Chapter 2: Networking with ModusToolbox™ for Wi-Fi

At the end of this chapter, you will understand the fundamentals of operating as a Wi-Fi Station (STA) and connecting to a Wi-Fi Access Point (AP). You will have an introduction to the TCP/IP Networking stack, and you will have a basic understanding of the first three layers of the Open Systems Interconnection (OSI) reference model for a network stack (i.e. physical, datalink and network layers). You will also have a basic understanding of the Wi-Fi datalink layer which handles connections and encryption. Finally, you will understand some of the basics of IP networking (addresses, netmasks).

Most importantly, you will be able to use ModusToolbox™ for Wi-Fi to connect your IoT device to a Wi-Fi Network.

Table of contents

Document conventions

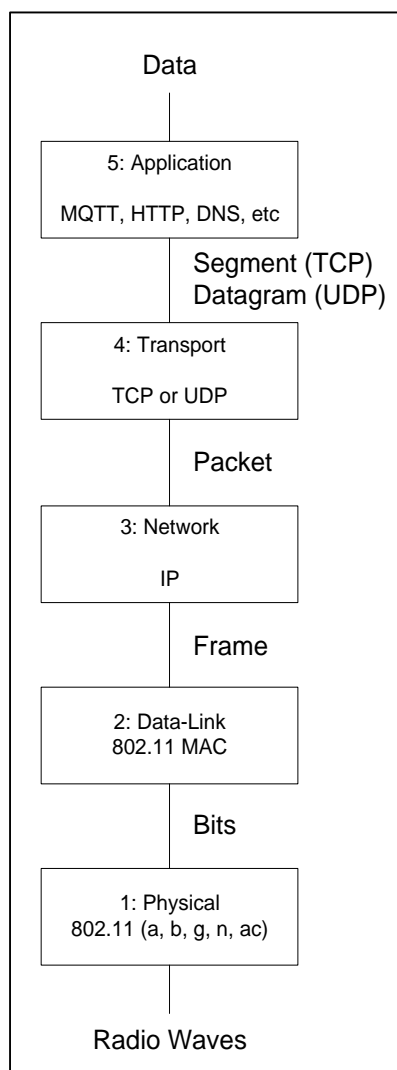| Convention	Usage	Example
Courier New	Displays code and text commands	CY_ISR_PROTO(MyISR); make build
Italics	Displays file names and paths	*sourcefile.hex*
[**bracketed, bold**]	Displays keyboard commands in procedures	[**Enter**] or [**Ctrl**] [**C**]
Menu > Selection	Represents menu paths	**File > New Project > Clone**
Bold	Displays GUI commands, menu paths and selections, and icon names in procedures	Click the **Debugger** icon, and then click **Next**.

2.1 TCP/IP networking stack

TCP/IP stands for Transmission Control Protocol/Internet Protocol. Almost all complicated systems manage the overall complexity by dividing the system into layers. The "Network Stack" or more accurately, the "TCP/IP Network Stack" is exactly that: a hierarchical system for reliably communicating over multiple networking mediums (Wi-Fi, Ethernet, etc.). Each layer isolates the user of that layer from the complexity of the layer below it, and simplifies the communication for the layer above it. You might hear about the OSI Network Model which is another, similar way to describe networking layers; however, it is easier to envision IP networks using the TCP/IP model.

Each layer takes the input of the layer above it and then embeds that information into one or more of the Protocol Data Units (PDUs) of that layer. A PDU is the atomic unit of data for a given layer: e.g. the Datalink Layer takes an IP packet and divides it up into 1 or more Wi-Fi Data Link Layer Frames. The physical layer takes Datalink Layer Frames and divides them up into bits.

The layers of the stack are:

Layer	Protocol	Protocol Data Unit	Comment
Layer 5 Application	DNS, DHCP, MQTT, HTTP, etc.	Data	The application layer is the protocol used to do something useful in the device e.g. HTTP (get or put data), DNS (find an IP address from a name), MQTT (publish or subscribe), etc.
Layer 4 Transport	TCP UDP	(TCP) Segments (UDP) Datagram	Reliable, ordered, error checked stream of bytes – think of it as a pipe between computers or as a phone call. An unreliable connectionless datagram flow– think of it like dropping an envelope in the mail to the post office, you don't know it is received until the other side confirms and delivery order is not guaranteed.
Layer 3 Network	IP	Packets	An IP network can send and receive IP packets with source and destination IP addresses to anywhere on the Internet. The IP layer deals with addressing and routing of packets.
Layer 2 Data-Link	802.11 MAC	Frame	A frame is the atomic unit of transmission in the network. Each frame is no more than one Maximum Transmission Unit (MTU) of data which is specific to each data-link layer. All the data from the layers above are broken into frames by the data link layer. Converts bits into unencrypted frames. This layer only communicates on the Local Area Network. A frame contains the MAC address for the source and destination which are mapped to/from the IP addresses.
Layer 1 Physical	802.11(a,b,g,n,ac)	Bits	Sends and receives streams of bits over the Wi-Fi Radio; handles carrier access and arbitration for the network medium.

In graphical form:

```
         Data
          |
  +-----------------+
  |  5: Application  |
  |                  |
  | MQTT, HTTP, DNS, etc |
  +-----------------+
         Segment (TCP)
         Datagram (UDP)
  +-----------------+
  |  4: Transport    |
  |                  |
  |  TCP or UDP      |
  +-----------------+
           Packet
  +-----------------+
  |  3: Network      |
  |                  |
  |      IP          |
  +-----------------+
           Frame
  +-----------------+
  |  2: Data-Link    |
  |  802.11 MAC      |
  +-----------------+
            Bits
  +-----------------+
  |  1: Physical     |
  | 802.11 (a, b, g, n, ac) |
  +-----------------+
          |
     Radio Waves
```

2.2 (Physical/Datalink) Wi-Fi basics

There are two ends of a Wi-Fi network: The Station (i.e. the IoT device) and the Access Point (i.e. the wireless router). In order for a Station to connect to a Wi-Fi Access Point, it must know the following information: **SSID, Encryption Scheme**, and **Password** (if required). The Wi-Fi chip will take care of selecting the proper band and channel. All Datalink Frames are labeled with the source and destination **MAC Addresses**.

2.2.1 SSID (the name of the wireless network)

SSID (https://en.wikipedia.org/wiki/Service_set_(802.11_network) stands for Service Set Identifier. The SSID is the network name and is composed of 1-32 bytes (a.k.a. octets - which is the same as an 8-bit byte - but for some reason which is lost in the mists of history, networking guys always call them octets). The name does not have to be human readable (e.g. ASCII) but because it is unencoded bytes, it is effectively case sensitive (be careful).

2.2.2 Band (either 2.4, 5 or 6 GHz)

Wi-Fi radios encode 1's and 0's with one of a number of different modulation schemes depending on the type of Wi-Fi network (a,b,g,n,ac,ax) and operating mode. The types of encoding are transparent to your IoT application since the chip, radio, and firmware will virtualize this for you. The data is then transmitted into the 2.4, 5 or 6 GHz band. The 5 GHz and 6 GHz bands have higher throughput and less latency, but less range, while the opposite is true for 2.4 GHz band. For example, 2.4 GHz can deliver up to 100 Mbps, 5 GHz can deliver up 1 Gpbs, and 6 GHz can deliver up to 2 Gbps.

2.2.3 Channel number

The available channels (https://en.wikipedia.org/wiki/List_of_WLAN_channels) are band (2.4, 5 or 6 GHz) and geographically (location) specific. Additionally, the FCC regulates which channels and bands may be used for different operating regions of the world. At the Wi-Fi layer, this is configured via a country-code setting which maps to a set of available channels for that region. 2.4 GHz is pretty simple, there are channels 1-14 with 1-11 available all over the world. 5 GHz and 6 GHz are region specific and regulatory bodies (e.g. the FCC) will mandate which channels you may use depending on the region.

However, from the station point of view (and therefore for this class) none of that matters since when you try to join an SSID the ModusToolbox™ for Wi-Fi run-time software will scan all channels looking for the correct SSID. It is up to the access point to use the correct bands for the region.

2.2.4 Encryption (Open, WEP, WPA, WPA2, WPA3)

In order to provide security for Wi-Fi networks it is common to use data link layer encryption (https://en.wikipedia.org/wiki/Wireless_security). The types of network encryption are Open (i.e. no security), Wired Equivalent Privacy (WEP) which is not completely secure (but may be OK for some type of limited legacy applications), Wi-Fi Protected Access (WPA), WPA2 and WPA3. From here on we will just call it WPA, but we generally mean WPA2 or WPA3. There are two versions of WPA: one called "Personal" or "Pre-Shared Key" (PSK) and one called "Enterprise."

WEP and WPA PSK both use a password—called a key—to encrypt the data. The WEP encryption scheme is not recommended as it is very easy to compromise (e.g. using tools like Wireshark and AirSnort). The PSK key scheme of WPA is very secure as it uses AES (Advanced Encryption Standard). However, sharing keys is a painful, unsecure process because it means that everyone has the same key. To solve the key distribution

problem, most enterprise networking solutions use WPA Enterprise which requires use of a [RADIUS](#) server to handle authentication of each station individually.

Enterprise security is an oncoming crisis for the IoT market and is a differentiating feature of ModusToolbox™ for Wi-Fi – when you use it, this is all taken care of for you – auto-magically!

2.2.5   Media Access Control (MAC) address

The Wi-Fi MAC address (https://en.wikipedia.org/wiki/MAC_address) is a 48-bit unique number comprised of an OUI (Organizationally Unique ID) and a station ID. The first three bytes of the MAC address are the OUI field which is assigned by IEEE to be unique per manufacturer (e.g. Infineon). For the datalink layer to send a frame it must address the frame with a source and destination MAC address. Other devices on the network will only pass frames into the higher levels of the stack that are addressed to them. Remember that the Datalink Layer does not know anything about the higher layers (e.g. IP). Finally, the most significant bit of the most significant byte (e.g. bit 47) specifies a multicast (Group) address and the special address of all 1's (e.g. ff:ff:ff:ff:ff:ff) is a broadcast address (send to everyone).

The datalink layer needs to be able to figure out the MAC address of a given IP address in order to send data to that IP address out on the Wi-Fi network. To figure out this mapping there is a protocol called Address Resolution Protocol.

2.2.6   Address Resolution Protocol (ARP)

An IP address can either be IPV4 or IPV6. We will focus on IPV4 addresses which are a 32-bit number that is generally expressed as four hex-bytes separated by periods. For example, 192.168.15.7 is a valid IPV4 address.

Inside of every device there is an ARP (https://en.wikipedia.org/wiki/Address_Resolution_Protocol) table that has a map of MAC addresses to IP addresses. To discover the MAC address of an IP address, an "ARP request" is broadcast to the network. All devices attached to a network listen for ARP requests. If you hear an ARP request with your IP address in it, you respond with your MAC address. From that point forward both sides add that information to their ARP table (and in fact if you hear others ARPing you can update your table as well).

The brilliant part of this scheme is that if you ARP for an IP address that is not on your local network, the router will respond with its MAC address. This indicates that any IP address meant for the wide area network (WAN), instead of the local area network (LAN) will be sent to your router. The router then handles returning it to the ultimate destination, often by going through multiple routers along the way. Similarly, on the way back, the router ensures any packet sent arrives to the correct device on its LAN.

2.3 IP Networking and Network Address Translation
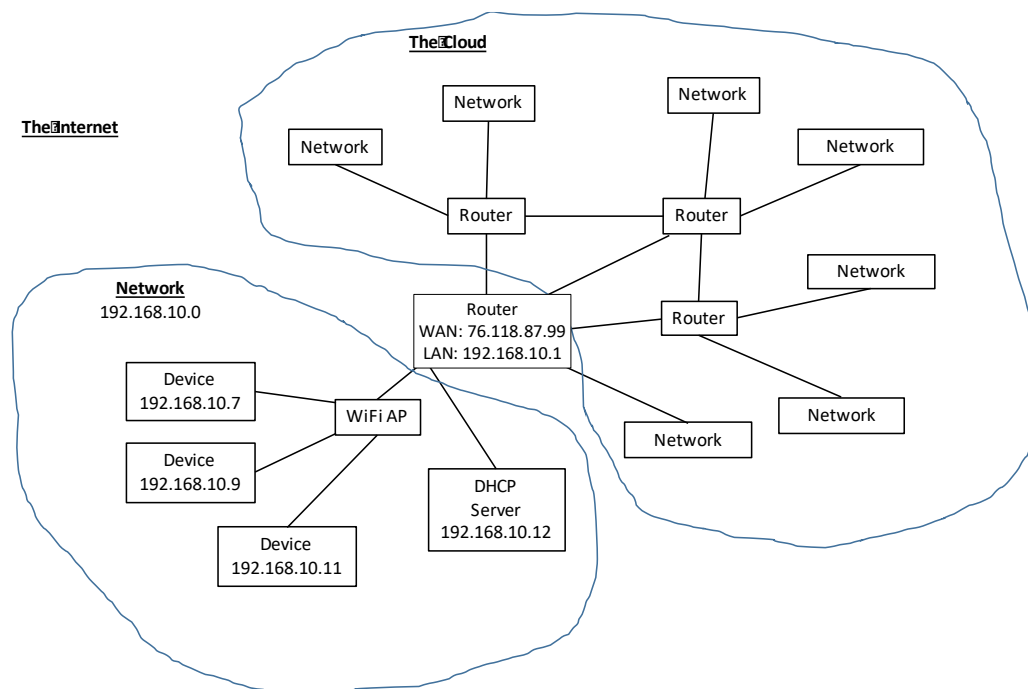
The Internet is a mesh of interconnected IP networks. The Cloud is all of the Internet that is accessible by your network but may also mean servers that are attached to a network somewhere on the Internet.

IPV4 addresses are divided into two parts: the network address (which is the first x number of bits) and the client address which is the last 32-x bits. The netmask defines the split of network/client. E.g. the netmask for 192.168.10.* is 255.255.255.0.

An IP Network (sometimes called an IP sub-network) is the collection of devices that all share the same network address e.g. all of the devices on 192.168.15.* (netmask 255.255.255.0) are all part of the same IP sub-network.

All devices on the Internet have a legal **IP address** and belong to an (IP) Network that is defined by a Netmask. Routers are devices that connect IP networks by taking IP packets from one network and forwarding them along to the correct next network. This is a complicated task that involves network address translation (NAT). Essentially, your local area network (LAN) has a set of private IP addresses. However, since everyone's LAN uses the same set of addresses, the full explanation is more complex.

Your router, in turn, has an IP address that it presents to the outside world. That address is provided by your internet service provider (ISP) and will be unique on that level of the network. Therefore, any packet meant for any device on your LAN is passed to your router by your ISP. Your router uses NAT to translate WAN IP address to the correct LAN IP address. Conversely, your router converts your IP address into a WAN IP address that makes sense to your ISP. This process continues up the chain – your ISP's router will connect to other routers that each perform NAT until the packet eventually reaches its destination.

Most commonly, IP addresses for IoT type devices are assigned dynamically by a Dynamic Host Control Protocol (DHCP) server. To dynamically assign a DHCP address you first send a Layer-2 broadcast datagram requesting an IP address (DHREQUEST). When a DHCP server hears the request, it responds with the required information. DHCP is integrated into ModusToolbox™ for Wi-Fi, it handles this exchange of information for you automatically when enabled.

2.4   Wi-Fi Connection Manager (WCM)

The Wi-Fi Connection Manager is a set of API's that are useful for establishing and monitoring Wi-Fi connections on Infineon platforms. The first step is to initialize the WCM:

```
cy_wcm_init(cy_wcm_config_t* config)
```

As you can see, it takes a configuration structure as an argument. The structure looks like this:

```
typedef struct
{
    cy_wcm_interface_t interface;  /**< Interface type. */
} cy_wcm_config_t;
```

The device can operate in three modes, Client mode (STA), Software Enabled Access Point mode (softAP), and concurrent Client/Access Point mode. This is set via the `interface` member of the structure. Possible choices are:

```
typedef enum
{
    CY_WCM_INTERFACE_TYPE_STA = 0,    /**< STA or Client interface.                                 */
    CY_WCM_INTERFACE_TYPE_AP,         /**< SoftAP interface. \note Not supported, will be added in future.    */
    CY_WCM_INTERFACE_TYPE_AP_STA      /**< Concurrent AP + STA mode. \note Not supported, will be added in future. */
} cy_wcm_interface_t;
```

If you configure the WCM as a station (`CY_WCM_INTERFACE_TYPE_STA`), you can then connect to an access point by calling the connect function:

```
cy_wcm_connect_ap(cy_wcm_connect_params_t* connect_params, cy_wcm_ip_address_t* ip_addr)
```

The first argument is a struct of type `cy_wcm_connect_params_t` which in turn holds several other structs containing all of the relevant data for connecting to Wi-Fi.

```
/**
 * Structure used to pass the Wi-Fi connection parameter information to \ref cy_wcm_connect_ap.
 *
 */
typedef struct
{
    cy_wcm_ap_credentials_t  ap_credentials;       /**< Access point credentials. */
    cy_wcm_mac_t             BSSID;                 /**< MAC address of Access Point (optional). */
    cy_wcm_ip_setting_t      *static_ip_settings;   /**< Static IP settings of the device (optional). */
    cy_wcm_wifi_band_t       band;                  /**< Radio band to be connected (optional). */
} cy_wcm_connect_params_t;
```

These connection parameters are built during the make process and written into the flash along with your application, but they can be modified (and written) on the fly by your application.

Before you can connect to Wi-Fi you need to populate some of the parameters with the appropriate data. To preconfigure the Wi-Fi section of connection parameters you will typically create the following `#defines` in a file called *wifi_config.h* or some other equivalent file.

```
#ifndef WIFI_CONFIG_H_
#define WIFI_CONFIG_H_

#include "cy_wcm.h"

/*******************************************************************
* Macros
********************************************************************/
/* SSID of the Wi-Fi Access Point to which the MQTT client connects. */
#define WIFI_SSID                       "MY_WIFI_SSID"

/* Passkey of the above mentioned Wi-Fi SSID. */
#define WIFI_PASSWORD                   "MY_WIFI_PASSWORD"

/* Security type of the Wi-Fi access point. See 'cy_wcm_security_t' structure
 * in "cy_wcm.h" for more details.
 */
#define WIFI_SECURITY                   CY_WCM_SECURITY_WPA2_AES_PSK

/* Maximum Wi-Fi re-connection limit. */
#define MAX_WIFI_CONN_RETRIES           (10u)

/* Wi-Fi re-connection time interval in milliseconds. */
#define WIFI_CONN_RETRY_INTERVAL_MS     (2000)

#endif /* WIFI_CONFIG_H_ */
```

To find the definition (or possible definitions) of the `#defines` you can highlight, right click, and select **Open declaration**. For example, if you open the declaration of CY_WCM_SECURITY_WPA3_WPA2_PSK:

```
typedef enum
{
    CY_WCM_SECURITY_OPEN             = 0,                                              /**< Open security.
    CY_WCM_SECURITY_WEP_PSK          = WEP_ENABLED,                                    /**< WEP PSK security with open authentication.
    CY_WCM_SECURITY_WEP_SHARED       = ( WEP_ENABLED   | SHARED_ENABLED ),             /**< WEP PSK security with shared authentication.
    CY_WCM_SECURITY_WPA_TKIP_PSK     = ( WPA_SECURITY  | TKIP_ENABLED ),               /**< WPA PSK security with TKIP.
    CY_WCM_SECURITY_WPA_AES_PSK      = ( WPA_SECURITY  | AES_ENABLED ),                /**< WPA PSK security with AES.
    CY_WCM_SECURITY_WPA_MIXED_PSK    = ( WPA_SECURITY  | AES_ENABLED | TKIP_ENABLED ), /**< WPA PSK security with AES and TKIP.
    CY_WCM_SECURITY_WPA2_AES_PSK     = ( WPA2_SECURITY | AES_ENABLED ),                /**< WPA2 PSK security with AES.
    CY_WCM_SECURITY_WPA2_AES_PSK_SHA256 = ( WPA2_SECURITY | WPA2_SHA256_SECURITY | AES_ENABLED ), /**< WPA2 PSK SHA256 Security with AES
    CY_WCM_SECURITY_WPA2_TKIP_PSK    = ( WPA2_SECURITY | TKIP_ENABLED ),               /**< WPA2 PSK security with TKIP.
    CY_WCM_SECURITY_WPA2_MIXED_PSK   = ( WPA2_SECURITY | AES_ENABLED | TKIP_ENABLED ), /**< WPA2 PSK security with AES and TKIP.
    CY_WCM_SECURITY_WPA2_FBT_PSK     = ( WPA2_SECURITY | AES_ENABLED | FBT_ENABLED),   /**< WPA2 FBT PSK security with AES and TKIP.
    CY_WCM_SECURITY_WPA3_SAE         = ( WPA3_SECURITY | AES_ENABLED ),                /**< WPA3 security with AES.
    CY_WCM_SECURITY_WPA2_WPA_AES_PSK = (WPA2_SECURITY | WPA_SECURITY | AES_ENABLED),   /**< WPA2 WPA PSK Security with AES
    CY_WCM_SECURITY_WPA2_WPA_MIXED_PSK = (WPA2_SECURITY | WPA_SECURITY | AES_ENABLED | TKIP_ENABLED), /**< WPA2 WPA PSK Security with AES & TKIP.
    CY_WCM_SECURITY_WPA3_WPA2_PSK    = ( WPA3_SECURITY | WPA2_SECURITY | AES_ENABLED ),/**< WPA3 WPA2 PSK security with AES.
    CY_WCM_SECURITY_WPA_TKIP_ENT     = (ENTERPRISE_ENABLED | WPA_SECURITY | TKIP_ENABLED), /**< WPA Enterprise Security with TKIP.
    CY_WCM_SECURITY_WPA_AES_ENT      = (ENTERPRISE_ENABLED | WPA_SECURITY | AES_ENABLED), /**< WPA Enterprise Security with AES
    CY_WCM_SECURITY_WPA_MIXED_ENT    = (ENTERPRISE_ENABLED | WPA_SECURITY | AES_ENABLED | TKIP_ENABLED), /**< WPA Enterprise Security with AES and TKIP.
    CY_WCM_SECURITY_WPA2_TKIP_ENT    = (ENTERPRISE_ENABLED | WPA2_SECURITY | TKIP_ENABLED), /**< WPA2 Enterprise Security with TKIP.
    CY_WCM_SECURITY_WPA2_AES_ENT     = (ENTERPRISE_ENABLED | WPA2_SECURITY | AES_ENABLED), /**< WPA2 Enterprise Security with AES.
    CY_WCM_SECURITY_WPA2_MIXED_ENT   = (ENTERPRISE_ENABLED | WPA2_SECURITY | AES_ENABLED | TKIP_ENABLED), /**< WPA2 Enterprise Security with AES and TKIP.
    CY_WCM_SECURITY_WPA2_FBT_ENT     = (ENTERPRISE_ENABLED | WPA2_SECURITY | AES_ENABLED | FBT_ENABLED), /**< WPA2 Enterprise Security with AES and FBT.

    CY_WCM_SECURITY_IBSS_OPEN        = ( IBSS_ENABLED ),                               /**< Open security on IBSS ad hoc network.
    CY_WCM_SECURITY_WPS_SECURE       = ( WPS_ENABLED | AES_ENABLED),                   /**< WPS with AES security.

    CY_WCM_SECURITY_UNKNOWN          = -1,                                             /**< Returned by \ref cy_wcm_scan_result_callback

    CY_WCM_SECURITY_FORCE_32_BIT     = 0x7fffffff                                      /**< Exists only to force whd_security_t type to
} cy_wcm_security_t;
```

You can see from the figure above that ModusToolbox™ for Wi-Fi supports any type of Wi-Fi security you may want. We will usually select CY_WCM_SECURITY_WPA3_WPA2_PSK which will attempt to connect using WPA3 but will fall back on WPA2 if the router you are connecting to doesn't support WPA3.

Once you have created the *wifi_config.h* file you can populate the relevant parts of the `cy_wcm_connect_params_t` struct with the following code:

```
/* Configure the connection parameters for the Wi-Fi interface. */
memset(&connect_param, 0, sizeof(cy_wcm_connect_params_t));
memcpy(connect_param.ap_credentials.SSID, WIFI_SSID, sizeof(WIFI_SSID));
memcpy(connect_param.ap_credentials.password, WIFI_PASSWORD, sizeof(WIFI_PASSWORD));
connect_param.ap_credentials.security = WIFI_SECURITY;
```

The second argument to the `cy_wcm_connect_ap` function is a pointer to a variable of type `cy_wcm_ip_address_t` used to hold the IP address of your device once the connection has been made.
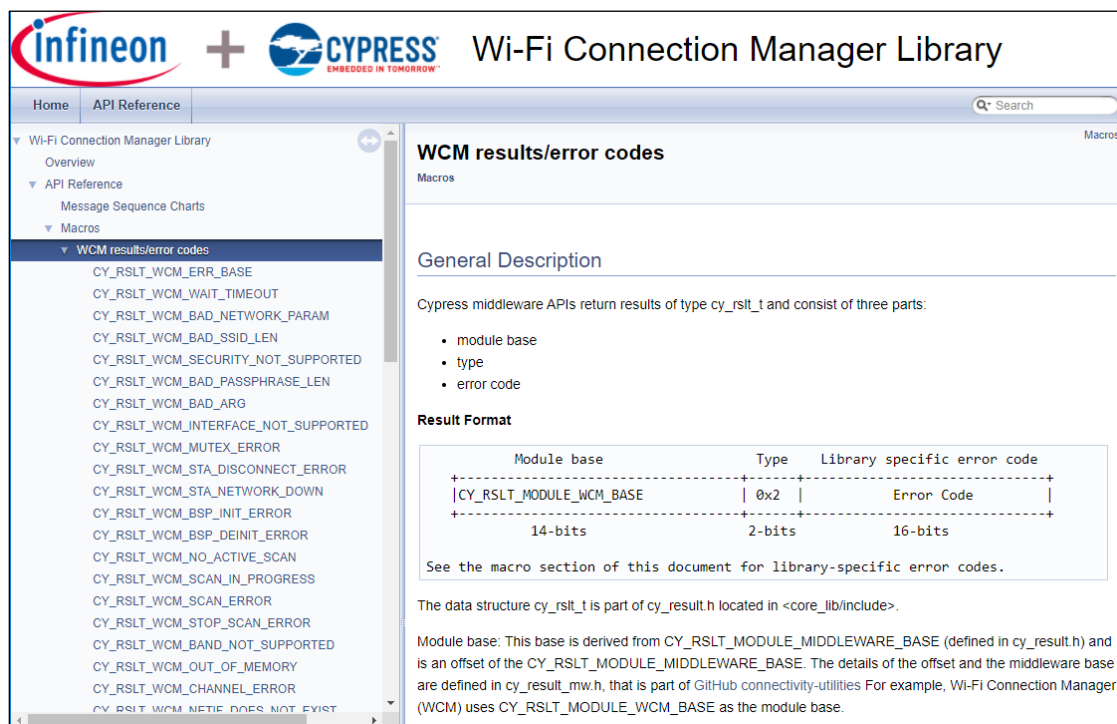
2.5 Task Priorities

The Wi-Fi functions that run as RTOS tasks use priorities as low as `CY_RTOS_PRIORITY_NORMAL`. Therefore, you should not use priorities higher than `CY_RTOS_PRIORITY_NORMAL` for your tasks since they may prevent Wi-Fi operations from working correctly.

Note: `CY_RTOS_PRIORITY_NORMAL has a value of 3 for FreeRTOS, so any FreeRTOS tasks created by the application should use a value of 3 or lower when Wi-Fi operation is active.`

2.6 CY_RSLT_T

Throughout ModusToolbox™ APIs, a value from many of the functions is returned telling you what happened. The return value is of the type `cy_rslt_t` which is a giant enumeration. Some values that are returned include `CY_RSLT_SUCCESS`, `CY_RSLT_PENDING` and `CY_RSLT_ERROR`. The `cy_rslt_t` type is a structured bitfield which encodes information about result type, the originating module, and a code for the specific error (or warning etc). In order to extract these individual fields from a `cy_rslt_t` value, the utility macros `CY_RSLT_GET_TYPE`, `CY_RSLT_GET_MODULE`, and `CY_RSLT_GET_CODE` are provided in the file *cy_result.h*. To see all of the potential WCM `cy_rslt_t` types, you can look in the Wi-Fi Connection Manager Library documentation under **API Reference > Macros > WCM results/error codes**.

2.7   Documentation

The relevant documentation for the networking management functions are in the Wi-Fi Connection Manager Library documentation under **API Reference > Functions**.

2.8   Onboarding

Onboarding is the process used to get an IoT device connected to the network. That is, it needs to know the Wi-Fi SSID to connect to, the password to use, the encryption keys to use, etc. There are many possible strategies for solving this problem including:

- Include the Cirrent ZipKey agent in your device
  - The agent uses a ZipKey hotspot (created by internet service providers such as Xfinity) to connect to the Cirrent Cloud and then automatically configures your IoT device to use your Wi-Fi network. See www.cirrent.com for additional details.
  - The Cirrent cloud also provides IoT network intelligence which allows you to monitor, diagnose, and improve performance of your solutions in the field.
- Start a Wi-Fi Access Point with a web server on the IoT device, then connect to the IoT device from a computer or a cellphone. A web browser on the computer or cellphone is used to configure the IoT device which then restarts in client mode using the stored configuration.
- Connect to the IoT device using Bluetooth® and then use a phone-based App to configure the device's Wi-Fi settings.
- Connect the IoT device to a computer using a USB or Serial connection and then configuring the device's Wi-Fi settings with a computer-based application.
- Preprogram the device with the required information.

ModusToolbox™ for Wi-Fi supports all these methods. In this class, we will mainly use the pre-programmed method in the interest of simplicity and time. Some examples in later chapters use a Wi-Fi Access Point with a web server on the IoT device. The other methods are demonstrated in ModusToolbox™ code examples.

2.9   Multicast DNS

2.9.1   Overview

The Dynamic Name Service (DNS) is how a device finds the IP address for a given network name (such as a web server). Traditionally, a device needs to be configured with a DNS server's address to be provided so that a device knows who to ask to look up IP addresses.

Multicast DNS, or mDNS, is a zero-configuration networking service for resolving hostnames to IP addresses within a local network. mDNS was designed to work as a stand-alone protocol and can provide local hostname to IP address resolution even in the absence of a standard DNS server. mDNS can also work alongside a DNS server without any issues. mDNS works by sending IP Multicast messages. Multicast is a method of sending IP messages to a group of interested receivers via a single transmission. As a result, when an mDNS client wishes to send a message to other mDNS clients, it only has to send one message, but that message will be delivered to every other mDNS client on the same network. mDNS also supports Unicast messaging, but only in specific circumstances.

By default mDNS exclusively resolves hostnames with the ".local" first level domain. (i.e. myComputer.local) As of July 2020, ".local" domains are not available for registration on the internet and are only used by local networks. When an mDNS client needs to resolve a hostname, it multicasts a query message that asks the host with the queried name to identify itself. The host that was just queried then multicasts a response message containing its IP address. Every mDNS device on the same network will receive both the query and the response messages and will update its mDNS caches accordingly.

2.9.2 Message structure

mDNS messages are sent using User Datagram Protocol (UDP) from the UDP port 5353 to the IPv4 address 224.0.0.152 or the IPv6 address FF02::FB. mDNS messages are based on the unicast DNS packet format and only differ slightly from that standard. Both queries and responses are in the same format but contain different information. An mDNS message contains five fields:

Header

Question

Answer

Authority

Additional

The header section details the information contained in the message and consists of the following fields:

Field	Description	Bit Length
ID	Query Identifier	16
QR	Query/Response Bit - Boolean flag indicating whether the message is a query (0) or reply (1)	1
OPCODE	Query Type - Only standard queries are supported over multicast, so this must always be 0	4
AA	Authoritative Answer Bit - Boolean flag indicating whether the message is a response from an authoritative nameserver. Queries must always have this bit set to 0	1
TC	Truncated Bit - In query messages if the TC bit is set it means that additional Known-Answer records may be following shortly. In response messages, the TC bit must be 0	1
RD	Recursion Desired Bit – This should always be 0	1
RA	Recursion Available Bit - This should always be 0	1
Z	Zero Bit - This should always be 0	1
AD	Authentic Data Bit - This should always be 0	1
CD	Checking Disabled Bit - This should always be 0	1
RCODE	Response Code - This should always be 0	1
QDCOUNT	Integer specifying the number of entries in the question section	16
ANCOUNT	Integer specifying the number of resource records in the answer section	16
NSCOUNT	Integer specifying the number of name server resource records in the authority records section	16

Field	Description	Bit Length
ARCOUNT	Integer specifying the number of resource records in the additional records section	16

The Question section contains all the information pertaining to any query any client may have. The question section consists of the following fields:

Field	Description	Bit Length
QNAME	Hostname of the device being queried	Variable
QTYPE	The type of query – This can be any of the defined DNS Record Types	16
UNICAST-RESPONSE	Boolean flag indicating whether a unicast response is desired	1
QCLASS	Class Code	15

The answer, authority, and additional sections all share the same format: a variable number of resource records, where the number of records is specified in the corresponding count field in the header. Each resource record has the following format:

Field	Description	Bit Length
RRNAME	Name of the node to which the record pertains	Variable
RRTYPE	The type of resource record	16
CACHE-FLUSH	Boolean flag indicating whether cached records should be purged or appended to	1
RRCLASS	Resource record class code	15
TTL	Time To Live - Number of seconds that that the resource record should be cached	32
RDLENGTH	Integer length (in bytes) of the RDATA field	16
RDATA	Resource Data; internal layout varies by RRTYPE	Variable

2.9.3　Service Discovery

mDNS is also commonly used for service advertising and discovery. DNS-SD (DNS Service Discovery) is another protocol that specifies how resource records are named and structured to facilitate the discovery of services supported by devices on your local network i.e. printing, file transfer, web pages, or other network services. DNS-SD queries can be sent via multicast, so that every device on the local network will receive the service discovery query. Any devices with services that were queried for can then send a response. The following are examples of a mDNS Service Discovery query and response as captured via Wireshark.

First is an mDNS query asking any devices on the network that support IPP (Internet Printing Protocol) to respond with their hostname.

```
Wireshark · Packet 24 · Wi-Fi

> Frame 24: 75 bytes on wire (600 bits), 75 bytes captured (600 bits) on interface \Device\NPF_{2DE1EC3A-2718-48AB-A656-D899ECD09706}, id 0
> Ethernet II, Src: Tp-LinkT_d4:73:7c (50:3e:aa:d4:73:7c), Dst: IPv4mcast_fb (01:00:5e:00:00:fb)
> Internet Protocol Version 4, Src: 192.168.86.69, Dst: 224.0.0.251
> User Datagram Protocol, Src Port: 5353, Dst Port: 5353
v Multicast Domain Name System (query)
    Transaction ID: 0x0000
    v Flags: 0x0000 Standard query
        0... .... .... .... = Response: Message is a query
        .000 0... .... .... = Opcode: Standard query (0)
        .... ..0. .... .... = Truncated: Message is not truncated
        .... ...0 .... .... = Recursion desired: Don't do query recursively
        .... .... .0.. .... = Z: reserved (0)
        .... .... ...0 .... = Non-authenticated data: Unacceptable
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    v Queries
        v _ipp._tcp.local: type PTR, class IN, "QU" question
            Name: _ipp._tcp.local
            [Name Length: 15]
            [Label Count: 3]
            Type: PTR (domain name PoinTeR) (12)
            .000 0000 0000 0001 = Class: IN (0x0001)
            1... .... .... .... = "QU" question: True
```

Next is a response from a printer on my network to the previous query.

```
Wireshark · Packet 34 · Wi-Fi

> Frame 34: 1408 bytes on wire (11264 bits), 1408 bytes captured (11264 bits) on interface \Device\NPF_{2DE1EC3A-2718-48AB-A656-D899ECD09706}, id 0
> Ethernet II, Src: HewlettP_c5:5c:64 (48:ba:4e:c5:5c:64), Dst: IPv4mcast_fb (01:00:5e:00:00:fb)
> Internet Protocol Version 4, Src: 192.168.86.34, Dst: 224.0.0.251
> User Datagram Protocol, Src Port: 5353, Dst Port: 5353
v Multicast Domain Name System (response)
    Transaction ID: 0x0000
    v Flags: 0x8400 Standard query response, No error
        1... .... .... .... = Response: Message is a response
        .000 0... .... .... = Opcode: Standard query (0)
        .... .1.. .... .... = Authoritative: Server is an authority for domain
        .... ..0. .... .... = Truncated: Message is not truncated
        .... ...0 .... .... = Recursion desired: Don't do query recursively
        .... .... 0... .... = Recursion available: Server can't do recursive queries
        .... .... .0.. .... = Z: reserved (0)
        .... .... ..0. .... = Answer authenticated: Answer/authority portion was not authenticated by the server
        .... .... ...0 .... = Non-authenticated data: Unacceptable
        .... .... .... 0000 = Reply code: No error (0)
    Questions: 0
    Answer RRs: 2
    Authority RRs: 0
    Additional RRs: 9
    v Answers
        > _ipp._tcp.local: type PTR, class IN, HP ENVY 5660 series [C55C64]._ipp._tcp.local
        > _ipps._tcp.local: type PTR, class IN, HP ENVY 5660 series [C55C64]._ipps._tcp.local
    v Additional records
        > HP ENVY 5660 series [C55C64]._ipp._tcp.local: type TXT, class IN, cache flush
        > HP ENVY 5660 series [C55C64]._ipps._tcp.local: type TXT, class IN, cache flush
        > HP48BA4EC55C64.local: type A, class IN, cache flush, addr 192.168.86.34
        > HP48BA4EC55C64.local: type AAAA, class IN, cache flush, addr fe80::4aba:4eff:fec5:5c64
        > HP ENVY 5660 series [C55C64]._ipp._tcp.local: type SRV, class IN, cache flush, priority 0, weight 0, port 631, target HP48BA4EC55C64.local
        > HP ENVY 5660 series [C55C64]._ipps._tcp.local: type SRV, class IN, cache flush, priority 0, weight 0, port 443, target HP48BA4EC55C64.local
        > HP ENVY 5660 series [C55C64]._ipp._tcp.local: type NSEC, class IN, cache flush, next domain name HP ENVY 5660 series [C55C64]._ipp._tcp.local
        > HP ENVY 5660 series [C55C64]._ipps._tcp.local: type NSEC, class IN, cache flush, next domain name HP ENVY 5660 series [C55C64]._ipps._tcp.local
        > HP48BA4EC55C64.local: type NSEC, class IN, cache flush, next domain name HP48BA4EC55C64.local
    [Unsolicited: True]
```

2.9.4   Service Advertising

Whenever a device starts up, wakes from sleep, or has any reason to believe that its network connectivity has changed in some way it must do two things. First it must "probe" for any devices on the local network that may have conflicting resource records with itself. The device sends mDNS queries for all of its resource records, then waits to make sure nothing responds. Once the device has confirmed there are no conflicts between its resource records and the records of other devices on the network it can then "announce" its records to the network. An announcement consists of a mDNS message whose answer section contains all of the resource records the device is claiming. The following are examples of a device probing and announcing as captured via Wireshark.

First is the probing message sent by a printer on my network during its startup.

Second is the announcement message sent by the same printer on my network during its startup.

2.9.5 Using mDNS in ModusToolbox™ for Wi-Fi

2.9.5.1 Responding to mDNS Queries

The lwIP Library includes an "mDNS Responder" that will automatically respond to mDNS queries sent to your device. To make use of this feature you must do the following:

1. Add the following `#defines` in *lwipopts.h*:

   ```
   #define LWIP_MDNS_RESPONDER 1
   #define LWIP_NUM_NETIF_CLIENT_DATA 1
   ```

2. In the application's *Makefile*, add the following path to the `SOURCES` variable:

   ```
   $(SEARCH_lwip)/src/apps/mdns/mdns.c
   ```

3. In the application code , `#include` the following header files:

   ```
   #include "mdns.h"
   ```

4. To initialize and start the mDNS responder, add the following code to your application:

   ```
   err_t error;
   mdns_resp_init();
   /* IP of my device */
   struct netif *myNetif;
   myNetif = cy_network_get_interface(CY_NETWORK_WIFI_STA_NW_INTERFACE,0);
   error = mdns_resp_add_netif(myNetif, "myDevice", 100);
   if(error == ERR_OK){
       printf("mDNS responder initialized successfully.\n");
   }
   ```

   Make sure to add the code above after your code to initialize the secure sockets library and connect to Wi-Fi.

   Replace "myDevice" with whatever you want the hostname of your device to be. The third argument of `mdns_resp_add_netif` is the time to live value that will be attached to all messages sent by the responder.
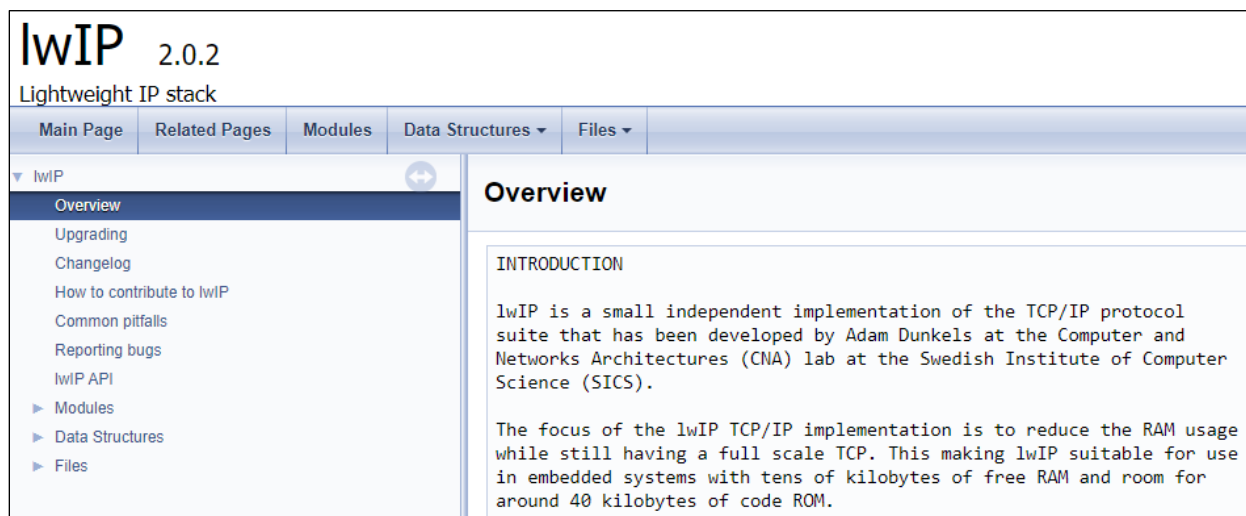
5. Add services to your device by calling the `mdns_resp_add_service` function.

2.9.5.2 Resolving mDNS hostnames

The secure sockets function `cy_socket_gethostbyname` is capable of resolving hostnames that end in ".local" via mDNS. To enable this, all you need to do is add the following `#define` to *lwipopts.h*:

```
#define LWIP_DNS_SUPPORT_MDNS_QUERIES 1
```

Documentation for the lwIP library can be found here.

The documentation related to the mDNS responder can be found under **Modules > Applications > MDNS**.

2.10 Exercises

Exercise 1: Connect to WPA2 or WPA3 Wi-Fi network

Create an App that attaches to a WPA2 or WPA3 AES PSK network, have LED1 turn on for success and blink for a failure.

☐ 1. Create a new application for the PSoC™ 6 kit you are using named **ch02_ex01_attach** based on the Empty_PSoC6_App template.

☐ 2. Open the Library manager and add the *wifi-core-freertos-lwip-mbed* and *retarget-io* libraries.

Note: The wifi-core-freertos-lwip-mbed *library relies on other Wi-Fi Middleware libraries, but as you learned earlier, they will be added automatically (wifi-connection-manager, lwIP, mbedTLS, secure-sockets, wifi-host-driver, wifi-mw-core, freertos, abstraction-rtos, etc.).*

☐ 3. Copy *FreeRTOSConfig.h* from *mtb_shared/freertos/release-vX.X.X/Source/portable/COMPONENT_CM4* to your root project directory. Open this file and delete the line that starts with `#warning`.

☐ 4. Copy the files from the *mtb_shared/wifi-core-freertos-lwip-mbed/release-vX.X.X/configs* directory to your root project directory. The files are:

lwipopts.h
mbedtls_user_config.h

☐ 5. Open the copied *mbedtls_user_config.h* file and verify that the following line is not commented out:

```
#define MBEDTLS_NO_PLATFORM_ENTROPY
```

☐ 6. Add the following lines to your project's *Makefile*:

```
COMPONENTS=FREERTOS LWIP MBEDTLS
DEFINES+=MBEDTLS_USER_CONFIG_FILE='"mbedtls_user_config.h"'
DEFINES+=CYBSP_WIFI_CAPABLE
```

Note: There are blank `COMPONENTS` and `DEFINES` lines in the file that you can modify.

☐ 7. Copy the following code into a new file called *wifi_config.h*:

```
#ifndef WIFI_CONFIG_H_
#define WIFI_CONFIG_H_

#include "cy_wcm.h"

/* SSID of the Wi-Fi Access Point to which the MQTT client connects. */
#define WIFI_SSID                       "MY_WIFI_SSID"

/* Passkey of the above mentioned Wi-Fi SSID. */
#define WIFI_PASSWORD                   "MY_WIFI_PASSWORD"

/* Security type of the Wi-Fi access point. See 'cy_wcm_security_t' structure
 * in "cy_wcm.h" for more details. */
#define WIFI_SECURITY                   CY_WCM_SECURITY_WPA3_WPA2_PSK

/* Maximum Wi-Fi re-connection limit. */
#define MAX_WIFI_CONN_RETRIES           (10u)

/* Wi-Fi re-connection time interval in milliseconds. */
#define WIFI_CONN_RETRY_INTERVAL_MS     (2000)

#endif /* WIFI_CONFIG_H_ */
```

☐ 8. Modify *wifi_config.h* for your local Wi-Fi AP credentials.

☐ 9. Change *main.c* to the following:

```c
#include "cy_pdl.h"
#include "cyhal.h"
#include "cybsp.h"
#include "FreeRTOS.h"
#include "task.h"
#include "wifi_config.h"
#include "cy_retarget_io.h"
#include "cy_wcm.h"

void wifi_connect(void *arg)
{
    cy_rslt_t result;
    cy_wcm_connect_params_t connect_param;
    cy_wcm_ip_address_t ip_address;
    uint32_t retry_count;

    /* Configure the interface as a Wi-Fi STA (i.e. Client) and initialize the WCM. */
    cy_wcm_config_t config = {.interface = CY_WCM_INTERFACE_TYPE_STA};
    cy_wcm_init(&config);

    printf("\nWi-Fi Connection Manager initialized.\n");

    /* Configure the connection parameters for the Wi-Fi interface. */
    memset(&connect_param, 0, sizeof(cy_wcm_connect_params_t));
    memcpy(connect_param.ap_credentials.SSID, WIFI_SSID, sizeof(WIFI_SSID));
    memcpy(connect_param.ap_credentials.password, WIFI_PASSWORD, sizeof(WIFI_PASSWORD));
    connect_param.ap_credentials.security = WIFI_SECURITY;

    /* Connect to the Wi-Fi AP. */
    for (retry_count = 0; retry_count < MAX_WIFI_CONN_RETRIES; retry_count++)
    {
        printf("Connecting to Wi-Fi AP '%s'\n", connect_param.ap_credentials.SSID);
        result = cy_wcm_connect_ap(&connect_param, &ip_address);

        if (result == CY_RSLT_SUCCESS)
        {
            printf("Successfully connected to Wi-Fi network '%s'.\n",
                    connect_param.ap_credentials.SSID);
            break;
        }
    }
    for(;;) {
        //Enter code to handle LED
        vTaskDelay(100);
    }
}

int main(void)
{
    cy_rslt_t result;

    /* Initialize the device and board peripherals */
    result = cybsp_init() ;
    if (result != CY_RSLT_SUCCESS)
    {
        CY_ASSERT(0);
    }

    /* Initialize retarget-io to use the debug UART port. */
    cy_retarget_io_init(CYBSP_DEBUG_UART_TX, CYBSP_DEBUG_UART_RX, CY_RETARGET_IO_BAUDRATE);

    __enable_irq();

    printf("\x1b[2J\x1b[;H\n"); /* ANSI ESC sequence to clear screen. */

    /* Create the WiFi connection task. It must have a priority of 3 or lower. */
    xTaskCreate(wifi_connect, "wifi_connect_task", 1024, NULL, 1, NULL);

    vTaskStartScheduler();          /* Never Returns */
    }
```

Note: The call to vTaskDelay in the for loop in the wifi_connect task is necessary so that it doesn't prevent Wi-Fi tasks from running. If you don't need the for loop to blink the LED, you could choose to exit the task using vTaskDelete.

Note: The code provided in this exercise is for connecting to a network with WPA2 or WP3 security. If your network has a different security type you will need to edit the `WIFI_SECURITY` macro in wifi_config.h and the variable `connect_param` in main.c. For the details on how these variables need to be updated for your specific network security, refer to the Wi-Fi Connection Manager Library Documentation.

Note: The Wi-Fi Connection Manager API does not support connecting to networks with WEP security. If your local network uses WEP security, you should consider updating it to use a different security protocol.

10. Edit this code so that your device turns on an LED if it connects and blinks an LED continuously if it is unable to.

Note: Use a serial terminal emulator to look at messages from the device as it boots and connects. If you need a refresher on using a serial terminal emulator, see ModusToolbox™ Level 1 Getting Started class, Tools chapter, Serial Terminal Emulator section.

Note: To test the failing case, intentionally put in the wrong SSID or password.

Exercise 2: Connect to an Open network

1. How would you modify the previous exercise to attach to a different network that is open (i.e. no security)?

Note: There are only two changes required.

Exercise 3: Exercise 3: Print network information

1. Create a new ModusToolbox™ application for the PSoC™ 6 kit you are using.

   On the application template page, use the **Browse** button to specify the completed **ch02_ex01_attach** exercise as a template.
   Name the new application **ch02_ex03_print**.

2. Add code to the `wifi_connect` function to print out networking information if the connection is successful:

   - Your IP address (`cy_wcm_ip_address_t`)
   - Netmask (`cy_wcm_connect_params_t`)
   - Router Gateway (`cy_wcm_connect_params_t`)
   - The IP address of www.infineon.com (`cy_socket_gethostbyname()`)
   - MAC Address of your device (`cy_wcm_get_mac_addr()`)

Note: Your IP address can be obtained from the `cy_wcm_ip_address_t` object that you passed into the `cy_wcm_connect_ap` function

Note: Your netmask and gateway addresses can be obtained from the `cy_wcm_connect_params_t` object that you passed into the `cy_wcm_connect_ap` function

Note: Be sure to `#include "cy_secure_sockets.h"` in order to use `cy_socket_gethostbyname`

Note: The addresses (IP address, Netmask, Gateway, and Infineon.com) are returned as a structure of type `cy_wcm_ip_address_t`. One element in the structure (called `ip.v4`) is a `uint32_t` which contains the IPV4 address as 4 hex bytes. You can mask off each of these bytes individually and print them as decimal values separated by periods to get the format that is typically seen. For example, the netmask of 255.255.255.0 will be returned as 0xFFFFFF00.

Note: Make sure the third argument you pass to `cy_wcm_get_mac_addr` is the length of the `cy_wcm_mac_t` pointer you passed in.

Note: The MAC address is returned as a structure of type `cy_wcm_mac_t`. This structure contains an array of `uint8_t` objects. You can print each of these bytes individually separated by ":" to see the MAC address in the typical format.

Exercise 4: Multiple network connectivity

Create an application that can switch between two different SSIDs.

Note: You will need a second network to do this exercise. If you don't have a second network, you could use a smartphone to temporarily create a second Wi-Fi network to use for this exercise. Most smartphones have this functionality.

☐ 1. Create a new ModusToolbox™ application for the PSoC™ 6 kit you are using.

☐ On the application template page, use the **Browse** button to specify the completed **ch05_ex03_print** exercise as a template.

2. Name the new application **ch02_ex04_multi**.

☐ 3. Create a function that can print the SSID/Passphrase and Security for the network that your device is currently connected to.

☐ 4. Create a function that takes input as (`char* ssid, char* passphrase, cy_wcm_security_t security`) and then connects to the network specified by that information:

    a. Take the network down (`cy_wcm_disconnect_ap`).
    b. Write the new parameters to an object of type `cy_wcm_connect_params_t` to update the ssid and passphrase:
       Hint Since the values are strings:
- Use `memcpy` to copy the values into the buffer.
- Make sure you update the string length in the structure (you can use `strlen` to find the length of the string).

    Restart the network (`cy_wcm_connect_ap`).

☐ 5. Use the console as input. When the user presses '0' or '1' switch between the two networks.

    If the user presses 'p', call the print function that you wrote in step 2.

Note: *Review the UART receive exercises from the peripherals chapter of the ModusToolbox ™-Level2-PSoC™ training class.*

☐   6.   Program the project to the kit. Test the functionality to change the selected network and print out the network details for each network.

2.11   Recommended reading

- [1] TCP/IP Illustrated – Volume 1: The Protocols, W.R. Stevens, ISBN 0201633469 – "aka" the Networking Bible, if there is one book to get on TCP/IP networking, this is it!
- [2] UNIX Network Programming – W.R. Stevens, ISBN 01394 – if you want to learn BSD Socket programming, there is no other reference – best book and the foundation of all networking software today.
- [3] RFC 1122 – "Requirements for Internet Hosts – Communications Layers" ; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc1122
- [4] RFC 826 – "An Ethernet Address Resolution Protocol" ; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc826
- [5] RFC 153 – "Dynamic Host Configuration Protocol"; Internet Engineering Task Force (IETF) - https://tools.ietf.org/html/rfc1531

2.12   Appendix

Answers to the questions asked in the exercises above are provided here.

2.12.1   Exercise 2 Answers

1.   How would you modify the previous exercise to attach to a different network that is open (i.e. no security)?

`WIFI_SSID` changes to `WW101OPEN`

`WIFI_SECURITY` changes to `CY_WCM_SECURITY_OPEN`

Note: *You can find all of the security types available by right clicking on CY_WCM_SECURITY_OPEN (or any other security name) and selecting **Open Declaration**.*

Trademarks
All referenced product or service names and trademarks are the property of their respective owners.