Technocrats Summer Task – 2

-By Vishnu Kumar

UART:

UART stands for Universal Asynchronous Receiver/Transmitter. It’s not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone IC. A UART’s main purpose is to transmit and receive serial data. One of the best things about UART is that it only uses two wires to transmit data between devices.

UARTs transmit data asynchronously, which means there is no clock signal to synchronize the output of bits from the transmitting UART to the sampling of bits by the receiving UART. Instead of a clock signal, the transmitting UART adds start and stop bits to the data packet being transferred. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits.

When the receiving UART detects a start bit, it starts to read the incoming bits at a specific frequency known as the baud rate. Both UARTs must operate at about the same baud rate. The baud rate between the transmitting and receiving UARTs can only differ by about 10% before the timing of bits gets too far off. Both UARTs must also must be configured to transmit and receive the same data packet structure.

The UART that is going to transmit data receives the data from a data bus. The data bus is used to send data to the UART by another device like a CPU, memory, or microcontroller. Data is transferred from the data bus to the transmitting UART in parallel form. After the transmitting UART gets the parallel data from the data bus, it adds a start bit, a parity bit, and a stop bit, creating the data packet. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long distance data transfers. Next, the data packet is output serially, bit by bit at the Tx pin. The receiving UART reads the data packet bit by bit at its Rx pin. The receiving UART then converts the data back into parallel form and removes the start bit, parity bit, and stop bits. Finally, the receiving UART transfers the data packet in parallel to the data bus on the receiving end

This type of interface has been largely replaced by USB and Ethernet, but UART is still used whenever simple low speed communication is needed.

SPI:

SPI (Serial peripheral Interface) is used to communicate between two devices, A master and a slave. One master can control one or many slaves. One advantage of SPI is that data can be transferred without any interruption.

The SPI bus can operate with a single master device and with one or more slave devices.

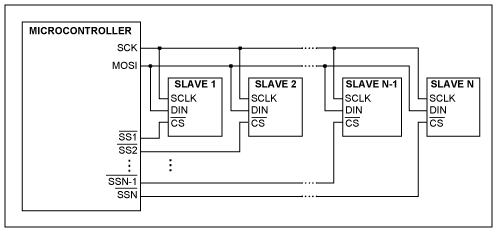
If a single slave device is used, the SS pin may be fixed to logic low if the slave permits it. Some slaves require a falling edge of the chip select signal to initiate an action. With multiple slave devices, an independent SS signal is required from the master for each slave device.

The SPI protocol uses 4 lines/wires to communicate.

**MOSI (Master Output/Slave Input)** – Line for the master to send data to the slave.(DIN on slave)  
**MISO (Master Input/Slave Output)** – Line for the slave to send data to the master.(DOUT on slave)  
**SCLK (Clock)** – Line for the clock signal.  
**SS/CS (Slave Select/Chip Select)** – Line for the master to select which slave to send data to.

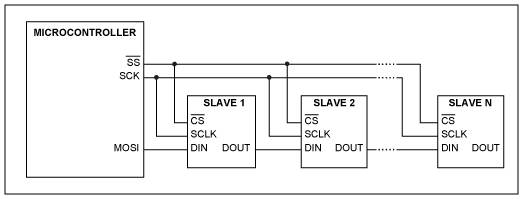
Slaves can be connected to the master in 2 configurations,

1. Independent slave configuration: - In this configuration, each slave is connected to the same SCLK, MOSI and MISO port but, each slave must be connected to a SS port of its own. Therefore, for each slave, we require an extra SS port on the master.



The slave is chosen by pulling down their respective SS/CS inputs and then it responds to the activity on the serial clock and data lines. This is suitable for small number of slaves, but as the number of slave’s increase, so does the layout complexity and the hardware. This is where we use the next configuration.

1. Daisy chain configuration: - In daisy chain configuration, the SCLK and SS from the all the slave is connected to the same SCLK and SS port on the master respectively. The MOSI of the master is connected to MOSI of the first slave. The MISO of the first slave is the MOSI of the second slave and so on till the last slave and the last MISO is connected to the MISO on the master.



For daisy-chaining to work successfully, the slave must be able to input a command at DIN during a given command-cycle (defined by the number of clock pulses required to clock in one command), and output the same command at DOUT during the subsequent command-cycle. Stated simply, there is a DIN-to-DOUT delay of one command-cycle. The slave must, moreover, only execute the command written to it on the rising edge of active-low CS. This means that as long as active-low CS remains low, the slave ignores the command and outputs it at DOUT on the following command-cycle. If active-low CS goes high after a given command-cycle, all slaves execute the commands just written to their respective DIN inputs. If active-low CS goes high, data is not output at DOUT. This process makes it possible for every slave in the chain to execute a different command. As long as these daisy-chain requirements are satisfied, the microcontroller only needs three signals (active-low SS, SCK, and MOSI) to control all the slaves in the network.

The SPI is use to talk to many peripherals, such as Sensors, Control devices, memory and so on...

I2C:

I2C combines the best features of SPI and UARTs. With I2C, you can connect multiple slaves to a single master (like SPI) and you can have multiple masters controlling single, or multiple slaves. This is really useful when you want to have more than one microcontroller logging data to a single memory card or displaying text to a single LCD.

Like UART communication, I2C only uses two wires to transmit data between devices:

**SDA (Serial Data)** – The line for the master and slave to send and receive data.

**SCL (Serial Clock)** – The line that carries the clock signal.

I2C is a serial communication protocol, so data is transferred bit by bit along a single wire (the SDA line). Like SPI, I2C is synchronous, so the output of bits is synchronized to the sampling of bits by a clock signal shared between the master and the slave. The clock signal is always controlled by the master.

With I2C, data is transferred in *messages.*Messages are broken up into *frames* of data. Each message has an address frame that contains the binary address of the slave, and one or more data frames that contain the data being transmitted. The message also includes start and stop conditions, read/write bits, and ACK/NACK bits between each data frame:

**Start Condition:**The SDA line switches from a high voltage level to a low voltage level before the SCL line switches from high to low.

**Stop Condition:** The SDA line switches from a low voltage level to a high voltage level after the SCL line switches from low to high.

**Address Frame:** A 7 or 10-bit sequence unique to each slave that identifies the slave when the master wants to talk to it.

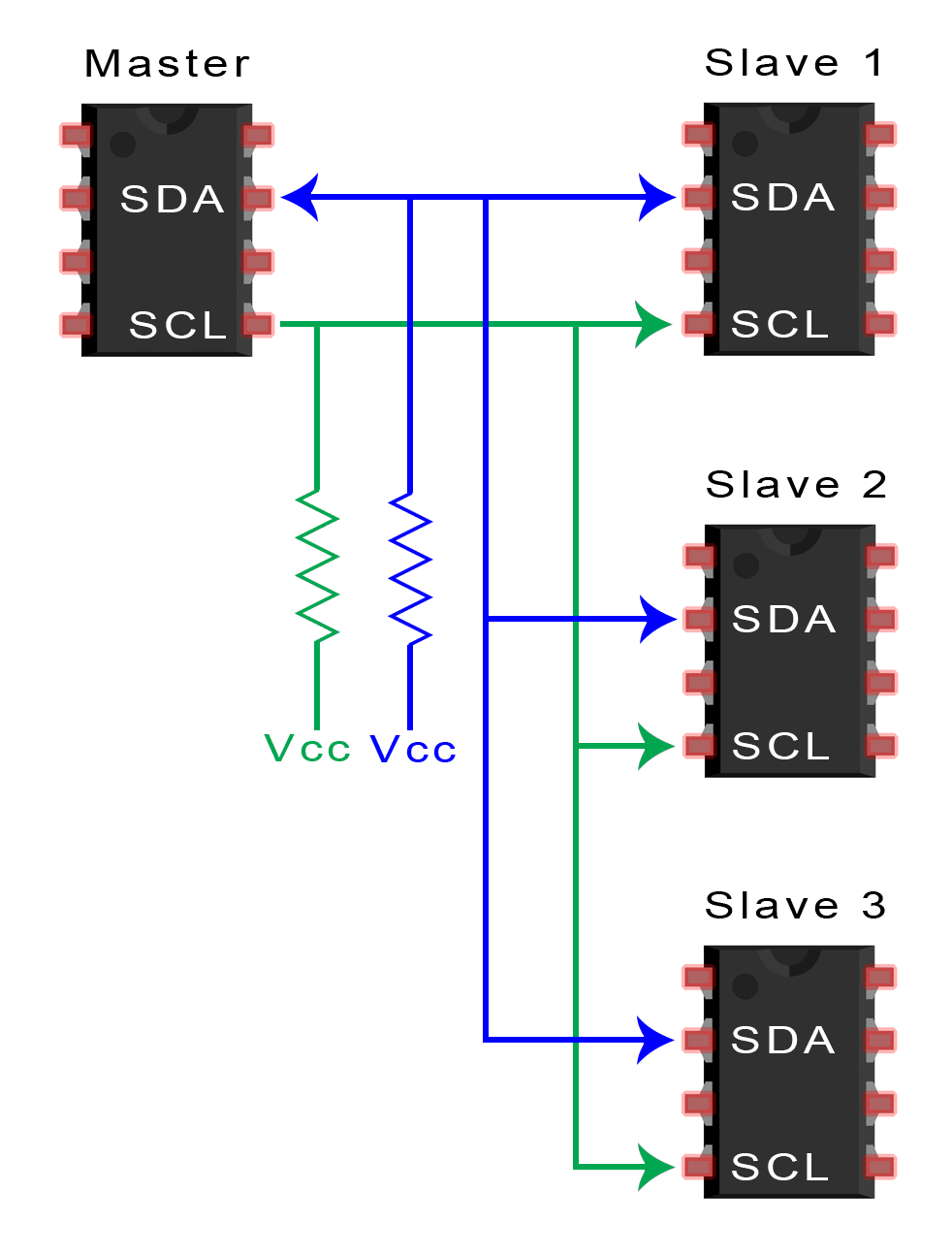
**Read/Write Bit:** A single bit specifying whether the master is sending data to the slave (low voltage level) or requesting data from it (high voltage level).

**ACK/NACK Bit:** Each frame in a message is followed by an acknowledge/no-acknowledge bit. If an address frame or data frame was successfully received, an ACK bit is returned to the sender from the receiving device.

Similar to SPI, there are 2 methods of connection,

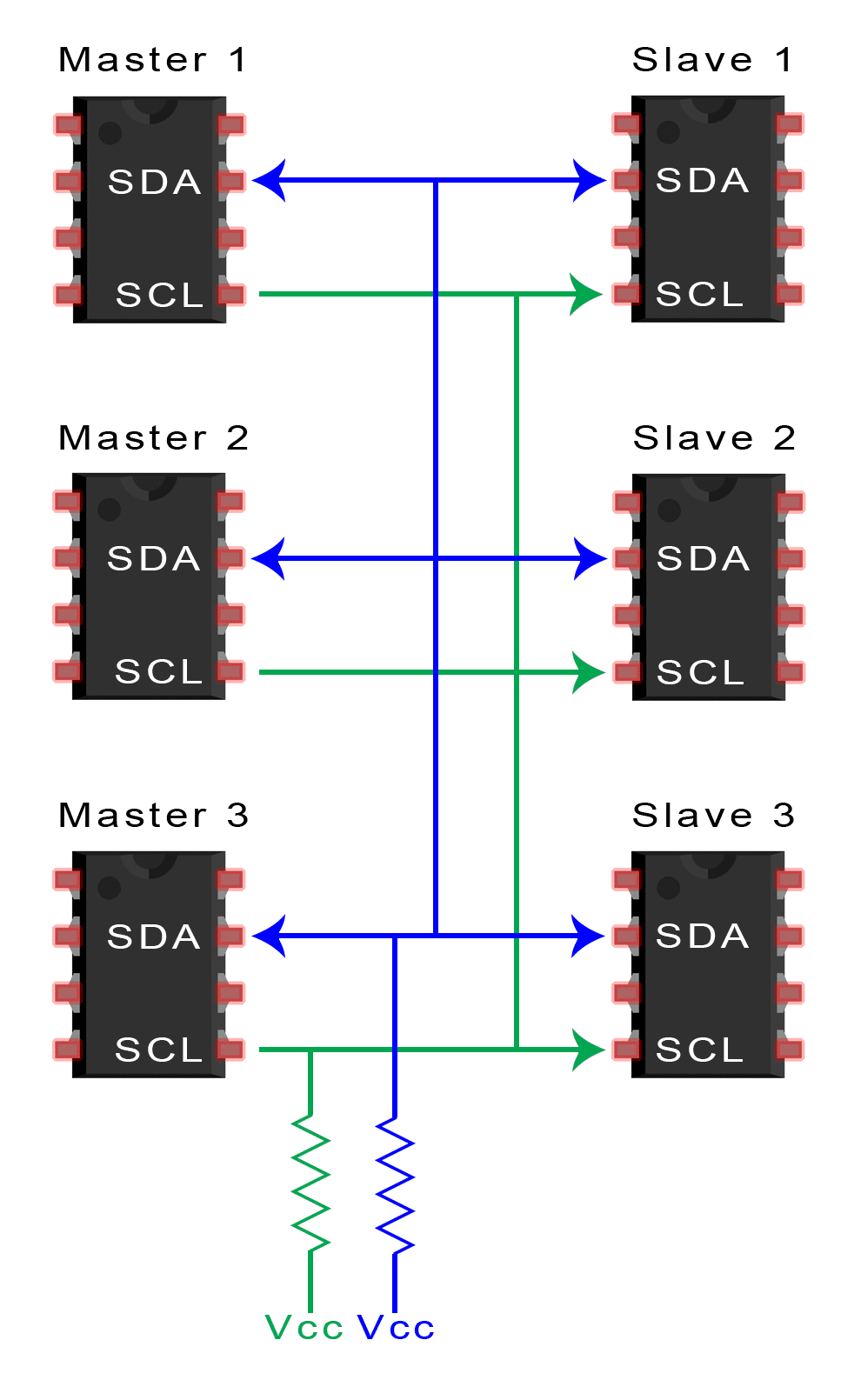
1. Single master – Multiple Slaves: -

Because I2C uses addressing, multiple slaves can be controlled from a single master. With a 7-bit address, 128 (27) unique address are available. Using 10 bit addresses is uncommon, but provides 1,024 (210) unique addresses. To connect multiple slaves to a single master, wire them like this, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



1. Multiple Masters – Multiple Slaves: -

Multiple masters can be connected to a single slave or multiple slaves. The problem with multiple masters in the same system comes when two masters try to send or receive data at the same time over the SDA line. To solve this problem, each master needs to detect if the SDA line is low or high before transmitting a message. If the SDA line is low, this means that another master has control of the bus, and the master should wait to send the message. If the SDA line is high, then it’s safe to transmit the message. To connect multiple masters to multiple slaves, use the following diagram, with 4.7K Ohm pull-up resistors connecting the SDA and SCL lines to Vcc:



The uses of I2C are: 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.

Bluetooth:

For Bluetooth communication, we use a HC – 05 modules. It has 6 pins:

1. Enable / Key: This pin is used to toggle between Data Mode (set low) and AT command mode (set high). By default, it is in Data mode
2. Vcc: Powers the module. Connect to +5V Supply voltage
3. Ground: Ground pin of module, connect to system ground.
4. Tx(Transmitter): Transmits Serial Data. Everything received via Bluetooth will be given out by this pin as serial data.
5. RX (Receiver): Receive Serial Data. Every serial data given to this pin will be broadcasted via Bluetooth
6. State: The state pin is connected to on board LED, it can be used as a feedback to check if Bluetooth is working properly.

Also it has a LED to indicate status of the module, and a button to control the Key/Enable pin to toggle between data and command mode.

The **HC-05** has two operating modes, one is the Data mode in which it can send and receive data from other Bluetooth devices and the other is the AT Command mode where the default device settings can be changed. We can operate the device in either of these two modes by using the key pin.

It is very easy to pair the HC-05 module with microcontrollers because it operates using the Serial Port Protocol (SPP). Simply power the module with +5V and connect the Rx pin of the module to the Tx of MCU and Tx pin of module to Rx of MCU.

On power up of the module, it enters paring mode and can be discovered as “HC – 05” and the password must be either 0000 or 1234. The module must always be disconnected from the microcontroller before uploading any program. This is an example code to control a led by using a Arduino and a smartphone with a Bluetooth app pre-configured and print the incoming values to the serial monitor.

const int LED = 13;

char Val;

void setup()

{

Serial.begin(9600);

pinMode(LED, OUTPUT);

}

void loop()

{

if(Serial.available() > 0)

{

Val = Serial.read();

Serial.print(Val);

Serial.print("\n");

if(Val == '1')

digitalWrite(LED, HIGH);

else if(Val == '0')

digitalWrite(LED, LOW);

}

}

RF:

RF Module is a cheap wireless communication module for low cost application. RF Module comprises of a transmitter and a receiver that operate at a radio frequency range. Usually, the frequency at which these modules communicate will be 315 MHz or 434 MHz.

The RF Transmitter Module consists of 4 – pins: VCC, GND, Data and Antenna. VCC and GND pins are connected to 5V and ground respectively. The data pin is connected to any of the digital input / output pin of Arduino. The RF Receiver Module consists of 4 – pins: VCC, GND, Data and Antenna. VCC and GND pins are connected to 3.3V pin of the Arduino and ground respectively. The data pin is connected to Pin 12 of the Arduino. An antenna must be connected to the antenna pin on both modules, which is nothing but a wire wound in the form of a coil

To use the RF modules with Arduino, we a using a library called VirtualWire. t is a communication library that allows two Arduino’s to communicate with each other using RF Module i.e. transmitter – receiver pair. 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 SendMsg[50];

int newmessage = 0;

void setup() {

vw\_setup(2000);

vw\_set\_tx\_pin(12);

Serial.begin(9600);

}

void loop()

{

char InMsg;

byte i = 0;

char msg[20];

while(Serial.available() > 1){

if(i < 49)

{

delay(10);

InMsg = Serial.read();

SendMsg[i] = InMsg;

i++;

SendMsg[i] = '\0';

}

newmessage = 1;

}

if(newmessage == 1){

SendMsg[0] = '-';

sprintf(msg, "%s", SendMsg);

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

vw\_wait\_tx();

Serial.println(msg);

delay(600);

newmessage = 0;

}

}

**Receiver code:**

#include <VirtualWire.h>

void setup()

{

Serial.begin(9600);

vw\_setup(2000);

vw\_rx\_start();

}

void loop()

{

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

byte MsgLen = VW\_MAX\_MESSAGE\_LEN;

if (vw\_get\_message(Msg, &MsgLen))

{

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

{

Serial.write(Msg[i]);

}

Serial.println();

}

}

LoRa:

The term LoRa stands for Long Range. It is a wireless Radio frequency technology introduced by a company called Semtech. This LoRa technology can be used to transmit bi-directional information to long distance without consuming much power. This property can be used by remote sensors which have to transmit its data by just operating on a small battery. Typically, Lora can achieve a distance of 15-20km (will talk more on this later) and can work on battery for years.

 In order to achieve high distance with Low power LoRa compromises on Bandwidth, it operates on very low bandwidth. 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. So, you cannot send Audio or Video through this technology, it works great only for transmitting less information like sensor values.

We can setup the LoRa modules to the Arduino as Transmitting side or Receiving end. Anyways the connection remains the same. Also it is important that we attach a antenna to the LoRa module, else it will damage the board. The antennas must be chosen such that it has the same rating as the frequency on the LoRa chip.

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 and hence the 3.3V pin on LoRa is connected to the 3.3v pin on the Arduino UNO board. Then we connect the SPI pin on the LoRa to the SPI pins on Arduino Board

To program the Arduino to use the LoRa module, we have to install the LoRa library (Preferably “LoRa Radio” by Sandeep Mistry) and then use the same library to write the code.(Example code is given under File -> Example -> LoRa ).

PS2 Controller:

The PS2 wireless controller is a standard controller for the PlayStation 2 and is identical to the original [DualShock](https://en.wikipedia.org/wiki/DualShock#DualShock_3) controller for the PlayStation console. It features twelve &analogs (pressure-sensitive) buttons (Χ, O, Π, Δ, L1, R1, L2, R2, Up, Down, Left and Right), five digital button (L3, R3 Start, Select and the analog mode button) and two analog sticks. The controller also features two vibration motors, the left one being larger and more powerful than the one on the right. It is powered by two AAA batteries. It communicates with the console using 2.4 GHz RF protocol.

1. **DATA:** This is the data line from Controller to PS2. This is an open collector output and requires a pull-up resistor (1 to 10k, maybe more). (A pull-up resistor is needed because the controller can only connect this line to ground; it can’t actually put voltage on the line).
2. **COMMAND:** This is the data line from PS2 to Controller.
3. **VIBRATION MOTOR POWER**
4. **GND:** Ground
5. **VCC:** VCC can vary from 5V down to 3V.
6. **ATT:** ATT is used to get the attention of the controller. This line must be pulled low before each group of bytes is sent / received, and then set high again afterwards. This pin considers as “Chip Select” or “Slave Select” line that is used to address different controllers on the same bus.
7. **CLK:**500kH/z, normally high on. The communication appears to be SPI bus.
8. **Not Connected**
9. **ACK:**Acknowledge signal from Controller to PS2. 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. This is a [open collector output](http://en.wikipedia.org/wiki/Open_collector) and requires a pull-up resistor (1 to 10k, maybe more).

PS2 wireless controller communicates with Arduino using a protocol that is basically SPI. 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. Additionally, there’s a “Attention” (ATT) channel which 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 (Reasonably, only one slave device should be active at a time). The PlayStation 2 actually uses this plus an additional line that is not specifically part of the SPI protocol – an “Acknowledge” (ACK) line.

The clock is held high until a byte is to be sent. It then drops low (active low) to start 8 cycles during which data is simultaneously sent and received. The logic level on the data lines is changed by the transmitting device on the falling edge of clock. This is then read by the receiving device on the rising edge allowing time for the signal to settle. After each Command is received from the controller, that controller needs to pull ACK low for at least one clock cycle. If a selected controller does not ACK the PS2 will assume that there is no controller present. LSBs (least significant bits) are transmitting first.

There are a variety of ways we can interface the PS2 controller to a Arduino to control a bot. For example, we can use the left analog stick for Forward/Reverse and the right stick for Left/Right, or use R2 as Forward, L2 as Reverse, and the box and circle buttons as Right/Left respectively and so on… it depends upon the user’s comfort and his responsiveness to each button in the controller.