Chapter 6A: Establishing Communication using TCP/IP Sockets

Time 1 ½ Hours

At the end of Chapter 6A you will understand how to use the WICED-SDK to send and receive data using TCP/IP sockets.

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).

WICED easily supports TCP sockets (*wiced_tcp_create_socket()*) 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.

To build a custom protocol, for instance, we can define the WICED Wi-Fi Example Protocol (WWEP) 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.

The open version of the protocol runs on port 27708 and the secure TLS version runs on port 40508. We will be using mainly the open version of the protocol in this class.

Some examples:

- "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 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).

Sockets are available in the WICED SDK and enable you to build your own custom protocol. However, in general developers are mostly using one of the standard Application Protocols (HTTP, MQTT etc.) which are discussed in Chapter 7.

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

In the examples below I use the WWEP protocol as defined in the previous section to demonstrate the steps to create a connection between a WWEP Client (198.51.100.14) and a WWEP Server (198.51.100.3) using sockets.
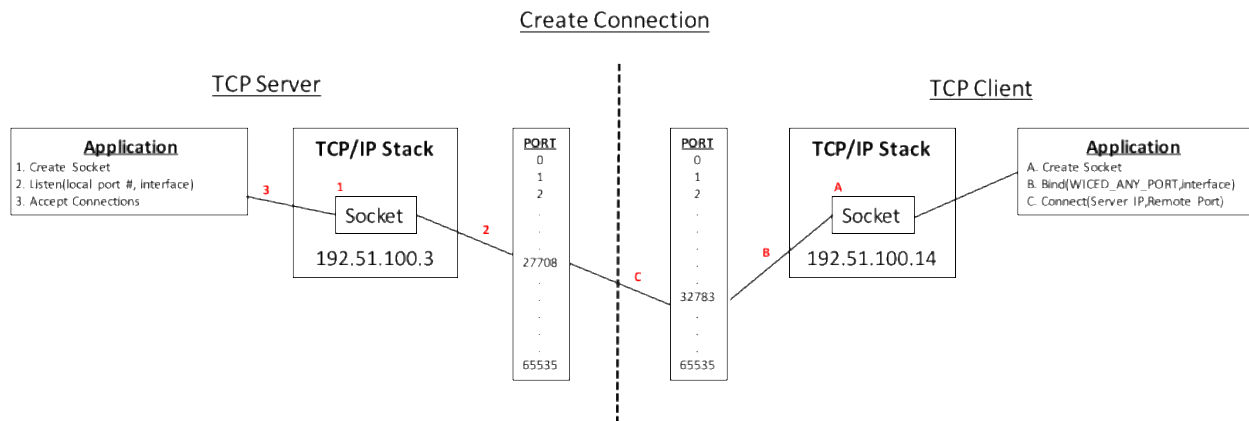
The picture above describes the steps required to make a TCP connection between two devices, a TCP Server (on the left of the dotted line) and a TCP Client (on the right). These two devices are already connected to an IP network and have been assigned IP addresses (192.51.100.3 and 14). There are 4 parts of each system:

- Your firmware applications (the boxes labeled Application). This is the firmware that you write to control the system using the WICED-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 4x ~1500 bytes of RAM where the Transmit "T" and Receive "R" packets are held.

To setup the TCP server connection, the server firmware will:

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

   ```
   wiced_tcp_create_socket( &socket, WICED_STA_INTERFACE );
   ```

2. Listen to the socket on WWEP server TCP port 27708 by calling:

   ```
   wiced_tcp_listen( &socket, 27708 );    // 27708 is the port number WWEP
   ```

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

   ```
   wiced_tcp_accept( &socket );
   ```

To setup the TCP client connection, the client firmware will:

A. Create the TCP socket by calling:

   ```
   wiced_tcp_create_socket( &socket, WICED_STA_INTERFACE );
   ```

B. "Bind" to some TCP port (it doesn't matter which one, so we specify WICED_ANY_PORT which lets the TCP/IP stack choose any available port) by calling:

```
wiced_tcp_bind( &socket, WICED_ANY_PORT );
```

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

a. Find the server address. This is passed as a WICED data structure of type *wiced_ip_address_t*. Let's assume you have defined a structure of that type called *serverAddress*.

You can initialize the structure in one of two ways – either statically or using DNS.

- To initialize it statically you can use the macros provided by the WICED SDK as follows:

```
SET_IPV4_ADDRESS( serverAddress, MAKE_IPV4_ADDRESS( 198, 51, 100, 3 ) );
```

- To initialize it by performing a DNS lookup, do the following:

```
wiced_hostname_lookup( "wwep.ww101.cypress.com", &serverAddress, 10000,
WICED_STA_INTERFACE );
```

b. Now that you have the address of the server, you make the connection to port 27708 through the network by calling *wiced_tcp_connect()* and waiting a TIMEOUT number of milliseconds for a connection. In our local network the timeout can be small <1s but in a WAN situation the timeout may need to be extended to several seconds:

```
wiced_tcp_connect( &socket, &serverAddress, 27708, TIMEOUT);
```

6A.3 Transmitting and Receiving Data using Streams

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 stream functions from the WICED SDK. The stream functions allow you to send and receive arbitrary amounts of data without worrying about the details of packetizing data into uniform packets (see the "further reading" section for details about packets).

To use a stream, you must first declare a stream structure and then initialize that with the socket for your network connection:

```
wiced_tcp_stream_t stream;

wiced_tcp_stream_init(&stream, &socket);
```

Once this is done it is simple to write data using the *wiced_tcp_stream_write()* function. This function takes the stream and message as parameters. The message is just an array of characters to send. When you are done writing to the stream you need to call the *wiced_tcp_stream_flush()* function, otherwise, the data won't be sent until a full packet is ready. The following code demonstrates writing a single message:

```
char sendMessage[] = "TEST_MESSAGE";

wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage));

wiced_tcp_stream_flush(&stream);
```

Reading data from the stream uses the *wiced_tcp_stream_read()* function. This function takes a stream and a message buffer as parameters. The function also requires you to specify the maximum number of bytes to read into the buffer and a timeout. The function returns a *wiced_result_t* value which can be used to ensure that reading the stream succeeded.

```
result = wiced_tcp_stream_read(&stream, rbuffer, 11, 500);
```

Behind the scenes, reading and writing via streams uses uniform sized packets. The stream functions in the WICED SDK hide the management of each of these packets from you so you can focus on the higher levels of your application. However, if you desire more control over the communication you can use the WICED SDK API to send and receive packets directly.

Once you are done with a stream, you need to call the deinit function before re-initializing it. Likewise, the socket must be deleted when you are done with it. This is fairly typical in server/client applications – that is, you open a socket, initialize a stream, read/write some data, then get rid of the stream and socket. A new socket and stream are created for the next set of data.

Given the above, the firmware to transmit data using streams might look something like this:

```
#define SERVER_PORT  (27708)
#define TIMEOUT (2000)
.
wiced_ip_address_t serverAddress;
wiced_tcp_socket_t socket;
wiced_tcp_stream_t stream;
char sendMessage[]="WABCD051234";
.
wiced_hostname_lookup( "wwep.ww101.cypress.com", &serverAddress, 10000,
WICED_STA_INTERFACE );
.
// Loop here for each message to be sent
wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);
wiced_tcp_bind(&socket, WICED_ANY_PORT );
wiced_tcp_connect(&socket, &serverAddress, SERVER_PORT, TIMEOUT);
wiced_tcp_stream_init(&stream, &socket);
wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage));
wiced_tcp_stream_flush(&stream);
wiced_tcp_stream_deinit(&stream);
wiced_tcp_delete_socket (&socket);
// End of loop
```
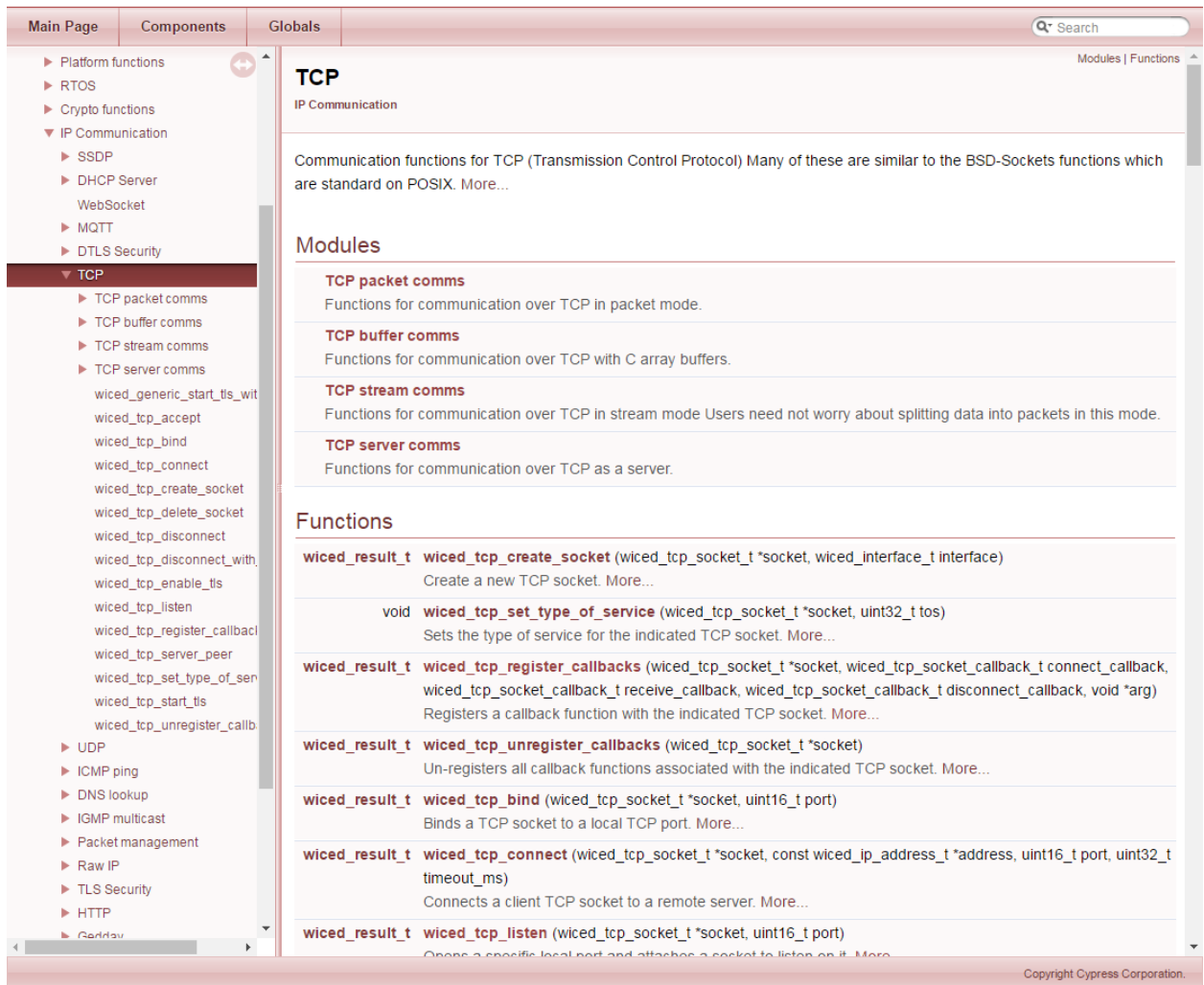
6A.4 WICED Socket Documentation

The WICED-SDK provides you a library of functions to do Socket based communication. The WICED documentation on sockets resides in Components → IP Communication → TCP. There are sub-sections for APIs specific for packet communication, buffer communication, stream communication, and server communication. We will mainly deal with stream communications, but the advanced exercises will also cover socket and server APIs.

6A.5 Exercise(s)

Exercise - 6A.1 Implement WWEP

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

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

- DNS name: wwep.ww101.cypress.com
- IP Address: 198.51.100.3
- Port: 27708

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

- The LED characteristic number is 5. That is, the LED state is stored in address 0x05 in the 256-byte register space.
- The "value" of the LED is 0 for OFF and 1 for ON.
- For the device ID, use the 16-bit checksum of your device's MAC address.
  - Hint: See the exercise on printing network information from the "Connecting to Access Points" 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.

The steps the application must perform are:

1. Connect to Wi-Fi.
   a. Hint: Use one of your projects from the previous chapter as a starting point.
2. Figure out your device number by adding the MAC bytes together in a uint16_t (effectively a checksum).
3. Use DNS to get the IP address of the server wwep.ww101.cypress.com or hardcode the IP address using INITIALIZER_IPV4_ADDRESS and MAKE_IPV4_ADDRESS).
4. Initialize the LED to OFF.
5. Setup the GPIO to monitor the button.
6. If the button is pressed:
   a. Flip the LED state.
   b. Send data to the server
      i. Format the message you want to send (using *sprintf()*)
         1. 'W<device number>05<state>'
         2. Hint: <device number> was calculated above
         3. Hint: <state> is '0000' for OFF and '0001' for ON
      ii. Open a socket to WWEP server (create, bind, connect).
      iii. Initialize a stream

      iv.   Write your message to the stream

      v.   Flush the stream

      vi.   Delete the TCP stream (Hint: *wiced_tcp_stream_deinit()*)

      vii.   Delete the socket

7. Go look at the console of the class WWEP server and make sure that your transactions happened.

8. Hint: Be sure to give any threads you create a large enough stack size (6200 should work).

Exercise - 6A.2 Update WWEP Client to check the return code

Remember that in the WWEP protocol the server returns a packet with either "A" or and "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 "*wiced_tcp_stream_read()*"

Exercise - 6A.3 (Advanced) Implement WWEP Server

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

Hint: use a linked list for the database so that it will start out with no entries and will then grow as data is stored. The WICED library has a linked list utility that can be found in the libraries/utilities directory. You can simply include it using #include "linked_list.h" which also provide the API documentation.

6A.6 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

6A.6.1 (Advanced) Transmitting Data using Packets as a TCP Client using the WICED SDK

At the beginning of your application, when you run the *wiced_init()* function, on the console you will see the message "Creating Packet pools". The packet pools are just RAM buffers which store either incoming packets from the network (i.e. Receive packets) or will hold outgoing packets which have not yet been sent (i.e. Transmit packets). By default, there are two receive packets and two transmit packets but this can be configured in your firmware. If you run out of receive packets then TCP packets will be tossed. If you run out of transmit packets you will get an error when you try to create one.

Each packet in the buffer contains:

- An allocation reference count
- The raw data
- A pointer to the start of the data
- A pointer to the end of the data
- The TCP packet overhead

Packet Buffer				
Type	Ref Count	Data Pointer		Buffer
		Start	End	
R	0	null	null	
R	0	null	null	
T	0	null	null	
T	0	null	null	

A packet starts its life unallocated, and as such, the reference count is 0. When you want to send a message, you call *wiced_tcp_packet_create()* which has the prototype of:

```
wiced_result_t wiced_packet_create_tcp(
        wiced_tcp_socket_t* socket,
        uint16_t content_length,
        wiced_packet_t** packet,
        uint8_t** data,
        uint16_t* available_space );
```

This function will look for an unallocated packet (i.e. the reference count == 0) and assign it to you. The arguments are:

- *socket*: A pointer to the socket that was previously created by *wiced_tcp_connect()*.
- *content_length*: How many bytes of data you plan to put in the packet.
- *packet*: a pointer to a packet pointer. This enables the create function to give you a pointer to the packet structure in the RAM. To use it, you declare: *wiced_packet_t *myPacket;* Then when you call the *wiced_packet_create_tcp()* you pass a pointer to your pointer e.g. *&myPacket*. When the function returns, *myPacket* will then point to the allocated packet in the packet pool.
- *data*: a pointer to a uint8_t pointer. Just as above, this enables the create function to give you a pointer to the packet structure in the RAM. To use it, you declare: *uint8 *myData;* then when you call the *wiced_packet_create_tcp()* you pass a pointer to your pointer e.g. *&myData*. When

the function returns, *myData* pointer will then point to the place inside of the packet buffer where you need to store your data.

- *available_space*: This is a pointer to an integer that will be set to the maximum amount of data that you are allowed to store inside of the packet. It works like the previous two in that the function changes the instance of your integer.

Once you have created the packet, you need to:

- Copy your data into the packet in the correct place i.e. using *memcpy()* to copy to the data location that was provided to you.
- Tell the packet where the end of your data is by calling *wiced_packet_set_data_end()*.
- Send the data by calling *wiced_tcp_send_packet()*. This function will increment the reference count (so it will be 2 after calling this function).

Finally, you release control of the packet by calling *wiced_packet_delete()*. This function will decrement the reference count. Once the packet is actually sent by the TCP/IP stack, it will decrement the reference count again, which will make the packet buffer available for reuse.After the call to wiced_tcp_packet_create_tcp:

- The pointer *myPacket* will point to the packet in the packet pool that is allocated to you.
- *availableDataSize* will be set to the maximum number of bytes that you can store in the packet (about 1500). You should make sure that you don't copy more into the packet than it can hold. In order to keep this example simple, I didn't perform this check in the above code.
- The pointer *data* will point to the place where you need to copy your message (which I do in the line with the *memcpy*).

Be very careful with the line that calls wiced_tcp_set_data_end as you are doing pointer arithmetic.

6A.6.2 (Advanced) Receiving Packets as a TCP Server using the WICED SDK

As a TCP Server you will probably have a thread that will:

- Call the *wiced_tcp_accept(&socket)* function which will suspend your thread and wait for data to arrive. Once data arrives it will wakeup your thread and continue execution. The RTOS has an "accept timeout", which by default will wake your thread after about 3 seconds. If it times out, the return value from *wiced_tcp_accept* will be something other than WICED_SUCCESS. It is then your choice what to do.
- Once the data has arrived you can call *wiced_tcp_receive*. This function has the prototype:

```
wiced_tcp_receive(
wiced_tcp_socket_t* socket,
wiced_packet_t** packet,
uint32_t timeout );
```

The *wiced_packet_t *** packet means that you need to give it a pointer to a pointer of type *wiced_packet_t* so that the receive function can set your pointer to point to the TCP packet in the packet pool. This function will also increment the reference count of that packet so when you are done you need to delete the packet by calling *wiced_packet_delete*.

- Finally, you can get the actual TCP packet data by calling *wiced_packet_get_data* which has the following prototype:

```
wiced_result_t wiced_packet_get_data(
wiced_packet_t* packet,
uint16_t offset,
uint8_t** data,
uint16_t* fragment_available_data_length,
uint16_t *total_available_data_length );
```

This function is designed to let you grab pieces of the packet, hence the offset parameter. To get your data you need to pass a pointer to a uint8_t pointer. The function will update your pointer to point to the raw data in the buffer.

Given the above, the receive firmware might look something like this:

```
while(1)
{
        wiced_packet_t *myPacket;
        uint8_t         *myData;
        uint16_t         frag_len,avail_len;

        result = wiced_tcp_accept( &socket ); // Suspend until packet is received

        if (result != WICED_SUCCESS) //  Probably a timeout occurred
                continue; //  Skip the rest of this iteration through the loop

        wiced_tcp_receive( &socket, &myPacket, WICED_WAIT_FOREVER );
        wiced_packet_get_data( myPacket, 0, &myData, &frag_len, &avail_len );
        myData[avail_len] = 0; // add null termination so we can print it

        WPRINT_APP_INFO(("Packet=%s\n", myData));

        wiced_packet_delete( myPacket );
        wiced_tcp_disconnect(&socket);
}
```

The code fragment assumes that it is a short string that you are receiving and it fits in one packet. And obviously, there is no error checking.

Note that the server disconnects the socket once it has received a packet (it does not DELETE the socket, it just disconnects from it). This is commonly done in TCP servers so that socket connections are not maintained when not necessary. Once the client opens another connection, the *wiced_tcp_accept()* call allows the server to receive the next packet.