Microprocessors

Fall 2020

10. Serial Communications II

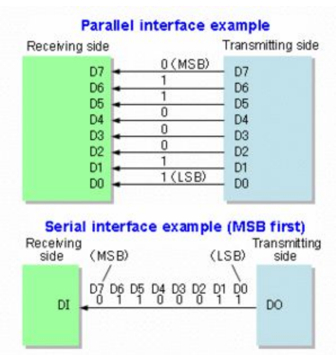
Tentative Weekly Schedule

- Week x1 - Introduction to Course
- Week x2 - Architecture
- Week x3 - Assembly Language Introduction
- Week x4 - Assembly Language Usage, Memory and Faults
- Week x5 - Embedded C and Toolchain
- Week x6 - Exceptions and Interrupts
- Week x7 - GPIO, External Interrupts and Timers
- Week x8 - Timers
- Week x9 - Serial Communications I
- **Week xA - Serial Communications II**
- Week xB - Analog Interfacing
- Week xC - DMA
- Week xD - RTOS
- Week xE - Wireless Communications

Review: Serial vs. Parallel Communication

There are two approaches for transmitting data between devices.

- **Parallel communication** - data is chunked into multiple bits, and sent at the same time using multiple **same length** channels
  - Faster, more wires and I/O, properly length match in hw, crosstalk (EMI)
- **Serial communication** - data is chunked into bits, and sent **one bit at a time** using one channel
  - Slower, one wire

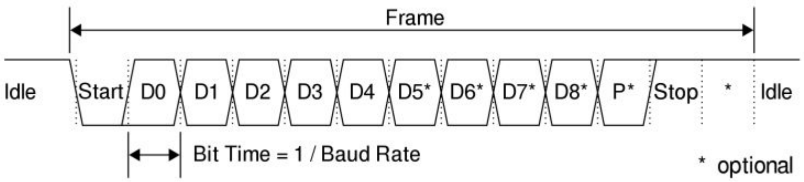


Review: Serial Communication Protocol Types

- **Synchronous** - includes a clock line
  - typically controlled by one device, and all transmitted bits are synchronized to that clock
  - each transmitted bit is valid at a defined time after a clock's rising, or falling edge depending on the protocols
- **Asynchronous** - does not include a clock line. Instead each computer needs to provide its own clock source for timing reference
  - Usually devices must agree on a clock frequency beforehand, and the actual frequencies must be very close to the agreed frequency
  - Requires a start condition (start bit) to synchronize the clocks

Review: UART

- **Asynchronous** serial communication
- Uses **TTL** levels with either 0 - 3.3V or 0 - 5V
- Data is transmitted at a specific **baud rate** with **LSB first**
  - up to 1 Mbps
  - common ones are **9600** and **115200** bps
- Point-to-point communication with two pins. **TX** and **RX**
- Configurable data bit size - 5, 6, 7, 8, 9
- Start and stop bits with optional parity bit
  - Most common is **8N1** - 8 bits, No parity, 1 stop bit
- **Idle** is logic 1, **start** is logic 0, **stop** is logic 1

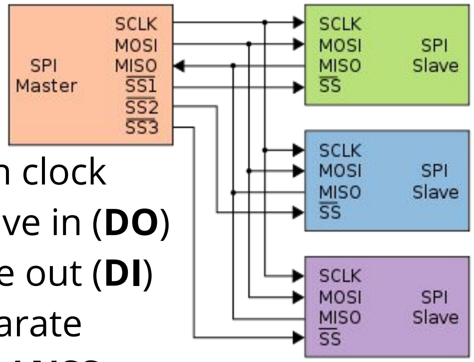


UART Alternatives

- Sensitive to noise and signal degradation.
- Fine for point-to-point communications, but if need to talk to more devices, becomes problematic
  - two lines per device for the MCU which can quickly fill up the available I/O
- MCU manufacturers have been developing their own serial communication systems
  - **I2C** - Inter Integrated Circuit communication
  - **SPI** - Serial Peripheral Interface
- Many MCUs support these interfaces

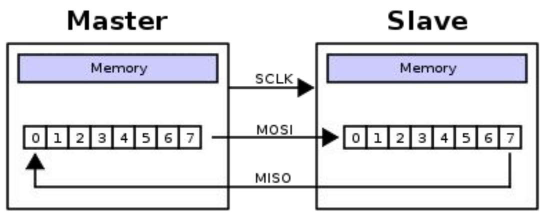
Serial Peripheral Interface (SPI)

- **Serial Peripheral Interface (SPI)** is a **synchronous** serial communication interface
- Developed by Motorola in 1980s
- Typically used in SD cards, EEPROMs, sensors and LCDs
- All chips share bus signals
  - **SCK / SCLK** - Common clock
  - **MOSI** - Master out slave in (**DO**)
  - **MISO** - Master in slave out (**DI**)
- Each chip requires a separate select line - **CS / SS / SS# / NSS**



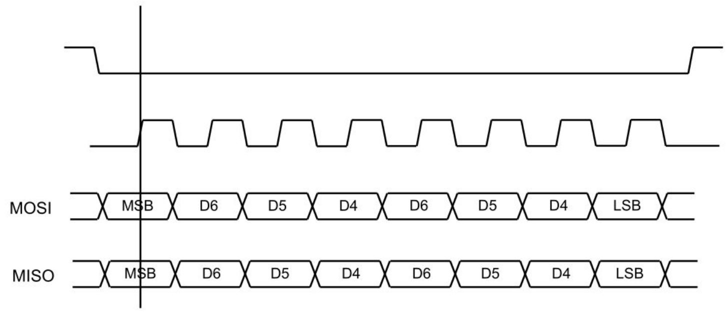
SPI Data transmission

- **Single Master** / Multi Slave operation
  - Bus master generates clock
  - Master initiates transfer in both directions
  - Asserts CS before transmission (usually active-low)
- The master sends bits on the MOSI line and slave sends bits on MISO line
- 8/16-bit data transfers
- Used in small distances and higher data rates

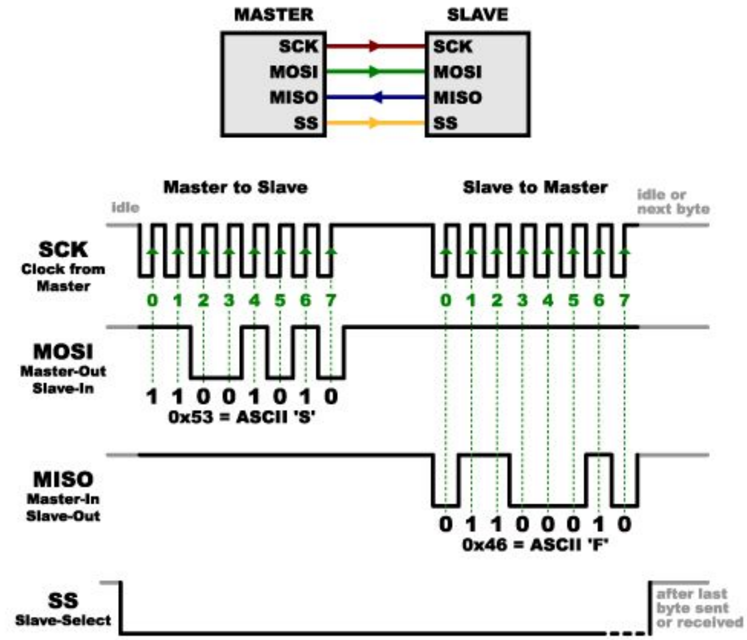


Example SPI Data Transmission

- Due to single bus master protocol, all the bus communications with the slave devices are initiated by the master device
- When the master intends to send/receive data to/from a slave device, it pulls the corresponding CS line low
- The master transmits data using MOSI line, and the incoming data from the selected slave is received by sampling MISO line



SPI Data transmission

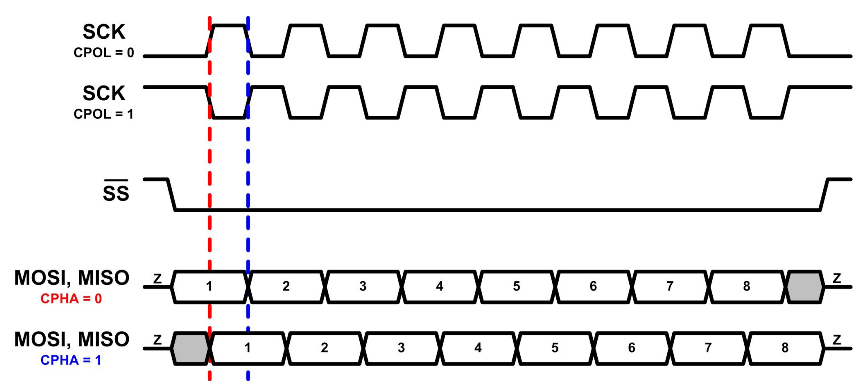


SPI Modes

- There are two different options called clock polarity (CPOL) and clock phase with respect to the data (CPHA) that creates four different ways of reading data
- For CPOL = 0 the clock idles at logic zero
  - If CPHA = 0, data are read on the rising edge and change on the falling edge of SCK
  - If CPHA = 1, data are read on the falling edge and change on the rising edge of SCK
- For CPOL = 1 the clock idles at logic high
  - If CPHA = 0, data are read on the falling edge and change on the rising edge of SCK
  - If CPHA = 1, data are read on the rising edge and change on the falling edge of SCK

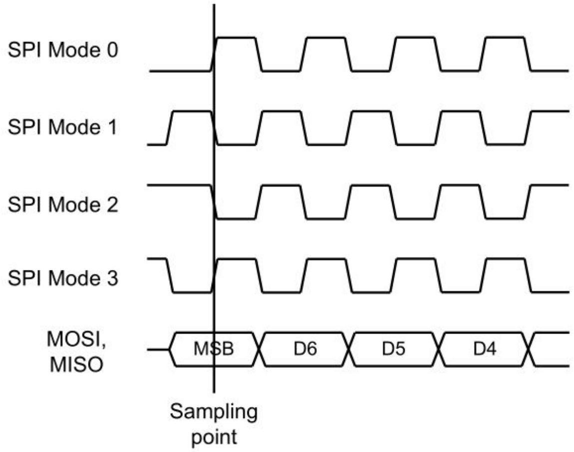
SPI Modes

- There are two different options called clock polarity (CPOL) and clock phase with respect to the data (CPHA) that creates four different ways of reading data



SPI Modes representation

- There are two different options called clock polarity (CPOL) and clock phase with respect to the data (CPHA) that creates four different ways of reading data



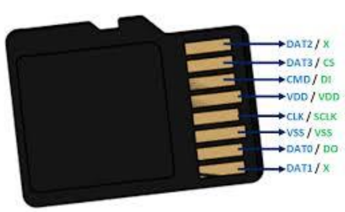
Example SPI read / write

```c
uint8_t spi_read(uint8_t reg)
{
    enable_chip();
    // send the register to be read
    // usually 16-bits and one of the bits
    // represents read/write operation
    // let's assume bit 15 is r'/w bit
    SPI1->TDR = (1 << 15) | (reg << 8);
    // wait until tx buffer is empty
    // wait until rx buffer is not empty
    // read contents of data register
    uint8_t data = (uint8_t)SPI1->RDR;
    disable_chip();
    return data;
}
```

```c
void spi_write(uint8_t reg, uint8_t data)
{
    enable_chip();
    // send the register to be read
    // usually 16-bits and one of the bits
    // represents read/write operation
    // let's assume bit 15 is r'/w bit
    SPI1->TDR = (reg << 8) | (data);
    // wait until tx buffer is empty
    // wait until rx buffer is not empty
    // dummy read
    (void)SPI1->RDR;
    disable_chip();
}
```

- Since each chip has a separate CS line, moving `enable_chip()` and `disable_chip()` out of these functions is better for multi chip configurations

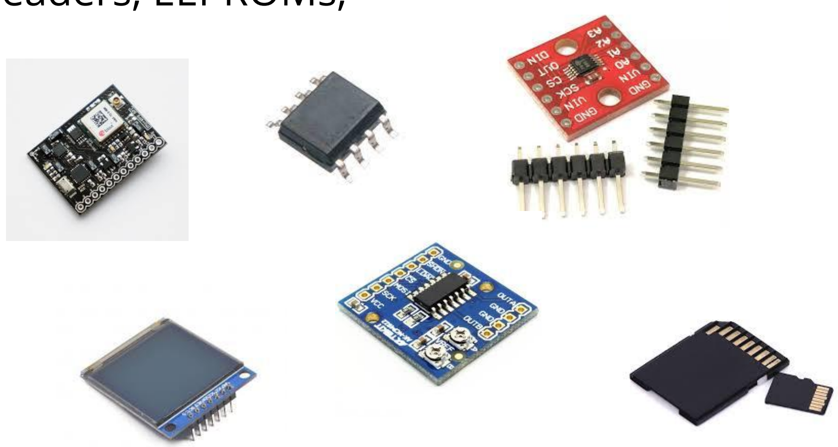
Example SD Card



- SD cards have two communication modes
  - Native 4-bit
  - SPI 1-bit
- Host sends a six-byte command packet to card
  - Index, argument, CRC
- Host reads bytes from card until card signals it is ready
  - Card returns
    - 0xFF while busy
    - 0x00 when ready without errors
    - 0x01 - 0x7F when error has occurred

SPI Usage

- Can be used with various modules such as ADC/DAC converter, IMU, SD Card, Displays, RFID readers, EEPROMs,

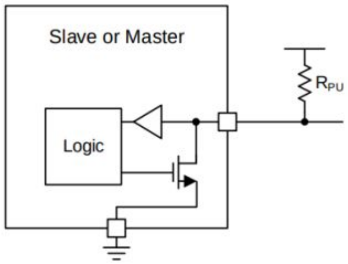


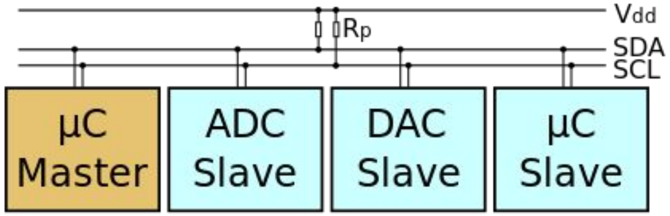
Inter-Integrated Circuit (I2C / IIC / I^{2}C)

- Inter-Integrated Circuit bus is a synchronous, **multi-master**, multi-slave serial communication bus
- Developed by Philips Semiconductor in 1980s
- Used in various sensors, EEPROMs and LCDs
- Can go from 100 kbits/s to 5 Mbits/s
  - 100 kbit/s Standard Mode (SM)
  - 400 kbit/s Fast Mode (FM)
  - 3.4 Mbit/s High-Speed Mode (HS)
- More complex than UART or SPI

Inter-Integrated Circuit (I2C / IIC / I^{2}C)

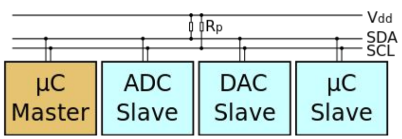
- Two bidirectional **open collector** / **open drain** lines.
  - **SDA** - Serial Data Line
  - **SCL** - Serial Clock Line
- Each line is **pulled up with a resistor**
- Since there is no extra chip select line, uses **addressing** to talk to different ICs

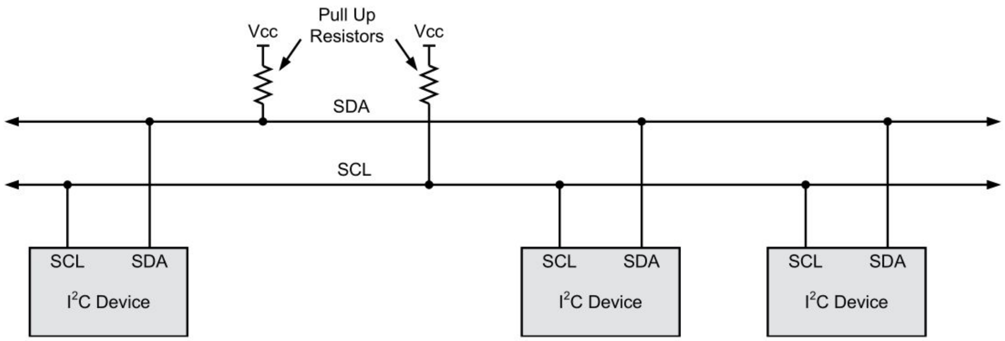




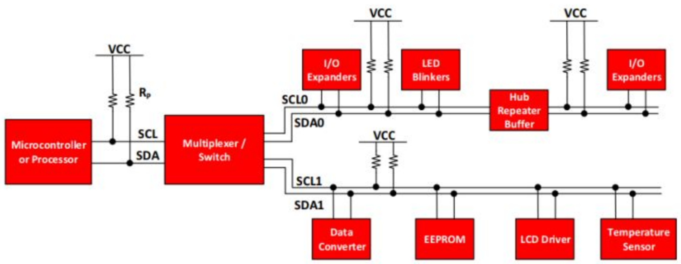
Example I2C Connecting

- Each line is **pulled up with a resistor**





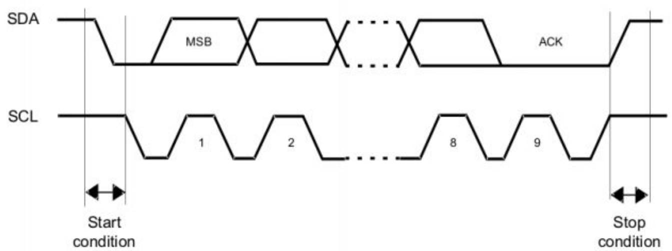
Example I2C connection



- if a master wants to **send data to slave**
  - master sends start condition and addresses to the slaves
  - master sends data to slave
  - master completes the transfer with stop condition
- if a master wants to **read data from slave**
  - master sends start condition and addresses to the slaves
  - master sends register to read to slave
  - slave sends the data to master
  - master completes the transfer with stop condition

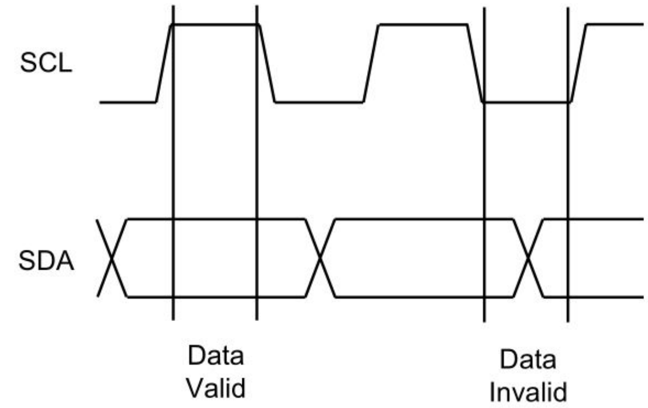
I2C signaling and start /stop condition

- **Start condition** is a high to low transition on SDA line when SCL is high
- **Stop condition** is a low to high transition on SDA line when SCL is high
- Data is placed on the SDA line after SCL goes low, and sampled after SCL line goes high



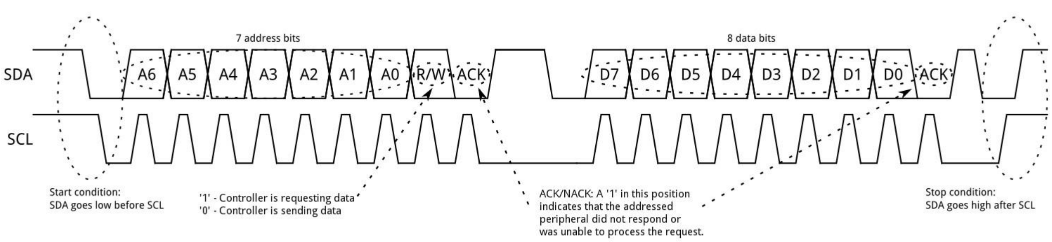
I2C data validity

- SDA needs to be **stable** when SCL is high

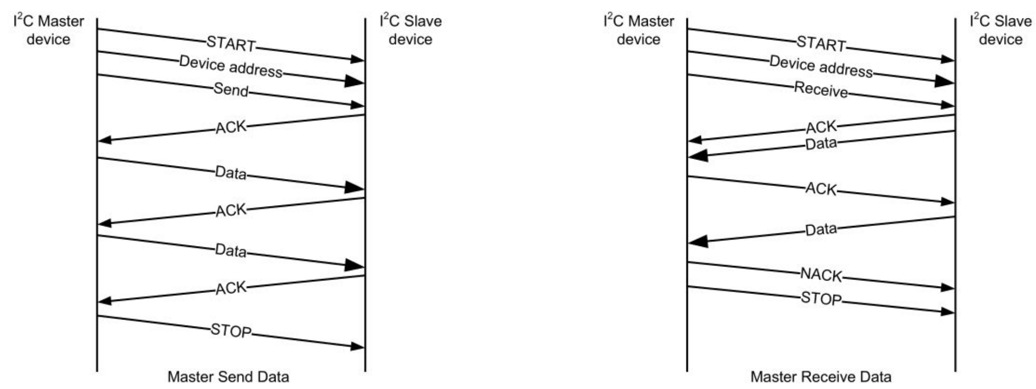


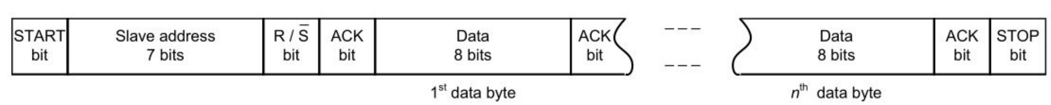
I2C protocol basics

- Messages are broken up into two types of frame
  - **Address frame** - where the controller indicates the recipient peripheral
    - 7-bit / 10-bit address modes
  - One or more **data frames** - which are 8-bit data messages passed from controller to peripheral and vice versa.



I2C Communication





I2C protocol basics

- Each device has 7-bit (/10-bit) address
- How do we connect two copies of the same peripheral to the bus?
- Vendors provide a pin to choose the LSB(s) of the address. Usually denoted as A0, A1, A2, ...
  - Example: Let's say we have an IMU and its address is **0b110100x**
    - By connecting A0 pin to ground, we can set its address as 0b1101000 - 0x68
    - By connecting A0 pin to Vdd, we can set its address as 0b1101001 - 0x69

Example I2C read / write

```
uint8_t i2c_read(uint8_t deviceAddr, uint8_t
regAddr) {
    i2c_start(); // send start condition
    // send device address in write mode
    I2C1->TDR = deviceAddr;
    // wait until address is sent
    // send register to be read
    I2C1->TDR = regAddr;
    i2c_start(); // restart transmission
    // send device address in read mode
    I2C1->TDR = deviceAddr | 0x01; // read
    // wait until address is sent
    // wait until receive buffer is not empty
    // read content
    uint8_t reg = (uint8_t)I2C1->RDR;
    // send stop condition
    i2c_stop();
    return reg; }
```

```
void i2c_write(uint8_t deviceAddr, uint8_t
regAddr, uint8_t data) {
    i2c_start(); // send start condition
     // send device address in write mode
    I2C1->TDR = deviceAddr;
    // wait until address is sent
    // send register to be read
    I2C1->TDR = regAddr;
    // wait until byte transfer complete
    I2C1->TDR = data;   // send data
    // wait until byte transfer complete
    // send stop condition
    i2c_stop();
}
```

- Reading / writing I2C is a little more involved, and depends heavily on the MCU implementations

I2C Usage

- Can be used with various modules such as ADC/DAC converter, IMU, GPIO Expander, Displays, EEPROMs,

Factors to consider when selecting protocols

- How fast can the data get through?
  - depends on bit rate and protocol overhead
- How many hardware signals do we need
  - may need clock line, chip select lines
- How do we connect multiple devices (topology)?
  - dedicated link and hardware per device (point to point)
- How do we address a target device?
  - discrete hardware signal
  - address embedded in packet
- How do these factors change as we add more devices?

This week

- Project 2 is due on 23rd December
- No Lab & HW this week