Let's Make a Drone

Debraj Chakraborty, Kishore Chatterjee, B.G. Fernandes, Joseph John,
P.C. Pandey, Dinesh Sharma, Narendra S. Shiradkar and Kushal Tuckley
with
P.K. Baburajan, Rajneesh Bharadwaj, K.P. Karunakaran, D. Pawaskar and
S.V. Prabhu,

EE and ME Departments
IIT Bombay, Mumbai

Academic Year: 2023-2024, Semester: II (Spring)

Part I

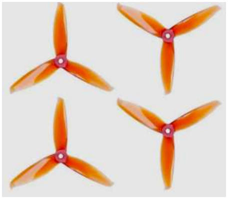
Making a Drone

Designing the Drone Top Down

Let us design our drone "top down" – from specifications to components.

What do we expect from our drone? . . .
First of all – it should fly!
We'll need powerful motors with propellers which push air downwards, so that the drone rises up in reaction. We'll use 4 such motors.

Due to rotation of propellers, the drone itself will try to rotate in the opposite direction. To prevent this, we'll use two propellers rotating clockwise while the other two rotate anticlockwise, with the tilt of propellers so adjusted that all push the air downwards.



What do we need to make the drone?

- The motors will need to be driven with heavy currents – and we should be able to control their speeds.
- so we need motor drivers which can supply the required drive voltages and currents and can control their speed using Pulse Width Modulation.
- We also need a source of power – a high current battery is required.
- We should minimize the weight of all components, so that it is easy for the drone to get air-borne. In particular, the frame should be made with a light, but strong material.

What else do we need to make the drone? . . .

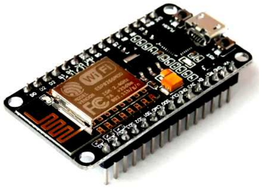
The Drone should be stable in its flight.

- How do we determine what PWM data to send to the electronic speed controllers such that it is stable? – We need sensors to determine the current angular orientation and angular speeds of the drone to adjust the relative speeds of the four motors. So we need a sensor card for this.
- Who will send PWM data to the motor driver (– the electronic speed controller)? Obviously, some sort of micro-controller is required and we'll use our familiar Arduino Nano for it.

How do we command the drone?

We should be able to control the flying drone from the ground..

- We should be able to send commands to the drone from an Earth position. This requires some wireless connectivity.
- There are several options for it – to cut costs, we'll use a mobile as the transmitter and an on-board WiFi unit to receive the data.

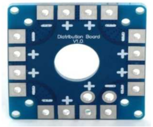


- We'll use a node-MCU with WiFi to receive commands and to pass these on to the Arduino Nano.
- The node MCU is also a micro-controller card like Arduino Nano. It uses a 32 bit ARM micro-controller and has an on-board WiFi unit.

What else do we need

We have listed most of the modules we need to perform the required functions on the drone.

- in addition to these, we need to generate 5 V from the battery voltage to power various boards.
  This is done using a switch mode buck converter.
- To distribute power to different components, we need a power distribution board and various wires and connectors.
- For securely mounting all these components, we shall need mechanical parts such as mounting brackets, nuts and bolts etc.

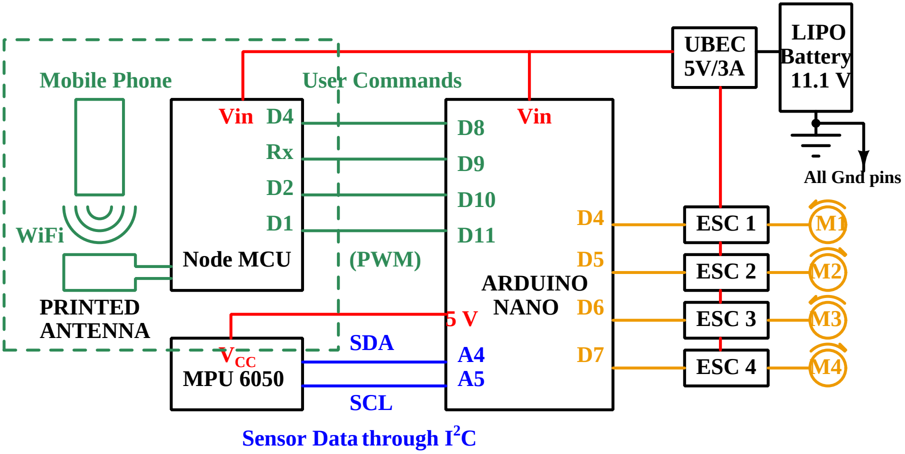


A list of all components required for the project is called a Bill of Materials or BOM.

Bill of Materials

BLDC motor:	GT2205 – 2x CW, 2x CCW
Propellers:	Orange HD 5152 pair 1 pair CW, 1 pair CCW
Electronic Speed Controller:	ESC – SimonK 30 A x4
Frame:	Carbon Fibre x1
Battery:	LiPo 3S Orange 1800 mAh x1
BEC Buck converter:	UBEC-5V/3A
Transmitter:	Use your mobile with app
Sensor:	Intertial Measurement Unit (IMU) MPU-6050 x1
WiFi unit:	Node MCU ESP 8266 x1
Power Distribution Board	100A x1
Battery Charger:	IMAX B3 AC x1
On board Flight Controller:	Arduino Nano x1
XT 60 Cable	With Male connector, Pigtail
Bullet connector:	Amass PolyMax 3.5mm Gold 2pairs x6

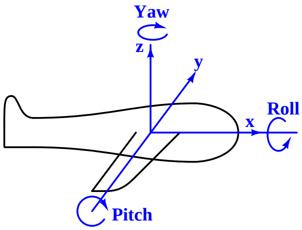
Putting it all together



Terminology

It is useful to understand terms which are frequently used for flying and sailing objects. Consider an object flying along the x direction.

Yaw:	is rotation around the vertical axis. (Left and right turn from the direction of motion).
Roll:	is rotation around the direction of motion (x axis). (Movement of wing tips in opposite up/down direction).
Pitch:	is rotation around the y axis. (Up/down movement of the nose of the aircraft with respect to its tail).



Commanding the Drone

To fly the drone, we need to send commands to it to control and adjust:

1. Its speed – (throttle),
2. its left-right turn orientation (Yaw),
3. its up/down orientation (Pitch) and
4. its left-right tilt (Roll).

This will be done by a mobile app with two joy-stick like controls …
One to control throttle and yaw, the other to control its pitch and roll.

We need to continuously measure these quantities and correct for deviation from the desired values using PID control. (Proportional/Integral/Differential compensation for errors).

How do we measure these? …

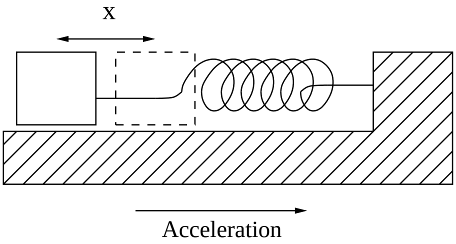
Part II

Sensors in the Drone

Measuring Acceleration

Accelerometers actually measure force. Then $a = F/m$.

One way of measuring acceleration is through the use of an inertial mass.



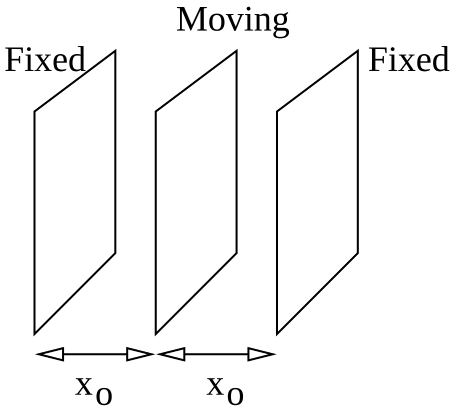
The frame (shown hashed) is rigidly coupled to the moving body. There is an inertial mass which is constrained to move along the direction of motion. It is fixed to the frame through a spring.

As the body **accelerates** to the right, it leaves the inertial mass behind, stretching the spring by x. Then,

$$-ma = kx \text{ so } a = -\frac{k}{m}x$$

Differential Capacitance measurement

How do we measure very small displacements?



$$\Delta C = \frac{\epsilon A}{x_o - x} - \frac{\epsilon A}{x_o + x}$$

$$= \epsilon A \frac{(x_o + x) - (x_o - x)}{x_o^2 - x^2}$$

$$\text{so } \Delta C = \frac{2\epsilon A}{x_o^2 - x^2} x$$

ΔC is linear in x if $x << x_o$.

To ensure this, we use the principle of force feedback.

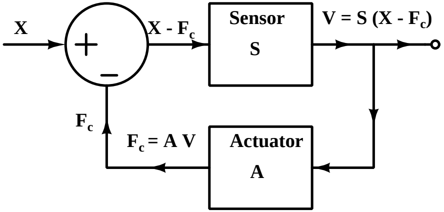
Principle of Force Balance

There is a trade off between sensitivity and range.
A sensitive sensor can be overloaded by a large input value.

How can we control a large value with good sensitivity?

This can be done by using force balance.

- Let $\mathbf{X}$ be a large value of some measurand (say force).
- We balance it with a *controlled* force $\mathbf{F_c}$, and measure the small resultant $\mathbf{X} - \mathbf{F_c}$ with good sensitivity.
- Since $\mathbf{F_c}$ is known, we can now calculate $\mathbf{X}$.



- A feedback loop subtracts $\mathbf{F_c}$ from $\mathbf{X}$ and adjusts $\mathbf{F_c}$ such that $\mathbf{X} - \mathbf{F_c}$ is very small.
- A sensitive sensor measures the small value $\mathbf{X} - \mathbf{F_c}$, producing an output $V = S(\mathbf{X} - \mathbf{F_c})$.
- This output is applied to an actuator, which produces the balancing force $\mathbf{F_c} = AV$.

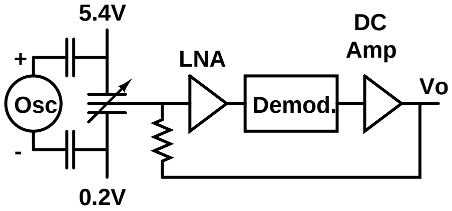
$$\mathbf{F_c} = AV = AS(\mathbf{X} - \mathbf{F_c})$$

$$\mathbf{F_c}(1 + AS) = AS\mathbf{X}$$

$$\mathbf{X} = \mathbf{F_C}(1/AS + 1) = AV(1/AS + 1) = V(1/S + A)$$

Thus V is a *linear* measure of X.

Accelerometer IC ADXL 150



Block Diagram of Accelerometer ADXL 150

- The oscillator produces out of phase signals which cancel when the moving plate is exactly centred.
- If the plate is displaced upwards, a positive phase signal is amplified by the Low Noise Amplifier LNA.
- It is detected by the phase sensitive detector (Demod).

The DC output of the detector is amplified and applied back to the centre plate.
A more positive DC bias on the centre plate reduces the electrostatic force w.r.t. the upper plate and increases it w.r.t. the lower plate, which brings it back to the centre.

Tilt Sensing for the Drone

- Our sensor board MPU 6050 contains 3 accelerometers aligned with x, y and z axes.
- 3 ADCs (with 16 bit resolution) convert the DC output of each accelerometer to digital values.
- These values are placed in $3 \times 2 = 6$ registers which are 8-bits wide and which can be read by Arduino through a serial interface.
- We measure tilt by measuring acceleration due to gravity along the three axes.
- If the drone is upright, g is felt only by the vertical axis and acceleration along the other two axes is zero.
- If there is a tilt, the angle of tilt can be computed from the ratio of the components of g along the three axes.

Measurement of Angular Velocities

- Apart from the 3 accelerometers, the MPU module 6050 contains a 3 axis gyroscope and a small microprocessor to handle serial communication and data storage.
- Isn't it amazing that 3 accelerometers, a 3 axis gyroscope, 6 ADCs with 16 bit resolution and a small micro-processor with local memory can be bought for a mere 200 Rs.?

- The gyroscope measures angular velocities around the three axes.
- Angular velocities are measured using the Coriolis force associated with a spinning body.

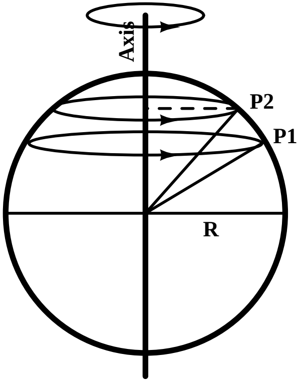
Coriolis Force

Consider a spinning spherical body like the Earth.

- An object on the spinning body will be thrown off along a tangent at its location due to its inertia. This is the centrifugal force.
- The object needs a force towards the centre of the spinning body to keep it at its position on the surface. This force is the centripetal force, – provided by gravitation in case of the Earth.
- If the object is not static on the spinning body, but moves towards one of the poles, there is another force which acts on the body.
- This force is the Coriolis force. It is the force which makes cyclones move clock wise in the norther hemisphere and anti-clockwise in the southern hemisphere.

Coriolis Force

- Consider an object moving from point P1 to point P2 on a spinning spherical body like the Earth.
- When viewed from a static frame outside the spinning body, the object has a velocity from West to East, which is maximum near the equator and zero at the north and south poles.
- As the object moves from P1 to P2 (northwards), it would keep its initial West to East velocity due to inertia.



- However, the Earth is moving at a slower surface velocity at point P2. Therefore the object will drift ahead of the Earth towards East.
- Thus it appears to have a force acting on it, pushing it eastwards. This is the Coriolis Force.

Coriolis Force

Coriolis force has been known for a long time.

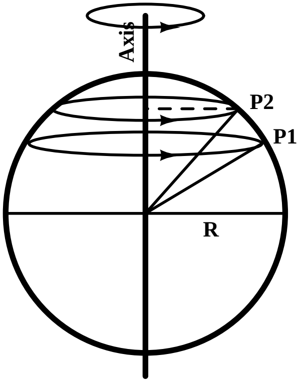
An interesting aside . . .

Here is a quote from the Wikipedia:

> *"In 1674, Claude Francois Milliet Dechales described in his* Cursus seu Mundus Mathematicus *how the rotation of the Earth should cause a deflection in the trajectories of both falling bodies and projectiles aimed toward one of the planet's poles.*
>
> *Riccioli, Grimaldi, and Dechales all described the effect as part of an argument* **against** *the heliocentric system of Copernicus.*
>
> *In other words, they argued that the Earth's rotation should create the effect, and so failure to detect the effect was evidence for an immobile Earth!"*

Measurement of angular velocity using Coriolis Force

- Coriolis force is proportional to the angular velocity.
- It will be observed only when the object moves along the axis of rotation. This effect is used in MEMS based gyroscopes.
- An inertial mass is made to vibrate along the axis of rotation.
- Coriolis force will try to displace the inertial mass perpendicular to its motion.



- This movement can be measured as was done for an accelerometer, using force balance technique.

Gyroscope to measure angular velocities

The sensor board MPU 6050 contains:

1. Three accelerometers – one along each axis, with three 16 bit ADCs to report acceleration values in digital format.
2. Three mechanical oscillators – one along each axis, which constitute a gyroscope for angular velocity measurement. Three additional 16 bit ADCs are used to convert the output voltages from the gyroscope to a digital format.
3. It can accept inputs from an external 3-axis compass to provide a complete 9-axis output to report the motion status of an object.
4. It contains more than 100 data registers, which store calibration and status data to represent the motion state of an object.
5. The MPU 6050 module supports a serial data transfer protocol (I^2C) which allows access to this data from an external controller without needing too many pins.

Calibration of the drone

- The system frame in its static state will not be exactly perpendicular to the gravitational force. The initial tilt value has to be recorded and accounted for when in flight.
- Similarly, the gyroscope may have some initial offset. This has to be measured and compensated during subsequent measurements.
- Each motor has a dead range where a low value of PWM drive produces no movement at all. Therefore a range of low PWM values has to be excluded from the motion control algorithm. This range needs to be programmed during initialisation and calibration.
- In the user interface, the amount of finger movement on the touch screen of the mobile phone used to control the drone has to be mapped to the dynamic range of useful PWM range. This will vary from phone to phone and needs to be calibrated.

Motion Control for the Drone

Speeds of rotation of the four motors on the drone are controlled based on:

1. User control based on the throttle, yaw, pitch and roll values set by the user through mobile communication.
2. PID based control values required for stabilizing the drone. These are determined through an algorithm which keeps track of set values and actual values for each of the 6 axial parameters (angular position and angular velocity).
3. Coefficients for proportional, integral and differential errors need to be optimized empirically for stable operation.

PRECAUTION

The drone contains thin and sharp blades of the propellers rotating at a very high speed.

These can cause serious harm if they colloid with people or objects.

Always use guards around the propellers and exercise extreme care when attempting to fly your drone!

Have fun building and flying your drone . . .

Do learn the underlying principles for all the mechanical design, hardware and software used for making your drone fly.

Part III

Appendix: Serial Interfaces

7. Serial Data Communication

8. Synchronization

9. Synchronous Serial I-O
   - Inter Integrated Circuit Bus: I^2C BUS
   - Serial Peripheral Interface: SPI

10. Asynchronous Serial I-O
    - Asynchronous Serial IO protocol

Serial Data Communication

Serial communications means sending data a single bit at a time.

We normally deal with multi-bit data. So why not send all bits of the data in parallel? Why bother with parallel to serial and serial to parallel conversions?

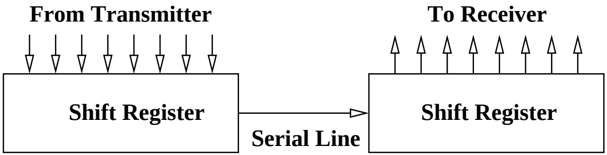
There are several situations where serial communication is preferable.

- If the transmitter and the receiver are remote, it is expensive to lay a parallel cable with multiple wires in it.
- It may be difficult to ensure that all bits on a parallel bus have the same delay. A ‘late’ bit from one transmission might arrive at the same time as an ‘early’ bit from the next. The transmission will then get garbled.
- If there are many units in a project which need to exchange data, too many pins will be required to interconnect them. The only option in this case is to exchange data using serial communication.

Serial Interfaces

How do we send the data serially?

- At the transmitter, this involves loading data in parallel to a shift register, and shifting out a bit at a time.
- At the receiving end, we have another shift register, which receives this data a single bit at a time and when all bits are received, the shift register can be read in parallel.



The transmitter and the receiver must agree on the order in which the bits will be sent.

Synchronization

- The rate at which the transmitter shifts the data out must be matched to the rate at which the receiver shifts the data in.
- Even if the clock frequencies at the transmitter and receiver are perfectly matched, the phase of the clock at the receiver should be carefully adjusted so that we sample the serial line when it is stable and not when the data on it is changing.

How can this be ensured? This can be done through

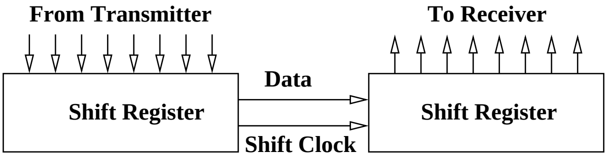
1. synchronous communication, or
2. asynchronous communication.

In our project, we shall mostly use synchronous serial communication.

Synchronous communication

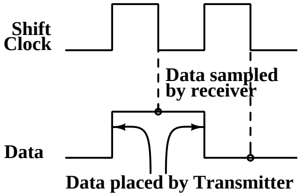
Synchronizing data transmission and reception can be easily managed If the transmitter and receiver use the same clock.

The transmitter can send the data through one wire and the shift clock through another.



The receiver uses the clock sent by the transmitter for sampling and shifting the data.

If the transmitter places data on the positive edge of the clock, while the receiver samples it at the negative edge, the data will be stable at the sampling instant.



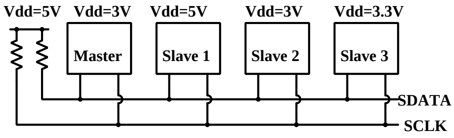
Synchronous communication

- If the clock is sent along with data, the sampling rate as well as timing for data change and sampling can be easily managed. This is called synchronous serial communication.
- However, this requires that the skew (differences in delay) between the clock and data must be very small. This will be the case if the transmitter and the receiver are located close to each other – as in our project.
- The advantage of this method is that it can achieve very high data rates.
- In this project, we shall use two popular protocols for synchronous serial communication:
    - I^{2}C bus
    - Serial Peripheral Interface or SPI.

Synchronous Serial Data Transfer using I^{2}C Bus

- The I^{2}C bus is a 2 wire serial interface originally developed by Philips.
- The name stands for Inter Integrated circuit bus. Writing 'I^{2}C' is difficult in ordinary text, so it is often called the 'I2C' bus.
- Its main application is in data communication between ICs on a board – such as peripherals chips and processors and between smart home appliances.
- Subsequent to its introduction by Philips, Intel made some modifications to it to ensure inter-operability between devices from various manufacturers. That version is sometimes referred to as System Management Bus or SMBUS.
- Arduino cards include modules to support the I^{2}C Bus/SMBUS.
- Restricted versions of this protocol are also known as the '2 wire protocol'.

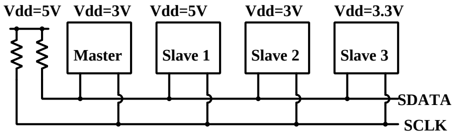
Synchronous Serial Data Transfer using I^2C Bus



Devices communicating with this protocol are connected in parallel across 2 wires, one of which carries serial data while the other carries the clock.

- All devices drive the bus lines using open drain drivers.
- Each line has a pull up resistor.
- Since devices provide only the pull down function, individual devices can use different supply voltages.
- The ability to connect devices using different supply voltages and the use of only 2 wires are the most attractive features of the I^2C Bus.

Synchronous Serial Data Transfer using I^{2}C Bus



The bus should have a master device, which provides the clock. Multiple slave devices can be connected to the bus.

- Each device has a unique address, which is normally 7 bit wide.
- The number of devices which can be connected across the bus is limited by the address space, or the maximum capacitive load of 400pF that can be placed on the bus.
- The master can act as the transmitter or the receiver.
- The addressed slave device then assumes a complementary role (receiver/transmitter).
- Multi master protocols are also supported.

Data speed on the I^{2}C Bus

- Because of its open drain drivers and resistive pull ups, this bus is used for low and medium speed communication.
- In the standard mode, Data is transferred at rates of up to 100 kbit/s on this bus.
- The protocol does not restrict one from using much lower clock frequencies.
- Data can be sent at up to 400 kbit/s in the Fast-mode and up to 1 Mbit/s in Fast-mode Plus.
- Recent versions of the bus protocol can support a speed of up to 3.4 Mbit/s in the High-speed mode.

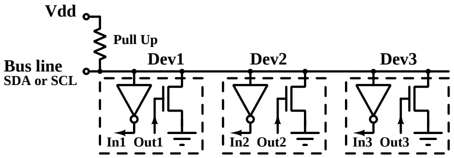
Evolution of I^{2}C Bus Specifications

Year	Version	Max. speed	Comments
1982	Original	100 kbit/s	Introduced by Philips
1992	1	400 kbit/s	Fast-mode and 10-bit addressing.
1998	2	3.4 Mbit/s	High-speed mode added.
2007	3	1 Mbit/s	Fast-mode plus with 20 mA drivers
2012	4	5 Mbit/s	Unidirectional Ultra Fast-mode (UFm) using push-pull logic without pull-up resistors

To use the 10-bit address scheme, the address frame consists of two bytes instead of one. A specific combination ('11110') of five most significant bits of the first byte is used to signal the 10-bit address mode. These 5 bits, the 10 bit address and the read/$\overline{\text{write}}$ bit make up the 16 bit address frame.

Open Drain drivers

Both lines (SDA and SCL) use open drain drivers. So it is worth recalling some characteristics of open drain logic.

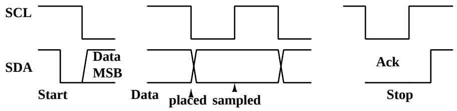


- Each driver has only a pull down transistor. Pull up is provided by the common pull up resistor on the bus.

- The bus can be 'High' only if *all* driver transistors are OFF.
- Any device can pull down the bus wire unconditionally.
- Different devices using different supply voltages can be easily connected as pull down devices.
- The bus driver transistor should be able to withstand the voltage provided by external V_{DD} – typically 5V.

Data transfer protocol

The actual data transfer is controlled by the Master, which provides the clock used for data transmission.

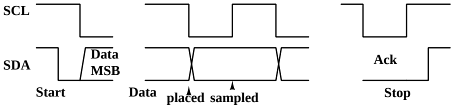


- It signals the 'Start' of transmission by a 'High' to 'Low' transition on the data line while the clock line is high.
- Similarly, the 'Stop' message is signaled by a 'Low' to 'High' transition on the data line while the clock line is high.

- These are the only instances when there are transitions on the data bus while the clock is high.
- Therefore these messages are easily distinguished from data transitions.

Data transfer protocol

For normal data transmission, the transmitter places data on the data line at the falling edge of clock.



Therefore all data transitions are seen after the clock goes low.

- The receiver samples data on the rising edge of the clock so that the data is stable at the time of sampling.
- In this protocol, the most significant bit is sent first.
- The master device can be the talker or the listener.
- Listener and talker roles are decided during the first message from the master.

Data transfer with master as the talker

Suppose the micro-controller is the master and it wants to send data serially to a slave device.

- The master generates the 'Start' message on the bus. (High SCL, 'High' to 'Low' transition on SDA).
- It then sends 8 bits on the serial data bus (Most significant bit first) which include the 7 bit address of the slave device and a $R/\overline{W}$ bit as the least significant bit.
- The last bit is 0 for a write operation (– Master being the talker in this example).
- After the transmitter has sent these 8 bits on the SDA line, the transmitter driver transistor goes off during the ninth bit time.
- The receiver pulls the data line 'Low' during this time to acknowledge successful receipt of the signal. (Recall that this is possible because of the open drain connection discussed earlier).

Data transfer with master as the talker

When the master is the talker, it generates the start condition, then sends the 7 bit address of the slave, followed by a ‘0’ to indicate that the master is the talker and the slave is the listener.

- During the ninth bit time, the driver transistor of the master is turned off and the receiver should pull down the SDA line if it received the data successfully.
- This is called the ‘Ack’ message.
- If the receiver driver transistor is also ‘OFF’ during the ninth bit period, the SDA line will be seen to be ‘High’ during this time.
- This is known as a ’Nack’ message. It is used by the slave to signal failure to receive when it is a listener (during write operation by master).

Data transfer with master as the talker

To summarize:

- The first transmission from the master establishes it as the talker and the slave as the listener if the last bit sent from the master is '0'.
- After the address has been acknowledged by the receiver, the master, (which is the talker), sends data serially to the slave receiver, checking for the 'Ack' signal at the end of each byte.
- After all bytes have been sent, the transmission is terminated by a 'Stop' signal on the line.

Data transfer with master as the listener

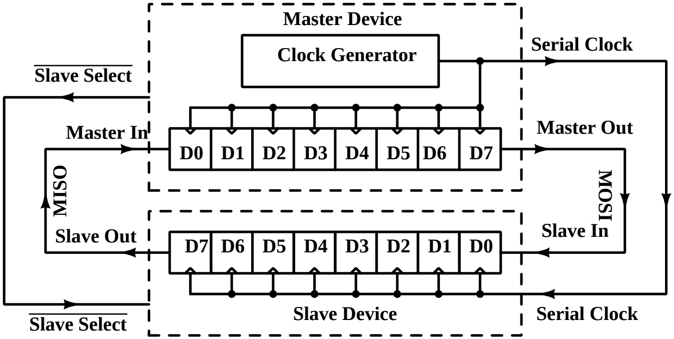
- When the master wants to be the listener, it creates the start condition, followed by a 7 bit address and then a ‘1’.
- As before, during the 9th bit time, the slave should pull down the SDA line to acknowledge that the byte has been received.
- After receiving the ‘Ack’ signal, the master releases the SDA line (turns off its open drain driver). It continues to drive SCL.
- The slave now becomes the transmitter and sends data. Each data byte is acknowledged by the master after reading it, through an ‘Ack’ signal.
- After reading the last byte, the master sends the ‘Nack’ signal to stop the slave from transmitting any further.
- The master follows up the ‘Nack’ signal with a stop condition to terminate this round of communication.

Serial Peripheral Interface: SPI

The Serial Peripheral Interface (SPI) is a synchronous serial communication interface.

- The interface was developed by Motorola in the mid-1980s and is widely used by many peripheral devices and memories.
- A distinguishing feature of this interface is that there is simultaneous transfer of data between the communicating devices.
- Thus there is no transmitter or listener – both devices send as well as receive data.
- The communicating devices are designated as master and slave. The master device provides the shift clock and initiates data transfers.

Serial Peripheral Interface: SPI

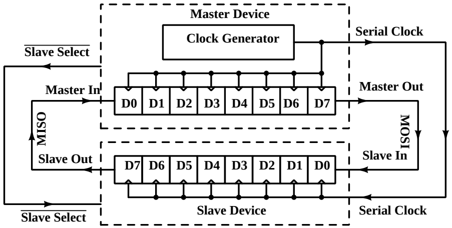


Output of the master shift register is connected to the input of the slave shift register, while the output of the slave shift register is connected to the input of the master shift register, thus forming a ring.

There is simultaneous transfer of data between the two devices with clock pulses generated by the master device.

Serial Peripheral Interface: SPI

- There is no talker or listener – both devices provide outputs and receive inputs simultaneously.
- The master device generates the clock which is used by both devices.
- The master also provides $\overline{\text{Slave select}}$ lines which enable a selected slave to take part in data transfer.



- The master can have multiple slave select lines – one for each slave.
- Slaves are connected in parallel to the data and clock signals.
- Only the selected device takes part in data transfer. Others do not drive their outputs.

Signals in SPI

The SPI interface defines the following signals:

MOSI: Master out, Slave in
: This is the output from the master and input to the slave for serial data.

MISO: Master in, Slave out
: This is the output from the slave and input to the master for serial data.

Sclk: Serial clock
: provided by the master. The master outputs a counted number of clock pulses to interchange the data in the two shift registers.

$\overline{SS_i}$: Slave select
: These are the slave select lines, one per slave. Typically, the microprocessor is the master device and multiple peripherals such as serial memory, ADC, DAC, real time clock chips etc. are the multiple slaves.

The master has to have a separate slave select line for each slave device. If the number of slaves is large and this many pins cannot be provided, a decoder can be used.

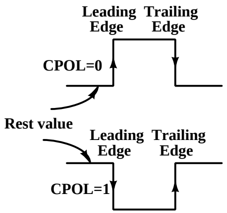
Data transfer in SPI

What if we don't want two way communication?

- Data transfer in SPI is always bi-directional.
- While simultaneous data transfer is an advantage in some applications such as data streaming, simplex (one way) data transfers are much more common.
- If the master wants only to write the data to the peripheral, it places the data it wants to write in the shift register and ignores the data that arrives from the slave.
- If the master wants only to read, it just leaves the stale data in the shift register or loads a dummy command and carries out a transfer. After the transfer, the received data is read. The slave is supposed to ignore the data it received.

Clock Polarity and Phase

The master and the slave must agree on the clock polarity and phase for data transfer. Motorola has defined four modes based on clock polarity (CPOL) and phase (CPHA) combinations.



- The clock polarity decides the resting or inactive value on the Sclk line.
- CPOL = 0 defines that the clock will rest at 0 when idle and a clock pulse will have a leading edge from $0 \rightarrow 1$ and a trailing edge from $1 \rightarrow 0$.
- CPOL = 1 defines a clock signal which rests at 1 when idle. A clock pulse consists of a $1 \rightarrow 0$ transition as the leading edge and a $0 \rightarrow 1$ transition as the trailing edge.

It is possible to convert between the two choices with a single inverter in the Sclk line.

Clock Polarity and Phase

Clock phase decides the timing of data changes on the MOSI and MISO lines.

- CPHA=0 indicates that the MOSI and MISO lines will change data on the trailing edge while the inside circuit captures data at (or shortly after) the leading edge of the clock cycle.
- The driver for the data lines should hold the data steady till the next trailing edge.
- For the first cycle of the clock, the first bit must already be on the MOSI line before the leading edge of the clock arrives.
- Thus, for CPHA=0, there is half a cycle with clock at idle, followed by half a cycle with clock asserted.

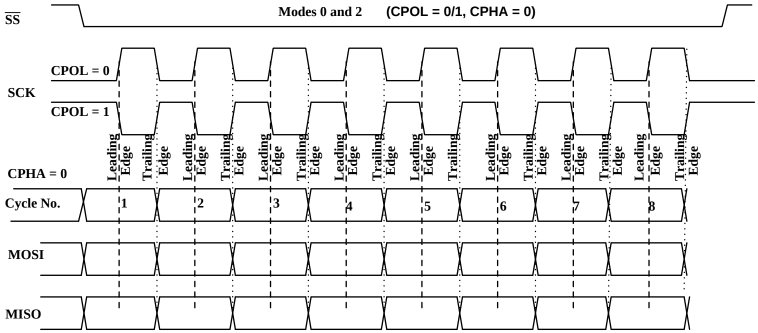
CPOL=0 and CPHA=0 are the most commonly used combination for SPI communication.

Clock Polarity and Phase

- CPHA=1 indicates that the “output” side will change data on the leading edge while the “input” side captures the data on (or shortly after) the trailing edge of the clock cycle.
- The “output” side should hold the data steady till the next leading edge.
- Thus, for CPHA=1, there is half a cycle with clock asserted, followed by half a cycle with clock idle.
- For the last cycle of the clock, drivers should hold the data steady till slave select is de-asserted.

The four combinations of clock polarity and phase are often summarised as a mode number, with CPOL as the more significant bit and CPHA as the less significant bit.

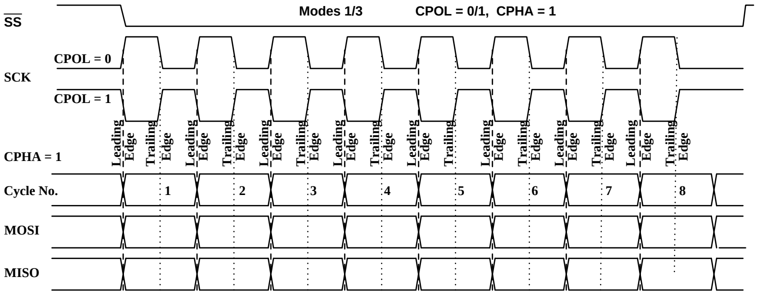
CPHA = 0 or Modes 0 and 2



Data sampled on leading edge, changed on trailing edge.
Half an idle cycle before first leading edge.

As long as the same mode is used by all devices, the exact clock frequency is immaterial.

CPHA = 1 or Modes 1 and 3



Notice that 9 clock cycles will be required for transferring 8 bits. This is because of the extra half clock cycle delay required to be inserted for ensuring that data change and sampling take place at different clock edges.

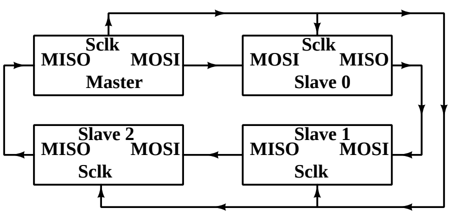
Data Size

- The most common size of data that is transferred at one go is a byte (8bits).
- However, other data sizes may be used by some devices.
- For example some audio codecs, such as the TSC2101 by Texas Instruments, use 16 bit Shift registers for transferring 16 bits of data at a time.
- Arduino supports both 8 bit and 16 bit transfers.
- Some other devices use twelve-bit shift registers – for example some 12 bit digital-to-analog or analog-to-digital converters.

Daisy chaining

One problem with SPI is that it needs a separate slave select signal for each slave. This may be a problem for many processors with a pin constraint.

Daisy Chained Slaves



- An alternative to connecting slave devices in parallel is to daisy chain these, with a single slave select used for all.
- Now we do not need multiple slave select signals.

The master can control where the data ends up by controlling the number of clock pulses to be supplied.

SPI interface for Arduino cards

Most Arduino cards support data transfer using SPI interface.

- Arduino uses the terms “Controller” for Master and “Peripheral” for Slave. Thus MOSI becomes COPI while MISO becomes CIPO.
- There is a library which supports SPI data transfer. This library becomes available if you
  #include <SPI.h>
  at the beginning of a sketch.
- Pins D11, D12 and D13 are used in Arduino UNO and Arduino Nano for COPI (MOSI), CIPO (MISO) and SCLK respectively.
- UNO allocates pin D10 for $\overline{\text{Slave-Select}}$, though any other I/O line(s) may be used for $\overline{\text{Slave-Select}}$(s).
- Several functions for setting parameters and for data transfer become available when this library is used.

Initialising the SPI bus

- SPI.begin( ); initialises the SPI bus (and the SPI class).
- SPI parameters are set by a method in SPI class as:

  SPI.beginTransaction(SPISettings(maxSpeed, BitOrder, Mode);
  For example,
  SPI.beginTransaction(SPISettings(14000000, MSBFIRST, SPI_MODE0));
- SPI.endTransaction( ) is called when you want to stop using the bus.
  It is normally called after de-asserting the $\overline{\text{Slave-Select}}$ signal.
- MSBFIRST Bit order and MODE0 are the most common settings for SPI.

Data transfer on the SPI bus

- Data transfer is carried out by the “SPI.transfer” method: receivedVal = SPI.transfer(val); for byte sized transfers and receivedVal16 = SPI.transfer16(val16); for 16bit transfers.
- Notice that SPI carries out 2 way transfers –
  so “val” or “val16” is transferred out while “receivedVal” or “receivedVal16” comes in – and is the return value from the method.
- For transfering a block of data, we may use:
  SPI.transfer(buffer, size);

  If SPI.transfer is called with 2 parameters, it carries out “in-place” data transfer for data buffers.

  The input values overwrite the outgoing values in buffer.

Asynchronous communication

- If the transmitter and the receiver are physically remote, they must use independent shift clock generators, set nominally to the same frequency.
- In this case, their clock frequencies will invariably have a small but non-zero mismatch.
- Unfortunately, this means that with time, the sampling instant will drift further and further away from the ideal sampling time.
- When the sampling time has shifted by more than a half period, the receiver might sample the wrong bit on the serial line, leading to errors.

Obviously some means of re-synchronization must be provided to avoid this and to ensure reliable transfer of data.

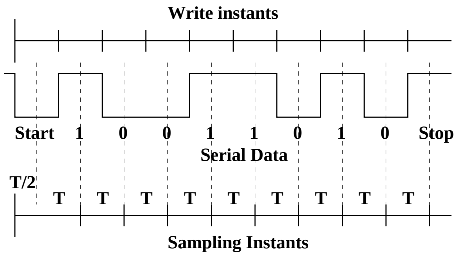
Asynchronous communication

- Synchronous communication, used for short distance data transfer, uses an additional wire for Serial Clock.
- However, for communication over relatively long distances, delay variations between the data and clock wires can introduce considerable skew between clock and data – which can lead to errors.
- So the same wire which carries data must also carry the extra information required for synchronization.
- This involves sending extra bits on the wire along with the data.
- The data bits along with these added bits constitute a **frame**

Asynchronous communication

- To synchronize two clocks, we need to create an edge.
- Serial transmission often uses ‘non return to zero’ or NRZ encoding to minimize noise and power dissipation.
- If successive bits are identical, no edge may be generated by the data itself.
- To ensure that an edge occurs at the start of every frame, we add a bit before and a bit after the useful data.
- These bits are chosen to be different – thus ensuring that each frame starts with a value different from the one with which the previous one ended.
- These are called start and stop bits.
- It is common to use a ‘1’ for a stop bit and a ‘0’ for a start bit. Then each frame begins with a negative edge.

Asynchronous communication



- Each new bit is placed on the serial line after time T.
- This is called bit time and the corresponding frequency the baud rate.
- Notice that the first sampling instant is offset from the writing instant by half a bit time.
- This is to ensure that a bit is sampled in the middle of the bit interval T, where it is stable.
- After sampling the start bit, each bit is sampled a time T after the previous bit.

Asynchronous communication

- Often a ‘Parity bit’ is also added to the data.
- The transmitter and the receiver agree to use either *even* or *odd* parity.
- The parity bit is chosen by the transmitter to be a ‘0’ or a ‘1’ so that the total number of bits in a frame is always even (or always odd).
- The receiver counts the number of ‘1’s and ensures that the count is ‘even’ (or odd) as previously agreed with the transmitter.
- If it is not, it can signal an error and the software can take appropriate action (such as a request for re-transmission).
- The use of a parity bit is optional.

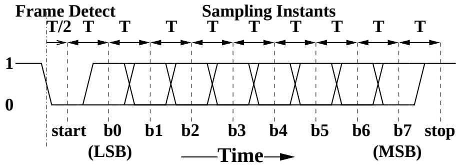
Frame synchronisation

- The receiver knows that a frame will always begin with an edge.
- It uses a locally generated clock for synchronization and timing.
- The ‘start of frame’ edge on the serial data may not coincide with an edge on the locally generated clock.
- We take the next edge of the local clock as the effective start of frame.
- The error in timing due to this could be a whole cycle of the local clock in the worst case.
- Obviously, the local clock should not operate at the data rate - a timing error of 1 bit time will then shift the data by 1 bit!
- So the local clock is operated at a multiple of the shift clock.

Frame synchronisation

- The locally generated clock should operate at a frequency much higher than the bit rate.
- This high frequency clock is then divided down to get the shift clock.
- For example, the locally generated clock might operate at 32 times the shift clock. A 5 bit counter is then used to divide the local clock.
- The worst case error in timing will now be one cycle of the locally generated (32X) clock.
- Since it is running at a frequency much higher than the shift clock, the frame synchronization error will be small.

Frame synchronisation



- Synchronization occurs at the beginning of each frame.
- Therefore timing errors do not accumulate from frame to frame.
- However, within a frame, timing errors will increase with each received bit of the frame because of small frequency difference between the transmitter and receiver shift clocks.
- The counter used to divide the local clock is reset to 0 when the 'start of frame' edge is detected.
- MSB of the counter can then be used as the sampling clock.

Frame synchronisation Example

Take the example where we have chosen the internal clock to be 32X bit rate, We reset the 5 bit counter when we detect the frame edge.

- Now, after 16 cycles of the internal clock, corresponding to time T/2 (where T is the interval between bits) the start bit is sampled.
- If it is not '0', a 'framing error has occurred and should be handled in software – say by a request to re-transmit.
- If the start bit is correctly sampled, data bits will be sampled every 32 counts, till all data bits have been sampled.
- In a typical frame, this might be 8 bits if no parity bit is appended and 9 bits if parity bit is used.
- After these bits, the next bit must be the stop bit (normally a '1'). If it is not the correct value, a frame error has occurred and must be handled appropriately.
- If the stop bit is correctly received, we accept the data and wait for the next 'start of frame'.

Frame synchronisation

Notice that this scheme will work equally well, if additional delay is inserted between the end of one frame and the beginning of the next.

- After the last bit has been sampled, no further timing needs to be performed.
- The serial line idles at the stop bit (normally ‘1’).
- The counter is reset to zero *whenever* the next frame edge is detected.
- Therefore it does not matter how much time elapses between the end of a frame and the beginning of the next.

Asynchronous Serial IO protocol

Before serial communications can be carried out, the transmitter and the receiver must agree on a number of parameters.

- At what rate will the bits be sent.
- How many bits will be sent at a time (frame size).
- Whether parity will be used and if so, which kind of parity (even or odd) will be used.
- Whether simultaneous transmission and reception (duplex) will be possible or only one of these will be carried out at a time (simplex).

These parameters constitute the asynchronous serial protocol.