Chapter 3a: Using TCP/IP Sockets

At the end of this chapter, you will understand how to use the ModusToolbox™ for Connectivity run-time software to send and receive data using a TCP/IP socket.

Note: *If you are taking this course on your own, you will need two PSoC™ kits to do the exercises in this chapter – one to act as a client and one to act as server. If you are taking this course in a classroom setting, you will only need one kit.*

Table of contents

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR); make build
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

3a.1　　Sockets – Fundamentals of TCP communication

For applications, e.g. a web browser, to communicate via the TCP transport layer they need to open a **Socket**. A Socket, or more properly a TCP Socket, is simply a reliable, ordered pipe between two devices on the internet. To open a socket you need to specify the IP Address and Port Number (just an unsigned 16-bit integer) on the server that you are trying to talk to. On the server there is a program running that listens on that Port for bytes to come through. Sockets are uniquely identified by two tuples (source IP:source port) and (destination IP:destination port) e.g. 192.168.15.8:3287 + 184.27.235.114:80.

There are a bunch of standard ports (which you might recognize) for Applications including:

- HTTP　　80
- HTTPS　443
- SMTP　　25
- DNS　　　53
- POP　　　110
- MQTT　　1883

These are typically referred to as "Well Known Ports" and are managed by the IETF Internet Assigned Numbers Authority (IANA); IANA ensures that no two applications designed for the Internet use the same port (whether for UDP or TCP).

The server will typically listen on a well-known port (so that clients know how to connect to it) while on the client end the socket will use some other (usually random) port number. This is one reason why there can be multiple open connections to a webserver running on port 80 even from a single device – as long as both connections have a different IP address or port number on the client side, each socket is uniquely identifiable.

ModusToolbox™ for Connectivity easily supports TCP sockets (`cy_socket_create`) via the Secure Sockets Library. You can create your own protocol to talk between your IoT device and a server or you can implement a protocol as defined by someone else. Note that "raw" sockets inherently don't have security. The TCP socket just sends whatever data it was given over the link. It is the responsibility of a layer above TCP such as SSL or TLS to encrypt/decrypt the data if security is being used (which we will cover later). Despite its name, the Secure Sockets Library supports both secure and non-secure sockets.

In general, developers mostly use one of the standard Application Protocols (HTTP, MQTT etc.) which are discussed in later chapters, but for now as an example of a custom protocol we will define an example protocol called AWEP as an ASCII text-based protocol. The client and the server both send a string of characters that are of the form:

- Command: 1 character representing the command (R=Read, W=Write, A=Accepted, X=Failed).
- Device ID: 4 characters representing the hex value of the device e.g. 1FAE or 002F. Each device will have its own unique register set on the server so you should use a unique ID (unless you want to read/write the same register set as another device).
- Register: 2 characters representing the register (each device has 256 registers) e.g. 0F or 1B.
- Value: 4 characters representing the hex value of a 16-bit unsigned integer. The value should be left out on "R" commands.
- Termination: the value 0x00 (the ASCII character for NULL which is automatically placed at the end of strings in C).

The client can send "R" and "W" commands. The server responds with "A" (and the data echoed) or "X" (with nothing else). The server contains a database that will store values that are written to it (when a client uses

the "W" command) and will send back requested values (when a client uses the "R" command). The server keeps track of a separate 256 register set for each device ID. For example, the register with address 0x0F for a device with ID 0x1234 is not the same as register with address 0x0F for a device with ID 0xABCD.

Since this protocol is one we just invented, the IANA doesn't have ports for it. We will choose two ports that are not already reserved: the open version of the protocol will run on port 50007 and the secure TLS version will run on port 50008. We will mainly be using the open version of the protocol in this class. Some AWEP example messages:

- "W0FAC0B1234" would write a value of 0x1234 to register 0x0B for device with an ID of 0x0FAC. The server would then respond with "A0FAC0B1234".
- "W01234" is an illegal packet and the server would respond with "X".
- "R0FAC0B" is a read of register 0x0B for a Device ID with an ID of 0x0FAC". In this case the server would respond with "A0FAC0B1234" (the value of 1234 was written in the first case).
- "R0BAC0B" is a legal read, but there has been no data written to that device so the server would respond with "X".

Note: Remember that $strlen$ doesn't include the null termination so when you send a string, be sure to use $strlen(<string>) + 1$ to determine the message length in order for the null termination to be included in the message.

3a.2   TCP server & client using sockets

In the examples below, the AWEP protocol, as defined in the previous section, is used to demonstrate the steps to create a connection between an AWEP client (192.51.100.14) and an AWEP server (192.51.100.3) using sockets.

Create Connection
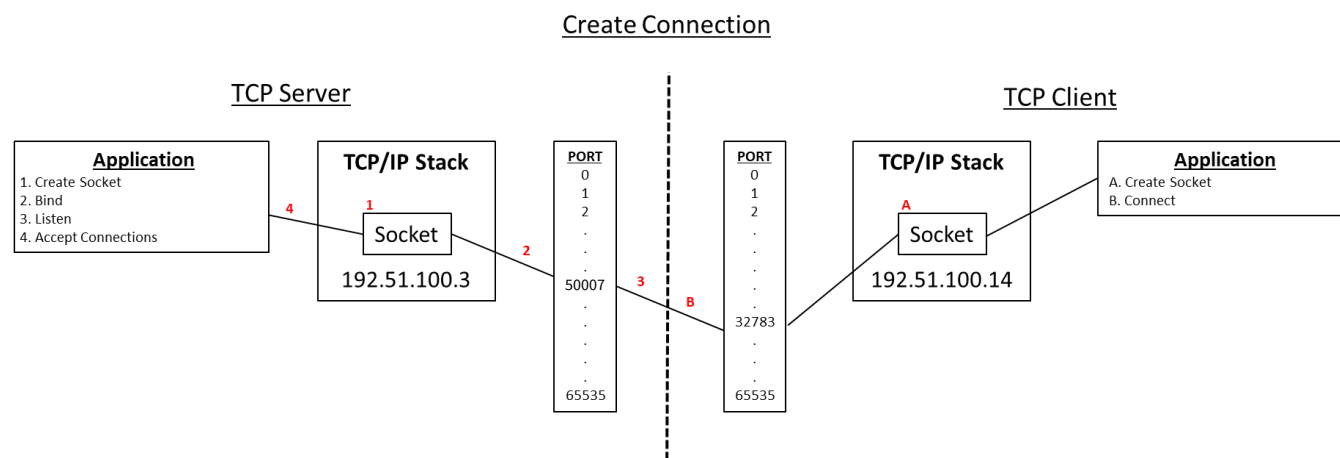
The picture above describes the steps required to make a TCP connection between two devices: a TCP server (on the left of the dotted line) and a TCP client (on the right). These two devices are already connected to an IP network and have been assigned IP addresses (192.51.100.3 and 14). There are 4 parts of each system:

- Your firmware applications (the boxes labeled Application). This is the firmware that you write to control the system using the run-time software. There is a separate application for the server and client.
- The TCP/IP stack which handles all the communication with the network.
- The Port, which represents the 65536 TCP ports (numbered 0-65535).

- The Packet Buffer, which represents the RAM where the Transmit "T" and Receive "R" packets are held. They are one of the things configured in the file lwipopts.h

Note that the server needs its socket connected to a specific port (50007 in this case) so that the client knows where to connect, but the client can use any port when it connects to the server. In fact, an available port will be chosen automatically when you connect from the client – you don't need to specify the client port.

To setup the TCP <u>server</u> connection, the server firmware will:

1. Create the TCP socket by calling (the socket is a structure of type `cy_socket_t`):

```
cy_socket_init();
cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,
CY_SOCKET_IPPROTO_TCP, &server_handle);
```

2. Bind and Listen to the socket on AWEP server TCP port 50007 by calling (the address is a structure of type `cy_socket_socketaddr_t`):

```
cy_socket_bind(server_handle, &tcp_server_addr, sizeof(tcp_server_addr));
cy_socket_listen(server_handle, TCP_SERVER_MAX_PENDING_CONNECTIONS);
```

3. Sleep the current thread and wait for a connection by calling:

```
cy_socket_accept(server_handle, &peer_addr, &peer_addr_len, &client_handle);
```

To setup the TCP <u>client</u> connection, the client firmware will:

A. Create the TCP socket by calling:

```
cy_socket_init();
cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,
CY_SOCKET_IPPROTO_TCP, &client_handle);
```

B. To create the actual connection to the server you need to do two things:

Specify the server's IP address and port. These are passed to the connect function as a data structure of type `cy_socket_sockaddr_t`. Let's assume you have defined a structure of that type called `serverAddress`.

You can initialize the structure in one of two ways depending on how you want to specify the server's IP address – either statically or using DNS.

- To initialize it statically you can use the helper function provided by the connectivity utilities library:

```
cy_nw_ip_address_t server_address;
cy_nw_str_to_ipv4("192.168.1.101", &server_address);
tcp_server_address.ip_address.ip.v4 = server_address.ip.v4;
serverAddress.ip_address.version = CY_SOCKET_IP_VER_V4;
serverAddress.port = 50007;
```

Note: *You must include the header file cy_nw_helper.h to get access to the cy_nw_str_to_ipv4 function.*

- To initialize it by performing a mDNS query, do the following:

```
cy_socket_gethostbyname("awep.local", CY_SOCKET_IP_VER_V4 ,
&serverAddress.ip_address);
```

```
                serverAddress.port = 50007;
```

Make the connection through the network by calling `cy_socket_connect` as follows:

```
cy_socket_connect(client_handle, &serverAddress,
sizeof(cy_socket_sockaddr_t));
```

3a.3 Socket options

Infineon's Secure Sockets library allows you to set socket options such as: callbacks for socket events, socket authentication mode, message receive timeout period, etc. To edit a socket's options, you must first create a socket. Then you must call `cy_socket_setsockopt`, which takes the following arguments:

- The handle of the socket who's options you wish to modify - type `cy_socket_t`
- The level at which the option resides – There are three options for this:
    - `CY_SOCKET_SOL_SOCKET` – Socket options
    - `CY_SOCKET_SOL_TCP` – TCP options
    - `CY_SOCKET_SOL_TLS` – TLS options
- The name of the option to be set – For example:
    - `CY_SOCKET_SO_RCVTIMEO` – Socket receive timeout
    - `CY_SOCKET_SO_RECEIVE_CALLBACK` – Socket message received callback function
    - `CY_SOCKET_SO_TLS_AUTH_MODE` – TLS authentication mode
- A buffer containing the value to set the option to
- The length of the buffer containing the value

3a.4 Transmitting and receiving data using sockets

Once the connection has been created, your application will want to transfer data between the client and server. The simplest way to transfer data over TCP is to use the send and receive functions from the run-time software. These functions allow you to send and receive TCP streams.

It is simple to write data using the `cy_socket_send` function. This function takes the `socket_handle`, a pointer to the buffer containing the data to be sent, the length of the data to send, flags, and a pointer to an `uint32_t` to record the number of bytes sent. The following code demonstrates writing a single message:

```
uint32_t bytes_sent;
char sendMessage[] = "TEST_MESSAGE";
cy_socket_send(socket_handle, sendMessage, strlen(sendMessage)+1,
CY_SOCKET_FLAGS_NONE, &bytes_sent);
```

Given the above, the firmware to transmit data from a client to a server might look something like this:

```
#define SERVER_PORT      (50007)
#define TIMEOUT          (2000)
…
cy_socket_init();
cy_socket_sockaddr_t serverAddress;
cy_socket_t client_handle;
char sendMessage[]="WABCD051234";
uint32_t bytes_sent;
…
cy_socket_gethostbyname("awep.local", CY_SOCKET_IP_VER_V4 ,
&serverAddress.ip_address);
serverAddress.port = 50007;
```

```
…
// Loop here for each message to be sent
cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,
CY_SOCKET_IPPROTO_TCP, &client_handle);
cy_socket_connect(client_handle, &serverAddress, sizeof(cy_socket_sockaddr_t));
cy_socket_send(client_handle, sendMessage, strlen(sendMessage)+1,
CY_SOCKET_FLAGS_NONE, &bytes_sent);
cy_socket_delete(client_handle);
// End of loop
```

Reading data uses the `cy_socket_recv` function. This function takes a `socket_handle`, a pointer to a buffer to write received data into, the size of the data buffer, flags, and a pointer to a `uint32_t` to record the number of bytes received. The function returns a `cy_rslt_t` value which can be used to ensure that reading the stream succeeded.

```
result = cy_socket_recv(socket_handle, rbuffer,  MAX_TCP_RECV_BUFFER_SIZE,
CY_SOCKET_FLAGS_NONE, &bytesReceived);
```

This function should be called from within a "message received" callback that you specify using the `cy_socket_setsockopt` function after creating the socket. This callback is a function you define and that the Secure Sockets library will automatically run whenever a message is received by its associated socket(s). The following is an example of how to do this:

Note: You should also set a timeout value for the `cy_socket_recv` function as shown below.

```
// Define the message received callback
static cy_rslt_t tcp_receive_msg_handler(cy_socket_t socket_handle, void *arg){
    char message_buffer[20u]; // Buffer to store the received message
    uint32_t bytes_received = 0; // Var to store number of bytes received
    // Receive the message
    cy_socket_recv(socket_handle, message_buffer, MAX_TCP_RECV_BUFFER_SIZE,
    CY_SOCKET_FLAGS_NONE, &bytes_received);
    // Process the received message
    …
    return CY_RSLT_SUCCESS;
}

cy_socket_init();
cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,
CY_SOCKET_IPPROTO_TCP, &socket_handle);

// Set the timeout for the recv function
uint32_t tcp_recv_timeout = 2000;
cy_socket_setsockopt(socket_handle, CY_SOCKET_SOL_SOCKET, CY_SOCKET_SO_RCVTIMEO,
&tcp_recv_timeout, sizeof(tcp_recv_timeout));

// Specify the message received callback
cy_socket_opt_callback_t tcp_receive_option;
tcp_receive_option.callback = tcp_receive_msg_handler;
tcp_receive_option.arg = NULL;
cy_socket_setsockopt(socket_handle, CY_SOCKET_SOL_SOCKET,
CY_SOCKET_SO_RECEIVE_CALLBACK, &tcp_receive_option,
sizeof(cy_socket_opt_callback_t));

// Bind/connect the socket
…
```
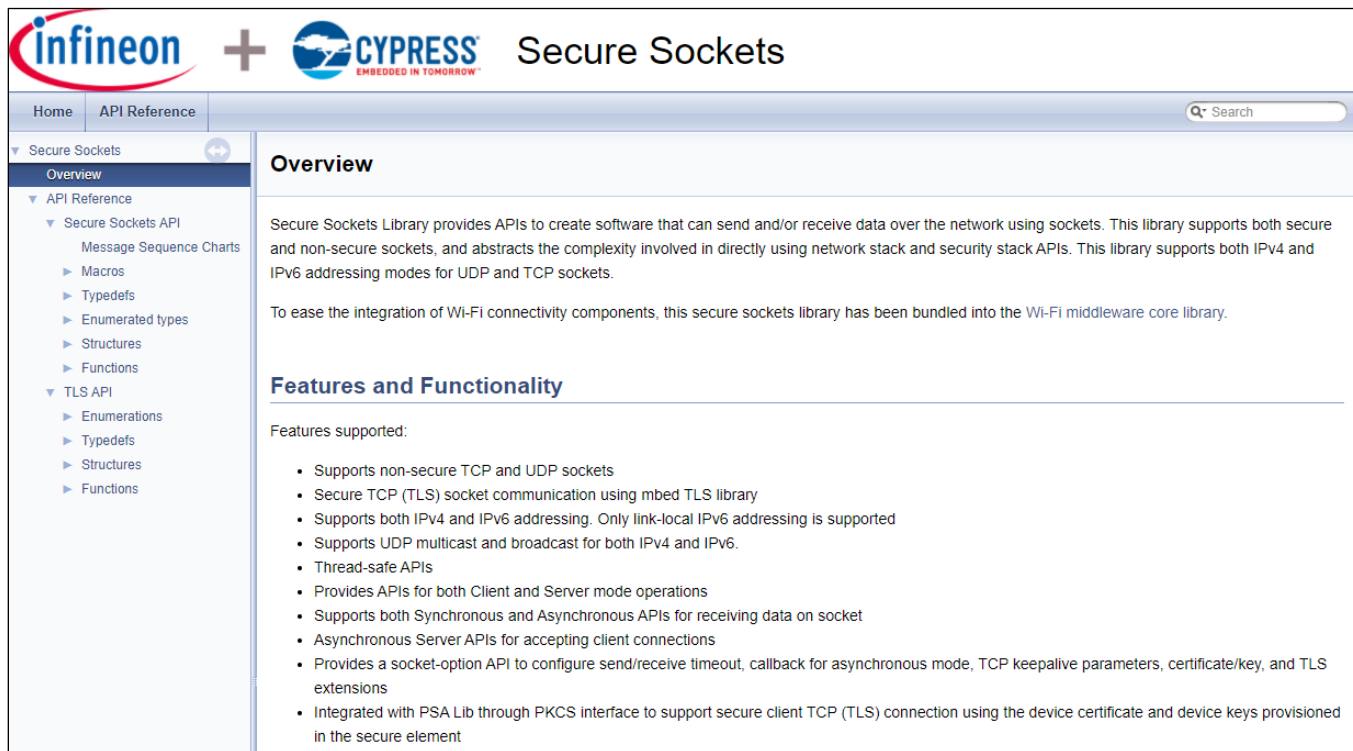
3a.5   Documentation

The documentation for Infineon's Secure Sockets library resides on the [Secure Sockets documentation page](#).

3a.6 Exercises

Exercise 1: Implement AWEP

Create an IoT client to write data to a server running AWEP when a button is pressed on the client.

Your application will monitor button presses on the board and will toggle an LED in response to each button press. In addition, each time the button is pressed, your application will connect to the AWEP server, send the state of the LED, and disconnect.

Note: *The connect/disconnect is done for each message so that the clients don't all keep a socket open to the server simultaneously. This is a fairly common IoT practice to reduce loading on the sever.*

For the application:

- The LED characteristic number is 05. That is, the LED state is stored in address 0x05 in the 256-byte register space.
- The "value" of the LED is 0000 for OFF and 0001 for ON.
- For the device ID, use the 16-bit checksum of your device's MAC address.

Server Setup

If you are taking this course in a classroom setting, you can skip these steps, as the AWEP server has already been set up for you. However, if you are taking this class on your own, you will need to set the server up yourself. To do this, you will need a second PSoC™ 6 kit.

1. Create a new application named **ch03_AWEP_server** using the **Browse** button to select the completed server application (***modustoolbox-level3-wifi/projects/ch03b_ex04_dual_server***) as a template.

2. Open the file *tcp_server.h* and edit the macros `WIFI_SSID` and `WIFI_PASSWORD` so that the server can connect to your local network.

3. Program the project to your second kit. Verify that the server connects to your network and begins listening for incoming connections.

   Note: *This server application runs both a non-secure (port 50007) and secure (port 50008) version of the AWEP protocol.*

Client Setup

It is recommended to implement this project in smaller steps rather than trying to tackle the entire thing at once. For example, start by writing a project to:

1. Connect to Wi-Fi.

   Note: *Use one of your projects from the previous chapter as a starting point.*

   Note: *There are multiple config files for the various run-time software libraries. It is best practice to move all of them to the root level of your project. These config files can be found in mtb_shared/wifi-mw-core/latest-vX.X/configs*

Note: If you are using the CY8CPROTO-062-4343W or CY8CPROTO-062S2-43439 kit, you must add `CY_WIFI_HOST_WAKE_SW_FORCE=0` to the `DEFINES` in the Makefile in order to disable the Wi-Fi host wake feature. This is necessary because the user button and host wake pin from the Wi-Fi device are connected to the same PSoC pin.

The following code can be pasted into the Makefile below the existing `DEFINES`.

```
# The CY8CPROTO-062-4343W and CY8CPROTO-062S2-43439 kits share
# the same GPIO for the user button (USER_BTN1) and the Wi-Fi host
# wake up pin (used by the PSoC to wake up the Wi-Fi device. Since this
# example uses the GPIO for interfacing with the user button, the
# SDIO interrupt to wake up the host is disabled by setting
# CY_WIFI_HOST_WAKE_SW_FORCE to '0'.
#
# If you wish to enable this host wake up feature, Comment out
# DEFINES+=CY_WIFI_HOST_WAKE_SW_FORCE=0. If you want this feature on
# one of the kits that share pins, change the GPIO pin for USER_BTN
# in design/hardware & comment out DEFINES+=CY_WIFI_HOST_WAKE_SW_FORCE=0.
DEFINES+=CY_WIFI_HOST_WAKE_SW_FORCE=0
```

☐ 2. Use mDNS to get the IP address of the server (awep.local).

Note: Recall from the prior chapter that ".local" is appended to the server name when using mDNS for name resolution.

☐ 3. Send a hard-coded message to the server ONCE.

Note: Send the message in the initialization section of application start rather than the `while(1)` loop so that you don't spam the server continuously. Each time you reset the kit the message will be sent one time.

Note: Use a hard-coded message like W<device_number>050000 where <device_number> is any 4-digit value such as your birth month and day.

1. Open a socket to the AWEP server (create, connect).

2. Initialize a stream

3. Send your message to the server

4. Delete the socket

☐ 4. Program and test the project to make sure your message is received by the server.

Note: You can find your device's IP address in a UART terminal window. Compare that to the IP address displayed by the server when it receives a message to verify your message was received.

☐ 5. Next, let's change the device ID to the MAC address checksum. That will more likely be a unique value than your birthday.

Note: See the exercise on printing network information from the Introductory Wi-Fi chapter for an example on getting the MAC address of your device.

    a.   To get the checksum, just take the six individual octets (bytes) of the MAC address and add them together. Store the result in a `uint16_t` variable.

    b.   Use `snprintf` to create the message that you want to send.

       For example, if your message is a character array of 12 bytes (11-byte message plus terminating \0) called "sendMessage" and your MAC address checksum is stored in a `uint16_t` called "macCheck", you could use the following:

```
snprintf(sendMessage, sizeof(sendMessage), "W%04X050000", macCheck);
```

6.   Program and test the project again to verify that it still works.

7.   Now, let's add the LED functionality to the message.

8.   Initialize the LED to OFF.

9.   Setup a GPIO to monitor a button.

Note: *An interrupt is a good choice here. You should use an RTOS construct like a "semaphore give" in the interrupt callback and then use a "semaphore take" inside a task rather than doing all the work directly inside the interrupt callback.*

10. When the button is pressed:

    a.   Change the LED state.

    b.   Update the message.

Note: *Again, use `snprintf` to format the message. In this case you will be providing 2 parameters to be substituted in the message string – the MAC address checksum and the alternating LED value of 0000 or 0001.*

    c.   Send the message.

Note: *This should be done inside the `while(1)` loop of a task so that the message is sent each time the button is pressed instead of just once during initialization.*

11. Program and test the project again to verify that it still works.

Exercise 2: Update AWEP client to check the return code

Remember that in the AWEP protocol the server returns a packet with either "A" or an "X" as the first character. For this exercise, read the response back from the server and make sure that your original write occurred properly. Test with a legal and an illegal packet.

Exercise 3: Implement the AWEP server

Implement the server side of the non-secure AWEP protocol that can handle one connection at a time. Use the client from the previous exercise on a second kit to test your server.

3a.7   Further reading

- [1] RFC1700 – "Assigned Numbers"; Internet Engineering Task Force (IETF) - https://www.ietf.org/rfc/rfc1700.txt
- [3] IANA Service Name and Port Registry - http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.