Websocket Client

# Introduction

This document provides information on using the WebSocket client APIs provided by the WebSocket module under components. This application demonstrates using these APIs to connect to WebSocket servers in secured (Secured WebSocket) and non-secured way.

The document also discusses ways to execute WebSocket GET and POST.

# Features and Limitations

The WebSocket implementation supports:

1. Secured and non-secured WebSocket client APIs
2. The server address can be either IP address or the host name (Domain Name)
3. APIs to send Text and Binary data
4. APIs for sending Close, Ping and Pong control packets

# WebSocket/S APIs

## WebSocket Open:

This is performed using the following API:

|  |
| --- |
| websock\_handle\_t  websock\_open(websock\_config\_t \*ws\_cfg) |

This API connects to the remote WebSocket server. The configuration needed for the connection is passed using the following data structure:

|  |
| --- |
| typedef struct {  char \*hostname;/\*\*<host name or the ip address of the server\*/  int port;/\*\*<port\*/  char \*uri;/\*\*<Websocket uri/path to connect to\*/  int secured; /\*secured websocket\*/  ssl\_wrap\_cfg\_t ssl\_config;  int time\_out;/\*\*<tcp connection timeout\*/  /\*\*<The headers to be set in the WebSocket Handshake request.  The format shall be "key:val". Use this to pass headers that are not  set implicitly. Refer websock\_open() for the list of headers set  implicitly\*/  websoc\_hndshk\_hdr\_t hndshk\_hdrs[WEBSOCK\_MAX\_HNDSHK\_HDRS];  int num\_hndshk\_hdrs;/\*\* Number of headers provided through,  hndshk\_hdrs[WEBSOCK\_MAX\_HNDSHK\_HDRS]\*/  } websock\_config\_t; |

**Note**: If a URL is available in the form: ws://<domain name>:port/<path>, use http\_client\_url\_to\_host() API to get the hostname port and the path from the URL.

The |Host| |Upgrade|, |Connection|, |Sec -Websocket-key| and |Sec-websocket-Version| headers are implicitly set by the websock\_Open() during the handshake. If any more headers need to be sent during handshake, it needs to be passed using hndshk\_hdrs[WEBSOCK\_MAX\_HNDSHK\_HDRS] in websock\_config\_t .

## WebSocket Send

Once the connection is established using websock\_open(), data can be sent to the server using the following APIs:

|  |
| --- |
| int  websock\_send\_text(websock\_handle\_t handle, char \*payload, int len) |

|  |
| --- |
| int  websock\_send\_binary(websock\_handle\_t handle, char \*payload, int len) |

websock\_send\_text is used for sending the Text frames (Opcode = 1). websock\_send\_binary is used for sending Binary frames (Opcode = 2).

## WebSocket Close:

A connection opened using websock\_open() can be closed using the following API:

|  |
| --- |
| void  websock\_close(websock\_handle\_t h); |

## PING and PONG

WebSocket provides Ping and Pong control packets to check and maintain the connection. These packets can be sent using the following APIs:

|  |
| --- |
| int  websock\_send\_ping(websock\_handle\_t handle, char \*payload, int len); |

|  |
| --- |
| int  websock\_send\_pong(websock\_handle\_t handle, char \*payload, int len); |

# Code Walkthrough

## Application Flow

WebSocket is a communication protocol providing full duplex, asynchronous communication between the connected endpoints.

Following are the steps:

1. Connect to Wi-Fi Network.
2. Connect to WebSocket server.
3. Send message to server.
4. Receive the message.

## Sample Code Walkthrough

A user-defined data struct is created to store the data of the Websocket:

|  |
| --- |
| #define APP\_NAME "WebSocket application"  #define APP\_VERSION "2.0"  OS\_APPINFO {.stack\_size = 4096};  #define INPUT\_PARAMETER\_HOST "host"  #define INPUT\_PARAMETER\_URL "url"  #define INPUT\_PARAMETER\_PATH "path"  #define INPUT\_PARAMETER\_PORT "port"  #define INPUT\_PARAMETER\_SECURED "secured"  #define INPUT\_PARAMETER\_METHOD "method"  #define INPUT\_PARAMETER\_CA "ca\_cert"  #define INPUT\_PARAMETER\_POST\_LEN "post\_len"  #define INPUT\_PARAMETER\_CLIENT\_CERT "client\_cert"  #define INPUT\_PARAMETER\_CLIENT\_KEY "client\_key"  #define INPUT\_PARAMETER\_TEST\_ITER "test\_iterations"  #define INPUT\_PARAMETER\_USE\_CA\_BUNDLE "use\_ca\_bundle"  #define NULL\_STR ""  struct param\_t {  const char \*url;  const char \*host;  const char \*path;  const char \*port;  const char \*secured;  const char \*ca\_cert;  const char \*client\_cert;  const char \*client\_key;  const char \*test\_iterations;  const char \*use\_ca\_bundle;  };  /\*CA certificate bundle\*/  extern uint8\_t ca\_bundle\_start[] asm("\_binary\_ca\_bundle\_start");  extern uint8\_t ca\_bundle\_end[] asm("\_binary\_ca\_bundle\_end");  struct param\_t param;  char default\_port[8];  char default\_secured[8];  struct os\_semaphore app\_wcm\_lock;  int wcm\_connect\_success = 0;  static char host[128];  static char path[128]; |

Following boot arguments are passed in this application:

1. URL, port, secured, ca\_cert, method, post\_len, client\_cert, client\_key value and ca\_bundle.
2. SD service type and proto, and action.

|  |
| --- |
| static int  parse\_boot\_args(void)  {  const char \*np\_path;  int ret = 0;  sprintf(default\_port, "%d", 80);  sprintf(default\_secured, "%d", 0);  param.url = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_URL);  param.host = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_HOST);  param.path = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_PATH);  param.port = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_PORT);  param.secured = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_SECURED);  param.ca\_cert = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_CA);  param.client\_cert = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_CLIENT\_CERT);  param.client\_key = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_CLIENT\_KEY);  param.test\_iterations = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_TEST\_ITER);  param.use\_ca\_bundle = os\_get\_boot\_arg\_str(INPUT\_PARAMETER\_USE\_CA\_BUNDLE); |

To connect to a Wi-Fi network, wcm\_create()API from the Wi-Fi Connection Manager is used. Initially, the Wi-Fi network interface is created using wcm\_create().

|  |
| --- |
| h = wcm\_create(NULL); |

wifi\_connect\_to\_network()API, from components library, connects to the Wi-Fi network using the AP credentials provided.

|  |
| --- |
| rval = wifi\_connect\_to\_network(&h, WCM\_CONN\_WAIT\_INFINITE, &wcm\_connect\_success);  if(rval < 0) {  os\_printf("\nError: Unable to connect to network\n");  return 0;  } |

Here, the data structure websoc\_config\_t is used to pass the parameters while opening a WebSocket connection with the remote server using the websock\_open API such as URL, port, secured, ca\_cert, method, post\_len, client\_cert, client\_key value and ca\_bundle.

|  |
| --- |
| websock\_config\_t cfg = {0};  websock\_handle\_t ws\_handle;  memset(&cfg, 0, sizeof(websock\_config\_t));  path[0]= '\0';  if(param.url){  os\_printf("\n[APP]URL = %s", param.url);  rval = http\_client\_url\_to\_host(param.url, host, sizeof(host), path,  sizeof(path), &cfg.port);  if(rval < 0){  os\_printf("\n[APP]URL is not proper");  os\_printf("\n\texample URLs:");  os\_printf("\n\t\twss://echo.websocket.org/");  os\_printf("\n\t\tws://192.168.1.155:8000/");  }  cfg.hostname = host;  cfg.uri = path;  os\_printf("\nuri/path = %s", cfg.uri);  }else{  cfg.hostname = (char \*)param.host;  }  if(param.port){/\*If specified explicietly, overide the port specified in URL\*/  cfg.port = atoi(param.port);  }  cfg.secured = atoi(param.secured);  if(cfg.secured) {  if(cfg.secured == 1){  cfg.ssl\_config.auth\_mode = SSL\_WRAP\_VERIFY\_NONE;  }else{  cfg.ssl\_config.auth\_mode = SSL\_WRAP\_VERIFY\_REQUIRED;  if(!param.use\_ca\_bundle || !atoi(param.use\_ca\_bundle)){  cfg.ssl\_config.ca\_cert.buf = utils\_file\_get(param.ca\_cert,  &cfg.ssl\_config.ca\_cert.len);  if(NULL == cfg.ssl\_config.ca\_cert.buf){  os\_printf("[APP]Error: No CA certificate found. Required");  goto exit;  }  }else if(atoi(param.use\_ca\_bundle)){  /\*If use\_ca\_bundle is set, initialise CA bundle\*/  os\_printf("[APP]Initializing the ca bundle");  ssl\_wrap\_crt\_bundle\_init((const char \*)ca\_bundle\_start);  }  }  if(param.client\_cert && strlen(param.client\_cert)){  cfg.ssl\_config.client\_cert.buf = utils\_file\_get(param.client\_cert,  &cfg.ssl\_config.client\_cert.len);  if(NULL == cfg.ssl\_config.client\_cert.buf){  os\_printf("Error: Could not open client certificate\n");  goto exit;  }  }  if(param.client\_key && strlen(param.client\_key)){  cfg.ssl\_config.client\_key.buf = utils\_file\_get(param.client\_key,  &cfg.ssl\_config.client\_key.len);  if(NULL == cfg.ssl\_config.client\_key.buf){  os\_printf("Error: Could not open client key\n");  goto exit;  }  }  cfg.secured = 1;  }  if(NULL != param.test\_iterations){  test\_iterations = atoi(param.test\_iterations);  if(0 == test\_iterations)test\_iterations = 1;  }  os\_printf("\n\*\* Test Iterations = %d \*\*\n", test\_iterations); |

This function opens a TCP socket to the server and performs the initial HTTP-based handshake to upgrade to the WebSocket protocol. Both secure (SSL) and non-secure WebSocket connections are supported.

Function websock\_send\_text is used to send “text” data over WebSocket connection.

|  |
| --- |
| websock\_send\_text(ws\_handle,APP\_MESSAGE, strlen(APP\_MESSAGE)); |

The data structure websock\_msg\_hdr\_t is used to pass information about the data received from the server.

Function websock\_recv receives WebSocket messages. This internally handles WebSocket close and ping messages.

|  |
| --- |
| /\*Receive message\*/  char recv\_buf[128];  websock\_msg\_hdr\_t msg\_hdr;  int recv\_len = 128;  websock\_recv(ws\_handle, &msg\_hdr,recv\_buf,&recv\_len,10);  recv\_buf[recv\_len] = 0;  os\_printf("\n[APP] Received Message = %s", recv\_buf);  exit:  os\_printf("\n\n[APP]------ Program Exit-------------\n\n");  return 0;  } |

# Running the Application

## WebSocket Server Setup

There is no specific server available in public domain for free testing of the websockets. A local server is required to test the same. One of the simple local server set-ups can be found at <https://github.com/dpallot/simple-websocket-server>.

This is a Python based server with a Readme for setup instructions.

## Programming Talaria TWO board with ELF

Program websock\_client.elf(sdk\_x.y\examples\websocket\bin) using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the websock\_client.elf by clicking on Select ELF File.
   3. Boot arguments: Pass the following boot arguments:

|  |
| --- |
| url=ws://192.168.1.111/,port=8000,secured=0 |

where,

* + 1. url: URL of the server to connect to
       1. *ws://<domain name OR ip address>:<port>/<path>* **or**
       2. *ws://<domain name OR ip address>/<path>*
    2. port: Server port. Default - 80 in case of non-secured (WebSocket), 443 in case of secured (Secured WebSocket)
    3. secured:

0 - WebSocket,

1 - Secured WebSocket without server verification

2 - Secured WebSocket with server verification

* + 1. ca\_cert: Certificate path in root FS in case of Secured WebSocket
  1. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

## Expected Output

|  |
| --- |
| ---------------PROG Flash: Start Time 05 Apr 2023 12:16:20 PM ------------  UART:SNWWWWAE  Build $Id: git-f1a4f00fb $  hio.baudrate=921600  flash: Gordon ready!  [9.547,958] partitions mounted  UART:SNWWWWAE  Build $Id: git-f1a4f00fb $  hio.baudrate=921600  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWWWAEBuild $Id: git-f1a4f00fb $  url=ws://192.168.0.166/ port=8000 secured=0 np\_conf\_path=/data/nprofile.json ssid=tplink\_conf passphrase=InnoQA2023$  $App:git-6519235e  SDK Ver: sdk\_2.6.3\_alpha  WebSocket Client Demo App  Application Information:  ------------------------  Name : WebSocket application  Version : 2.0  Build Date : Apr 3 2023  Build Time : 20:57:22  Heap Available: 331 KB (339096 Bytes)  [APP]Bootparams :  --------------------  url=ws://192.168.0.166/  host= <null>  port=8000  path= <null>  secured= 0  ca\_cert=<null>  test\_iterations = <null>  use\_ca\_bundle = <null>  [APP]Bootparams end here....  [APP]Bootparams check done....ret = 0  addr e0:69:3a:00:0a:6a  Connecting to added network : tplink\_conf  [1.226,105] CONNECT:60:32:b1:3a:83:ee Channel:1 rssi:-55 dBm  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_LINK\_UP  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_ADDRESS  [2.324,746] MYIP 192.168.0.155  [2.324,793] IPv6 [fe80::e269:3aff:fe00:a6a]-link  wcm\_notify\_cb to App Layer - WCM\_NOTIFY\_MSG\_CONNECTED  Connected to added network : tplink\_conf  [APP]URL = ws://192.168.0.166/  uri/path = /  \*\* Test Iterations = 1 \*\*  [APP]Calling websock\_open(). heap size = 264272  [APP] Received Message = Hello World  [APP]------ Program Exit------------- |