



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

1. Introduction

The Silicon Laboratories TCP/IP stack is designed to add network connectivity to the C8051F12x and C8051F13x 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 and virtual file system.
- Up to 127 simultaneous TCP or UDP sockets. Direct access to sockets allows custom application development.
- Support for PPP and SLIP with customizable modem settings. Interfaces to an Si2457 modem through the serial port.

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 Modem Development Kit (Modem-DK), which includes:

- C8051F12x Target Board, Debug Adapter, and power supply.
- Si2457FT18-EVB Modem Board with an 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 K object code generated from application code. TCP/IP library code does not count towards 4 K limit.
- 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 program interface (API) on the C8051F12x and C8051F13x microcontrollers. These functions provide the microcontroller a dial-up network interface via an Si2457 modem connected to the serial port. 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 frequently-used API functions are listed below:

<code>mn_init()</code>	Initializes all sockets and stack variables
<code>mn_modem_connect()</code>	Establishes a connection between the MCU and the modem
<code>mn_ppp_add_pap_user()</code>	Adds a user name and password to use during authentication
<code>mn_ppp_open()</code>	Establishes a PPP connection
<code>mn_server()</code>	Starts an HTTP or FTP Server
<code>mn_smtp_send_mail()</code>	Sends an e-mail to an SMTP mail server

3. Getting Started

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

- Modifying the HTTP Web Server Example.
- Modifying the SMTP Mail Client Example.
- Modifying the TCP Socket Example.
- Modifying the UDP Socket Example.
- 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:

<code>mn_userconst.h</code>	Header file containing user settings.
<code>TCPIP_Project.wsp</code>	Project file that can be opened from the Silicon Labs IDE.
<code>main.c</code>	Contains the main routine and callback functions.
<code>mn_stack000.lib</code>	TCP/IP stack library. Note the three-digit library number.

Group 2:

<code>mn_defs.h</code>	Contains type definitions used by the TCP/IP stack
<code>mn_errs.h</code>	Contains error code definitions
<code>mn_funcs.h</code>	Contains function prototype information
<code>mn_stackconst.h</code>	Contains constants required by the TCP/IP stack
<code>mn_vars.c</code>	Contains global variables used by the TCP/IP stack

Group 3:

<code>VFILE_DIR</code>	Optional subdirectory containing HTML files, images, and other content
<code>VFILE_DIR\html2c.exe</code>	Optional html2c.exe utility that converts content to file arrays
<code>VFILE_DIR\update.bat</code>	Optional batch file to automate conversion to file arrays
<code>VFILE_DIR\mn_defs.h</code>	Optional header file required only if using file arrays
<code>VFILE_DIR\index.html</code>	Optional main HTML page for web server
<code>VFILE_DIR\index.h</code>	Optional main HTML page converted to file array using <i>html2c.exe</i>
<code>VFILE_DIR\index.c</code>	Optional main HTML page converted to file array using <i>html2c.exe</i>

3.2. TCP/IP Configuration Wizard Output

The TCP/IP Configuration Wizard generates a custom library with a unique three-digit library number that describes the selected protocol configuration. 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 code examples provided with this guide are a good starting point for new projects. If the protocols used in the example meet the needs of the end application, such as the HTTP Web Server, the example can easily be modified to include additional application code and the HTML files changed to suite 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 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, please 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	Functions related to opening, closing, and managing sockets
Modem Functions	Functions to manage the connection between the MCU and the modem
PPP Functions	Functions to open, close, and manage a PPP connection
Application Functions	Functions to start and use application layer services (HTTP Web Server, FTP Server, TFTP Client, and SMTP Mail Client)
Callback Functions	Event handlers called by the stack to notify the application layer
VFILE Functions	Functions to manage the virtual file system
Support Functions	Functions to perform conversion between 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 30 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 (<i>const unsigned char*</i>)

Socket Data Types:

PSOCKET_INFO	Pointer to a <i>SOCKET_INFO_T</i> structure.
--------------	--

Virtual File System Data Types:

VF_PTR	Pointer to a <i>VF</i> structure
POST_FP	Function pointer to a CGI content creation function
PF_PTR	Pointer to a <i>POST_FUNCS</i> structure

SMTP Mail Client Data Types:

PSMTP_INFO	Pointer to an <i>SMTP_INFO_T</i> structure
------------	--

4.3. Important Notes

- The `mn_init()` function must be called prior to calling any other stack function.
- 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 timebase 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 timebase of system clock divided by 48. This 16-bit value should be set such 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 timebase 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.

- 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 the total program code size is greater than 64 kB, the project must be set up for code banking, and the TCP/IP library must be placed in the common area. 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. If the desired library does not fit in the common area, contact MCU Applications for a solution.

4.4. Socket Functions

The TCP/IP stack provides direct access to TCP and UDP sockets through the functions listed below.

<code>mn_init()</code>	Section 4.4.1 on page 6
<code>mn_open()</code>	Section 4.4.2 on page 7
<code>mn_send()</code>	Section 4.4.3 on page 8
<code>mn_recv()</code>	Section 4.4.4 on page 9
<code>mn_recv_wait()</code>	Section 4.4.5 on page 9
<code>mn_close()</code>	Section 4.4.6 on page 10
<code>mn_abort()</code>	Section 4.4.7 on page 10
<code>mn_find_socket()</code>	Section 4.4.8 on page 10

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);`

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.

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:

- *dest_ip*—Destination IP address to which packets are being sent.
- *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.
- *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.
- *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.
- *proto*—Defines the socket type. Can be one of the following values:
 - *PROTO_TCP*—A TCP socket is allocated.
 - *PROTO_UDP*—A UDP socket is allocated.
- *type*—Should be set to *STD_TYPE*.
- *recv_buff*—Address of the buffer used to store the received data.
- *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-allocated 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.

Note: 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.

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

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

Parameters:

- *socket_no*—The socket number returned from *mn_open()*.
- *msg_ptr*—Address of the buffer containing data to send.
- *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_TRY*s 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.

4.4.4. mn_rcv

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.

Note: This function will loop until a packet is received on the passed socket or a timeout occurs. The `callback_app_rcv_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_rcv_idle()` returns `NEED_TO_EXIT`.

Note: 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.

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

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

Parameters:

- `socket_no`—The socket number returned from `mn_open()`.
- `buff_ptr`—Address of the buffer to hold received data.
- `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_rcv_idle()` returned `NEED_TO_EXIT`.
- Any other negative number—there was a checksum or FCS error or the TCP connection is closed.

4.4.5. mn_rcv_wait

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

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

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

Parameters:

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

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

4.4.6. mn_close

Description: Closes a previously opened socket.

Prototype: `int mn_close(SCHAR);`

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

Parameters: ■ *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: ■ *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: ■ *src_port*—The local port number.
■ *dest_port*—The remote port number.
■ *dest_ip*—The IP address of the remote machine.
■ *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. 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 login 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:

<code>mn_modem_connect()</code>	Section 4.5.1 on page 11
<code>mn_modem_disconnect()</code>	Section 4.5.2 on page 12
<code>mn_modem_send_string()</code>	Section 4.5.3 on page 12
<code>mn_modem_wait_reply()</code>	Section 4.5.4 on page 12

4.5.1. 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:

- `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.

4.5.2. 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();`

4.5.3. 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:

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

4.5.4. 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:

- *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.
- *len*—The number of bytes in *str*, including the carriage return ('\r').
- *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*.

4.6. PPP Functions

The TCP/IP stack allows a choice between PPP and SLIP for the data link layer. If SLIP is chosen, all data link management is automatically performed by the stack. If PPP is selected, application code should use the following functions to establish a PPP connection with a remote client/server prior to opening any TCP sockets.

<code>mn_ppp_open()</code>	Section 4.6.1 on page 13
<code>mn_ppp_close()</code>	Section 4.6.2 on page 13
<code>mn_ppp_reset()</code>	Section 4.6.3 on page 14
<code>mn_ppp_add_pap_user()</code>	Section 4.6.4 on page 14
<code>mn_ppp_del_pap_user()</code>	Section 4.6.5 on page 14

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 4.5.1 on page 11](#) for more information about authentication at the physical layer.

4.6.1. mn_ppp_open

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

Note: All modem connections and stack initialization must be complete prior to calling this function.

Note: 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`.

Prototype: `int mn_ppp_open(byte);`

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

Parameters:

- `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`.

4.6.2. 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();`

4.6.3. 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();`

4.6.4. 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:

- *username*—Null terminated character string containing the user name.
- *password*—Null terminated character string containing the password.

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

4.6.5. 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:

- *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.

4.7. Application Layer Functions

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

<code>mn_server()</code>	Section 4.7.1 on page 15
<code>mn_http_find_value()</code>	Section 4.7.2 on page 16
<code>mn_tftp_get_file()</code>	Section 4.7.3 on page 16
<code>mn_smtp_start_session()</code>	Section 4.7.4 on page 16
<code>mn_smtp_end_session()</code>	Section 4.7.5 on page 17
<code>mn_smtp_send_mail()</code>	Section 4.7.6 on page 17

Note: These functions, in conjunction with callback and virtual file system functions described in the next two sections, provide complete control over application layer services.

4.7.1. 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 PPP error occurs or a callback function (`callback_app_server_idle()` or `callback_app_server_process_packet()`) returns `NEED_TO_EXIT`.

Note: 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.

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, then 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 compile time.
- 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.

4.7.2. 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.

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

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

Parameters:

- *source_ptr*—Address to buffer containing the message body which will be searched.
- *field_name*—Null terminated search string containing the field-name.
- *field_value*—String buffer where the field-value will be copied.

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

4.7.3. 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:

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

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

4.7.4. mn_smtp_start_session

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

Note: Modem and PPP connections must be established prior to calling this function.

Prototype: `SCHAR mn_smtp_start_session(word16);`

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

Parameters:

- *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.

4.7.5. mn_smtp_end_session

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

Prototype: `void mn_smtp_end_session(SCHAR);`

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

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

4.7.6. 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, PSMTMP_INFO);`

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

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

4.8. Callback Functions

The TCP/IP stack uses callback functions to notify application code of various events. Figure 1 shows the callback function code execution flow. The following four callback functions should be defined in every project which uses application layer services provided by the TCP/IP stack. The callback functions should contain the appropriate event handling code.

<code>callback_app_process_packet()</code>	Section 4.8.1 on page 19
<code>callback_app_server_process_packet()</code>	Section 4.8.2 on page 19
<code>callback_app_rcv_idle()</code>	Section 4.8.3 on page 20
<code>callback_app_server_idle()</code>	Section 4.8.4 on page 20

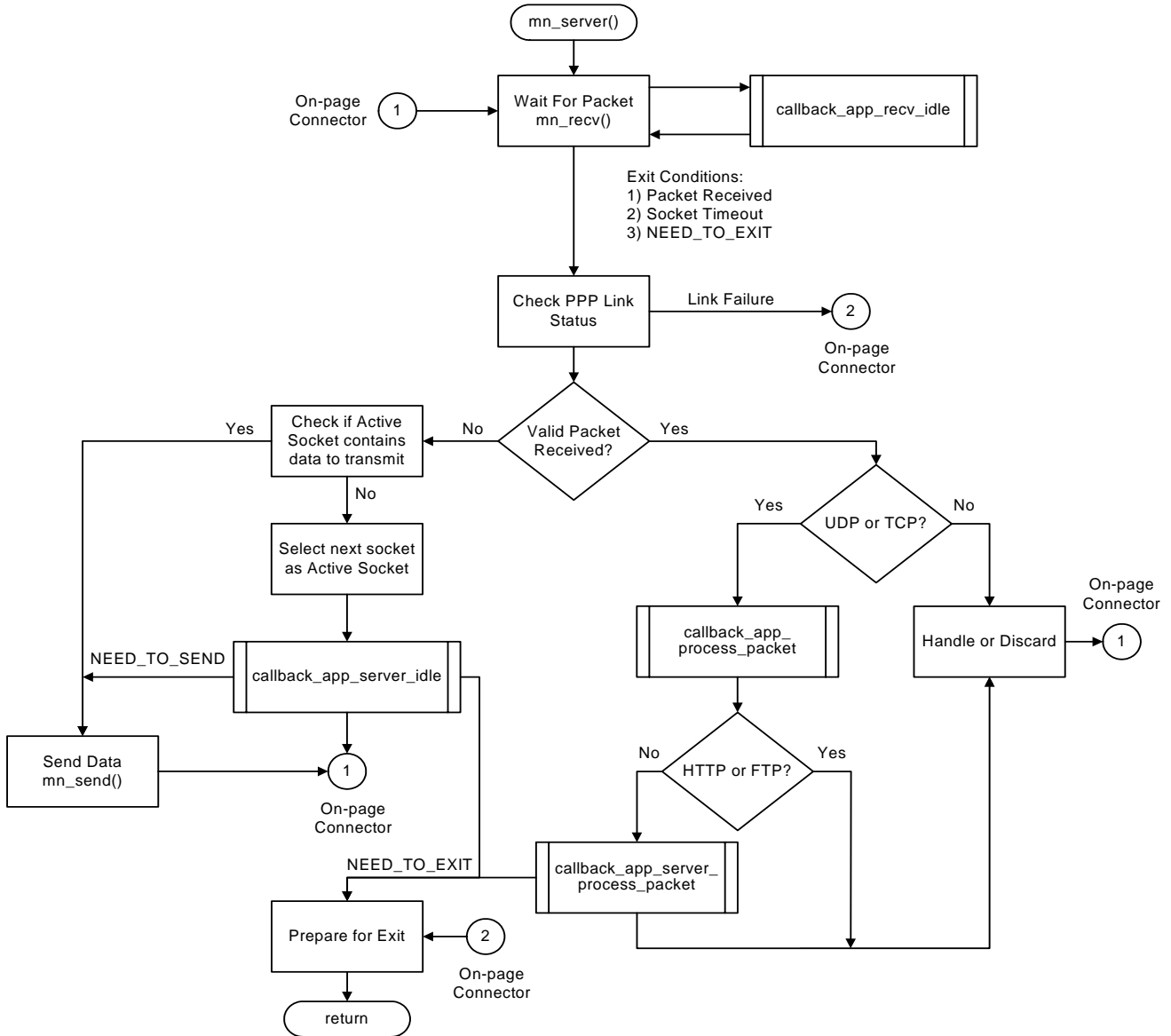


Figure 1. Callback Function Flow Diagram

4.8.1. 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(P SOCKET_INFO);`

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

Parameters: ■ *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.

4.8.2. 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_server_process_packet(P SOCKET_INFO);`

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

Parameters: ■ *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.

4.8.3. callback_app_rcv_idle

Description: Called repeatedly while *mn_rcv()* 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_rcv_idle(void);`

Example Call: `status = callback_app_rcv_idle();`

Return Value: The following are valid return values:

- *NEED_TO_EXIT*—The *mn_rcv()* 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_rcv()* routine will continue to wait for data.

4.8.4. 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(P SOCKET_INFO*);`

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

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

Note: The socket handle may be re-assigned 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.

4.9. 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. Add the C source file (e.g., *image.c*) to the project build.
2. Include the header file (e.g., *image.h*) at the beginning of *main.c* using the `#include` directive.
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.

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

HTTP/FTP Server File System Functions:

<code>mn_vf_get_entry()</code>	Section 4.9.1 on page 22
<code>mn_vf_set_entry()</code>	Section 4.9.2 on page 22
<code>mn_vf_set_ram_entry()</code>	Section 4.9.3 on page 23
<code>mn_vf_del_entry()</code>	Section 4.9.4 on page 23

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 a semicolon. 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:

<code>mn_pf_get_entry()</code>	Section 4.9.5 on page 24
<code>mn_pf_set_entry()</code>	Section 4.9.6 on page 24
<code>mn_pf_del_entry()</code>	Section 4.9.7 on page 24

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.

4.9.1. 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 27 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: ■ *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.

4.9.2. mn_vf_set_entry

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

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

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: ■ *filename*—Null terminated string containing the file name (e.g., “index.html”).
■ *file_size*—Number of bytes in the file.
■ *file_ptr*—Pointer to the start of the file.
■ *mem_seg*—Type of memory where the file is stored. Should be set to *VF_PTYPE_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.

4.9.3. 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:

- *filename*—Null terminated string containing the file name (e.g., *index.html*).
- *file_size*—Number of bytes in the file.
- *file_ptr*—Pointer to the start of the file.
- *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.

4.9.4. 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:

- *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.

4.9.5. 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: ■ *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.

4.9.6. 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: ■ *function_name*—Null terminated string containing the name of the desired function.
■ *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 27](#) for a definition of the PF structure.

4.9.7. 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: ■ *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.

4.10. Support Functions

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

<code>mn_ustoa()</code>	Section 4.10.1 on page 25
<code>mn_uctoa()</code>	Section 4.10.2 on page 25
<code>mn_getMyIPAddr_func()</code>	Section 4.10.3 on page 26
<code>mn_atous()</code>	Section 4.10.4 on page 26

4.10.1. 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:

- `dest_buff`—Address to a character array to store the null-terminated string result.
- `source`—The unsigned integer that will be converted to a string.

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

4.10.2. 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:

- `dest_buff`—Address to a character array to store the null-terminated string result.
- `source`—The unsigned char that will be converted to a string.

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

4.10.3. 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: ■ *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*.

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

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

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. The tables below list all possible constants, however, the header file generated will only contain constants that are applicable to the generated library.

Table 2. IP 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

Table 3. 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.
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 4. TCP/IP Stack Adjustments

Constant Name	Description
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.
xmit_buff_size	Sets the size of the buffer used for transmission.
recv_buff_size	Sets the size of the buffer used for reception.
socket_wait_ticks	Number of 10 ms system ticks to wait for a packet.
ip_time_to_live	Sets the “time to live” field in the IP packet.
multicast_ttl	Sets the “time to live” field in an IP packet for multicast packets.
tl0_flash th0_flash	Timer 0 reload values such that Timer 0 overflows in 10 ms. This defines a system tick.
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.
pap_num_users	Number of entries in the PAP table.
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.
uart_reload	Reload value for the UART. The maximum standard UART baud rate is automatically selected by the TCP/IP Configuration Wizard.
arp_keep_ticks	Number of system ticks to keep entries in the ARP cache.
arp_resend_trys	Number of times an ARP packet is re-transmitted.
arp_wait_ticks	Number of system ticks to wait for an ARP packet.
arp_cache_size	Number of entries in the ARP cache.
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.
ftp_max_param	Size of the buffer to hold received command line parameters. This value must be at least 23.
ftp_buffer_len	FTP Receive Buffer Size. Must be large enough to hold the largest expected file size.
ftp_num_users	Number of username/password combinations to store. If set to zero, authentication will not be performed.
mem_pool_size	RAM memory pool available to the <i>malloc()</i> function.
http_buffer_len	Buffer used to process HTTP includes. Should be the same size as TCP window.

Table 4. TCP/IP Stack Adjustments (Continued)

Constant Name	Description
tftp_resend_trys	Number of times a TFTP packet is transmitted before terminating the connection.
ping_buff_size	If PING is enabled the value is the size of the data from a PING request that can be stored. 9 bytes are added to the value to store part of the PING request header also. 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.
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 will yield better throughput but require 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 <i>RECV_BUFFER_SIZE</i> and <i>XMIT_BUFFER_SIZE</i> should be at least double the TCP window to allow for escaped characters. If using ethernet the <i>RECV_BUFFER_SIZE</i> and <i>XMIT_BUFFER_SIZE</i> should be at least <i>TCP_WINDOW</i> + 58.
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.
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 <i>TCP_WINDOW</i> .
num_vf_pages	The number of entries in the directory table in the virtual file system. Can be 1 to 255.
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.

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;
#ifdef 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;
} 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;
```



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 MODEM TO A PC

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 running Windows 2000 or Windows XP that has a modem can be configured to accept calls or dial into the embedded modem.

Configuring the PC to Accept Calls (Server Mode)

1. Go to the “Network Connections” dialog in the Control Panel.
2. Click on “Create New Connection”. The New Connection Wizard should appear.
3. If using Windows XP, select “Setup an Advanced Connection” and click “Next”.
4. Select “Accept Incoming Connections” and click “Next”.
5. Place a checkmark next to the modem name.
6. Select “Do not allow virtual private connections”.
7. Place a checkmark next to all user’s who will be allowed to use the modem.
8. Select Internet Protocol (TCP/IP) and click “Properties”. From this dialog, you can specify the IP address configuration and provide or restrict access to the local area network (LAN). Providing the embedded modem access to the LAN allows the embedded system to send e-mail using the SMTP mail server on a corporate network.
9. Click “Finish” to complete the connection.

Configuring the PC to Dial the Embedded Modem (Client Mode)

1. Go to the “Network Connections” dialog in the Control Panel.
2. Click on “Create new connection”. The New Connection Wizard should appear.
3. Select “Connect to the Internet” and click “Next”.
4. Select “Setup my connection manually” and click “Next”.
5. Select “Connect using a dial-up modem ” and click “Next”.
6. Specify a name for the connection and click “Next”.
7. Specify the phone number of the embedded modem.
8. Select the user’s allowed to dial the connection.
9. Specify the user name and password used to log into the embedded modem.
10. Click “Finish” to complete the connection.

CONTACT INFORMATION

Silicon Laboratories Inc.
4635 Boston Lane
Austin, TX 78735
Tel: 1+(512) 416-8500
Fax: 1+(512) 416-9669
Toll Free: 1+(877) 444-3032
Email: MCUinfo@silabs.com
Internet: www.silabs.com

The information in this document is believed to be accurate in all respects at the time of publication but is subject to change without notice. Silicon Laboratories assumes no responsibility for errors and omissions, and disclaims responsibility for any consequences resulting from the use of information included herein. Additionally, Silicon Laboratories assumes no responsibility for the functioning of undescribed features or parameters. Silicon Laboratories reserves the right to make changes without further notice. Silicon Laboratories makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Silicon Laboratories assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. Silicon Laboratories products are not designed, intended, or authorized for use in applications intended to support or sustain life, or for any other application in which the failure of the Silicon Laboratories product could create a situation where personal injury or death may occur. Should Buyer purchase or use Silicon Laboratories products for any such unintended or unauthorized application, Buyer shall indemnify and hold Silicon Laboratories harmless against all claims and damages.

Silicon Laboratories and Silicon Labs are trademarks of Silicon Laboratories Inc.

Other products or brandnames mentioned herein are trademarks or registered trademarks of their respective holders.