Communication Protocols Between Arduinos and Wireless WiFly Chips

Sending Data Wirelessly by Utilising Protocols

Module: **Mobile Computer Engineering**

Focus: **Communicating Between Computers**

A report originally created to fulfil the degree of

BSc (Hons) in Computer Science

Submitted 25/11/2019

Abstract

This paper utilises Arduino communication protocols in order to send data, firstly between two Arduinos, followed by a wireless connection between an Arduino and a WI-FLY chip. The paper will showcase example code that makes this possible, on both the client's computer and on the server side. Finally, the differences between various wireless communications protocols will be discussed.

Contents

Task 1 – Serial Communication Between Two Arduinos 2
Transmission Example 3
Subprocess A: Inputting Data onto Arduino #1 3
Subprocess B: Receiving Code on Arduino #2 4
Hardware Set Up 4
The Entire Program Flowchart 5
Task 2 – Configuring the WiFly Chip using the Tera Term Package 6
The XMODEM-1K Protocol 6
Connection Code Example 8
Task 3 – Connecting to the Machine's IP Address using Java Code 9
Client-Side Java Code Example 9
Server-Side Java Code Example 9
Result of Compiling Both Examples 10
Task 4 – The Differences Between 802.11n and 802.15.4 Protocols 11
Reflection 11

Task 1 – Serial Communication Between Two Arduinos

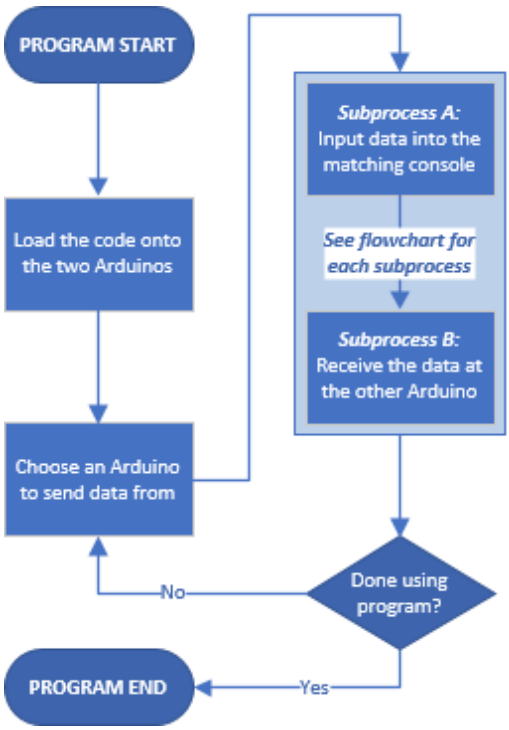


Figure 1: *The basic setup of the Arduinos.*

Serial transmission is asynchronous when using the Arduino IDE; this means that data is transmitted intermittently rather than in a steady stream. In this case, two Arduino's have the same code uploaded to them, allowing two users to send and receive data to and from each other.

How this is done is laid out in detail in the Subprocess A flowchart on the following page. In simple terms, one Arduino acts as a sender, transmitting a byte worth of binary data at a time, whilst the opposite Arduino acts as a receiver, checking to make sure that the data received is valid data.

In order to check that the data received has not been corrupted, the information is sent using a Start / Stop Protocol, which encompasses the data inside 1 start bit and 2 stop bits, alongside a calculated parity bit. Since 4 bits are being used to detect that the data received is corrected, this leaves 4 bits for the data itself, resulting in a 50% efficiently as seen below.

$$\text{Asynchronous Transmission Efficiency} = \frac{data\ transmitted}{total\ bits\ sent} \times 100 = \frac{4\ bits}{8\ bits} \times 100 = 50\%$$

Since only 50% of the data received by the second Arduino is actually data inputted by the user, this type of transmission is inefficient compared to synchronous transmission, where efficiency of up to 99.9% can be achieved by sending much more data in-between start and stop bits. Asynchronous transmissions are however much simpler to produce and inexpensive to implement. It is mainly used with Serial Ports, such as communicating between two Arduinos as seen in this particular example.

The calculated parity bit is used to help work out if the data sent has become corrupted. This is done by working out how many 1's there are being sent in the byte. If there is an odd number of 1's, the parity will be even, to ensure an even number of 1's that are sent across. If there is an even number, the parity bit will instead be odd. This is called even parity, although the inverse could be used (odd parity), so long as both the receiver and the sender are using the same. The two must also agree on the number of bits per package of data, the bits per second (equal to the Baud rate, set to 9600 in this example) and what to do when an incorrect parity bit is detected. In this example, the sender Arduino resends the previously sent byte of data. Once all the data has been sent to the receiver Arduino and concatenated together, it then converts the binary number into the message that the user inputted from the first Arduino. A breakdown of this can be seen in the following flowcharts.

Transmission Example

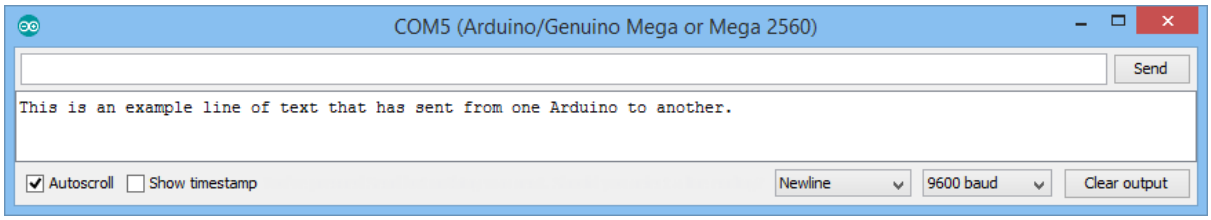


Figure 2: *An example output transmitted through the serial communication between the Arduinos.*

Subprocess A: Inputting Data onto Arduino #1

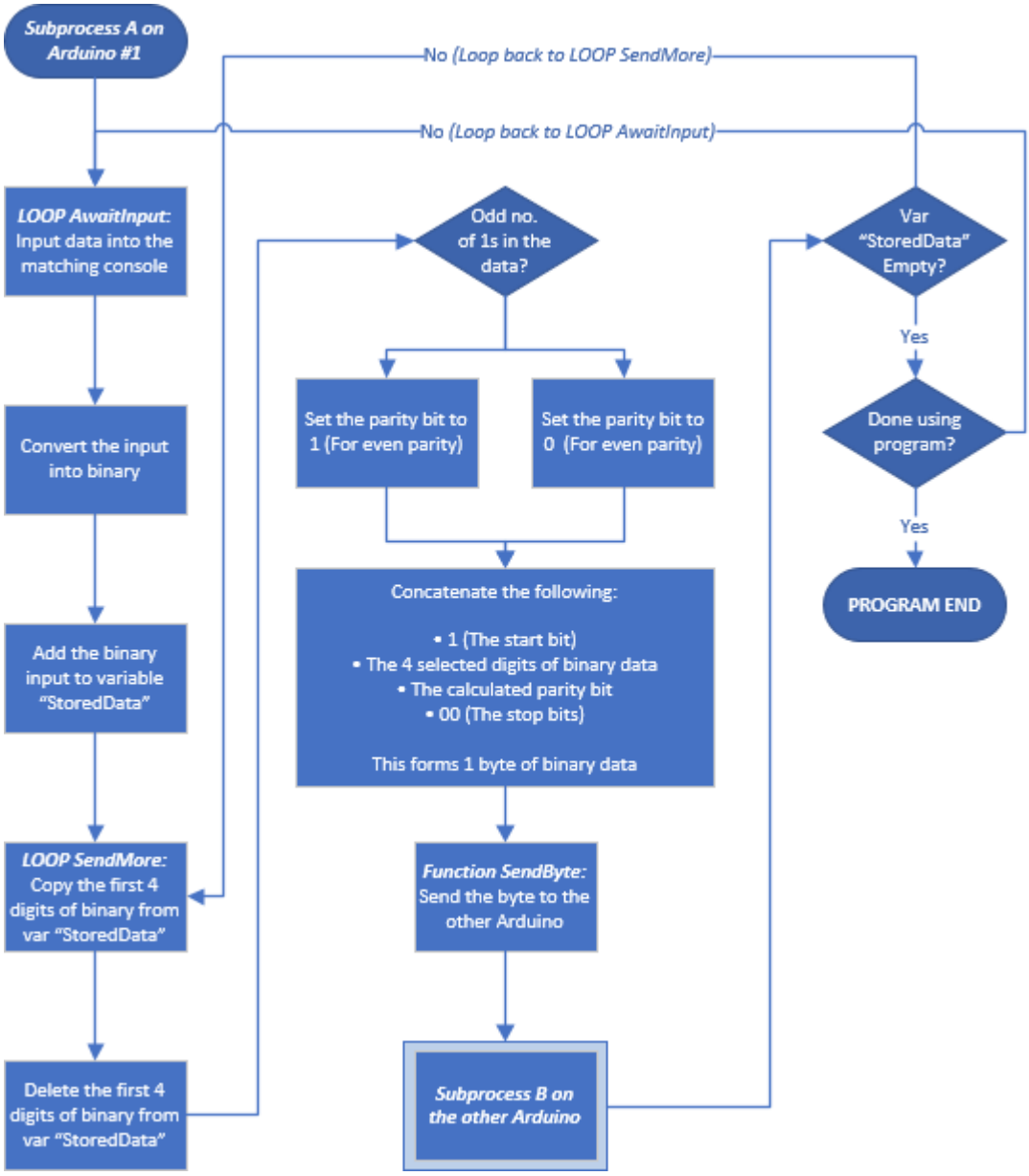


Figure 3: *A breakdown of Subprocess A, which takes the inputted data and converts it into binary.*

Subprocess B: Receiving Code on Arduino #2

Figure 4: *A breakdown of Subprocess B, which takes the received binary data and converts it back.*

Hardware Set Up

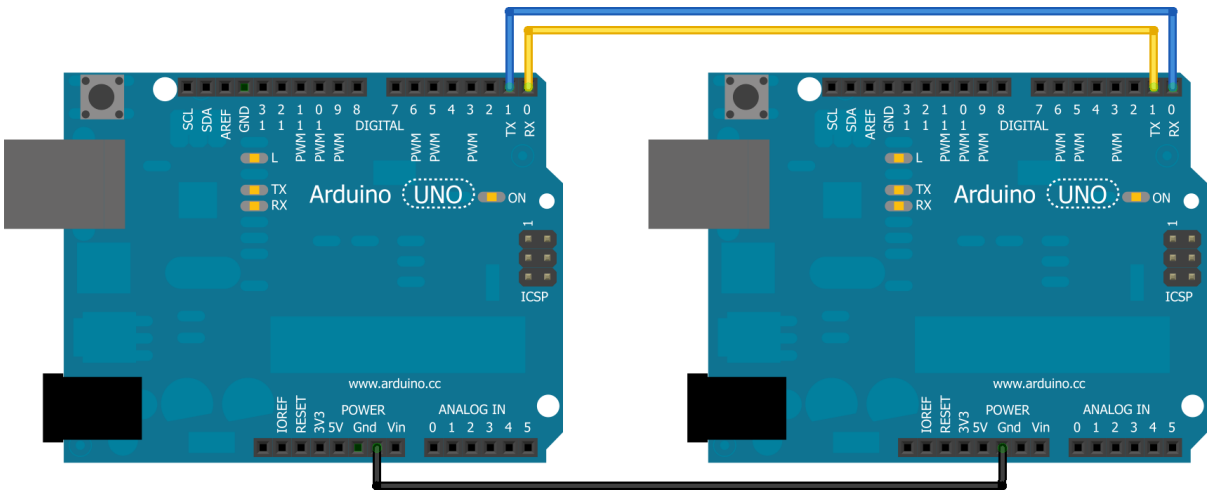


Figure 5: *Connecting two Arduino Uno's in order to send data through their serial communication.*

Individual bytes are sent and received via the TX (1) and RX (0) pins between two separate Arduinos, via the use of the UART chip on each Arduino board. Each Arduino has its TX pin connected to the other Arduinos RX pin and vice versa and are both grounded together. Both Arduinos also share the same uploaded code, so that either of them is able to send and receive user inputted data.

The Entire Program Flowchart

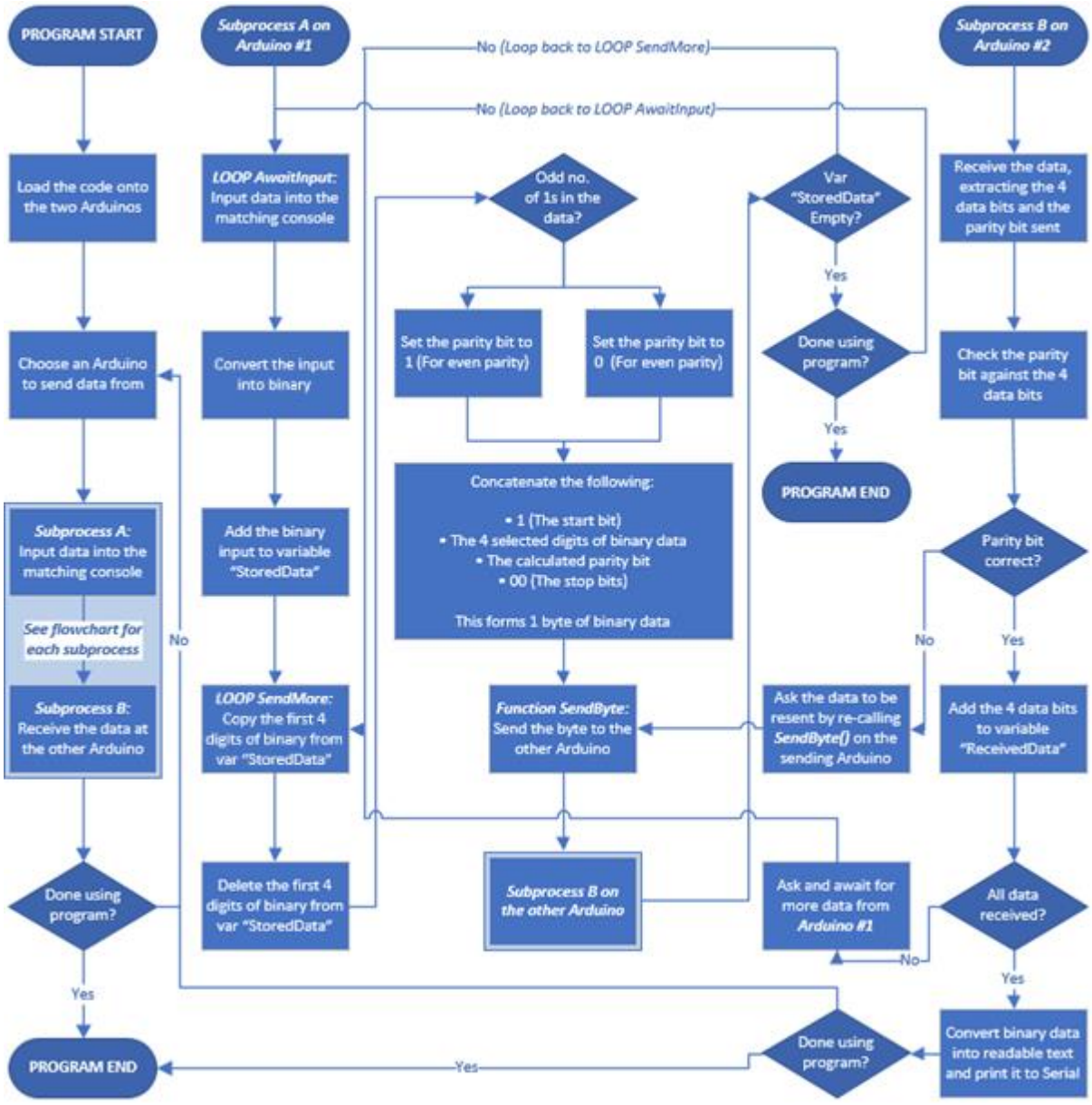


Figure 6: *An entire breakdown of the previous processes required to transmit between the Arduinos.*

Task 2 – Configuring the WiFly Chip using the Tera Term Package

The WiFly chip, based on the common 802.15.4 XBee footprint, uses a low power consumption of only 38mA whilst in use, and only 4uA in sleep mode. Additionally, the chip also has a built-in power management with programmable wakeups, which when combined with its low power allows for devices to be extremely durable and long lasting. Configured with the Tera Term package, a software implemented terminal emulator, it can be used to transmit data via the XMODEM-1K protocol.

The Tera Term package supports serial port connections, allowing data to be sent and received with the WiFly chip. It was designed for users who want to migrate their current 802.15.4 architecture to a standard TCP/IP based platform, without needing to redesign any of their existing hardware. This allows for projects set up for XBee, for example, to be moved to a standard WiFi network without needing any other new hardware.

The XMODEM-1K Protocol

Start off by connecting the RN module to the PC and open Tera Term. For an increase in download speeds, you can set the baud rate to 230400. This can be done via the following WiFly commands.

```
set uart baud 230400
set uart flow 1              // enables the UART flow control save reboot
save
reboot
```

Once the baud rate is set to 230400, enable Hardware flow control, and enter the flowing WiFly command to enable Xmodem mode.

```
xmodem <option> <filename>
```

where `<option>` is:

`u` – download firmware and set as boot image, `<filename>` is the name of the firmware (this file type can be either `.img` or `.mif`)

`c` – clean the file system before performing firmware update, deleting all the files on the flash file system (including user defined configuration files), except the current boot image and the factory default boot image.

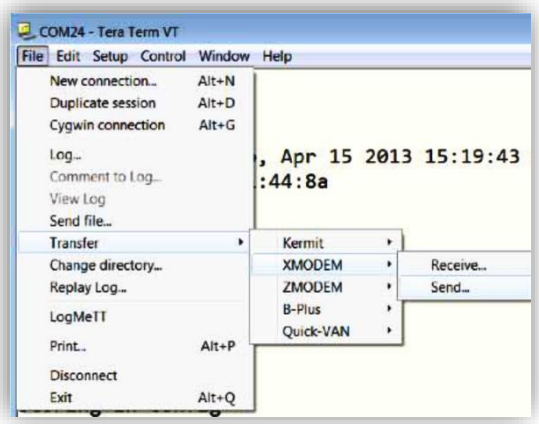
An example WiFly command is given below.

```
xmodem cu wifly7-400.mif
```

Once the command is entered, the flowing outputs appears.

```
xmodem cu wifly7-400.mif
del 4 wifly-EZX-405
del 5 config
del 6 reboot
del 8 logo.png
del 13 wps_app-EZX-131
del 14 eap_app-EZX-105
del 15 web_app-EZX-112
del 16 web_config.html
del 17 link.html
xmodem ready...
```

Once XMODEM is ready on the UART, proceed to the XMODEM file transfer option in Tera Term. To do this, select ***File > Transfer > XMODEM > Send***, as seen below in **Figure 7**. In the following options, select ***1K***, enter the `.img` or `.mif` file path, and click ***Open***, as seen below in **Figure 8**.



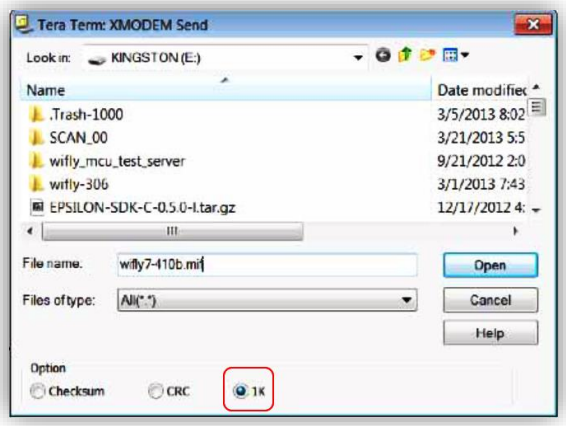


Figure 7: *The XMODEM File Transfer Option.*

Figure 8: *The 1K and file path sending options.*

The new firmware and any required web files will now download to the RN module, followed by an ***XMOD OK*** message. To check that the firmware has been downloaded correct, issue a ls command to ensure everything is working as it should. By default, the XMODEM has a 30 second timeout, which can be disabled with the command `set ftp timeout 0`.

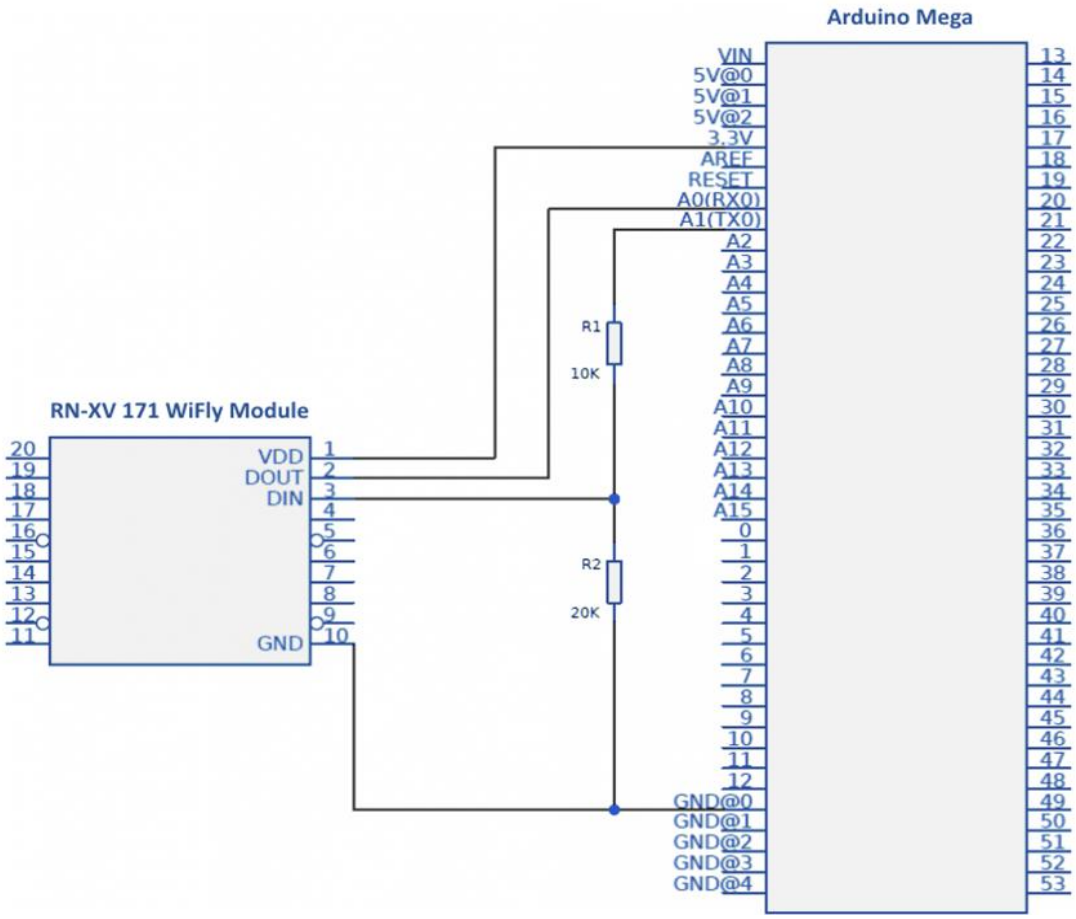


Figure 9: *A hardware configuration example of the WiFly Chip connected to an Arduino Mega.*

To access the UART (Universal Asynchronous Receiver-Transmitter) interface for communication, you can use an Arduino Mega as a USB to TTL converter by connecting the boards Reset and Ground pins, as seen in **Figure 9**. This enables you to use the RX and TX pins with the WiFly Module. The serial communications settings should match the WiFly module's stored settings; by default, the settings are 9.600 bauds, 8 bits, no parity, 1 stop bit and hardware flow controlled disabled.

Connection Code Example

```
String cmd = "";

void setup() {
	Serial.begin(9600);
	pinMode(13, OUTPUT);
	configureContact();
}

void configureContact() {

	while (Serial.available() <= 0) {
		Serial.print('X', BYTE);		// sends a capital X
		delay(1000);				// delays for 1 second
	}
}

void loop() {

	if (Serial.available() > 0) {		// incoming byte
		char incomeByte = Serial.read();
		cmd = cmd + incomeByte;
		checkCmd();
	}
}

void checkCmd() {

	if(cmd.endsWith("\r\n")) {

		if (cmd == "OFF\r\n") {
			digitalWrite(13, LOW);
		} else if (cmd == "ON\r\n") {
			digitalWrite(13, HIGH);
		}
		cmd = "";
	}
}
```

The setup function initializes serial communication and sets digital pin 13 as an output for turning the built in LED on and off. The `configureContact()` function begins the communication, where an ASCII character is sent through the serial connection until there is a reply from the WiFly module, relaying the character over a WiFi network. The main loop continues indefinitely whilst the hardware is enabled, resulting in the code constantly waiting for a byte of data from the WiFly module. When there is data, it is appended to the global variable `cmd`, which is currently an empty string object. As the global variable `cmd` is being amended, it is being checked whether it is a command or not. This is done by simply checking whether it is a line ended by carriage return and line feed character *(CR = \r and LF = \n)*. Depending on the command, the LED connected to digital pin 13 is either turned on or turned off. At the end of the check function, the global variable `cmd` is cleared, ready to be filled up by another series of character that will build the next command.

Task 3 – Connecting to the Machine's IP Address using Java Code

Client-Side Java Code Example

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

public class ConnectingClient {

    public static void main(String [] args) {

        String serverID = args[0];
        int portNum = Integer.parseInt(args[1]);

        try {
            System.out.println("Attempting connection with " + serverID + " on port number
                " + portNum);
            Socket clientID = new Socket(serverID, portNum);

            System.out.println("Connected successfully to " +
                clientID.getRemoteSocketAddress());
            OutputStream outputToServer = clientID.getOutputStream();
            DataOutputStream out = new DataOutputStream(outputToServer);

            out.writeUTF("Greetings from " + clientID.getLocalSocketAddress());
            InputStream inputFromServer = clientID.getInputStream();
            DataInputStream in = new DataInputStream(inputFromServer);

            System.out.println("Server says " + in.readUTF());
            clientID.close();

        } catch (IOException e) {

            e.printStackTrace();
      }
   }
}
```

Server-Side Java Code Example

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

public class ConnectingServer extends Thread {

    private ServerSocket serverSocket;

    public ConnectingServer(int portNum) throws IOException {
        serverSocket = new ServerSocket (portNum);
        serverSocket.setSoTimeout(20000);
    }

    public void run() {

        while(true) {

            try {
```

```
                System.out.println("Waiting for the client on port number " +
                    ServerSocket.getLocalPort() + "...");
                Socket serverID = ServerSocket.accept();

                System.out.println("Connected successfully to " +
                    serverID.getRemoteSocketAddress());
                DataInputStream in = new DataInputStream(serverID.getInputStream());

                System.out.println(in.readUTF());
                DataOutputStream out = new DataOutputStream(serverID.getOutputStream());
                out.writeUTF("thanks for connecting to " + serverID.getLocalSocketAddress()
                    + "\nEnd of transmission.");
                serverID.close();

            } catch (SocketTimeoutException s) {

                System.out.println("Socket has timed out!");
                break;

            } catch (IOException e) {

                e.printStackTrace();
                break;
            }

        }
    }

    public static void main(String [] args) {

        int port = Integer.parseInt(args[0]);

        try {

            Thread thread = new ConnectingServer(portNum);
            thread.start();

            } catch (IOException e) {

                e.printStackTrace();
        }
    }
}
```

Result of Compiling Both Examples

Server:

```
$ java ConnectingServer 1234
Waiting for the client on port number 1234...
```

Client:

```
$ java ConnectingClient localhost 1234
Attempting connection with localhost on port number 1234
Connected successfully to localhost/127.0.0.1:1234
Server says thanks for connecting to /127.0.0.1:1234
End of transmission.
```

Task 4 – The Differences Between 802.11n and 802.15.4 Protocols

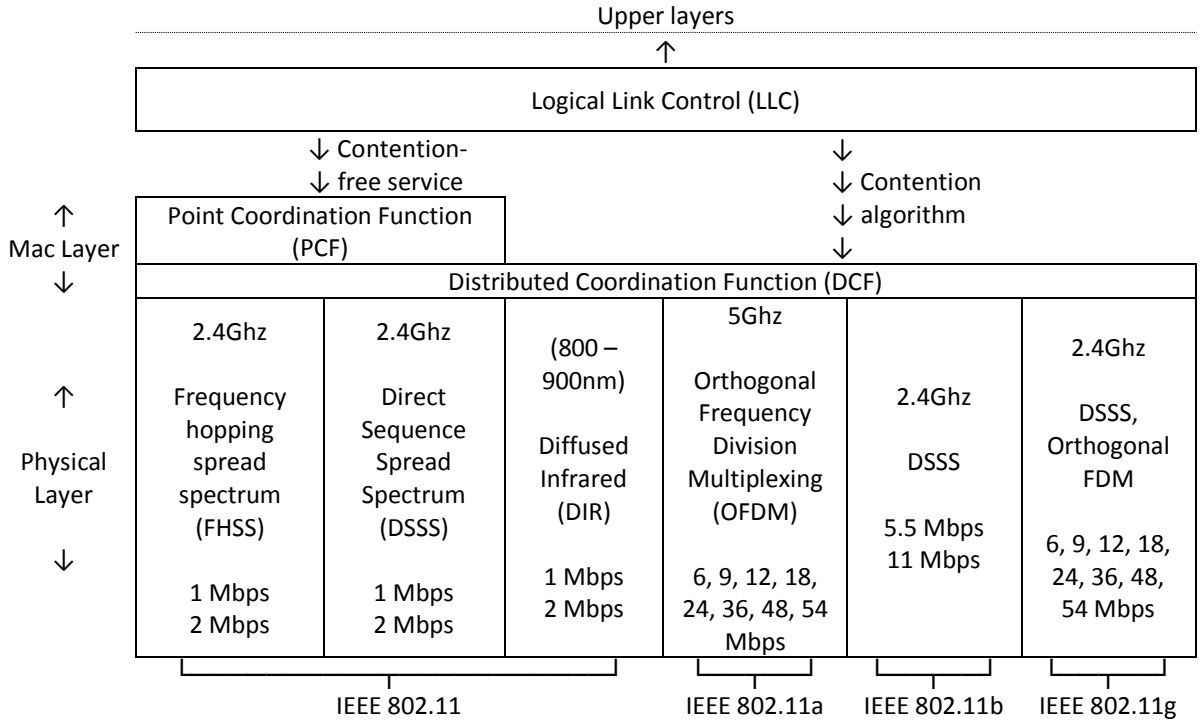


Figure 10: *The upper layers of the IEEE 802.11n Protocols alongside their slightly different standards.*

The lowest layer of the IEEE 802.11 protocol is the physical layer, defining the operating frequency bands, the supported data rates, and the details of radio transmission. All the standards have slightly different physical criterions, caused by the operating frequency band and modulation techniques used as seen in **Figure 10**. The IEEE 802.11b protocol is the most widespread, capable of delivering a throughput of up to 11 Mbps, although the observed throughput is considerably lesser at around 6 Mbps, due to possible interference from microwave ovens, cordless phones, and other such devices.

IEEE 802.15.4 in particular was designed for lightweight devices to allow even lower cost and power consumption than the 802.11 variation. This was achieved by opting for a 10-meter communications range with a decreased transfer rate of only 250 Kbps, with the ability to go as low as 20 Kbps. This trade-off allowed WPAN's (Wireless Personal Area Network) to have exceptionally low operation and manufacturing costs of various applications, without sacrificing flexibility or generality.

Reflection

Upon reflection of my work, I believe I have expanded on my knowledge of how data is transferred. While I was able to understand the logic behind the Start and Stop protocol itself, I had some trouble utilizing it in my code to begin with; developing the flow charts however made this far easier. I made sure to test my code in order to make sure that no bugs creeped up, using the flowcharts to outline the logic of my program to aid me in visualising the steps when coding the task. If I were to attempt a similar assignment in the future, I would definitely create flowcharts again, as they helped me to visualise my code. I would also research further how an Arduino could be utilised, in order to further my understanding of utilising WI-FI adapters, potentially finding a way to connect it to a Raspberry Pi for example. Overall, I am happy with the knowledge I have gained from this assignment.