|  |  |  |
| --- | --- | --- |
| Cell | **Visuals** |  |
| 1 | TEXT ON SCREEN:  Download the manual and solution projects at:  [www.cypress.com/training/wicedwifi-101](http://www.cypress.com/training/wicedwifi-101) | Hi, I’m Alan Hawse, welcome back to WICED WiFi 101. Last time we talked in detail about the HTTP protocol. Now I'm going to show you how to use HTTP inside WICED. |
| 2 |  | 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 then the server responds with a message that has a response start line, headers and an optional body. |
| 3 |  | This sounds like a bit of a pain, but fortunately, WICED provides you APIs to build the Client Request including the Start Line, Headers, and Content. It also provides you with functions to parse the output that comes back in the form of a Server Response so that you can find the information you are 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 in protocols/HTTP\_client, which provides support for HTTP 1.1 Clients. |
| 4 | TEXT ON SCREEN:  See text at end of this document (TEXT1) | The process flow to make the HTTP\_client library work, is:  …  …  …  … |
| 5 | SCREEN CAPTURE: | Once you do all that, the request is sent so you just wait for a callback function which you 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 like this…  As you can see, the structure separates out the header and 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. |
| 6 |  | You are responsible for parsing what you need from the payload, but again, the WICED libraries can help – remember the JSON parser that we talked about in Chapter 4? That library will come in handy here since most IoT devices use JSON formatting for their messages. Cool. |
| 7 |  | The last item in the structure is "remaining length". This is included because very long responses may not fit in 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 until the remaining length is 0. In most IoT devices the messages are short, so you will probably never have to deal with a message that is split over multiple transactions, but the functionality is there if you need it. |
| 8 | TEXT ON SCREEN:  See text at end of this document (TEXT2) | All this data is freed by calling http\_request\_deinit. Typically, you will call that function from the HTTP callback for the event HTTP\_DATA\_RECEIVED when the value of remaining\_length in the response structure is equal to zero.  There is also a function http\_client\_deinit that you should call after the server disconnects, or after you decide to disconnect by calling http\_client\_disconnect. The deinit function must NOT be done inside the HTTP callback, because you would be removing a thread that is currently running. Therefore, in the HTTP callback, when you get the HTTP\_DISCONNECTED event you should set a flag and then call http\_client\_deinit from a different thread (like application\_start) when the flag is set.  That's all there is too it! Just to illustrate, let me walk you through a couple of completed exercises that use HTTP. |
| 9 | SCREEN CAPTURE: | First, I'll show you a project that connects to the server at [http://httpbin.org](http://httpbin.org/anything). The site httpbin.org is meant to be used by developers to test out things like GET and POST. It has a resource called /html that will return (you guessed it - an html document) and /anything that will just echo back anything that is passed to it as a content body. You can go there from a Web browser first to see what it looks like. In fact, let me do that now. In my browser, I'll 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 have tried over and over for several years… until I finally made my way through it… it was totally worth it as the book is amazing. Anyway…  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 various headers – here is the required Host header – and no data. |
| 10 | SCREEN CAPTURE: | Now let's open the WICED Studio project that will connect to httpbin.org and will issue a GET method on the /html and the /anything resource. We will just print out what each one returns to a terminal window.  First off, in the make file you'll see that I included the protocols/HTTP\_client library and at the top of the C file I included the "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 client information and one will hold request information. These will be initialized later. |
| 11 | SCREEN CAPTURE: | In application\_start I declare a header structure - I will only have one header – the required host header – so I'll make that 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 that field, then the value to set "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. |
| 12 | SCREEN CAPTURE: | 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 is called event\_handler. Which we'll see in a minute. |
| 13 | SCREEN CAPTURE: | 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 in 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. |
| 14 | SCREEN CAPTURE: | Jumping down to the event handler, you can see that for the HTTP\_DATA\_RECEICED state, I pull out the header, print it to the UART, and then do the same thing for the payload.  If the remaining length is 0 then I know this response is done so I deinit the request (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 to come, and the callback will get called again. |
| 15 | SCREEN CAPTURE: | 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 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 - and flush it.  Once again, I wait for the callback by getting the semaphore. The callback goes through the same process and then deinits the request and sets the semaphore. After that, I disconnect and deinit from the server to clean things up before exiting. |
| 16 | SCREEN CAPTURE: | 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'll get this 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 it was already connected. |
| 17 | SCREEN CAPTURE: | That's not too bad, right? Now let's run the project and see what happens.  OK, the HTTP GET to /html returned a lot of data, so it came 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 that 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 that we sent a GET method with a Host header and a Connection header – the second one was added by the library. |
| 18 | SCREEN CAPTURE: | At this point adults use secure connections…. So let’s do the same thing except now we will 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 get 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 if 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 length. |
| 19 | SCREEN CAPTURE: | 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 its identity. The httpbin.org doesn't do that, 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 – the web site will allow anyone to connect. Later on, when we talk about connecting to a Cloud provider like 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 a secure TLS socket to the server.  Everything else – building the request, sending it, and receiving data back from the server – is exactly the same. |
| 20 | SCREEN CAPTURE: | Now when I run the project, you will see that instead of connecting to <http://httpbin.org>, it connects to <https://httpbin.org> but otherwise I get the same data back. In this case, I'm sure that the site I connected to is really who they claim to be, and the data was sent encrypted so no one could see it during transmission. |
| 21 |  | 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 check them out. |
|  | TEXT ON SCREEN:  Cypress Developer Community  community.cypress.com  Show video of email and twitter windows. | As always, you can post your comments and questions in our Wifi developer community or you are welcome to email me at alan\_hawse@cypress.com or tweet me @askioexpert with your comments, suggestions, criticisms and questions. Thank You! |

TEXT1:

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

TEXT2:

Free up the space from the request http\_request\_deinit

Disconnect the Client from the Server http\_client\_disconnect

De-initialize the Client http\_client\_deinit