**Chapter 7**

**HTTP (Video 7-2)**

Hi, I’m Alan Hawse, welcome back to WICED WiFi 101. Last time we talked in detail about the HTTP protocol – particularly version 1.1. Now I'm going to show you how to use HTTP inside of a WICED IoT device.

First, remember that HTTP is just a simple ASCII text-based protocol. You send a client start line, optional headers and then a body… and the server responds back to you with a message that has a response start line, headers and an optional body.

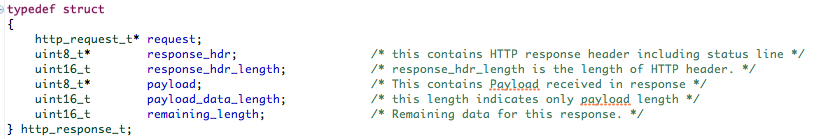
This sounds like, I don't know, kind of a bit of a pain, but fortunately WICED provides you APIs to build the Client Request Start Line, Headers, and the Content. It also provides you with functions to parse the output that comes back to you in the form of a Server Response so you can find all of the information that you're looking for.

The WICED SDK has several built-in HTTP libraries for both HTTP 2.0 and HTTP 1.1. In this video, I'm only going to talk about the library that's in protocols/HTTP\_client, which provides support for HTTP 1.1 Clients.

The process flow to make the HTTP\_client library work, is:

* Initialize the http client http\_client\_init
* Optionally initialize the client identity if using TLS wiced\_tls\_identity
* Optionally configure the TLS properties http\_client\_configuration\_info\_t
* Optionally initialize the HTTP Server root cert wiced\_tls\_init\_root\_ca\_certificates
* Make a connection to the HTTP server http\_client\_connect
* Initialize an HTTP request http\_request\_init
* Initialize an array of HTTP headers http\_header\_field\_t[]
* Write the headers http\_request\_write\_header
* Write the end (the blank line "\r\n") http\_request\_write\_end\_header
* Optionally write the content body http\_request\_write
* Flush the writes http\_request\_flush

Once you do all of that, the request is sent so you just wait for the callback function which you had registered when you initialized the client. In the callback, you get an event code such as HTTP\_CONNECTED, HTTP\_DISCONNECTED, or HTTP\_DATA\_RECEIVED, and the actual response data. The data comes in the format of a structure that looks just like this…



Once you do all of that, the request is sent so you just wait for the callback function which you had registered when you initialized the client. In the callback, you get an event code such as HTTP\_CONNECTED, HTTP\_DISCONNECTED, or HTTP\_DATA\_RECEIVED, and the actual response data. The data comes in the format of a structure that looks just like this…

As you can see, the structure separates out the header and the payload. The library provides a function called http\_parse\_header which will search through the headers and return the value for the header that you specify.

You are responsible for parsing what you need from the payload, but again, the WICED libraries can help – remember we talked a lot about the JSON parser in Chapter 4. That library will come in handy since most of the IoT devices in the world use JSON formatting for their messages.

The last item in the structure is the remaining length. This is included because very long responses may not fit into a single message. In that case, the remaining length will be a number larger than 0. The callback function will get called again when the next set of data is available. This will happen over and over again until the remaining length is 0. In most IoT devices I expect the messages to be short, so you will probably never have to deal with a message that is split over multiple transactions, but the functionality needs to be there if you need it for long responses.

All this data is freed by calling the http\_request\_deinit. Typically, you'll call a function from the HTTP callback for the event HTTP\_DATA\_RECEIVED when the value of remaining\_length in the response structure is equal to 0.

There is also a function called http\_client\_deinit that you should call after the server disconnects, or after you decide to disconnect by calling the http\_client\_disconnect function. The deinit function must NOT be called inside the HTTP callback, because you would be removing a thread that's currently running. Therefore, in the HTTP callback, when you get the HTTP\_DISCONNECTED event you should set a flag and then call the http\_client\_deinit from a different thread (like say the application\_start thread) when that flag gets set. Or, often I just use a semaphore for that function.

That's all there is too it! Just to illustrate, let me walk you through a couple of completed exercises that use HTTP.

First, I'll show you a project that connects to a server at http://httpbin.org. The site httpbin.org is meant to be used by developers to test their HTTP verbs like GET and POST. That site has a resource called /html that will return - you guessed it - an html document - and another one called /anything that will just echo back anything that is passed to it as content body. You can go there from a Web browser first to see what it looks like. In fact, let me do that for you now. In my browser, I type httpbin.org/html and hit Enter. Here you can see when I did that, my browser returns the first bit of Moby Dick as an HTML document. Did you guys read Moby Dick? I didn't in high school like I should have and then when I got older I tried over and over again for several years until I finally made my way through it. It was totally worth it as the book is amazing. Alright, let's get on to HTTP.

Now let's go to httpbin.org/anything. This resource is setup to just echo back what it got so you can see that my browser sent an HTTP GET method with all these various headers – here is the required Host header – and look - no data.

Now let's open the WICED Studio project that will connect to httpbin.org and it will issue a GET method on the /html and we'll use the /anything resource. We will just print out what each one returns to the terminal window, so you can get some practice trying this out.

First off, in the make file you'll see that I included protocols/HTTP\_client library and at the top of the C file I included "http\_client.h".

I then setup a macro for the server host which is httpbin.org, and the port, which is the standard port of 80.

Next, I declare two structures – one will hold the client information and one will hold the request information. These will be initialized later in my project.

In application\_start I declare a header structure - I will only have one header – the required host header – so I'll make an array of 1 header field. Now I will enter the header information into the structure. The entry for the field is "HTTP\_HEADER\_HOST" which is a macro that just inserts "Host: ". Next is the length of the field, then the value to set which is "SERVER\_HOST" which just inserts "httpbin.org". Then finally, the length of the value. All of this will result in a header of "Host: httpbin.org" which is exactly what I need to send.

Next, I initialize WICED and create a semaphore, connect to my WiFi network, and lookup the IP address of httpbin.org.

Then I initialize the HTTP client by passing it the empty client structure and telling it the name of my callback handler function – in this case it's called event\_handler, which we'll see in a minute.

Now I connect to the server from our HTTP client. I tell it the IP address to connect to, the port, and to use the non-secure version of HTTP. The connect timeout is set to 3 seconds – it needs to be long enough for the server to respond and unless you are dialing up on a 300 baud modem from the 1980's, that should be plenty of time.

Then I do a sequence of calls to initialize the request, write the header, write the end header, and flush the request. At this point, our GET request is sent to the server. We just need to wait for the callback function, so I wait for the semaphore that I setup earlier.

Jumping down to the event handler, you can see that for the HTTP\_DATA\_RECEICED state, I pull the header out, I print it to the UART, and then I do the same thing for the payload.

If the remaining length is 0 then I know the response is done, so I deinit the request – remember, to free up the memory - and I set the semaphore. If it isn't 0, I don't set the semaphore because I know there is more data to come, and the callback will just be called again.

Jumping back to the main program, once the semaphore has been set, I know the previous response is done so I can start another new request.

Just to be safe I make sure that I'm still connected – remember the host can disconnect at any time it wants to. If I'm not connected, I will try to reconnect before moving on. Then I setup a new GET request – this time to /anything instead of /html that I used before – and don't forget to flush it.

Once again, I wait for the callback by getting the semaphore. The callback goes through the same process then deinits the request and sets the semaphore. After that, I disconnect and deinit from the server to clean things up before exiting.

One last thing – notice in the callback function that I called http\_client\_disconnect when I get an HTTP\_DISCONNECTED event. Remember that the host can disconnect any time it wants. When it does, I get the callback event. I disconnect on my side so that the two ends stay synchronized. If I didn't do that, when I tried to reconnect it wouldn't work because the client on my end would think that it's already connected.

That's not too bad, right? Now let's run the program and see what happens.

OK, the HTTP GET to /html returned a lot of data, so it came back in three chunks -we get the response header and then 3 separate response payloads. See how the callback didn't print the "End Response" message until after the last chunk? That's because it saw the remaining length wasn't 0 until the 3rd chunk was delivered.

After that we get the echo back from the HTTP GET to /anything. This says we sent a GET method with a Host header and a Connection header – the second one was added by your library.

At this point adults use secure connections…. So let’s do the same thing now except now we'll use a secure TLS connection.

First off, I'll change the port from 80 to 443 since HTTPS runs on port 443. Next, I'll declare a client configuration structure that will be used later.

All the initialization and WiFi network connection is the same. Once the device has connected to WiFi, I'll read the root certificate for httpbin.org from the DCT. I got the certificate by going to https://httpbin.org with a web browser, viewing the root certificate, and downloading it. If you don't remember how to do this, or you want a refresher about certificates, take a look at chapter 6B.

Next, I need to setup the client configuration structure to define a max TLS fragment.

Then I initialize the client and call the function to configure it. Notice that I still used NULL as the last argument to the client initialization function. That argument would be used if the other side of the connection wanted a certificate from my device to verify my identity. That website doesn't do it, so I don't need to supply my own certificate. That is, my device verifies who they are, but they don't verify who I am. That makes sense in this case because the httpbin web site will allow anyone to connect to it. Later on, when we talk about connecting to the Cloud provider Amazon Web Services, the connection will need to be verified by both sides.

The final change is to connect the client to the host using HTTP\_USE\_TLS instead of HTTP\_NO\_SECURITY. That tells the library function to open up a secure TLS socket to the web server.

Everything else – building the request, sending it, receiving data back from the server – is exactly the same. It's not hard to use secure HTTP and this is something that you absolutely should do. If you don't do it, you may find your IoT devices hijacked by somebody bad. So please, please use secure sockets.

Now when I run the project, you will see that instead of connecting to httpbin.org on just http, it will connect to https://httpbin.org but otherwise I'll get the same data back. In this case, I'm sure the site I connected to is really who they claim to be, and the data was sent encrypted, so nobody could see it during the transmission. Remember, the purpose of the certificate is to ensure you data is encrypted and there's no man in the middle. Once again, go back to chapter 6 and listen to my security talk.

That's all there is to using the WICED HTTP client library in WICED. The manual and solution projects have lots of other examples that do HTTP POST, use Web APIs, and other cool things, so you should check them out.

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