baud rates and reload values for selected system clock frequencies are shown in Table 1.

Table 1. UART1 Baud Rate Selection

System Clock	Baud Rate	Reload Value
3.0625 MHz	245760 bps	0xFA
24.5 MHz	307200 bps	0xD9
49 MHz	307200 bps	0xB1
98 MHz	307200 bps	0x61

Note: The TCP/IP Configuration Wizard can generate the appropriate reload values for any system clock configuration. To prevent overwriting an existing project, direct the Wizard's output to a temporary folder.

4.4. Socket Functions

The TCP/IP stack uses sockets to send or receive data over the network. A socket is a data structure that contains information about the data that is sent or received. When a packet is received, the TCP/IP stack verifies the destination IP address and port number. If it finds an open socket that matches the protocol (UDP or TCP) and port number, it will add the received data to the socket's buffer and notify the application software. Otherwise, the packet will be discarded. The socket data structure *SOCKET_INFO_T* is defined in <techpubs: add link (Section + Page Number) to SOCKET_INFO_T paragraph in appendix B >.

The TCP/IP stack is initialized, and sockets are opened and closed through the following functions:

Note: The only required socket function in all projects that use the TCP/IP stack is `mn_init()`. When using application layer services, such as HTTP Web Server, FTP Server, or TFTP client, sockets are automatically opened and closed as needed without management from application code.

4.4.1. mn_init

Description: Performs all initialization required by the TCP/IP stack.

Note: This function should be called prior to calling any other stack function.

Prototype: `int mn_init (void);`

Example Call: `mn_init();`

Return Value: Returns *TRUE* if initialization was successful or negative number on failure.

4.4.2. mn_open

Description: Allocates and optionally opens a TCP or UDP socket.

Note: Modem and PPP connections must be established prior to opening a TCP socket if using a modem as the physical layer.

Prototype: `SCHAR mn_open(byte[],word16, word16, byte, byte, byte, byte *, word16);`

Example Call:
```
socket_no =
mn_open(dest_ip,src_port,dest_port,open_mode,proto,type,recv_buff,buff_len);
```

Parameters:

1. *dest_ip*—Destination IP address to which packets are being sent.
2. *src_port*—The port number used by the application. This must be a well known port number (see RFC 1700) or a number larger than 1024. If set to 0, the TCP/IP stack will not automatically assign a random port number.
3. *dest_port*—The port number used by the remote side, if known. If the remote port number is not known, *dest_port* should be set to zero. If the destination port is set to zero, it will be filled in automatically by the TCP/IP stack.
4. *open_mode*—Used only by TCP sockets. Can be one of the following values:
   - *ACTIVE_OPEN*—A TCP connection is initiated by the TCP/IP stack.
   - *PASSIVE_OPEN*—The TCP/IP stack waits for the remote side to initiate a TCP connection.
   - *NO_OPEN*—The TCP/IP stack places the socket into a listen state, but does not wait for a TCP connection. Select this mode for UDP sockets.
5. *proto*—Defines the socket type. Can be one of the following values:
   - *PROTO_TCP*—A TCP socket is opened.
   - *PROTO_UDP*—A UDP socket is opened.
6. *type*—Should be set to *STD_TYPE*.
7. *recv_buff*—Address of the buffer used to store the received data.
8. *buff_len*—Size of the buffer used to store the received data.

Return Value: If successful, returns a valid socket number between 0 and 126. The `MK_SOCKET_PTR()` macro can be used to obtain a pointer to the newly-opened socket. Otherwise, returns one of the following error codes:

- *NOT_SUPPORTED*—A socket was requested for an unsupported protocol.
- *NOT_ENOUGH_SOCKETS*—No sockets are available.
- *TCP_OPEN_FAILED*—Attempt to open a TCP socket has failed.

4.4.3. mn_send

Description: Sends data on a previously opened socket. When sending a TCP packet, this function does not successfully return until an acknowledgement for the sent packet has been received.

Notes:
1. If the socket is TCP, a call to this function may cause data to be received. The socket's *recv_len* field should be checked for values greater than zero after each call to this function.
2. If using a dedicated socket in addition to application layer services, such as HTTP or FTP, this function should not be called after adding data to the socket. The socket will be checked, and data will be automatically sent by *mn_server()*. If this function is called from a callback function while *mn_server()* is running, it will result in the same packet being transmitted twice.

Prototype: `int mn_send(SCHAR, byte *, word16);`

Example Call: `status = mn_send(socket_no, msg_ptr, msg_len);`

Parameters:
1. *socket_no*—The socket number returned from `mn_open()`.
2. *msg_ptr*—Address of the buffer containing data to send.
3. *msg_len*—Number of bytes to send.

Return Value: If successful, returns the number of bytes sent. If the number of bytes sent is zero, the packet needs to be resent. Otherwise, returns one of the following error codes (all negative values):

- *BAD_SOCKET_DATA*—An invalid socket number was passed to the function.
- *SOCKET_NOT_FOUND*—The socket number passed belongs to an inactive socket.
- *TCP_ERROR*—The packet was sent more than *TCP_RESEND_TRYS* times without receiving an ACK (TCP sockets only).
- *TCP_TOO_LONG*—An attempt was made to send a packet that is larger than the available TCP window (TCP sockets only).
- *TCP_NO_CONNECT*—Cannot send because a TCP connection is not established.
- *DHCP_LEASE_EXPIRED*—The DHCP lease has expired.

4.4.4. mn_recv

Description: Receives data on a previously opened socket with a fixed wait time of *SOCKET_WAIT_TICKS*. The wait time is in units of 10 ms system ticks. This function is typically not called if *mn_server()* is running. Application code can process received packets using the *callback_app_server_process_packet()* callback function.

Notes:

1. This function will loop until a packet is received on the passed socket or a timeout occurs. The `callback_app_recv_idle()` callback function will be continuously called while waiting for a packet to arrive. This function will stop waiting and return immediately if `callback_app_recv_idle()` returns *NEED_TO_EXIT*.
2. When a TCP or UDP packet is successfully received, the sender's IP Address and Port number are copied into the socket's *dest_ip* and *dest_port* fields. To read this information, a pointer to the socket can be obtained by calling the `MK_SOCKET_PTR()` macro with the socket number as an argument. When a TCP or UDP packet is successfully received, the socket is bound to the sender's IP address and Port number. The socket cannot send or receive data to/from any other sender until the socket is reset. A socket may be reset by closing and re-opening it using the *mn_close()* and *mn_open()* library routines.
3. For TCP sockets, responses to TCP control packets, such as SYN and FIN, will be automatically sent. This function may return *NEED_TO_LISTEN* indicating that the socket should listen for a reply from the remote side rather than send another packet. This routine only waits for packets on the specified socket. Any packet received which is not addressed to the specified socket will be discarded. For multiple socket applications, we recommend using *mn_server()* and the *callback_app_server_process_packet()* callback function to receive data on multiple sockets simultaneously.

Prototype:
```
int mn_recv(SCHAR, byte *, word16);
```

Example Call:
```
status = mn_recv(socket_no, buff_ptr, buff_len);
```

Parameters:

1. *socket_no*—The socket number returned from `mn_open()`.
2. *buff_ptr*—Address of the buffer to hold received data.
3. *buff_len*—Size of the receive buffer.

Return Value: If successful, returns the number of bytes received. Otherwise, returns one of the following error codes (all negative values):

- *BAD_SOCKET_DATA*—An invalid socket number was passed to the function.
- *SOCKET_NOT_FOUND*—The socket number passed belongs to an inactive socket.
- *SOCKET_TIMED_OUT*—A socket timeout occurred without receiving a packet.
- *NEED_TO_LISTEN*—A reply to the received packet was automatically sent, and the socket should wait for an answer (TCP sockets only).
- *NEED_TO_EXIT*—The callback function `callback_app_recv_idle()` returned *NEED_TO_EXIT*.
- *LINK_FAIL*—The CP220x has been disconnected from the network
- Any other negative number; there was a checksum or FCS error or the TCP connection is closed.

4.4.5. mn_recv_wait

Description: Same as `mn_recv()` except uses the wait time passed as the fourth parameter.

Prototype: `int mn_recv_wait(SCHAR, byte *, word16, word16);`

Example Call: `status = mn_recv_wait(socket_no, buff_ptr, buff_len, wait_ticks);`

Parameters:
1. *socket_no*—The socket number returned from `mn_open()`.
2. *buff_ptr*—Address of the buffer to hold received data.
3. *buff_len*—Size of the receive buffer.
4. *wait_ticks*—Number of system ticks to wait before a timeout.

Return Value: See description for `mn_recv()`.

4.4.6. mn_close

Description: Closes a previously opened socket.

Prototype: `int mn_close(SCHAR);`

Example Call: `status = mn_close(socket_no);`

Parameters: **1.** *socket_no*—The socket number returned from `mn_open()`.

Return Value: If successful, returns *FALSE*. Otherwise, returns one of the following error codes (all negative values):

- *BAD_SOCKET_DATA*—An invalid socket number was passed to the function.
- *SOCKET_NOT_FOUND*—The socket number passed belongs to an inactive socket.

4.4.7. mn_abort

Description: Immediately closes a previously opened socket without negotiating a close or sending a FIN (TCP only). The `mn_close()` and `mn_abort()` functions are identical for UDP sockets.

Prototype: `int mn_abort(SCHAR);`

Example Call: `status = mn_abort(socket_no);`

Parameters: **1.** *socket_no*—The socket number returned from `mn_open()`.

Return Value: If successful, returns *FALSE*. Otherwise, returns one of the following error codes (all negative values):

- *BAD_SOCKET_DATA*—An invalid socket number was passed to the function.
- *SOCKET_NOT_FOUND*—The socket number passed belongs to an inactive socket.

4.4.8. mn_find_socket

Description: Used to obtain a pointer to a socket matching the passed source port, destination port, destination IP address, and socket type.

Prototype: `PSOCKET_INFO mn_find_socket(word16, word16, byte*, byte);`

Example Call: `socket_ptr = mn_find_socket(src_port, dest_port, dest_ip, socket_type);`

Parameters:

1. *src_port*—The local port number.
2. *dest_port*—The remote port number.
3. *dest_ip*—The IP address of the remote machine.
4. *socket_type*—The socket type specified when opening the socket. Can be one of the following values:
   - *PROTO_TCP*—A TCP socket.
   - *PROTO_UDP*—A UDP socket.

Return Value: If successful, returns a pointer to the matching socket. Otherwise, returns *PTR_NULL*.

4.5. Ethernet Functions

When using Ethernet as the physical layer, the TCP/IP stack requires initializing the Ethernet controller prior to calling any other functions. The following functions and global bits are provided by the TCP/IP stack to manage the physical layer Ethernet controller:

Functions

Global Bits

link_status	See description for *mn_ether_init()*
ether_reset	See description for *mn_ether_init()*
link_lost	See description for *mn_ether_init()*
flash_busy	See description for *CPFLASH_ByteWrite()*

4.5.1. ether_reset_low

Description: Sets the CP220x's reset pin low. This function allows the user to change the port pin used to reset the CP2200. This function is defined in *main.c* and called from *mn_ether_init()* by the TCP/IP stack.

Prototype: `void ether_reset_low(void);`

Example Call: `Call: ether_reset_low();`

4.5.2. ether_reset_high

Description: Sets the CP220x's reset pin high. This function allows the user to change the port pin used to reset the CP2200. This function is defined in *main.c* and called from *mn_ether_init()* by the TCP/IP stack.

Prototype: `void ether_reset_high(void);`

Example Call: `ether_reset_high();`

4.5.3. mn_ether_init

Description: Resets and initializes the Ethernet controller.
If the CP220x is selected as the Ethernet Controller, the following tasks are performed:

- The CP2200 is reset, and reset initializations are performed.
- Specific CP220x registers are read to verify presence of the Ethernet Controller.
- CP2200 Interrupts are enabled.
- The MAC address is programmed.
- The device is configured for half or full duplex operation, or auto-negotiation is started.
- The global *link_status* bit is set to indicate a good link or cleared to indicate that the device is not plugged into a network. The *link_status* bit is only valid after *mn_ether_init()* has been called for the first time. After this, it is always valid as long as Interrupt 0 and global interrupts are enabled.
- The global *ether_reset* bit is cleared. This bit will be set any time the CP220x enters, then exits the reset state. If *ether_reset* is ever set, the link status bit becomes invalid until *mn_ether_init()* is called. User code should not perform any network operations until the Ethernet controller is re-initialized. If this bit is frequently set, then check the board and verify that the power supply meets the current demands of the Ethernet controller.
- The global *link_lost* bit is set to indicate that the CP220x has lost link. It remains set if the link returns. This bit is cleared when *mn_ether_init()* succeeds.

Prototype: `int mn_ether_init(byte, byte, byte);`

Example Call: `status = mn_ether_init(duplex_mode, num_autoneg_attempts, loopback);`

Parameters:

1. *duplex_mode*—Selects Full-Duplex, Half-Duplex, or Auto-Negotiation. Can be one of the following values:
   - FULL_DUPLEX—The Ethernet controller is configured to full-duplex mode.
   - HALF_DUPLEX—The Ethernet controller is configured to half-duplex mode.
   - AUTO_NEG—Auto-Negotiation selects between full-duplex and half-duplex modes.
2. *num_autoneg_attempts*—Specifies the number of times to attempt autonegotiation. If set to 0, and autonegotiation is enabled, it will not return until autonegotiation is successful.
3. *loopback*—Set to TRUE to enable internal loopback. Set to FALSE to disable internal loopback.

Return Value: If successful, returns FULL_DUPLEX or HALF_DUPLEX. Otherwise, returns one of the following negative error codes:

- INVALID_DUPLEX_MODE — A duplex mode other than the three allowed values was passed in parameter 1.
- INVALID_MAC_ADDRESS — Returned if the MAC address is FF:FF:FF:FF:FF:FF.
- LINK_FAIL — A valid link was not detected. The global *link_status* bit can now be polled to determine when the Ethernet controller has been plugged into a network.
- ETHER_INIT_ERROR — A hardware error has occurred.

4.5.4. CPFLASH_ByteRead

Description: Reads a single byte from the specified Flash address in the CP220x.

Prototype: `byte CPFLASH_ByteRead(word16);`

Example Call: `flash_value = CPFLASH_ByteRead(addr);`

Parameters: **1.** addr -- The address of the Flash byte to read.

Return Value: Returns the value of the Flash byte at address *addr*.

4.5.5. CPFLASH_ByteWrite

Description: Writes a single byte to the specified Flash address in the CP220x.
Note: This function initiates a Flash Write operation and sets the global *flash_busy* flag to true. The *flash_busy* flag can be polled to determine when the Flash operation is complete.

Prototype: `void CPFLASH_ByteWrite(word16, byte);`

Example Call: `Call: CPFLASH_ByteWrite(addr, flash_value);`

Parameters: **1.** addr -- The address of the Flash byte to write.

2. flash_value -- The value to write to Flash.

4.5.6. CPFLASH_PageErase

Description: Erases a single 512-byte Flash page at the specified address in the CP220x.
Note: This function initiates a Flash Erase operation and sets the global *flash_busy* flag to true. The *flash_busy* flag can be polled to determine when the Flash operation is complete.

Prototype: `void CPFLASH_PageErase(word16, byte);`

Example Call: `CPFLASH_PageErase(addr);`

Parameters: **1.** addr -- The address of the Flash page to erase. Any address on a Flash page will erase the entire 512-byte page.

4.6. Modem Functions

When using a modem as the physical layer, the TCP/IP stack requires establishing a connection with the modem prior to establishing connections using higher level protocols, such as PPP or TCP. Establishing a connection with the modem causes it to dial and log in to a remote network or accept incoming calls. The following functions are provided by the TCP/IP stack to manage the physical layer connection with the modem:

4.6.1. mn_modem_connect

Description: Establishes a connection between the MCU and the modem and performs modem initialization.

Modem initialization sequence for Answer Mode:

1. Initialize country code and protocol; then, send the *MODEM_INIT_ANSWER* string.
2. Wait for “OK”, “RING”, and “CONNECT” string sequence.
3. If *USE_PASSWORD* is set to 1 and *USE_PAP* is set to 0, the *ANS_LOGIN_PROMPT* and *ANS_PASSWORD_PROMPT* will be sent, and the return strings will be verified against the user name and password stored in *LOGIN_NAME* and *PASSWORD*.

Modem initialization sequence for Dial Mode:

1. Initialize country code and protocol; then, send the *MODEM_INIT_DIAL* string.
2. Wait for “OK”; then, send *MODEM_DIAL*.
3. Wait for “CONNECT”
4. If *USE_PASSWORD* is set to 1 and USE_PAP is set to 0, the user name and password stored in *LOGIN_NAME* and *PASSWORD* will be used to log in to the remote server

Note: This function initializes the modem using the strings defined in *mn_userconst.h*. The default maximum string length is 10 characters.

Prototype: `int mn_modem_connect(byte);`

Example Call: `status = mn_modem_connect(connect_mode);`

Parameters:
1. *connect_mode*—Determines whether the modem will be configured to answer incoming calls or initiate outgoing calls. Can be one of the following values:
   - *ANSWER_MODE*—The modem is configured to answer incoming calls.
   - *DIAL_MODE*—The modem is configured to dial into a remote server or ISP.

Return Value: If successful, returns *TRUE*. Otherwise, returns a negative number to indicate that a connection could not be established.

4.6.2. mn_modem_disconnect

Description: Closes the connection between the MCU and the modem and causes the modem to disconnect from the phone line.

Note: All TCP sockets and PPP connections should be closed prior to calling this function.

Prototype: `void mn_modem_disconnect(void);`

Example Call: `mn_modem_connect();`

4.6.3. mn_modem_send_string

Description: Sends a variable initialization string to the modem.

Note: The string must end in a carriage return ('\r').

Prototype: `void mn_modem_send_string(PCONST_BYTE, word16);`

Example Call: `mn_modem_send_string(str, len);`

Parameters:
1. *str*—Address to the first character in a constant byte array (e.g., "ATM1L1\r").
2. *len*—The number of bytes in str, including the carriage return ('\r').

4.6.4. mn_modem_wait_reply

Description: Waits for a specific response from the modem with a specified timeout.

Note: The timeout is specified in 10 ms system ticks.

Prototype: `int mn_modem_wait_reply(PCONST_BYTE, word16, word16);`

Example Call: `status = mn_modem_wait_reply(str, len, timeout);`

Parameters:
1. *str*—Address to the first character in a constant byte array. The response received from the modem is compared to this string to determine success or failure.
2. *len*—The number of bytes in *str*, including the carriage return ('\r').
3. *timeout*—The maximum number of 10 ms system ticks to wait without receiving a response from the modem. This function will return failure on a timeout condition.

Return Value: If successful, returns *TRUE*. Otherwise, returns a negative number to indicate that a timeout has occurred or the modem response did not match the contents of *str*.

4.7. PPP Functions

The TCP/IP stack allows a choice between PPP and SLIP for the data link layer when a modem is used as the physical layer.

Note: If Password Authentication Protocol (PAP) is enabled, application code should add username and password entries to the PAP table prior to opening a PPP connection. If PAP is disabled, authentication should be enabled at the modem level. See Section 4.6.1 on page 19 for more information about authentication at the physical layer.

4.7.1. mn_ppp_open

Description: Establishes a PPP connection with a remote PPP client/server.

Notes:

1. All modem connections and stack initialization must be complete prior to calling this function.
2. The *USE_PAP* constant determines whether or not password authentication protocol will be used. If *USE_PAP* is set to *TRUE*, the `mn_ppp_add_pap_user()` must be called prior to calling this function. If *USE_PAP* is set to *FALSE*, login information is handled by `mn_modem_connect()` using the information specified in *mn_userconsts.h*.

int mn_ppp_open(byte);

Example Call: *status = mn_ppp_open(open_mode);*

Parameters:

1. *open_mode*—Determines if the local PPP will be a server or a client. Can be one of the following values:
   - *ACTIVE_OPEN*—The local PPP client attempts to establish a connection with a remote PPP server. The first username/password combination added using `mn_ppp_add_pap_user()` will be used to login to the remote server.
   - *PASSIVE_OPEN*—The local PPP server waits for a remote PPP client to initiate a connection. All username/password combinations added using `mn_ppp_add_pap_user()` will be checked.

Return Value: If successful, returns *TRUE*. Otherwise, returns *FALSE*.

4.7.2. mn_ppp_close

Description: Closes a PPP connection without waiting for a response and resets the PPP state machine.

Note: This function should only be called if a `mn_ppp_open()` was successful.

Prototype: `void mn_ppp_close(void);`

Example Call: `mn_ppp_close();`

4.7.3. mn_ppp_reset

Description: Resets the PPP state machine.

This function should only be called if an error condition exists that does not allow a *mn_ppp_close*.

Prototype: `void mn_ppp_reset(void);`

Example Call: `mn_ppp_reset();`

4.7.4. mn_ppp_add_pap_user

Description: Adds a username/password pair to the password authentication protocol (PAP) table.

Note: The default maximum string length is twenty characters including the null terminator.

Prototype: `byte mn_ppp_add_pap_user(char*, char*);`

Example Call: `status = mn_ppp_add_pap_user(username, password);`

Parameters:
1. *username*—Null terminated character string containing the user name.
2. *password*—Null terminated character string containing the password.

Return Value: If username/password pair is successfully added, returns *TRUE*. Otherwise, returns *FALSE*.

4.7.5. mn_ppp_del_pap_user

Description: Removes a username/password pair from the password authentication protocol (PAP) table.

Note: The default maximum string length is twenty characters including the null terminator.

Prototype: `byte mn_ppp_del_pap_user(char*);`

Example Call: `status = mn_ppp_del_pap_user(username);`

Parameters:
1. *username*—Null terminated character string containing the user name of the username/password pair to be deleted.

Return Value: If username/password pair successfully removed, returns *TRUE*. Otherwise, returns *FALSE* indicating that the user name was not found.

4.8. DHCP/BOOTP Functions

When using Ethernet as the physical layer, the TCP/IP stack allows the MCU to specify or obtain an IP address in three ways:

- Specifying the static IP address in *mn_userconst.h.*
- Changing the IP address in the 4-byte global array `ip_src_addr[]`.
- Obtaining a dynamic IP address using the DHCP or BOOTP functions described in this section.
- Obtaining a static IP address through ping gleaning. Ping gleaning allows the MCU to set its IP address to the address inside a ping packet if and only if the first byte of the current address is zero and the device is pinged directly using its MAC address. Note: The device must be inside mn_server() to accept and respond to Ping packets.

If DHCP is used, application code should monitor the state of the IP address lease in the global structure *dhcp_lease* and renew the lease as needed. If application code calls `mn_send()` or `mn_server()`, the DHCP lease will be automatically renewed. Below is a summary of the DHCP/BOOTP functions available:

Notes:

1. The MAC address for the Ethernet controller specified in *mn_userconst.h* is overwritten by the actual MAC Address if the CP220x is selected as the Ethernet Controller.
2. DHCP/BOOTP cannot be used if a modem is selected as the physical layer.

4.8.1. mn_dhcp_start

Description: Obtains a new IP address using DHCP and writes it to the 4-byte global array `ip_src_addr[]`. The global `gateway_ip_addr[]` and `subnet_mask[]` arrays are also updated with data from the DHCP transaction.

If the DHCP server provides a boot file name, it is copied into the global *dhcp_info* structure. After this function returns successfully, the boot file contents can be retrieved from the server using TFTP. See “Appendix B—TCP/IP Stack Data Structures” on page 47 for a definition of the global *dhcp_info* structure.

Note: If multiple boot files exist on the network, a specific boot file name can be specified as the first parameter to `mn_dhcp_start()`. DHCP servers typically discard the request if the passed file name does not exactly match the name of an existing boot file. In most implementations, PTR_NULL will be passed as the first parameter.

Prototype:

```
int mn_dhcp_start(byte*, word32);
```

Example Call:

```
status = mn_dhcp_start(boot_file_name, request_time);
```

Parameters:

1. *boot_file_name*—Null-terminated character array containing the requested boot file name. This parameter should be set to PTR_NULL unless a specific boot file name is known.
2. *request_time*—The requested lease time in seconds. The actual lease time provided by the DHCP server can be read from the global *dhcp_lease* structure after this function returns. See “Appendix B—TCP/IP Stack Data Structures” on page 47 for more information on the global *dhcp_lease* structure.

Return Value: Returns TRUE if successful. Otherwise, returns one of the following error codes:

- *DHCP_ERROR*—An error occurred while processing the DHCP packets.
- Any Negative Number—Could not establish a connection with a DHCP server.

4.8.2. mn_dhcp_release

Description: This function is used to release a lease obtained with a successful call to `mn_dhcp_start()` and may be called any time before the lease expires. After the lease has been released or has expired, packets cannot be sent or received until a new lease is obtained using `mn_dhcp_start()`.

Prototype: `int mn_dhcp_release(void);`

Example Call: `status = mn_dhcp_release();`

Return Value: Returns TRUE if successful. Otherwise, returns one of the following error codes:

- *FALSE*—Could not find an active DHCP session.
- *DHCP_ERROR*—An error occurred while processing the DHCP packets.
- Any Negative Number—Could not establish a connection with a DHCP server.

4.8.3. mn_dhcp_renew

Description: This function is used to renew a lease obtained with a successful call to `mn_dhcp_start()` and may be called any time before the lease expires. After the lease has been released, or has expired, packets cannot be sent or received until a new lease is obtained using `mn_dhcp_start()`.

Note: Software monitoring the global *dhcp_lease* structure should use the *lease_time* member variable to determine the number of seconds remaining in the lease and renew the lease before it expires.

Prototype: `int mn_dhcp_renew(word32);`

Example Call: `status = mn_dhcp_renew(request_time);`

Parameters:

1. *request_time*—The requested lease time in seconds. The actual lease time provided by the DHCP server can be read from the global *dhcp_lease* structure after this function returns.

Return Value: Returns TRUE if successful. Otherwise, returns one of the following error codes:

- *DHCP_ERROR*—An error occurred while processing the DHCP packets.
- Any Negative Number—Could not establish a connection with a DHCP server.

4.8.4. mn_bootp

Description: Uses BOOTP to obtain an IP address, gateway IP address, subnet mask, and, optionally, a boot file name from a BOOTP server when first starting up. The 4-byte global `gateway_ip_addr[]` and `subnet_mask[]` arrays are updated with data from the BOOTP transaction.

The global `ip_src_addr[]` array is not automatically updated. After the BOOTP transaction, application software should copy the assigned address stored in the *yiaddr* field of the *BOOTP_INFO_T* structure to the `ip_src_addr[]` array.

BOOTP allows the MCU to request a specific IP address from the BOOTP server. The address in the `ip_src_addr[]` array prior to calling this function will be requested if the *bootp_request_ip* constant in *mn_userconst.h* is set to 1. Otherwise, the MCU will request the next available address. If the requested address is not available, the BOOTP server assigns any available address.

If the BOOTP server provides a boot file name, it is copied into the *BOOTP_INFO_T* structure passed as the second parameter. After this function returns successfully, the boot file contents can be retrieved from the server using TFTP. See “Appendix B—TCP/IP Stack Data Structures” on page 47 for a definition of the *BOOTP_INFO_T* structure.

Note: If multiple boot files exist on the server, a specific boot file name can be specified as the first parameter to `mn_dhcp_start()`. BOOTP servers typically discard the request if the passed file name does not exactly match the name of an existing boot file. In most implementations, PTR_NULL will be passed as the first parameter.

Prototype: `int mn_bootp(byte*, BOOTP_INFO_T*);`

Example Call: `status = mn_bootp(boot_file_name, pBOOTP_INFO);`

Parameters:
1. *boot_file_name*—Null-terminated character array containing the requested boot file name. This parameter should be set to *PTR_NULL* unless a specific boot file name is known.
2. *pBOOTP_INFO*—Pointer to an empty *BOOTP_INFO_T* structure. This structure will be filled if the BOOTP transaction succeeds.

Return Value: Returns TRUE if successful. Otherwise, returns one of the following error codes:

- *FALSE*—Did not receive a reply from the BOOTP server.
- *DHCP_ERROR*—An error occurred while processing the DHCP packets.
- Any Negative Number—Could not establish a connection with the BOOTP server.

4.9. Application Layer Functions

The TCP/IP stack provides access to application layer services, such as HTTP Web Server, FTP Server, TFTP Client, and SMTP Mail Client, through the following functions:

These functions, in conjunction with callback and virtual file system functions described in the next two sections, provide complete control over application layer services. Figure 1 shows the typical software flow for starting application layer services.

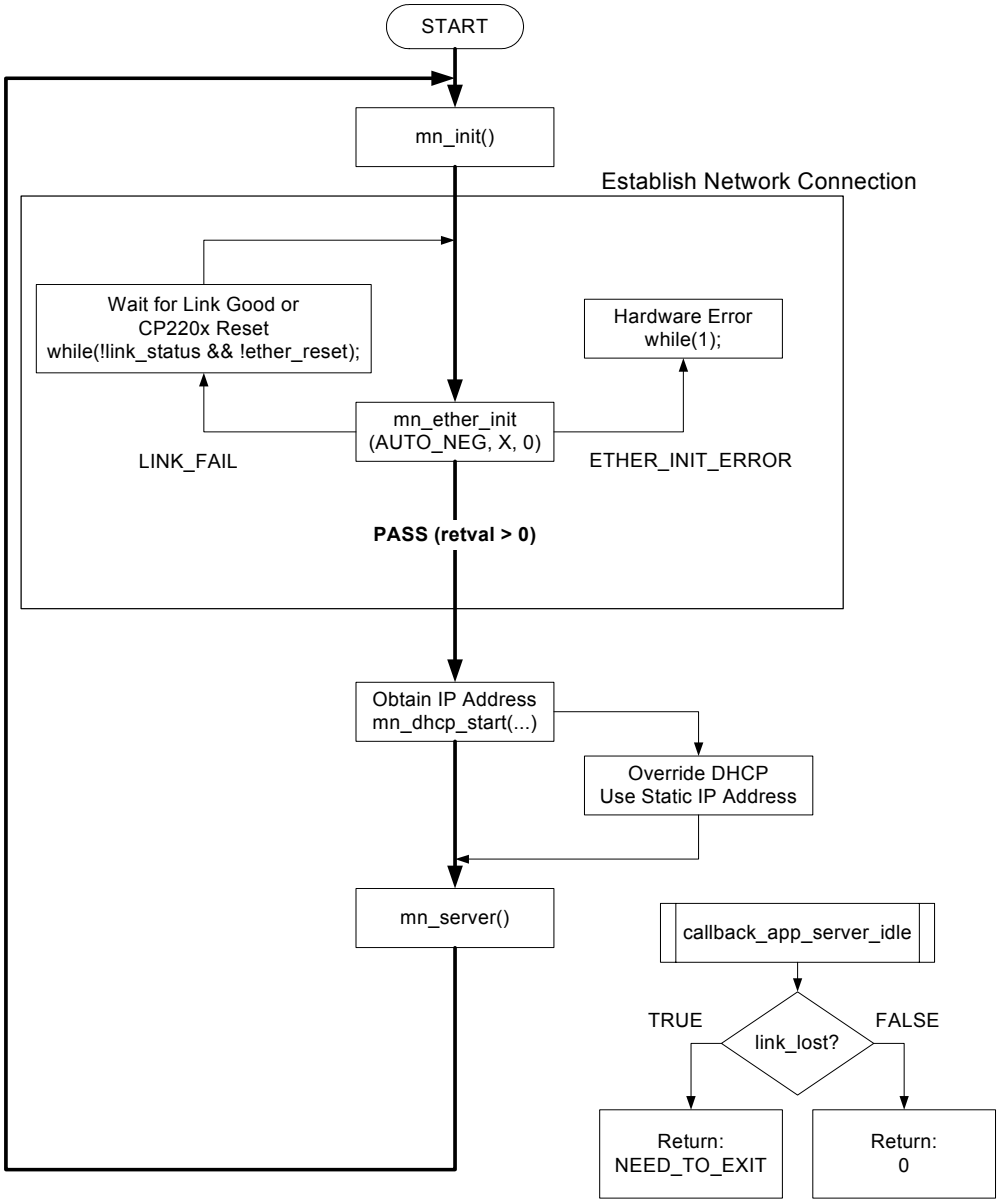

Figure 1. Typical Software Flow when Using *mn_server()*

4.9.1. mn_server

Description: Used to start application layer services. When called, all enabled server applications, such as HTTP Web Server and FTP Server, will be started. Client applications, such as SMTP Mail Client and TFTP Client, are started using the functions described in this section. This function will not return until a DHCP Lease Expires, a PPP error occurs, or a callback function (`callback_app_server_idle()` or `callback_app_server_process_packet()`) returns *NEED_TO_EXIT*.

Notes:

1. This function will automatically open and close sockets as needed to handle incoming requests. Any additional sockets, such as UDP sockets, that are used by application software when the HTTP or FTP server is idle should be opened prior to calling this function.
2. If the TCP/IP Library contains DHCP, *mn_server()* will not start unless the device has been able to acquire an IP address through DHCP or DHCP has been overridden. DHCP may be overridden as follows:

   dhcp_lease.infinite_lease = 1;

   dhcp_lease.dhcp_state = DHCP_OK;

   Important notes about the FTP Server:

   - The FTP server has been designed to work with Windows GUI and command-line-based FTP clients. The FTP server returns directory listings in Unix Standard Format. If multiple formatting options are available in the FTP client, Unix Standard Format should be selected.
   - FTP commands supported are USER, QUIT, RETR, STOR, DELE, PORT, TYPE, MODE, STRU, NOOP, PWD, LIST, and, optionally, PASS. The FTP server will always check for the user name and password defined in the ftp_user array in the *mn_vars.c* source file. This array must be initialized with all the allowable user names and passwords at the time of compiling.
   - The virtual file system does not use subdirectories; therefore, PWD always returns "/", and CWD is not allowed.
   - The FTP server uses a buffer for temporary storage whose size is set by the *ftp_buffer_len* constant in *mn_userconst.h*. This buffer should be large enough to hold the largest file you expect to receive. After a file has been received, memory is allocated for it using `malloc()`, and a virtual file system entry with the memory segment, *VF_PTYPE_DYNAMIC*, is created for that file. Deleting a file from the virtual file system will free any dynamically-allocated memory associated with the file.

Prototype: `int mn_server(void);`

Example Call: `status = mn_server();`

Return Value: The following are valid return values:

- *FALSE*—Either `callback_app_server_idle()` or `callback_app_server_process_packet()` returned *NEED_TO_EXIT*.
- *PPP_LINK_DOWN*—PPP connection was terminated.
- *DHCP_LEASE_EXPIRED*—The DHCP lease has expired.

4.9.2. mn_http_find_value

Description: Searches “field-name=field-value-&” pairs for the passed field-name and copies the decoded field-value into the passed buffer. CGI content creation functions use this routine to determine the value of variables sent from the web page.

Note: In most cases, the *source_ptr* parameter will be set to the global variable *BODYptr.*

Prototype: `int mn_http_find_value(byte*, byte*, byte*);`

Example Call: `status = mn_http_find_value(source_ptr, field_name, field_value);`

Parameters:
1. *source_ptr*—Address to buffer containing the message body to be searched.
2. *field_name*—Null terminated search string containing the field-name.
3. field_value—String buffer where the field-value is copied.

Return Value: Returns *TRUE* if the field-name is found. Otherwise returns *FALSE*.

4.9.3. mn_tftp_get_file

Description: Gets a file from a remote TFTP server and stores it in the specified buffer.

Prototype: `long mn_tftp_get_file(byte*, byte*, byte*, long);`

Example Call: `num_bytes = mn_tftp_get_file(ip_addr, filename, buffer, buff_len);`

Parameters:
1. *ip_addr*—Pointer to a 4-byte character array containing the IP address of the TFTP server.
2. *filename*—Null-terminated search string containing the file name.
3. *buffer*—Pointer to a buffer in RAM to hold the file.
4. *buff_len*—Number of bytes in the buffer.

Return Value: Returns the number of bytes received. Otherwise, returns a negative number.

4.9.4. mn_smtp_start_session

Description: Opens a TCP connection with the SMTP server specified in *mn_userconst.h*.

Note: The physical layer must be initialized prior to calling this function.

Prototype: `SCHAR mn_smtp_start_session(word16);`

Example Call: `socket_num = mn_smtp_start_session(port);`

Parameters:
1. *port*—The port number to be used by the SMTP socket. Can be between 1025 and 65535.

Return Value: Returns a socket number on success or a negative number on error.

4.9.5. mn_smtp_end_session

Description: Closes the connection to an SMTP server opened with `mn_smtp_start_session()`.

Prototype: `void mn_smtp_start_session(SCHAR);`

Example Call: `socket_num = mn_smtp_end_session(socket_num);`

Parameters: **1.** *socket_num*—The socket number returned from `mn_smtp_start_session()`.

4.9.6. mn_smtp_send_mail

Description: Sends an e-mail message with an optional attachment to an SMTP mail server.

Note: A call to `mn_smtp_start_session()` must return successful prior to sending an e-mail.

Prototype: `int mn_smtp_send_mail(SCHAR, PSMTP_INFO);`

Example Call: `status = mn_smtp_send_mail(socket_num, mail_info_ptr);`

Parameters: **1.** *socket_num*—The socket number returned from `mn_smtp_start_session()`.
2. *mail_info_ptr*—Address of a *SMTP_INFO_T* structure that has been initialized.

Return Value: Returns zero or a positive number on success and a negative number on error.

4.9.7. mn_dns_get_addr

Description: Issues a domain name service request to the DNS server specified in *ip_dns_addr[]*. The domain name service provides the IP address of the specified domain name. A domain name uses numbers and letters. An IP address uses only numbers.

Note: Recursive DNS searches are not supported.

Prototype: `int mn_dns_get_addr(char*, byte*);`

Example Call: `status = mn_dns_get_addr(name, addr_ptr);`

Parameters: **1.** *name*—Pointer to a null-terminated string containing the domain name to look up. The string must not contain any '.' characters and should include the length of each segment at the beginning of the segment. For example,

The string: www.silabs.com

Should be encoded as: 0x03, w, w, w, 0x06, s, i, l, a, b, s, 0x03, c, o, m, /0

2. addr_ptr—Address of the buffer that will receive the IP address. This buffer should be at least 4 bytes long.

Return Value: Returns zero or a positive number on success and one of the following negative error codes on error:

- SOCKET_ALREADY_EXISTS—The specified DNS socket is already opened.
- NOT_ENOUGH_SOCKETS—There are no available sockets for the DNS query.
- DHCP_LEASE_RENEWING—DHCP is renewing or rebinding, so only DHCP packets may be sent.
- DNS_BUFFER_OVERFLOW—The DNS buffer is not large enough to hold the response.
- DNS_NOT_FOUND—The server did not respond with a Host Address.
- DNS_COUNT_ERROR—There were no answers in the DNS response from the server.
- DNS_ID_ERROR—The Identification field of the response did not match the Identification field of the query.
- SOCKET_TIMED_OUT—The socket timed out without receiving a response to the DNS query.
- UDP_BAD_CSUM—There was a bad checksum on the UDP packet received from the DNS server.

4.10. Callback Functions

The TCP/IP stack uses callback functions to notify application code of various events. Figure 2 shows the callback function code execution flow. The following four callback functions should be defined in every project that uses application layer services provided by the TCP/IP stack. The callback functions should contain the appropriate event handling code.

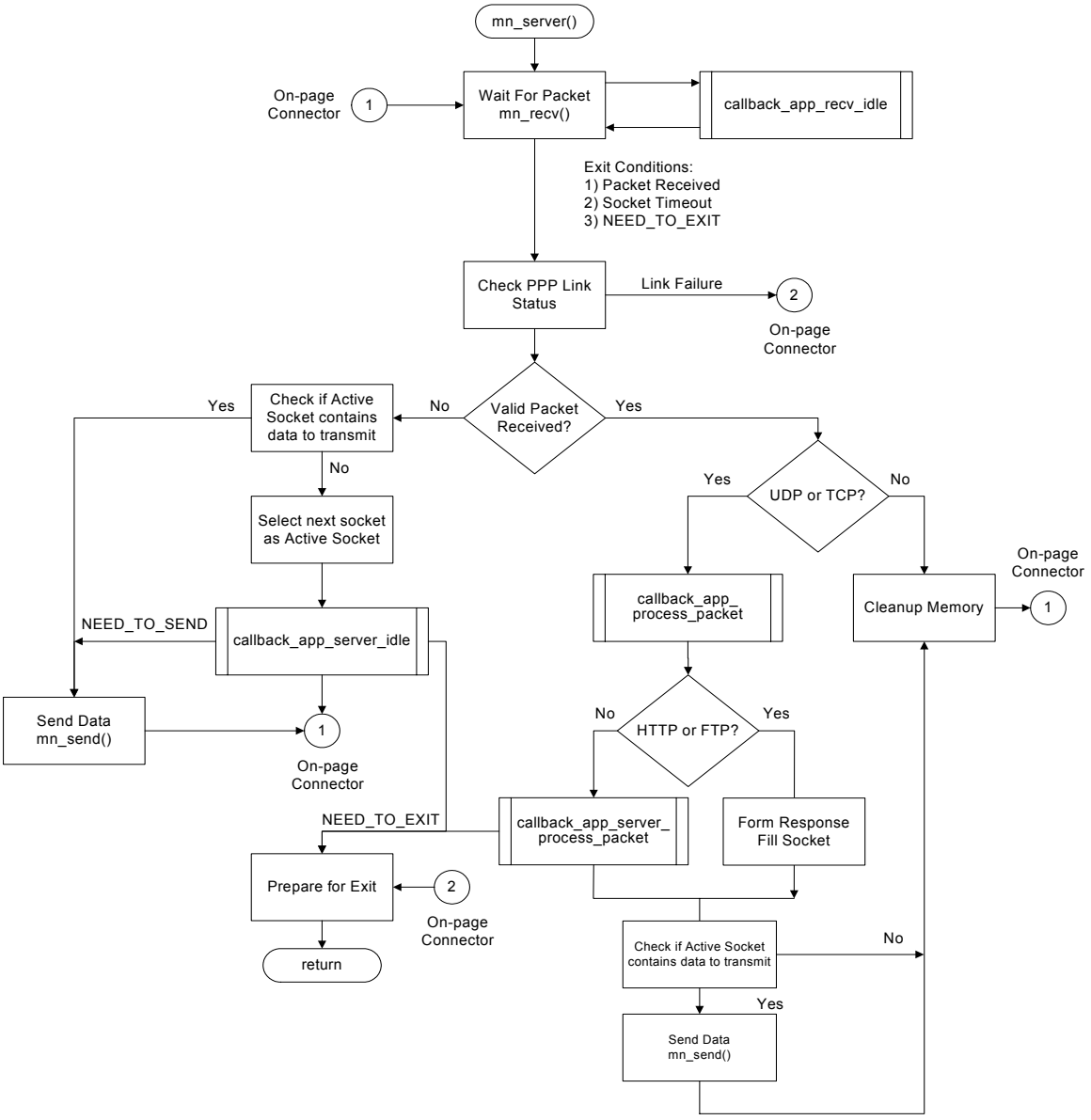

Figure 2. Callback Function Flow Diagram

4.10.1. callback_app_process_packet

Description: Called by the TCP/IP stack after any TCP or UDP packet is received.

Note: The return value is ignored for UDP packets.

Prototype: `byte callback_app_process_packet(PSOCKET_INFO);`

Example Call: `status = callback_app_process_packet(socket_ptr);`

Parameters: **1.** *socket_ptr*—Pointer to the socket that contains the data.

Return Value: The following are valid return values:

- *NEED_IGNORE_PACKET*—The TCP/IP stack will not ACK the TCP packet.
- Any Other Value—The TCP/IP stack will ACK the TCP packet.

4.10.2. callback_app_server_process_packet

Description: Called by the TCP/IP stack after any TCP or UDP packet that is not HTTP or FTP is received. HTTP and FTP packets are automatically handled by the server.

Prototype: `SCHAR callback_app_process_packet(PSOCKET_INFO);`

Example Call: `status = callback_app_process_packet(socket_ptr);`

Parameters: **1.** *socket_ptr*—Pointer to the socket that contains the data.

Return Value: The following are valid return values:

- *NEED_TO_EXIT*—The `mn_server()` routine will exit immediately, returning control to the `main()` routine.
- Any Other Value—The server will discard the packet.

4.10.3. callback_app_recv_idle

Description: Called repeatedly while `mn_recv()` is waiting for data. This function should only be used for low priority tasks. Any high priority tasks should be placed in an interrupt service routine.

Prototype: `SCHAR callback_app_recv_idle(void);`

Example Call: `status = callback_app_recv_idle();`

Return Value: The following are valid return values:

- *NEED_TO_EXIT*—The `mn_recv()` routine will exit immediately. If the server is running, it will stop waiting for data and advance to the next state.
- Any Other Value—The `mn_recv()` routine will continue to wait for data.

4.10.4. callback_app_server_idle

Description: Periodically called from `mn_server()` when it is not transmitting or receiving data. This function should only be used for low-priority tasks. Any high-priority tasks should be placed in an interrupt service routine.

Prototype: `SCHAR callback_app_server_idle(PSOCKET_INFO*);`

Example Call: `status = callback_app_server_idle(psocket_ptr);`

Parameters: **1.** *psocket_ptr*—Pointer to a pointer to a socket that can be used for transmitting data.

Note: The socket handle may be reassigned to a different socket (e.g., **psocket_ptr* = *new_socket_ptr*;).

Return Value: The following are valid return values:

- *NEED_TO_SEND*—The TCP/IP stack will immediately send the data stored in the socket.
- *NEED_TO_EXIT*—The `mn_server()` routine will exit immediately, returning control to the `main()` routine.
- Any Other Value—The `mn_server()` routine will continue to function normally.

4.10.5. callback_socket_empty

Description: Called after all data in a TCP socket has been sent. This callback function allows the user's application to send additional data using the same TCP connection.

Note: This callback function allows application code to send large amounts of non-contiguous data using a small memory buffer.

Prototype: *callback_socket_empty(PSOCKET_INFO);*

Example Call: *callback_socket_empty(socket_ptr);*

Parameters: **1.** *socket_ptr*-Pointer to a socket that can be used for transmitting data.

4.10.6. callback_socket_closed

Description: Called after a TCP socket has been closed. This callback function alerts the user's application that the TCP connection to the remote host is closed and that the socket may now connect to a different host.

Note: If used in conjunction with callback_socket_empty to send non-contiguous data, application code should free resources associated with the closed socket.

Prototype: *callback_socket_closed(SCHAR);*

Example Call: *callback_socket_closed(socket_no);*

Parameters: **1.** *socket_no*-Socket number for the closed socket.

4.11. Virtual File System (VFILE) Functions

The TCP/IP stack includes a virtual file system accessible by application code or application level services, such as the HTTP Web Server and FTP server. The files added to the file system can be requested by a web browser or FTP Client and are stored as binary arrays in Flash or Ram. This allows images, applets, and other content to be embedded inside static or dynamic HTML pages.

To add static content to the virtual file system, it must first be converted to a file array using the HTML2C utility. The HTML2C utility reads a content file (e.g., image.gif) and generates two files (e.g., *image.c* and *image.h*) that can be added to the project. The static content file can be added to the file system in three steps:

1. Include the header file (e.g., *image.h*) at the beginning of *main.c* using the #include directive.
2. Add the C source file (e.g., *image.c*) to the project build.
3. Add the file to the file system during runtime using the `mn_vf_set_entry()` or `mn_vf_set_ram_entry()`. These functions map the starting address and length of the file array to a file name that is accessible from a web browser or FTP client.
4. Modify the *num_vf_pages* constant in *mn_userconst.h* such that the value is greater than or equal to the total number of files arrays added to the file system.

The following functions can be used to add, remove, or access static files in the file system.

HTTP/FTP Server File System Functions:

The virtual file system allows dynamic web page content creation through CGI scripting. When the HTTP Web Server receives a recognized script name, it calls a “content creation” function to generate the requested content. Requests to the HTTP Web Server can be sent in as part of an HTML form or directly in the URL. Below is an example of a web browser requesting dynamic data from a script called “*get_data*”:

http://10.10.10.163/get_data?type=temperature

In a CGI script request passed through the URL, all text after the question mark is interpreted as arguments passed to the script. In the above example, “type” is considered a field-name, and “temperature” is the field-value. Multiple field-name/field-value arguments can be separated by an ampersand. The script name, *get_data*, is recognized by the HTTP Server because it has been added to the file system by application code. CGI scripts can be added and removed from the file system using the following functions:

CGI Script Functions:

The `mn_pf_set_entry()` function maps a script name to a “content creation” function pointer that is called each time the script name appears in the URL or in the ACTION field of an HTML form. The “content creation” function uses the arguments following the question mark to generate the requested data. Once the function is finished generating data, it specifies the starting address and length of the data it wishes to send back to the browser. The TCP/IP stack handles all further communication with the web browser until a new request is received. See “AN292: Embedded Ethernet System Design Guide" for a discussion of application development with the TCP/IP Library.

4.11.1. mn_vf_get_entry

Description: Used to obtain a pointer to the *VF* structure corresponding to a file in the virtual file system. The *VF* structure contains information about the file, such as starting address, file size, and memory segment. See “Appendix A—TCP/IP Stack User Constants” on page 43 for a definition of the *VF* structure.

Note: This function should not be called from an ISR.

Prototype: `VF_PTR mn_vf_get_entry(byte*);`

Example Call: `pVF = mn_vf_get_entry(filename);`

Parameters: **1.** *filename*—Null terminated string containing the name of the desired file (e.g., *index.html*).

Return Value: Returns a valid pointer to a VF structure or *PTR_NULL* if the search string did not match any file names added to the file system.

4.11.2. mn_vf_set_entry

Description: Used to add a file stored in on-chip Flash to the virtual file system.

Notes:

1. This function should not be called from an ISR.
2. If storing files in the CP220x Flash, only addresses less than 0x1FFA should be used. The address range 0x1FFA to 0x1FFF contains the pre-programmed MAC address. Addresses 0x2000 and above are invalid.

Prototype: `VF_PTR mn_vf_set_entry(byte*, word16, PCONST_BYTE, byte);`

Example Call: `pVF = mn_vf_set_entry(filename, file_size, file_ptr, mem_seg);`

Parameters: **1.** *filename*—Null terminated string containing the file name (e.g., “index.html”).

2. *file_size*—Number of bytes in the file.

3. *file_ptr*—Pointer to the start of the file.

4. *mem_seg*—Type of memory where the file is stored (should be set to *VF_PTYPE_FLASH*).

- *VF_PTYPE_FLASH*—The file is stored in MCU Flash.
- *VF_PTYPE_CP2200_FLASH*—The file is stored in CP220x Flash.

Return Value: Returns a valid pointer to a VF structure or *PTR_NULL* if the maximum number of files has already been added to the file system.

4.11.3. mn_vf_set_ram_entry

Description: Used to add a file stored in RAM to the virtual file system.

Note: This function should not be called from an ISR.

Prototype: `VF_PTR mn_vf_set_ram_entry(byte*, word16, byte*, byte);`

Example Call: `pVF = mn_vf_set_ram_entry(filename, file_size, file_ptr, mem_seg);`

Parameters: **1.** *filename*—Null terminated string containing the file name (e.g., *index.html*).

2. *file_size*—Number of bytes in the file.

3. *file_ptr*—Pointer to the start of the file.

4. *mem_seg*—Type of memory where the file is stored (should be set to zero).

Return Value: Returns a valid pointer to a *VF* structure or *PTR_NULL* if the maximum number of files has already been added to the file system.

4.11.4. mn_vf_del_entry

Description: Used to remove a file from the virtual file system. Files removed from the virtual file system will not be visible to the HTTP or FTP server. The FTP server stores received files in dynamically allocated RAM. If a deleted file is stored in dynamically-allocated RAM, the memory buffer will be freed.

Note: This function should not be called from an ISR.

Prototype: `SCHAR mn_vf_del_entry(byte*);`

Example Call: `status = mn_vf_del_entry(filename);`

Parameters: **1.** *filename*—Null terminated string containing the name of the desired file (e.g., *index.html*).

Return Value: Returns one of the following values:

- *TRUE*—The file was successfully removed.
- *FALSE*—The file was not found.
- *VFILE_ENTRY_IN_USE*—The file was in use and could not be removed.

4.11.5. mn_pf_get_entry

Description: Used to obtain a function pointer to a CGI content creation function.

Note: This function should not be called from an ISR.

Prototype: `POST_FP mn_pf_get_entry(byte*);`

Example Call: `function_ptr = mn_pf_get_entry(function_name);`

Parameters: **1.** *function_name*—Null terminated string containing the name of the desired function.

Return Value: Returns a valid function pointer to a CGI content creation function or *PTR_NULL* if the search string did not match any function names added to the file system.

4.11.6. mn_pf_set_entry

Description: Used to add a CGI content creation function to the virtual file system.

Note: This function should not be called from an ISR.

Prototype: `PF_PTR mn_pf_set_entry(byte*, POST_FP);`

Example Call: `pPFStruct = mn_pf_set_entry(name, function_ptr);`

Parameters: **1.** *function_name*—Null terminated string containing the name of the desired function.
2. *function_ptr*—Pointer to the start of the function.

Return Value: Returns a valid pointer to a *POST_FUNCS* structure or *PTR_NULL* if the maximum number of functions has already been added to the file system. See “Appendix A—TCP/IP Stack User Constants” on page 43 for a definition of the PF structure.

4.11.7. mn_pf_del_entry

Description: Used to remove a CGI content creation function from the virtual file system.

Note: This function should not be called from an ISR.

Prototype: `byte mn_pf_del_entry(byte*);`

Example Call: `status = mn_pf_del_entry(function_name);`

Parameters: **1.** *function_name*—Null terminated string containing the name of the desired function.

Return Value: Returns *TRUE* if the function was removed or *FALSE* if the function name was not found.

4.12. Support Functions

The TCP/IP stack provides the following support functions used for string conversion:

4.12.1. mn_ustoa—unsigned int to ascii

Description: Converts an unsigned integer to an ascii string.

Note: This function should not be called from an ISR.

Prototype: `byte mn_ustoa(byte*, word16);`

Example Call: `num_bytes = mn_ustoa(dest_buff, source);`

Parameters: **1.** *dest_buff*—Address to a character array to store the null-terminated string result.
2. *source*—The unsigned integer that will be converted to a string.

Return Value: Returns the number of bytes added to *dest_buff*.

4.12.2. mn_uctoa—unsigned char to ascii

Description: Converts an unsigned character to an ascii string.

Note: This function should not be called from an ISR.

Prototype: `byte mn_uctoa(byte*, word16);`

Example Call: `num_bytes = mn_uctoa(dest_buff, source);`

Parameters: **1.** *dest_buff*—Address to a character array to store the null-terminated string result.
2. *source*—The unsigned char that will be converted to a string.

Return Value: Returns the number of bytes added to dest_buff.

4.12.3. mn_getMyIPAddr_func

Description: Fills a string with the current IP address in the following format: “255.255.255.255”.

Note: This function should not be called from an ISR.

Prototype: `word16 mn_getMyIPAddr_func(byte**);`

Example Call: `num_bytes = mn_getMyIPAddr_func(dest_buff);`

Parameters: **1.** *dest_buff*—Pointer to pointer to a character array to store the null-terminated IP address string.

Return Value: Returns the number of bytes added to *dest_buff*.

4.12.4. mn_atous—ascii to unsigned int

Description: Converts an ascii string to an unsigned integer.

Note: This function should not be called from an ISR.

Prototype: `word16 mn_atous(byte*);`

Example Call: `result = mn_atous(src_buff);`

Parameters: **1.** *src_buff*—Address to a null-terminated string that will be converted.

Return Value: Returns an unsigned integer representing the value in the string.

5. Netfinder Protocol

The Netfinder protocol allows a PC application to search for embedded systems on a network. The PC application finds the embedded systems by broadcasting an "Identity Request" packet to all nodes on the local network. Each embedded system that supports Netfinder replies with an "Identity Reply" packet which contains information that identifies and differentiates it from other embedded systems on the network. This information includes: IP address, Elapsed time from an event (e.g. Time Powered, Time on Network), MAC Address, and two text descriptions of the device. This information can be customized for each application.

The primary benefit of the Netfinder protocol is to reduce the amount of hardware required to place an embedded system on a network. For DHCP enabled systems, it eliminates the requirement of an LCD to display the IP address. For embedded systems on a static network, Netfinder can eliminate the need for a separate UART interface or push-button switches used to program the IP address.

The preferred port for Netfinder is UDP 3040, however, any available UDP port may be used. Table 2 lists the packet format for the Identity Request and Reply packets:

Table 2. Broadcast Identity Request—4 Bytes

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Version/Type 0x00	Reserved 0x00	Random Sequence Identifier

Table 3. Identity Reply—Variable Length

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Version/Type 0x01	Alert Level 0x00 – Device OK 0x01 – No Address/Warn 0xFF – Device Failure	Replication of Random Sequence Identifier
Event 1 Days		Event 1 Hours / Event 1 Minutes
Event 2 Days		Event 2 Hours / Event 2 Minutes
Event 1 Seconds	Event 2 Seconds	MAC Address (Most Significant Octets)
MAC Address (Least Significant Octets)		
IP Address		
Subnet Mask		
Default Gateway		
Four Variable Length Null-Terminated Strings String A – Name/Type of Embedded System (~ 20 characters) String B – Description/Miscellaneous Text (~ 30 characters) String C – Event 1 Description, String D – Event 2 Description		

When using Netfinder in an embedded system on a static network, an IP address may be assigned to the embedded system. The PC application sends an identity assignment packet to the embedded system. The embedded system replies with an acknowledgement stating success or failure.

In order to send a packet to an embedded system that does yet have an IP address, it may be necessary for the PC application first assign it an IP address using Ping Gleaning. Ping Gleaning is a method of specifying an embedded system's IP address by pinging it. It can be used by adding a static ARP entry to the PC's ARP table, then pinging the embedded system. If the MAC address matches the embedded system's MAC address, then the embedded system will respond to future packets sent to the new IP address.

The benefit of using Netfinder, instead of only Ping Gleaning, is that the PC application receives an acknowledgement from the embedded system and is able to program other important fields such as the subnet mask and the default gateway.

Table 4. Identity Assignment—24 Bytes

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Version/Type 0x02	Reserved 0x00	Random Sequence Identifier
IP Address		
Subnet Mask		
Default Gateway		
MAC Address (Most Significant Octets)		
MAC Address (Least Significant Octets)		Reserved – All Zeros

Table 5. Identity Assignment Acknowledgement—4 Bytes

0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15	16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Version/Type 0x03	Response Code 0x01 – Address Accepted 0x00 – Address Rejected due to MAC mismatch 0xFF – Other Error	Replication of Random Sequence Identifier

6. Custom Driver Support

The TCP/IP library supports a custom driver mode that allows the TCP/IP stack to be used with any Ethernet controller. The custom driver mode can be selected in the TCP/IP Configuration Wizard by checking the “Custom Ethernet Device” option as the communications adapter. Both polled mode and interrupt-driven drivers are supported. When the custom driver mode is selected, the TCP/IP Configuration Wizard outputs two additional files containing a custom driver template: *custom_ethernet.c* and *custom_ethernet.h*. These files are automatically added to the generated project.

The generated files are provided as a template and will not compile until all the missing pieces have been filled in by the user. The sections of the generated files that require modification by the user can be identified by a large comment block with instructions inside. The steps required to turn the custom driver template into a functioning driver are described in the following paragraphs.

6.1. Modifying the Custom Driver Header File

The *custom_ethernet.h* file should be modified to include the register names and addresses of the Ethernet controller. The Ethernet driver communicates with an Ethernet controller over the external memory interface (EMIF). The EMIF is driven by reading and writing to a pointer in pdata space named *ETHER_RDRW*. This pointer points to the first direct mapped register in the Ethernet controller. For example, if the first direct mapped register is at 0x2000, the address high byte (named *BASE_ADDRESS* and defined in mn_userconst.h) is set to 0x20, and the address low byte (named *ETHER_RDRW* and defined in *custom_ethernet.h*) is set to 0x00.

During initialization, the TCP/IP stack writes the value of *BASE_ADDRESS* to EMI0CN. *ETHER_RDRW* is defined as a constant using the compiler directive *“#define”*. The remaining register definitions (such as *TX_PORT*, *RX_PORT*, etc.) are defined as offsets from the *ETHER_RDRW* and can be accessed using the following notation:

Example Read from RX_PORT:

```
temp_data = ETHER_RDRW[RX_PORT];
```

Example Write to TX_PORT:

```
ETHER_RDRW[TX_PORT] = temp_data;
```

6.2. Modifying the *ether_init()* Routine

The `ether_init()` routine should be modified to reset and initialize the Ethernet controller. Device initialization may include:

1. Reseting the Ethernet controller using its reset pin or a software reset.
2. Verifying communication with the Ethernet controller.
3. Enabling/disabling features of the Ethernet controller as desired. This includes configuring the Duplex mode (half/full duplex) or enabling Auto-Negotiation, enabling transmission/reception, and configuring the receive filter.
4. Writing the MAC address stored in `eth_src_hw_addr[]` to registers on the controller. If a different MAC address is used, the `eth_src_hw_addr[]` array should be changed to reflect the new address.
5. Returning TRUE on success or ETHER_INIT_ERROR to indicate an initialization failure.

6.3. Modifying the *ether_send()* Routine

The `ether_send()` routine is called by the TCP/IP stack when it needs to send an Ethernet packet. An Ethernet packet consists of a preamble and start of frame delimiter, 14-byte Ethernet header, and the payload. The preamble and start of frame delimiter are automatically generated by most Ethernet controllers. The ethernet header and payload are provided to this function by the TCP/IP stack.

The `ether_send()` routine has access to the following four pieces of information:

- Starting address of the Ethernet header (*send_out_ptr*, a global pointer).
- Length of the Ethernet header (*head_buff_len*, calculated from *send_out_ptr* - *send_in_ptr*).
- Starting address of the payload (*data_buff_ptr*, obtained from passed socket).
- Length of the payload (*data_buff_len*, passed as an argument).

The `ether_send()` routine should be modified to perform the following functions:

1. Send the Ethernet header by writing *head_buff_len* bytes starting at *send_out_ptr* to the Ethernet controller. The global *send_out_ptr* should be incremented after each byte is transmitted.
2. Send the payload by writing *data_buff_len* bytes starting at *data_buff_ptr* to the Ethernet controller.
3. Initiate packet transmission.
4. Wait for the transmission to complete.
5. If writing a polled mode driver, call the *MN_XMIT_BUSY_CLEAR* macro to clear the transmit busy bit.
6. Return the number of bytes transmitted on success or *ETHER_SEND_ERROR* on failure.

6.4. Modifying the *ether_recv() or ether_poll_recv()* Routine

Depending on the polled-mode/interrupt-mode selection in the TCP/IP Configuration Wizard, different receive functions will be generated. The `ether_poll_receive()` function will be generated in both modes but should only be modified if using polled mode. The `ether_recv()` will only be generated in interrupt mode.

When using polled mode, the `ether_poll_recv()` routine is periodically called by the TCP/IP stack to determine if the Ethernet controller has received any new packets. The function polls the Ethernet controller for the time interval defined by *ETHER_WAIT_TICKS* then returns *SOCKET_TIMED_OUT* if a packet has not been received. The user should modify the routine with code that checks if a packet has been received.

When using interrupt mode, the `ether_recv()` function is called by the interrupt handler after a packet has been received and should perform the functions below. The `ether_poll_recv()` should also perform the functions below in addition to checking for received packets.

1. Determine if the packet was received with any errors. If errors are detected, discard the packet from the Ethernet controller's buffer and return *ETHER_RECV_ERROR*.
2. Determine the length of the received packet. The Frame Check Sequence (FCS) field should not be included in the length calculation. Write the length to the variable *recv_len*. The code provided in the template will determine if there is space in the TCP/IP stack's receive buffer for the new packet. If there is no space in the buffer, the packet will be discarded and the function will return *ETHER_RECV_ERROR*.
3. If there is sufficient free space in the receive buffer, code execution will vector to a loop which will guide the user's code to copy the packet into the receive buffer. The user needs to provide a single line of code in three places to copy a byte from the Ethernet controller into the address pointed to by *recv_in_ptr*.
4. Perform any finalization if required by the Ethernet controller. This may include clearing a valid bit, discarding any extra bytes, such as the FCS, etc.
5. Return the number of bytes received. This step will only be reached on successful execution of the above steps.

6.5. Modifying the *ether_ISR() Interrupt Handler*

The `ether_ISR()` routine is only generated when interrupt mode is selected in the TCP/IP Configuration Wizard. This interrupt is mapped to /INT0 but can be changed to another interrupt by modifying the initialization code and the interrupt number of the function. The interrupt handler should perform the following functions:

1. Determine the cause of the interrupt.
2. Call `ether_recv()` if a packet has been received.
3. Call the *MN_XMIT_BUSY_CLEAR* macro to clear the transmit busy bit if a transmit complete event has occurred.
4. Service any other Ethernet controller specific events (e.g. link failure, FIFO Overflow, etc.).

APPENDIX A—TCP/IP STACK USER CONSTANTS

The TCP/IP stack user constants located in the *mn_userconst.h* header file allow the user to customize the stack according to the requirements of the application. Most of the constants in the *mn_userconst.h* header file are also configurable using the TCP/IP Configuration Wizard.

The number of user constants in the *mn_userconst.h* file will vary based on the generated library. Tables 6 and 7 list all possible constants; however, the header file generated will only contain constants that are applicable to the generated library.

Table 6. IP and MAC Address Configuration Constants

Constant Name	Description
IP_SRC_ADDR	IP address for the MCU.
IP_DEST_ADDR	IP address for the destination, if known.
IP_SMTP_ADDR	IP address for the SMTP server.
ETH_SRC_HW_ADDR	MAC Address for the Ethernet controller. Not applicable for the CP220x.
ETH_DEST_HW_ADDR	Destination MAC address if not using ARP.
GATEWAY_IP_ADDR	IP Address of the default gateway or router.
SUBNET_MASK_ADDR	Subnet mask for the network used to determine whether an address belongs to a device connected to the local network or must go through the router to an external network.

Table 7. Default Modem Settings

Constant Name	Description
MODEM_COUNTRY_CODE	Modem initialization string to set the country code. Usually contains AT commands and must end in a carriage return (‘\r’).
MODEM_PROTOCOL	Modem initialization string to set the protocol. Usually contains AT commands and must end in a carriage return (‘\r’).
MODEM_INIT_DIAL	Modem initialization string used when making an outgoing call. Usually contains AT commands and must end in a carriage return (‘\r’).
MODEM_INIT_ANSWER	Modem initialization string used when configuring the modem to receive calls. Usually contains AT commands and must end in a carriage return (‘\r’).
MODEM_DIAL	Modem initialization string containing the outgoing phone number. Usually contains AT commands and must end in a carriage return (‘\r’).
LOGIN_NAME	Login name to use when logging into a remote modem or the login name to check for when a remote modem is logging into the local modem. This constant is only used if PAP is disabled.
PASSWORD	Password to use when logging into a remote modem or the login name to check for when a remote modem is logging into the local modem. This constant is only used if PAP is disabled.
DIAL_LOGIN_PROMPT	Login prompt to expect from a remote modem when logging in. This constant is only used if PAP is disabled.

Table 7. Default Modem Settings (Continued)

Constant Name	Description
DIAL_PASSWORD_PROMPT	Password prompt to expect from a remote modem when logging in. This constant is only used if PAP is disabled.
ANS_LOGIN_PROMPT	Login prompt to provide to a remote modem when answering a call. This constant is only used if PAP is disabled.
ANS_PASSWORD_PROMPT	Password prompt to provide to a remote modem when answering a call. This constant is only used if PAP is disabled.

Table 8. TCP/IP Stack Adjustments

Constant Name	Description
arp_auto_update	When set to 1, the ARP cache is updated after every valid packet is received. The ARP cache is always updated on PING requests.
arp_cache_size	Number of entries in the ARP cache.
arp_keep_ticks	Number of system ticks to keep entries in the ARP cache.
arp_resend_trys	Number of times an ARP packet is retransmitted.
arp_wait_ticks	Number of system ticks to wait for an ARP packet.
BASE_ADDRESS	The high byte of the Ethernet controller base address on the external memory interface bus. For example, if the controller is at address 0x2000, BASE_ADDRESS should be set to 0x20. This value is written to EMI0CN.
body_buffer_len	Size of the buffer to hold characters from an HTTP GET request following the question mark.
device_id	Specifies the MCU. Can be C8051F120, C8051F340, or C8051F020 if Ethernet is used as the physical layer. Must be C8051F120 if a modem is selected as the physical layer.
dns_buff_size	Size of the DNS receive buffer. This should be set to the maximum expected size for received DNS packets. The DNS specification specifies a maximum length of 512 bytes, however, this buffer may be set to a smaller value if the actual maximum packet size is known.
dns_send_trys	Number of times DNS should attempt to send a query before returning a failure.
dns_wait_ticks	Number of 10ms system ticks to wait before retransmitting a DNS query. This value should be set between 2 to 5 seconds. The default value is 400 (equivalent to 4 seconds).
EMIF_TIMING	The value written to EMI0TC to set the external memory bus timing.
ether_wait_ticks	Number of system ticks to wait for an Ethernet packet.
ftp_buffer_len	FTP Receive Buffer Size. Must be large enough to hold the largest expected file size.

Table 8. TCP/IP Stack Adjustments (Continued)

Constant Name	Description
ftp_max_param	Size of the buffer to hold received command line parameters. This value must be at least 23.
ftp_num_users	Number of username/password combinations to store. If set to zero, authentication will not be performed.
http_buffer_len	Buffer used to process HTTP includes. Should be the same size as TCP window.
ip_time_to_live	Sets the “time to live” field in the IP packet.
mem_pool_size	RAM memory pool available to the `malloc()` function.
multicast_ttl	Sets the “time to live” field in an IP packet for multicast packets.
num_post_funcs	This value is the number of entries in the post-function table in the virtual file system. The value can be 1 to 255.
num_sockets	Sets the number of sockets that can be used. The value must be between 1 and 127. Each socket uses approximately 46 bytes of XRAM.
num_vf_pages	The number of entries in the directory table in the virtual file system. Can be 1 to 255.
pap_num_users	Number of entries in the PAP table.
ping_buff_size	If PING is enabled, this value is the size of the data from a PING request that can be stored. Nine bytes are added to this value to store part of the PING request header. If the PING request contains more data than the specified value, the packet will be discarded and no reply sent. The default value is 32.
ppp_resend_ticks	Number of system ticks to wait before retransmitting a PPP packet.
ppp_resend_trys	Number of times to send a PPP packet before terminating connection.
ppp_terminate_trys	Number of times to a PPP-Terminate request is sent before resetting connection.
recv_buff_size	Sets the size of the buffer used for reception.
smtp_buffer_len	This value is the size of the temporary buffer for SMTP commands, it must be at least 46. The recommended value is *TCP_WINDOW*.
socket_wait_ticks	Number of 10 ms system ticks to wait for a packet.
tcp_resend_ticks	Number of system ticks to wait before retransmitting a TCP packet.
tcp_resend_trys	Number of times a TCP packet is transmitted before aborting the connection.

Table 8. TCP/IP Stack Adjustments (Continued)

Constant Name	Description
tcp_window	This value is both the amount of data you are willing to accept from the remote connection and the amount of data you are sending in a single packet. This value must be greater than 0 and less than or equal to 1460. A larger value yields better throughput but requires larger buffers. **Note:** The TCP/IP Stack uses a fixed window when receiving, not a sliding window as specified in RFC 793. If using PPP, the *RECV_BUFF_SIZE* and *XMIT_BUFF_SIZE* should be at least double the TCP window to allow for escaped characters. If using ethernet, the *RECV_BUFF_SIZE* and *XMIT_BUFF_SIZE* should be at least TCP_WINDOW + 58.
tftp_resend_trys	Number of times a TFTP packet is transmitted before terminating the connection.
tl0_flash th0_flash	Timer 0 reload values such that Timer 0 overflows in 10 ms. This defines a system tick.
uart_reload	Reload value for the UART. The maximum standard UART baud rate is automatically selected by the TCP/IP Configuration Wizard.
uri_buffer_len	Size of the buffer to hold characters from an HTTP GET request preceding the question mark.
use_password	When set to 1, user authentication is performed at the modem level. Should be set to zero if PAP is used for authentication.
xmit_buff_size	Sets the size of the buffer used for transmission.

APPENDIX B—TCP/IP STACK DATA STRUCTURES

The following data structures are defined by the TCP/IP stack:

Struct: SOCKET_INFO_T

```
typedef struct socket_info_s {
   word16 src_port;
   word16 dest_port;
   byte ip_dest_addr[IP_ADDR_LEN];
   byte *send_ptr;
   word16 send_len;
   byte *recv_ptr;
   byte *recv_end;
   word16 recv_len;
   byte ip_proto;
   byte socket_no;
   byte socket_type;
   byte socket_state;
#if TCP
   byte tcp_state;
   byte tcp_resends;
   byte tcp_flag;
   byte recv_tcp_flag;
   byte data_offset;
   word16 tcp_unacked_bytes;
   word16 recv_tcp_window;
   SEQNUM_U RCV_NXT;
   SEQNUM_U SEG_SEQ;
   SEQNUM_U SEG_ACK;
   SEQNUM_U SND_UNA;
   TIMER_INFO_T tcp_timer;
#endif
} SOCKET_INFO_T;
```

Struct: VF

```
typedef struct vf {
   byte filename[VF_NAME_LEN];
   word16 page_size;
   PCONST_BYTE page_ptr;
   byte * ram_page_ptr;
   byte page_type;
   byte in_use_flag;
#if CP220x
unsigned int CP2200_PAGE_PTR;
#endif
} VF;
```

Struct: POST_FUNCS

```
typedef struct post_funcs {
   byte func_name[FUNC_NAME_LEN];
   POST_FP func_ptr;
} POST_FUNCS;
```

Struct: SMTP_INFO_T

```
typedef struct smtp_info_s {
   byte *from;
   byte *to;
   byte *subject;
   byte *message;
   byte *attachment;
   byte *filename;
} SMTP_INFO_T;
```

Struct: DHCP_INFO_T

```
typedef struct dhcp_info_s {
   byte op;                      /* opcode, request or reply */
   byte htype;                   /* hardware type */
   byte hlen;                    /* hardware address length */
   byte hops;                    /* always zero for clients */

   byte xid[4];                  /* random transaction ID */

   byte secs[2];                 /* seconds elapsed since trying to boot */
   byte flag[2];                 /* broadcast flag */

   byte ciaddr[IP_ADDR_LEN];     /* client IP address submitted */
   byte yiaddr[IP_ADDR_LEN];     /* client IP address returned by server */
   byte siaddr[IP_ADDR_LEN];     /* server IP address returned by server */
   byte giaddr[IP_ADDR_LEN];     /* optional gateway IP address */

   byte chaddr[DHCP_MAC_LEN];    /* client hardware address */

   byte sname[DHCP_SNAME_LEN];   /* optional server host name */
   byte file[DHCP_FILE_LEN];     /* boot file name */

   byte options[DHCP_OPT_LEN];   /* options */
} DHCP_INFO_T;
```

Struct: DHCP_LEASE_T

```
typedef struct dhcp_lease_s {
   word32 org_lease_time;                /* last requested lease time */
   volatile word32 lease_time;           /* seconds left in current lease */
   word32 t1_renew_time;                 /* time to make renew request */
   word32 t2_renew_time;                 /* time to make rebind request */
   volatile byte dhcp_state;             /* current dhcp state */
   byte infinite_lease;                  /* infinite lease TRUE or FALSE */
   byte server_id[DHCP_SERVER_ID_LEN];   /* DHCP server IP address */
} DHCP_LEASE_T;
```

Struct: BOOTP_INFO_T

```
typedef struct bootp_s {
   byte op;                      /* opcode, request or reply */
   byte htype;                   /* hardware type */
   byte hlen;                    /* hardware address length */
   byte hops;                    /* always zero for clients */

   byte xid[4];                  /* random transaction ID */

   byte secs[2];                 /* seconds elapsed since trying to boot */
   byte flag[2];                 /* broadcast flag */

   byte ciaddr[IP_ADDR_LEN];     /* client IP address submitted */
   byte yiaddr[IP_ADDR_LEN];     /* client IP address returned by server */
   byte siaddr[IP_ADDR_LEN];     /* server IP address returned by server */
   byte giaddr[IP_ADDR_LEN];     /* optional gateway IP address */

   byte chaddr[BOOTP_MAC_LEN];   /* client hardware address */

   byte sname[BOOTP_SNAME_LEN];  /* optional server host name */
   byte file[BOOTP_FILE_LEN];    /* boot file name */

   byte vend[BOOTP_VENDOR_LEN];  /* optional vendor-specific area */
} BOOTP_INFO_T;
```

APPENDIX C—FIRMWARE LIBRARY MEMORY-MODEL COMPILER SETTINGS

The firmware API library was created using the LARGE memory model. Using this library in a project with a default memory model of SMALL or COMPACT can cause warnings to occur, depending on warning level settings. To avoid this, set the default memory model to LARGE, and override this setting by defining each function with the “small” compiler keyword.

APPENDIX D—CONNECTING THE EMBEDDED SYSTEM TO A PC

For Systems that use Ethernet as the Physical Layer

The TCP/IP stack allows the embedded system to connect to an Ethernet network using a static or dynamically-allocated IP address. The embedded system can also be directly connected to a PC (without being connected to a network) using a crossover cable. When using a crossover cable, both nodes need static IP addresses in order to communicate. Refer to the Embedded Ethernet Development Kit User's Guide for step-by-step instructions on how to configure the PC and the embedded system for Ethernet communication.

For Systems that use a Modem as the Physical Layer

The TCP/IP stack allows the embedded modem to be configured as a client or server. The embedded modem can communicate with any other modem through a standard telephone line (POTS) or telephone simulator. Any PC that has a modem and is running Windows 2000 or Windows XP can be configured to accept calls or dial into the embedded modem. Refer to Appendix B of the Embedded Modem Development Kit User's Guide for step-by-step instructions on how to configure a modem on a PC.

APPENDIX E—ERROR CODES DEFINED IN MN_ERRS.H

```
#define NOT_ENOUGH_SOCKETS       -128     // 0xFF80
#define SOCKET_ALREADY_EXISTS    -127     // 0xFF81
#define NOT_SUPPORTED            -126     // 0xFF82
#define PPP_OPEN_FAILED          -125     // 0xFF83
#define TCP_OPEN_FAILED          -124     // 0xFF84
#define BAD_SOCKET_DATA          -123     // 0xFF85
#define SOCKET_NOT_FOUND         -122     // 0xFF86
#define SOCKET_TIMED_OUT         -121     // 0xFF87
#define BAD_IP_HEADER            -120     // 0xFF88
#define NEED_TO_LISTEN           -119     // 0xFF89
#define RECV_TIMED_OUT           -118     // 0xFF8A
#define ETHER_INIT_ERROR         -117     // 0xFF8B
#define ETHER_SEND_ERROR         -116     // 0xFF8C
#define ETHER_RECV_ERROR         -115     // 0xFF8D
#define NEED_TO_SEND             -114     // 0xFF8E
#define UNABLE_TO_SEND           -113     // 0xFF8F
#define VFILE_ENTRY_IN_USE       -112     // 0xFF90
#define TFTP_FILE_NOT_FOUND      -111     // 0xFF91
#define TFTP_NO_FILE_SPECIFIED   -110     // 0xFF92
#define TFTP_FILE_TOO_BIG        -109     // 0xFF93
#define TFTP_FAILED              -108     // 0xFF94
#define SMTP_ALREADY_OPEN        -107     // 0xFF95
#define SMTP_OPEN_FAILED         -106     // 0xFF96
#define SMTP_NOT_OPEN            -105     // 0xFF97
#define SMTP_BAD_PARAM_ERR       -104     // 0xFF98
#define SMTP_ERROR               -103     // 0xFF99
#define NEED_TO_EXIT             -102     // 0xFF9A
#define FTP_FILE_MAXOUT          -101     // 0xFF9B
#define DHCP_ERROR               -100     // 0xFF9C
#define DHCP_LEASE_EXPIRED       -99      // 0xFF9D
#define PPP_LINK_DOWN            -98      // 0xFF9E
#define GET_FUNC_ERROR           -97      // 0xFF9F
#define FTP_SERVER_DOWN          -96      // 0xFFA0
#define ARP_REQUEST_FAILED       -95      // 0xFFA1
#define NEED_IGNORE_PACKET       -94      // 0xFFA2
#define TASK_DID_NOT_START       -93      // 0xFFA3
#define DHCP_LEASE_RENEWING      -92      // 0xFFA4
#define IGMP_ERROR               -91      // 0xFFA5
#define MN_INIT_ERROR            -90      // 0xFFA6
#define MN_VERIFY_ERROR          -89      // 0xFFA7
#define INVALID_DUPLEX_MODE      -88      // 0xFFA8
#define INVALID_MAC_ADDRESS      -87      // 0xFFA9
```

```
#define AUTO_NEG_FAIL            -86      // 0xFFAA
#define LINK_FAIL                -85      // 0xFFAB
#define DNS_ID_ERROR             -75      // 0xFFB5
#define DNS_OPCODE_ERROR         -74      // 0xFFB6
#define DNS_RCODE_ERROR          -73      // 0xFFB7
#define DNS_COUNT_ERROR          -72      // 0xFFB8
#define DNS_TYPE_ERROR           -71      // 0xFFB9
#define DNS_CLASS_ERROR          -70      // 0xFFBA
#define DNS_NOT_FOUND            -69      // 0xFFBB
#define DNS_BUFFER_OVERFLOW      -68      // 0xFFBC

// TCP error codes
#define TCP_ERROR       -1
#define TCP_TOO_LONG    -2
#define TCP_BAD_HEADER  -3
#define TCP_BAD_CSUM    -4
#define TCP_BAD_FCS     -5
#define TCP_NO_CONNECT  -6

// UDP error codes
#define UDP_ERROR       -1
#define UDP_BAD_CSUM    -4
#define UDP_BAD_FCS     -5
```

DOCUMENT CHANGE LIST

Revision 0.5 to Revision 0.6

- Added support for Netfinder.
- Added support for DNS.
- Added support for storing the transmit buffer in USB FIFO space.
- Added support for serving web pages from the CP220x Flash.
- Added "Appendix E—Error Codes Defined in mn_errs.h" on page 52.

NOTES:

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Email: MCUinfo@silabs.com
Internet: www.silabs.com