# Hardware component

**1. nRF24L01 Module**

* 2.4GHz RF transceiver Module
* Operating Voltage: 3.3V
* Nominal current: 50mA
* Range : 50 – 200 feet
* Operating current: 250mA (maximum)
* Communication Protocol: SPI
* Baud Rate: 250 kbps - 2 Mbps.
* Channel Range: 125
* Maximum Pipelines/node : 6
* Low cost wireless solution

-Power consumption is around 12 mA during transmission which is even lesser then the led.

-It can operate with baud rates from 250 Kbps up to 2 Mbps

-Its range can reach up to 100 meters if used in open space and with antenna..

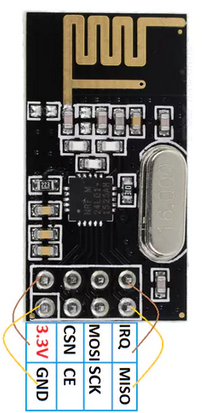
- Its can both send and receive the data simultaneously.

- Each module can communicate with up to 6 other modules.

- It uses the 2.4 GHz band.

- It can send 1->25 bytes of raw data at the transmission rate of 1 MB

- It has 125 different channels.



Three of these pins are for SPI communication and they need to be connected to the SPI pins of the MCU Master. The pins CSN and CE can be connected to any digital of the Master MCU and they are used for setting the module in standby or active mode, as well as for switching between transmit or command code. The last pin is an interrupt.

Software:

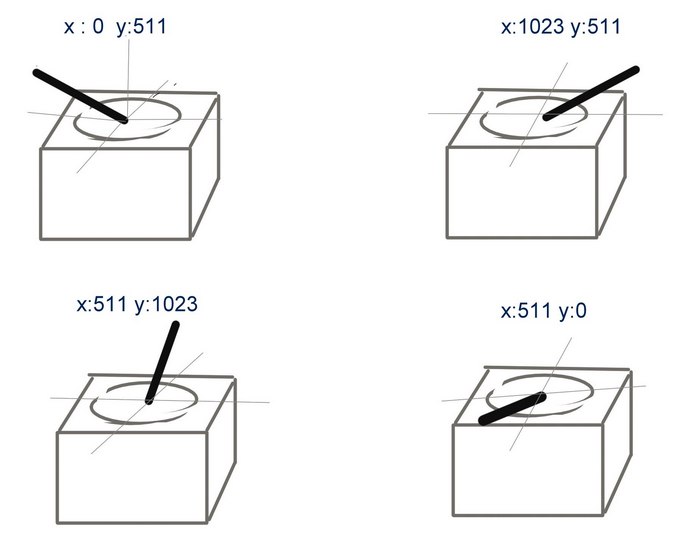
Useful library for RF module at link: [**http://tmrh20.github.io/RF24/**](http://tmrh20.github.io/RF24/)

https://github.com/nRF24/RF24

Go into Library Manager -> Search RF24L -> Install

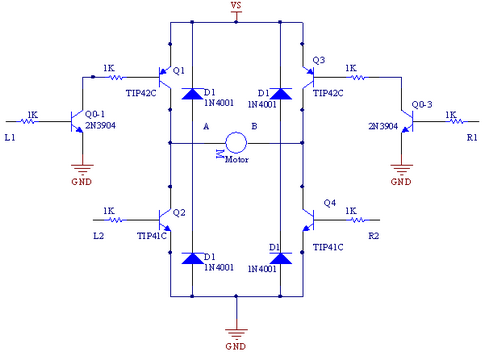
2. **JoyStick**

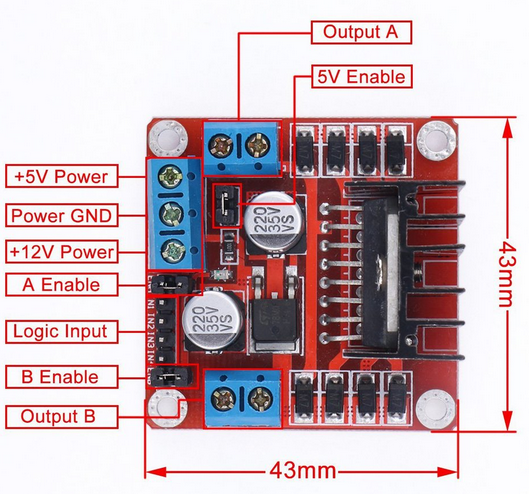
The analog joystick is similar to two potentiometer connected together, one for the vertical movement (Y-axis) and other for the horizontal movement (X-axis). The joystick also comes with a Select switch



Software: read analog signal.

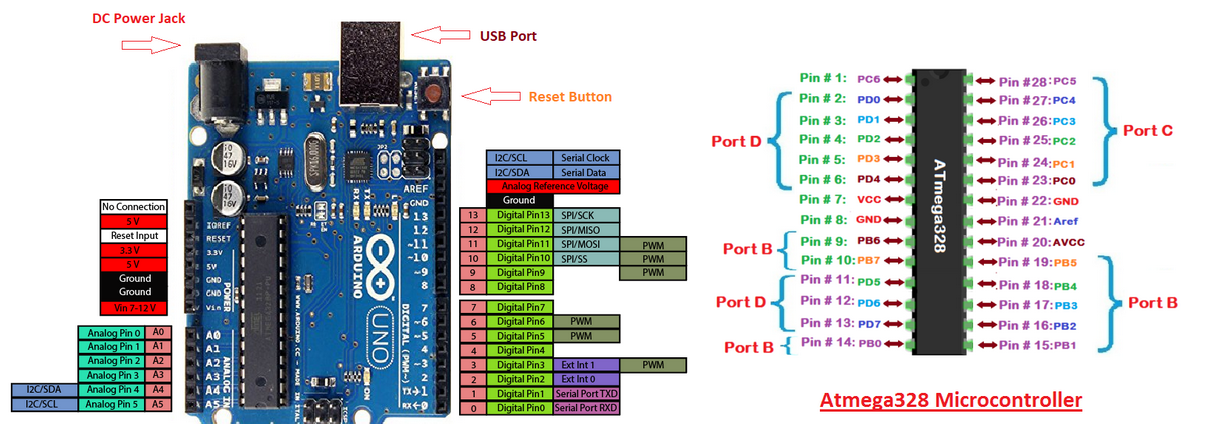
**3. L298 HBridge Circuit**.





**4. Arduino Uno**

Arduino Uno is base on AVR microcontroller called Atmega328. This controller comes with 2KB SRAM, 32 KB of flash memory, 1 KB of EEPROM. Arduino Board comes with 14 digital pins and 6 analog pins. On-Chip ADC is used to sample these pins. A 16 MHz frequency crystal oscillator is equipped on the board.



5. ESP8266

Most of the microcontroller functionality of the ESP uses exactly the same syntax as a normal Arduino, making it really easy to get started.

### Digital I/O

Just like with a regular Arduino, you can set the function of a pin using pinMode(pin, mode); where pin is the GPIO number\*, and mode can be either INPUT, which is the default, OUTPUT, or INPUT\_PULLUP to enable the built-in pull-up resistors for GPIO 0-15. To enable the pull-down resistor for GPIO16, you have to use INPUT\_PULLDOWN\_16.

(\*) NodeMCU uses a different pin mapping, read more [here](https://github.com/esp8266/Arduino/blob/master/doc/boards.md#nodemcu-09). To address a NodeMCU pin, e.g. pin 5, use D5: for instance: pinMode(D5, OUTPUT);

To set an output pin high (3.3V) or low (0V), use digitalWrite(pin, value); where pin is the digital pin, and value either 1 or 0 (or HIGH and LOW).

To read an input, use digitalRead(pin);

To enable PWM on a certain pin, use analogWrite(pin, value); where pin is the digital pin, and value a number between 0 and 1023.

You can change the range (bit depth) of the PWM output by using analogWriteRange(new\_range);

The frequency can be changed by using analogWriteFreq(new\_frequency);. new\_frequency should be between 100 and 1000Hz.

### Analog input

Just like on an Arduino, you can use analogRead(A0) to get the analog voltage on the analog input. (0 = 0V, 1023 = 1.0V).

The ESP can also use the ADC to measure the supply voltage (VCC). To do this, include ADC\_MODE(ADC\_VCC); at the top of your sketch, and use ESP.getVcc(); to actually get the voltage.  
If you use it to read the supply voltage, you can’t connect anything else to the analog pin.

### Communication

#### Serial communication

To use UART0 (TX = GPIO1, RX = GPIO3), you can use the Serial object, just like on an Arduino: Serial.begin(baud).

To use the alternative pins (TX = GPIO15, RX = GPIO13), use Serial.swap() after Serial.begin.

To use UART1 (TX = GPIO2), use the Serial1 object.

All Arduino Stream functions, like read, write, print, println, ... are supported as well.

#### I²C and SPI

You can just use the default Arduino library syntax, like you normally would.

### Sharing CPU time with the RF part

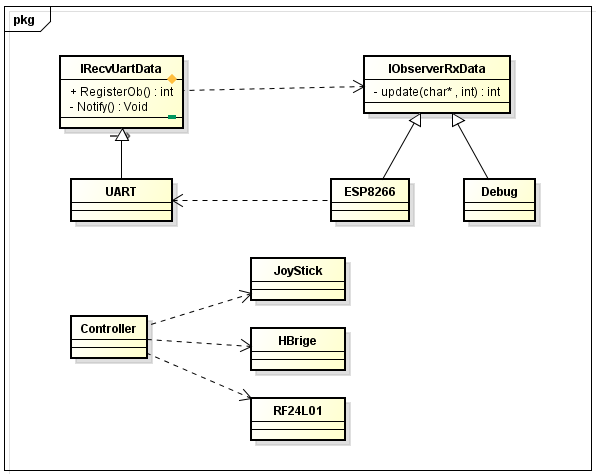
One thing to keep in mind while writing programs for the ESP8266 is that your sketch has to share resources (CPU time and memory) with the Wi-Fi- and TCP-stacks (the software that runs in the background and handles all Wi-Fi and IP connections).   
If your code takes too long to execute, and don’t let the TCP stacks do their thing, it might crash, or you could lose data. It’s best to keep the execution time of you loop under a couple of hundreds of milliseconds.

Every time the main loop is repeated, your sketch yields to the Wi-Fi and TCP to handle all Wi-Fi and TCP requests.

If your loop takes longer than this, you will have to explicitly give CPU time to the Wi-Fi/TCP stacks, by using including delay(0); or yield();. If you don’t, network communication won’t work as expected, and if it’s longer than 3 seconds, the soft WDT (Watch Dog Timer) will reset the ESP. If the soft WDT is disabled, after a little over 8 seconds, the hardware WDT will reset the chip.

From a microcontroller’s perspective however, 3 seconds is a very long time (240 million clockcycles), so unless you do some extremely heavy number crunching, or sending extremely long strings over Serial, you won’t be affected by this. Just keep in mind that you add the yield(); inside your for or while loops that could take longer than, say 100ms.

# Software Architecture



MQTT Protocol

MQTT (Message Queuing Telemetry Transport) is a publish/subscribe messaging protocol for constrained Internet of Things devices and low-bandwidth, high-latency or unreliable networks.

Because MQTT specializes in low-bandwidth, high-latency environments, it is an ideal protocol for machine-to-machine (M2M) communication.

So what is MQTT? To get a good overview simply search for “what is mqtt” or “mqtt slides”. In this article, we will briefly review the main definitions: “subscribe”, “publish”, "qos", "retain", "last will and testament" (lwt).

In an MQTT system, many nodes (called mqtt clients) connect to a mqtt server (called broker). Each client will typically register a few channels, eg "/client1/channel1", "/client1/channel2", this registration process is called **subscribe**. Each subscribing node will receive data when another client sends data to the subscribed channel. When a node sends data to a channel it is called **publish**.

* The three levels **QoS(Qualities of service)** are:
  + **QoS0** The broker/client will deliver the message once, with no confirmation (fire and forget).
  + **QoS1** The broker/client will deliver the message at least once, with confirmation required.
  + **QoS2** The broker/client will deliver the message exactly once by using a four-step handshake.

**Use cases QoS**: <https://code.google.com/p/mqtt4erl/wiki/QualityOfServiceUseCases>

Messages can be sent at any QoS level, and clients may attempt to subscribe to topics at any QoS level, which means that the client chooses the maximum QoS level they will receive. For example, if a message is published at QoS 2 and a client is subscribed with QoS 0, the message will be delivered to that client with QoS 0. If a second client is also subscribed to the same topic, but with QoS 2, then it will receive the same message but with QoS 2.

Another example could be if a client is subscribed with QoS 2 and a message is published on QoS 0, the client will receive it on QoS 0. Higher levels of QoS are more reliable, but involve higher latency and have higher bandwidth requirements.

* **Retain**:

If the RETAIN flag is set to 1, in a PUBLISH Packet sent by a Client to a Server, the Server MUST store the Application Message and its QoS, so that it can be delivered to future subscribers whose subscriptions match its topic name. When a new subscription is established, the last retained message, if any, on each matching topic name MUST be sent to the subscriber. If the Server receives a QoS 0 message with the RETAIN flag set to 1 it MUST discard any message previously retained for that topic. It SHOULD store the new QoS 0 message as the new retained message for that topic, but MAY choose to discard it at any time - if this happens there will be no retained message for that topic.

When sending a PUBLISH Packet to a Client the Server MUST set the RETAIN flag to 1 if a message is sent as a result of a new subscription being made by a Client . It MUST set the RETAIN flag to 0 when a PUBLISH Packet is sent to a Client because it matches an established subscription regardless of how the flag was set in the message it received.

A PUBLISH Packet with a RETAIN flag set to 1 and a payload containing zero bytes will be processed as normal by the Server and sent to Clients with a subscription matching the topic name. Additionally any existing retained message with the same topic name MUST be removed and any future subscribers for the topic will not receive a retained message . “As normal” means that the RETAIN flag is not set in the message received by existing Clients. A zero byte retained message MUST NOT be stored as a retained message on the Server.

If the RETAIN flag is 0, in a PUBLISH Packet sent by a Client to a Server, the Server MUST NOT store the message and MUST NOT remove or replace any existing retained message

* **LWT**:

LWT messages are not really concerned about detecting whether a client has gone offline or not (that task is handled by keepAlive messages). LWT messages are about what happens after the client has gone offline.

**A fictitious example:**

I have a sensor, which sends crucial data, but very infrequently. It has formulated a last will statement in the form of [topic: '/node/gone-offline', message: ':id'], with :id being a unique id for the sensor. I also have a emergency-subscriber for the topic 'node/gone-offline', which will send a SMS to my phone every time a message is published on that channel.

During normal operation, the sensor will keep the connection to the MQTT-broker open by sending periodic keepAlive messages interspersed with the actual sensor readings. If the sensor goes offline, the connection to the broker will time out, due to the lack of keepAlives.

This is where LWT comes in: If no LWT is specified, the broker doesn't care and just closes the connection. In our case however, the broker will execute the sensor's last will and publish the LWT-message '/node/gone-offline: :id'. The message will then be consumed to my emergency-subscriber and I will be notified of the sensor's ID via SMS so that I can check up on what's going on.

**In short:**

Instead of just closing the connection after a client has gone offline, LWT messages can be leveraged to define a message to be published by the broker on behalf of the client, since the client is offline and cannot publish anymore.

WebSocket reference at:

<https://tttapa.github.io/ESP8266/Chap14%20-%20WebSocket.html>