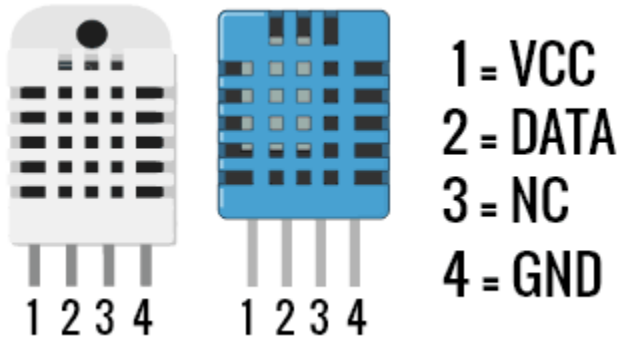
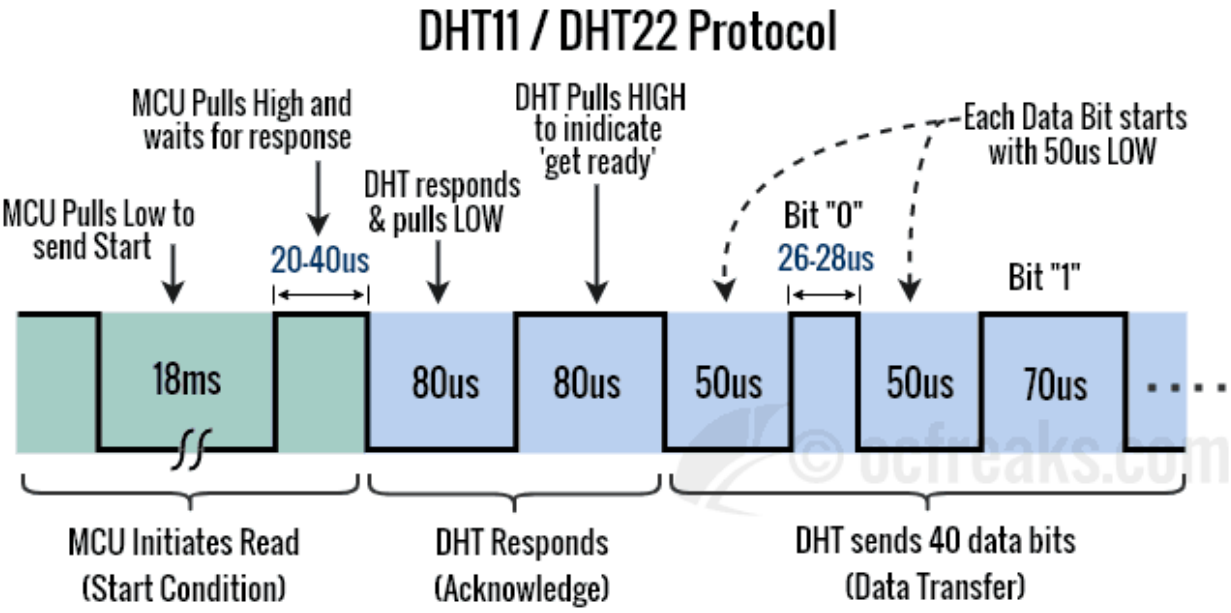
Reading Data from DTH11 or DTH22



The DATA wire used for communication between microcontroller and DHT11/DHT22 is pulled HIGH using a 4.7K or 10K pull-up resistor. This is to bring the bus in an IDLE state when there is no communication taking place. A continuous HIGH on the line denotes an IDLE state. The microcontroller acts as the bus master and hence is responsible for initiating communication (i.e. Read). DHTxx Humidity and Temperature sensor always remains as slave and responds with data when MCU asks for it. The protocol used for communication is simple and can be summarized as follows:



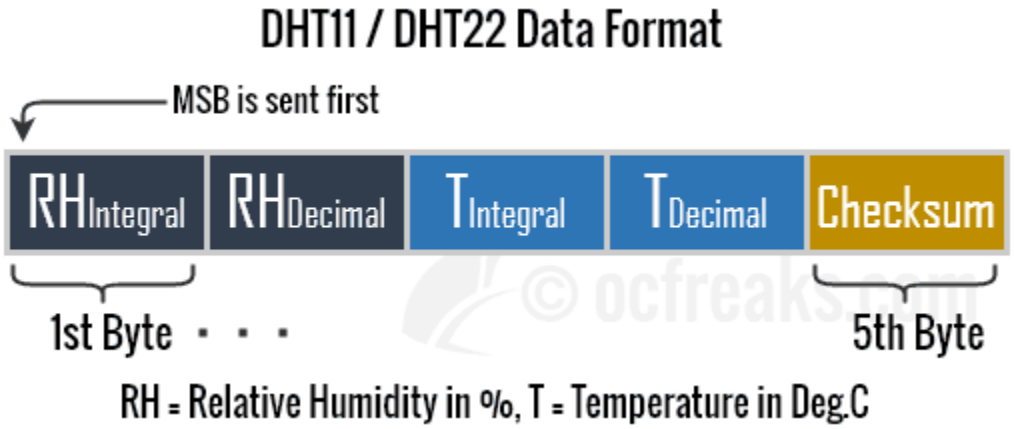
1. When the Line is IDLE the microcontroller pulls it to a LOW for 18ms.

2. After this MCU pulls it HIGH for around 20 to 40us.
3. DHTxx will detect it as a START from the MCU and respond by pulling the line LOW for 80us.
4. Next, DHTxx will pull it HIGH for 80us which indicates that it is ready to send data or “get ready”.
5. Next it will send 40 bits of Data. Each bit starts with a 50us LOW followed by 26-28us for a “0” or 70us for a “1”.
6. After communication ends, the Line is pulled HIGH by the pull-up resistor and enters IDLE state.

DHTxx Data Format

When a Humidity and Temperature sensor sends data, it sends the MSB first. The 40bits of data is divided into 5 bytes. For DHT11 sensors the 2nd and 4th byte is always Zero. The significance of these bytes is as follows:

- 1st Byte: Relative Humidity Integral Data in % (Integer Part)
- 2nd Byte: Relative Humidity Decimal Data in % (Fractional Part) – Zero for DHT11
- 3rd Byte: Temperature Integral in Degree Celsius (Integer Part)
- 4th Byte: Temperature in Decimal Data in % (Fractional Part) – Zero for DHT11
- 5th Byte: Checksum (Last 8 bits of {1st Byte + 2nd Byte + 3rd Byte+ 4th Byte})



Reading Data From MXX30100:

MAX30100 Data can be read by I2C protocol.The MAX30100 features an I2C/SMBus-compatible, 2-wire serial interface consisting of a serial data line (SDA) and a serial clock line (SCL). SDA and SCL facilitate communication between the MAX30100 and the master at clock rates up to 400kHz

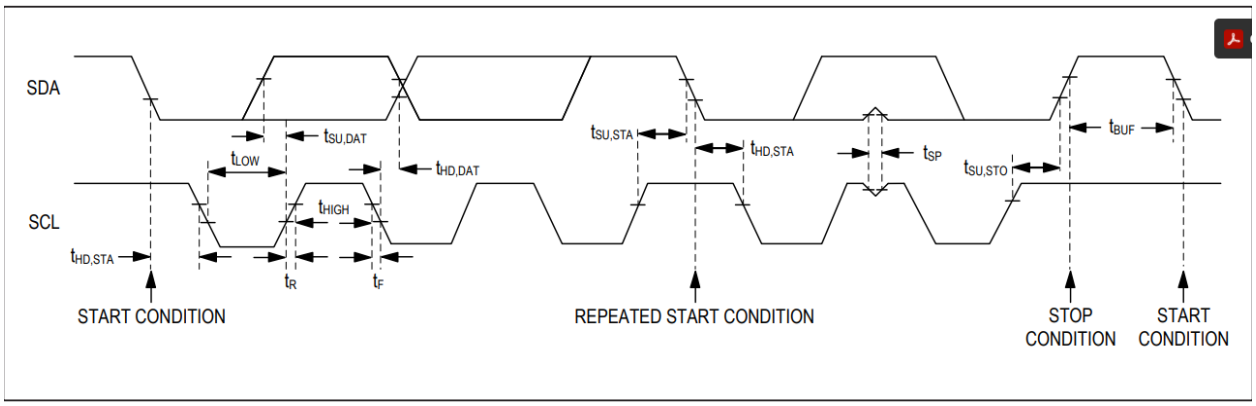
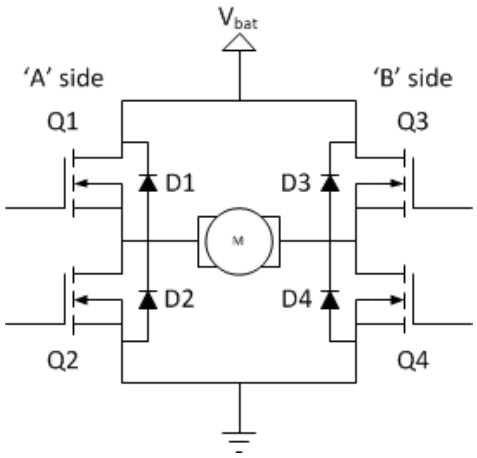


Figure shows the 2-wire interface timing diagram. The master generates SCL and initiates data transfer on the bus. The master device writes data to the MAX30100 by transmitting the proper slave address followed by data. Each transmit sequence is framed by a START (S) or REPEATED START (Sr) condition and a STOP (P) condition. Each word transmitted to the MAX30100 is 8 bits long and is followed by an acknowledged clock pulse. A master reading data from the MAX30100 transmits the proper slave address followed by a series of nine SCL pulses.The MAX30100 transmits data on SDA in sync with the

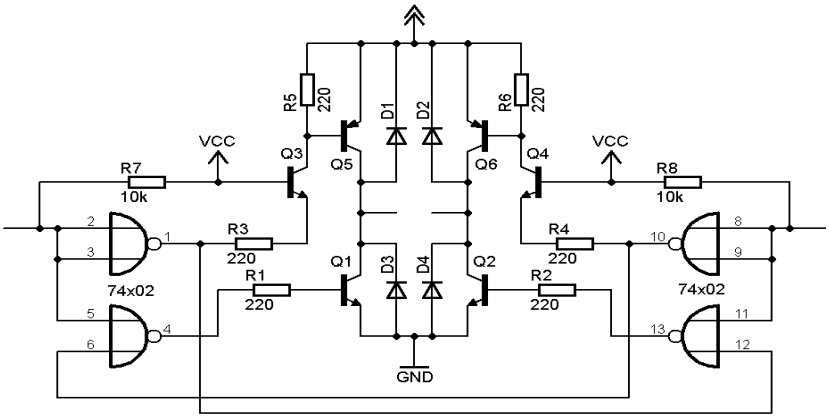
master-generated SCL pulses. The master acknowledges receipt of each byte of data. Each read sequence is framed by a START (S) or REPEATED START (Sr) condition, a not acknowledged, and a STOP (P) condition. SDA operates as both an input and an open-drain output. A pullup resistor, typically greater than 500Ω, is required on SDA. SCL operates only as an input. A pullup resistor, typically greater than 500Ω, is required on SCL if there are multiple masters on the bus, or if the single master has an open-drain SCL output.

Controlling Motor:

In our project, we are controlling a servo motor by L298N motor driver which has dual H-bridge configuration. This dual bidirectional motor driver is based on the very popular L298 Dual H-Bridge Motor Driver Integrated Circuit. The circuit will allow you to easily and independently control two motors of up to 2A each in both directions.It is ideal for robotic applications and well suited for connection to a microcontroller requiring just a couple of control lines per motor. It can also be interfaced with simple manual switches, TTL logic gates, relays, etc. This board equipped with power LED indicators, on-board +5V regulator and protection diodes.In general an H-bridge is a rather simple circuit, containing four switching element, with the load at the center, in an H-like configuration:



The switching elements (Q1..Q4) are usually bi-polar or FET transistors, in some high-voltage applications IGBTs. Integrated solutions also exist but whether the switching elements are integrated with their control circuits or not is not relevant for the most part for this discussion. The diodes (D1..D4) are called catch diodes and are usually of a Schottky type.The internal schematics of L298N is given below:



Interacting Camera and Speaker with Raspberry pi:

The Logitech C90 camera and Adafruit speaker have Usb cable and it uses a USB interface to interact with Raspberry pi. Here we will use driver fswebcam to interact with C90 in our host system which uses Usb protocol.

USB systems consist of a host, which is typically a personal computer (PC) and multiple peripheral devices connected through a tiered-star topology. This topology may also include hubs that allow additional connection points to the USB system. The host itself contains two components, the host controller and the root hub. The host controller is a hardware chipset with a software driver layer that is responsible for these tasks:

Detect attachment and removal of USB devices

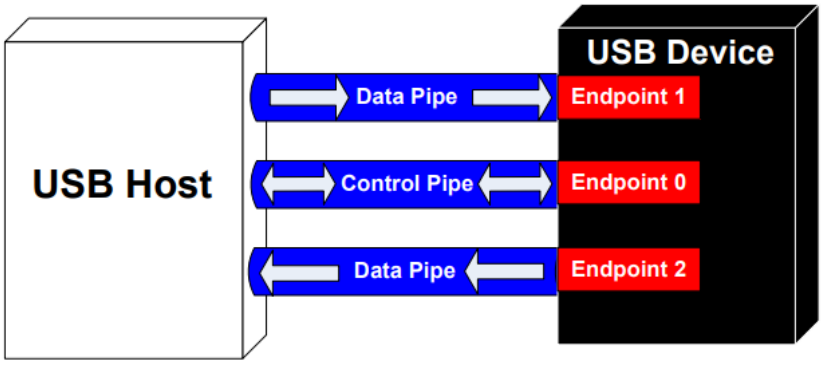
Manage data flow between host and devices

Provide and manage power to attached devices

Monitor activity on the bus

At least one host controller is present in a host and it is possible to have more than one host controller. Each controller allows connection of up to 127 devices with the use of external USB hubs. The root hub is an internal hub that connects to the host controller(s) and acts as the first interface layer to the USB in a system. Currently on your PC, there are multiple USB ports. These ports are part of the root hub in your PC. For simplicity, look at the root hub and host controller from the abstract view of a ―black box‖ that we call the host.

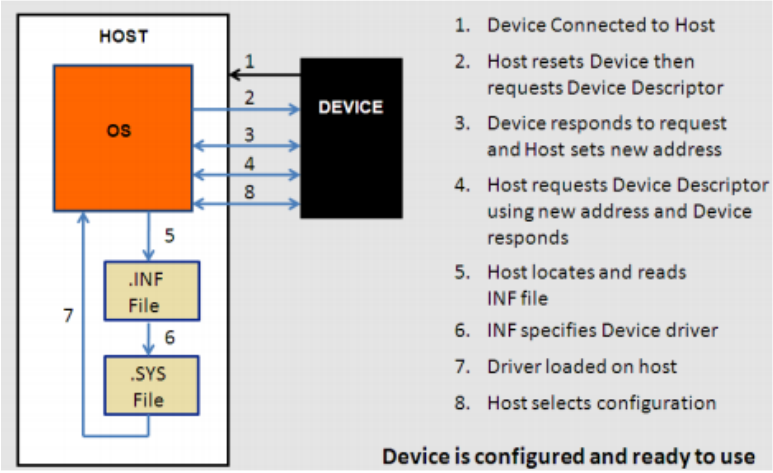
USB devices consist of one or more device functions, such as a mouse, keyboard, or audio device for example. Each device is given an address by the host, which is used in the data communication between that device and the host. USB device communication is done through pipes. These pipes are a connection pathway from the host controller to an addressable buffer called an endpoint. An endpoint stores received data from the host and holds the data that is waiting to transmit to the host. A USB device can have multiple endpoints and each endpoint has a pipe associated with it. This is shown in Figure.



When a USB device is first connected to a host, the USB enumeration process is initiated. Enumeration is the process of exchanging information between the device and the host that includes learning about the device. Additionally, enumeration includes assigning an address to the device, reading descriptors (which are data structures that provide information about the device), and assigning and loading a device driver. This entire process can occur in seconds. When this process is complete, the device is ready to transfer data to the host. The flow chart of the general enumeration process is shown in Figure 2. Two files are affiliated with enumeration and the loading of a driver. They exist on the host side.

.INF – A text file that contains all the information necessary to install a device, such as driver names and locations, Windows registry information, and driver version information.

.SYS – The driver needed to communicate effectively with the USB device.



After a device is enumerated, the host directs all traffic flow to the devices on the bus. Because of this, no device can transfer data without a request from the host controller.

Interacting PIR sensor:

It has 3 pinouts. 1. VCC 2. Signal 3. GND. Its output is high when it detects infrared waves emitted by the body and gives output as a high to 2nd pi which will be interfaced to GPIO of raspberry pi.

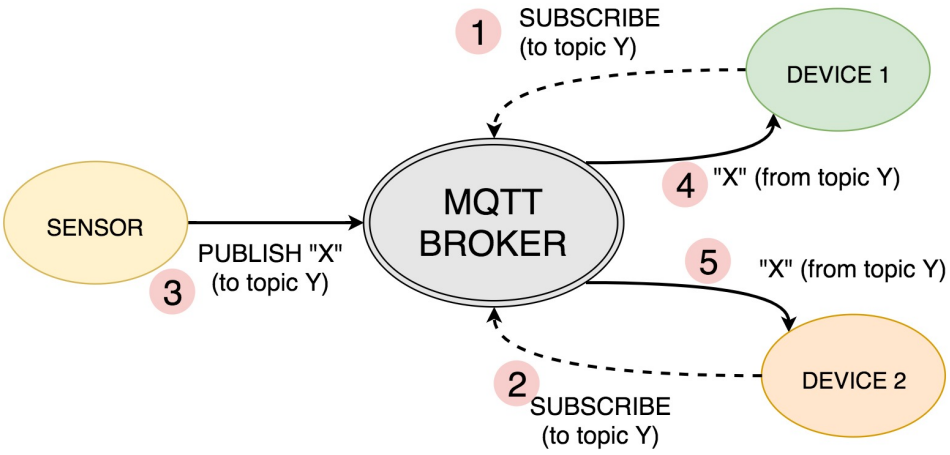
Microsoft Azure IoT Protocol:

The Azure IoT protocol gateway is a framework for protocol adaptation that is designed for high-scale, bidirectional device communication with IoT Hub. The protocol gateway is a pass-through component that accepts device connections over a specific protocol. It bridges the traffic to IoT Hub over AMQP 1.0.

The Azure IoT protocol gateway includes an MQTT protocol adapter that enables you to customize the MQTT protocol behavior if necessary. Since IoT Hub provides built-in support for the MQTT v3.1.1 protocol, you should only consider using the MQTT protocol adapter if protocol customizations or specific requirements for additional functionality are required.

The MQTT adapter also demonstrates the programming model for building protocol adapters for other protocols. In addition, the Azure IoT protocol gateway programming model allows you to plug in custom components for specialized processing such as custom authentication, message transformations, compression/decompression, or encryption/decryption of traffic between the devices and IoT Hub.

For flexibility, the Azure IoT protocol gateway and MQTT implementation are provided in an open-source software project. You can use the open-source project to add support for various protocols and protocol versions, or customize the implementation for your scenario.



GSM interface with Raspberry pi:

SIM 868 module is being used in this device which is a complete Quad-Band GSM/GPRS module which combines GNSS technology for satellite navigation. In the left side of the module you can see a yellow jumper connected to the suitable pins. There are four pairs of pins in which the jumpers must be shorted to make three terminal pairs (A,B,C).

A: control the SIM868 through USB TO UART

B: control the SIM868 through Raspberry Pi

C: access Raspberry Pi through USB TO UART

Here we make use of A terminal, i.e. the jumpers must be connected vertically in the first two pins as shown in the figure given below.