

Chapter 3a: Using TCP/IP Sockets

At the end of this chapter you will understand how to use the ModusToolbox™ any cloud SDK to send and receive data using a TCP/IP socket.

Note: If you are taking this course on your own, you will need two CY8CKIT-062S2-43012 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

3a.1	Sockets – Fundamentals of TCP communication	2
3a.2	ModusToolbox™ any cloud TCP server & client using sockets	3
3a.3	Transmitting and receiving data using sockets	5
3a.4	Socket options	6
3a.5	Documentation	6
3a.6	Exercises	7
	Exercise 1: Implement AWEPP	7
	Exercise 2: Update AWEPP client to check the return code	9
	Exercise 3: Implement the AWEPP server.....	9
3a.7	Further reading.....	9

Document conventions

Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR) ; make build
<i>Italics</i>	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™ any cloud 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 the ModusToolbox™ any cloud 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 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

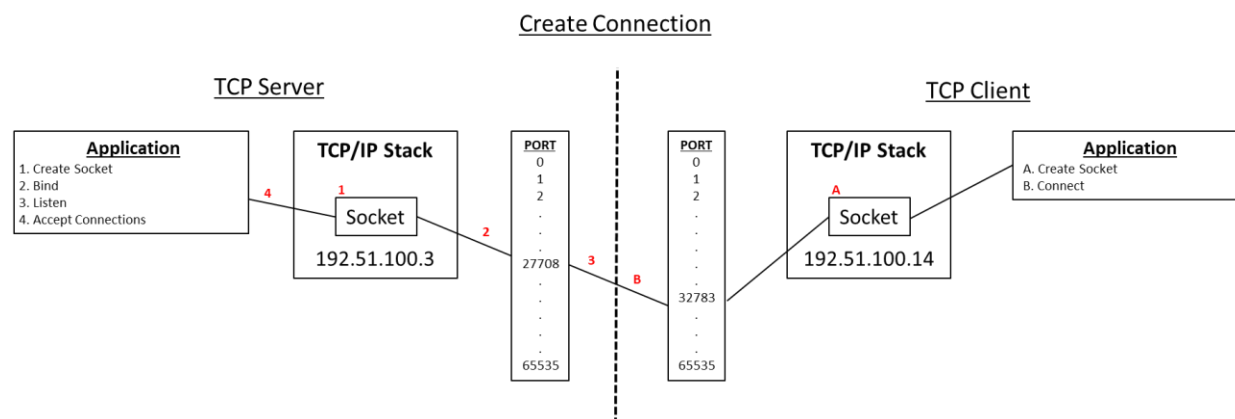
keeps track of a separate 256 register set for each device ID. For example, the register with address 0x0F for a device with ID 0x1234 is not the same as register with address 0x0F for a device with ID 0xABCD.

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 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 AWEPP 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".

3a.2 ModusToolbox™ any cloud TCP server & client using sockets

In the examples below, the AWEPP protocol, as defined in the previous section, is used to demonstrate the steps to create a connection between an AWEPP client (192.51.100.14) and an AWEPP server (192.51.100.3) 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 ModusToolbox™ any cloud SDK. 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 (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 will be 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 AWEF server TCP port 27708 by calling (the address is a structure of type `cy_socket_sockaddr_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 ModusToolbox™ any cloud 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("awef.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));
```

3a.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 ModusToolbox™ any cloud 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
```

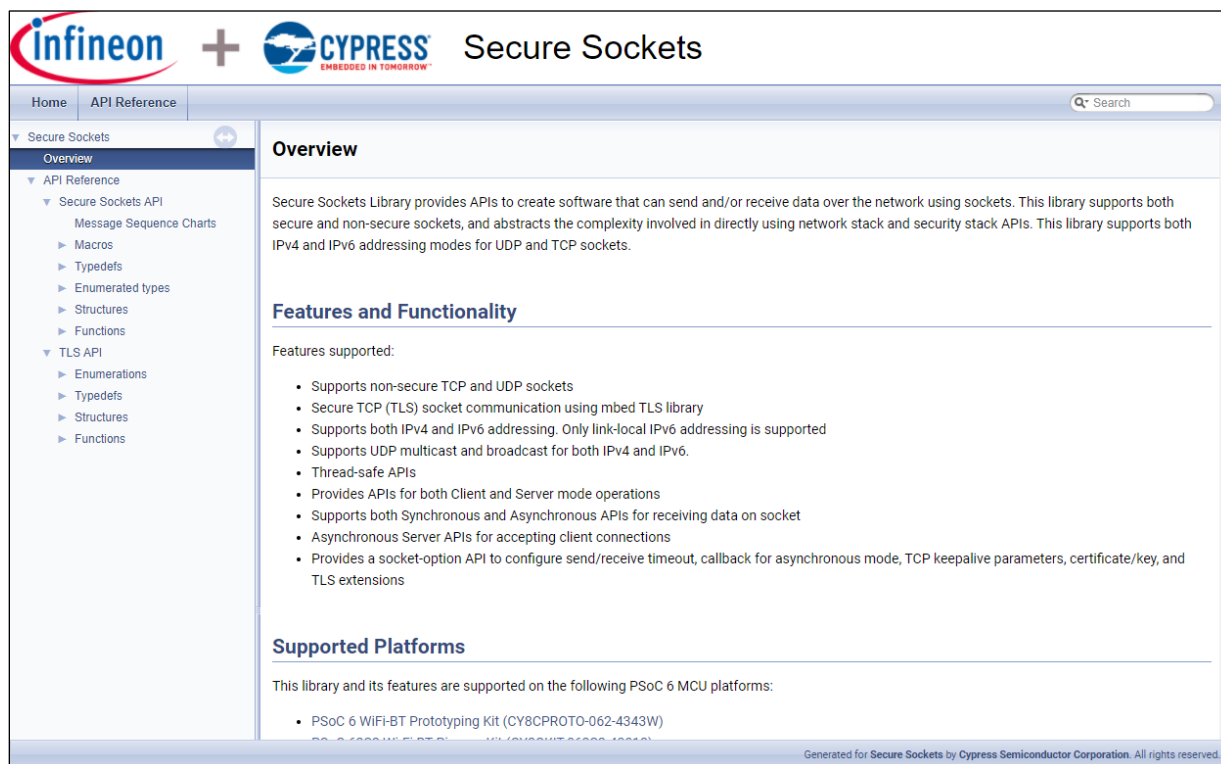
3a.4 Socket options

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

3a.5 Documentation

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



Secure Sockets

Home | API Reference | Search

Secure Sockets

Overview

Secure Sockets Library provides APIs to create software that can send and/or receive data over the network using sockets. This library supports both secure and non-secure sockets, and abstracts the complexity involved in directly using network stack and security stack APIs. This library supports both IPv4 and IPv6 addressing modes for UDP and TCP sockets.

Features and Functionality

Features supported:

- Supports non-secure TCP and UDP sockets
- Secure TCP (TLS) socket communication using mbed TLS library
- Supports both IPv4 and IPv6 addressing. Only link-local IPv6 addressing is supported
- Supports UDP multicast and broadcast for both IPv4 and IPv6.
- Thread-safe APIs
- Provides APIs for both Client and Server mode operations
- Supports both Synchronous and Asynchronous APIs for receiving data on socket
- Asynchronous Server APIs for accepting client connections
- Provides a socket-option API to configure send/receive timeout, callback for asynchronous mode, TCP keepalive parameters, certificate/key, and TLS extensions

Supported Platforms

This library and its features are supported on the following PSoC 6 MCU platforms:

- PSoC 6 Wi-Fi-BT Prototyping Kit (CY8CPROTO-062-4343W)

Generated for Secure Sockets by Cypress Semiconductor Corporation. All rights reserved.

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 CY8CKIT-062S2-43012 kit.

- ☐ 1. Create a new application named **ch03_AWEP_server** using the **Import** button to select the completed server application (**modustoolbox-level2-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 27708) and secure (port 40508) version of the AWEP protocol.

Client Setup

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.

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

Note: There are multiple config files for the various ModusToolbox™ any cloud 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.

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, bind, 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 AWEPP client to check the return dode

Remember that in the AWEPP 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 AWEPP server

Implement the server side of the non-secure AWEPP 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.

Published by
Infineon Technologies AG
81726 Munich, Germany

© 2021 Infineon Technologies AG.
All Rights Reserved.

IMPORTANT NOTICE

The information given in this document shall in no event be regarded as a guarantee of conditions or characteristics ("Beschaffheitsgarantie").

With respect to any examples, hints or any typical values stated herein and/or any information regarding the application of the product, Infineon Technologies hereby disclaims any and all warranties and liabilities of any kind, including without limitation warranties of non-infringement of intellectual property rights of any third party.

In addition, any information given in this document is subject to customer's compliance with its obligations stated in this document and any applicable legal requirements, norms and standards concerning customer's products and any use of the product of Infineon Technologies in customer's applications.

The data contained in this document is exclusively intended for technically trained staff. It is the responsibility of customer's technical departments to evaluate the suitability of the product for the intended application and the completeness of the product information given in this document with respect to such application.

For further information on the product, technology, delivery terms and conditions and prices please contact your nearest Infineon Technologies office (www.infineon.com).

WARNINGS

Due to technical requirements products may contain dangerous substances. For information on the types in question please contact your nearest Infineon Technologies office.

Except as otherwise explicitly approved by Infineon Technologies in a written document signed by authorized representatives of Infineon Technologies, Infineon Technologies' products may not be used in any applications where a failure of the product or any consequences of the use thereof can reasonably be expected to result in personal injury.