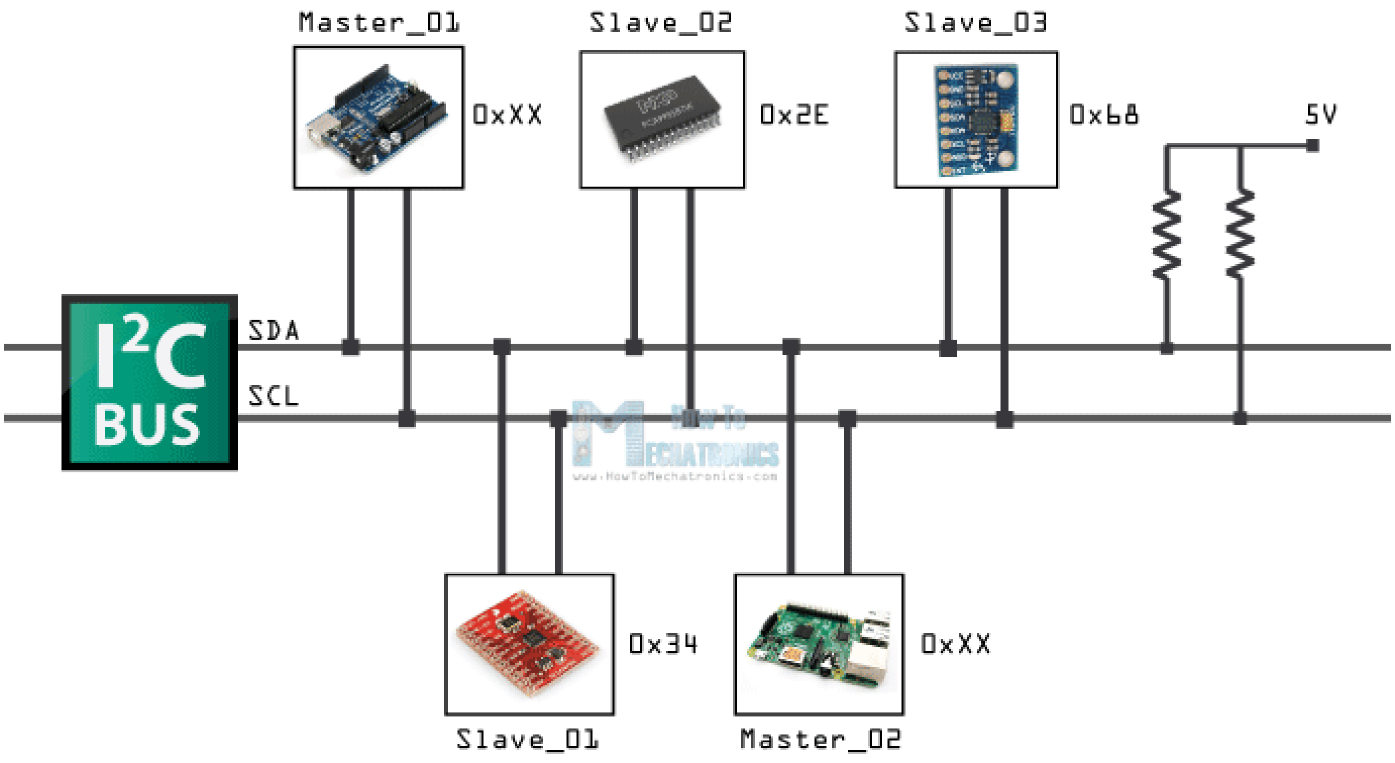
Lesson I2C

I2C

What is I^2C

I2C (Inter-Integrated Circuit) is pronounced "eye-squared see". It is also known as "TWI" because of the initial patent issues regarding this BUS. This is a popular, low throughput (100-1000Khz), half-duplex BUS that only uses two wires regardless of how many devices are on this BUS. Many sensors use this BUS because of its ease of adding to a system.



I2C devices connected up to an I^2C bus

Pins of I^2C

There are two pins for I2C:

- **SCL**: Serial clock pin
- **SDA**: Serial data pin

The **clock** line is usually controlled by the Master with the exception that the slave may pull it low to indicate to the master that it is not ready to send data.

The **data** line is bi-directional and is controlled by the Master while sending data, and by the slave when it sends data back after a repeat-start condition described below.

Open-Collector/Open-Drain BUS

I2C is an open-collector BUS, which means that no device shall have the capability of internally connecting either SDA or SCL wires to power source. The communication wires are instead connected to the power source through a "pull-up" resistor. When a device wants to communicate, it simply lets go of the wire for it to go back to logical "high" or "1" or it can connect it to ground to indicate logical "0". This achieves safe operation of the bus (no case of short circuit), even if a device incorrectly assumes control of the bus.

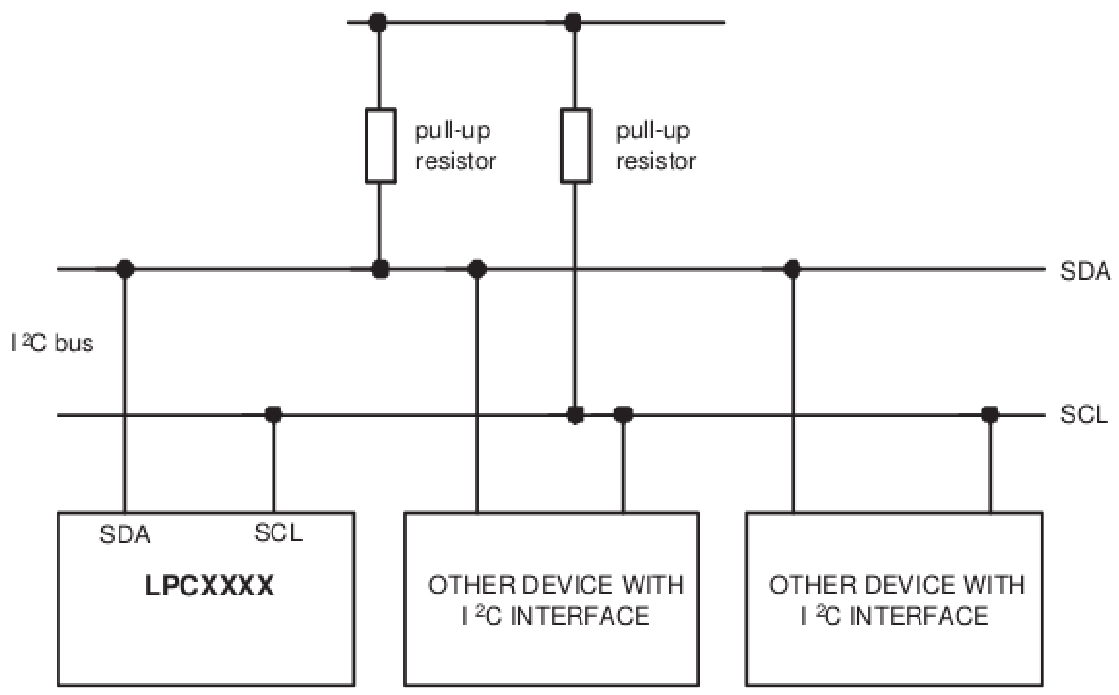
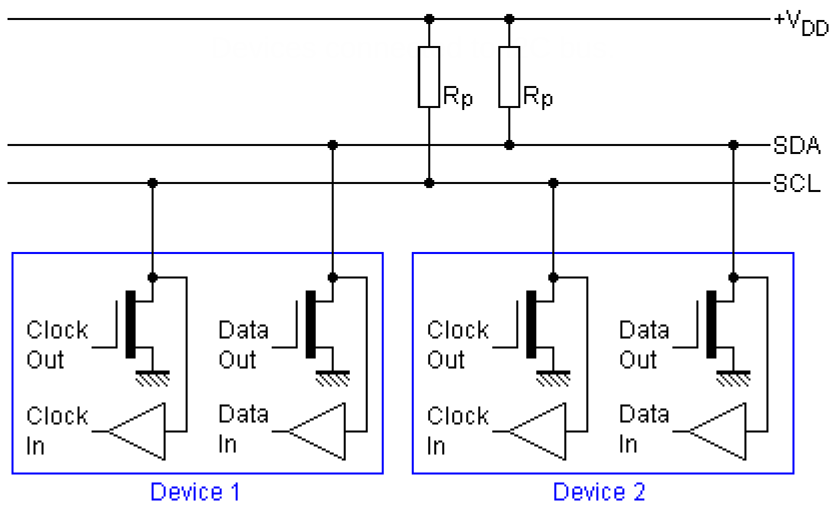

pull-up resistor
pull-up resistor
SDA
I²C bus
SCL
SDA
SCL
LPCXXXX
OTHER DEVICE WITH I²C INTERFACE
OTHER DEVICE WITH I²C INTERFACE

Fig 84. I^{2}C-bus configuration



I2C device pin output stage.

Pull-up resistor

Using a smaller pull-up can achieve higher speeds, but then each device must have the capability of sinking that much more current. For example, with a 5v BUS, and 1K pull-up, each device must be able to sink 5mA.

Try this link, and if the link doesn't work, import this into the circuit simulator:

```
$ 3 0.000005 10.20027730826997 50 5 50
172 352 200 352 152 0 6 5 5 0 0 0.5 Voltage
r 352 240 352 304 0 1000
g 352 368 352 384 0
c 352 304 352 368 0 0.00001 0
S 384 304 432 304 0 1 false 0 2
w 352 240 352 200 0
w 352 304 384 304 0
w 432 320 432 368 0
w 432 368 352 368 0
o 6 64 0 4098 5 0.025 0 2 6 3
```

Why Use I^{2}C

Pros

- **IO/Pin Count:**
  - 2 pins bus regardless of the number of devices.
- **Synchronous:**
  - No need for agreed timing before hand
- **Multi-Master**
  - Possible to have multiple masters on a I2C bus
- **Multi-slave:**
  - 7-bit address allows up to an absolute maximum of 119 devices (because 8 addresses are reserved)

Cons

- **Slow Speed:**
  - Typical I2C devices have a maximum speed of 400kHz
  - Some devices can sense speeds up to 1000kHz or more
- **Half-Duplex:**
  - Only one device can talk at a time
- **Complex State Machine:**
  - Requires a rather large and complex state machine in order to handle communication
- **Master Only Control:**
  - Only a master can drive the bus
  - An exception to that rule is that a slave can stop the clock if it needs to hold the master in a wait state
- **Hardware Signal Protocol Overhead**
  - This protocol includes quite a few bits, not associated with data to handle routing and handshaking. This slows the bus throughput even further

Protocol Information

I2C was designed to be able to read and write memory on a slave device. The protocol may be complicated, but a typical "transaction" involving read or write of a register on a slave device is simple granted a "sunny-day scenario" in which no errors occur.

I2C at its foundation is about sending and receiving bytes, but there is a layer of unofficial protocol about how the bytes are interpreted. For instance, for an I2C write transaction, the master sends three bytes and 99% of the cases, they are interpreted like the following:

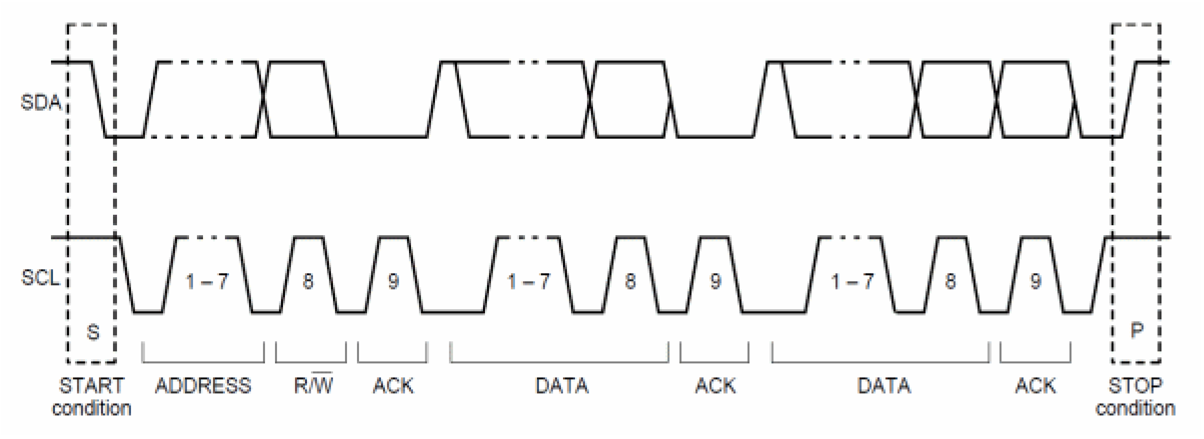
1. I2C Slave Device Address (required first byte in I2C protocol) `slave_address`
2. Device Register `[r]`
    - For I2C standard, this is just a plain old byte, but 99% of the devices treat this as their "register address"
    - Your I2C state machine has same state for this one, and the next data byte
    - Your software will treat this first byte as a special case
3. Data `[data]`
    - data to write at the "Device Register" above

`slave_address[r] = data`

> The code samples below illustrates I2C transaction split into functions, but this is the wrong way of writing an I2C driver. An I2C driver should be "transaction-based" and the entire transfer should be carried out using a state machine. The idea is to design your software to walk the I2C hardware through its state to complete an I2C transfer.

Signal Timing Diagram

?



I2C communication timing diagram

For both START and STOP conditions, SCL has to be high. A high to low transition of SDA is considered as **START** and a low to high transition as **STOP**.

Write Transaction

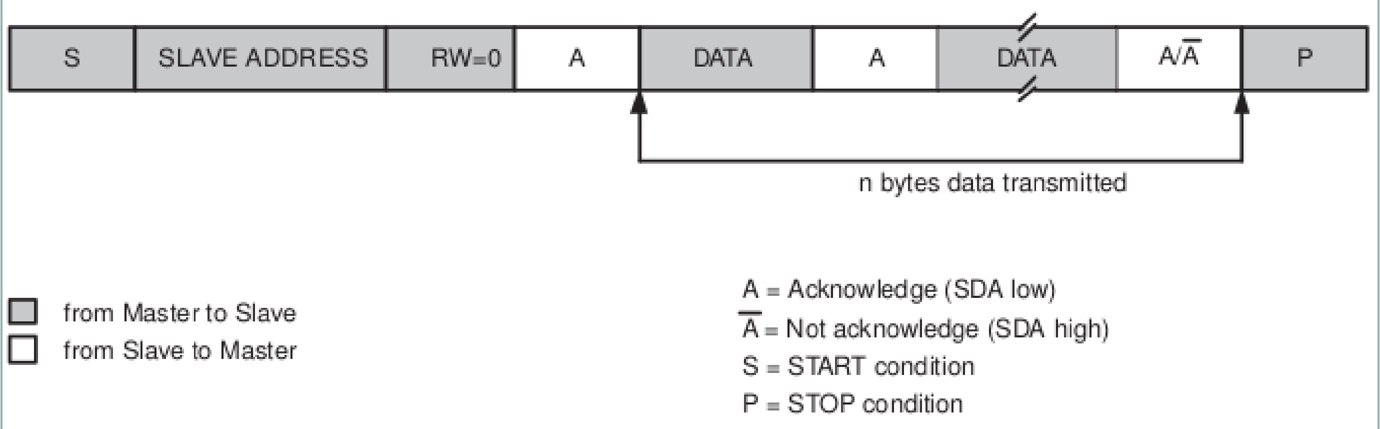


Fig 85. Format in the Master Transmitter mode

Master Transmit format

The master always initiates the transfer, and the device reading the data should always "ACK" the byte. For example, when the master sends the 8-bit address after the START condition, then the addressed slave should ACK the 9th bit (pull the line LOW). Likewise, when the master sends the first byte after the address, the slave should ACK that byte if it wishes to continue the transfer.

A typical I2C write is to be able to write a register or memory address on a slave device. Here are the steps:

1. Master sends START condition followed by device address.
   Device that is addressed should then "ACK" using the 9th bit.
2. Master sends device's "memory address" (1 or more bytes).
   Each byte should be ACK'd by the addressed slave.
3. Master sends the data to write (1 or more bytes).
   Each byte should be ACK'd by the addressed slave.
4. Master sends the STOP condition.

To maximize throughput and avoid having to send three I2C bytes for each slave memory write, the memory address is considered "starting address". If we continue to write data, we will end up writing data to M, M+1, M+2 etc.

The ideal way of writing an I2C driver is one that is able to carry out an entire transaction given by the function below.

> **NOTE:** that the function only shows the different actions hardware should take to carry out the transaction, but your software will be a state machine.

```
void i2c_write_slave_reg(void)
{
    // This will accomplish this:
    // slave_addr[slave_reg] = data;

    i2c_start();
    i2c_write(slave_addr);
    i2c_write(slave_reg);  // This is "M" for "memory address of the slave"
    i2c_write(data);

    /* Optionaly write more data to slave_reg+1, slave_reg+2 etc. */
    // i2c_write(data); /* M + 1 */
    // i2c_write(data); /* M + 2 */
    i2c_stop();}
```

Read Transaction

An I2C read is slightly more complex and involves more protocol to follow. What we have to do is switch from "write-mode" to "read-mode" by sending a repeat start, but this time with an ODD address. This transition provides the protocol to allow the slave device to start to control the data line. **You can consider an I2C even address being "write-mode" and I2C odd address being "read-mode".**

When the master enters the "read mode" after transmitting the read address after a repeat-start, the master begins to "ACK" each byte that the slave sends. When the master "NACKs", it is an indication to the slave that it doesn't want to read anymore bytes from the slave.

> Again, the function shows what we want to accomplish. The actual driver should use state machine logic to carry-out the entire transaction.

```
void i2c_read_slave_reg(void)
{
  i2c_start();
  i2c_write(slave_addr);
  i2c_write(slave_reg);

  i2c_start();                  // Repeat start
  i2c_write(slave_addr | 0x01); // Odd address (last byte Master writes, then Slave begins to control
  char data = i2c_read(0);      // NACK last byte
  i2c_stop();
}
void i2c_read_multiple_slave_reg(void)
{
  i2c_start();
  i2c_write(slave_addr);
  i2c_write(slave_reg);

  // This will accomplish this:
  // d1 = slave_addr[slave_reg];
  // d2 = slave_addr[slave_reg + 1];
  // d3 = slave_addr[slave_reg + 2];
  i2c_start();
  i2c_write(slave_addr | 0x01);
  char d1 = i2c_read(1);      // ACK
  char d2 = i2c_read(1);      // ACK
  char d3 = i2c_read(0);      // NACK last byte
  i2c_stop();}
```

I2C Slave State Machine Planning

Before you jump right into the assignment, do the following:

- Read and understand how an I2C master performs slave register read and write operation
    - Look at existing code to see how the master operation handles the I2C state machine function
    - This is important so you can understand the existing code base
- Next to each of the master state, determine which slave state is entered when the master enters its state
- Determine how your slave memory or registers will be read or written

It is important to understand the states, and use the datasheet to figure out what to do in the state to reach the next desired state given in the diagrams below.

In each of the states given in the diagrams below, your software should take the step, and the hardware will go to the next state granted that no errors occur. To implement this in your software, you should:

1. Perform the planned action after observing the current state
2. Clear the "SI" (state change) bit for HW to take the next step
3. The HW will then take the next step, and trigger the interrupt when the step is complete

Master Write

In the diagram below, note that when the master sends the "R#", which is the register to write, then the slave state machine should save this data byte as it's INDEX location. Upon the next data byte, the indexed data byte should be written.

Stop here and do the following:

1. Check `i2c.c`
2. Compare the code to the state diagram below

?

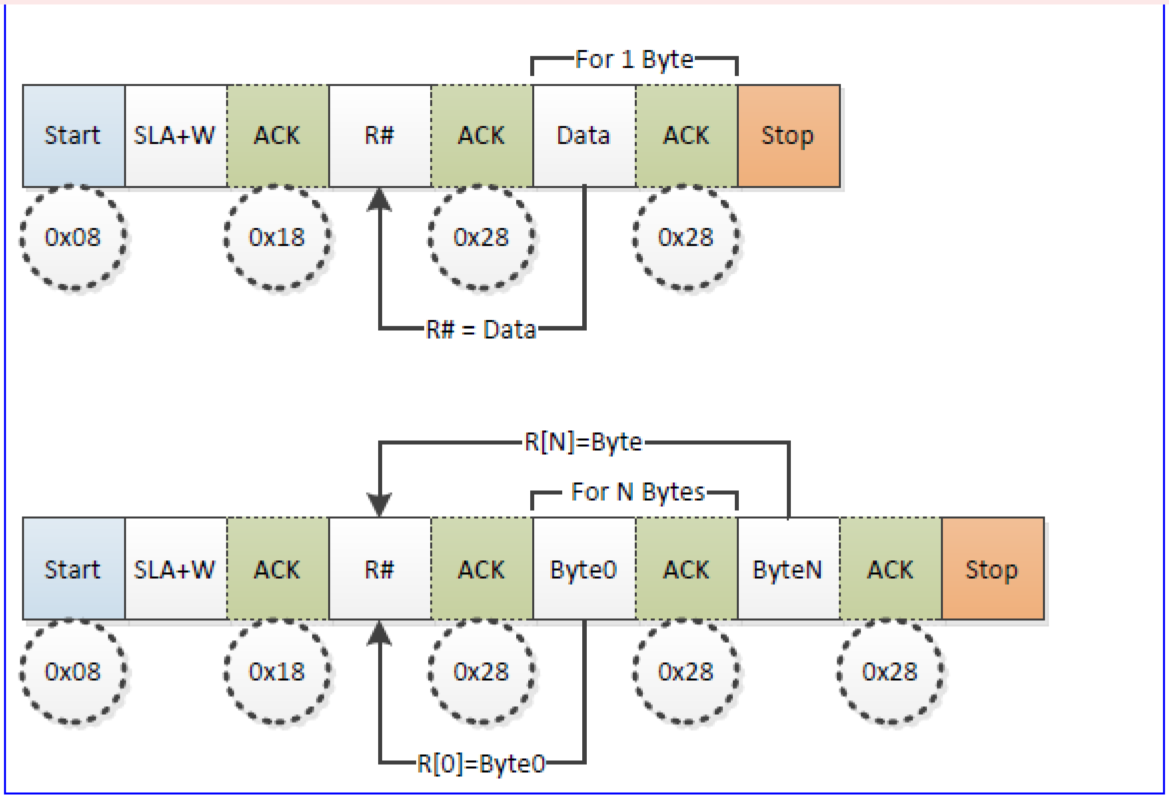


Figure x. I2C Master Write Transaction

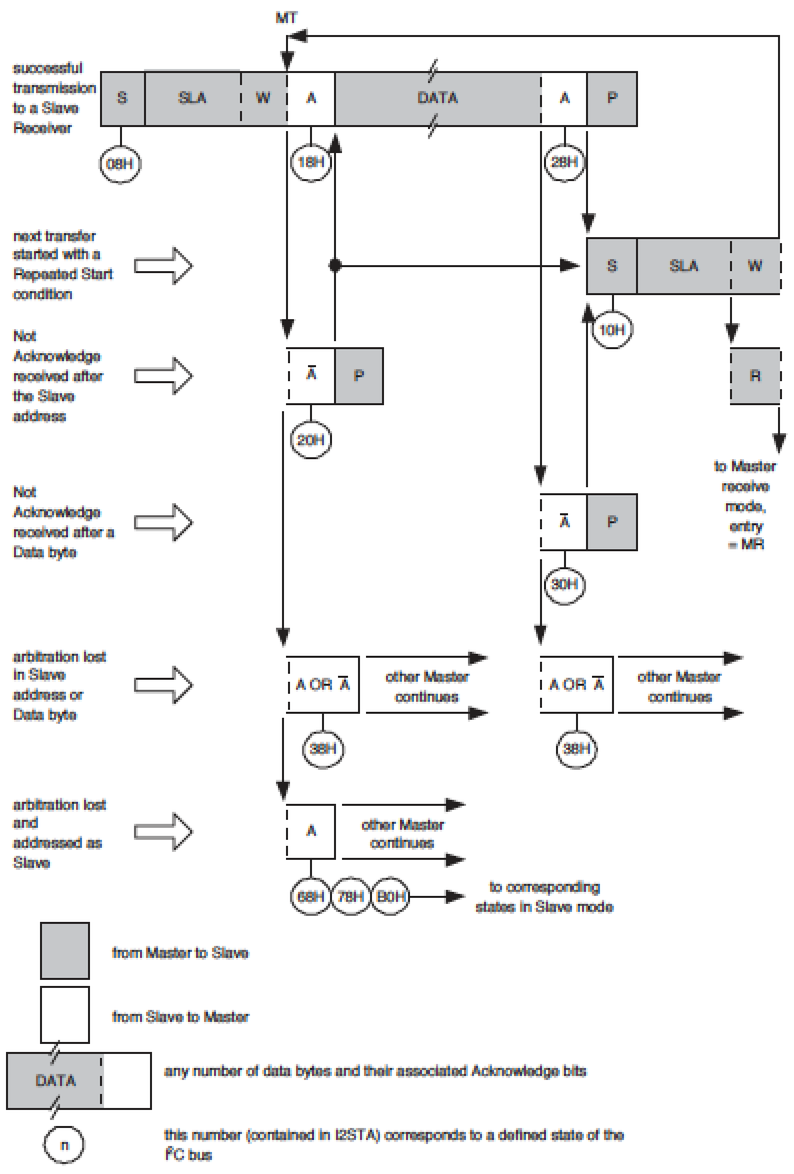
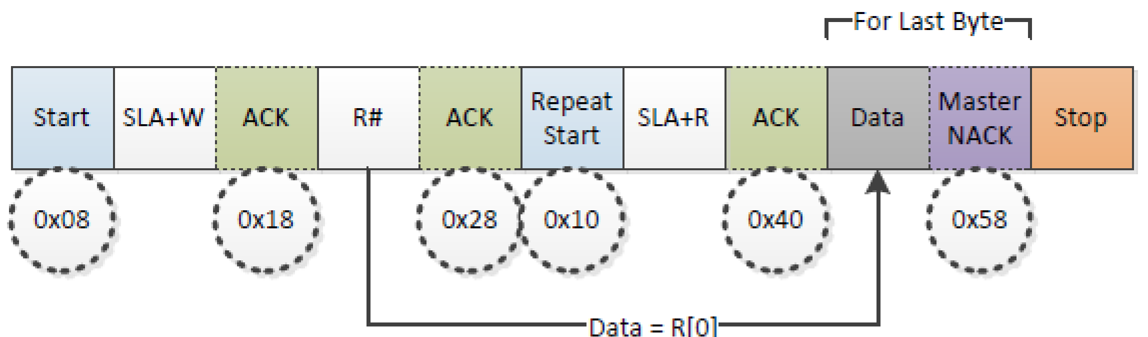

MT
successful transmission to a Slave Receiver
S
SLA
W
A
DATA
A
P
08H
18H
28H
next transfer started with a Repeated Start condition
S
SLA
W
10H
R
Not Acknowledge received after the Slave address
Ā
P
20H
to Master receive mode, entry = MR
Not Acknowledge received after a Data byte
Ā
P
30H
arbitration lost in Slave address or Data byte
A OR Ā
other Master continues
38H
A OR Ā
other Master continues
38H
arbitration lost and addressed as Slave
A
other Master continues
68H
78H
B0H
to corresponding states in Slave mode
from Master to Slave
from Slave to Master
DATA
A
any number of data bytes and their associated Acknowledge bits
n
this number (contained in I2STA) corresponds to a defined state of the I²C bus

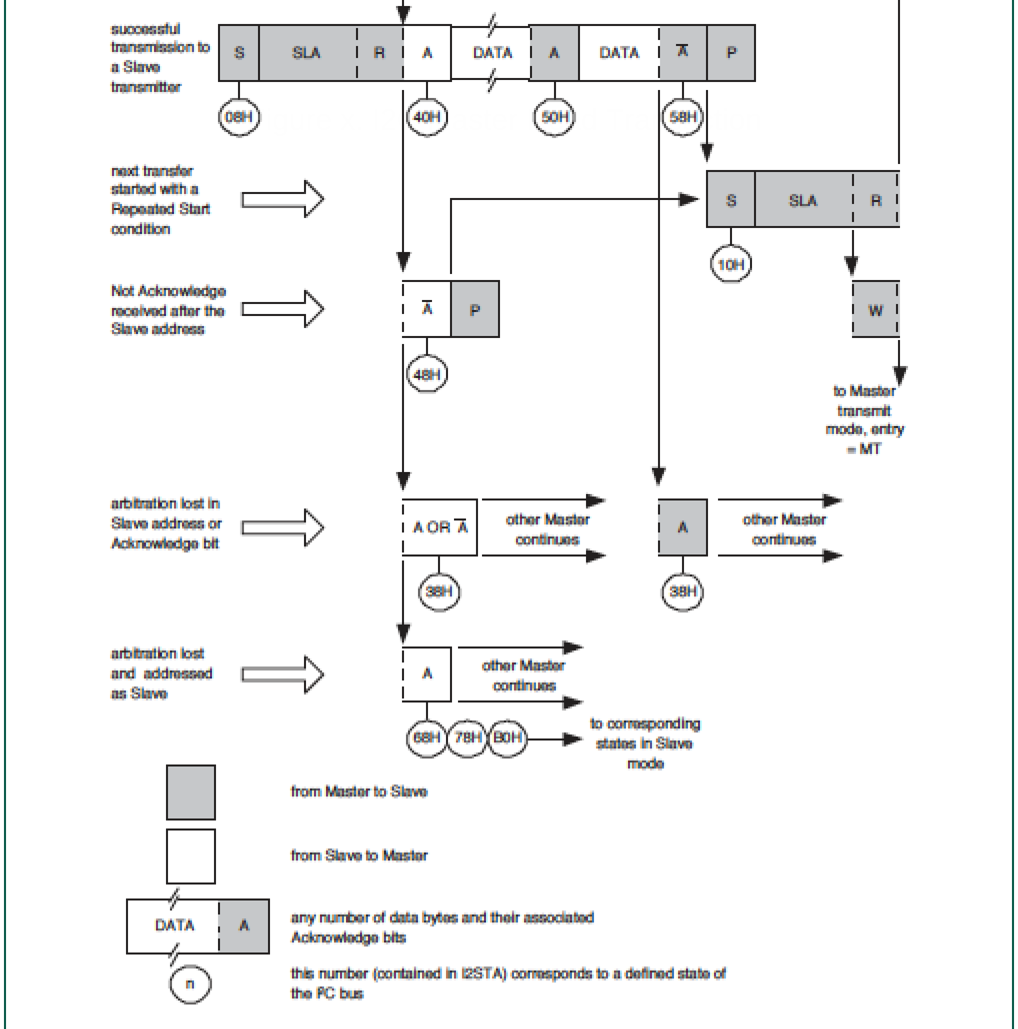
Fig 94. Format and states in the Master Transmitter mode

Figure x. Section 19.9.1 in LPC40xx User Manual

Master Read

In the diagram below, the master will write the index location (the first data byte), and then perform a repeat start. After that, you should start returning your indexed data bytes.





successful transmission to a Slave transmitter
S
SLA
R
A
DATA
A
DATA
Ā
P
08H
40H
50H
58H
next transfer started with a Repeated Start condition
S
SLA
R
10H
Not Acknowledge received after the Slave address
Ā
P
48H
W
to Master transmit mode, entry = MT
arbitration lost in Slave address or Acknowledge bit
A OR Ā
other Master continues
38H
A
other Master continues
38H
arbitration lost and addressed as Slave
A
other Master continues
68H
78H
B0H
to corresponding states in Slave mode
from Master to Slave
from Slave to Master
DATA
A
any number of data bytes and their associated Acknowledge bits
n
this number (contained in I2STA) corresponds to a defined state of the I²C bus

Fig 95. Format and states in the Master Receiver mode

Figure x. Section 19.9.2 in LPC40xx User Manual

Assignment

Design your I2C slave state machine diagrams with the software intent in each state.

- Re-draw the *Master Read* and *Master Write* diagrams while simultaneously showing the slave state.
    - In each slave state, show the action you will perform.
      (Refer to section 19.9.3 and 19.9.4 (for SJ1 board) OR 22.9.3 and 22.9.4 (for SJ2 board) in LPC40xx user manual for the slave states)
    - For instance, when the Master is at state 0x08, determine which state your slave will be at.

    In this state when your slave gets addressed, an appropriate intent may be to reset your variables.

- You will be treated as an engineering professional, and imagine that your manager has given you this assignment and is asking for the state machine diagrams before you start the code implementation.
    - Demonstrate excellence, and do not rely on word-by-word instructions. If you get points deducted, do not complain that "I was not asked to do this". Do whatever you feel is necessary to demonstrate excellence.

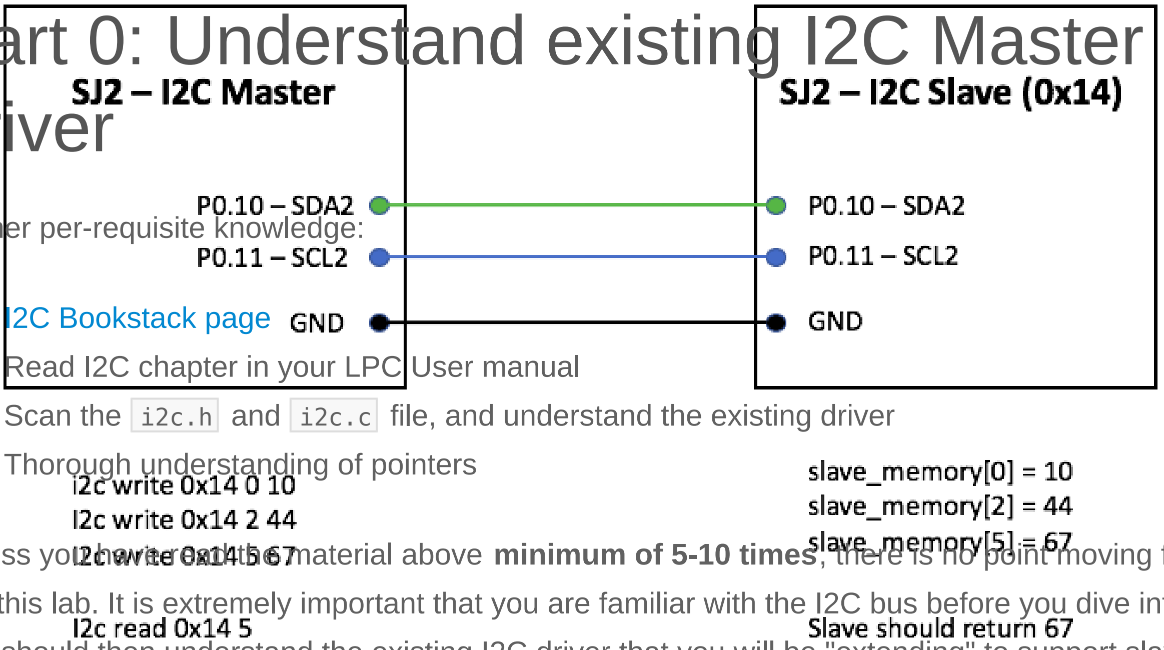
Brainstorming!

- I2C is talking to two accelerometers and started at the same time then which one will first get the bus?
- What is clock stretching in the I2C Bus, when it happens and how can we solve this problem?

Lab: I2C Slave

Overall Objective

We will setup one SJ2 board as a Master board, and another as a Slave board, and the Master board will attempt to read and write memory to/from the Slave board.



```
i2c write 0x14 0 10
I2c write 0x14 2 44
I2c write 0x14 5 67

I2c read 0x14 5
```

```
slave_memory[0] = 10
slave_memory[2] = 44
slave_memory[5] = 67

Slave should return 67
```

Part 0: Understand existing I2C Master driver

Gather per-requisite knowledge:

1. I2C Bookstack page
2. Read I2C chapter in your LPC User manual
3. Scan the `i2c.h` and `i2c.c` file, and understand the existing driver
4. Thorough understanding of pointers

Unless you have read the material above **minimum of 5-10 times**, there is no point moving forward into this lab. It is extremely important that you are familiar with the I2C bus before you dive into this lab. You should then understand the existing I2C driver that you will be "extending" to support slave operation. Start with the header file interface, and correlate that to the other reading material mentioned above.

Part 1: Attempt to read and write existing devices

Part 1.a: Setup

- Load sample project, reboot, and take a note of the I2C discovered devices on startup

    - There is a boot message indicating various different addresses of the existing I2C slaves
- Turn on debug printfs at `i2c.c`, re-compile, and re-load the modified project to the board
    - This will let you trace through I2C states as a read or write transaction is happening
    - Look for `#define I2C__ENABLE_DEBUGGING` and set this to non-zero

Part 1.b: Read and Write Slave memory

Next step is to get familiar with the I2C CLI command and the existing I2C master driver, which allows you to read and write an I2C Slave devices' memory.

- Load the sample project
- Try the `i2c read` and `i2c write` CLI commands and attempt to read data from a device, such as the acceleration sensor.
    - This sensor is at address `0x38`
    - Open up `MMA8452Q.pdf` and read the `WHO_AM_I` register
    - Write the `CTRL_REG1` register and activate the sensor
    - Attempt to read X and the Z-axis and observe that the Z-axis should output the effect of gravity
- **Pause here**, and go through the output debug printfs and cross check them against the I2C Master state machine diagram

---

Part 2: Draw your I2C Slave State Machine

Your objective is to act like an I2C slave device. Just like the acceleration sensor, you will act like a slave device with your own memory locations that a Master device can read or write. The first step in doing so is to draw your I2C Slave driver state machine that you will implement in your code.

Build the following table for your I2C Slave states:

HW State	Why you got here?	Action to take

The key to making the table above is:

- Understanding the I2C Master driver
    - For each HW state of the Master driver, there is a corresponding Slave state

- **Table 512 - 515** in the LPC User Manual

Part 3: Implement your I2C Slave driver

Part 3.a

In this step, we will configure your Slave board to respond to a particular I2C bus address. We will add minimal code necessary to be able to be recognized by another master.

Believe it or not, you only have to write a single function to be recognized as a slave on the I2C bus. Since the I2C Master driver is already initialized, you already have the setup for the Pin FUNC, and the I2C clock, so merely setting up the slave address recognition is needed. Locate the I2C registers and configure your Slave Address registers.

Pause here, and attempt to reboot the I2C Master board, and it should recognize your Slave address on startup. Do not proceed further until you can confirm that your slave address setup is working.

```
// TODO: Create i2c_slave_init.hvoid i2c2__slave_init(uint8_t slave_address_to_respond_to);
```

Notice that on the I2C slave board, it will print an entry to a new (un-handled) state each time the master board reboots and tries to "discover" the slave devices.

Hint

- The I2C `DAT` register is only valid prior to clearing the SI bit
  - Clear the SI bit AFTER you read the `DAT` register
- I2C `ADDR` and `MASK` register
  - MASK can be set to non-zero to allow multiple address recognition
  - For example, if MASK is `0xC0` and ADDR is `0xF0`, then your slave will recognize the following addresses:
    `0x30`, `0x70`, `0xB0`, and `0xF0` because MASK bit7 and bit6 indicate that those will not be "matched"

Part 3.b

In this step, you will do the following:

- Add extra states to the I2C Interrupt Handler that are relevant to your I2C slave driver
- Invoke callbacks which will interface your virtual I2C device

```
// TODO: Create a file i2c_slave_functions.h and include this at the existing i2c.c file
/**
 * Use memory_index and read the data to *memory pointer
 * return true if everything is well
 */
bool i2c_slave_callback__read_memory(uint8_t memory_index, uint8_t *memory);
/**
 * Use memory_index to write memory_value
 * return true if this write operation was valid
 */
bool i2c_slave_callback__write_memory(uint8_t memory_index, uint8_t memory_value);
// TODO: You can write the implementation of these functions in your main.c (i2c_slave_functionc.c is
```

The functions you added above should be called by your i2c.c state machine handler when the right state has been encountered. **This should directly be related to your Part 2**.

Inside the interrupt, you may have to save data variables as the interrupt cycles through the I2C states. The best way to handle this is to add more data members to `typedef i2c_s;` after which you can access members of this struct inside the I2C ISR.

Part 3.c

In this step, we provide the skeleton code that you need to integrate in your `main.c` at your Slave board.

```
#include "i2c_slave_functions.h
#include "i2c_slave_init.h"
static volatile uint8_t slave_memory[256];
bool i2c_slave_callback__read_memory(uint8_t memory_index, uint8_t *memory) {
  // TODO: Read the data from slave_memory[memory_index] to *memory pointer
  // TODO: return true if all is well (memory index is within bounds)
}
bool i2c_slave_callback__write_memory(uint8_t memory_index, uint8_t memory_value) {
  // TODO: Write the memory_value at slave_memory[memory_index]
```

```
    // TODO: return true if memory_index is within bounds
  }
  int main(void) {
    i2c__init_slave(0x86);

    /**
     * Note: When another Master interacts with us, it will invoke the I2C interrupt
     *.       which will then invoke our i2c_slave_callbacks_*() from above
     *        And thus, we only need to react to the changes of memory
     */
    while (1) {
      if (slave_memory[0] == 0) {
        turn_on_an_led(); // TODO
      } else {
        turn_off_an_led(); // TODO
      }

      // of course, your slave should be more creative than a simple LED on/off
    }
    return -1;}
```

Final Requirements

I2C State Machine Lab Submission:

1. For part 1 (Attempt to read and write existing devices) - Serial terminal screenshots of memory being read and written.
2. Logic analyzer screenshots for WHO_I_AM, CTRL_REG1 and for reading Accelerometer XYZ axis data.
3. For Part 2 - Design I2C slave state machine for **Master Write and Slave Read**, **Master Read and Slave Write** along with the table of I2C slave state in both cases.

I2C Lab:

1. Complete Part 3 - Attach Screenshots if you have any.

2. Extra Credit for Creative Slave device and Multi-byte read/write.

- Creative Slave device - Your slave device should act like a real device and do things based on the setting of its slave memory registers
- Your slave should be able to read and write multiple registers in a single I2C transaction.

I2C communication on the same board

An alternative to test the I2C Leader-member (master-slave) communication on the same board is to make one of the i2c port(I2C2) as Leader(Master) and any of the available ports (I2C0/I2C1) as member (Slave).

Steps:

- Create `i2c_slave_init.h and i2c_slave_init.c` files. Define a function `i2c_slave_init(...)` which assigns slave address and slave configuration for a given i2c port. (Refer i2c__initialize function in i2c.c)
- Also refer `peripherals_init.c` to get more info on how I2C2 is initialized.
- Call the `i2c_slave_init(...)` function in your main.c
- Connect the SDA and SCL pins of I2C2 with the respective SDA and SCL pins of I2C0/1
- Flash the code and try issuing `i2c detect` command to see if you see a response from member device

You will still need to check a couple of things (`CLK, IOPIN` registers) before the driver becomes fully functional.