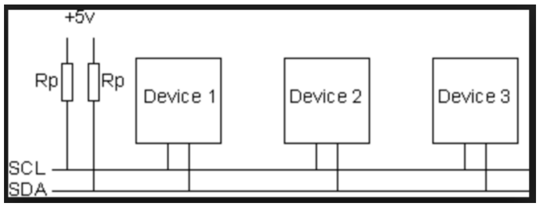
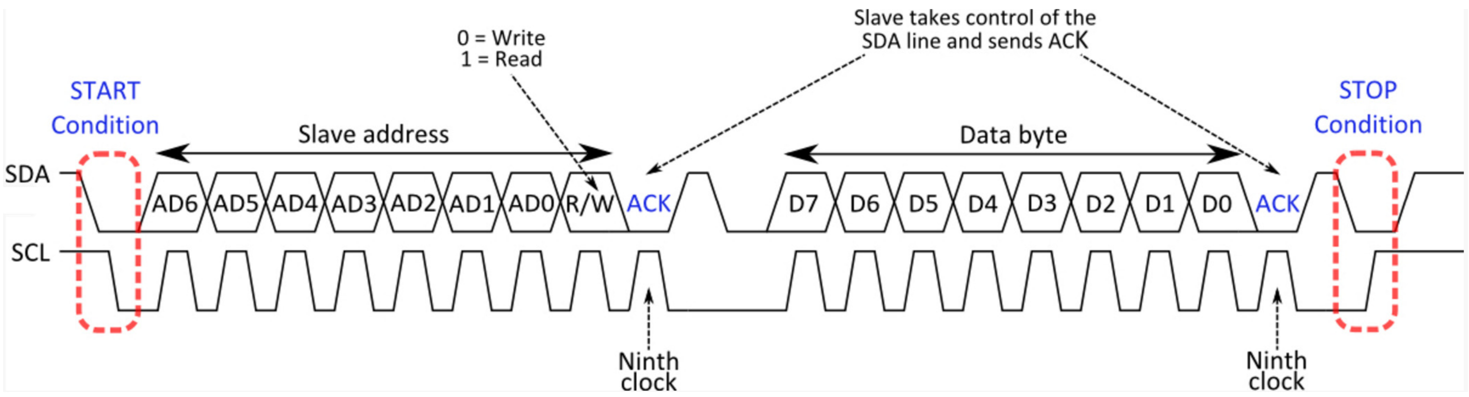
I^{2}C – Inter-integrated Circuit

The i^{2}c has two lines: a data line (**S**erial **DA**ta or SDA) and a clock line (**S**erial **CL**ock or SCL).
Just two traces can connect a bunch of peripherals to your microprocessor.



The pull-ups are commonly 4.7kΩ.
SCL and SDA are high (+5V) unless the master or slave pulls them low.

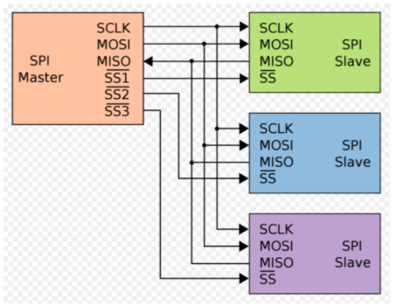
The 'master' provides all the SCL and most of the SDA signal.
The 'slave' indicates acknowledgement by pulling SDA low (ACK).
In read operations, roles switch: the master ACKs and the slave does data.


START Condition
0 = Write
1 = Read
Slave takes control of the SDA line and sends ACK
STOP Condition
Slave address
Data byte
SDA
SCL
AD6 AD5 AD4 AD3 AD2 AD1 AD0 R/W ACK
D7 D6 D5 D4 D3 D2 D1 D0 ACK
Ninth clock
Ninth clock

Note: SDA does not change while SCL is high – except during the START Condition and STOP Condition.

SPI – Serial Peripheral Interface

SPI has an output data line (**M**aster **O**ut **S**lave **I**n or MOSI), input data line (MISO), a clock (SCLK), and select lines.



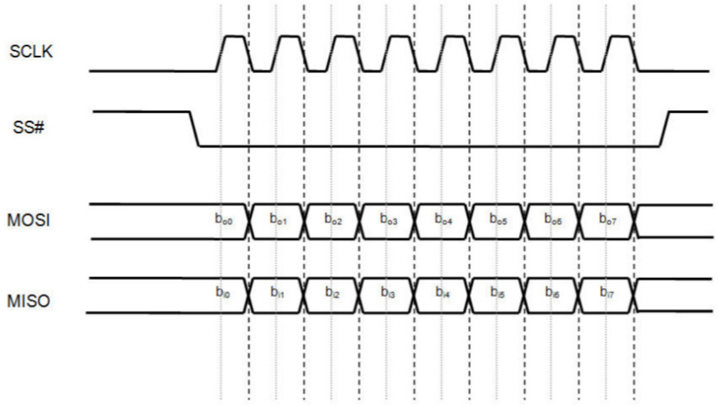

SCLK
SS#
MOSI
MISO
b_{o0} b_{o1} b_{o2} b_{o3} b_{o4} b_{o5} b_{o6} b_{o7}
b_{i0} b_{i1} b_{i2} b_{i3} b_{i4} b_{i5} b_{i6} b_{i7}

Figure 2 : A simple SPI communication. Data bits on MOSI and MISO toggle on the SCLK falling edge and are sampled on the SCLK rising edge. The SPI mode defines which SCLK edge is used for toggling data and which SCLK edge is used for sampling data.

Images on this page are from: robot-electronics.co.uk, artekit.eu, Wikipedia, byteparadigm.com

Main Loop Scheduler

Embedded systems often need to run multiple tasks. You do not need a real-time operating system to do that! A much simpler approach is usually better. Use a "main loop scheduler" along the lines of the following:

```
while (1)
{
  Task1();
  Task2();
  Task3();
}
```

The idea is simple: The main loop runs all the tasks that need to be run, over and over, until the system is shut down or reset.

This design is ***easy to understand and debug***, and leads to ***very robust*** systems. Avoid real-time operating systems unless you really need them!

Go one step further – avoid interrupts unless you really need them! Polling is easy to understand and debug.

Tasks

Write the subroutine for each task so it can ***pick up where it left off*** the last time it was called. To do this, it must keep track of its ***state***.

Whenever a task has to wait for something to happen, the subroutine should return – that way other tasks have a chance to run. For example, if a task needs to read a line of ASCII text input from a UART, the subroutine should return if the UART has no new data available. If the UART has a new input character available, the subroutine's state must be updated to incorporate the new input – for example, the subroutine's state might have a string buffer that contains the ASCII collected previously, and an index variable that tells it where to append the new ASCII character into that string buffer.

If a task has a huge amount of processing to do, the processing should be broken down into smaller pieces. The subroutine should return when it completes a smaller piece of processing– that way other tasks have a chance to run. The subroutine must keep state information so it can proceed to the next piece of processing the next time it is called.