**Chapter 6 TCP/IP Sockets**

**Introduction (Video 6-0)**

Hi, I’m Alan Hawse, Senior Vice President of Technical Staff for Solutions and Software at Cypress Semiconductor. Welcome to Chapter 6 of the Cypress Academy WICED WiFi 101 course. In the previous videos I have talked about how you build projects, how to use the WICED peripherals, how the RTOS works, what is in the library and finally how to get connected to the network. All that is cool… but your IoT device isn’t worth much without sending some data.

You could make an entire semester course on networking… well actually most engineering schools do. In fact here is Richard Stevens standard text on networking … quite the tome eh… But don’t despair. I am going to teach you enough in the next 6 videos to actually get things going (thumbs up). This is possible because WICED puts all the complexity together in an easy to use set of APIs.

As I thought about how to explain the material in this chapter, I struggled with making a nice linear flow of ideas (you could flash to a crazy spiders web or something) because all of the topics in this chapter have circular dependencies. So, be patient, you might find yourself re-watching a video or skipping forward or looking at the textbook. That is OK because all that matters to me is that you learn.

Now, without further ado, the videos in this chapter are:

The first video is about TCP/IP sockets - generally referred to as just “sockets” - where I will explain the fundamental mechanism of the internet to exchange data. In addition, I will explain a basic protocol I created to teach sockets called the WICED WIFI EXMAPLE PROTOCOL or “WWEP”

Then in video two I will explain how to implement my “real world” protocol using the WICED Socket API

Next, I will deal with the problem that everything so far has been unencrypted – and we know there are tons of people out there to get you. So, in video three I will introduce you to the basics of encryption in a gentle Alan Hawse kind of way. Specifically, I will introduce public and private key encryption.

Next I will show you the man in the middle attack… an insidious method to attack your privacy and the scheme used to prevent it

Video 5 is about X.509 certificates. These are data files that drive the tool used to prevent man in the middle attacks

And finally, in video 6, I will show you a complete WICED secure socket example.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions.

Lets exchange some data!

**TCP/IP Sockets & a Simple Application Protocol (Video 6-1)**

Hi, I’m Alan Hawse. Welcome back to Cypress Academy WICED WIFI. In this video, I will show you the tcp/ip socket – often just referred to as “sockets”. Sockets are the fundamental method of communication of internet applications. They are the plumbing of the internet.

A socket is simply a data pipe between a client and a sever. Both the client and server can put bytes in their side of a socket and the bytes are guaranteed to come out on the other side of the socket correctly and in order.

In other words, a client (probably your program) can open a socket to a server (probably in the cloud), then they will have a bidirectional pipe through the internet which they can use to communicate. For you science fiction fans, a wormhole through the internet which magically connects two points.

Every device on the internet has a unique IP address. In addition to the IP address, each device also has 65536 logical ports. You can think of the ports as numbered plugs where connections can be made. (a graphic here)

With the combination of an IP address and port number you can now uniquely identify all sockets on the internet. A socket is just Two pairs of numbers, server IP address and port number and the client IP address and port number.

Most of the application protocols that you are familiar with on the internet use sockets, http, dns, smtp, etc. you may not realize it yet… but I bet you already know some port numbers … for example 80 is used by HTTP servers and 443 by secure HTTP servers, and 25 by SMTP servers. In fact, the internet engineering task force (ietf) has a spec with a map of all the well-known ports. This spec is important because clients need to know which port to connect to for there to be communication that makes sense. There is no technical reason why your HTTP server must run on port 80, it could in fact run on port 25, but if it does that, no one will know what port to actually connect to. More over port 25 is assigned to SMTP, and if someone trys to open a socket to that port it will find your http server instead of an SMTP server. When this happens you will have total chaos.. wrath of god type stuff… dogs and cats living together (https://www.youtube.com/watch?v=WfVcvyxLj-s)

Now let me show you how this works. When I started working on this section I knew that I needed a very simple socket based protocol to demonstrate with. So, I created the WICED Wifi example protocol … and because everything needs an abbreviation, I call it WWEP

The WWEP is representative of many cloud protocols. My protocol is a simple database like protocol. A WWEP client can open a socket connection to a WWEP server, then it can write to or read from the database on the wwep server.

The wwep database acts like a 256 location register file for each of 65536 devices. Each location in the register file can hold a value up to 0xFFFF. Said more simply. The database has a 16-bit device id, an 8-bit location id and a 16-bit value. A client can then read or write into the database by specifying the device id, location id and value.

To use the WWEP database, the client can send a command to the server of r for read or w for write, followed by a device id, location and optional value … all in hex ASCII strings.

Then the WWEP database server will respond with an “A” for accepted followed by the device id, location id and value or just an “x” for rejected.

For example, a WWEP client can write the value 0x98AB to device ID 0x1234 in location 0xEF it would open a socket, then send W1234EF09AB. The “W” means write. The 1234 means device ID 1234 the EF means register ID EF and the 09AB is the value to save.

Then the server would respond with “A1234EF09AB”. The “A” means I accept your write to device “1234”, register EF and value “09AB”

Then a client could read back the value by opening a socket to the server and sending “R1234EF” which would make the server respond with “A1234EF09AB”. In this case the “R” means read, the 1234 means device ID 1234 and the EF means register EF. The servers response of “a” means accepted, 1234 means device 1234, the EF means register EF and the 09AB is the value we previously wrote.

WWEP is simple text based protocol. But it works almost exactly like HTTP, which also is a simple text based protocol which I will talk about in detail in chapter 7.

In the next video, I will show you the WICED APIs required to implement sockets.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions.

**WICED Socket API (Video 6-2)**

I’m Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. Lets get going. In the previous video I introduced you to tcp/ip sockets and explained a basic application protocol called WWEP. In this video I am going to explain the WICED APIs that you will need to use to create a basic socket connection.

In the beginning, there was the internet and attached to the internet through the magic of WICED there was a server and a client (pop them onto the screen) and it was good. Remember from the previous chapters that connecting your IoT device to the internet is simply done by configuring wifi\_config\_dct.h, then calling “wiced\_init” and “wiced\_network\_up”. WICED will then take care of doing all the networking and wifi stuff required to make a robust connection.

To make a connection through the internet the client will first call wiced\_tcp\_create\_socket. This function will turn on a socket inside of the network stack. This socket won’t be connected to anything, either inside of your firmware, or out on the internet…. But it is the point that eventually will be the connection between your client firmware and the server.

Next you call “wiced\_tcp\_bind” which will connect your socket to a TCP/IP port on your device. But which port? The answer to that question is “WICED\_ANY\_PORT” but why any port? The answer is, it doesn’t matter what your outgoing port is. The client only cares what the port is on the server. So, you should always let WICED decide which outgoing port to give you, as it will assign some number automatically for you that is not currently in use… and is outside of the numbers that are “well known”.

After you have a socket and it is bound to a port, the next step is to call “wiced\_tcp\_connect” which will make the connection through the “cloud” all the way to the server on the other side. The arguments to this function are “socket” which you just created with wiced\_tcp\_create\_socket, “server” which is the IP address of the server (which you can find using wiced\_hostname\_lookup), PORT which is the port number on the server, and finally timeout… how long you want to wait.

Now lets look at the server. In many ways the server is the mirror of the client. It starts by calling wiced\_tcp\_create\_socket. Then you call wiced\_tcp\_listen, which attaches your socket to the local port specified in the arugments. Finally you need to enable the server by calling wiced\_tcp\_accept, which turns on the socket to accept data from clients.

Let me show you the whole picture. On the client you call wiced\_tcp\_create\_socket to make the socket, wiced\_tcp\_bind to hook it to a local port, wiced\_tcp\_connect to hook it to a server. On the server side you call wiced\_tcp\_create\_socket, wiced\_tcp\_listen to connect the server to a local port and wiced\_tcp\_accept to turn on the incoming socket.

Once you have a connection made via a socket from the client to the server, now what? The answer to that is the final piece of the puzzle. Stream. It is possible to use WICED to look at each individual TCP/IP packet… but that is painful. So, we created “wiced\_tcp\_stream” which abstracts all of that mumbo-jumbo into a simple byte stream.

To attach a stream to your socket, first call wiced\_tcp\_stream\_init which will create a stream… then you can call wiced\_tcp\_stream\_write to send bytes one or more times … after you have finished writing you should call “wiced\_tcp\_stream\_flush” to send any remaining bytes the write buffer.

To read from a stream you just call wiced\_tcp\_stream\_read to read bytes.

Both the write and read will return after the bytes have been written… or read … or after a timeout. You should look at the wiced\_result\_t returned by both of these functions to figure out what happened. Many things can go wrong, for instance the connection gets dropped or the server is busy. In which case you would want your program to return and let you decide what to do next.

In the next video I am going to show you the exact implementation of the WWEP client and server

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions.

**Explain the WWEP Client and Server (Video 6-3)**

Hi, I’m Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. Lets get back to it. In the previous videos I took you through 1 what sockets are, 2 a simple application protocol I called WWEP, and 3 how to create, read and write to sockets using the WICED apis.

Lets put this all together and make an end to end application that implements the WICED WIFI Example Protocol.

Remember from video 2 that the WWEP is a simple database protocol. The client can write values into the database in the form of W for Write, Device Number – a 16 bit device number, Register – an 8-bit register, and a 16-bit value. For instance W1234567890 will write the value 7890 into device 1234 register 56. The client can also read by sending “R123456” which the server will respond with “A1234567890” in other words, the value 7890 which is in register 56 on device 1234.

First, let me show you the client: The client is really simple. I will just read the state of a button, and then send alternating 0s and 1s, to a register which is just the 16bit checksum of my mac address on the server.

The project starts with wiced\_init and wiced\_network up. I then create a thread to process the button presses and send them to the server. In that thread I start by getting my mac address by calling wiced\_wifi\_get\_mac\_addres, then I calculate a checksum. Then I use wiced\_hostname\_lookup to find the address of the WWEP server. Finally I wait until the user preses a button, using the semaphore … remember the semaphore from the RTOS videos

The last piece of the client is the sendData function which actually does the Socket connection. First it calls wiced\_tcp\_create socket, then wiced\_tcp\_bind, and finally wiced\_tcp\_connect. Notice that I am carefully checking the error codes from each of these API calls.

Now that I have a connection I format the message using sprintf, then I make a stream with wiced\_tcp\_stream\_init, write and flush the message with wiced\_tcp\_stream\_write and wiced\_tcp\_stream\_flush and finally close everything and move on.

Now let me show you the server. It is also pretty simple, though I have added a good bit more error checking because people kept crashing it during my class. The server has a little bit of initialization with wiced\_init and wiced\_network\_up. I use the linked list from the library (remember chapter 4) to be a simple “database” I also create a thread which I call nonsecure (or is it insecure or unsecure, I don’t know… oh well ). The reason I call it nonsecure is that all of these transactions are taking place on the “open” internet in plain text. In the next several videos I will introduce encryption to this system.

The server thread creates a socket with wiced\_tcp\_create\_socket, initializes a stream on that socket with wiced\_tcp\_stream\_init … then I attach my socket to the port with wiced\_tcp\_listen. Now I need to make an infinite loop that accepts a connection (wiced\_tcp\_accept) …. When I get a connection from a client, I read its IP address (for printing on the screen) read the data (and send it to the database with processClientCommand, I print the output with displayResult, send the response with wiced\_tcp\_stream\_write and wiced\_tcp\_stream\_flush… and finally clean everything up with wiced\_tcp\_stream\_deinit… and wiced\_tcp\_stream init… then I am ready to go again.

You can dig through the database read & write functions by looking at the function processClientCommand and the output function displayResult which are available on the GitHub site for this class. Neither function has anything to do with tcp/ip sockets.

In the next video I am going to fix the problem of a lack of security by introducing encryption.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions.

**Encryption basics (Video 6-4)**

Hi, I am Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. Lets get back to it. In the previous videos I showed you how to create and use TCP/IP sockets with the wiced api. That is cool, but with the NSA and every other criminal organization on the planet spying, you had better get some security for your customers.

Big picture, in the world of IoT you need to do two things to have security in your communication. One, you need encryption so that you are sure that what you send and receive can only be read by you (the client) and the server in the cloud. That is the focus on this video.

The second thing you must do to have a secure connection is to ensure that you know the identity of who you are talking to. I will talk about this problem in a later video.

The first part of the security problem, how to send & receive data securely is handled with encryption. At this point encryption is basically a big nasty math problem… and there are a bunch of people who have gotten PhDs working on it. I am not going to get into that at all…and quite frankly do not need to because the world of encryption is divided up into two simple areas, private key or symmetric key and public key or asymmetric key.

First, private key or symmetric. This system works by both sides of the connection knowing a secret key. To perform an encryption you take your un-encrypted or plain text data and you combine it mathematically with the private key to create encrypted data. This scheme are awesome in that it is easy from a computation standpoint to perform the encryption and de-encryption… but it is really hard computationally to unencrypt without the key. There are a bunch of these private key schemes including AES, triple des, blowfish etc. Here is the cool part. You don’t really need to know about the algorithms because the ones that matter are already built into the Cypress tools.

So, what is the problem with private key? The answer is simple to say but hard to solve, how do you get both sides of a connection to have a shared private key without sending something over the internet where it can be discovered? The answer to this question is really really super cool. Public Key Encryption.

Public key encryption was created by a crazy mathematical genius. It works like this. There are a pair of keys, one is named the public key and one is named the private key. You can publish your public key, and anyone can encrypt data with it. But only your private key can unencrypt data that was encrypted with your public key. This means that anyone can send you a message, that only you can read, by encrypting the message with your public key. These public/private keys always occur in 1:1 pairs. The other attribute of a keypair is that it doesn’t matter which one you call public and which one you call private because data encrypted by one can only be unencrypted by the other. One thing I will add.. and it should be obvious… but Ill say it anyway. Keeping the private key private is really important because if someone discovers the private key then the whole system falls apart. This is one place where Cypress excels, our security supports really private, private keys.

So, what is the problem? The answer to that question is that it is computationally expensive to do unencryption with public/private keys. You would not want to have to do this on a low powered IoT device all of the time. So, how do you solve that problem? Simple, you use public key encryption to exchange a symmetric key… then all your exchanges after that are with the symmetric key. Like this

1. A client attaches to a server
2. The server sends the client its public key
3. The client creates a big random symmetric key
4. The client encrypts the symmetric key using the servers public key
5. The client sends the encrypted key to the server
6. The server uses its private key to un-encrypt the symmetric key
7. All future exchanges happen with the symmetric key.

There are several ways to make this whole scheme more secure… and in fact wiced implements those for you automatically. There is one more weakness called “man in the middle” which I will talk about in the video after next.

Now that you know the basics of making a secure connection what do you do next? The answer to that question is: you make secure sockets. This was originally called “SSL” or secure socket layer by Netscape. More recently it is called “transport layer security” or “TLS”. That is what the “s” is in “https” stands for.

How do you do that in wiced? On the client side you just modify your socket code to call “wiced\_tls\_init\_context” after you wiced\_tcp\_bind the socket. Then you call “wiced\_tcp\_enable\_tls” right before you call wiced\_tcp\_connect to the server. And wah-lah… magically you have a secured socket. Of course, all of this assumes that you are talking to a server on the right port that support secure sockets.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions

**Man in the middle (Video 6-5)**

Hi, I am Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. In this video I am going to talk about the man in the middle… no Scott not the Bee-gees probably most of the people watching this video haven’t ever heard of them… I am sorry about that, my editor is stuck disco era. No, the man in the middle in I am talking about the internet security attack where you think you are talking to a specific server, but really a bad actor is intercepting the message, de-encrypting and then relaying it on. This puts him in the position of being able to decode the traffic to and from the server.

Let me show you

If your client attempts to make a connection to a server I will called Real Server but an evil server called MIM gets in the middle. The first thing that happens is the mim will send you its public key. Then it will open a connection to the realserver. You will get the public key of the mim, and you will send it a symmetric key. The real server, will send its public key to the MIM. Then MIM will send a symmetric key back to the real server.

Now you think that you are connected to the real server securely. But you are really connected to the MIM. And the real server thinks it is connected to you, but really it is also connected to the MIM. Then you encrypt traffic and send it to the MIM, the MIM decrypts it because it has your key, then it reencrypts it and sends it onto the real server.

This can happen in the real world fairly easily as you don’t have control over any of the routers between you and the server. The mostly likely bad actor is a compromised wifi access point, but it could be anywhere.

Now that you know how this happens. How do you prevent it? The answer to that question is certificates…. To understand certificates you need to know two new things. First, a cryptographic hash. And second, a certificate authority.

A cryptographic hash is just a fancy checksum often called a hash. The fancy part of the cryptographic hash involves a little bit more crazy math. Specifically, the algorithm you follow is essentially a guarantee that any known input will only generate one hash… and that any change to the input will always generate a different hash. These hashs are sometimes called message digests because they take a big bunch of data and squish it down into a small amount of unique data. For instance sha-256 takes any amount of data and generates a unique 256 bit number.

Now that you know what a cryptographic hash is, let me explain a little but more about the certificate.

In order to be sure that the server you are talking to is actually the server you mean to be talking to, you can build the public key of that server into your firmware. Then, when you make a connection, the server can send you a secret message that is encrypted with its PRIVATE key. You can then decode that message using the servers public key, which you have built into your firmware… and then look at the message. If the message makes sense then you are for sure that it was sent only by that server.

But wait. That means you need to have the public keys of all of the servers you want to talk to built into your firmware. That is not practical. So now what? Well the solution to that problem is to use a certificate. A certificate is a file that has the public key of a server that you know and trust and the public key of the server you are talking. It also includes the DNS name of the server you are talking to, PLUS a signature.

What is this signature you are talking about? Well the signature is the public key of the certificate authority and the public key of the server you are talking to, hashed with a cryptographic hash and then encrypted with the private key of the certificate authority.

When you open a connection to a server, it sends you its certificate. You then take the public key of its certificate authority and the public key of the server, you hash it. Then you decrypt the signature that was sent you in the certificate, using the certificate authorities public key. Then you compare the decrypted signature with the hash you just created. If they are the same then you know that the hash was created using the certificate authorities private key. Which can only have been done by the certificate authority.

By extension if you trust the certificate authority, then you trust the server. Basically, you are depending on the fact that a certificate authority would only sign certificates for servers that are real.

There are a bunch of other things that can happen in certificates, for instance you can create a chain of trust in other words, a trusts b and b trusts c … so if you trust a then by extension you trust c. Certificates can expire, they can be restricted to a particular host, or domain, etc. However, I have given you all of the fundamental properties required to understand and I recommend that you start by letting wiced take care of the details.

I described how a client verifies a servers certificate. This exact method can optionally also be used by a server to verify a clients identity. It is common for HTTP connections to not verify the client identity. But many other connections including MQTT often verify both sides. I think that in the future we will see many more full certificate verification protocols.

The last trick of this whole puzzle is that you start a connection using public key cryptography, then you use one of the many key exchange systems, for example elliptic curve diffe helman, to switch your encryption to the more efficient symmetric key encryption.

These certificates that I have been talking about are generally in a format called X.509. In the next video, I will talk more about how you actually get, create, store and use them in your project.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me @askioexpert with your comments, suggestions, criticisms and questions

**X.509 Certificates (Video 6-6)**

Hi, I am Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. In the last several videos I have talked about the fundamentals of encryption and secure sockets. In this video I will talk to you about X.509 Certificates. If go to youtube you will find tons of videos that have x.509 in the title… oh no… there goes Scott my editor again. No, Scott I am not talking about portugese rap stars… Im talking about all of the other videos that discuss X.509 certificates. What is it with you and the music?

Anyway.. remember from the previous video that a certificate contains the public key of the certificate authority and the public key of the server you are trying to talk to, along with a signature that is created with a cryptographic hash. Well there is a standard format for all of this information called “X.509”. In order for your IoT device to verify the identity of a remote server, you will need to build in the X.509 certificate of the certificate authority of your server.

In the rest of this video I am going to show you what an X.509 certificate looks like, how you can get one, how you can look at them, how to create your own certificate, how to store them into your wiced firmware and finally how to retrieve them for use in your project.

Mostly commonly these x.509 certificate are stored in “dot pem” or privacy enhance email format. PEM looks like this, it starts with a “begin certificate” then it has a bunch of text in base64 encoding DER format, then it has an “end certificate”. OK. That does not do you much good unless your brain happens to be able to decode base 64 encoded DER format.

But don’t despair. There are several ways to decode these files. First, you can use a command line program like “openssl”. If I run openssl x509 -in Let’s Encrypt Authority X3.pem file that I got from the website HTTPbin.org I will see a bunch of stuff including the public key, who signed it, the expiration date, the signature etc. I can also look at the information about the certificate by pasting it into the sslshopper.com website which has a decoder – basically it just runs openssl for you.

On a mac you can also look at any websites certificate by pressing the lock and then show certificate. When you press details and scroll down you will see all of the specifics.

On a PC in chrome you can press the three dots, go to the developer tools and then view certificates and then you can click through the information.

So, now you know what they look like and how to examine them. How do you get them? The most surefire way in my experience is to download them from the website of the server you are trying to talk to. Ill show you how to do that on httpbin.org. First I go to <https://httpbin.org>. Then I click the lock. Finally, I click the certificate and drag it onto my desktop. This file will be in a binary DER format which I can change to a text PEM format by running openssl.

On a PC you can follow a similar process. First you load the website. Then you go to the developer tools. Then view certificate and finally details. On that tab you can “copy to file”… and then you are off to the races.

In my text book on the internet you can look at each of these steps on several OSs and browsers with step by step screen shots.

If you want to create your own secure server you can run openssl to create your own self signed certificates. Openssl is built into mac and linux, and can be downloaded into CyGwin on PCs. You should be aware that if you self sign a certificate there are some systems which wont trust it because they wont have your certificate authority built into their software.

There are three techniques for putting a certificate authorities root certificate into WICED. The first technique is to install it into the Device Configuration Table … remember the DCT which we talked about in chatpter 5. To make this work you need to first

1. Convert the certificate to PEM format
2. Then store that file in resources/apps/yourapp
3. Add the “certificate: key to the make file. Like this
4. Then in your program you can load the certificate out of the DCT using wiced\_dct\_read\_lock.
5. Finally you can register the certificate using wiced\_tls\_init\_root\_ca\_certificates – which I will talk more about in the next video

The second technique is to store the certificate in the Resources filesystem. To do this you

1. Convert the certificate to a pem file
2. Save the file into the resources/apps/yourapp/ directory
3. Add the $NAME\_RESOUCE:= apps/yourappname/yourfile.pem
4. Add the #include “resources.h” to your project
5. Then you can load the certificate using the wiced api resource\_get\_readonly\_buffer
6. And finally register the certificate using wiced\_tls\_init\_root\_ca\_certificates

The last technique is painful and I am not a huge fan of. However, you can hand edit the PEM file to be a c character array with \r\ns on the end of each line. It will look like this. Then embed that string into your program. And finally call wiced\_tls\_init\_root\_ca\_certificates on that string.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions

**A complete secure connection (Video 6-7)**

Hi, I am Alan Hawse. Welcome back to Cypress Academy WICED Wifi101. Chapter 6 has been all about how you exchange data on the internet. I showed you how to use sockets, then we talked about how encryption works, and how to prevent man-in-the-middle attacks. In the final video for this chapter I will show you how to modify the non-secure WWEP client program from earlier to include security specifically transport layer security.

Lets get started. The first step is to copy the 02\_client project into a new project called 03\_secure\_client. As always don’t forget to modify the make target, filenames, and makefile.

The first step in the process of modifying the original project is to copy my file “wwep\_cert.pem” into the resources/certificates/ directory. The file wwep\_cert.pem is a self signed certificate that I created by running openssl like this …. This certificate is used for the secure version of the wwep server to support a secure TLS socket.

The next step is to add the CERTIFICATE keyword to my make file with the path to the wwep servers pem file. This will cause the certificate to be inserted into my DCT.

Now I need to modify the c file. At the beginning of the application\_start function I want to read the secure wwep servers certificate out of the DCT by calling wiced\_dct\_read\_lock. Then I call the function wiced\_tls\_init\_root\_ca\_certificates.

I showed this function earlier… but I didn’t talk about it. What this function does is add the certificate to the list of certificates that WICED uses to verify secure tls sockets. In our case there will be only one certificate but, you could have multiple certificates. For instance If you think about a web browser, it may have 100’s of root certificates built in. Obviously, this is not possible on a smallish IoT device. So you should think carefully about getting only the root certificates you need in order to safely connect built into your program.

Once you have registered the server root certificate using wiced\_tls\_init\_root\_ca\_certificates, wiced will then automatically verify all TLS connections using that certificate.

The last step in fixing the non-secure wwep client program to be secure is to modify the sendData function. Remember from the earlier video that each time the button is pressed, the sendData function is called. That function opens up a socket, formats the data, sends it, reads the response, then closes the connection.

For us to make a secure connection we will make two small modifications to that function. Specifically

1. After the socket is bound with wiced\_tcp\_bind, I will create and initialize a security context by calling wiced\_tls\_init\_context.
2. Then I will call wiced\_tcp\_enable\_tls which will automatically make wiced open up a secure socket connection when I call wiced\_tcp\_connect

Remember that to get a secure socket going you have to exchange certificates, verify them, and the do the symmetric key exchange. The nice thing about wiced is that you don’t need to think about any of that as wiced does it automatically for you.

Great! That’s it for chapter 6. Don’t forget that you can look at my textbook that goes with all of these videos, and review the actually projects on our website.

In the next chapter we will start digging into the important web protocols including http, mqtt, coap and amqp.

You can post your comments and questions in our Wifi developer community or as always you are welcome to email me at alan\_hawse@cypress.com or tweet me at @askioexpert with your comments, suggestions, criticisms and questions