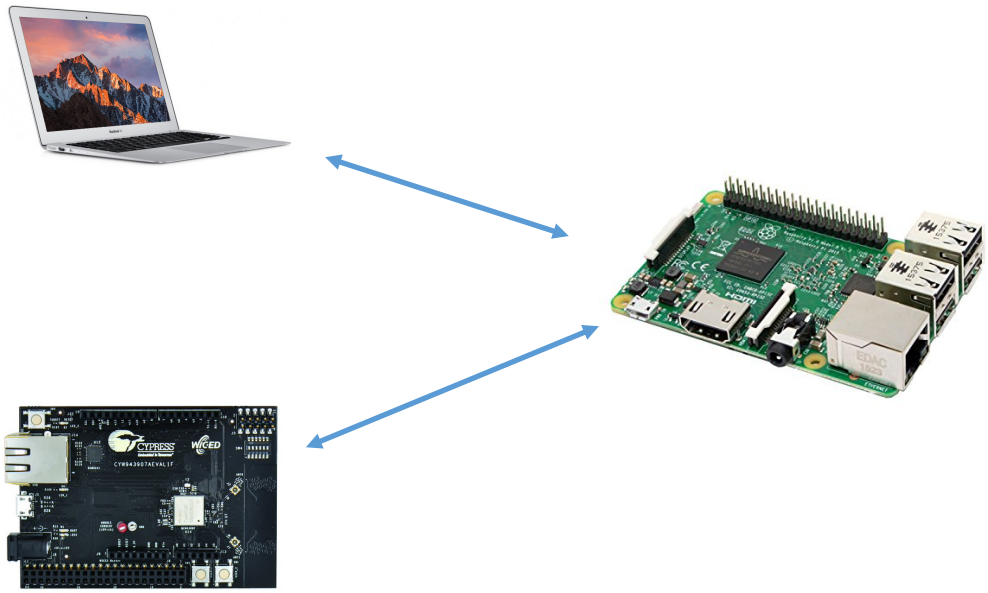
INTERNET OF THINGS | COEN 243
DEPARTMENT OF COMPUTER ENGINEERING, SANTA CLARA UNIVERSITY, CALIFORNIA
BEHNAM DEZFOULI, PHD

Internet of Things

Hands-On: Application Layer Protocols

Outline

- Raspberry Pi: Socket Programming
- Implementing an Application Layer Protocol using Cypress WICED Studio and Java on Raspberry Pi
- Connection to an MQTT server



Raspberry Pi: Socket Programming

Raspberry Pi: Socket Programming

- **Client - Server Model**
  In a client-server model of computing, a **server** hosts a resource or a service which is accessed by **clients**
- A **server** creates a socket in its communication end and binds itself to a port and then listens (waits) for incoming connections from **client**
- A client creates a socket in its end and connects to the listening server at the specified port

Raspberry Pi: Socket Programming

❑ **Example:**

- We intend to develop an application in which **a client (running on a PC) sends a message to sever program running on RPi**

- **`ServerSocket`** class: used by the server side program to obtain and bind to a port and listen for incoming connections from client
  - **`accept()`:** Listens for a connection to be made to this socket and accepts it; returns a **`socket`**
- **`Socket`** class: is **used by both client and server** to communicate with each other
  - **`getInputStream()`:** Returns an `inputStream` for this socket
  - **`getOutputStream()`:** Returns an `outputStream` for this socket

- **`ObjectOutputStream`** and **`ObjectInputStream`** contain necessary methods which provide support for **serializing** and **deserializing** an object
- **`readObject()`** and **`writeObject()`** are used for serialization and deserialization

Raspberry Pi: Socket Programming

Example: **PC-Rpi** (`simple-client-server-1`)
RPi Side - Part 1/1

```
import java.io.*; import java.net.*;

public class Server {
    public static void main(String args[]) {
        try {

            ServerSocket ss = new ServerSocket(6999);
            System.out.println("Server is lisenting ... ");


            Socket skt = ss.accept();

            InputStream is = skt.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(is);

            // receive message from client
            String msg = (String)ois.readObject();
            System.out.println("Server received message : " + msg);
            System.out.println("Server is exiting ... ");
            ois.close();
            is.close();
            skt.close();
            ss.close();
        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Using `ServerSocket`, we create a new server socket that listens on port 6999

`accept()`: waits for connection from client, and returns a `socket`

Server socket receives message

Close open streams and sockets

Raspberry Pi: Socket Programming

Example: **PC-RPi** (simple-client-server-1)
PC Side - Part 1/1

```
import java.io.*;
import java.net.*;
import java.io.BufferedReader;
public class Client2Pi {
    public static void main(String args[]) {
        try {
            Socket skt = new Socket("192.168.1.138", 6999);
            System.out.println("Connected to server");

            BufferedReader br = new BufferedReader(new
                                        InputStreamReader(System.in));
            String accStr;
            System.out.println("Enter text: ");
            String msg = br.readLine();

            OutputStream os = skt.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            oos.writeObject(msg);
            System.out.println("Sent a message to server");

            oos.close();
            os.close();
            skt.close();
        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Send a message to server

Close open streams and sockets

Raspberry Pi: Socket Programming

❑ **Example:**

- Write an application in which **a client (running on a PC) connects to a server (running in RPi) to receive an status message**

Raspberry Pi: Socket Programming

Example: **PC-RPi** (simple-client-server-2)
RPi Side - Part 1/1

```
import java.io.*; import java.net.*; import java.net.InetAddress;

public class Server {
    public static void main(String args[]) {
        try {

            ServerSocket ss = new ServerSocket(7999);
            System.out.println("Server is listenting ... ");
            Socket skt = ss.accept();
            OutputStream os = skt.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);

            String msg = "Sensor data returned!";
            oos.writeObject(msg); // send a message to client
            System.out.println("Sent a message to client");
            System.out.println("Server is exiting ... ");

            oos.close();
            os.close();
            skt.close();
            ss.close();

        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Server socket sends message

Close open streams and sockets

Raspberry Pi: Socket Programming

Example: **PC-RPi** (simple-client-server-2)
PC Side - Part 1/1

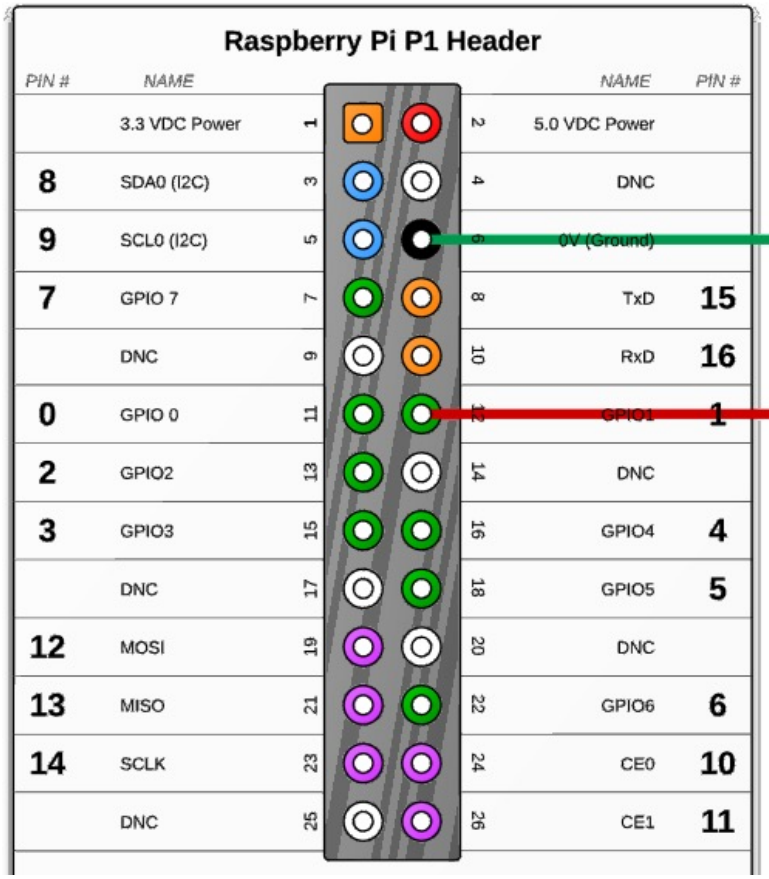
```
import java.io.*;
import java.net.*;

public class Client {
    public static void main(String args[]) {
        try {

            Socket skt = new Socket("192.168.1.138", 7999);
            System.out.println("Connected to server");
            InputStream is = skt.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(is);
            String msg = (String)ois.readObject();

            System.out.println("Received message from Server : " + msg);

            ois.close();
            is.close();
            skt.close();

        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

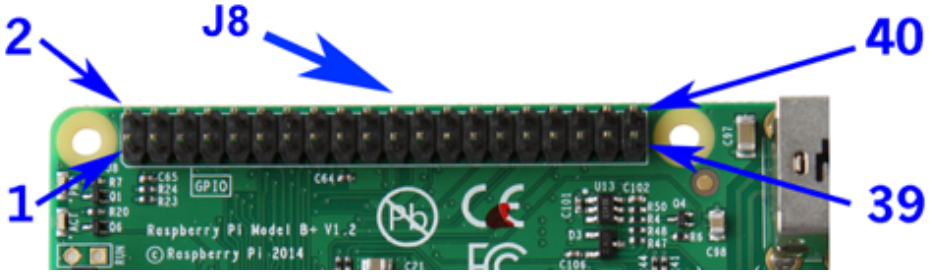
Client socket receives message

Close open streams and sockets

❑Example:

- We intend to develop an application in which **a client (running on a PC) connects to a server (running in RPi) to control blinking rate of an LED**





Raspberry Pi: Socket Programming

Example: **PC-RPi** (client-server-LED)
RPi Side - Part 1/3

```
import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalOutput;
import com.pi4j.io.gpio.PinState;
import com.pi4j.io.gpio.RaspiPin;
import java.io.*; import java.net.*; import java.util.Timer;

public class Server {
    public static void main(String args[]) {
    // create gpio controller
    final GpioController gpio = GpioFactory.getInstance();

    // provision gpio pin #01 as an output pin and turn on
    final GpioPinDigitalOutput pin =
       gpio.provisionDigitalOutputPin(RaspiPin.GPIO_01, "MyLED", PinState.HIGH);

        Timer timer = new Timer();
        Reminder reminderTask = new Reminder(pin);

        try {
            ServerSocket ss = new ServerSocket(6999);
            System.out.println("Server is lisenting ... ");
            Socket skt = ss.accept();

            OutputStream os = skt.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            InputStream is = skt.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(is);
```

Create a new GPIO controller instance

Provision GPIO pin 1

Create a timer and a task to be scheduled by the timer

Raspberry Pi: Socket Programming

Example: **PC-RPi** (client-server-LED)
Pi Side - Part 2/3

```
            BufferedReader br = new
                        BufferedReader(new InputStreamReader(System.in));

            int rcv_msg;
            String send_msg;
            timer.schedule(reminderTask, 0, 1000);

            while (true) {
                /* Receive a message from client */
                if ((rcv_msg = (int) ois.readObject()) != 0) {
                    System.out.println("New interval : " + rcv_msg);
                }

                reminderTask.cancel();
                reminderTask = new Reminder(pin);
                timer.schedule(reminderTask, 0, (long) rcv_msg);

                oos.writeObject("Interval changed!");
            }
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

Raspberry Pi: Socket Programming

Example: **PC-RPi** `(client-server-LED)`
Pi Side - Part 3/3

```
import java.util.Timer;
import java.util.TimerTask;
import com.pi4j.io.gpio.GpioController;
import com.pi4j.io.gpio.GpioFactory;
import com.pi4j.io.gpio.GpioPinDigitalOutput;
import com.pi4j.io.gpio.PinState;
import com.pi4j.io.gpio.RaspiPin;


public class Reminder extends TimerTask {

    GpioPinDigitalOutput pin;

    Reminder (GpioPinDigitalOutput pin) {
        this.pin = pin;
    }

    public void run() {
        pin.toggle();
    }

}
```

The constructor of the task takes a `pin` as input

Raspberry Pi: Socket Programming

Example: **PC-RPi** (client-server-LED)
PC Side

```
import java.io.*; import java.net.*;
public class Client {
    public static void main(String args[]) {
        try {
            Socket skt = new Socket("192.168.1.138", 6999);
            System.out.println("Connected to server");
            InputStream is = skt.getInputStream();
            ObjectInputStream ois = new ObjectInputStream(is);
            OutputStream os = skt.getOutputStream();
            ObjectOutputStream oos = new ObjectOutputStream(os);
            BufferedReader br=new
                        BufferedReader(new InputStreamReader(System.in));
            String rcv_msg;   int send_msg;
            while(true) {
                System.out.print("To Server : ");
                send_msg = Integer.parseInt(br.readLine());
                oos.writeObject(send_msg);

                System.out.println("Waiting for server to respond ... ");
                if((rcv_msg = (String)ois.readObject()) != null) {
                    System.out.println("\nFrom Server : " + rcv_msg);
                }
            }
        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Raspberry Pi: Socket Programming

- You must first create a new GPIO controller instance
- The `GpioFactory` includes a `createInstance()` method to create the GPIO controller
- Your project **should only instantiate a single GPIO controller instance** and that instance should be shared across your project

```
// create gpio controller instance
final GpioController gpio = GpioFactory.getInstance();
```

- To access a GPIO pin, you must first **provision** the pin
- Provisioning configures the pin based on how you intend to use it

```
// provision gpio pins #04 as an output pin and make sure is is set
// to LOW at startup
GpioPinDigitalOutput myLed = gpio.provisionDigitalOutputPin(
        RaspiPin.GPIO_04, // PIN NUMBER
        "My LED", // PIN FRIENDLY NAME (optional)
        PinState.LOW); // PIN STARTUP STATE (optional)
```

Raspberry Pi: Socket Programming

- `public abstract class` **`TimerTask`**
  - A task that can be scheduled for one-time or repeated execution by a Timer
    - extends `Object`
    - implements `Runnable`
  - **`public abstract void run()`**
    - The action to be performed by this timer task
  - **`public boolean cancel()`**
    - Cancels this timer task
    - If the task has been scheduled for one-time execution and has not yet run, or has not yet been scheduled, it will never run

Raspberry Pi: Socket Programming

- `public class Timer`
  - A facility for threads to schedule tasks for future execution in a background thread
  - Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals
  - Corresponding to each `Timer` object is a single background thread that is used to execute all of the timer's tasks, sequentially

- `schedule (TimerTask task, long delay, long period)`
  - Schedules the specified task for repeated **fixed-delay** *execution*, beginning after the specified delay
  - **Parameters:**
    - `task` - task to be scheduled
    - `delay` - delay in milliseconds before task is to be executed
    - `period` - time in milliseconds between successive task executions

Implementing an Application Layer Protocol

Software Development using WICED Studio and JAVA

Implementing an Application Layer Protocol

- We first assign a name to the RaspberryPi server
- **This would enable us to access the device using its name, rather than its IP address**

1. Open the host files: `sudo nano /etc/hosts`

   Change the very last entry labeled 127.0.1.1 with the hostname "`scu_iot_server`"

2. Open the hostname file: `sudo nano /etc/hostname`

   Replace the default "`raspberrypi`" with the same hostname you put in the previous step (i.e., "`scu_iot_server`")

3. Enter the following command to commit the changes:

   `sudo /etc/init.d/hostnames`

4. Reboot the systems: `sudo reboot`

Implementing an Application Layer Protocol

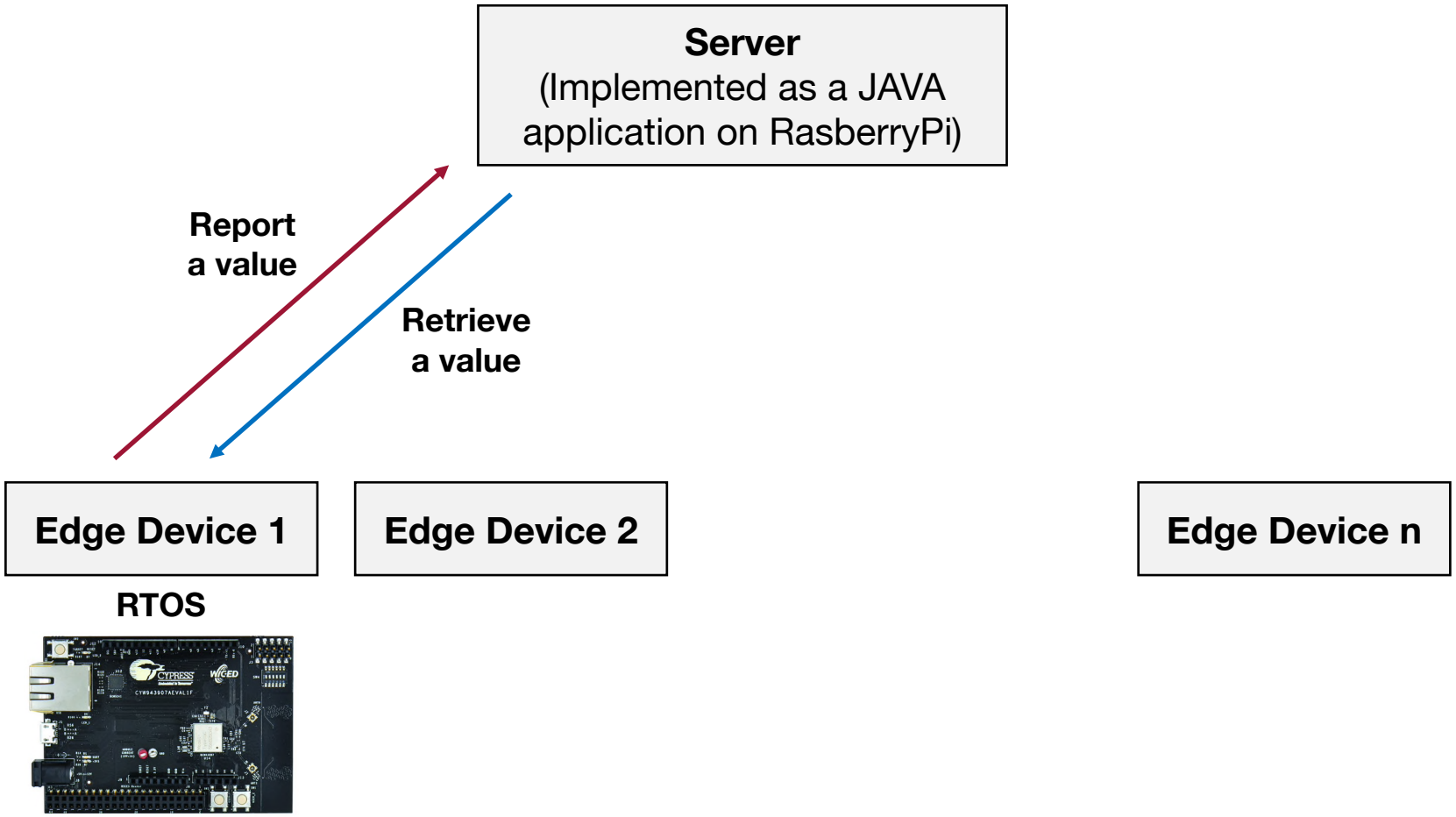
- Although people usually use standard application layer protocols such as HTTP and MQTT, **here we would like to develop our own application layer protocol**

- **Example:** We are interested to develop the following application:

- **Create an IoT client to write data to a server when a button is pressed on the client**
- Your application will monitor button presses on the board and will toggle an LED in response to each button press
- In addition, your application will connect to the server and will send the state of the LED each time the button is pressed
- For the device ID, use your device's MAC address

- The client sends "R" or "W" command
- The server responds with "A" (and the data is echo'd)
- The server contains a database that will store values that are written to it (when a client uses the "W" command) and will send back requested values (when a client uses the "R" command)

Implementing an Application Layer Protocol



Implementing an Application Layer Protocol

Our protocol works as follows:

- The client and the server both send a string of characters that are of the form:
  - **Command**: [1 character] representing the command (R=Read, W=Write, A=Accepted, X=Failed)
  - **Name Initials** [4 characters]
  - **Device ID**: [12 characters] representing the MAC of the device
  - **Register**: [2 character] representing the register (each device has 256 registers) e.g. 0F or 1B
  - **Value**: [4 characters] representing the hex value of a 16-bit unsigned integer; The value should be left out on “R” commands

COMMAND (1 Byte)	-	NAME INIT (4 Bytes)	-	DEVICE ID (12 Bytes)	-	REGISTER (2 Bytes)	-	VALUE (4 Bytes)

Implementing an Application Layer Protocol

COMMAND (1 Byte)	-	NAME INIT (4 Bytes)	-	DEVICE ID (12 Bytes)	-	REGISTER (2 Bytes)	-	VALUE (4 Bytes)

❑Example

- "**W-BDSC-000A959D6816-0B-1234**" would write a **value of 0x1234** to **register 0x0B** for device with MAC address 00:0a:95:9d:68:16; The server would then respond with "**A-BDSC-000A959D6816-0B-1234**"
- "W01234" is an illegal packet and the server would respond with "I"
- "**R-BDSC-000A959D6816-0B**" is a read of register 0x0B for a Device ID with MAC 00:0a:95:9d:68:16
  - In this case the server would respond with "**A-BDSC-000A959D6816-0B-1234**" (the value of 1234 was written in the first case)
  - Otherwise, the served returns a "I"

Implementing an Application Layer Protocol

- **Note that "raw" sockets inherently don't have security**
- The TCP socket just sends whatever data it was given over the link
- It is the responsibility of a layer above TCP such as SSL or TLS to encrypt/decrypt the data if security is being used (which we will cover later on)

Implementing an Application Layer Protocol

Programming Edge Devices

To setup the **TCP client connection**, the client firmware will:

- **Create** the TCP socket by calling:

```
wiced_tcp_create_socket( &socket, WICED_STA_INTERFACE );
```

- **Bind** to TCP port 6999 by calling:

```
wiced_tcp_bind( &socket, 7001);
```

  - We may use `WICED_ANY_PORT` if the port number is not important

- **Connect** to port 6999 through the network by calling `wiced_tcp_connect()` and waiting a TIMEOUT number of milliseconds for a connection

```
wiced_tcp_connect( &socket, &serverAddress, 7000, TIMEOUT);
```

  - `serverAddress` is the server's IP address
  - In our local network the timeout can be small <1s
  - In a WAN situation the timeout may need to be extended to as long as a few seconds

To find the server address:

- IP address is a WICED data structure of type `wiced_ip_address_t`
- You can initialize the structure in one of two ways – either statically or using DNS
  - **Static** method: use the macros provided by the WICED SDK as follows:

```
SET_IPV4_ADDRESS( serverAddress, MAKE_IPV4_ADDRESS(
                                    198, 51, 100, 3 ) );
```

  - **DNS**: To initialize it by performing a DNS loop

```
wiced_hostname_lookup( "iotserver2 ",
                              &serverAddress, 10000 );
```

Implementing an Application Layer Protocol

Programming Edge Devices

- Once the connection has been created, your application will want to transfer data between the client and serve
- The simplest way to transfer data over TCP is **to use the stream functions** from the SDK
- The stream functions allow you to send and receive arbitrary amounts of data **without worrying about the details of packetizing data** into uniform packets

- To use a stream you must first declare a stream structure and then initialize that with the socket for your network connection:

```
wiced_tcp_stream_t stream;
wiced_tcp_stream_init(&stream, &socket);
```

- Once this is done it is simple to write data using the `wiced_tcp_stream_write()` function
  - This function takes the stream and message as parameters
  - The message is just an array of characters to send
- When you are done writing to the stream, you need to call the `wiced_tcp_stream_flush()` method

Implementing an Application Layer Protocol

Programming Edge Devices

❑ Example: The following code demonstrates writing a single message:

```
char sendMessage[] = "TEST_MESSAGE";
wiced_tcp_stream_write(&stream, sendMessage,
                                    strlen(sendMessage));
wiced_tcp_stream_flush(&stream);
```

- **Reading data from the stream** uses the `wiced_tcp_stream_read()` function
- This method takes a stream and a message buffer as parameters
- The function also requires you to specify the maximum number of bytes to read into the buffer and a timeout
- The function returns a `wiced_result_t` value which can be used to ensure that reading the stream succeeded

Implementing an Application Layer Protocol

Programming Edge Devices

Read data from a TCP stream

```
 *
 * @param[in,out] tcp_stream    : A pointer to a stream handle where data will
                                                                 be written
 * @param[out]    buffer        : The memory buffer to write data into
 * @param[in]     buffer_length : The number of bytes to read into the buffer
 * @param[in]     timeout       : Timeout value in milliseconds or
                                                       WICED_NEVER_TIMEOUT
 *
 * @return @ref wiced_result_t
 */
wiced_result_t wiced_tcp_stream_read( wiced_tcp_stream_t* tcp_stream,
                      void* buffer, uint16_t buffer_length, uint32_t timeout );
```

Implementing an Application Layer Protocol

Programming Edge Devices

- Example: Putting all together…

```
#define SERVER_PORT (6999)
#define TIMEOUT (2000)
.
.
wiced_tcp_socket_t socket;
wiced_tcp_stream_t stream;
char sendMessage[]="W000A959D68160B1234";
.
.
wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);
wiced_tcp_bind(&socket, WICED_ANY_PORT );


wiced_tcp_connect(&socket, &serverAddress, SERVER_PORT, TIMEOUT);
wiced_tcp_stream_init(&stream, &socket);
wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage);
wiced_tcp_stream_flush(&stream);


wiced_tcp_stream_deinit(&stream);
wiced_socket_delete(&socket);
```

Implementing an Application Layer Protocol

Programming Edge Devices

- Behind the scenes, reading and writing via streams uses uniform sized packets
- The stream functions in the SDK hides the management of each of these packets from you so you can focus on the higher levels of your application
- However, if you desire more control over the communication, you can use the WICED SDK API to send and receive packets directly

Implementing an Application Layer Protocol

Example: **New App Protocol**
C Code: Edge Side – (client-serverDB-stream)
Part 1/6

```
#include "wiced.h"
#include "register_map.h"

#define TCP_CLIENT_STACK_SIZE      (6200)
#define SERVER_PORT                (6999)


static wiced_ip_address_t serverAddress; //Server address
static wiced_semaphore_t button0_semaphore, button1_semaphore;
static wiced_thread_t buttonUpdate, buttonInquiry;
static wiced_mac_t myMac; //MAC address



void button_isr1(void *arg)
{
    wiced_rtos_set_semaphore(&button1_semaphore);
}


void button_isr0(void *arg)
{
    wiced_rtos_set_semaphore(&button0_semaphore);
}
```

Button 1 **sends status** update to the server

Button 0 **requests status** from the server

Implementing an Application Layer Protocol

Example: **New App Protocol**
C Code: Edge Side – (`client-serverDB-stream`) Part 2/6

```
void sendData(char type, uint8_t regAdd, int regVal) {
        wiced_tcp_socket_t socket; // The TCP socket
        wiced_tcp_stream_t stream; // The TCP stream
        char sendMessage[28];
        wiced_result_t result, conStatus;

        if (type == 'W')
            sprintf(sendMessage,"W-BDSC-%02X%02X%02X%02X%02X%02X-%02X-%04X\n",
                myMac.octet[0], myMac.octet[1], myMac.octet[2],
                myMac.octet[3], myMac.octet[4], myMac.octet[5],
                regAdd, regVal);
        else if (type == 'R')
            sprintf(sendMessage,"R-BDSC-%02X%02X%02X%02X%02X%02X-%02X\n",
                    myMac.octet[0], myMac.octet[1], myMac.octet[2],
                    myMac.octet[3], myMac.octet[4], myMac.octet[5],
                    regAdd);
        else{
            WPRINT_APP_INFO(("Invalid command type\n"));
            return;
        }
```

Preparing a server update message

Preparing a server inquiry message

Establish connection with the server

```
        wiced_tcp_create_socket(&socket, WICED_STA_INTERFACE);
        wiced_tcp_bind(&socket,WICED_ANY_PORT);
        conStatus = wiced_tcp_connect(&socket,&serverAddress,SERVER_PORT,2000);
```

Implementing an Application Layer Protocol

```
    if(conStatus == WICED_SUCCESS)
         WPRINT_APP_INFO(("Successful connection!\n"));
    else {
        WPRINT_APP_INFO(("Failed connection!\n"));
         wiced_tcp_delete_socket(&socket);
        return; // Return from the function
    }

    // Initialize the TCP stream
    wiced_tcp_stream_init(&stream, &socket);

    // Send the data via the stream
    wiced_tcp_stream_write(&stream, sendMessage, strlen(sendMessage));
    wiced_tcp_stream_flush(&stream);


    char rbuffer[28] = {0};
    result = wiced_tcp_stream_read(&stream, rbuffer, 27, 500);

    if(result == WICED_SUCCESS)
        WPRINT_APP_INFO(("Server Response = %s\n\n\n",rbuffer));
    else
        WPRINT_APP_INFO(("Malformed response = %s\n\n\n",rbuffer));

    // Delete the stream and socket
    wiced_tcp_stream_deinit(&stream);
    wiced_tcp_delete_socket(&socket);
}
```

Example: **New App Protocol**
C Code: Edge Side – `(client-serverDB-stream)`
Part 3/6

Force the data to be sent right away even if the packet isn't full yet

Get the response back from the server

Implementing an Application Layer Protocol

Example: **NewALProtocol**
C Code: Edge Side – (client-serverDB-stream)
Part 4/6

```
// This is a thread function
void buttonUpdateMain()
{
    while(1)
    {
        wiced_rtos_get_semaphore(&button1_semaphore,WICED_WAIT_FOREVER);
        wiced_gpio_output_low( WICED_SH_LED1 );
        sendData('W', reg_LED1, 0);

        wiced_rtos_get_semaphore(&button1_semaphore,WICED_WAIT_FOREVER);
        sendData('W', reg_LED1, 1);
        wiced_gpio_output_high( WICED_SH_LED1 );
    }
}
```

Toggle LED and update the server

```
// This is a thread function
void buttonInquiryMain()
{
    while(1)
    {
        wiced_rtos_get_semaphore(&button0_semaphore,WICED_WAIT_FOREVER);
        sendData('R', reg_LED1, 0);
    }
}
```

Inquire the server

Implementing an Application Layer Protocol

Example: **New App Protocol C Code: Edge Side** – (client-serverDB-stream) Part 5/6

```
void application_start(void) {
  wiced_init( );
  wiced_network_up( WICED_STA_INTERFACE, WICED_USE_EXTERNAL_DHCP_SERVER, NULL );

  wiced_result_t result;

  wiced_network_set_hostname("WICED001");

  wwd_wifi_get_mac_address(&myMac, WICED_STA_INTERFACE );

  // Use DNS to find the address
  //if you can't look it up after 5 seconds then hard code it.
  WPRINT_APP_INFO(("DNS Lookup iotserver2\n"));
  result = wiced_hostname_lookup( "iotserver2", &serverAddress,
                                          5000, WICED_STA_INTERFACE );

  if ( result == WICED_ERROR || serverAddress.ip.v4 == 0 )    {
      WPRINT_APP_INFO(("Error in resolving DNS using hard coded address\n"));
      SET_IPV4_ADDRESS( serverAddress, MAKE_IPV4_ADDRESS( 192,168, 2, 167 ) );
  }
  else     {
      WPRINT_APP_INFO(("iotserver2 IP : %u.%u.%u.%u\n\n",
          (uint8_t)(GET_IPV4_ADDRESS(serverAddress) >> 24),
              (uint8_t)(GET_IPV4_ADDRESS(serverAddress) >> 16),
                  (uint8_t)(GET_IPV4_ADDRESS(serverAddress) >> 8),
                      (uint8_t)(GET_IPV4_ADDRESS(serverAddress) >> 0)));
     }
```

Assign the devices hostname

Implementing an Application Layer Protocol

Example: **New App Protocol**
C Code: Edge Side – (client-serverDB-stream) Part 6/6

```
    WPRINT_APP_INFO(("MY MAC Address: "));
    WPRINT_APP_INFO(("%X:%X:%X:%X:%X:%X\r\n",
            myMac.octet[0], myMac.octet[1], myMac.octet[2],
            myMac.octet[3], myMac.octet[4], myMac.octet[5]));


    // Setup the Semaphore and Button Interrupt
    wiced_rtos_init_semaphore(&button0_semaphore);
    wiced_rtos_init_semaphore(&button1_semaphore);


    wiced_gpio_input_irq_enable(WICED_SH_MB0, IRQ_TRIGGER_FALLING_EDGE,
         button_isr0, NULL); // call the ISR when the button is pressed
    wiced_gpio_input_irq_enable(WICED_SH_MB1, IRQ_TRIGGER_FALLING_EDGE,
         button_isr1, NULL); // call the ISR when the button is pressed

    wiced_rtos_create_thread(&buttonUpdate, WICED_DEFAULT_LIBRARY_PRIORITY,
         "Button Update", buttonUpdateMain, TCP_CLIENT_STACK_SIZE, 0);

    wiced_rtos_create_thread(&buttonInquiry, WICED_DEFAULT_LIBRARY_PRIORITY,
         "Button Inquiry", buttonInquiryMain, TCP_CLIENT_STACK_SIZE, 0);

    WPRINT_APP_INFO(("Activated button threads..."));

}
```

Implementing an Application Layer Protocol

Programming the Server

- For the server side, we implement a Java application that creates a thread per connection
- Each client thread parses the command received and either writes to or reads from a database
- The database is implemented as a linked list of objects storing client information

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side – (client-serverDB-stream) –** Part 1/9

```
package applayerprotocol;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader; import java.io.PrintWriter;
import java.math.BigInteger; import java.net.ServerSocket;
import java.net.Socket; import java.util.LinkedList;

class DeviceInfo {
    long macAddress;
    int[] registers;
}

class DeviceDB {  //Database to hold device information
    LinkedList<DeviceInfo> devices;

    public DeviceDB() {
        devices = new LinkedList<>();
    }

    public int findDevice(long macAddress) {
        for (int index = 0; index < devices.size(); ++index) {
            if (devices.get(index).macAddress == macAddress) {
                return index;
            }
        }
        return -1;
    }
```

For each client we store its MAC address, and the value of up to 255 registers

This class stores clients' information as a linked list

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – `(client-serverDB-stream)`
Part 2/9

```
public int addDevice(long macAddress) {
        DeviceInfo newDevice = new DeviceInfo();
        newDevice.macAddress = macAddress;
        newDevice.registers = new int[256];
        devices.addLast(newDevice);
        return devices.size() - 1;
    }


    public void setReg(int index, int regAdd, int regVal) {
        devices.get(index).registers[regAdd] = regVal;
    }


    public int getReg(int index, int regAdd) {
        return devices.get(index).registers[regAdd];
    }
}
```

When a new device contacts the server, we add a new entry to the database

We use this method to set the value of a register

Requires `findDevice` to be called first to retrieve the index of that entry

We use this method to get the value of a register

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – (client-serverDB-stream) Part 3/9

```
public class Server {
    public static int transID = 0;

    public static void main(String args[]) {
        ServerSocket ss2 = null;
        DeviceDB db = new DeviceDB();

        try {
            ss2 = new ServerSocket(6999);
         } catch (IOException e) {
            System.out.println("Server error");
            java.lang.System.exit(1);
        }
        System.out.println("Server started...");

        while (true) {
            try {
                Socket s = null;
                s = ss2.accept();
                ServerThread st = new ServerThread(s, db);
                st.start();

            } catch (Exception e) {
                System.out.println("Connection Error");
            }
        }
    }
}
```

A thread is created when a new connection is established

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – (client-serverDB-stream) Part 4/9

```
class ServerThread extends Thread {

    String line = null;
    BufferedReader is = null;
    PrintWriter os = null;
    Socket s = null;
    DeviceDB db;

    public ServerThread(Socket s, DeviceDB db) {
        this.s = s;
        this.db = db;
    }

    public Boolean ValidateWriteCmd(String[] command, PrintWriter os) {
        if (command[0].length() != 1 || command[1].length() != 4
                || command[2].length() != 12 || command[3].length() != 2 ||
                command[4].length() != 4) {
            System.out.println("Invalid write command!");
            SendFailResponse(os);
            return false;
        }
        return true;
    }
```

This method enables us to check the validity of a write command
Each part of the command is a string of the string array

Reminder: Command format is

COMMAND (1 Byte)	-	NAME INIT (4 Bytes)	-	DEVICE ID (12 Bytes)	-	REGISTER (2 Bytes)	-	VALUE (4 Bytes)

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – (client-serverDB-stream) – Part 5/9

```
public Boolean ValidateReadCmd(String[] command, PrintWriter os) {
    if (command[0].length() != 1 || command[1].length() != 4
            || command[2].length() != 12 || command[3].length() != 2) {
        System.out.println("Invalid read command!");
        SendFailResponse(os);
        return false;
    }
    return true;
}
```

This method enables us to check the validity of a read command

```
public void SendFailResponse(PrintWriter os) {
    char[] response = {'I', '\n'};
    os.println(response);
    os.flush();
}
```

This method sends a message to the client to indicate message parsing failure

```
public void SendResponse(String retChar, String parts1, String parts2,
                         String parts3, String parts4, PrintWriter os) {
    String sentResponse =
         retChar + "-" + parts1 + "-"  + parts2 + "-" + parts3 + "-" + parts4;
    os.println(sentResponse);
    os.flush();
    System.out.println("Sent response: " + sentResponse);
}
```

This method prepares and sends a response to a client after successful execution of the received command

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – (client-serverDB-stream)
Part 6/9

```
@Override
public void run() {
    Server.transID++;
    System.out.println("----Transaction: " + Server.transID + "----");

    try {
        is = new BufferedReader(new InputStreamReader(s.getInputStream()));
        os = new PrintWriter(s.getOutputStream());

    } catch (IOException e) {
        System.out.println("IO error in server thread");
    }

    try {
        line = is.readLine();
        System.out.println("Received: " + line);

        String parts[] = line.split("-");
```

We split the command received into strings

Implementing an Application Layer Protocol

```
if ( (parts[0].equals("W") && ValidateWriteCmd(parts, os))
        || (parts[0].equals("R") && ValidateReadCmd(parts, os))) {

    long mac = new BigInteger(parts[2], 16).longValue();
    int regAdd = Integer.valueOf(parts[3], 16);
```

We parse the hexadecimal values and convert them to proper types

```
    if (regAdd < 0 || regAdd > 255) {
        SendFailResponse(os);
    }

    if (parts[0].equals("W")) {
        int regVal = Integer.valueOf(parts[4], 16);

        int index = db.findDevice(mac);
        if (index != -1) {
            System.out.println("Device found at index: "+ index);
            db.setReg(index, regAdd, regVal);
        } else {
            System.out.println("New device added!");
            int indexNew = db.addDevice(mac);
            db.setReg(indexNew, regAdd, regVal);
        }

        SendResponse("A", parts[1], parts[2], parts[3], parts[4], os);
    }
```

Example: **New App Protocol - Java Code: Server Side** – `(client-serverDB-stream)` Part 7/9

Add a new entry to the DB or update an existing entry

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – (client-serverDB-stream) Part 8/9

Enquire the database for the value of the requested register

```
 if (parts[0].equals("R")) {
        int index = db.findDevice(mac);
        if (index != -1) {
            System.out.println("Device found at index: "+ index);

            String regVal = String.format("%04X", db.getReg(index, regAdd));
            SendResponse("A", parts[1], parts[2], parts[3], regVal, os);
        } else {
            SendFailResponse(os);
        }
    }

} else {
    SendFailResponse(os);
}
```

Implementing an Application Layer Protocol

Example: **New App Protocol - Java Code: Server Side** – `(client-serverDB-stream)` Part 9/9

```
 } catch (IOException e) {
            line = this.getName();
            System.out.println("IO Error/ Client " + line +
                                              " terminated abruptly");
        } finally {
            try {
                if (is != null) is.close();
                if (os != null) os.close();
                if (s != null) s.close();
                System.out.println("Connection Closed.");
            } catch (IOException ie) {
                System.out.println("Socket Close Error");
            }
        }//end finally
    }
}
```

Implementing an Application Layer Protocol

Programming Edge Devices

- When a connection is established, instead of transmitting data through **streams**, we can send data using **packets**
- **Note:**
  - Using streams, we just write to the stream without worrying about packet size, packet preparation, and deletion
  - Using packets, we must be careful about the size of data written to the packet, and the buffer must be released after the packet is sent
- At the beginning of your application, when you run the `wiced_init()` function, on the console you will see the message "`Creating Packet pools`"
- The packet pools are just RAM buffers which store either incoming packets from the network (i.e. receive packets) or will hold outgoing packets which have not yet been sent (i.e. transmit packets)
- **By default, there are two receive packets and two transmit packets, but this can be configured in your firmware**
- If you run out of receive packets then TCP packets will be tossed
- If you run out of transmit packets you will get an error when you try to create one

Implementing an Application Layer Protocol

Programming Edge Devices

- Each packet in the buffer contains:
  - An allocation reference count
  - The raw data
  - A pointer to the start of the data
  - A pointer to the end of the data
  - The TCP packet overhead

Packet Buffer				
Type	Ref Count	Data Pointer		Buffer
		Start	End	
R	0	null	null	
R	0	null	null	
T	0	null	null	
T	0	null	null	

- A packet starts its life unallocated, and as such, the reference count is 0

Implementing an Application Layer Protocol

Programming Edge Devices

- When you want to send a message, you call `wiced_tcp_packet_create()` which has the prototype of:

```
wiced_result_t wiced_packet_create_tcp(
        wiced_tcp_socket_t* socket, uint16_t content_length,
                wiced_packet_t** packet, uint8_t** data,
                        uint16_t* available_space );
```

- **This function will look for an unallocated packet (i.e., the reference count == 0) and assigns it to you**

- `socket`: A pointer to the socket that was previously created by `wiced_tcp_connect()`
- `content_length`: How many bytes of data you plan to put in the packet

Implementing an Application Layer Protocol

Programming Edge Devices

```
wiced_result_t wiced_packet_create_tcp(
        wiced_tcp_socket_t* socket, uint16_t content_length,
                wiced_packet_t** packet, uint8_t** data,
                        uint16_t* available_space );
```

- `packet`: a pointer to a packet pointer
  - This enables the create function to give you a pointer to the packet structure in the RAM
  - To use it, you declare: `wiced_packet_t *myPacket;` Then when you call the `wiced_packet_create_tcp()` you pass a pointer to your pointer e.g. `&myPacket`
  - When the function returns, `myPacket` will then point to the allocated packet in the packet pool

Implementing an Application Layer Protocol

Programming Edge Devices

```
wiced_result_t wiced_packet_create_tcp(
        wiced_tcp_socket_t* socket, uint16_t content_length,
                wiced_packet_t** packet, uint8_t** data,
                        uint16_t* available_space );
```

- `data`: a pointer to a `uint8_t` pointer
  - To use it, you declare: `uint8 *myData;` then when you call the `wiced_packet_create_tcp()` you pass a pointer to your pointer e.g. `&myData`
  - When the function returns, **`myData` pointer will then point to the place inside of the packet buffer where you need to store your data**

- `available_space`: This is a pointer to an integer that will be set to **the maximum amount of data that you are allowed to store inside of the packet**
  - It works like the previous two in that the function changes the instance of your integer

Implementing an Application Layer Protocol

Programming Edge Devices

- Once you have created the packet, you need to:
  - **Copy your data** into the packet in the correct place i.e. using `memcpy()` to copy to the data location that was provided to you
  - **Tell the packet where the end of your data is** by calling `wiced_packet_set_data_end()`
  - **Send the data** by calling `wiced_tcp_send_packet()`
    - This function will increment the reference count (so it will be 2 after calling this function)
  - **Release control of the packet** by calling `wiced_packet_delete()`
    - This function will decrement the reference count
  - Once the packet is actually sent by the TCP/IP stack, it will decrement the reference count again, which will make the packet buffer available for reuse

Implementing an Application Layer Protocol

Programming Edge Devices

- After the call to `wiced_tcp_packet_create_tcp`:
  - The pointer `myPacket` points to the packet in the packet pool that is allocated to you
  - `available_space` will be set to the maximum number of bytes that you can store in the packet (about 1500)
    - You should make sure that you don't copy more into the packet than it can hold
  - The pointer `data` will point to the place where you need to copy your message

Implementing an Application Layer Protocol

- Here is how we need to change the `sendData()` method

```
wiced_packet_t* tx_packet; //Pointer to the allocated packet
uint8_t *tx_data; //Pointer to the payload of the packet
uint16_t available_data_length; //How much data you can insert
```

Allocate a TCP packet from the pool

```
wiced_packet_create_tcp(&socket, strlen(sendMessage),
        &tx_packet, (uint8_t**)&tx_data, &available_data_length);

memcpy(tx_data, sendMessage, strlen(sendMessage));
```

Put data in the packet

Set the size of data in a packet
If data has been added to a packet, this function should be called to ensure the packet length is updated

```
wiced_packet_set_data_end(tx_packet,
                (uint8_t*)&tx_data[strlen(sendMessage)]);


wiced_tcp_send_packet(&socket, tx_packet);
```

Send TCP data packet

```
wiced_packet_delete(tx_packet);
```

Releases a packet that is in use, back to the main packet pool, allowing re-use

Implementing an Application Layer Protocol

Programming Edge Devices

- To receive data we use:

```
wiced_tcp_receive( wiced_tcp_socket_t* socket,
      wiced_packet_t** packet, uint32_t timeout )
```

- The `wiced_packet_t` ** packet means that you need to give it a pointer of type `wiced_packet_t` so that the receive function can set your pointer to point to the TCP packet in the packet pool

- This function will also increment the reference count of that packet
- When you are done, you need to delete the packet by calling `wiced_packet_delete`

Implementing an Application Layer Protocol

Programming Edge Devices

- Finally, you can get the actual TCP packet data by calling *wiced_packet_get_data* which has the following prototype:

```
wiced_result_t wiced_packet_get_data( wiced_packet_t* packet,
       uint16_t offset, uint8_t** data,
                uint16_t* fragment_available_data_length,
                      uint16_t *total_available_data_length )
```

- This function is designed to let you grab pieces of the packet, hence the offset parameter
- To get your data you need to pass a pointer to a uint8_t pointer
- The function will update your pointer to point to the raw data in the buffer

Implementing an Application Layer Protocol

- Here is how we need to change the `sendData()` method

```
wiced_packet_t *rx_packet;


result = wiced_tcp_receive(&socket, &rx_packet, 500);

if(result == WICED_SUCCESS)
{
    char *rbuffer;
    uint16_t request_length;


    wiced_packet_get_data( rx_packet, 0, (uint8_t**) &rbuffer,
                           &request_length, &available_data_length );

    if(available_data_length < 30)
    {
         rbuffer[available_data_length]=0;
         WPRINT_APP_INFO(("Server Response=%s\n",rbuffer));
    }
    else
        WPRINT_APP_INFO(("Malformed response\n"));

        wiced_packet_delete(rx_packet);
    }
```

Attempts to receive a TCP data packet from the remote host
If a packet is returned successfully, then ownership of it has been transferred to the caller, and it must be released as soon as it is no longer needed

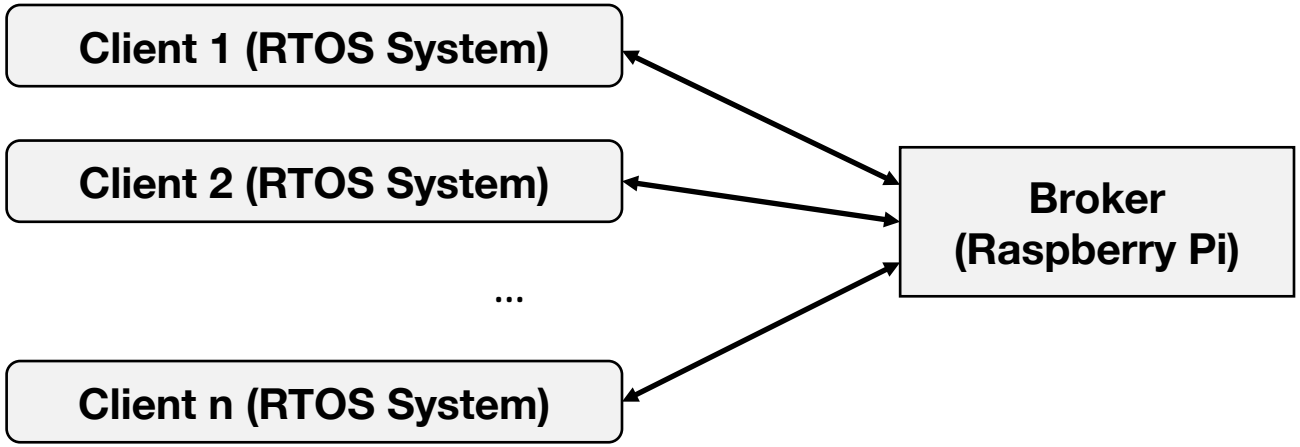
Retrieves a data buffer pointer for a given packet handle at a particular offset

MQTT Project

Software Development using WICED® Studio and Open-Source Tools

MQTT Project

- We are interested to implement the following system:



- All clients subscribe to topic `light/led1`
- When button1 is pressed, the light status is changed and a messages is published to MQTT broker
- So the light status of all the clients is affected by other clients

MQTT Project

Server Side

Mosquitto

An Open Source MQTT v3.1/v3.1.1 Broker

- We use the **open-source MQTT implementation** from Eclipse
- This implementation is called **Mosquitto**

- Install **Mosquitto** on Raspberry Pi
- First import the repository package signing key:

```
wget http://repo.mosquitto.org/debian/mosquitto-
                                        repo.gpg.key
sudo apt-key add mosquitto-repo.gpg.key
```

- To make the repository available to apt: `cd /etc/apt/sources.list.d/`
- Then run: `sudo wget http://repo.mosquitto.org/debian/mosquitto-jessie.list`
- Update apt information: `sudo apt-get update`
- Install Mosquitto: `sudo apt-get install mosquitto`

Server Side

Configuring the server

- You can find the documentation of the configuration file at: `https://mosquitto.org/man/mosquitto-conf-5.html`
- Open the file as follows: `sudo nano /etc/mosquitto/mosquitto.conf`
- Delete the line `include_dir /etc/mosquitto/conf.d` and add your configuration

- We perform the following configuration

`persistence [ true | false ]`

- If true, connection, subscription and message data will be written to the disk in `mosquitto.db` at the location dictated by `persistence_location`
- When mosquitto is restarted, it will reload the information stored in `mosquitto.db`
- The data will be written to disk when mosquitto closes and also at periodic intervals as defined by `autosave_interval`

MQTT Project

Server Side

`allow_anonymous [ true | false ]`
Boolean value that determines whether clients that connect without providing a username are allowed to connect

`password_file` *`file path`*
Set the path to a password file. If defined, the contents of the file are used to control client access to the broker. If mosquitto is compiled without TLS support, then the password file should be a text file with each line in the format "username:password"

Configuration file:

```
pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto/

log_dest file /var/log/mosquitto/mosquitto.log

#include_dir /etc/mosquitto/conf.d

allow_anonymous false
password_file /etc/mosquitto/pwfile
listener 1883
```

MQTT Project

Server Side

- We create password file using the `mosquitto_passwd` command
- The documentation of this file can be found at: `https://mosquitto.org/man/mosquitto_passwd-1.html`

- We create the password file and add a username/password to that:

```
sudo mosquitto_passwd  -c  /etc/mosquitto/pwfile/  iotstudent
```

- Then we enter the password: "`coen`"

- Next, we need to restart the Pi: `sudo reboot`

- We can run the MQTT server by: `sudo mosquitto -c /etc/mosquitto/mosquitto.conf`

- You can subscribe to a topic using: `mosquitto_sub  -t  light/#`
- You can publish to a topic using: `mosquitto_pub  -t  'light/led1' –m 'LIGHT OFF'`

MQTT Project

Client Side

Pseudo-code

```
mqtt_connection_event_cb ()
{
    if (publish message received from the broker) {
         print the message;
         change LED status accordingly;
    }
}


application_start() {
    do {
         establish MQTT connection;
         if (connection was unsuccessful) break;

         While(1)
         {
            wait for button semaphore;
            publish the button status;
            if (publish was unsuccessful) break;
         }
         close the connection;
       }

    house keeping;
}
```

MQTT Project

Client Side

MQTT Client
(subscriber_publisher) - Part 1

```
#include "wiced.h"
#include "mqtt_api.h"

#define MQTT_BROKER_ADDRESS                "iotserver2"
#define WICED_TOPIC                        "light/led1"
#define CLIENT_ID                          "Behnam_Dezfouli"
#define MQTT_REQUEST_TIMEOUT               (5000)
#define MQTT_DELAY_IN_MILLISECONDS         (1000)
#define MQTT_PUBLISH_RETRY_COUNT           (3)
#define MSG_ON                             (3)
#define MSG_OFF                            "LIGHT ON-"
                                           "LIGHT OFF-"


static wiced_ip_address_t                  broker_address;

static wiced_mqtt_event_type_t             received_event;

static wiced_semaphore_t                   event_semaphore;
static wiced_semaphore_t                   wake_semaphore;
static uint8_t                             pub_in_progress = 0;
```

The MQTT event we just received

MQTT Project

Client Side

- Using this function, we can check if a desired event happens within a given deadline
- `WICED_ERROR` happens when no event is received or the event is not what we want

MQTT Client
(`subscriber_publisher`) - Part 2

```
/*
 * A blocking call to an expected event.
 */
static wiced_result_t wait_for_response(
        wiced_mqtt_event_type_t expected_event, uint32_t timeout )
{
    if ( wiced_rtos_get_semaphore( &event_semaphore, timeout ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }

    else
    {
        if (expected_event != received_event )
        {
            return WICED_ERROR;
        }
    }
    return WICED_SUCCESS;
}
```

- The `event_semaphore` is set when we receive any event

- Check if the received event is what we are waiting for
- `received_event` is a global variable

MQTT Project

Client Side

MQTT Client
(subscriber_publisher) - Part 3

- This function handles the MQTT events received
- This function is passed to the `mqtt_conn_open` function to serve as the event callback function, which is used for notifying the events from library

```
static wiced_result_t mqtt_connection_event_cb( wiced_mqtt_object_t mqtt_object,
                                                wiced_mqtt_event_info_t *event )
{
    static char data[30];

    switch ( event->type )
    {
    case WICED_MQTT_EVENT_TYPE_SUBCRIBED:
    {
        WPRINT_APP_INFO(( "\nSubscription acknowledged!\n" ));
        received_event = event->type;
        wiced_rtos_set_semaphore( &event_semaphore );
    }
        break;
    case WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS:
    case WICED_MQTT_EVENT_TYPE_DISCONNECTED:
    case WICED_MQTT_EVENT_TYPE_PUBLISHED:
    case WICED_MQTT_EVENT_TYPE_UNSUBSCRIBED:
    {
        received_event = event->type;
        wiced_rtos_set_semaphore( &event_semaphore );
    }
        break;
```

See the list of events in the next slide

We set the `recived_event` global variable and the semaphore so that function `wait_for_response()` can check event validity

MQTT Project

Client Side

Documentation

`WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS`	Event sent when broker accepts CONNECT request
`WICED_MQTT_EVENT_TYPE_DISCONNECTED`	Event sent when broker accepts DISCONNECT request, or when there is any network issue
`WICED_MQTT_EVENT_TYPE_PUBLISHED`	Event sent for QoS-1 and QoS-2 for the published when successfully delivered. No event will be sent for QOS-0.
`WICED_MQTT_EVENT_TYPE_SUBCRIBED`	Event sent when broker accepts SUBSCRIBED request
`WICED_MQTT_EVENT_TYPE_UNSUBSCRIBED`	Event sent when broker accepts UNSUBSCRIBED request
`WICED_MQTT_EVENT_TYPE_PUBLISH_MSG_RECEIVED`	Event sent when PUBLISH message is received from the broker for a subscribed topic

MQTT Project

Client Side

Documentation

```
/* MQTT Event info */
typedef struct wiced_mqtt_event_info_s
{
    wiced_mqtt_event_type_t  type; /* Message event type */

    union
    {
        /* Valid only for WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS event.
        Indicates the error identified while connecting to Broker */
       wiced_mqtt_conn_err_code_t      err_code;

        /* Valid only for WICED_MQTT_EVENT_TYPE_PUBLISHED,
        WICED_MQTT_EVENT_TYPE_SUBCRIBED, WICED_MQTT_EVENT_TYPE_UNSUBSCRIBED
        events. Indicates message ID */
        wiced_mqtt_msgid_t    msgid;

        /* Valid only for WICED_MQTT_EVENT_TYPE_PUBLISH_MSG_RECEIVED event.
        Indicates the message received from Broker */
        wiced_mqtt_topic_msg_t   pub_recvd;

     } data;                    /* Event data */

} wiced_mqtt_event_info_t;
```

See the previous table

When the event type is `WICED_MQTT_EVENT_TYPE_PUBLISH_MSG_RECEIVED`, we need to get the actual received data from `pub_recvd`

MQTT Project

Client Side

Documentation

```
/**
 * Contains the message received for a topic from the Broker
 */
typedef struct wiced_mqtt_topic_msg_s
{
 /* Name of the topic associated with the message. It's not 'null' terminated */
    uint8_t*    topic;

    uint32_t    topic_len; /* Length of the topic */

    uint8_t*    data;      /* Payload of the message */

    uint32_t    data_len;  /* Length of the message payload */

} wiced_mqtt_topic_msg_t;
```

MQTT Project

Client Side

MQTT Client (`subscriber_publisher`) – Part 4

Function `mqtt_connection_event_cb()` (cont'd)

```
case WICED_MQTT_EVENT_TYPE_PUBLISH_MSG_RECEIVED:
    {
        WPRINT_APP_INFO(( "\nReceived message from broker!\n" ));
        wiced_mqtt_topic_msg_t msg = event->data.pub_recvd;
        memcpy( data, msg.data, msg.data_len );
        data[ msg.data_len + 1 ] = '\0';
        if ( !strncmp( data, "LIGHT ON", 8 ) )
        {
            wiced_gpio_output_high( WICED_SH_LED1 );
            WPRINT_APP_INFO(( "LIGHT ON\n" ));
        }
        else
        {
            wiced_gpio_output_low( WICED_SH_LED1 );
            WPRINT_APP_INFO(( "LIGHT OFF\n" ));
        }
    }
        break;
    default:
        break;
    }
    return WICED_SUCCESS;
}
```

Event sent when PUBLISH message is received from the broker for a subscribed topic

MQTT Project

Client Side

MQTT Client (`subscriber_publisher`) – Part 5

This function is called to establish a connection

```
/*
 * Open a connection and wait for MQTT_REQUEST_TIMEOUT period to receive a
connection open OK event
 */
static wiced_result_t mqtt_conn_open( wiced_mqtt_object_t mqtt_obj,
         wiced_ip_address_t *address, wiced_interface_t interface,
            wiced_mqtt_callback_t callback, wiced_mqtt_security_t *security )
{
    wiced_mqtt_pkt_connect_t conninfo;
    wiced_result_t ret = WICED_SUCCESS;

    memset( &conninfo, 0, sizeof( conninfo ) );
    conninfo.port_number = 1883;
    conninfo.mqtt_version = WICED_MQTT_PROTOCOL_VER4;
    conninfo.clean_session = 1;
    conninfo.client_id = (uint8_t*) CLIENT_ID;
    conninfo.keep_alive = 5;
    conninfo.username = (uint8_t*)"iotstudent";
    conninfo.password = (uint8_t*)"coen";
```

MQTT Project

Client Side

MQTT Client (`subscriber_publisher`) – Part 6

```
    ret = wiced_mqtt_connect( mqtt_obj, address, interface, callback,
                                                    security, &conninfo );
```

See the next slide for the documentation

```
    if ( ret != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }

    if ( wait_for_response( WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS,
                                    MQTT_REQUEST_TIMEOUT ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }

    return WICED_SUCCESS;
}
```

Wait to receive `WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS`

MQTT Project

Client Side

Documentation

```
/** Contains information related to establishing connection with Broker */
typedef struct wiced_mqtt_pkt_connect_s
{
    /* Indicates mqtt broker port number to which publisher/subscriber want to
    communicate ( 1883 as open port, 8883 as secure port) */
    uint16_t    port_number;

    /* Indicates mqtt version number. Supported versions are 3 and 4. Any value
    other than 4 will be treated as 3 (default)*/
    uint8_t     mqtt_version;

    /* Indicates keep alive interval to Broker */
    uint16_t    keep_alive;

    /* Indicates if the session to be cleanly started */
    uint8_t     clean_session;

    uint8_t*    client_id;   /* Client ID */
    uint8_t*    username;    /* User name to connect to Broker */
    uint8_t*    password;    /* Password to connect to Broker */
    uint8_t*    peer_cn;
} wiced_mqtt_pkt_connect_t;
```

MQTT Project

Client Side

Documentation

```
wiced_result_t wiced_mqtt_connect (wiced_mqtt_object_t  mqtt_obj,
        wiced_ip_address_t *    address,
        wiced_interface_t   interface,
        wiced_mqtt_callback_t   callback,
        wiced_mqtt_security_t *     security,
        wiced_mqtt_pkt_connect_t *  conninfo
    )
```

Establishes connection with MQTT broker

NOTE: This is an asynchronous API. Connection status will be notified using callback function. `WICED_MQTT_EVENT_TYPE_CONNECTED` event will be sent using callback function

Parameters

- **`mqtt_obj`**: Contains address of a memory location which is passed during MQTT init
- **`address`**: IP address of the Broker
- **`interface`**: Network interface to be used for establishing connection with Broker
- **`callback`**: Event callback function which is used for notifying the events from library
- **`security`**: Security related information for establishing secure connection with Broker. If NULL, connection with Broker will be unsecured.
- **`conninfo`**: MQTT connect message related information

Returns

- **`wiced_result_t`**
- NOTE: Allocate memory for `conninfo->client_id`, `conninfo->username`, `conninfo->password` in non-stack area. And free/resuse them after getting event `WICED_MQTT_EVENT_TYPE_CONNECT_REQ_STATUS` or `WICED_MQTT_EVENT_TYPE_DISCONNECTED`

MQTT Project

Client Side

This function is called to subscribe to a topic

MQTT Client
(`subscriber_publisher`) – Part 7

```
/*
 * Subscribe to WICED_TOPIC and wait for 5 seconds to receive an ACK.
 */
static wiced_result_t mqtt_app_subscribe( wiced_mqtt_object_t mqtt_obj,
                                          char *topic, uint8_t qos )
{
    wiced_mqtt_msgid_t pktid;
    pktid = wiced_mqtt_subscribe( mqtt_obj, topic, qos );

    if ( pktid == 0 )
    {
        return WICED_ERROR;
    }

    if ( wait_for_response( WICED_MQTT_EVENT_TYPE_SUBCRIBED,
                            MQTT_REQUEST_TIMEOUT ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }

    return WICED_SUCCESS;
}
```

Wait to receive `WICED_MQTT_EVENT_TYPE_SUBCRIBED`

MQTT Project

Client Side

Documentation

Subscribe for a topic with MQTT Broker

```
 *
 * NOTE: This is an asynchronous API. Subscribe status will be notified
 * using callback function.
 * WICED_MQTT_EVENT_TYPE_SUBCRIBED event will be sent using callback function
 *
 * @param[in] mqtt_obj          : Contains address of a memory location which is
 * passed during MQTT init
 * @param[in] topic             : Contains the topic to be subscribed to
 * @param[in] qos               : QoS level to be used for receiving the message
 * on the given topic
 *
 * @return wiced_mqtt_msgid_t   : ID for the message being subscribed
 * NOTE: Allocate memory for topic in non-stack area.
 *       And free/resuse them after getting event WICED_MQTT_EVENT_TYPE_SUBCRIBED
 *       or WICED_MQTT_EVENT_TYPE_DISCONNECTED for given message ID
(wiced_mqtt_msgid_t)
 */
wiced_mqtt_msgid_t wiced_mqtt_subscribe( wiced_mqtt_object_t mqtt_obj,
                                         char *topic, uint8_t qos );
```

MQTT Project

Client Side

This function is called to publish to a topic

MQTT Client
(subscriber_publisher) – Part 8

```
/*
 * Publish (send) message to WICED_TOPIC and wait for 5 seconds to receive a
PUBCOMP (as it is QoS=2).
 */
static wiced_result_t mqtt_app_publish( wiced_mqtt_object_t mqtt_obj,
                  uint8_t qos, uint8_t *topic, uint8_t *data, uint32_t data_len )
{
    wiced_mqtt_msgid_t pktid;

    pktid = wiced_mqtt_publish( mqtt_obj, topic, data, data_len, qos );

    if ( pktid == 0 )
    {
        return WICED_ERROR;
    }

    if ( wait_for_response( WICED_MQTT_EVENT_TYPE_PUBLISHED,
                            MQTT_REQUEST_TIMEOUT ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }
    return WICED_SUCCESS;
}
```

Wait to receive event
WICED_MQTT_EVENT_TYPE_PUBLISHED

MQTT Project

Client Side

MQTT Client
(`subscriber_publisher`) – Part 9

This function is called to close the MQTT connection

```
/*
 * Close a connection and wait for 5 seconds to receive a connection close OK
event
 */
static wiced_result_t mqtt_conn_close( wiced_mqtt_object_t mqtt_obj )
{
    if ( wiced_mqtt_disconnect( mqtt_obj ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }
    if ( wait_for_response( WICED_MQTT_EVENT_TYPE_DISCONNECTED,
                                    MQTT_REQUEST_TIMEOUT ) != WICED_SUCCESS )
    {
        return WICED_ERROR;
    }
    return WICED_SUCCESS;
}
```

MQTT Project

Client Side

```
wiced_mqtt_msgid_t wiced_mqtt_publish (
        wiced_mqtt_object_t mqtt_obj, uint8_t *topic,
                uint8_t *data,  uint32_t data_len, uint8_t qos
)
```

Publish message to MQTT Broker on the given Topic.

- NOTE: This is **an asynchronous API**. Publish status will be notified using using callback function. `WICED_MQTT_EVENT_TYPE_PUBLISHED` event will be sent using callback function

Parameters

- **`mqtt_obj`**: Contains address of a memory location which is passed during MQTT init
- **`topic`**: Contains the topic on which the message to be published
- **`message`**: Pointer to the message to be published
- **`msg_len`**: Length of the message pointed by 'message' pointer
- **`qos`**: QoS level to be used for publishing the given message

Returns

`wiced_mqtt_msgid_t`: ID for the message being published

MQTT Project

Client Side

MQTT Client (`subscriber_publisher`) – Part 10

- This is the callback function of the button interrupt
- When the button is pressed, this function sets a semaphore that unlocks MQTT publishing

```
static void button_callback( void* arg )
{
    if(pub_in_progress == 0)
    {
        pub_in_progress = 1;
        wiced_rtos_set_semaphore( &wake_semaphore );
    }
}
```

MQTT Project

Client Side

The main thread

MQTT Client
(subscriber_publisher) – Part 11

```
void application_start( void )
{
    wiced_mqtt_object_t   mqtt_object;
    wiced_result_t        ret = WICED_SUCCESS;
    int                   connection_retries = 0;
    int                   pub_sub_retries = 0;
    int                   count = 0;
    char                  msg[30];

    wiced_init( );


    /* Bring up the network interface */
    ret = wiced_network_up( WICED_STA_INTERFACE,
                            WICED_USE_EXTERNAL_DHCP_SERVER, NULL );

    if ( ret != WICED_SUCCESS )
    {
        WPRINT_APP_INFO( ( "\nNot able to join the requested AP\n\n" ) );
        return;
    }

    /* configure push button to publish a message */
    wiced_gpio_input_irq_enable( WICED_SH_MB1, IRQ_TRIGGER_RISING_EDGE,
                                 button_callback, NULL );
```

The MQTT object is used for connection establishment, subscribing, publishing, and closing connection

Client Side

MQTT Client
(subscriber_publisher) – Part 12

```
/* Allocate memory for MQTT object*/
mqtt_object = (wiced_mqtt_object_t) malloc(
                        WICED_MQTT_OBJECT_MEMORY_SIZE_REQUIREMENT );
if ( mqtt_object == NULL )
{
    WPRINT_APP_ERROR("Don't have memory to allocate for MQTT object...\n");
    return;
}

WPRINT_APP_INFO( ( "Resolving IP address of MQTT broker...\n" ) );
ret = wiced_hostname_lookup( MQTT_BROKER_ADDRESS, &broker_address, 10000,
                                                    WICED_STA_INTERFACE);

WPRINT_APP_INFO(("Resolved Broker IP: %u.%u.%u.%u\n\n",
        (uint8_t)(GET_IPV4_ADDRESS(broker_address) >> 24),
        (uint8_t)(GET_IPV4_ADDRESS(broker_address) >> 16),
        (uint8_t)(GET_IPV4_ADDRESS(broker_address) >> 8),
        (uint8_t)(GET_IPV4_ADDRESS(broker_address) >> 0)));

if ( ret == WICED_ERROR || broker_address.ip.v4 == 0 )
{
    WPRINT_APP_INFO(("Error in resolving DNS\n"));
    return;
}
```

MQTT Project

Client Side

MQTT Client (subscriber_publisher) – Part 13

```
wiced_rtos_init_semaphore( &wake_semaphore );
wiced_mqtt_init( mqtt_object );
wiced_rtos_init_semaphore( &event_semaphore );

do
{
```

Whenever the inner loop fails, this loop retries connection establishment

```
    WPRINT_APP_INFO(("[MQTT] Opening connection..."));
    do
    {
        ret = mqtt_conn_open( mqtt_object, &broker_address,
                              WICED_STA_INTERFACE, mqtt_connection_event_cb, NULL );

        connection_retries++ ;
    } while ( ( ret != WICED_SUCCESS ) && ( connection_retries <
                                WICED_MQTT_CONNECTION_NUMBER_OF_RETRIES ) );

    if ( ret != WICED_SUCCESS )
    {
        WPRINT_APP_INFO(("Failed connection!\n"));
        break;
    }
    WPRINT_APP_INFO(("Successful connection!\n"));
```

Note that waiting to receive a proper event occurs inside this function, which we have previously defined

MQTT Project

Client Side

MQTT Client (`subscriber_publisher`) – Part 14

```
WPRINT_APP_INFO(("[MQTT] Subscribing..."));

do
{

    ret = mqtt_app_subscribe( mqtt_object, WICED_TOPIC,
                              WICED_MQTT_QOS_DELIVER_AT_MOST_ONCE );
    pub_sub_retries++ ;

} while ( ( ret != WICED_SUCCESS ) &&
                  ( pub_sub_retries < MQTT_SUBSCRIBE_RETRY_COUNT ) );


if ( ret != WICED_SUCCESS )
{
    WPRINT_APP_INFO((" Failed subscribing!\n"));
    return;
}
```

Note that waiting to receive a proper event occurs inside this function, which we have previously defined

MQTT Project

Client Side

In this loop, we wait for a button press, and then publish to the broker

MQTT Client (`subscriber_publisher`) – Part 15

```
while ( 1 )   {

        wiced_rtos_get_semaphore( &wake_semaphore, WICED_NEVER_TIMEOUT );

        if ( pub_in_progress == 1 )
        {
            WPRINT_APP_INFO(("[MQTT] Publishing..."));
            if ( count % 2 )
            {
                strcpy(msg, MSG_ON);
                strcat(msg, CLIENT_ID);
            }
            else
            {
                strcpy(msg, MSG_OFF);
                strcat(msg, CLIENT_ID);
            }
            pub_sub_retries = 0;
           // reset pub_sub_retries to 0 before going
           // into the loop so that the next publish after a
           // failure will still work
```

MQTT Project

Client Side

Note that waiting to receive a proper event occurs inside this function, which we have previously defined

MQTT Client (`subscriber_publisher`) – Part 16

```
do
{
    ret = mqtt_app_publish( mqtt_object, WICED_MQTT_QOS_DELIVER_AT_LEAST_ONCE,
                            (uint8_t*) WICED_TOPIC, (uint8_t*) msg, strlen( msg ) );
    pub_sub_retries++ ;
} while ( ( ret != WICED_SUCCESS ) &&
                        ( pub_sub_retries < MQTT_PUBLISH_RETRY_COUNT ) );

    if ( ret != WICED_SUCCESS )
    {
      WPRINT_APP_INFO((" Failed publishing!\n"));
      break;//break the loop and reconnect
    }
    else
    {
      WPRINT_APP_INFO((" Successful publishing!\n"));
    }

      pub_in_progress = 0;
      count++ ;
 }

 wiced_rtos_delay_milliseconds( 100 );
}
```

MQTT Client
(subscriber_publisher) – Part 17

```
        pub_in_progress = 0; // Reset flag if we got a failure so that another
         // button push is needed after a failure

        WPRINT_APP_INFO(("[MQTT] Closing connection..."));
        mqtt_conn_close( mqtt_object );

        wiced_rtos_delay_milliseconds( MQTT_DELAY_IN_MILLISECONDS * 2 );
    } while ( 1 );
```

The main loop is repeated to re-establish the connection if necessary

```
    wiced_rtos_deinit_semaphore( &event_semaphore );
    WPRINT_APP_INFO(("[MQTT] Deinit connection...\n"));
    ret = wiced_mqtt_deinit( mqtt_object );
    wiced_rtos_deinit_semaphore( &wake_semaphore );
    free( mqtt_object );
    mqtt_object = NULL;

    return;
}
```