EMBEDDED SYSTEM

RASPBERRY PROGRAMMING

HCMUTE – Faculty of Mechanical Engineering

Lecturer: PhD. Bui Ha Duc

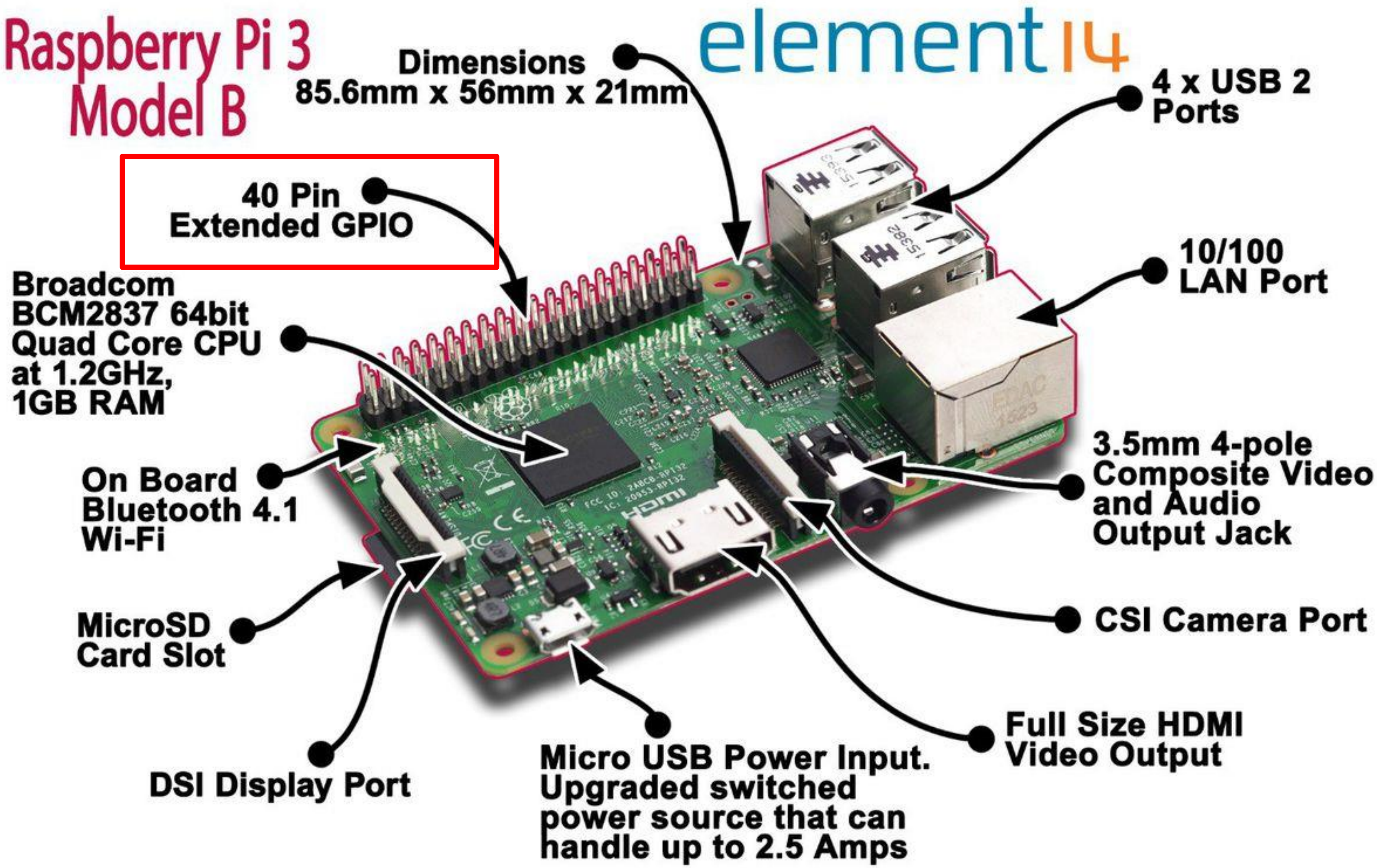
Overview

Knowledge will be covered in this chapter

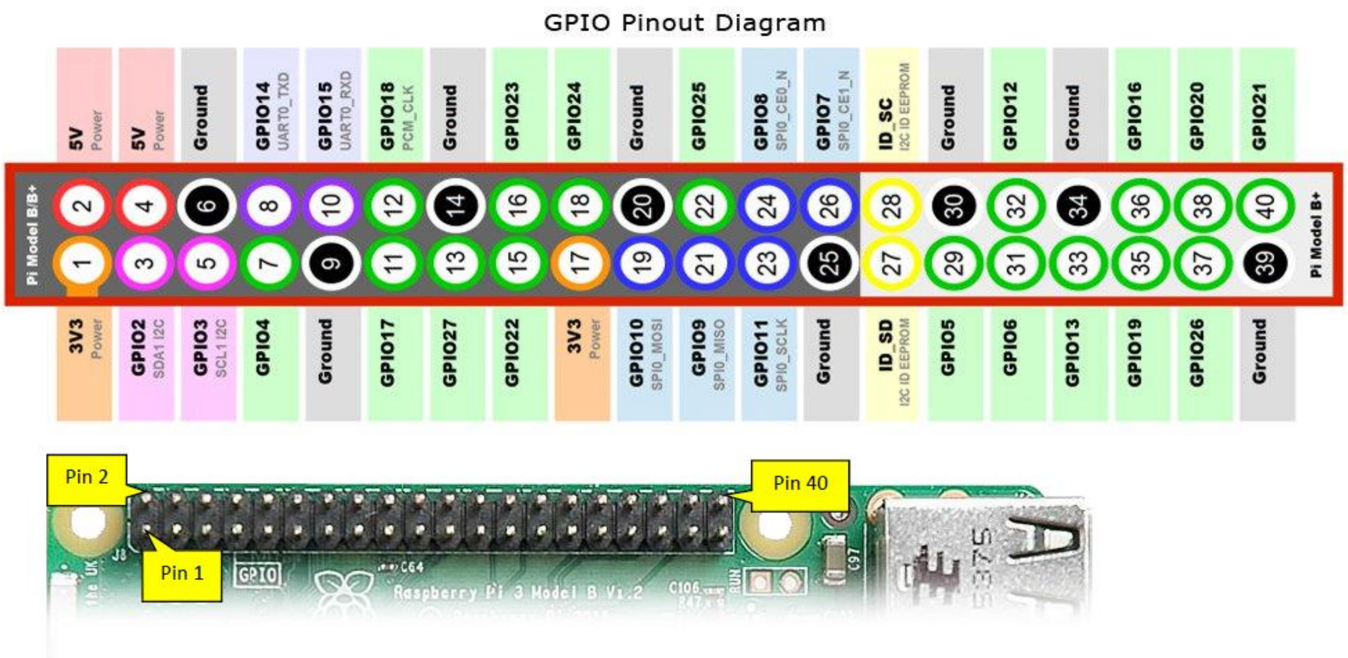
- Linux User space
- Raspberry GPIO
- Wired communication
  - SPI
  - I2C
  - UART

GENERAL PURPOSE IOs

Raspberry GPIO

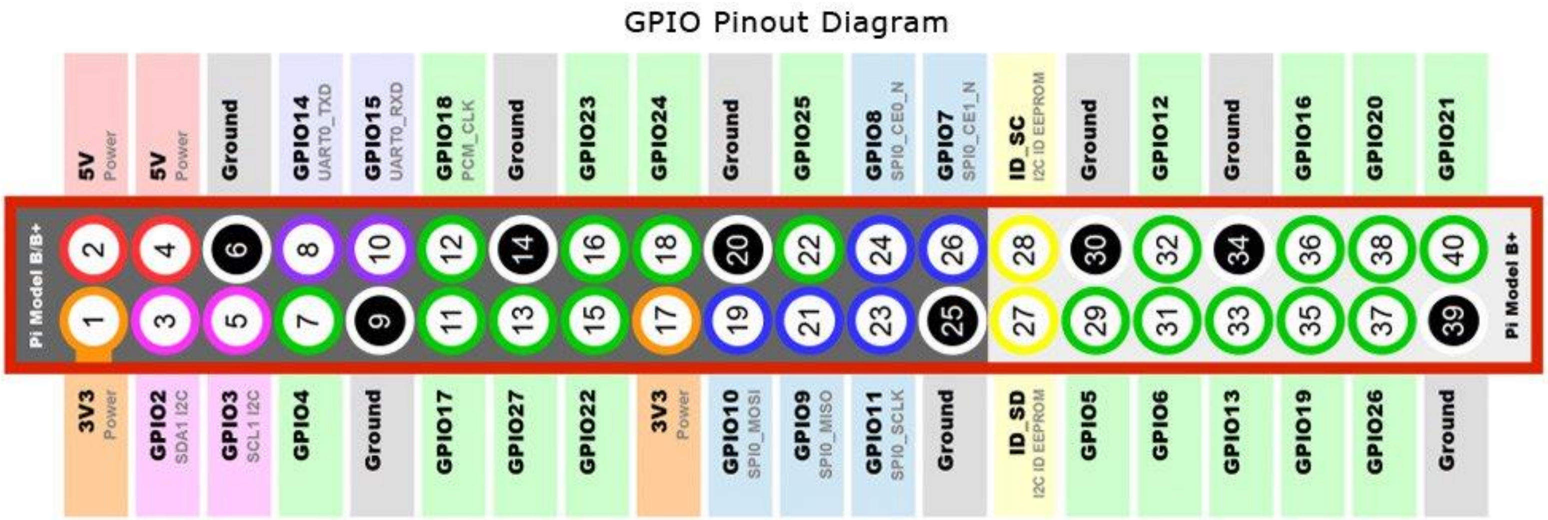


Raspberry GPIO



- Power pins:
  - 2 x 5V, 2 x 3.3V , 8 x **GND**
- **GPIO** pins
- Peripheral pins: UART, SPI, I2C

Raspberry GPIO



- **Pin numbering**:
  - There are **two numbering systems** for GPIO pins.
  - Physical numbering is the easiest to understand: it tells you where to find the pin on the physical RPi board
  - BCM numbers are only for I/O pins. They don't match the physical numbers but come from the BCM2837 processor.

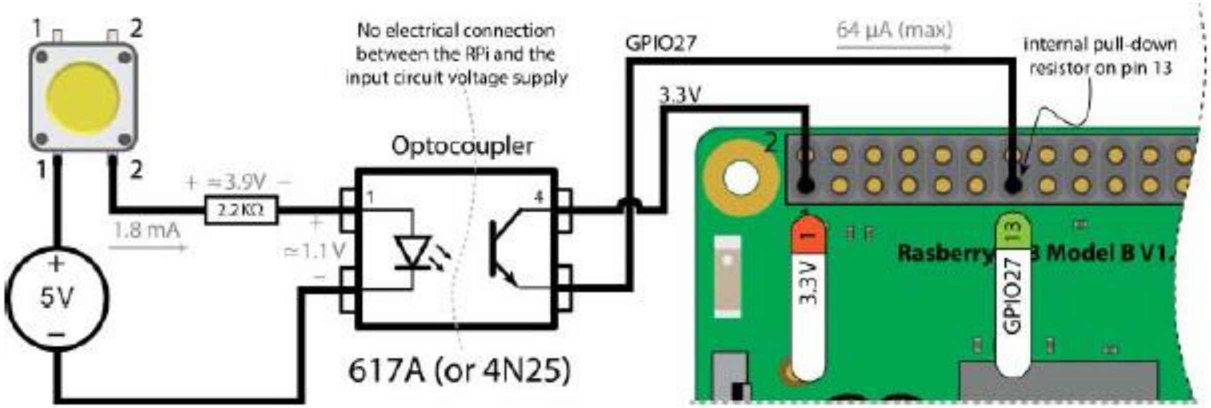
Raspberry GPIO

- **Notes:**
  - All I/O pins support interrupts
  - All pins have internal pull-up and pull down resistors
  - Several pins have shared functions

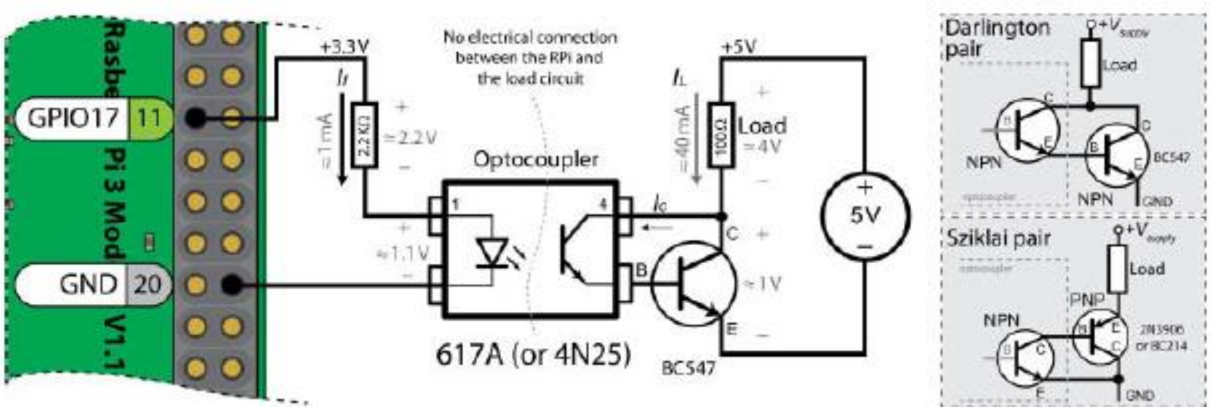
- **Cautions:**
  - Short circuits (connecting +3v or +5 either from a power pin or an output pin directly to ground) will damage the Rpi
  - All I/O pin are **3.3V**. Sending 5V directly to a pin may kill the Pi
  - Maximum permitted current draw from a pin is 50mA

How to access/control these IO?

Interfacing to Powered DC Circuits



The optocoupler input circuit

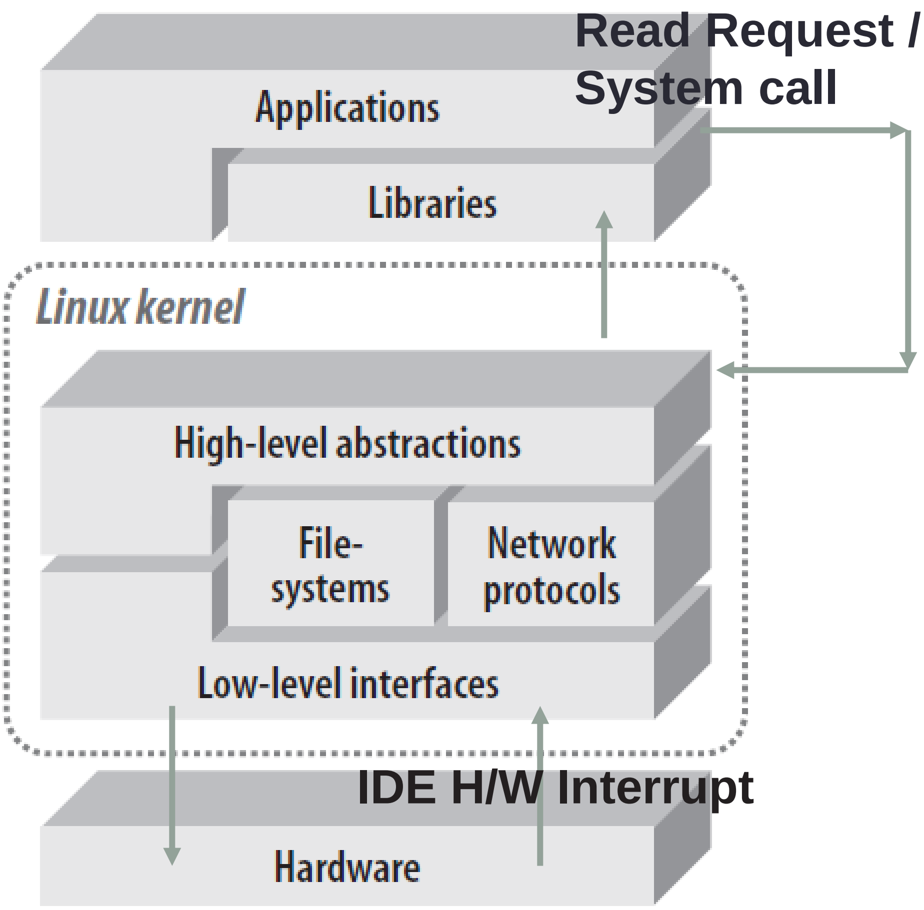


The optocoupler output circuit

Hardware accessing in Linux

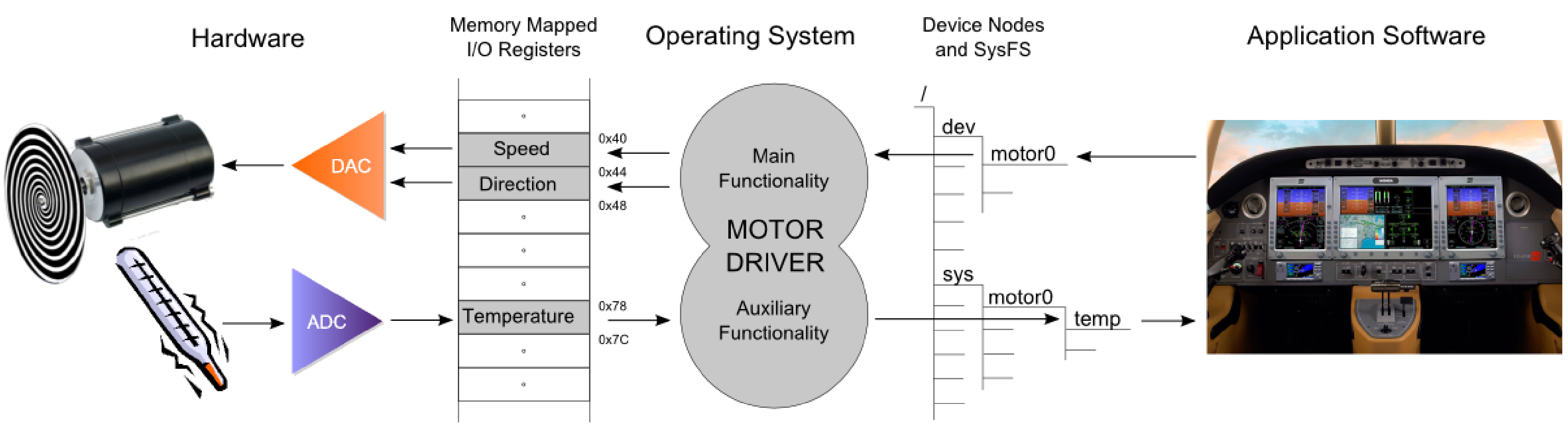
- Interfacing between hardware and software is crucial.
- The communication is handled using **memory mapped hardware device registers**
  - Device driver in kernel space
  - Application software in user space

→ A “bridge” is needed



Hardware accessing in Linux

- There are many ways to implement hardware / software bridge
  - Create a device node in /dev
  - Access device via virtual file system sysfs



https://www.missinglinkelectronics.com/devzone/index.php/source-menu-processing-sysfs

Control GPIO Using sysfs

- **sysfs** provide a virtual file system for device access at **/sys/class**
- Using sysfs, GPIO can be control in Shell language
- Functions:
  - Export the particular GPIO pin for user control.

    echo 30 > /sys/class/gpio/export
  - Change the GPIO pin direction to in/out

    echo "out" > /sys/class/gpio/gpio30/direction

    echo “in" > /sys/class/gpio/gpio160/direction
  - Change the value

    echo 1 > /sys/class/gpio/gpio30/value

    echo 0 > /sys/class/gpio/gpio30/value
  - Unexport the GPIO pin

    echo 30 > /sys/class/gpio/unexport

Control GPIO Using sysfs

- Example: Create a file name blink.sh

```
#!/bin/sh
echo 17 > /sys/class/gpio/export
echo out > /sys/class/gpio/gpio17/direction
while true
do
  echo 1 > /sys/class/gpio/gpio17/value
  sleep 1
  echo 0 > /sys/class/gpio/gpio17/value
  sleep 1
done
```

- Run the code with command: `./blink.sh`

WiringPi library

- WiringPi is a GPIO access library for Raspberry
  - wiringPi is writte in C and preinstalled in raspbian
  - Source code: https://github.com/WiringPi
  - Website: https://wiringpi.com

- Features:
  - Support command-line GPIO control in terminal
  - Support digital/analog reading and writing
  - Support wired communication: SPI, I2C
  - Provide software/hardware PWM
  - Support external interrupts, delay.

WiringPi Shell commands

(for more detail, refer to http://wiringpi.com/the-gpio-utility/)

- **gpio –v** : prints wiringPi version.
- **gpio mode <pin> in/out/pwm/clock/up/down/tri**
  - sets the mode for a pin
- **gpio write <pin> 0/1** : sets an output pin to high (1) or low (0)
- **gpio read <pin>** : Reads and prints the logic value of the given pin
- **gpio edge <pin> rising/falling/both/none** : enables the given pin for edge interrupt triggering

WiringPi Shell commands

- **gpio readall** : reads all the normally accessible pins and prints a table of their numbers

```
pi@raspberrypi:~$ gpio readall
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 |     |     |    3.3v |      |   |  1 || 2  |   |      | 5v      |     |     |
 |   2 |   8 |   SDA.1 | ALT0 | 1 |  3 || 4  |   |      | 5v      |     |     |
 |   3 |   9 |   SCL.1 | ALT0 | 1 |  5 || 6  |   |      | 0v      |     |     |
 |   4 |   7 | GPIO. 7 |   IN | 1 |  7 || 8  | 1 | ALT5 | TxD     | 15  | 14  |
 |     |     |      0v |      |   |  9 || 10 | 1 | ALT5 | RxD     | 16  | 15  |
 |  17 |   0 | GPIO. 0 |   IN | 0 | 11 || 12 | 0 | IN   | GPIO. 1 | 1   | 18  |
 |  27 |   2 | GPIO. 2 |   IN | 0 | 13 || 14 |   |      | 0v      |     |     |
 |  22 |   3 | GPIO. 3 |   IN | 0 | 15 || 16 | 0 | IN   | GPIO. 4 | 4   | 23  |
 |     |     |    3.3v |      |   | 17 || 18 | 0 | IN   | GPIO. 5 | 5   | 24  |
 |  10 |  12 |    MOSI | ALT0 | 0 | 19 || 20 |   |      | 0v      |     |     |
 |   9 |  13 |    MISO | ALT0 | 0 | 21 || 22 | 0 | IN   | GPIO. 6 | 6   | 25  |
 |  11 |  14 |    SCLK | ALT0 | 0 | 23 || 24 | 1 | OUT  | CE0     | 10  | 8   |
 |     |     |      0v |      |   | 25 || 26 | 1 | OUT  | CE1     | 11  | 7   |
 |   0 |  30 |   SDA.0 |   IN | 1 | 27 || 28 | 1 | IN   | SCL.0   | 31  | 1   |
 |   5 |  21 | GPIO.21 |   IN | 1 | 29 || 30 |   |      | 0v      |     |     |
 |   6 |  22 | GPIO.22 |   IN | 1 | 31 || 32 | 0 | IN   | GPIO.26 | 26  | 12  |
 |  13 |  23 | GPIO.23 |   IN | 0 | 33 || 34 |   |      | 0v      |     |     |
 |  19 |  24 | GPIO.24 |   IN | 0 | 35 || 36 | 0 | IN   | GPIO.27 | 27  | 16  |
 |  26 |  25 | GPIO.25 |   IN | 0 | 37 || 38 | 0 | IN   | GPIO.28 | 28  | 20  |
 |     |     |      0v |      |   | 39 || 40 | 0 | IN   | GPIO.29 | 29  | 21  |
 +-----+-----+---------+------+---+----++----+---+------+---------+-----+-----+
 | BCM | wPi |   Name  | Mode | V | Physical | V | Mode | Name    | wPi | BCM |
 +-----+-----+---------+------+---+---Pi 3---+---+------+---------+-----+-----+
```

WiringPi functions in C

(for more detail, refer to http://wiringpi.com/reference/core-functions/)

- **wiringPiSetup ()** : initialize WiringPi library
- **pinMode (int pin, int mode)** : sets the mode of a pin
- **digitalWrite (int pin, int value) :** Writes the value **HIGH** or **LOW** (1 or 0) to the given pin
- **digitalRead (int pin)** : returns the value read at the given pin
- **analogRead (int pin)** : returns the value read on the supplied analog input pin

GPIO control with wiringPi

```
#include <stdio.h>
#include <wiringPi.h>

// LED Pin - wiringPi pin 0 is BCM_GPIO 17.
#define LED     0

int main (void) {
  printf ("Raspberry Pi blink\n") ;
  wiringPiSetup () ;              // note the setup method chosen
  pinMode (LED, OUTPUT) ;

  for (;;) {
    digitalWrite (LED, HIGH) ;  // On
    delay (500) ;               // delay 500 mS
    digitalWrite (LED, LOW) ;   // Off
    delay (500) ;
  }
  return 0 ;
}
```

GPIO control with wiringPi

- Compile and run the blink program

```
gcc -Wall blink.c -o blink -l wiringPi
./blink
```

- The program will run forever, stop it with Ctrl-C

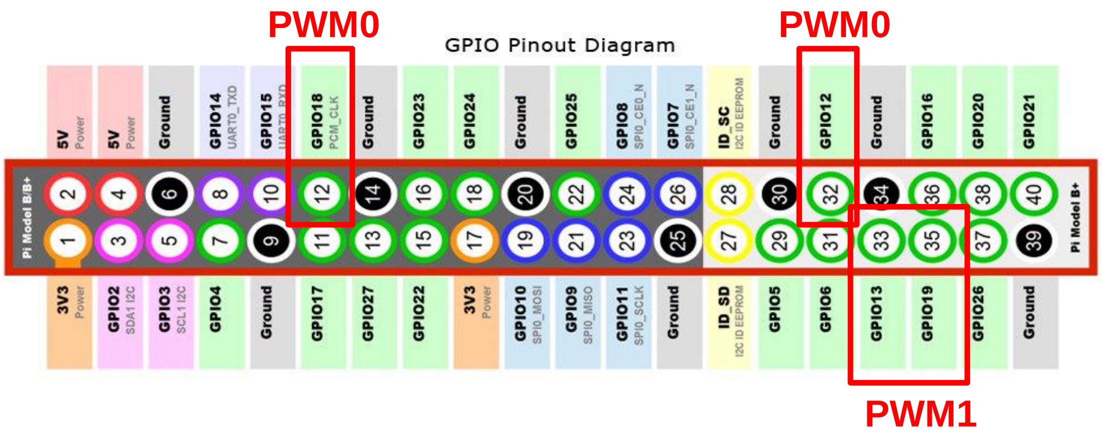
WiringPi GPIO Interrupt handling

- Structure of an interrupt program

```
void interrupt_func (void)
{
    // your code
}

int main(void)
{
    wiringPiSetup();
    wiringPiISR (pin, edge_detect, &interrupt_func);
}
```

PWM with WiringPi

- **Hardware PWM:**
  - PWM0 – Pin 12, 32
  - PWM1 – Pin 33, 35



- These PWM uses fixed base-clock frequency: 19.2 MHz
- Set PWM frequency:

PWM frequency = 19.2 MHx / (**divisor** x **range**)

pwmSetClock (int divisor)
(0-4095)

pwmSetRange(uint range)
(0-1024, default 1024)

PWM with WiringPi

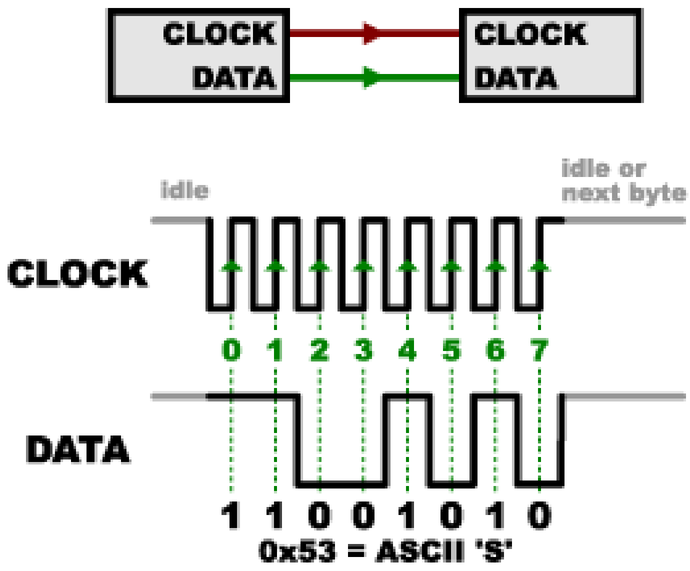
- **Software PWM:**
  - SoftwarePWM can be generate on any GPIO pins
  - Base frequency: 100Hz
  - Duty cycle: 0 – 100 (10ms)
- To use softPwm:

```
#include <wiringPi.h>
#include <softPwm.h>

int main(){
   wiringPiSetup();
   softPwmCreate (int pin, int initialValue, int pwmRange) ;
   …
   softPwmWrite(int pin, int value);
```

SPI INTERFACE

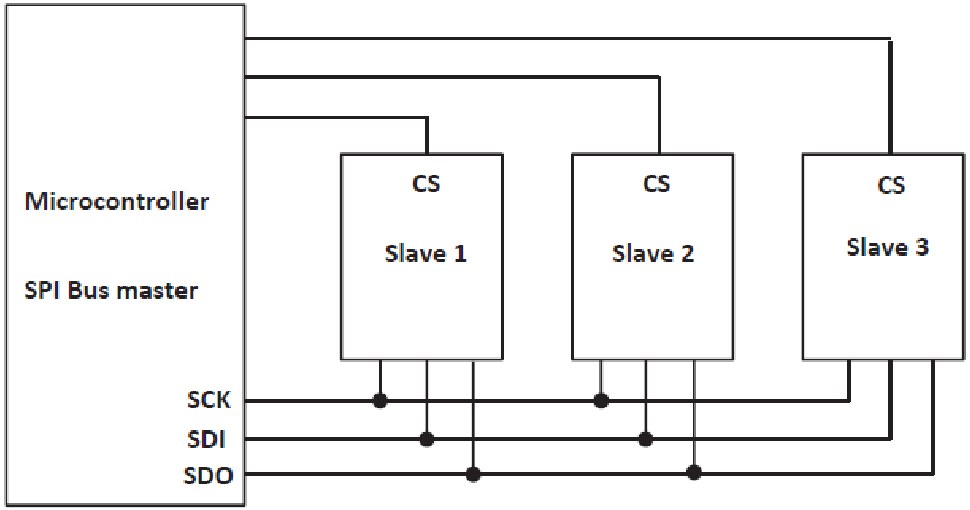
SPI - Serial Peripheral Interface

- SPI is a “synchronous” data bus
- St uses separate lines for data and a “clock” that keeps both sides in perfect sync.



SPI - Serial Peripheral Interface

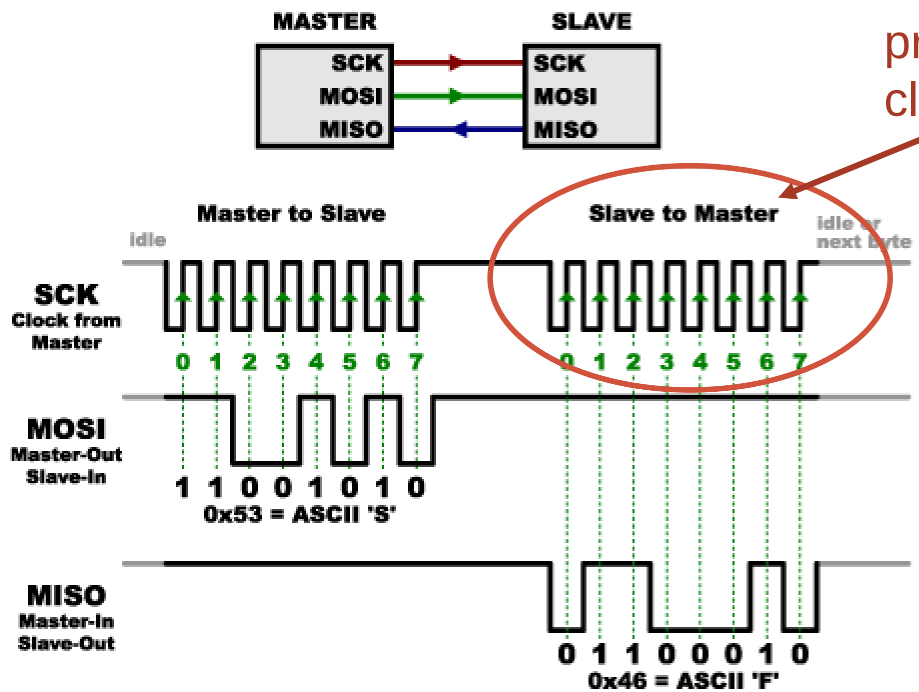
- In SPI, only one side generates the clock signal (usually called CLK or SCK for Serial ClocK).
- The side that generates the clock is called the **“master”**, and the other side is called the **“slave”**.



Receiving data from Slaves

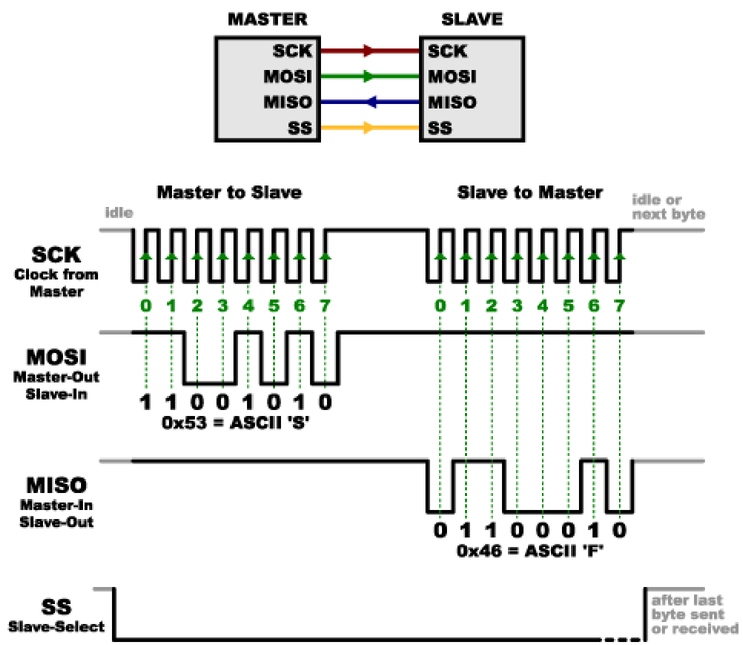
- How do you send data back from slaves to master?
  - master always generates the clock signal
  - MCU must know in advance when a slave needs to return data and how much data will be returned

prearranged clock cycles



Slave select

- MCU tells a slave that it should wake up and receive / send data via **Slave select (SS)** line
- SS = 1 : disconnects the slave from the SPI bus.
- SS = 0 activates the slave.



SPI Serial Frame Format

- SPI Serial Frame Format is selected based on connected devices
- The SPI interface format can be configured with
  - Clock Polarity – CPOL
  - Clock Phase – CPHA
- Clock Polarity Bit – CPOL
  - Clock Polarity = 0, the SCK line idle state is LOW.
  - Clock Polarity = 1, the SCK line idle state is HIGH.
- Clock Phase Bit – CPHA
  - Clock Phase = 0, the data is sampled on the first SCK clock transition.
  - Clock Phase = 1, the data is sampled on the second SCK clock transition.

SPI Serial Frame Format

CPOL = 0, CPHA = 0

- received data is sampled on the SCK line rising edge
- transmitted data is changed on the SCK line falling edge.

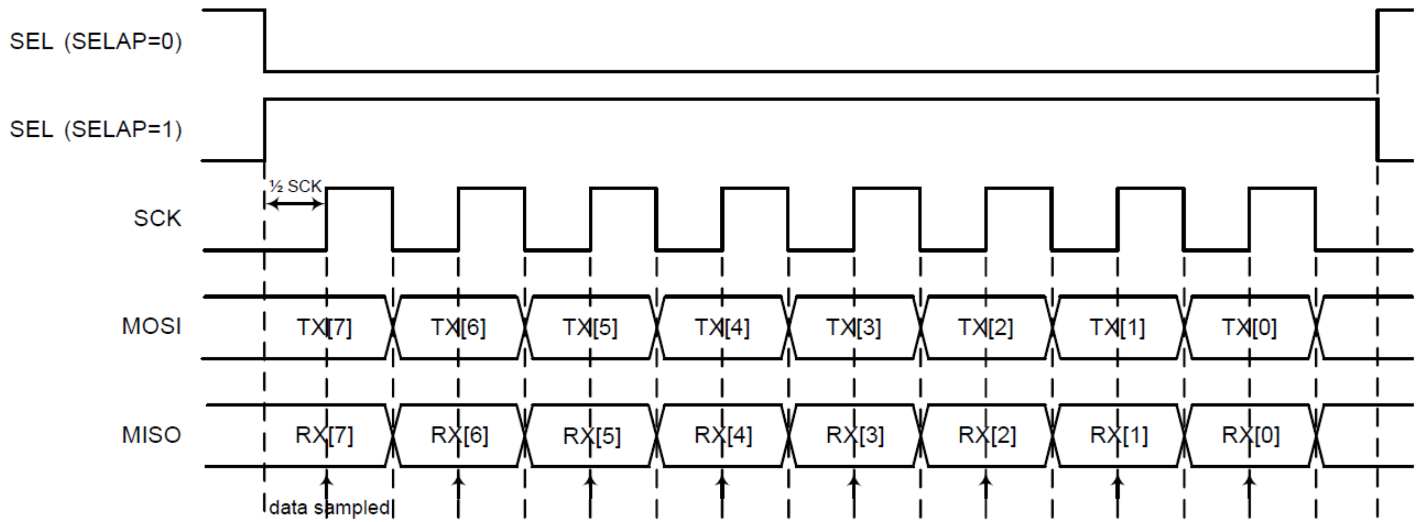


Figure 151. SPI Single Byte Transfer Timing Diagram – CPOL = 0, CPHA = 0

SPI Serial Frame Format

CPOL = 0, CPHA = 1

- received data is sampled on the SCK line falling edge
- transmitted data is changed on the SCK line rising edge
- SEL signal must remain active until the last data transfer has completed.

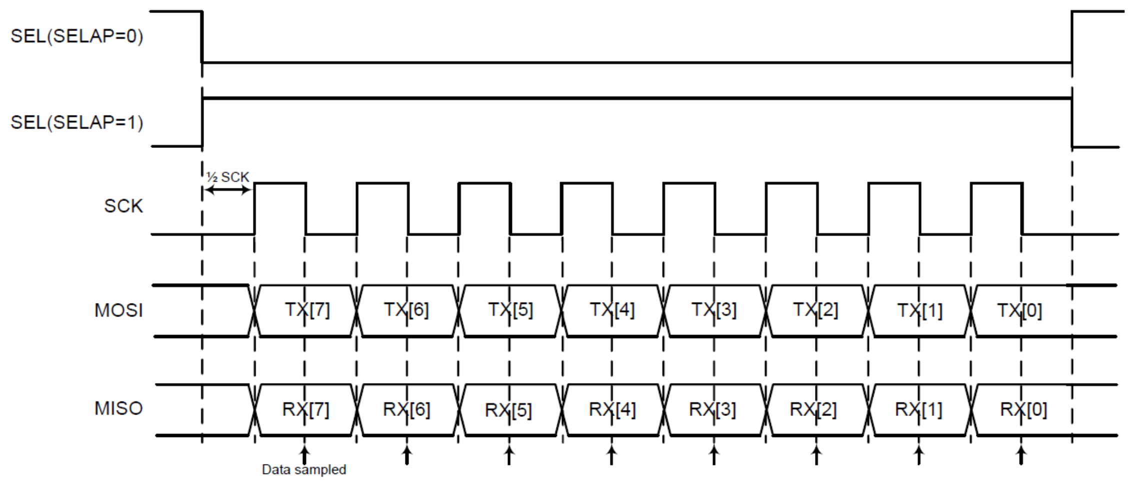


Figure 153. SPI Single Byte Transfer Timing Diagram – CPOL = 0, CPHA = 1

SPI Serial Frame Format

CPOL = 1, CPHA = 0

- received data is sampled on the SCK line falling edge
- transmitted data is changed on the SCK line rising edge
- SEL signal must change to an inactive level between each data frame.

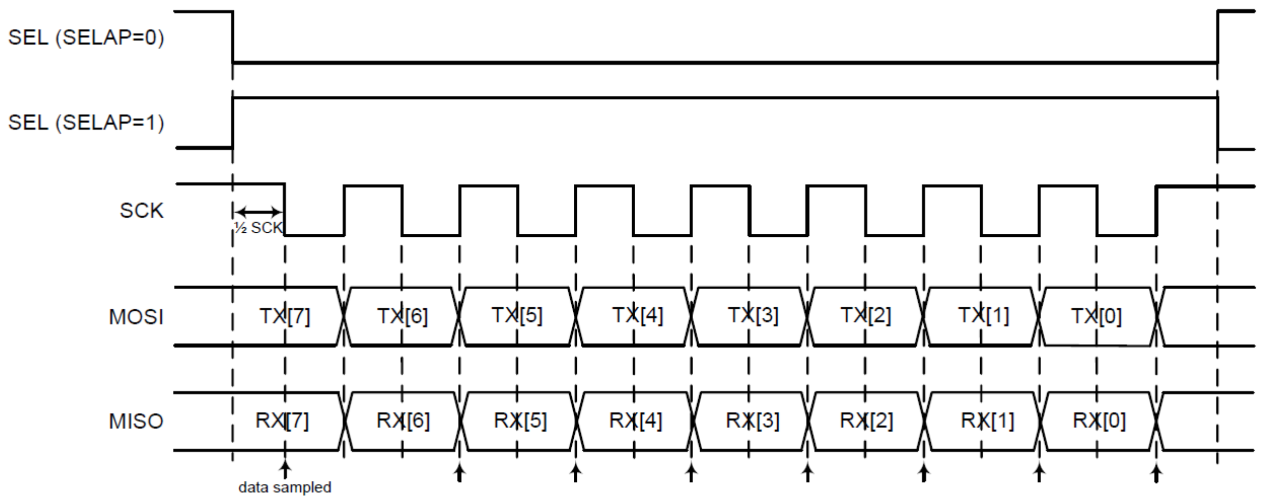


Figure 155. SPI Single Byte Transfer Timing Diagram – CPOL = 1, CPHA = 0

SPI Serial Frame Format

CPOL = 1, CPHA = 1

- received data is sampled on the SCK line rising edge
- transmitted data is changed on the SCK line falling edge
- signal must remain active until the last data transfer has completed.

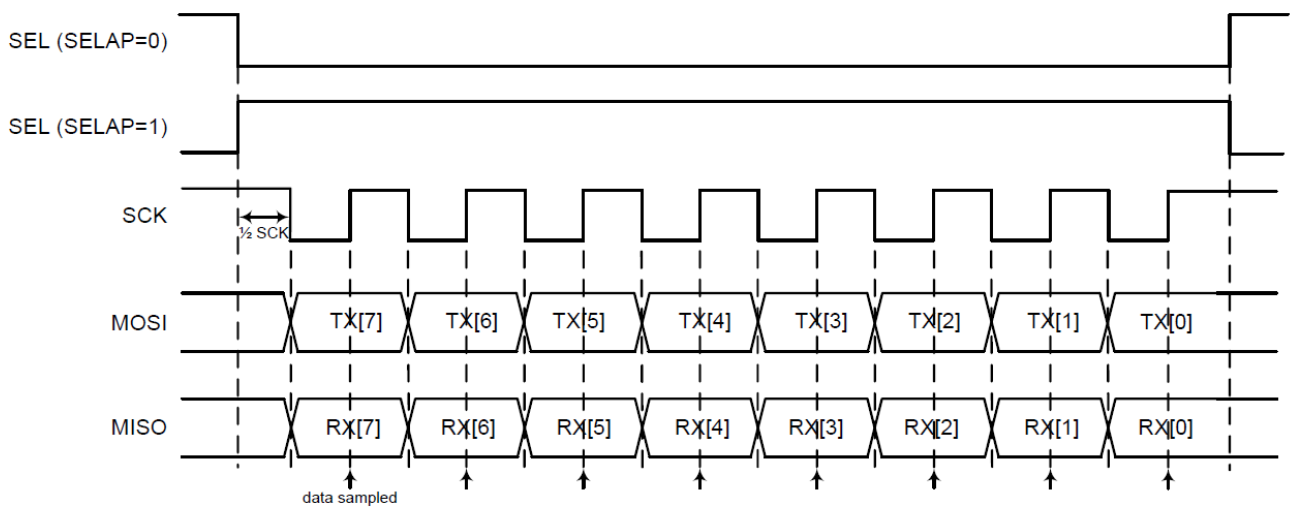
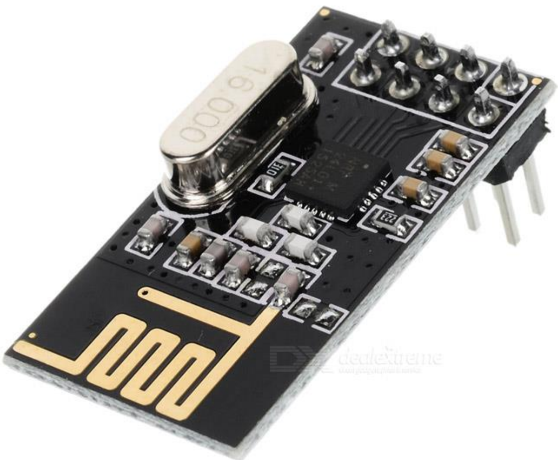


Figure 157. SPI Single Byte Transfer Timing Diagram – CPOL = 1, CPHA = 1

Example

- nRF24L01
  - Digital interface (SPI) speed 0-8 Mbps

Pin Name	Direction	TX Mode	RX Mode	Standby Modes	Power Down
CE	Input	High Pulse >10µs	High	Low	-
CSN	Input	SPI Chip Select, active low			
SCK	Input	SPI Clock			
MOSI	Input	SPI Serial Input			
MISO	Tri-state Output	SPI Serial Output			
IRQ	Output	Interrupt, active low			



nRF24L01

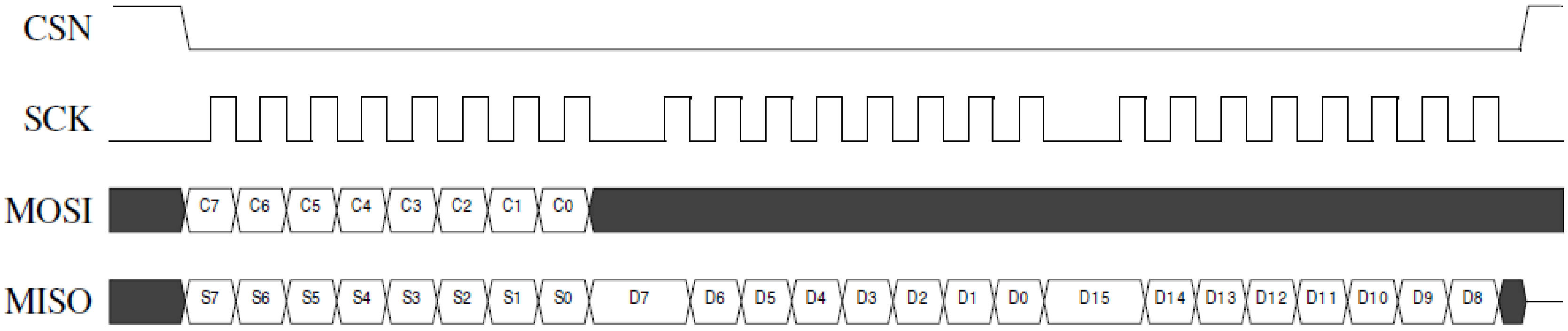
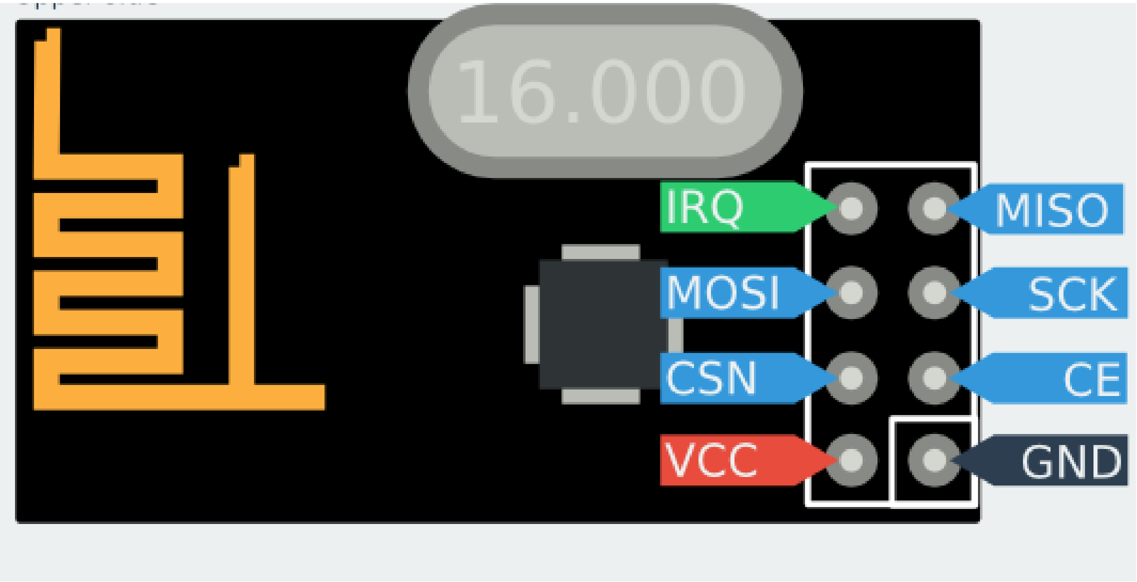


Figure 8 SPI read operation.

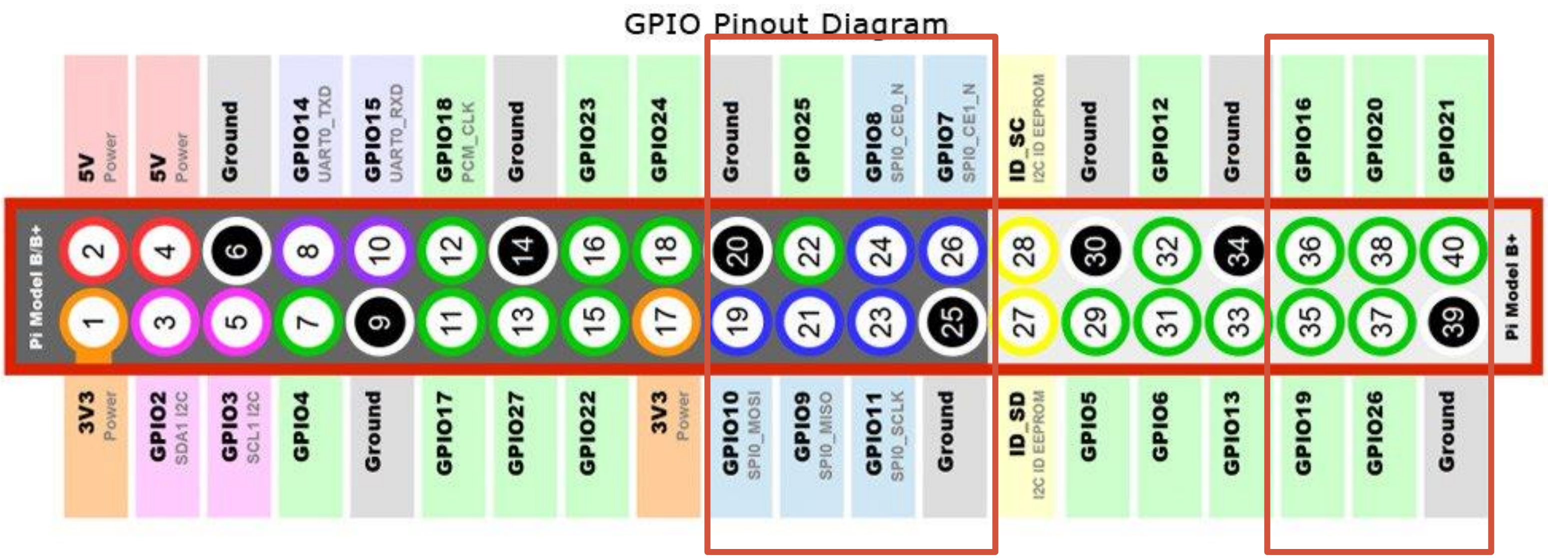
Frame Format ?



How to connect to HT32F5 MCU?

Pin	nrf24		MCU
1	GND		GND
2	VCC		3.3V
3	CE	Chip Enable	output
4	CSN	Chip Select Not	output
5	SCK	SPI Shift Clock	SCK
6	MOSI	Master-Out-Slave-In	SDO
7	MISO	Master-In-Slave-Out	SDI
8	IRQ	Optional Interrupt Request	input

Raspberry SPI



Raspberry Pi is equipped with **two** SPI bus: SPI0 and SPI1

SPI0	
MOSI : pin 19	
MISO : pin 21	
SCLK : pin 23	pin 24 : CE0
GND : pin 25	pin 26 : CE1

SPI1	
MISO : pin 35	pin 36 : CE0
	pin 38 : MOSI
GND : pin 39	pin 40 : SCLK

Raspberry SPI

- **WiringPi** includes a library for Raspberry Pi's SPI interface
- To use SPI interface, follow these steps:
  - **Step 1**: load the SPI drivers into the kernel with command:

    ```
    gpio load spi
    ```
  - **Step 2**: In C file, include the wiringPi SPI library with

    ```
    #include <wiringPiSPI.h>
    ```
  - **Step 3**: Initialize SPI with:

    ```
    wiringPiSPISetup (int channel, int speed)
    ```

    channel : 0 or 1;   speed in range 500,000Hz through 32,000,000Hz
  - **Step 4**: transfer data with :

    ```
    wiringPiSPIDataRW (int channel, unsigned char *data, int len);
    ```

    Data that was in buffer is **overwritten by data returned** from the SPI bus

Example

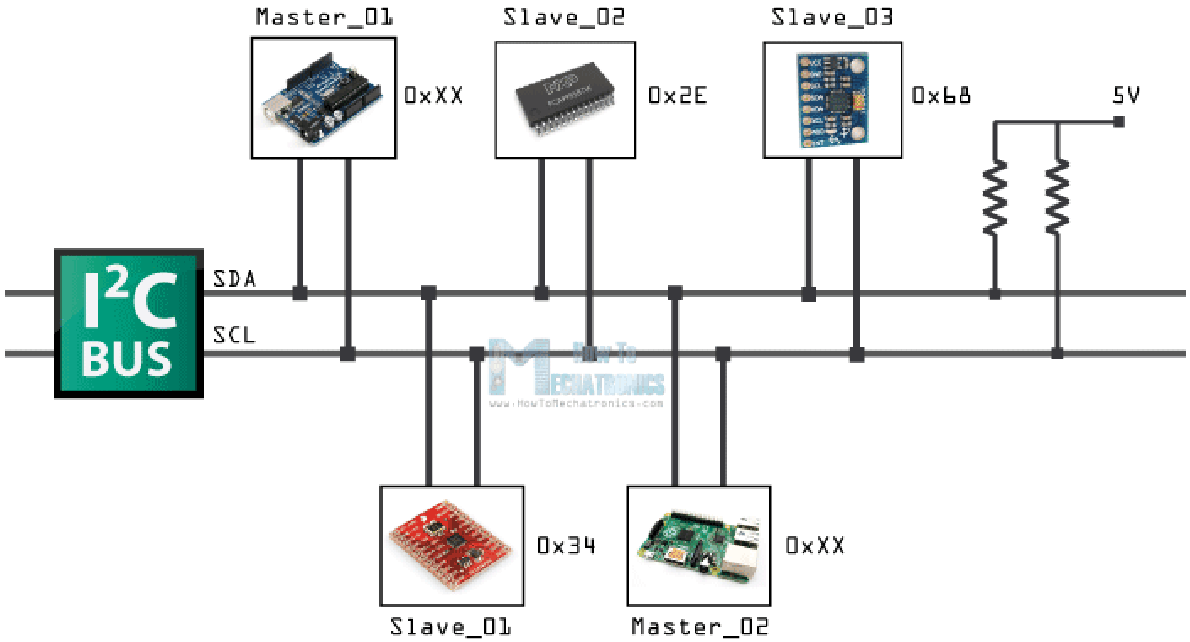
```
int main(void)
{
    unsigned char buffer[100];
    // initialize SPI1 interface, speed 500 kHz
    wiringPiSPISetup(1, 500000);

    // send and read 1 byte
    buffer[0] = 0x76;
    wiringPiSPIDataRW(1, buffer, 1);

    // send and read 2 byte
    buffer[0] = 0x7e;
    buffer[1] = 0x65;
    wiringPiSPIDataRW(1, buffer, 2);

    return 0;
}
```

I2C INTERFACE

Inter-Integrated Circuit

- I^2C is an industry standard two line serial interface used for connection to external hardware
  - serial data line – SDA
  - serial clock line – SCL
  - The SCL and SDA lines are both bidirectional and must be connected to a pull-high resistor
  - Both pin are active low
- I^2C module has 3 data transfer rates:
  - 100 kHz - Standard mode
  - 400 kHz - Fast mode
  - 1MHz - Fast-mode plus

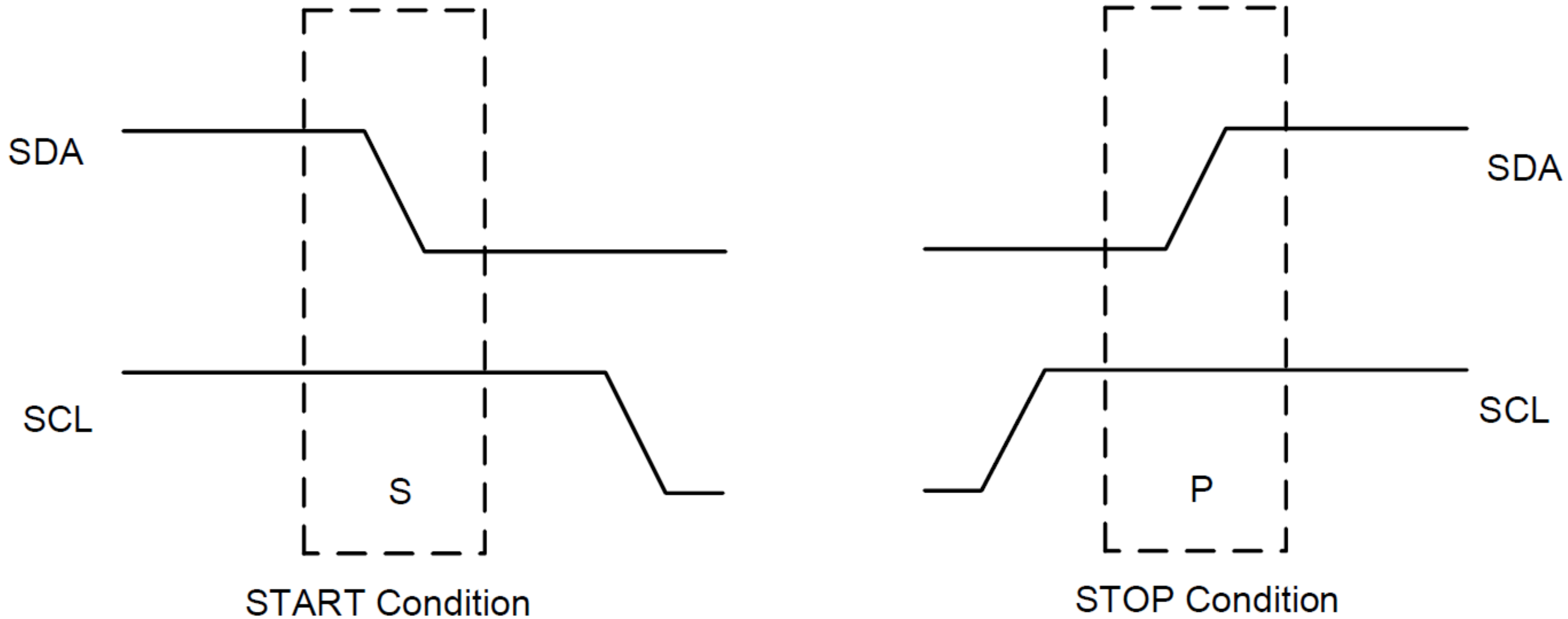
Inter-Integrated Circuit



- Can have more than 1 master
- Each device has an address
- SCL line is the clock signal
  - Synchronize the data transfer between the devices on the I2C bus
  - Generated by the master device
- SDA line is used for the transmission and reception of data

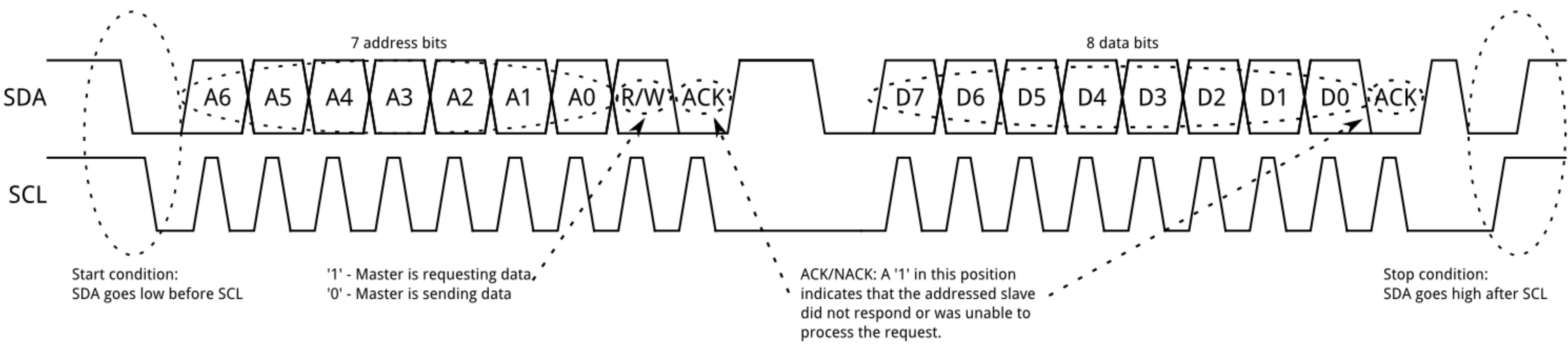
How I^{2}C works

- **START and STOP Conditions**
  - A master device can initialize a transfer by sending a START signal (“S” bit) and terminate the transfer with a STOP signal (“P” bit).

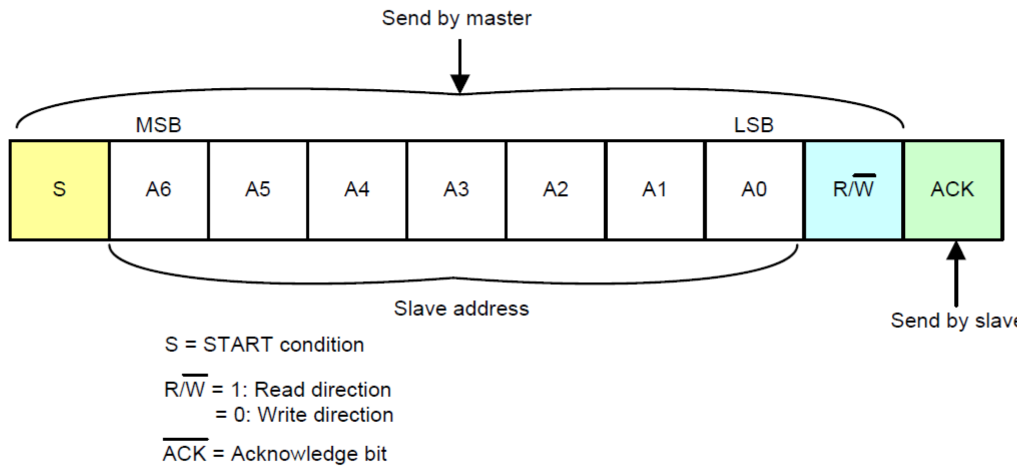


How I^{2}C works

- **Transmit and Receive data**



How I²C works



Addressing Format

- 7-bits Address Format:

7-bit Addressing Mode

  - slave address 7 bits [A6:A0]
  - $R/\overline{W}$ bit
    - $R/\overline{W}$=0 (Write): The master transmits data to the addressed slave.
    - $R/\overline{W}$=1 (Read): The master receives data from the addressed slave
  - Acknowledge bit ACK
    - ACK = 0: Slave successfully received the previous sequence
    - ACK = 1: Slave didn’t receive the previous sequence
  - It is forbidden to own the same address for two slave devices

Write data

Single-Byte Write Sequence

Master	S	AD+W		RA		DATA		P
Slave			ACK		ACK		ACK	

Burst Write Sequence

Master	S	AD+W		RA		DATA		DATA		P
Slave			ACK		ACK		ACK		ACK	

Write Sequence on MPU-6050

Read data

Single-Byte Read Sequence

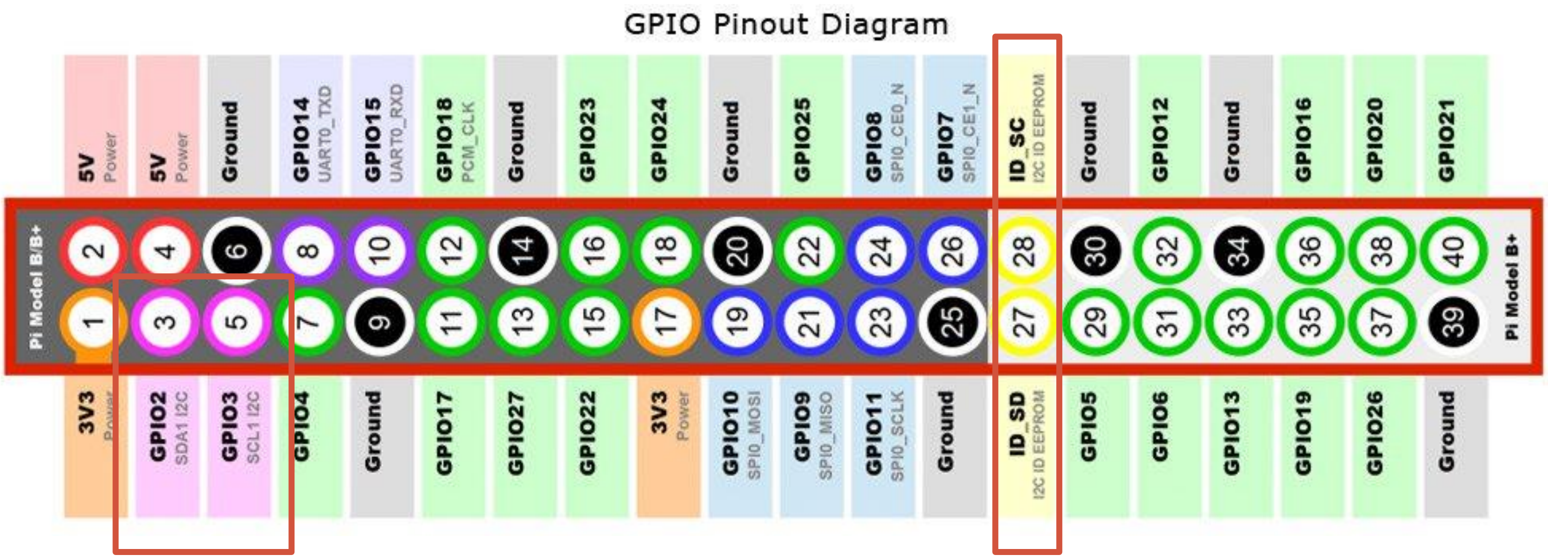
Master	S	AD+W		RA		S	AD+R			NACK	P
Slave			ACK		ACK			ACK	DATA		

Burst Read Sequence

Master	S	AD+W		RA		S	AD+R			ACK		NACK	P
Slave			ACK		ACK			ACK	DATA		DATA		

Read Sequence on MPU-6050

Raspberry I2C



Raspberry Pi is equipped with **two** I2C bus: i2c-0 and i2c-1

i2c-0	
IDSD : pin 27	pin 28 : IDSC

I2c-1
SDA : pin 3
SCL : pin 5

Raspberry I2C

- **WiringPi** includes a library for Raspberry Pi's I2C interface
- To use I2C interface, follow these steps:
  - **Step 1**: load the I2C drivers into the kernel with command:

    `gpio load i2c`
  - **Step 2**: Get slave address with command:

    `i2cdetect -y 1` or `gpio i2cdetect`
  - **Step 3**: In C file, include the wiringPi I2C library with

    `#include <wiringPiI2C.h>`
  - **Step 4**: Initialize I2C with:

    `wiringPiI2CSetup (device_address);`

```
pi@raspberrypi:~/$ i2cdetect -y 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: 60 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Raspberry I2C

- **Step 5**: transfer data with :

```
wiringPiI2CRead (int fd);
wiringPiI2CWrite (int fd, int data);
```

Write an 8 or 16-bit data value into the device register

```
wiringPiI2CWriteReg8 (int fd, int reg, int data);
wiringPiI2CWriteReg16 (int fd, int reg, int data);
```

Read an 8 or 16-bit value from the device register

```
wiringPiI2CReadReg8 (int fd, int reg);
wiringPiI2CReadReg16 (int fd, int reg);
```

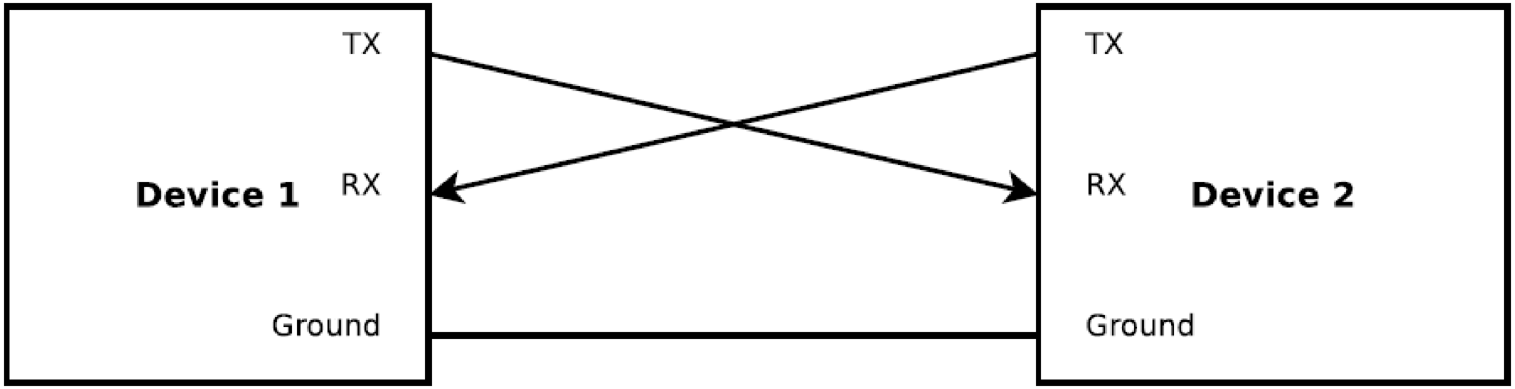
Example

```
int main(void)
{
   int fd;
   int data;

   // initialize I2C interface, speed 100 kHz
   fd = wiringPiI2CSetup(0x60);

   // send 16bit data to register
   wiringPiI2CWriteReg16(fd, 0x40, 0xfff);

   // read 16bit data from a register
   data = wiringPiI2CReadReg16(fd, 0x36);

   return 0;
}
```

UART INTERFACE

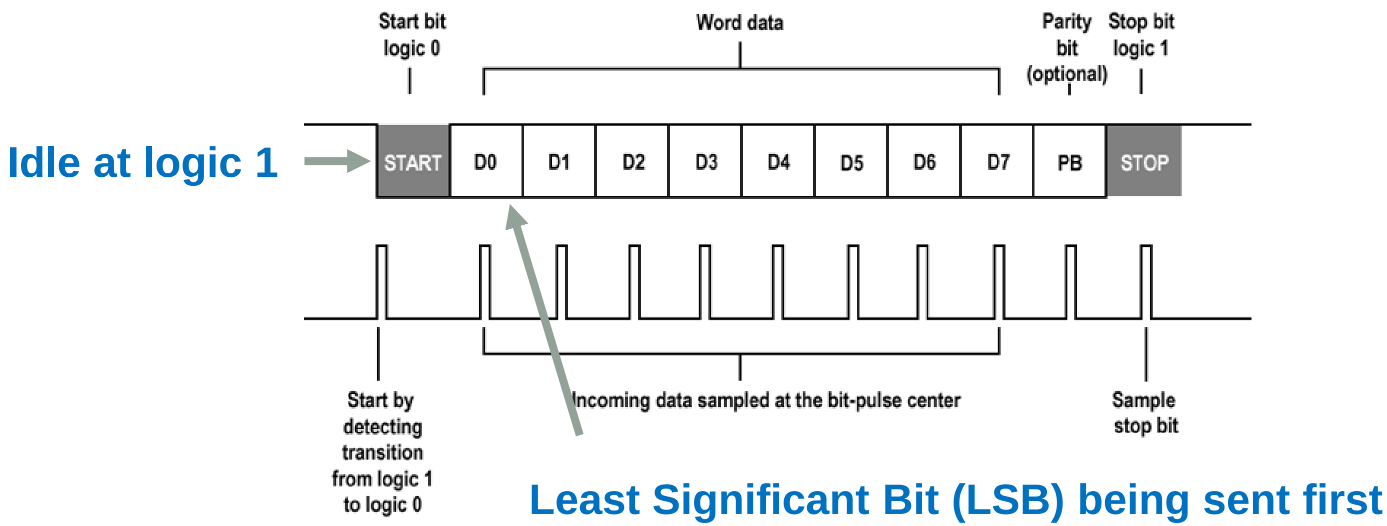
UART

- UART - **U**niversal **A**synchronous **R**eceiver/**T**ransmitter
- It is commonly referred as **serial port**
- It is a peripheral for point-to-point communication between two devices.
- Communication occurs **in serial**, i.e. one bit at time
- Two communication PINs: **RX** and **TX**



UART

- When no transmission, the line is set to **Logical “1”**
- **0011**Data are sent character by character (e.g. “C”, hexcode **43**, binary **0100)**
- First a **Logical “0”** is transmitted, called **start bit**
- Then the byte is transmitted **LSB first**
- An additional **parity bit** may follow (not in the example); it used for error checking
- One or two **stop bits** (**Logical “1”**) ends the transmission



UART

- Each bit in the transmission is transmitted for exactly the same amount of time as all of the other

- The sender does not know when the receiver has “looked” at the value of the bit.

- To send/receive data the sender and receiver clocks must be running at the same speed
  - **Baud Rate** represents the number of bits that are actually being sent over the media

- The following parameters must be set in the UART hardware:
  - **transmission speed**, in **bps = Bit Per Second** or **baud**
  - **number of bits per character**, usually **8**
  - **presence/absence of partity bit**, usually **absent**
  - **number of stop bits**, usually **1**
- A setting **19200,8,N,1** means:
  - speed = 19200 bit-per-second;
  - bits = 8;
  - parity = None;
  - stop bits = 1.

Raspberry UART

Raspberry 3 has **2 built-in** UARTs, both are **3.3V**

- PrimeCell UART (PL011)
  - Connected to Bluetooth module → need to reassign
  - Driver: **/dev/ttyAMA0**
  - Tx: pin 8; Rx: pin 10
- Mini UART
  - Used for Linux console output
  - Driver: **/dev/ttyS0**
  - Baudrate link to core frequency of VPU → need to fix VPU frequency at 250MHz
  - Lack of flow control → prone to losing data

UART function reassign

- To change Raspberry UART function, open /boot/config.txt in terminal

  `sudo geany /boot/config.txt`

- Add a line to config.txt to reassign uart:
  - `dtoverlay=pi3-disable-bt` : disables the Bluetooth device and restores /ttyAMA0 to GPIOs
  - `dtoverlay=pi3-miniuart-bt` : switches the Raspberry Pi 3 Bluetooth function to use the mini UART (ttyS0), and restores /ttyAMA0 to GPIOs.
- disable the Bluetooth system service with:
  - `sudo systemctl disable hciuart`
- Reboot raspberry

Test UART module

Raspberry UART

- **WiringPi** includes a library for UARTinterface
- To use UART interface, follow these steps:
  - **Step 1**: In C file, include the wiringPi UART library with:

```
#include <wiringPiSerial.h>
```

  - **Step 2**: Initialize serial interface:

```
SerialOpen(char *device, int baud)
```

```
Vd: fd = serialOpen ("/dev/ttyAMA0", 115200);
```

  - **Step 3**: Send/Read data via UART with

```
serialPutchar (int fd, unsigned char c) ;
serialPrintf (int fd, char *s) ;
serialGetchar (int fd)
```

Step 4: Check if data available with:

```
if(serialDataAvail (int fd))
```

Example

```
#include <stdio.h>
#include <wiringPi.h>
#include <wiringPiSerial.h>
int main (void)
{
   int fd ;
   fd = serialOpen ("/dev/ttyAMA0", 115200);
   wiringPiSetup ();
   printf("serial test begin... \n");
   serialPrintf(fd, "Hello guys \n");
   while (1){
        while (serialDataAvail (fd)){
            serialPutchar(fd, serialGetchar(fd)) ;
        }
   }
   serialClose(fd) ;
   return 0 ;
}
```