Fachhochschule
Münster University of
Applied Sciences

Lab for Semiconductor Devices and Bussystems

Exercise Bussystems SS2012/13
Tutor: Dipl. Informatiker Gregor Rebel
MSc. Francisco Estevez

Exercise 2
I²C Management

I²C uses only two bidirectional open-drain lines, Serial Data Line (SDA) and Serial Clock Line (SCL), pulled up with resistors. Typical voltages used are 5 V or 3.3 V. The I²C reference design has a 7-bit or a 10-bit (depending on the device used) address space. Common I²C bus speeds are the 100 kbit/s standard mode and the 10 kbit/s low-speed mode. Recent revisions of I²C can host more nodes and run at faster speeds like 400 kbit/s Fast mode, 1 Mbit/s Fast mode plus or Fm+, and 3.4 Mbit/s High Speed mode. These speeds are more widely used on embedded systems than on PCs.
Note the bit rates are quoted for the transactions between master and slave without clock stretching or other hardware overhead. Protocol overheads include a slave address and perhaps a register address within the slave device as well as per-byte ACK/NACK bits.
The maximum number of nodes is limited by the address space, and also by the total bus capacitance of 400 pF, which restricts practical communication distances to a few meters. When there are many I²C devices in a system or it is necessary to send over long cables, then can be a need to include bus buffers to split large bus segments into smaller ones. This can be necessary to keep the capacitance of a bus segment below the allowable value (400 pF).

I²C is appropriate for peripherals where simplicity and low manufacturing cost are more important than speed. A particular strength of I²C is the capability of a microcontroller to control a network of devices with just two general purpose I/O pins and software. Many other bus technologies used in similar applications, such as Serial Peripheral Interface Bus (SPI), require more pins and signals to connect devices.

Reference design
There fore mentioned reference design is a bus with a clock (SCL) and data (SDA) lines with 7-bit addressing. The bus has two roles for nodes: master and slave:

- Master: Node that generates the clock and initiates communication with slaves
- Slave: Node that receives the clock and responds when addressed by the master

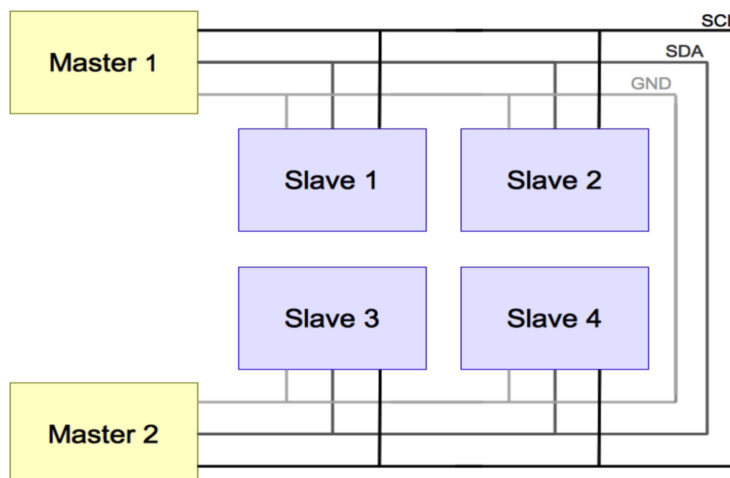


Image 1: I2C Devices schema

The bus is a multi-master bus which means any number of master nodes can be present. Additionally, master and slave roles may be changed between messages (after a *STOP* is sent).
There are four potential modes of operation for a given bus device, although most devices only use a single role and its two modes:

- Master transmit: master node is sending data to a slave
- Master receive: master node is receiving data from a slave
- Slave transmit: slave node is sending data to the master
- Slave receive: slave node is receiving data from the master

The master is initially in master transmit mode by sending a start bit followed by the 7-bit address of the slave it wishes to communicate with, which is finally followed by a single bit representing whether it wishes to write(0) or to read(1) from the slave. At the end of every transaction a stop bit is sent.
If the slave exists on the bus then it will respond with an ACK bit (active low for acknowledged) for that address. The master then continues in either transmit or receive mode, and the slave continues in its complementary mode.
The address and the data bytes are sent most significant bit first. If the master wishes to write to the slave then it repeatedly sends a byte with the slave sending an ACK bit. If the master wishes to read from the slave then it repeatedly receives a byte from the slave, the master sending an ACK bit after every byte but the last one. The master then either ends transmission with a stop bit, or it may send another START bit if it wishes to retain control of the bus for another transfer.

Message protocols
I²C defines basic types of messages, each of which begins with a START and ends with a STOP:
- Single message where a master writes data to a slave;
- Single message where a master reads data from a slave;
- Combined messages, where a master issues at least two reads and/or writes to one or more slaves.

In a combined message, each read or write begins with a START and the slave address. After the first START, these are also called repeated START bits; repeated START bits are not preceded by STOP bits, which is how slaves know the next transfer is part of the same message.

Pure I²C systems support arbitrary message structures. SMBus is restricted to nine of those structures, such as read word N and write word N, involving a single slave. PMBus extends SMBus with a Group protocol, allowing multiple such SMBus transactions to be sent in one combined message. The terminating STOP indicates when those grouped actions should take effect. For example, one PMBus operation might switch on several lamps (using a different I²C slave address per lamp), and their new state would take effect at the same time: when they receive that STOP.
With only a few exceptions, neither I²C nor SMBus define message semantics, such as the meaning of data bytes in messages.
In practice, most slaves adopt request/response control models, where one or more bytes following a write command are treated as a command or address. Those bytes determine how subsequent written bytes are treated and/or how the slave responds on subsequent reads. Most SMBus operations involve single byte commands.

Physical layer
At the physical layer, both SCL and SDA lines are of open-drain design, thus, pull-up resistors are needed. Pulling the line to ground is considered a logical zero while letting the line float is a logical one. This is used as a channel access method. High speed systems also add a current source pull up, at least on SCL; This supports faster rise times and higher bus capacitance.
An important consequence of this is that multiple nodes may be driving the lines simultaneously. If any node is driving the line low, it will be low. Nodes that are trying to transmit a logical one can see this, and thereby know that another node is active at the same time.
When used on SCL, this is called *clock stretching* and gives slaves a flow control mechanism. When used on SDA, this is called *arbitration* and ensures there is only one transmitter at a time.
When idle, both lines are high. To start a transaction, SDA is pulled low while SCL remains high. Releasing SDA to float high again would be a stop marker, signaling the end of a bus transaction. Although legal, this is typically pointless immediately after a start, so the next step is to pull SCL low. Except for the start and stop signals, the SDA line only changes while the clock is low. Transmitting a data bit consists of pulsing the clock line high while holding the data line

steady at the desired level. While SCL is low, the transmitter sets SDA to the desired value with a small delay to let the value propagate, then lets SCL float high. Once SCL is high, the master waits a minimum time, 4 µs for standard speed, to ensure the receiver has seen the bit, then pulls it low again. This completes transmission of one bit.
After every 8 data bits in one direction, an ack bit is transmitted in the other direction. The transmitter and receiver switch roles for one bit and the erstwhile receiver transmits a single 0 bit (ACK) back. If the transmitter sees a 1 bit (NACK) instead, it learns that:

- Master to slave: The slave is unable to accept the data. No such slave, command not understood, or unable to accept any more data.
- Slave to master: The master wishes the transfer to stop after this data byte.

After the acknowledge bit, the master may do one of three things:

- Prepare to transfer another byte of data
- Send a "Stop"
- Send a "Repeated start"

***Clock stretching* using SCL**

One of the more significant features of the I²C protocol is *clock stretching*. An addressed slave device may hold the clock line (SCL) low after receiving or sending a byte, indicating that it is not yet ready to process more data. The master that is communicating with the slave may not finish the transmission of the current bit, but must wait until the clock line actually goes high. If the slave is clock stretching, the clock line will still be low. The same is true if a second, slower, master tries to drive the clock at the same time.
The master must wait until it observes the clock line going high, and an additional minimum time, typically 4 µs, before pulling the clock low again.
Although the master may also hold the SCL line low for as long as it desires, the term *clock stretching* is normally used only when slaves do it. *Clock stretching* is the only time in I²C where the slave drives SCL.
To ensure a minimum bus throughput, SMBus places limits on how far clocks may be stretched. Master and slaves adhering to those limits cannot block access to the bus for more than a short time, which is not a guarantee made by pure I²C systems.

***Arbitration* using SDA**

Every master monitors the bus for start and stop bits, and does not start a message while another master is keeping the bus busy. However, two masters may start transmission at about the same time, in this case, *arbitration* occurs. I²C has a deterministic arbitration policy. Each transmitter checks the level of the data line and compares it with the levels it expects. If they do not match, that transmitter has lost arbitration, and drops out of this protocol interaction.
If one transmitter sets SDA to 1 and a second transmitter sets it to 0, the result is that the line is low. The first transmitter then observes that the level of the line is different than expected, and concludes that another node is transmitting. The first node to notice such a difference is the one that loses *arbitration*: it stops driving SDA. If it is a master, it also stops driving SCL and waits for a STOP signal. Then it may try to reissue its entire message. In the meantime, the other node has not noticed any difference between the expected and actual levels on SDA, and therefore continues transmission.
If the two masters are sending a message to two different slaves, the one sending the lower slave address always wins *arbitration* in the address stage.
Since the two masters may send messages to the same slave address, and addresses sometimes refer to multiple slaves, *arbitration* must continue into the data stages.
Arbitration occurs very rarely, but is necessary for proper multi-master support.

I²C Exercises:

To begin to use I²C interface, we will start with our examples in the Toolchain. Later, you should complete exercises depicted below. Files where I²C interface is depicted are:

- ttc_i2c.h
- ttc_i2c.c
- ttc_i2c_types.h

To activate I^2C library and set all libraries that you need, you should do it via *activate_project.sh*, like we saw in the last session. You can start using I^2C example in the Toolchain, but later you should write your own source code to complete the exercises.
Toolchain's example will be *activate.600_example_i2c_master.s*. To execute this example you must use Olimex LCD board and you need to include also right libraries.
This example should show accelerometer values on the LCD, like you can see on the next picture.

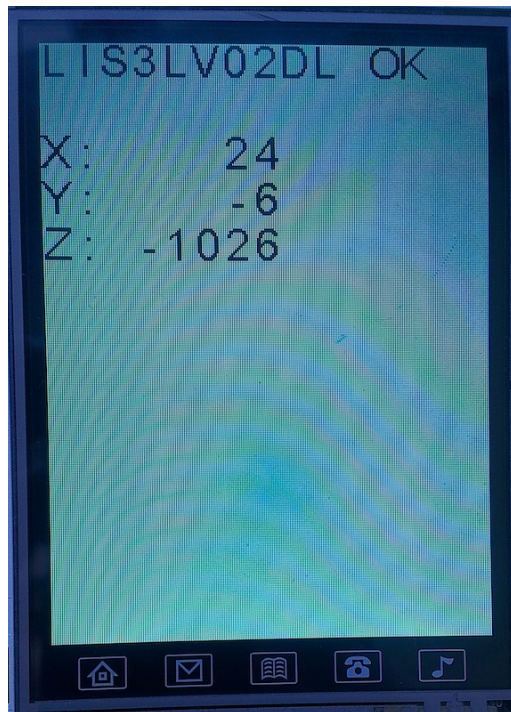


Image 2: Toolchain I2C_Master example execution

Now you should debug this example. To start the debugger, you will execute on your terminal *_/debug.sh*. This command should start a gdb interface which connecting directly with our microcontroller.
Once you are executing the debugger you will have an interface to type commands and interact with our program. The main commands that you will need to use on the debugger are:

- b *filename.extension*:*line_number*, for example: b example_i2c.h:14 ; This command set a breakpoint in our code and will stop the program execution at this point.
- c; This command continues the execution of one program when it was halted for a breakpoint or a manual stop.
- n; This command executes the next command on the program flow.
- p variable_name, for example p i ; This command will show the value of one variable in one determined time.
- p variable_name = value, for example p i = 2; This command also set a value in one variable.
- clear; This command will remove all the breakpoints that you set before.
- bt; This command will show a backtrace of our program. It is very interesting when an error appears.
- quit; This command will finish the debugger interface.

Now, you are ready to start with I²C exercises. First of all, you need to know how to debug your microcontroller, and in this practice, an Olimex LCD. To do this, you will need to use the debugger like we explained before. For the second exercise, you should review the schematic from Olimex (you can find this schematic at https://www.olimex.com/Products/ARM/ST/STM32-LCD/resources/STM32-LCD-schematic.pdf).

1. In this exercise you should debug the I2C example that you execute previously. You will need to explain how works I2C interface, understanding and describing the program flow and writing the value of some key parameters.

   - Key Parameters on *example_i2c_start()*:

   I2C_Generic->Clock: ________________
   I2C_Generic->Timeout: ________________
   I2C_Generic->Slave_NoStretch: ________________
   MaxI2C: ________________
   I2C_Generic->Acknowledgement: ___________
   I2C_Generic->All: ________________
   I2C_Arch->Base: ________________

   - Key Parameters on *task_I2C_master()*:

   I2C_ADDRESS_LIS3LV02DL: ________________
   StatusRegister: ________________
   LIS_REG_WHO_AM_I: ________________

   - *stm32_i2c_read_register()* flow description:

2. For the second exercise you will need to recognize I²C signals on your oscilloscope. You will change the example, increasing the time between accelerometer readings up to 5 seconds. Now, you should write and explain over your drawing, the different parts that one I²C transfer has. In this case you should recognize, start and stop signals, address, data, and acknowledgments, explaining in each moment what it happens, but only with X axis readings. *DATA* values are not relevant because can change during the exercise.

SDA and SCL examples for a read operation of the device with address 0x25h:

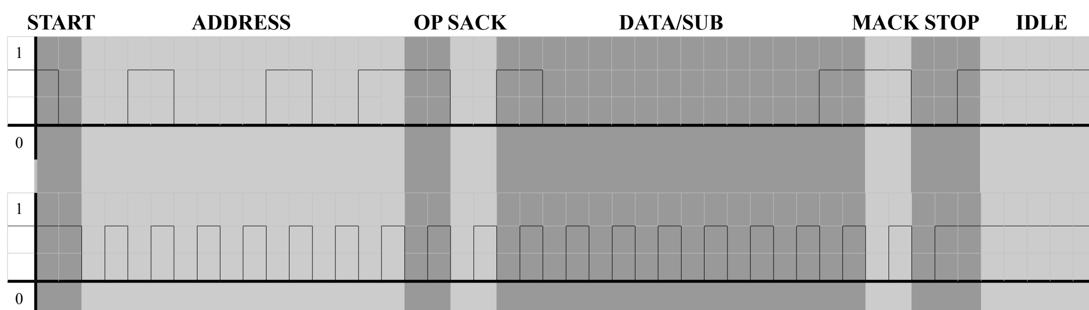


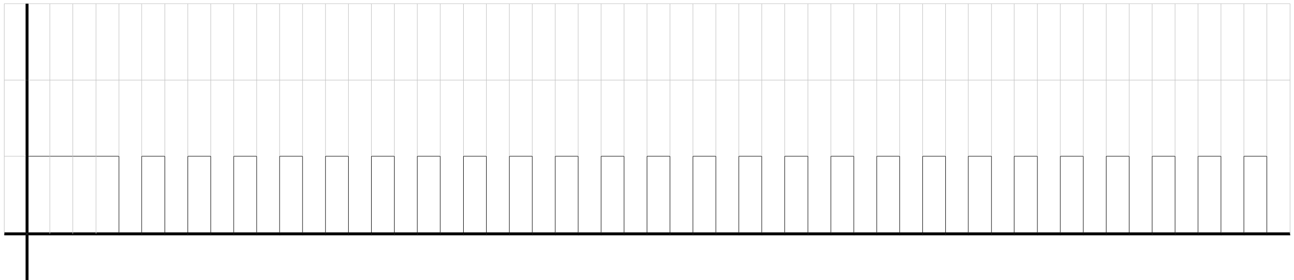
Chart 3: I²C *signals on oscilloscope. Up, SDA signal, bottom SCL signal.*

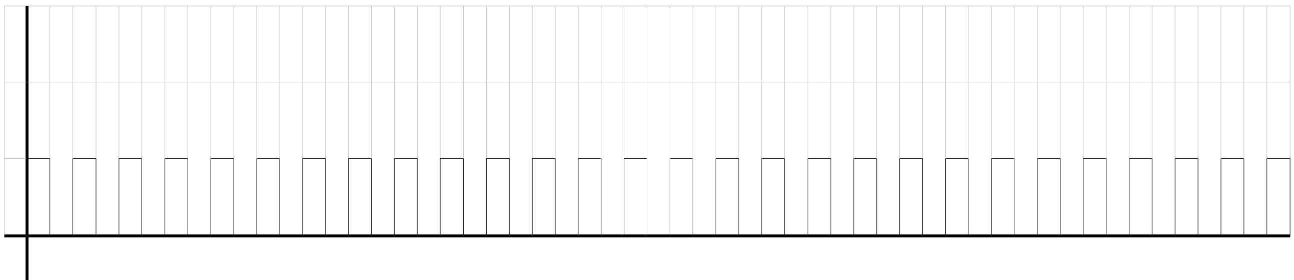
Time division value: 50 us

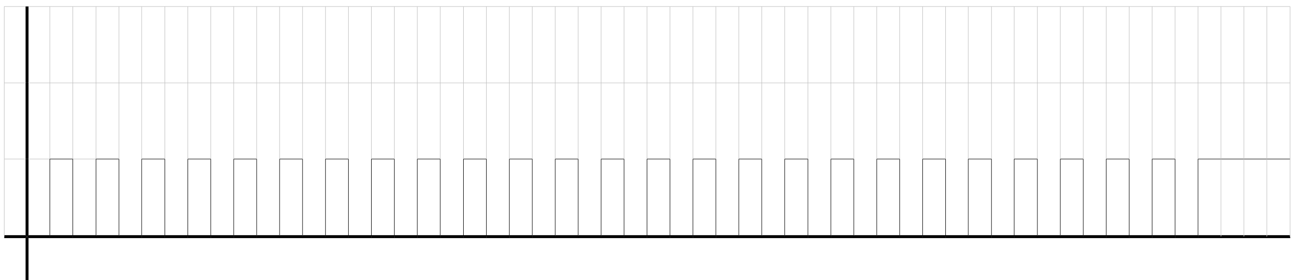
Accelerometer interesting details:

- OP makes reference to read or write operation. 1 represents read operation (DATA is read) and 0 write operation (SUB is written).
- ADDRESS uses 7 bit to be represented.
- SACK makes reference to *Slave ACK* and uses only 1 bit. MACK makes reference to *Master ACK* and uses also 1 bit.
- DATA has always 8 bits.
- When SDA and SCL are in idle state, both are at high level.
- STOP signal is used only when a complete transmission is finished. In other cases you can have another START signal representing another transmission.

SDA:







SCL is represented at 200mV
Time division value: 50 us

Information from www.en.wikipedia.org

PC: ______________________ Group: A | B | C

Name: ______________________________ Name: ______________________________

Sign: ______________________________ Sign: ______________________________