**Kirti Bhagat Report 1**

**Communication**

**Serial Peripheral Interface (SPI) protocol:**

It is a four wire bus with a master (single) slave architecture. The wires are:

• SCK or SCLK : Serial Clock, clock generated by master

• MOSI: Master Output Slave Input, data output from master

• MISO: Master Input Slave Output, data output from slave

• SS : Slave Select, often an active low input of a slave

Each device is connected to the master with a SS line, therefore increasing the number of pins required with every slave device.

The communication starts when the master device selects a slave device for communication by pulling down corresponding SS line as this line is active low.

The clock is generated by the Master which means that the flow of data is controlled by the master. For every clock cycle, one bit of data is transmitted from master to slave and from slave to master.

This process happen simultaneously and after 8 clock cycles, a byte of data is transmitted in both directions. The Master simultaneously transmits data on MOSI line and receives data from slave on MISO line.

Clock Polarity determines the state of the clock.

Clock Phase determines the clock transition i.e. rising (LOW to HIGH) or falling (HIGH to LOW), at which the data is transmitted.

SPI does not require complex hardware or individual addresses for slaves and offers high speed of data transfer but every slave requires an added pin and there is no flow control.

SPI is used to talk to a variety of peripherals, such as temperature sensor , pressure sensor , ADC, touchscreens, video game controllers. (PS2 controller)

**I2C Protocol:**

Serial bus protocol consisting of two signal lines :

SCL: The clock line. It is used to synchronize all data transfers over the I2C bus.

SDA: Data line.

Protocol:

1. A start sequence

2. I2C address of the slave with the R/W bit high/low.

3. The internal register number to read from or write to.

4. The data byte

5. Send the stop sequence.

The start sequence and stop sequence are special in that these are the only places where the SDA (data line) is allowed to change while the SCL (clock line) is high.

Modules and the common chips mostly have 7 bit addresses. This means that we can have up to 128 devices on the I2C bus, since a 7bit number can be from 0 to 127.

Although I2C requires only 2 signal wires and it allows multiple devices on the bus to be the master, its communication speed in slow.

Uses:

Accessing low-speed DACs and ADCs.

Changing sound volume in intelligent speakers.

Controlling small (e.g. feature phone) OLED or LCD displays.

Reading hardware monitors and diagnostic sensors, e.g. a fan's speed.

**UART:**

Universal Asynchronous Receiver Transmitter, unlike SPI and I2C, is not a protocol but a piece of hardware that acts as a bridge between the processor and the serial communication protocol or port.

In UART Serial Communication, the data is transmitted asynchronously i.e. there is no clock involved between the sender and receiver. Instead of clock signal, UART uses Start and Stop bits which are added to the actual data packet at the beginning and end respectively.

In UART, the transmitter and receiver must agree on timing parameters beforehand.

The transmitting UART receives parallel data from the controlling device through the data bus.

To this data, the UART adds Start, Parity and Stop bits in order to convert it into a data packet. The data packet is then converted from parallel to serial with the help of shift register and is transmitted bit – by – bit from the TX pin.

The receiving UART receives this serial data at the RX pin and detects the actual data by identifying the start and stop bits.

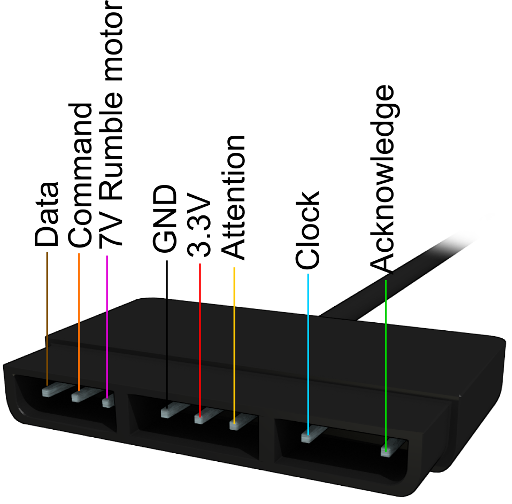
The data is again converted to parallel data with the help of shift register.

Although the speed of data transfer is less, UART requires only two wires for full transmission and does not require any clock signal.

Also the parity bit ensures basic error checking.

Some very low-cost home computers or embedded systems dispense with a UART.

**PS2 controller**

The standard PS2 controller has 15 buttons; all of them, except for Analog, Start and Select are analog (pressure sensitive buttons).

PS2 controllers also have two analog joysticks. Two potentiometers, variable resistors, are positioned at right angles to each other below the joystick. Resistance is increased or decreased based on the position of the joystick. By monitoring the output of each potentiometer, the PS2 can determine the exact angle at which the joystick is being held.

To interface a PS2 controller the following lines are connected:

**Brown** - Data: Controller -> PlayStation. This is an open collector output and requires a pull-up resistor (1 to 10k, maybe more).

**Orange** - Command: PlayStation -> Controller.

**Purple**- Vibration Motors Power: Powers the motors in each handgrip, for force feedback.

**Black** - Ground

**Red** – Power

**Yellow** - Attention: This line must be pulled low before each group of bytes is sent / received, and then set high again afterwards. It tells the slave whether or not it is “active” and should listen to data bits coming across the CMD channel, or send data bits across the DATA channel.

**Blue** - Clock: 500kH/z, normally high on. The communication appears to be **SPI** bus.

**Green** - Acknowledge: This normally high line drops low about 12us after each byte for half a clock cycle, but not after the last bit in a set. After each Command is received from the controller, that controller needs to pull ACK low for at least one clock cycle.

The play station sends a byte at the same time as it receives one (full duplex) via serial communication. There’s a clock (SCK) to synchronize bits of data across two channels: DATA and CMD. The clock is held high until a byte is to be sent. It then drops low (active low) to start 8 cylces during which data is simultaneously sent and received. When the clock edge drops low, the values on the line start to change. When the clock goes from low to high, value are actually read. Bytes are transferred LSB (least significant bit) first, so the bits on the left (earlier in time) are less significant.

**To control 4 wheel bot:**

We can use 4 motors for 4 wheel bot. and left joystick to control left side of 2 motors & right joystick for right side of 2 motors.

Joysticks give analog values which can be used to give PWM values to the motors and control its speed.

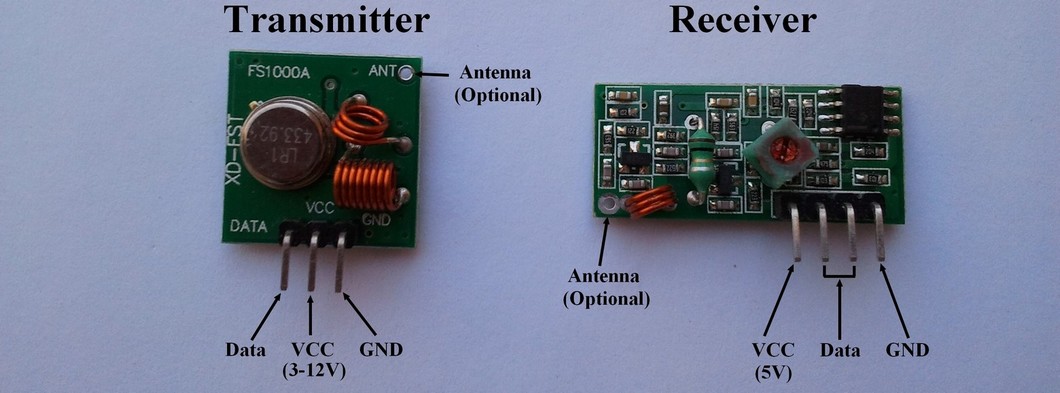
When we move the joystick upward, the wheels should move forward & also when we move it downwards, the wheels should move backward.

To move the bot left, we can move the right wheels forward while not moving the left wheels.

We can use an RF transmitter to make it wireless.

**RF module:**

RF modules are 433 MHz RF transmitter and receiver modules. The data is sent serially from the transmitter which is received by the tuned receiver. Transmitter and the receiver are duly interfaced to two microcontrollers for data transfer.

Conections:

Transmitter: VCC and GND connections are straightforward, the DATA or "ATAD" pin goes to the any number digital slot on the Arduino Board.

Receive: VCC and GND connections are straightforward, one of the DATA pins goes to the any number digital slot on the Arduino Board.

Code to read data from serial monitor and transfer it through the transmitter.

We use the VirtualWire library. This library consists of several functions that are used for configuring the modules, transmission of data by the transmitter module and data reception by the receiver module.

**Transmitter code:**

#include <VirtualWire.h>

char inData[50];

int newmessage = 0;

void setup() {

vw\_setup(2000);

vw\_set\_tx\_pin(12);

Serial.begin(9600);

}

void loop()

{

char inChar;

byte index = 0;

char mss[20];

while(Serial.available() > 1){

if(index < 49)

{

delay(10);

inChar = Serial.read();

inData[index] = inChar;

index++;

inData[index] = '\0';

}

newmessage = 1;

}

if(newmessage == 1){

inData[0] = '-';

sprintf(mss, "%s", inData);

vw\_send((uint8\_t \*)mss, strlen(mss));

vw\_wait\_tx();

Serial.println(mss);

delay(600);

newmessage = 0;

}

}

**Receiver code:**

#include <VirtualWire.h>

void setup()

{

Serial.begin(9600);

Serial.println("Listening");

vw\_setup(2000);

vw\_rx\_start();

}

void loop()

{

byte message[VW\_MAX\_MESSAGE\_LEN];

byte messageLength = VW\_MAX\_MESSAGE\_LEN;

if (vw\_get\_message(message, &messageLength))

{

for (int i = 0; i < messageLength; i++)

{

Serial.write(message[i]);

}

Serial.println();

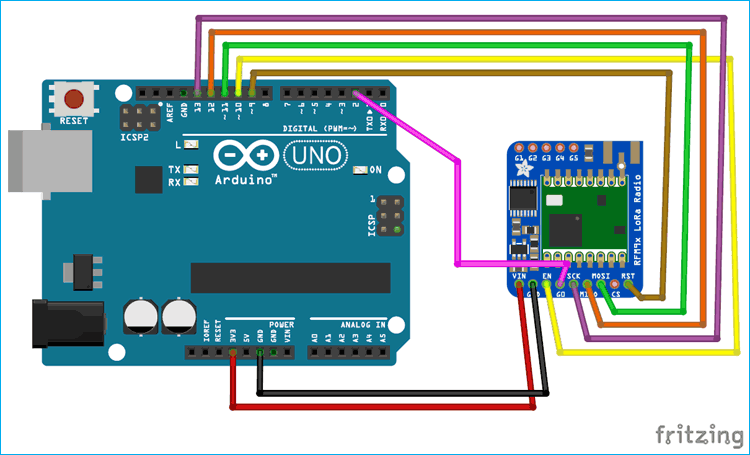
}

}

**LORA:**

LoRa(long range) is a wireless data communication technology that can achieve high distance communication without using much power The maximum bandwidth for Lora is around 5.5 kbps, this means that you will be able to send only small amount of data through LoRa. All LoRa modules are transceivers, which means they can send and receive information.

Signal from one LoRa Node reaches another Node through a LoRa Gateway. These Gateways then take the information to the internet and finally to the end user through an application interface. Similarly the data from the user will also reach the node through the network server and the Gateway. A LoRa Node usually operates on a Battery and consists of a Radio Module and Microprocessor. The Microprocessor is used to read the data from the senor and send it in the air through the Radio module which will then be picked up by a LoRa Gateway. The LoRa Gateway also has a Radio Module and a Microprocessor but is normally operated over AC mains since they require more power

The LoRa module consists of 16 pins with 8 pins on each side. Out of these 16 pins, six are used by GPIO pins ranging from DIO0 to DIO5 and four are used by Ground pins. The module operates in 3.3V . We can connect the **SPI** pin on the LoRa to the SPI pins on Arduino Board. These connections are made at LoRa nodes.

**Bluetooth module:**

HC-05 is a Bluetooth module which is designed for wireless comunication. This module can be used in a master or slave configuration.

It has 6 pins,

1. Key/EN: It is used to bring Bluetooth module in AT commands mode. If Key/EN pin is set to high, then this module will work in command mode. Otherwise by default it is in data mode. The default baud rate of HC-05 in command mode is 38400bps and 9600 in data mode.

HC-05 module has two modes,

1. Data mode: Exchange of data between devices.

2. Command mode: It uses AT commands which are used to change setting of HC-05. To send these commands to module serial (USART) port is used.

2. VCC: Connect 5 V or 3.3 V to this Pin.

3. GND: Ground Pin of module.

4. TXD: Transmit Serial data (wirelessly received data by Bluetooth module transmitted out serially on TXD pin)

5. RXD: Receive data serially (received data will be transmitted wirelessly by Bluetooth module).

6. State: It tells whether module is connected or not.

HC-05 has red LED which indicates connection status, whether the Bluetooth is connected or not. Before connecting to HC-05 module this red LED blinks continuously in a periodic manner. When it gets connected to any other Bluetooth device, its blinking slows down to two seconds.

HC 05/06 works on serial communication

Remove Bluetooth module Tx Rx connection before uploading the program.

To control a LED on pin 10 of arduino using a bluetooth module and an android app:

char Incoming\_value = 0;

void setup()

{

Serial.begin(9600);

pinMode(10, OUTPUT);

}

void loop()

{

if(Serial.available() > 0)

{

Incoming\_value = Serial.read();

Serial.print(Incoming\_value);

Serial.print("\n");

if(Incoming\_value == '1')

digitalWrite(10, HIGH);

else if(Incoming\_value == '0')

digitalWrite(10, LOW);

}

}