

Chapter 6A: Using TCP/IP Sockets

Time 1 ½ Hours

At the end of this chapter you will understand how to use the AnyCloud SDK to send and receive data using TCP/IP socket

6A.1	SOCKETS – FUNDAMENTALS OF TCP COMMUNICATION.....	2
6A.2	ANYCLOUD-SDK TCP SERVER & CLIENT USING SOCKETS.....	3
6A.3	TRANSMITTING AND RECEIVING DATA USING SOCKETS	5
6A.4	SOCKET OPTIONS.....	6
6A.5	SOCKET DOCUMENTATION	6
6A.6	EXERCISE(S)	7
6A.6.1	EXERCISE 1: IMPLEMENT AWEP	7
6A.6.2	EXERCISE 2: UPDATE AWEP CLIENT TO CHECK THE RETURN CODE.....	9
6A.6.3	EXERCISE 3: (ADVANCED) IMPLEMENT AWEP SERVER.....	9
6A.7	FURTHER READING	9

6A.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. This is one reason why there can be multiple open connections to a webserver running on port 80. The local (or ephemeral port) is allocated by the TCP stack and new ports are allocated on the initiator (client) for each connection to the receiver (server).

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).

AnyCloud easily supports TCP sockets (`cy_socket_create`) via the Secure Sockets Library and 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 the AnyCloud Wi-Fi Example Protocol (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.

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

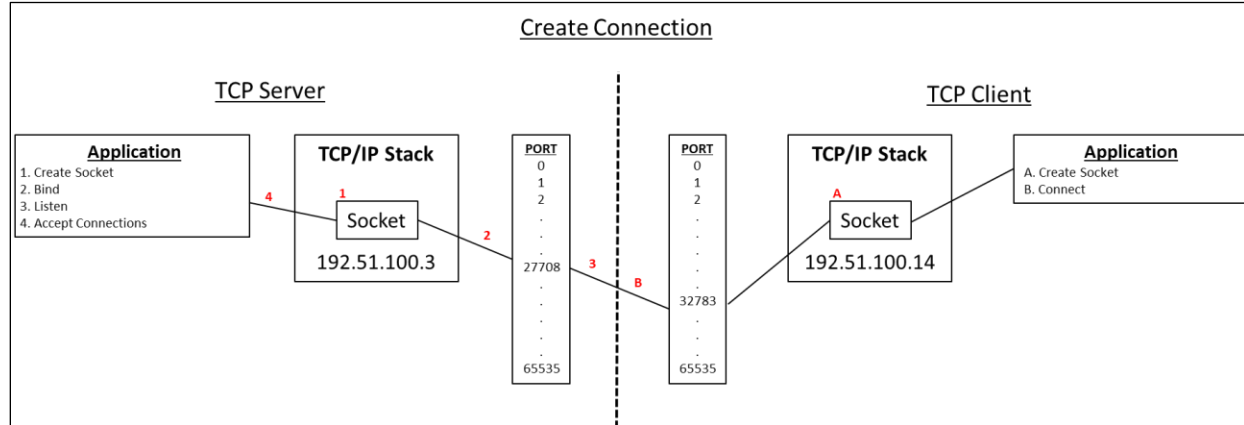
client uses the "W" command) and will send back requested values (when a client uses the "R" command). The server keeps track of a separate 256 register set for each device ID. For example, the register with address 0x0F for a device with ID 0x1234 is not the same as register with address 0x0F for a 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 27708 and the secure TLS version will run on port 40508. We will be using mainly the open version of the protocol in this class. Some AWEF 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".

6A.2 AnyCloud-SDK TCP Server & Client using Sockets

In the examples below, the AWEF protocol, as defined in the previous section, is used to demonstrate the steps to create a connection between an AWEF Client (198.51.100.14) and an AWEF Server (198.51.100.3) using sockets.



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 AnyCloud-SDK. There is firmware for both 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 (27708 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 is chosen automatically when you connect from the client – you don't need to specify the client port.

To setup the TCP server 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 27708 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 client 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 macros provided by the AnyCloud SDK as follows:

```
serverAddress.ip_address.ip.v4 = MAKE_IPV4_ADDRESS(192, 51, 100, 3);  
serverAddress.ip_address.version = CY_SOCKET_IP_VER_V4;  
serverAddress.port = 27708;
```

- 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 = 27708;
```

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

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

6A.3 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 AnyCloud SDK. 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:

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

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);
```

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

```
#define SERVER_PORT (27708)
#define TIMEOUT (2000)
...
cy_socket_init();
cy_socket_sockaddr_t serverAddress;
cy_socket_t socket_handle;
char sendMessage[]="WABCD051234";
...
cy_socket_gethostbyname("awep.local", CY_SOCKET_IP_VER_V4 ,
    &serverAddress.ip_address);
serverAddress.port = 27708;
...
// Loop here for each message to be sent
cy_socket_create(CY_SOCKET_DOMAIN_AF_INET, CY_SOCKET_TYPE_STREAM,
    CY_SOCKET_IPPROTO_TCP, &socket_handle);
cy_socket_connect(socket_handle, &serverAddress, sizeof(cy_socket_sockaddr_t));
cy_socket_send(socket_handle, sendMessage, strlen(sendMessage),
    CY_SOCKET_FLAGS_NONE, &bytes_sent);
cy_socket_delete(socket_handle);
// End of loop
```

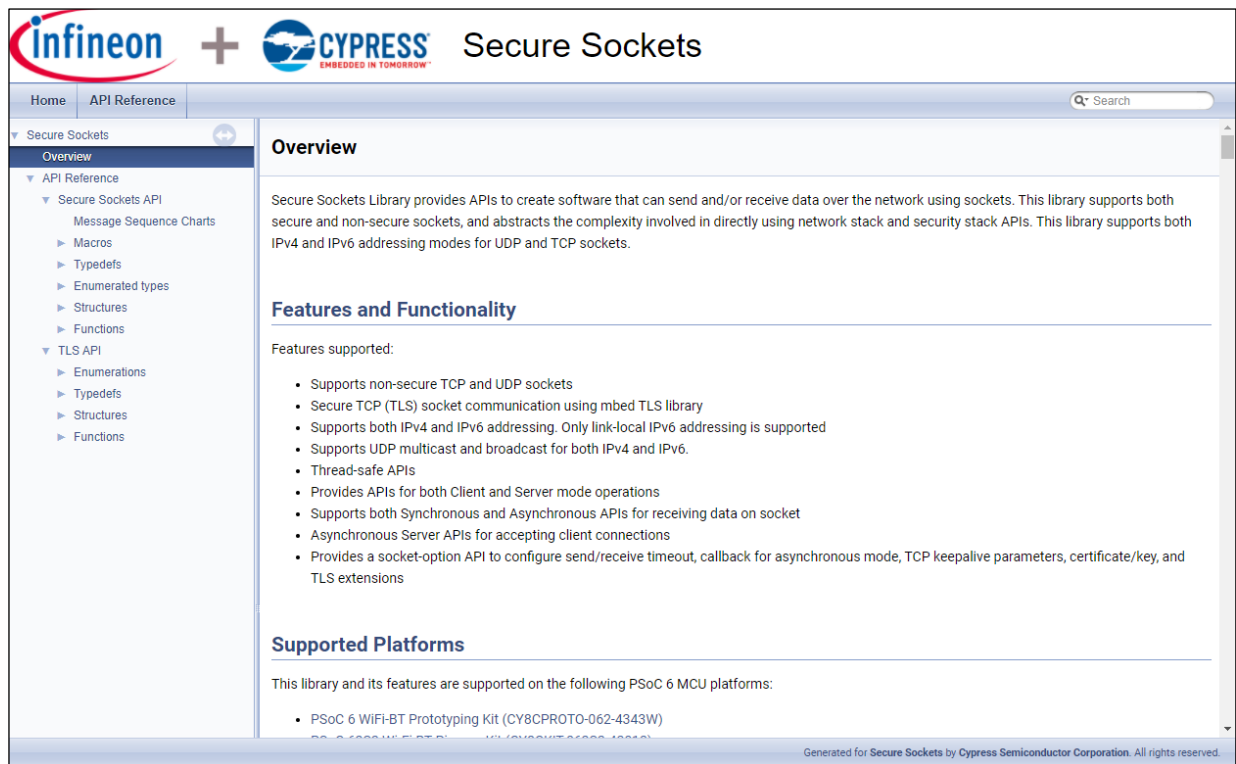
6A.4 Socket Options

The 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_setopt`, which takes the following arguments

- The handle of the socket whose 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

6A.5 Socket Documentation

The AnyCloud-SDK provides you a library of functions to do socket-based communication. The documentation on sockets resides on the Secure Sockets documentation page.



The screenshot shows the 'Secure Sockets' documentation page. The header includes the Infineon and Cypress logos, followed by the title 'Secure Sockets'. Below the header is a navigation bar with 'Home' and 'API Reference' tabs, and a search bar. The left sidebar shows a tree view under 'Secure Sockets' with 'Overview' selected. The main content area has three sections: 'Overview' (describing the library's purpose), 'Features and Functionality' (listing supported features like non-secure sockets, TLS, IPv4/IPv6, etc.), and 'Supported Platforms' (listing PSoc 6 MCU platforms).

6A.6 Exercise(s)

6A.6.1 Exercise 1: Implement AWEPP

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

We have implemented a server using the AnyCloud-SDK running the non-secure version of the AWEPP protocol as described above with the following:

- Hostname: awep
- Port: 27708

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 AWEPP server, send the state of the LED, and disconnect. The connect/disconnect is done for each message so that the clients don't all keep a socket open to the server simultaneously. For the application:

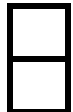
- The LED characteristic number is 05. That is, the LED state is stored in address 0x05 in the 256-byte register space.
- The "value" of the LED is 0000 for OFF and 0001 for ON.
- For the device ID, use the 16-bit checksum of your device's MAC address.

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



1. Connect to Wi-Fi.

- Hint** Use one of your projects from the previous chapter as a starting point.
- Hint** There are multiple config files for the various AnyCloud 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*



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

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

- Hint** 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.
- Hint** 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.
- Open a socket to the AWEPP server (create, bind, connect).
- Initialize a stream
- Send your message to the server
- Delete the socket



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

Hint 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.

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

Hint 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:

- Change the LED state.
- Update the message.

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.

- Send the message.
- Hint** 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.

6A.6.2 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.

Hint This can be done by calling `cy_socket_recv`.

6A.6.3 Exercise 3: (Advanced) Implement AWEP Server

Implement the server side of the non-secure AWEP protocol that can handle one connection at a time.

6A.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>