EE 337: Synchronous Serial Data Transfer- I^2C Bus
Lab 12

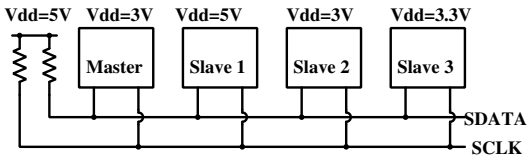
4^{th} November, 2019

In the last experiment, we had used SPI as a synchronous data transfer protocol. This time we shall use another synchronous data transfer protocol known as I^2C.

Introduction

The I^2C bus is a 2 wire serial interface originally developed by Philips. The name stands for Inter Integrated circuit bus. Writing 'I^2C' is difficult in ordinary text, so it is often called the 'I2C' bus. It is used widely for connecting peripherals to processors and by smart home appliances to communicate with each other. 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. Restricted versions of this protocol are also known as the '2 wire interface' or TWI. We shall be using built in TWI modules in the micro-controller for this experiment. The AT89C5131A data sheet describes the TWI interface starting from **page 102**.

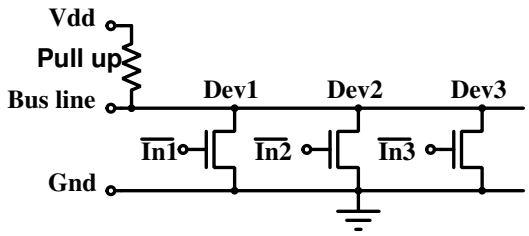
Devices communicating with this protocol are connected in parallel across 2 wires. One of these wires carries serial data while the other carries the clock.



Each device has a unique address, which is normally 7 bit wide. All devices drive the bus lines using open drain drivers. Each line has a pull up resistor.

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 device provides the clock. It can act as the transmitter or the receiver. The addressed slave device then assumes a complementary role (receiver/transmitter). The protocol provides for multiple master deices on the bus and arbitration procedures for one of these to become the active master. Not all implementations provide this feature and those that do refer to it as multi-master capability. Data is normally sent at 100 Kilo bits per second, though some systems use a 'fast' version which runs at 400 Kbps. The protocol does not restrict one from using much lower clock frequencies.

Since both lines use open drain drivers, 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 on its own.

Also, different devices using different supply voltages can be easily connected to each other, since the drivers provide only a pull down function. This is a desirable feature when connecting devices from different manufacturers.

The actual data transfer is controlled by the Master. The Master 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. 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.

Suppose the micro-controller is the master and it wants to send data serially to a slave device. It generates the 'Start' message on the bus, and 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/W bit as the least significant bit. (This bit is 1 for read operations and 0 for a write operation). After the transmitter has sent these 8 bits on the data 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 message. (Recall that this is possible because of the open drain connection discussed earlier). This is called the 'Ack' message. If all receiver driver transistors are 'OFF' during the ninth bit period, the data line will be seen to be 'High' during this time. This is known as a 'Nack' message. 'Nack' is used by the slave to signal failure to receive when it is a listener (Write operation by master). After the address has been acknowledged by the receiver, master (which is the transmitter), 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 of the line.

When the master is the listener, it sends 'Ack's after receiving each data byte (except the last one), inviting the slave talker to send the next byte. It sends 'Nack' after the last byte is received and follows it up with a 'Stop' condition to terminate data transfer.

Please refer to the accompanying document on I^2C protocol (written by Yogesh Patil) for details about using the protocol for this problem. We shall use the I2C protocol to communicate with a real time clock chip (DS 1307). Data sheet for DS 1307 is also being uploaded.

Lab work

Connect up DS 1307 to the card using appropriate port lines. (Remember to provide power and ground lines to DS 1307 in addition to Serial Data and clock!)

Assume that the program memory contains an array of **7 bytes** holding information of **your system's date and time**, in the format described in Table 2 on page 8 of the data sheet for DS1̃307. An excerpt based on that table is given below. In this problem, we display data read from the real time clock chip. Offset is measured from the start of the array.

All Data is stored as **BCD** digits.

Offset	function	Range	upper nibble	lower nibble
00	Seconds	0-59	Tens	Units
01	Minutes	0-59	Tens	Units
02	Hours	0-59	Tens	Units
03	Day	01-07	0	Day
04	Date	01-31	Tens	Units
05	Month	01-12	Tens	Units
06	Year	00-99	Tens	Units

You have been provided a

1. You have been provided with a template code, modify the code to write **once** those seven bytes to RTC, then **endlessly** read from it.

2. Observe the waveforms of SDA (Data line) and SCL (Clock line) on DSO and show it to your TA.

3. In the next step you are required to display the data read in the following format on the LCD display:

   Line 1 should contain time information: "Time HH:MM:SS AM" or "Time HH:MM:SS PM", where HH is the hour in 12 hr. format.

   Line 2 should contain the day information as: "Date WW DD/MM/YY", where WW is a 2 letter abbreviation for the day of the week, DD is the date, MM represents the month and YY are the last 2 digits of the year.

Bonus Part

1. You might observe that whenever the reset button is pressed, the RTC resets to the original Date and Time provided by you in the code. Your task is to make the clock time **reset-immune**, i.e. once the RTC has been programmed with initial Date and Time, there should be no furthur change reflected in the clock's Date and Time on reset press.

2. Define two keys on the board as the function and increment keys.
   Each time the function key is toggled, the program goes into setting mode in the following order:
   Day → Date (tens) → Date (units) → Month (tens) → Month (units) →Year (tens) → Year (units) → Hours (tens) → Hours (Units) → Minutes (tens) → Minutes (units) → Seconds (tens) → Seconds (units) → Plain Display → Day ....
   The corresponding digit (or the three letter abbreviation) on the display should blink (except for Plain Display where there is no blinking).

   Each time the increment key is toggled, it should increment the item being set, reverting back to its minimum value after its maximum value is reached. For example, minutes (tens) digit should increase to 5 and then go back to 0.