OSU Mechanical Engineering

Smart Products Laboratory

Digital Communication | Four

ME Course Number
Spring 2019

Table of Contents

List of Figures

1 Overview

In lab four, the fundamentals of digital communications are demonstrated through implementing the commonly used I2C protocol. You will construct classes in C++ to manage multiple EEPROM through the use of a multiplexer.

2 Background

The information below should provide a basic introduction to the concepts.

2.1 Definitions

The key concepts for this lab are defined and demonstrated below

- **Word:** The data which is read from or written across a digital communication line or to a memory register. A word length is the number of bits contained with the data packet. The word address is the location in memory that the data is stored.
- **Address:** The unique slave address of the external i2c device.
- **Register:** external i2c devices have registers which may require you pointing to.
- **Base:** the address that points to the beginning of a sequence of registers
- **Offset:** the position of a register in this sequence relative to the base address

The following information will provide a general overview of the necessary information needed to create and implement your solution. For more extensive information see the data sheets for the required hardware.

Functions available

- **int wiringPiI2CSetup (int devId) ;**

This initialises the I2C system with your given device identifier. The ID is the I2C number of the device and you can use the **i2cdetect** program to find this out. wiringPiI2CSetup() will work out which revision Raspberry Pi you have and open the appropriate device in /dev.

The return value is the standard Linux filehandle, or -1 if any error – in which case, you can consult *errno* as usual.

E.g. the popular MCP23017 GPIO expander is usually device Id 0x20, so this is the number you would pass into wiringPiI2CSetup().

> *For all the following functions, if the return value is negative then an error has happened and you should consult errno.*

Figure 2 – I2C setup for wiringPi

- **int wiringPiI2CRead (int fd) ;**

Simple device read. Some devices present data when you read them without having to do any register transactions.

- **int wiringPiI2CWrite (int fd, int data) ;**

Simple device write. Some devices accept data this way without needing to access any internal registers.

- **int wiringPiI2CWriteReg8 (int fd, int reg, int data) ;**
- **int wiringPiI2CWriteReg16 (int fd, int reg, int data) ;**

These write an 8 or 16-bit data value into the device register indicated.

- **int wiringPiI2CReadReg8 (int fd, int reg) ;**
- **int wiringPiI2CReadReg16 (int fd, int reg) ;**

These read an 8 or 16-bit value from the device register indicated.

Figure 1 – I2C read/write functions for wiringPi

2.3 External Memory and Multiplexer

The external memory and multiplexer you will be using in lab are an EEPROM, MC24xx01, and a dual 1 to 4 multiplexer, CD4052B. The EEPPROM allows for storage of information external to the RPi and the multiplexer allows for multiple EEPROMs to be used with one pair of I2C signal lines. The multiplexer is bidirectional meaning that signals can travel in both directions. You will need to choose the right pull up resistors to ensure that the power and rise time requirements are met.

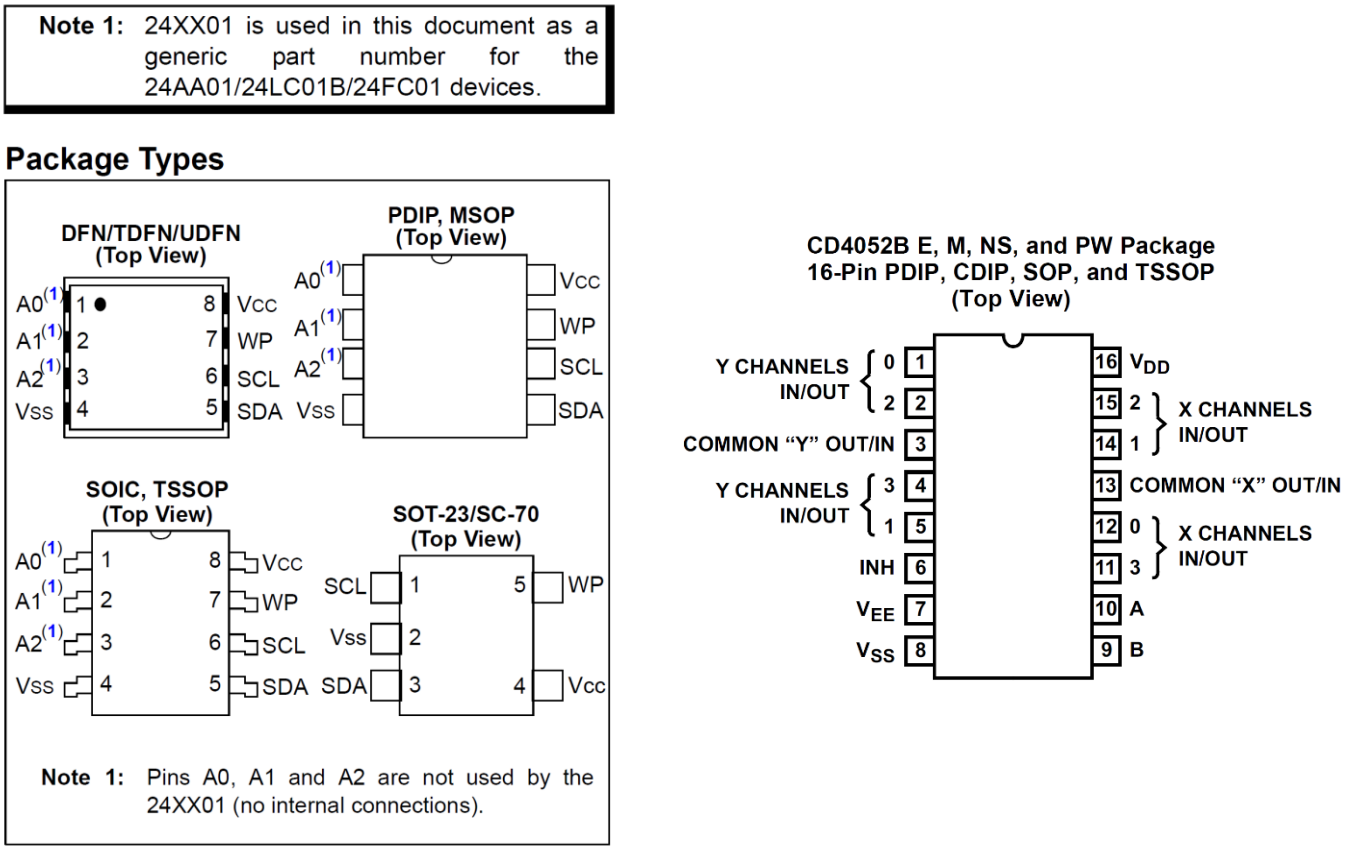


Figure 3 – CD4052B and MC24XX01 pinouts

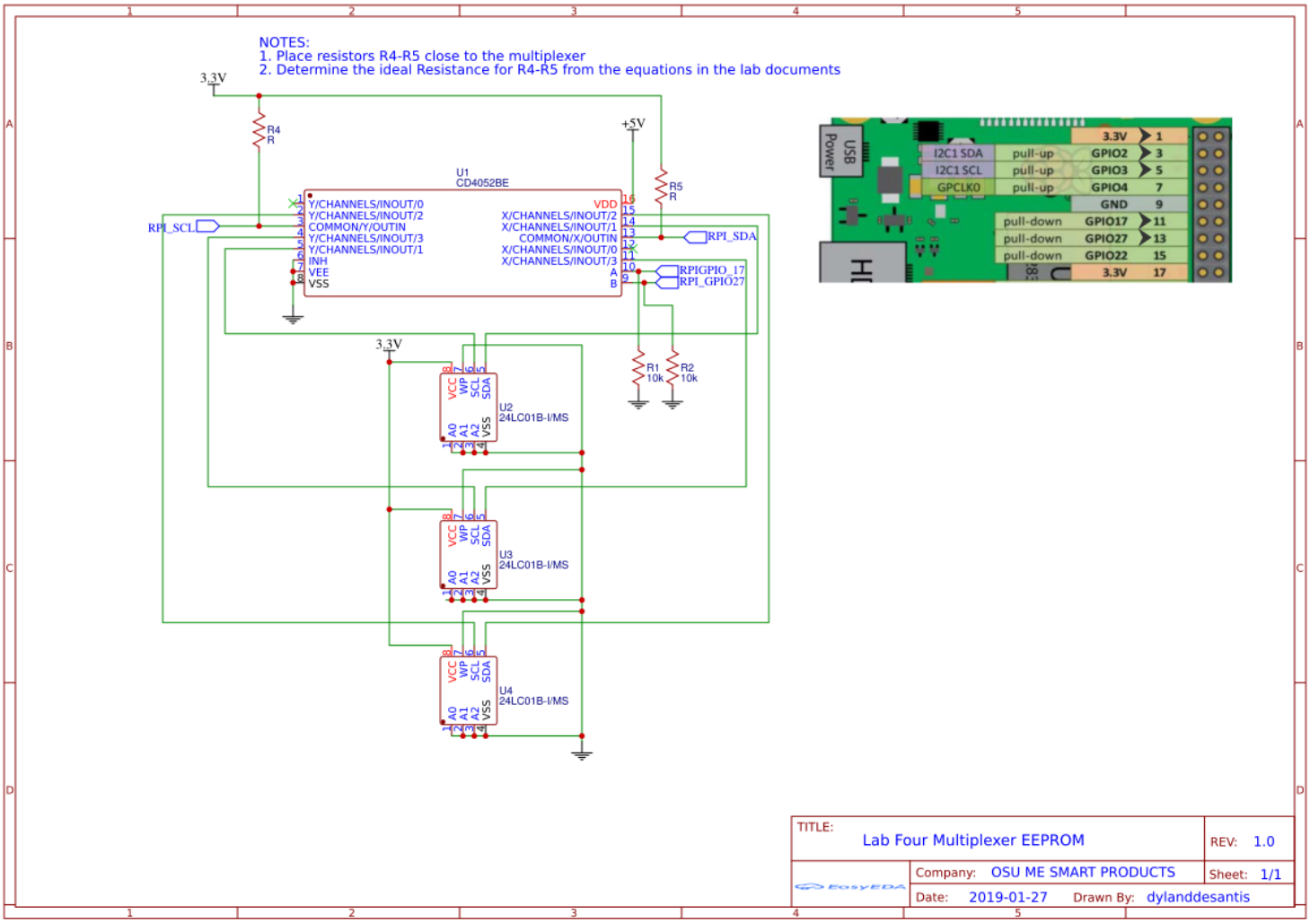


Figure 4 – CD4052B and MC24XX01 circuit for Lab

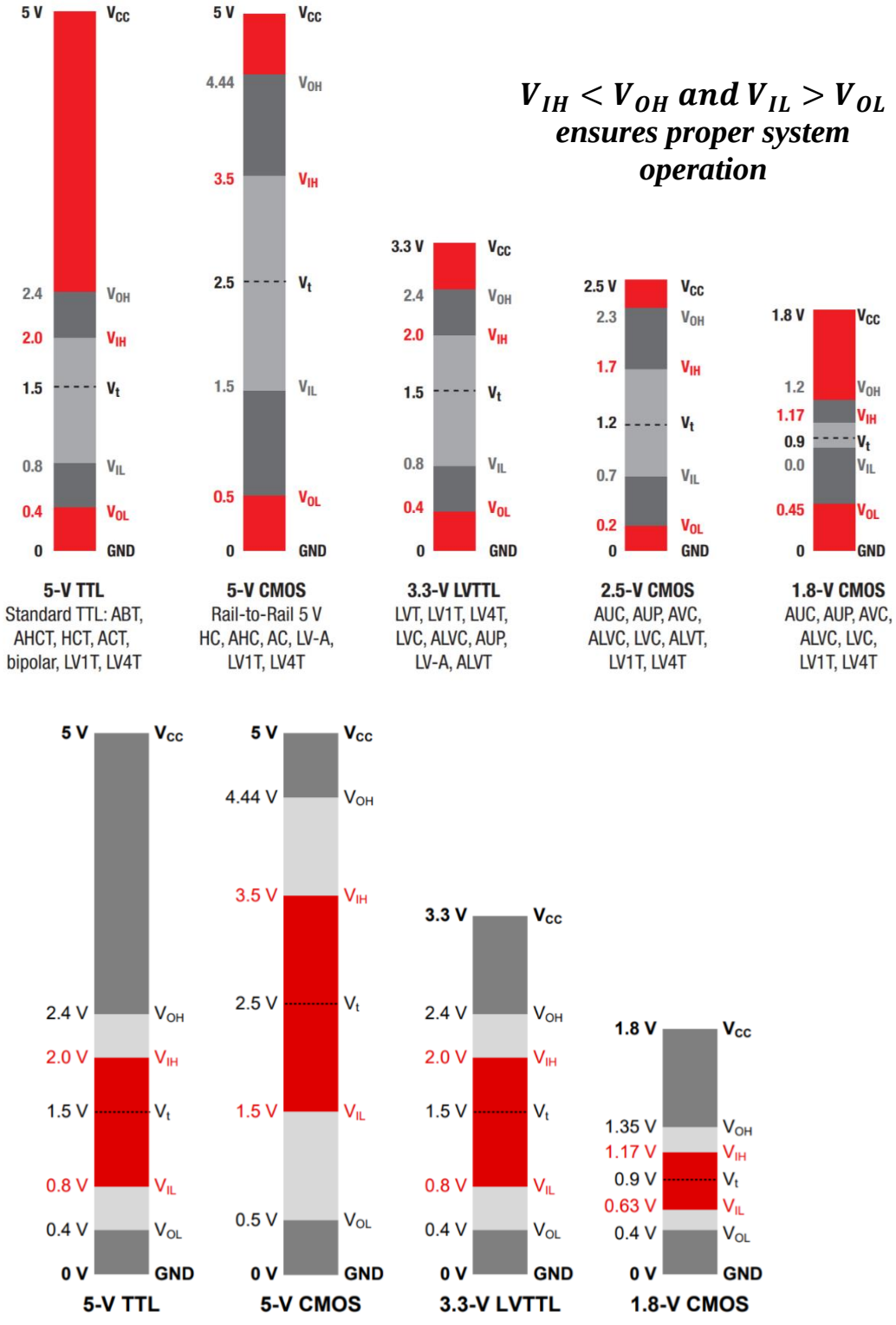


Figure 5 – Logic Level Thresholds

SPI bus (Serial Peripheral Interface) hardware overview

- In SPI interfaces the master can connect to one or more slave devices
- In cases when multiple slave devices are used, the master will use multiple chip select ($\overline{CS}$) lines

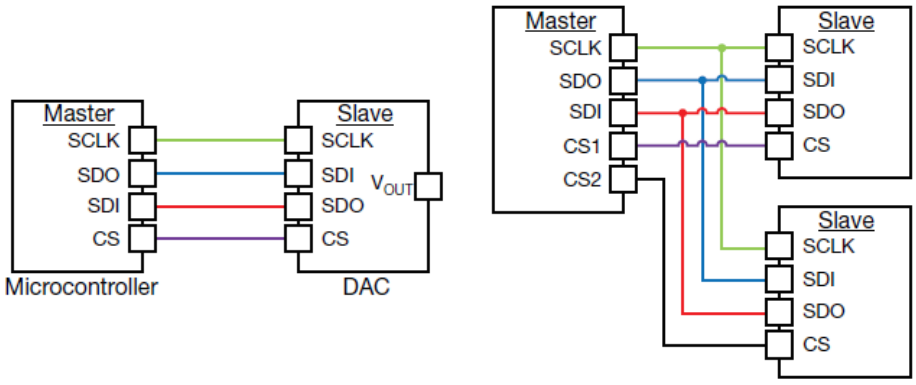


Figure 69: SPI master and slave configurations

Data and control lines

$\overline{CS}$ (chip select) = sometimes referred to as slave select. $\overline{CS}$ is driven by the master and arbitrates over the SPI bus. When driven low, the SPI bus is active.

SDO/SDI (serial data in and serial data out) = these names describe data flow for the device. The system names describe the data flow relationship between the master and slave. System names: MOSI = Master Out Slave In and MISO = Master In Slave Out. Example: SDO on a slave is MISO in the system and SDI is MOSI in the system.

SCLK (serial clock) = this is a square wave driven by the SPI master. Data on SDO and SDI have relative timing to the SCLK signal which controls the latching of the data on the SPI bus.

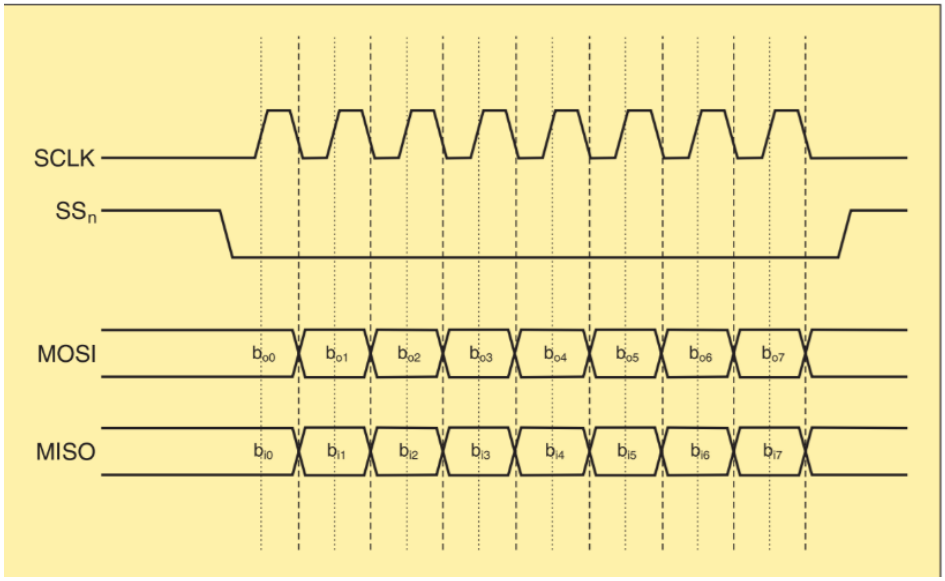


Figure 6 – SPI overview

I²C bus (Inter-Integrated Circuit) hardware overview

- On I²C buses the master can connect to one or more slave devices
- The slave is selected by its I2C address. This allows one controller to connect to many slaves on the two-wire bus.

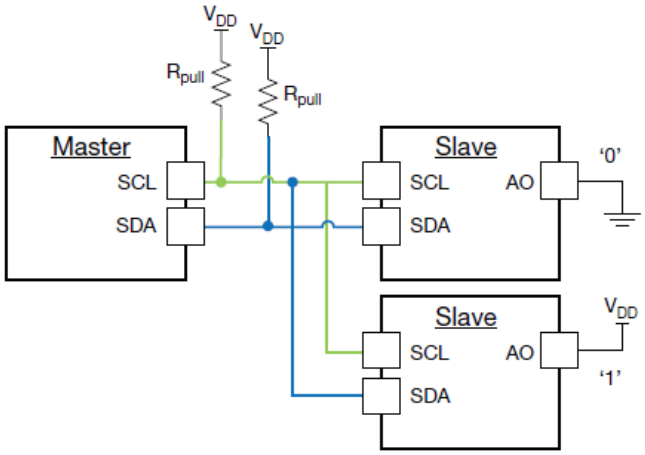


Figure 74: : I²C master and slave hardware connections

Data and control lines

SCL (serial clock) = this is a square wave driven by the master that controls how fast data is sent and when data is latched to the slave device(s)

SDA (serial data) = both master and slave place data on this line in sync with the clock pulses in a half-duplex fashion. Data on this line includes address, control, and communication data.

Master Controls SDA Line

Slave Controls SDA Line

Write to One Register in a Device

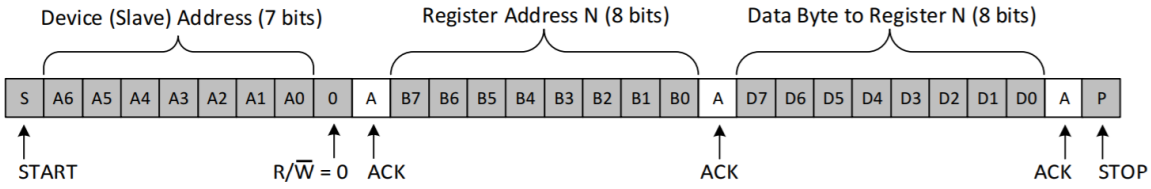


Figure 8. Example I²C Write to Slave Device's Register

Read From One Register in a Device

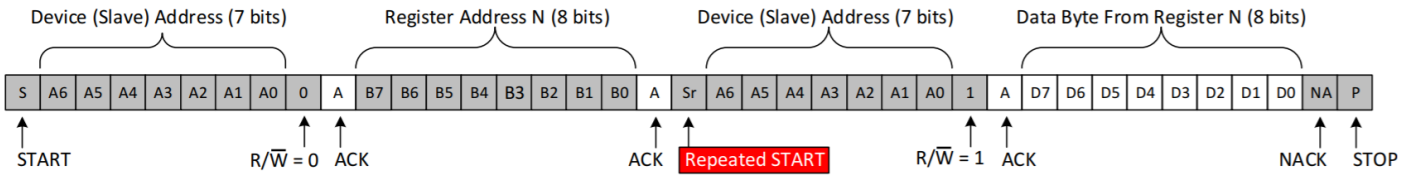
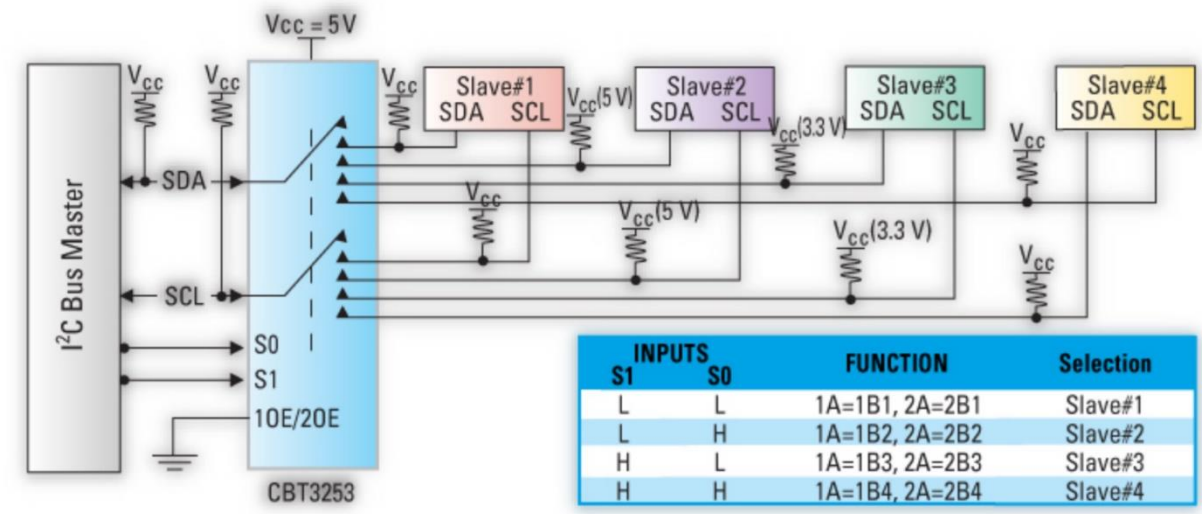


Figure 9. Example I²C Read from Slave Device's Register

Figure 7 – I2C overview



INPUTS S1	INPUTS S0	FUNCTION	Selection
L	L	1A=1B1, 2A=2B1	Slave#1
L	H	1A=1B2, 2A=2B2	Slave#2
H	L	1A=1B3, 2A=2B3	Slave#3
H	H	1A=1B4, 2A=2B4	Slave#4

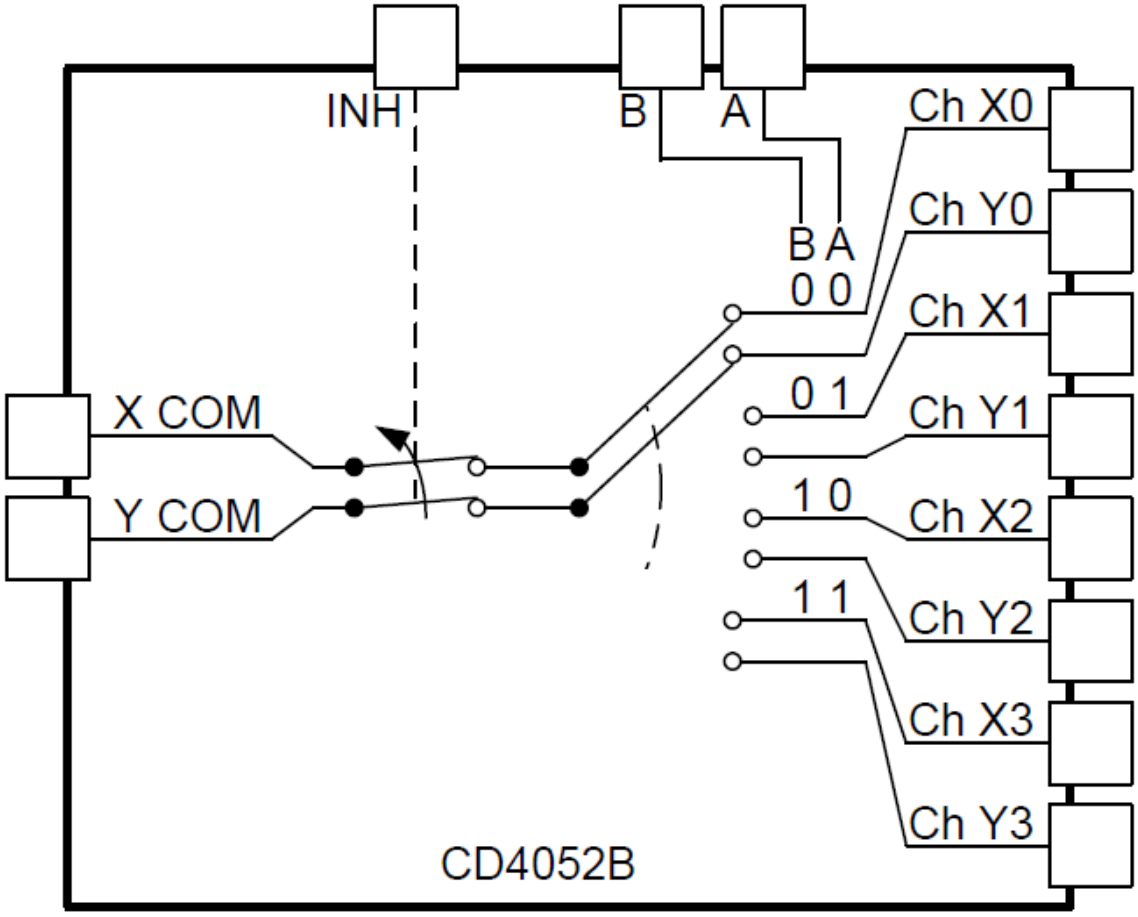


Figure 8 – Example Multiplexer Circuit

Pin Functions CD4052B

PIN		I/O	DESCRIPTION
NO.	NAME		
1	Y CH 0 IN/OUT	I/O	Channel Y0 in/out
2	Y CH 2 IN/OUT	I/O	Channel Y2 in/out
3	Y COM OUT/IN	I/O	Y common out/in
4	Y CH 3 IN/OUT	I/O	Channel Y3 in/out
5	Y CH 1 IN/OUT	I/O	Channel Y1 in/out
6	INH	I	Disables all channels. See Table 1.
7	V_{EE}	—	Negative power input
8	V_{SS}	—	Ground
9	B	I	Channel select B. See Table 1.
10	A	I	Channel select A. See Table 1.
11	X CH 3 IN/OUT	I/O	Channel X3 in/out
12	X CH 0 IN/OUT	I/O	Channel X0 in/out
13	X COM IN/OUT	I/O	X common out/in
14	X CH 1 IN/OUT	I/O	Channel in/out
15	X CH 2 IN/OUT	I/O	Channel in/out
16	V_{DD}	—	Positive power input

Table 1. Truth Table(1)

INPUT STATES				ON CHANNEL(S)
INHIBIT	C	B	A	
CD4051B				
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	X	X	X	None
CD4052B				
0		0	0	0x, 0y
0		0	1	1x, 1y
0		1	0	2x, 2y
0		1	1	3x, 3y
1		X	X	None
CD4053B				
0	X	X	0	ax
0	X	X	1	ay
0	X	0	X	bx
0	X	1	X	by
0	0	X	X	cx
0	1	X	X	cy
1	X	X	X	None

(1) X = Don't Care

Figure 9 – CD4052B Functional Descriptions

```
#include <iostream>
#include <chrono>
#include <thread>

    enum class TimeScale : int {
        seconds = 1000000,
        milliseconds = 1000,
        microseconds = 1
    };

    class Util
    {
    public:
        static void delay(int, TimeScale);

    };

    /**
    General function to provide a delay in the program in microseconds.

    @param time the amount of time to delay the program in microseconds
    */
    void Util::delay(int time, TimeScale ts)
    {


        switch (ts)
        {
        case TimeScale::seconds:
            std::this_thread::sleep_for(std::chrono::seconds(time));
            break;
        case TimeScale::milliseconds:
            std::this_thread::sleep_for(std::chrono::milliseconds(time));
            break;
        case TimeScale::microseconds:
            std::this_thread::sleep_for(std::chrono::microseconds(time));
            break;
        default:
            break;
        }
        return;
    }
int main()
{
    std::cout<<" starting...."<<std::endl;
    Util::delay(5, TimeScale::seconds);
    std::cout<<" ending...."<<std::endl;
}
```

Figure 10 – Example main file "exampleDelay.cpp" for executing the delay function

FIGURE 5-2: ADDRESS SEQUENCE BIT ASSIGNMENTS

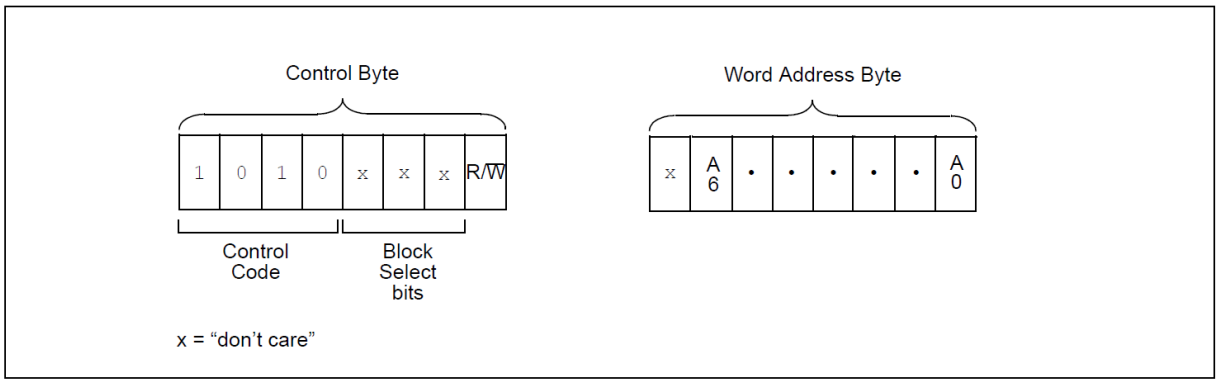


FIGURE 6-1: BYTE WRITE

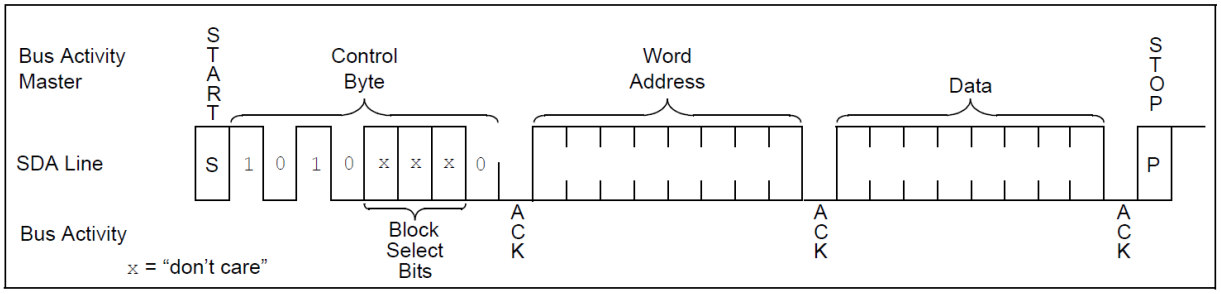


FIGURE 8-1: CURRENT ADDRESS READ

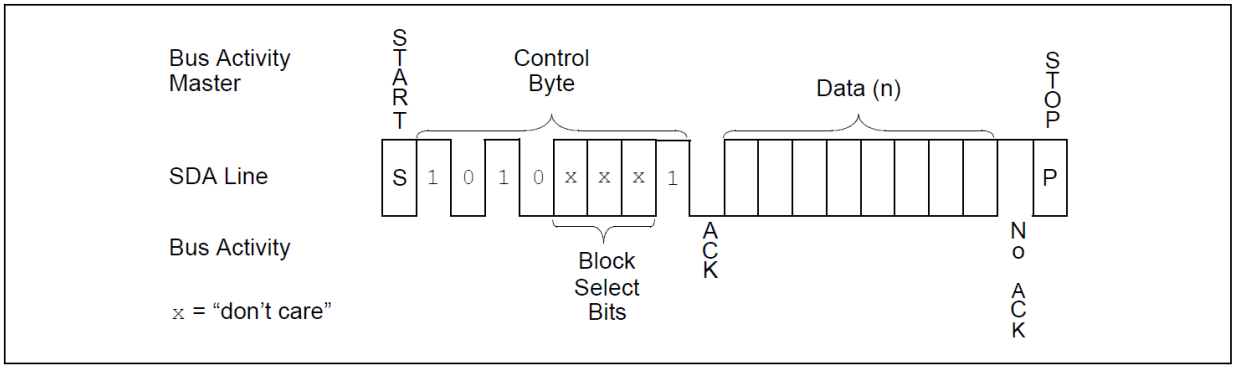


Figure 11 – MC24xx01 I2C Descriptions

I²C pull-up resistor selection

$$R_{Pull(Min)} = \frac{(V_{DD} - V_{OLMAX})}{I_{SinkMax}} \quad \text{(179) Minimum I²C pull-up resistance}$$

$$R_{Pull(Max)} = \frac{t_r}{(0.8473 \times C_b)} \quad \text{(180) Maximun I²C pull-up resistance}$$

Where

$R_{Pull(Min)}$ = this is the minimum pull-up resistance. This will give the shortest rise time. Using a pull-up smaller than this will draw too much current when the output transistor is on (logic low) and violate the maximum logic low output specification.

$R_{Pull(Max)}$ = maximum pull-up resistance. This will give the longest rise time. Using a pull-up resistance larger than this will violate timing requirements.

V_{DD} = supply voltage

V_{OLMAX} = maximum logic low output found in device data sheet. Typically 0.4V.

$I_{SinkMax}$ = maximum sink current when the output transistor is on (logic low) found in device data sheet. Typically 3mA.

C_b = bus capacitance. Depends on width and length of PCB trace (see equation 155), and the capacitance of the devices connected to the bus.

Example

Find a pull-up resistor for: $V_{DD} = 5V$, $V_{OLMAX} = 0.4V$, $t_r = 300ns$, and $C_b = 100pF$.

Answer

$$R_{Pull(Min)} = \frac{(V_{DD} - V_{OLMAX})}{I_{SinkMax}} = \frac{(5V - 0.4V)}{0.003A} = 1.53k\Omega$$

$$R_{Pull(Max)} = \frac{t_r}{(0.8473 \times C_b)} = \frac{300ns}{0.8473 \times 100pF} = 3.54k\Omega$$

$R_{Pull} = 2k\Omega$ Selected as a standard value between $R_{Pull(Min)}$ and $R_{Pull(Max)}$

Figure 12 – Selecting Pull up resisters for I2C

3 References & Resources

3.1 References

The following references may be helpful to complete the lab.

1. http://www.ti.com/solution/service_robots?variantid=11713&subsystemid=19430
2. https://e2e.ti.com/blogs_/b/analogwire/archive/2017/11/02/how-to-reduce-the-number-of-i-o-pins-with-a-switch-matrix-module
3. http://wiringpi.com/reference/i2c-library/
4. http://www.ni.com/tutorial/6552/en/
5. https://chromium.googlesource.com/chromium/src/+/HEAD/styleguide/c++/c++-dos-and-donts.md
6. https://google.github.io/styleguide/cppguide.html
7. http://alanclements.org/clocks%20and%20timing.html
8. http://www.physics.ohio-state.edu/~hughes/cdf_osu/xft/documents/layout.pdf

4 Pre-lab

The prelab should haven been completed in the previous week and will be turned in with this lab report.

5 Laboratory

Complete the exercises and save all code and follow the procedure to turn in your work.

5.1 Requirements & Hardware

1. Raspberry Pi
2. Multiplexer and EEPROM – find the data sheets at the links below or in Carmen.

Component	Link	Part Number
Dual 4x1 Multiplexer	https://www.digikey.com/short/pt9r88	CD4052BE
EEPROM – I2C	https://www.digikey.com/short/pt98zz	24LC01B-I/P

3. Bread Board
4. Oscilloscope
5. Wires
6. Resistors – refer to the example diagram in the lab section to calculate the needed resistance
7. USB memory stick.

5.2 Procedure

1. Two options:
    a. (Easy) Go to http://wiringpi.com/download-and-install/ and install wiringPi on your raspberry pi according to the Plan A instructions the website provides you.
        i. http://wiringpi.com/reference/i2c-library/
    b. (Hard) Use the linux based device driver commands to interface with the hardware directly. See the following sites for this option
        i. https://www.kernel.org/doc/Documentation/i2c/dev-interface
        ii. https://www.kernel.org/doc/Documentation/i2c/
        iii. https://www.kernel.org/doc/Documentation/i2c/i2c-protocol
        iv. https://elinux.org/Interfacing_with_I2C_Devices
        v. https://gist.github.com/JamesDunne/9b7fbedb74c22ccc833059623f47beb7
2. If you need help getting jump started go to https://learn.sparkfun.com/tutorials/raspberry-pi-spi-and-i2c-tutorial/all
3. Make sure you enable GPIO and I2C functionalities in the raspberry pi interface settings.
4. You may solve the software portion of this lab using any combination of classes, MUST USE CLASSES, however if you may use the following classes structures if you want
    a. I2C class with the following members and methods
        i. Member – mDevCom: file descriptor for i2c device driver
        ii. Member – mSAddress: slave address of the current i2c peripheral
        iii. Member – mStatus: Boolean check to see if the file descriptor.
        iv. Member – mAddressSet: Boolean check to see if slave address has been set.
        v. Method – i2cBegin: set the slave address if needed and open the file descriptor if needed.
        vi. Method – setAddress: set the address of the current peripheral
        vii. Method – read8: read a byte from a register of the peripheral
        viii. Method – write8: write a byte from a register of the peripheral
        ix. Constructors – necessary constructors
    b. MC24xx01: class to control the communication with the EEPROM and inherits the I2C class
        i. Member – mAddress: the slave address of the EEPROM
        ii. Method – readWord: read a byte from a register of the EEPROM
        iii. Method – writeWord: write a byte from a register of the EEPROM
        iv. Constructors – necessary constructors
    c. CD4052: class that can control the multiplexer and switch between configurations
        i. Member – mGPIO: member that holds the GPIO instance to control the multiplexed signal.
        ii. Member – mControlPins: member that contains an ordered list of which GPIO pins are being used to control the multiplexer selection line.
        iii. Member – mTruthTable: member that contains the logic table needed for switching the multiplexed lines.

iv. Method – setControlPins: method that takes and sets the control pins as well as setup any necessary configuration for the GPIO control pins.
v. Method – setTruthTable: method that sets the desired logic table for switching the configurations.
vi. Method – switchConfig: method for switching the multiplexed lines given a desired configuration.
vii. Constructors – necessary constructors

d. SPmultimemory : class that contains the necessary methods and members for overseeing the multiplexer and EEPROM. This class inherits the MC24xx01 and CD4052 classes.
i. Nested enumerated class that contains the options for communicating to the different EEPROMS
ii. Method – readMem: method that performs the necessary operations for switching and reading from the desired register from a desired EEPROM.
iii. Method – writeMem: method that performs the necessary operations for switching and writing a desired word to the desired register from a desired EEPROM.
iv. Constructors – necessary constructors

5. If you end up using wiringPi, when you compile you will need to include the library. At the end of your compilation command for the g++ module, include the following
a. "g++ -o blah blah.h blah.cpp … blah2.cpp -lwiringPi"
6. You may want to test your code incrementally, refer to my main code to see the tests I implemented.
7. When finished with the exercises in the next section, you will then generate a brief report and submit your work on Carmen. The submission will be in the form of a zip file and must conform to the following requirements:
a. The zip file should be named as follows
i. SP19_lab04_Lastname_dotnumber.zip
b. Inside the zip files should be only the following
i. All programming files that you created to implement the solution, including a cpp file which demonstrates your solution.
ii. One report in PDF format which will contain only the following in 5 pages or less,
1. Date, Name, Class, and Lab number(s) at the top.
2. Section one - prelab:
a. Report your solutions and explanations that were completed in the prelab.
3. Section two – Code
a. Describe how you implemented your solution.
b. What happens when you don't successfully implement timing control when performing consecutive reads and writes to the EEPROM.

   c. How did you implement incremental testing to efficiently develop your code?
4. Section three – Oscilloscope analysis
   a. Report your measurements and screen shots of your digital and analog signals.
   b. Describe what observations you made about the timing of signal propagating when using multiplexers.

5.3 Exercises

Create the necessary software to be able to control the multiple EEPROMs using a multiplexer.

1. Overall your code should be able to:
   a. Change which EEPROM you want to communicate with through I2C
   b. Read a byte of data from a register from a specified EEPROM
   c. Write a byte of data to a register of a specified EEPROM.
   d. Implement the necessary timing control to assure that data is successfully read from and written to the EEPROM.
2. Calculate the correct pullup resistors needed given the data sheets for the EEPROM and CD4052 chips.
3. Build the circuit, referring the example circuit provided previously. Hookup the digital probe and analog probe, to the SCL and SDA common lines that connect directly to the Pi. Hookup additional analog probes on the SCL and SDA data lines that connect directly to the EEPROM. Try to capture the bus data using the bus analysis on the oscilloscope. Use the necessary functionality on the oscilloscope, i.e. trigger, cursors, etc., to capture the signals. Calculate:
   a. The clock frequency
   b. The rise time for rising edge on the data line.
   c. The propagation delay of each of the I2C lines between either side of the multiplexer.
   d. The overshoot if any for the rising and falling line.

You may demonstrate that your code is working using the main file code as shown in Figure 13. I will have a similar test script that will test out your code and you will be graded on its performance, as well as other factors.

6 Rubric

Exercise	*Performance*	*Completeness*	*Cleanliness*	*Clarity*	*Total*
Code	/ 200	/ 30	/ 10	/ 10	/ 250
Prelab	/ 100	-	-	/ 10	/110
Report	-	/ 50	-	-	/ 50
Total	**/ 300**	**/ 80**	**/10**	**/20**	**/410**

1. **Performance**– Does your code produce the correct results.
2. **Completeness**– How much of the exercise did you complete.
3. **Cleanliness** – Of the completed portion of the exercise how well organized and structured is the work (i.e. is your code compact and clean, does it flow from one section to another)
4. **Clarity –** Of the completed portion of the exercise, how easy is it to infer what is occurring from the context of your work (i.e. do you use concise and proper use of commenting, if needed, did you explain your work/methodology in your report).

// Example main implementation code for lab four

```
#include "SPmultimemory.h"
#include <iostream>

int main()
{
    /*
        // Commented out after testing
        // Tests for checking to see if the GPIO class is working
        sp::RP3GPIO gpio1;
        gpio1.openGPIO();
        sp::RP3GPIO gpio = gpio1;
        std::cout<< "Toggling Pin Mode ...";
        gpio.pinMode(17, sp::GPIO::PinModes::OUTPUT);
        gpio.pinMode(27, sp::GPIO::PinModes::OUTPUT);// set the pinmode
        std::cout << "toggle complete" << std::endl;
        std::cout<< "Writing Low : "<< std::endl;
        gpio.digitalWrite(17, sp::GPIO::DigitalOut::LOW); // initally write set the pin to low
        int pin_val = gpio.digitalRead(17); // read the value of the pin
        std::cout << "Value of pin " << 17 << " is  " << pin_val << std::endl;
        std::cout<< "Writing High : "<< std::endl;
        gpio.digitalWrite(17, sp::GPIO::DigitalOut::HIGH); //write high to pin 27
        gpio.digitalWrite(27, sp::GPIO::DigitalOut::LOW); // initally write set the pin to low
        pin_val = gpio.digitalRead(17);// read the pin
        std::cout << "Value of pin " << 17 << " is  " << pin_val << std::endl;

        int sAddress = 0x50;

        */
        /*
        // Commented out after testing
        // Tests for checking to see if the I2C class is working
        sp::I2C i2c_test;
        i2c_test.i2cBegin(sAddress);
        i2c_test.setAddress(sAddress);
        int word = i2c_test.read8(0x01);
        std::cout<<"Read: "<<word<<std::endl;
        i2c_test.write8(0x0E, 0x01);
        word = i2c_test.read8(0x01);
        word = i2c_test.read8(0x01);
        gpio.digitalWrite(27, sp::GPIO::DigitalOut::HIGH);//switching configurations
        std::cout<<"Read: "<<word<<std::endl;
        word = i2c_test.read8(0x01);
        std::cout<<"Read: "<<word<<std::endl;
        */

        /*
        // Commented out after testing

        // Tests for checking to see if the MC24xx01 class is working
        sp::MC24xx01 memory_test(sAddress);
        int word = memory_test.readWord(0x00);
        std::cout<<"Read: "<<word<<std::endl;
        memory_test.writeWord(0x4D, 0x00);
        word = memory_test.readWord(0x00);
        std::cout<<"Read: "<<word<<std::endl;

        // Tests for checking to see if the CD4052 class is working
        sp::CD4052 mux_test(gpio);
```

```
59.          int t = 0;
60.          std::cout<<"test "<< t<< std::endl;
61.          std::vector<std::vector<bool>> tTable = { {blah, blah}, {blah, blah}, {blah, blah} };
62.          std::cout<<"test "<< ++t<< std::endl;
63.          std::vector<int> cPins = { 17, 27 };
64.          std::cout<<"test "<< ++t<< std::endl;
65.          mux_test.setTruthTable(tTable);
66.          std::cout<<"test "<< ++t<< std::endl;
67.          mux_test.setControlPins(cPins);
68.          std::cout<<"test "<< ++t<< std::endl;
69.          mux_test.switchConfig(0);
70.          std::cout<<"test "<< ++t<< std::endl;
71.
72.          // Tests for checking to see if the combination of the MC24xx01 class and CD4052 class is working
73.          word = memory_test.readWord(0x00);
74.          std::cout<<"Read: "<<word<<std::endl;
75.          memory_test.writeWord(0x4F, 0x00);
76.          word = memory_test.readWord(0x00);
77.          std::cout<<"Read: "<<word<<std::endl;
78.          mux_test.switchConfig(1);
79.
80.          word = memory_test.readWord(0x00);
81.          std::cout<<"Read: "<<word<<std::endl;
82.          memory_test.writeWord(0x6F, 0x00);
83.          word = memory_test.readWord(0x00);
84.          std::cout<<"Read: "<<word<<std::endl;
85.
86.          mux_test.switchConfig(2);
87.          word = memory_test.readWord(0x00);
88.          std::cout<<"Read: "<<word<<std::endl;
89.          memory_test.writeWord(0x7B, 0x00);
90.          word = memory_test.readWord(0x00);
91.          std::cout<<"Read: "<<word<<std::endl;
92.          */
93.
94.
95.      // Final tests for checking to see if the multimemory class is working
96.      sp::SPmultimemory memory_banks;
97.      int word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B0);
98.      std::cout << "data in memory bank 0: " << std::hex << word << std::endl;
99.      memory_banks.writeMem(0, sp::SPmultimemory::MemBank::B0, 0x01);
100.     word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B0);
101.     std::cout << "data in memory bank 0: " << std::hex << word << std::endl;
102.     word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B1);
103.     std::cout << "data in memory bank 1: " << std::hex << word << std::endl;
104.     memory_banks.writeMem(0, sp::SPmultimemory::MemBank::B1, 0x0A);
105.     word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B1);
106.     std::cout << "data in memory bank 1: " << std::hex << word << std::endl;
107.     word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B2);
108.     std::cout << "data in memory bank 2: " << std::hex << word << std::endl;
109.     memory_banks.writeMem(0, sp::SPmultimemory::MemBank::B2, 0x0C);
110.     word = memory_banks.readMem(0, sp::SPmultimemory::MemBank::B2);
111.     std::cout << "data in memory bank 2: " << std::hex << word << std::endl;
112.
113.     return 0;
114. }
```

Figure 13 – Example main file "labfour.cpp" for executing the multimemory class operations

Appendix

Understanding the Data Sheet – Example: I2C Temperature Sensor (MCP9800)

This example will show you how to interpret a data sheet for a wired-communication based peripheral using a I2C to read from a temperature sensor integrated chip. The data sheet for this example can be found at:

- http://ww1.microchip.com/downloads/en/DeviceDoc/21909d.pdf

2-Wire High-Accuracy Temperature Sensor

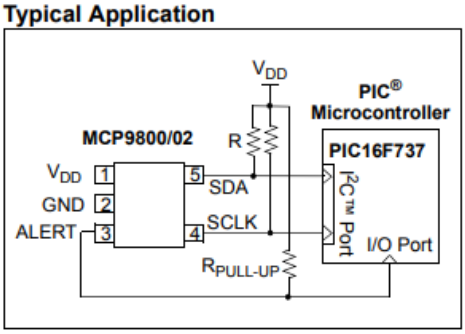
Features:

- Temperature-to-Digital Converter
- Accuracy with 12-bit Resolution:
  - ±0.5°C (typical) at +25°C
  - ±1°C (maximum) from -10°C to +85°C
  - ±2°C (maximum) from -10°C to +125°C
  - ±3°C (maximum) from -55°C to +125°C
- User-selectable Resolution: 9-12 bit
- Operating Voltage Range: 2.7V to 5.5V
- 2-wire Interface: I²C™/SMBus Compatible
- Operating Current: 200 µA (typical)
- Shutdown Current: 1 µA (maximum)
- Power-saving One-shot Temperature Measurement
- Available Packages: SOT-23-5, MSOP-8, SOIC-8

Typical Applications:

- Personal Computers and Servers
- Hard Disk Drives and Other PC Peripherals
- Entertainment Systems
- Office Equipment
- Data Communication Equipment
- Mobile Phones
- General Purpose Temperature Monitoring

Typical Application



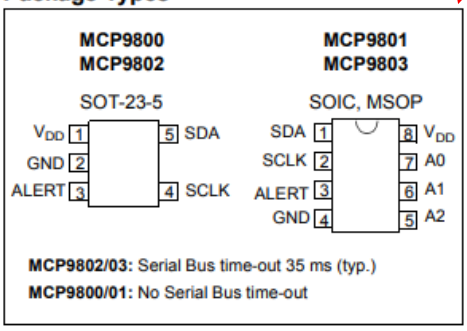
Description:

Microchip Technology Inc.'s MCP9800/1/2/3 family of digital temperature sensors converts temperatures between -55°C and +125°C to a digital word. They provide an accuracy of ±1°C (maximum) from -10°C to +85°C.

The MCP9800/1/2/3 family comes with user-programmable registers that provide flexibility for temperature sensing applications. The register settings allow user-selectable 9-bit to 12-bit temperature measurement resolution, configuration of the power-saving Shutdown and One-shot (single conversion on command while in Shutdown) modes and the specification of both temperature alert output and hysteresis limits. When the temperature changes beyond the specified limits, the MCP9800/1/2/3 outputs an alert signal. The user has the option of setting the alert output signal polarity as an active-low or active-high comparator output for thermostat operation, or as temperature event interrupt output for microprocessor-based systems.

This sensor has an industry standard 2-wire, I²C™/SMBus compatible serial interface, allowing up to eight devices to be controlled in a single serial bus. These features make the MCP9800/1/2/3 ideal for sophisticated multi-zone temperature-monitoring applications.

Package Types



MCP9802/03: Serial Bus time-out 35 ms (typ.)
MCP9800/01: No Serial Bus time-out

Most of the time, the most important information is on the front page. So, what to look for:

Operating Ranges: How am I to power the device, how much power will it consume, is the current and voltage compatible with my main MCU

Communication Options: How am I to communicate with the device: I2C, SPI, CAN, Analog…

Package types: how am I building the circuit around the device. Two general options: through-hole and surface mount. These are surface mounts.

MCP9800/1/2/3

More details on the electrical characteristics tells you what the values of device's properties should be under both operating conditions and maximum conditions.

1.0 ELECTRICAL CHARACTERISTICS

†Notice: Stresses above those listed under "Maximum ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at those or any other conditions above those indicated in the operational listings of this specification is not implied. Exposure to maximum rating conditions for extended periods may affect device reliability.

Absolute Maximum Ratings †

V_{DD} 6.0V
Voltage at all Input/Output pins .....GND – 0.3V to 5.5V
Storage temperature .....-65°C to +150°C
Ambient temp. with power applied .....-55°C to +125°C
Junction Temperature (T_J) ..... 150°C
ESD protection on all pins (HBM:MM) .....(4 kV:400V)
Latch-Up Current at each pin..... ±200 mA

DC CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, V_{DD} = 2.7V to 5.5V, GND = Ground, and T_A = -55°C to +125°C.

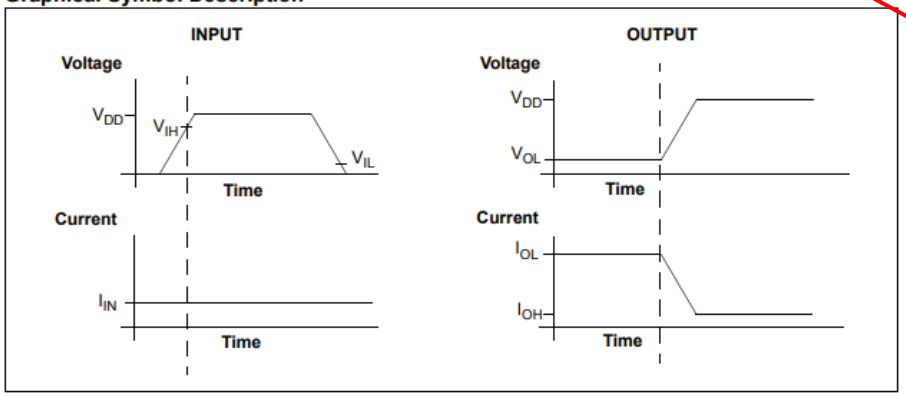
Parameters	Sym	Min	Typ	Max	Unit	Conditions
Power Supply						
Operating Voltage Range	V_{DD}	2.7	—	5.5	V	
Operating Current	I_{DD}	—	200	400	µA	Continuous Operation
Shutdown Current	I_{SHDN}	—	0.1	1	µA	Shutdown mode
Power-on-Reset Threshold (POR)	V_{POR}	—	1.7	—	V	V_{DD} falling edge
Line Regulation	Δ°C/ΔV	—	0.2	—	°C/V	V_{DD} = 2.7V to 5.5V
Temperature Sensor Accuracy						
Accuracy with 12-bit Resolution:						
T_A = +25°C	T_{ACY}	—	±0.5	—	°C	V_{DD} = 3.3V
-10°C < T_A ≤ +85°C	T_{ACY}	-1.0	—	+1.0	°C	V_{DD} = 3.3V
-10°C < T_A ≤ +125°C	T_{ACY}	-2.0	—	+2.0	°C	V_{DD} = 3.3V
-55°C < T_A ≤ +125°C	T_{ACY}	-3.0	—	+3.0	°C	V_{DD} = 3.3V
Internal ΣΔ ADC						
Conversion Time:						
9-bit Resolution	t_{CONV}	—	30	75	ms	33 samples/sec (typical)
10-bit Resolution	t_{CONV}	—	60	150	ms	17 samples/sec (typical)
11-bit Resolution	t_{CONV}	—	120	300	ms	8 samples/sec (typical)
12-bit Resolution	t_{CONV}	—	240	600	ms	4 samples/sec (typical)
Alert Output (Open-drain)						
High-level Current	I_{OH}	—	—	1	µA	V_{OH} = 5V
Low-level Voltage	V_{OL}	—	—	0.4	V	I_{OL} = 3 mA
Thermal Response						
Response Time	t_{RES}	—	1.4	—	s	Time to 63% (89°C) 27°C (Air) to 125°C (oil bath)

DIGITAL INPUT/OUTPUT PIN CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, V_{DD} = 2.7V to 5.5V, GND = Ground and T_A = -55°C to +125°C.

Parameters	Sym	Min	Typ	Max	Units	Conditions
Serial Input/Output (SCLK, SDA, A0, A1, A2)						
Input						
High-level Voltage	V_{IH}	0.7 V_{DD}	—	—	V	
Low-level Voltage	V_{IL}	—	—	0.3 V_{DD}	V	
Input Current	I_{IN}	-1	—	+1	µA	
Output (SDA)						
Low-level Voltage	V_{OL}	—	—	0.4	V	I_{OL} = 3 mA
High-level Current	I_{OH}	—	—	1	µA	V_{OH} = 5V
Low-level Current	I_{OL}	6	—	—	mA	V_{OL} = 0.6V
Capacitance	C_{IN}	—	10	—	pF	
SDA and SCLK Inputs						
Hysteresis	V_{HYST}	0.05 V_{DD}	—	—	V	

Graphical Symbol Description



TEMPERATURE CHARACTERISTICS

Electrical Specifications: Unless otherwise indicated, V_{DD} = +2.7V to +5.5V, GND = Ground.

Parameters	Sym	Min	Typ	Max	Units	Conditions
Temperature Ranges						
Specified Temperature Range	T_A	-55	—	+125	°C	(Note 1)
Operating Temperature Range	T_A	-55	—	+125	°C	
Storage Temperature Range	T_A	-65	—	+150	°C	
Thermal Package Resistances						
Thermal Resistance, 5L-SOT23	θ_{JA}	—	256	—	°C/W	
Thermal Resistance, 8L-SOIC	θ_{JA}	—	163	—	°C/W	
Thermal Resistance, 8L-MSOP	θ_{JA}	—	206	—	°C/W	

Note 1: Operation in this range must not cause T_J to exceed Maximum Junction Temperature (+150°C).

Digital input/output reading levels tell you what is the minimum or maximum voltage/ current that a pin must be at for it know whether the digital state is high or low.

VIH: What is the minimum voltage I can send into the device for it to know that I am telling it the digital state is high.

VIL: What is the maximum voltage I can send into the device for it to know that I am telling it the digital state is low.

VOH: What is the minimum voltage that the device will send to me to indicate a digital state is high. Usually VOH = VDD (the supply voltage).

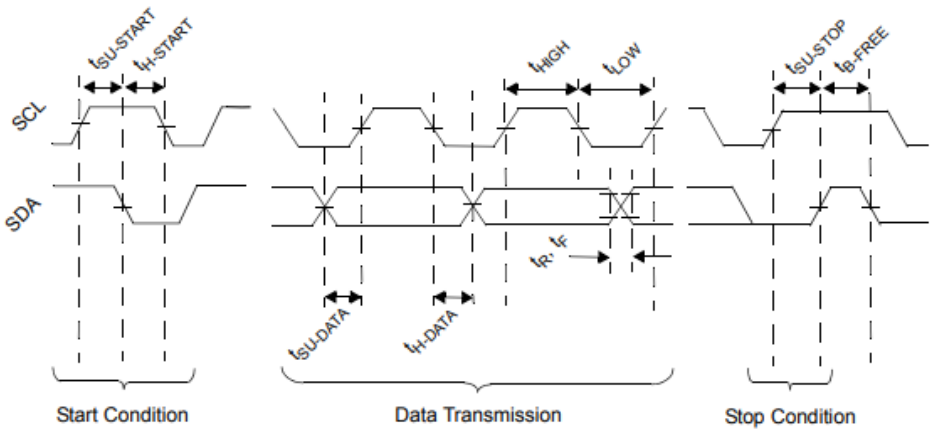
VOL: What is the maximum voltage that the device will send to me to indicate a digital state is low.

SERIAL INTERFACE TIMING SPECIFICATIONS

Electrical Specifications: Unless otherwise indicated, V_{DD} = 2.7V to 5.5V, GND = Ground, -55°C < T_A < +125°C, C_L = 80 pF, and all limits measured to 50% point.

Parameters	Sym	Min	Typ	Max	Units	Conditions
2-Wire I²C™/SMBus Compatible Interface						
Serial Port Frequency	f_{SC}	0	—	400	kHz	I²C **MCP9800/01**
	f_{SC}	10	—	400	kHz	SMBus **MCP9802/03**
Clock Period	t_{SC}	2.5	—	—	µs	
Low Clock	t_{LOW}	1.3	—	—	µs	
High Clock	t_{HIGH}	0.6	—	—	µs	
Rise Time	t_R	20	—	300	ns	10% to 90% of V_{DD} (SCLK, SDA)
Fall Time	t_F	20	—	300	ns	90% to 10% of V_{DD} (SCLK, SDA)
Data Setup Before SCLK High	$t_{SU-DATA}$	0.1	—	—	µs	
Data Hold After SCLK Low	t_{H-DATA}	0	—	0.9	µs	
Start Condition Setup Time	$t_{SU-START}$	0.6	—	—	µs	
Start Condition Hold Time	$t_{H-START}$	0.6	—	—	µs	
Stop Condition Setup Time	$t_{SU-STOP}$	0.6	—	—	µs	
Bus Idle	t_{IDLE}	1.3	—	—	µs	
Time Out	t_{OUT}	25	35	50	ms	**MCP9802/03 only**

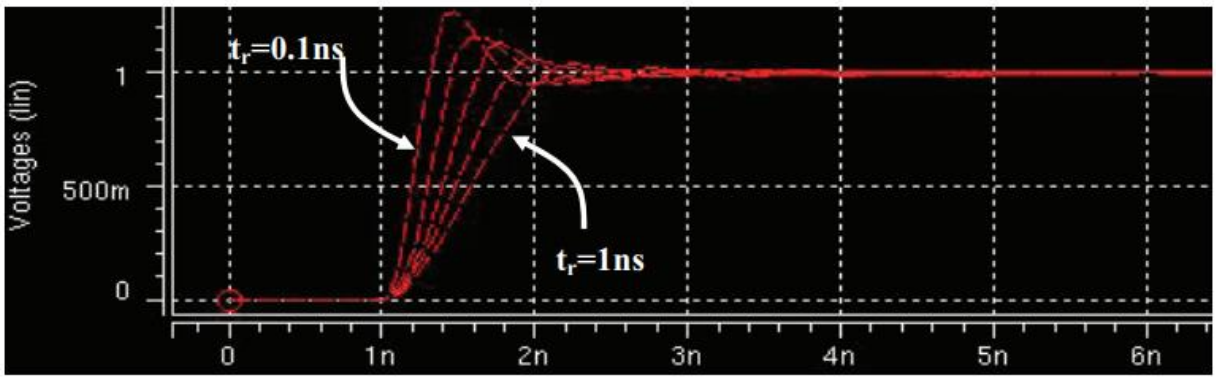
Timing Diagram



Serial timing or sometimes referred to as AC characteristics provide information on how to setup the communication lines and how to engineer the circuitry to avoid missed bits.

Clock times & Frequency: What is the range of frequency which I can transmit in data at. In other words what frequency should my clock line be switch from high to low.

Time constants: Time constants or rise and fall times of the communication lines determine the time in which it should take for digital states to transition from one to the other. Too fast of a transient probably will result in overshoot. Too slow and you could start dropping bits of information



3.0 PIN DESCRIPTION

The descriptions of the pins are listed in Table 3-1.

TABLE 3-1: PIN FUNCTION TABLE

MCP9800 MCP9802 SOT-23-5	MCP9801 MCP9803 MSOP, SOIC	Symbol	Function
5	1	SDA	Bidirectional Serial Data
4	2	SCLK	Serial Clock Input
3	3	ALERT	Temperature Alert Output
2	4	GND	Ground
—	5	A2	Address Select Pin (bit 2)
—	6	A1	Address Select Pin (bit 1)
—	7	A0	Address Select Pin (bit 0)
1	8	V_{DD}	Power Supply Input

3.1 Serial Data Pin (SDA)

The SDA is a bidirectional input/output pin, used to serially transmit data to and from the host controller. This pin requires a pull-up resistor to output data.

3.2 Serial Clock Pin (SCLK)

The SCLK is a clock input pin. All communication and timing is relative to the signal on this pin. The clock is generated by the host controller on the bus.

3.3 Power Supply Input (V_{DD})

The V_{DD} pin is the power pin. The operating voltage, as specified in the DC electrical specification table, is applied on this pin.

3.4 Ground (GND)

The GND pin is the system ground pin.

3.5 ALERT Output

The MCP9800/1/2/3's ALERT pin is an open-drain output pin. The device outputs an alert signal when the ambient temperature goes beyond the user-programmed temperature limit.

3.6 Address Pins (A2, A1, A0)

These pins are device or slave address input pins and are available only with the MCP9801/03. The device addresses for the MCP9800/02 are factory-set.

The address pins are the Least Significant bits (LSb) of the device address bits. The Most Significant bits (MSb) (A6, A5, A4, A3) are factory-set to <1001>. This is illustrated in Table 3-2.

TABLE 3-2: SLAVE ADDRESS

Device	A6	A5	A4	A3	A2	A1	A0
MCP9800/02A0	1	0	0	1	0	0	0
MCP9800/02A1	1	0	0	1	0	0	1
MCP9800/02A2	1	0	0	1	0	1	0
MCP9800/02A3	1	0	0	1	0	1	1
MCP9800/02A4	1	0	0	1	1	0	0
MCP9800/02A5	1	0	0	1	1	0	1
MCP9800/02A6	1	0	0	1	1	1	0
MCP9800/02A7	1	0	0	1	1	1	1
MCP9801/03	1	0	0	1	X	X	X

Note: User-selectable address is shown by X.

Pin descriptions will tell you what each pin does. Some devices allow for programmable slave addresses to be able to have multiple slaves on one line.

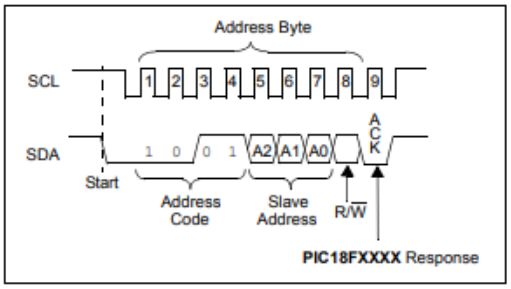


FIGURE 4-1: *Device Addressing.*

4.1.5 DATA VALID

After the Start condition, each bit of data in transmission needs to be settled for a time specified by $t_{SU-DATA}$ before SCL toggles from low-to-high (see **"Serial Interface Timing Specifications" on Page 5**).

With I2C slaves usually have 7-bit address that uniquely identify them. This is always the first packet that is sent of the data line. For this example lets say that we pulled all of the user selectable address pins to low (A0=A1=A2=0). Then our first packet sent would be…

Int slaveAddress = 0x48;

The LSB tells the slave whether the next word transmitted will be written to (W) it or sent from it (R)

REGISTER 5-1: REGISTER POINTER

U-0	U-0	U-0	U-0	U-0	U-0	R/W-0	R/W-0
0	0	0	0	0	0	P1	P0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 7-2 **Unimplemented:** Read as '0'

bit 1-0 **Px<1:0>:** Pointer bits

00 = Temperature register (T_A)
01 = Configuration register (CONFIG)
10 = Temperature Hysteresis register (T_{HYST})
11 = Temperature Limit-set register (T_{SET})

Register Pointer: The pointer to a register may have to be set before accessing information from that specific register. If we want to read the temperature first we have to write to the slave address 0x48 the value …

Int register_readTemp = 0x00;

Or set the configuration

Int register_Config = 0x01;

TABLE 5-1: BIT ASSIGNMENT SUMMARY FOR ALL REGISTERS

Register Pointer P1 P0	MSB/ LSB	Bit Assignment							
		7	6	5	4	3	2	1	0
Ambient Temperature Register (T_A)									
0 0	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	2^{-1}°C	2^{-2}°C	2^{-3}°C	2^{-4}°C	0	0	0	0
Sensor Configuration Register (CONFIG)									
0 1	LSB	One-Shot	Resolution		Fault Queue		ALERT Polarity	COMP/INT	Shutdown
Temperature Hysteresis Register (T_{HYST})									
1 0	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	2^{-1}°C	0	0	0	0	0	0	0
Temperature Limit-Set Register (T_{SET})									
1 1	MSB	Sign	2^6°C	2^5°C	2^4°C	2^3°C	2^2°C	2^1°C	2^0°C
	LSB	2^{-1}°C	0	0	0	0	0	0	0

Bit functionality for Register: Each register will have definition definitions of what each bit of the word it holds. Some registers might need two words to store all the data, in which case two registers are used and the pointer, points to the beginning of first register. Some bits may be data values and other may be functions.

5.3.1 AMBIENT TEMPERATURE REGISTER (T_A)

The MCP9800/1/2/3 has a 16-bit read-only Ambient Temperature register that contains 9-bit to 12-bit temperature data. (0.5°C to 0.0625°C resolutions, respectively). This data is formatted in two's complement. The bit assignments, as well as the corresponding resolution, is shown in the register assignment below.

The refresh rate of this register depends on the selected ADC resolution. It takes 30 ms (typical) for 9-bit data and 240 ms (typical) for 12-bit data. Since this register is double-buffered, the user can read the register while the MCP9800/1/2/3 performs Analog-to-Digital conversion in the background. The decimal code to ambient temperature conversion is shown in Equation 5-2:

EQUATION 5-2:

$$T_A = \text{Code} \times 2^{-4}$$

Where:

T_A = Ambient Temperature (°C)
Code = MCP9800 output in decimal

REGISTER 5-2: AMBIENT TEMPERATURE REGISTER (T_A) – ADDRESS <0000 0000>b

Upper Half:							
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
Sign	2^6 °C	2^5 °C	2^4 °C	2^3 °C	2^2 °C	2^1 °C	2^0 °C
bit 15							bit 8

Lower Half:							
R-0	R-0	R-0	R-0	R-0	R-0	R-0	R-0
2^{-1} °C/bit	2^{-2} °C	2^{-3} °C	2^{-4} °C	0	0	0	0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

Note 1: When the 0.5°C, 0.25°C or 0.125°C resolutions are selected, bit 6, bit 7 or bit 8 will remain clear <0>, respectively.

Here we want to read the data of the temperature readings. So we first would send the address with a write intention then write the pointer to the register.

```
setSlaveAddress(slaveAddress)

write8(register_readTemp);

int tempData_16bit = read16();

int left = (tempData & 0xFF00)>>8;

int right = (tempData & 0xFF);

float temp =((float)(left)) + ((float)(right))/255.0;
```

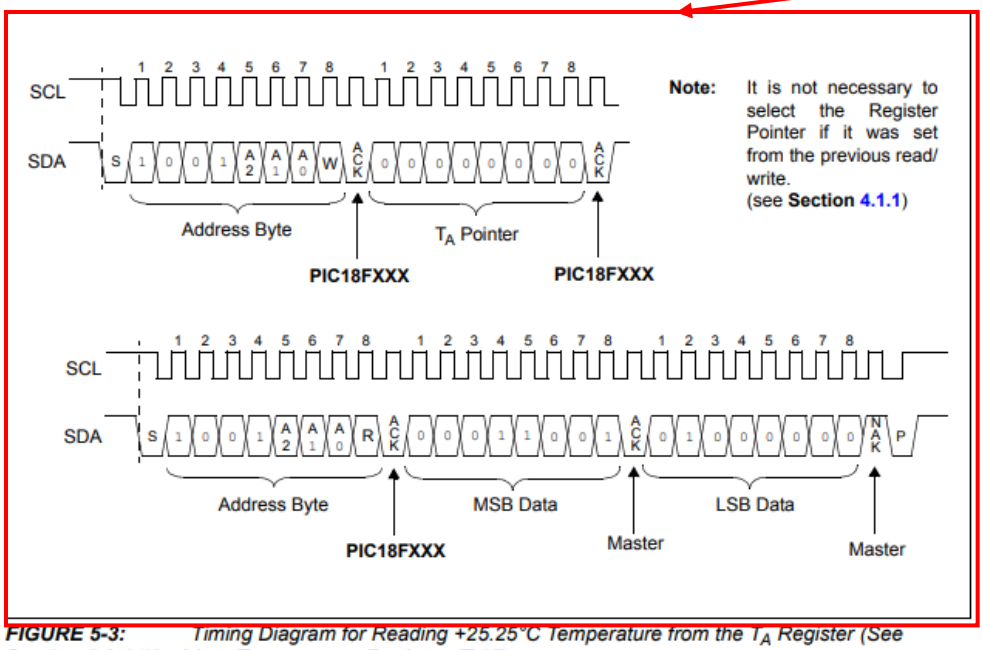


FIGURE 5-3: *Timing Diagram for Reading +25.25°C Temperature from the T_A Register (See Section 5.3.1 "Ambient Temperature Register (T_A)").*

```
#include <iostream>
#include <wiringSerial.h>
#include <wiringPiSPI.h>
#include <wiringPiI2C.h>

#include <sys/stat.h>
#include <linux/i2c-dev.h>
#include <linux/i2c.h>
#include <sys/ioctl.h>


#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <unistd.h>
#include <string.h>


int setSlaveAddress(int fd_i2c, int slaveAddress)
{
    /*
    Sets the I2C to communicate with this device
    */
    if (ioctl(this->fd_i2c, I2C_SLAVE, slaveAddress) < 0)
    {
        printf("problem setting slave\n");
        return -1;
    }
    return 0;
}


int main()
{
    /*
        Code to show how to use WiringPi I2C
    */
    int slaveAddress = 0x48;
    int register_readTemp = 0x00;
    int i2c1_fd = wiringPiI2CSetup(slaveAddress);
    setSlaveAddress(i2c1_fd, slaveAddress);
    int temp_word  = wiringPiI2CRead16(i2c1_fd, register_readTemp, slaveAddr
    int left = (tempData & 0xFF00) >> 8;
    int right = (tempData & 0xFF);
    float temperature = ((float)(left)) + ((float)(right)) / 255.0;
    return 0;
}
```

Figure 14 – Example main file "exampleI2C.cpp" for executing the I2C protocol

SPI data latching

- SPI data is latched on the rising or falling edge of SCLK
- The edge data is latched on is called the critical edge
- The figure below illustrates latching logic 1 on rising edge and logic 0 on falling edge

Figure 70: SPI SCLK critical edge

SPI read sequence example

1. Critical edge is rising edge
2. Master output writing to slave (SDI label relative to slave device)
3. The active low $\overline{CS}$ pin is driven low to 0V, activating the slave SPI bus
4. Data is clocked in from MSB to LSB on the rising edge of SCLK
5. Completed SPI transaction data is binary 1011001

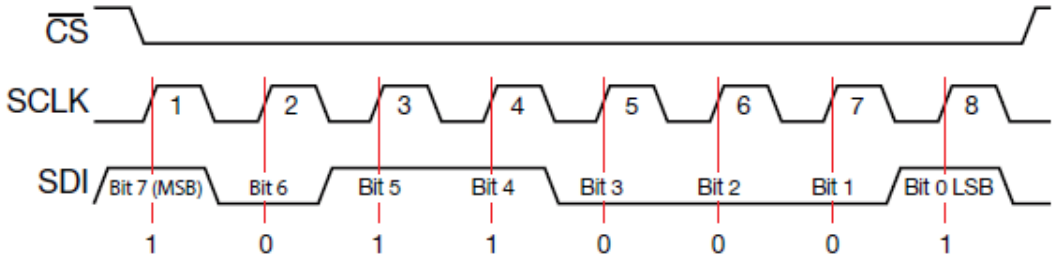


Figure 71: Example SPI write sequence

Figure 15 – SPI data latching and Read/ Write Sequence

SPI critical edge

t_{SU} (setup time) = defines how long before the critical edge that the data on SDI must already be set and settled

t_{HO} (hold time) = defines how long after the critical edge data must be maintained on SDI.

t_{DO} (delay time) - defines the delay before data is valid after the critical edge for SDO.

Violation of any timing requirement could result in corruption of data.

The timing parameters, t_{SU}, t_{HO} and t_{DO}, are defined relative to the critical edge. In the example below for SDI the rising edge of SCLK is the critical edge and for SDO the falling edge of SCLK is the critical edge.

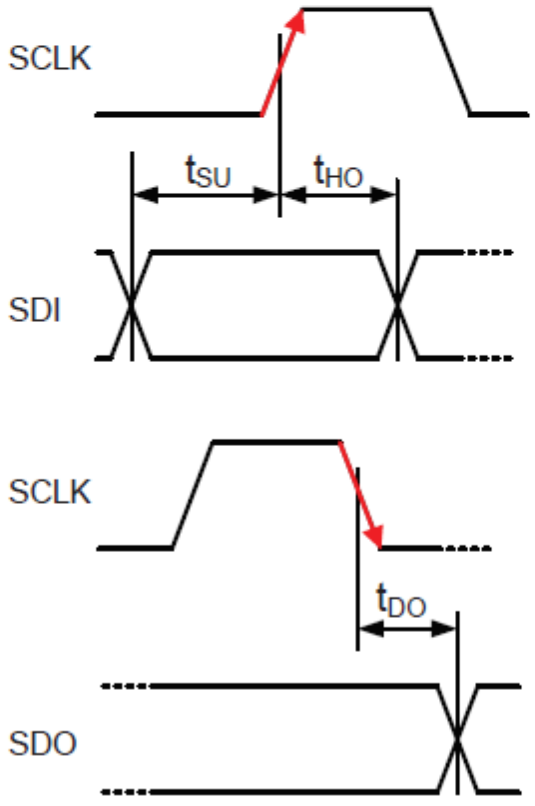


Figure 72: Setup and hold timing illustration

Figure 16 – SPI critical edges

SPI modes

CPHA (clock phase) = defines which edge data is latched on, a 0 representing the first edge and a 1 representing the second edge

CPOL (clock polarity) = defines whether the clock idles high or low in between SPI frames. CPOL = 0 idles low, CPOL = 1 idles high.

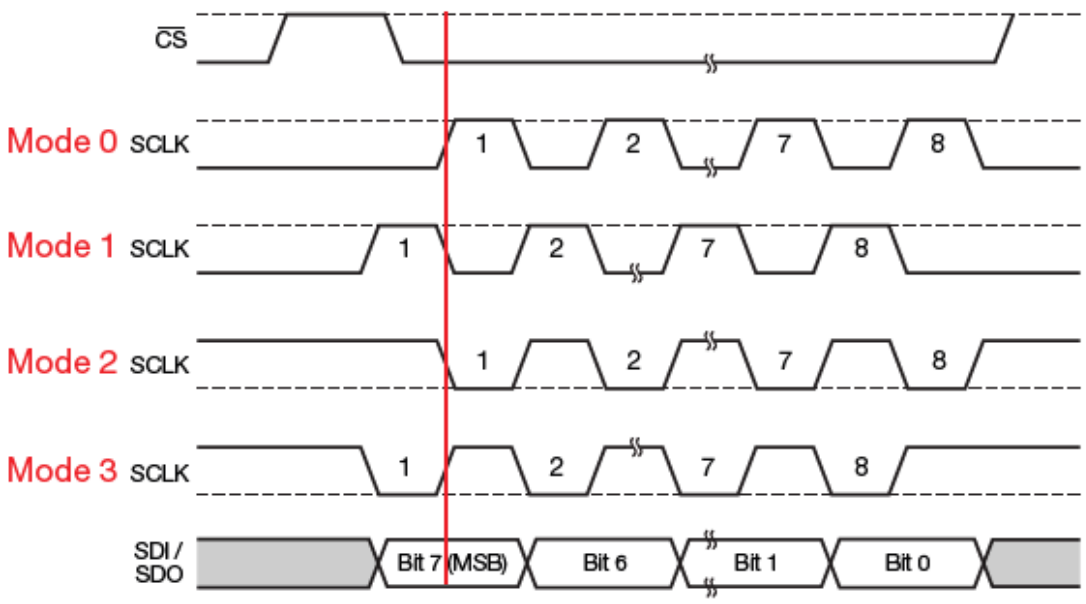


Figure 73: SPI modes of operation

Mode	CPOL	CPHA	Critical edge	Clock phase
0	0	0	Rising edge	Idles low
1	0	1	Falling edge	Idles low
2	1	0	Rising edge	Idles high
3	1	1	Falling edge	Idles high

Figure 17 – SPI Modes

I²C communication

START = initiated by the master pulling SDA low while SCL is high

STOP = initiated by the master releasing the SDA pin high while SCL is high

ACK (acknowledge) = each transfer in I²C is a single byte or 8-bits, with one SCL pulse per bit. The 9th pulse in each exchange is reserved for an acknowledgement signal from the slave, or an ACK signal. The ACK signal indicates that the previous transfer was successful.

Example I²C write sequence:

1. The master pulls down SDA to generate a START condition
2. The first bit is set up and the master pulls down and releases SCL to clock data into the DAC
3. On the 9th bit the master does not pull down SDA. If the slave pulls down SDA the 8-bit transaction is acknowledged.
4. The completed transaction in binary is 11001101

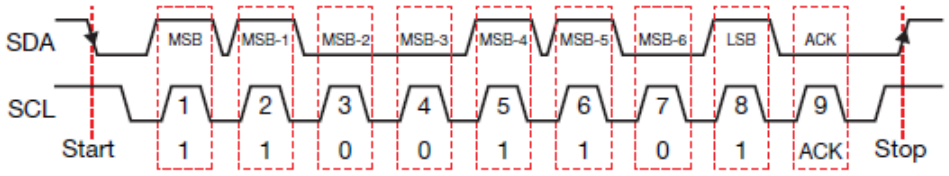


Figure 76: Complete I²C transaction

For valid data transfer:

- SDA must remain stable the entire time that SCL is high for a bit transfer to be valid
- SDA is only allowed to transition in between SCL pulses when SCL is low
- Instances where SDA changes while SCL is high are interpreted as START, RESTART, or STOP conditions

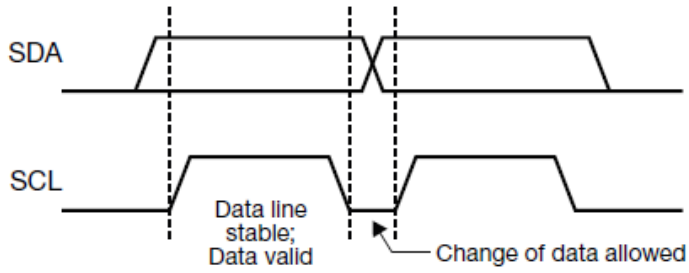


Figure 77: I²C data transfer

Figure 18 – I2C communication

I²C addressing

- Typical addressing in I²C is 7-bit addressing with an additional bit for read or write indication
- Each device on the I²C bus must have a unique address
- Duplicate addresses will result in communication errors
- Some devices may have pin programmable I²C addresses

Address byte

MSB						LSB	R/$\overline{W}$
1	0	0	1	1	0	A0	1/0

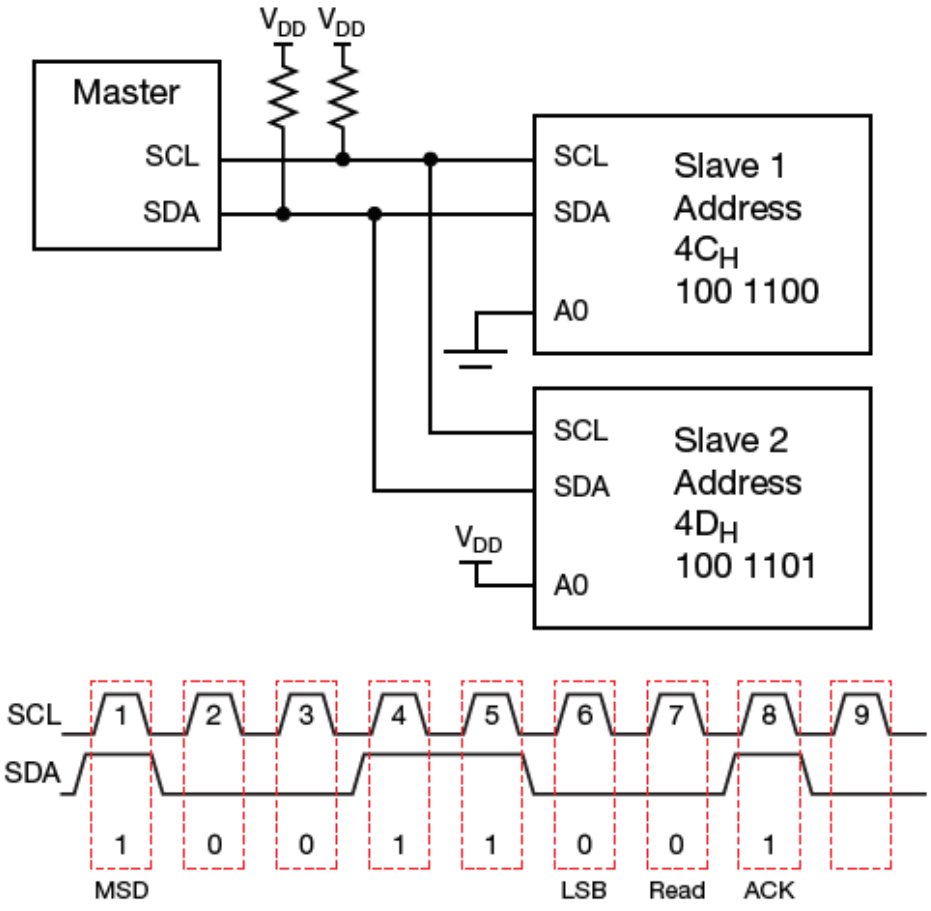


Figure 75: : I²C addressing

Figure 19 – I2C addressing

I²C interface circuitry and rise/fall timing

The figure below illustrates the internal structure for an I²C SCL or SDA pin. The transistor will turn on for logic low discharging C_b to logic low. The transistor will turn off for a logic high and the pull-up, R_{pull}, will charge C_b to a logic high.

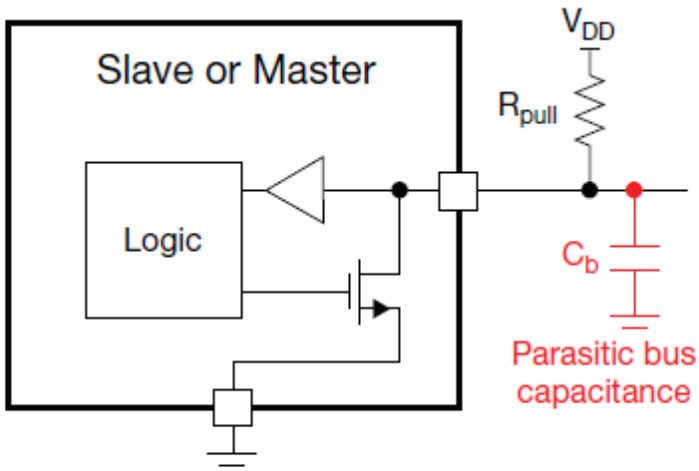


Figure 78: I²C data transfer

t_r (rise time) = the maximum amount of time for the signal to transition from logic low to logic high. Since I²C data is an open drain signal, rise time is set by the RC time constant of the pull-up resistance and the bus capacitance.

t_f (fall time) = the maximum amount of time for the signal to transition from logic high to logic low

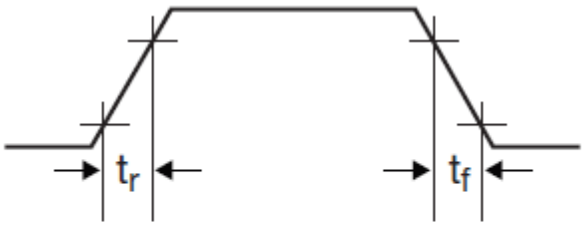


Figure 79: I²C rise and fall timing

Figure 20 – I2C Time Constants and Parasitic Bus Capacitance

CMOS switch construction

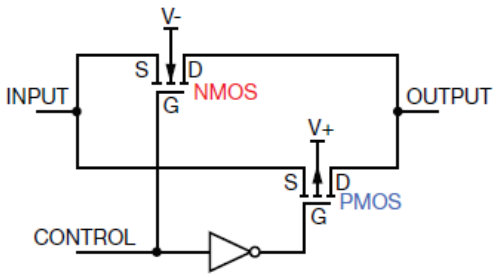


Figure 97: Typical CMOS switch construction

Typical CMOS switch elements

- Parallel combination of N channel and P channel FET
- Control signal that controls the state of the switch

ON-resistance (R_{ON})

Resistance between sources to drain terminal when switch is closed

- PMOS conducts for positive input voltages
- NMOS conducts for negative input voltages
- Combined RON is lower than individual resistance of the NMOS or PMOS that form the switch

Varies with

- MUX input voltage
- Operating ambient temperature

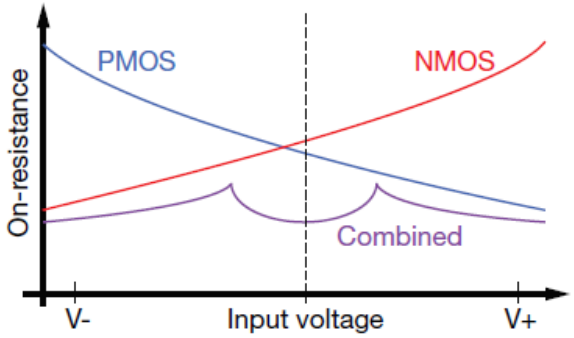


Figure 98: Typical MUX ON-resistance curve vs input voltage

Design tips

- Use high impedance stage between MUX output and signal conditioning
- Use a multiplexer with lower R_{ON} (at the expense of other design trade-offs)

Figure 21 – Multiplexer On Resistance Design

R_{ON} flatness

Difference between maximum and minimum ON-resistance over a specified input range

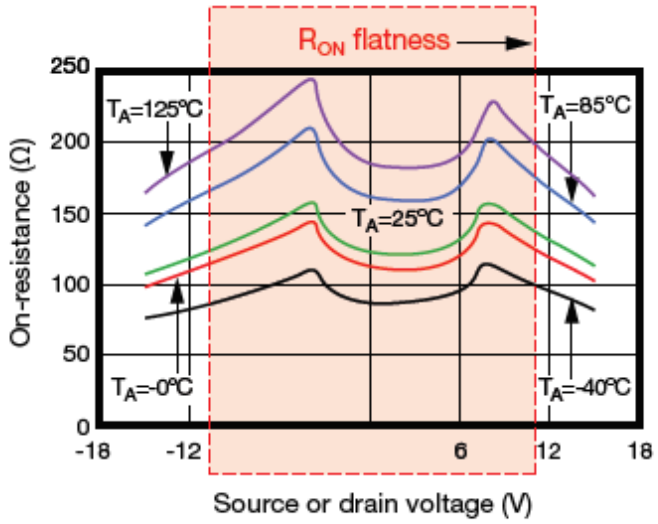


Figure 99: R_{ON} flatness illustration

Effective op amp gain including MUX R_{ON}

This section shows the impact of R_{ON} flatness on gain error and non-linearity. Figure 99 shows the circuit and Figure 100 shows the measured results.

$$AG = \frac{V_{OUT}}{V_{IN}} = \frac{-RF}{(R1+R_{ON})}$$

(215) Effective gain for op amp with MUX

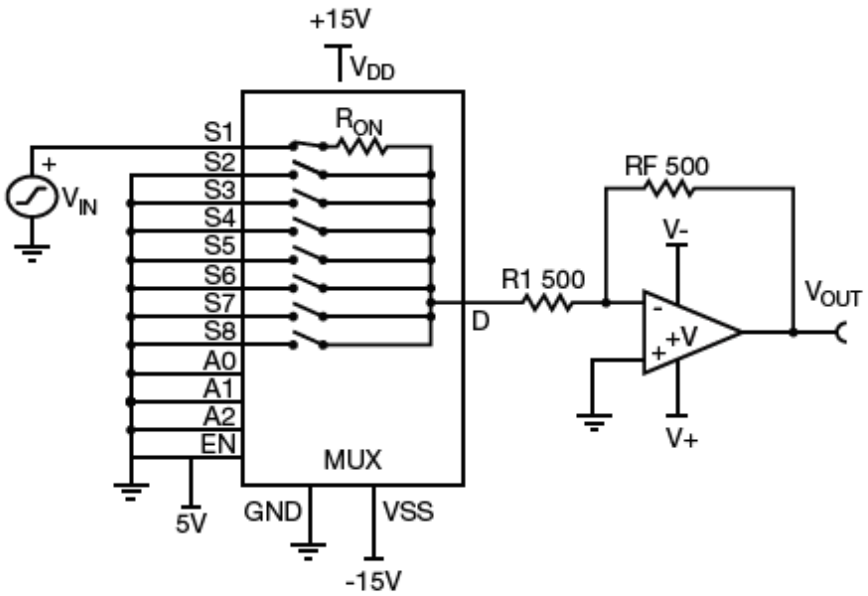


Figure 100: Front-end MUX followed by an inverting amplifier stage

Figure 22 – Multiplexer On Resistance Impacts on Error

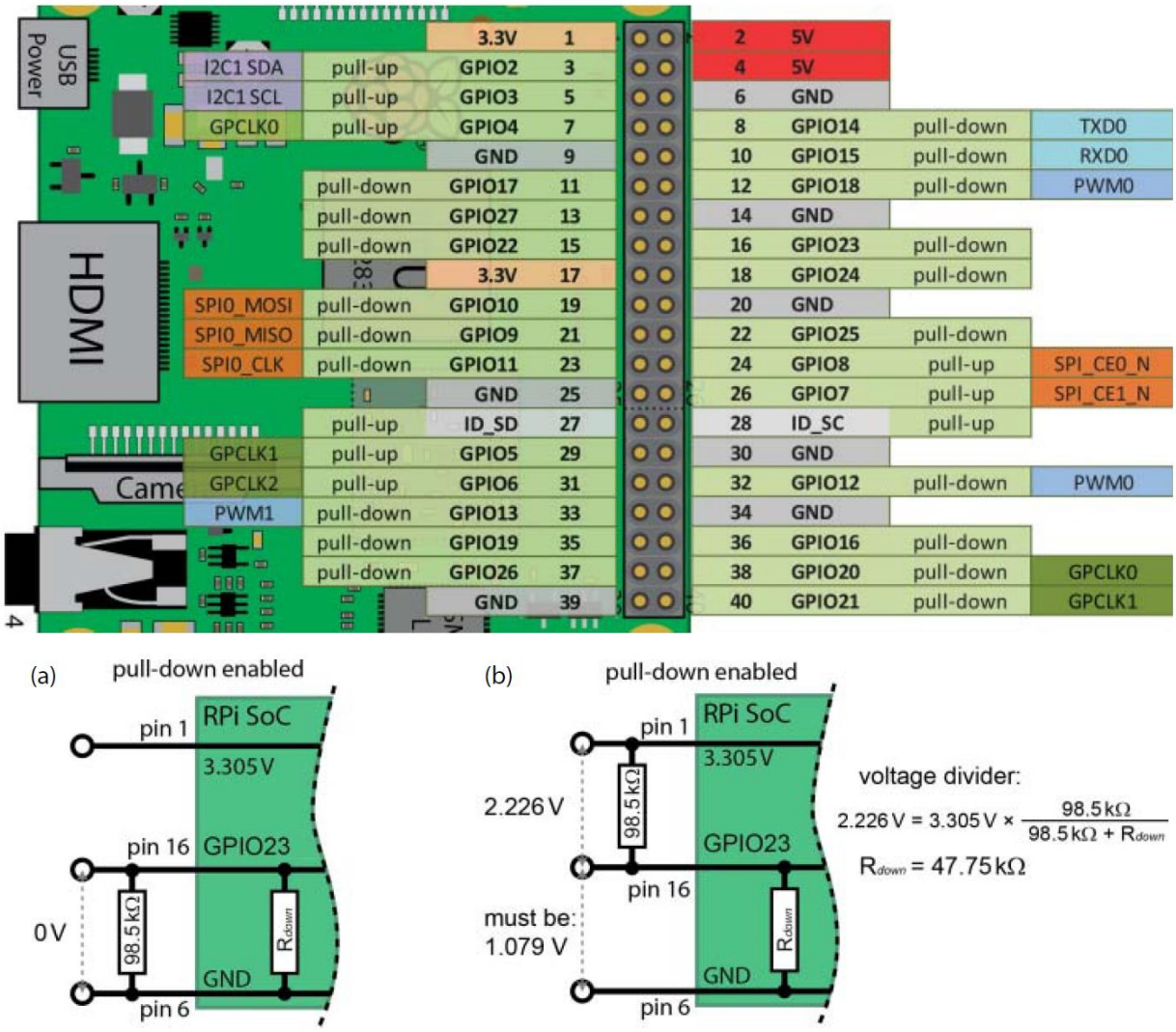


Figure 6-6: Internal pull-down resistor value determination, using a 100 kΩ resistor connected (a) from the GPIO pin to GND, and (b) from the GPIO pin to the 3.3 V supply

Figure 23 – Raspberry Pi 2/3 GPIO and pull-up/pull-down pinouts

	ALT5	ALT4	ALT3	ALT2	ALT1	ALT0	WPi	Pull	Mode	Pin Numbers		Mode	Pull	WPi	ALT0	ALT1	ALT2	ALT3	ALT4	ALT5	
RPi A, B, A+, B+, 2/3	50mA maximum on 3.3V supply								3.3V	1	2	5V	max current draw ~300mA (cover when not in use)								
	(note 2)			reserved	SA3	I2C1 SDA	8	up	GPIO2	3	4	5V	max current draw ~300mA (cover when not in use)								
	(note 2)			reserved	SA2	I2C1 SCL	9	up	GPIO3	5	6	GND									
	ARM_TDI			reserved	SA1	GPCLK0	7	up	GPIO4	7	8	GPIO14	down	15	TXD0	SD6				TXD1	
									GND	9	10	GPIO15	down	16	RXD0	SD7				RXD1	
	RTS1	SPI1_CE1_N	RTS0	reserved	SD9	reserved	0	down	GPIO17	11	12	GPIO18	down	1	PCM_CLK	SD10		BSCSL_SDA/MOSI	SPI1_CE0_N	PWM0	
		ARM_TMS	SD1_DAT3	reserved	reserved	reserved	2	down	GPIO27	13	14	GND									
		ARM_TRST	SD1_CLK	reserved	SD14	reserved	3	down	GPIO22	15	16	GPIO23	down	4	reserved	SD15	reserved	SD1_CMD	ARM_RTCK		
	50mA maximum on 3.3V supply								3.3V	17	18	GPIO24	down	5	reserved	SD16	reserved	SD1_DAT0	ARM_TDO		
				reserved	SD2	SPI0_MOSI	12	down	GPIO10	19	20	GND									
				reserved	SD1	SPI0_MISO	13	down	GPIO9	21	22	GPIO25	down	6	reserved	SD17	reserved	SD1_DAT1	ARM_TCK		
				reserved	SD3	SPI0_CLK	14	down	GPIO11	23	24	GPIO8	up	10	SPI_CE0_N	SD0	reserved				
									GND	25	26	GPIO7	up	11	SPI_CE1_N	SWE_N / SRW_N	reserved				
RPi A+, B+, 2/3 Only	Do not use (GPIO0) -- see note 3			reserved	SA5	SDA0	30	up	ID_SD	27	28	ID_SC	up	31	SCL0	SA4	reserved	Do not use (GPIO1) -- see note 3			
	ARM_TDO			reserved	SA0	GPCLK1	21	up	GPIO5	29	30	GND									
	ARM_RTCK			reserved	SOE_N/SE	GPCLK2	22	up	GPIO6	31	32	GPIO12	down	26	PWM0	SD4	reserved			ARM_TMS	
	ARM_TCK			reserved	SD5	PWM1	23	down	GPIO13	33	34	GND									
				reserved	SD11	PCM_FS	24	down	GPIO19	35	36	GPIO16	down	27	reserved	SD8	reserved	CTS0	SPI1_CE2_N	CTS1	
		ARM_TDI	SD1_DAT2	reserved	reserved	reserved	25	down	GPIO26	37	38	GPIO20	down	28	PCM_DIN	SD12	reserved	BSCSL / MISO	SPI1_MOSI	GPCLK0	
									GND	39	40	GPIO21	down	29	PCM_DOUT	SD13	reserved	BSCSL / CE_N	SPI1_SCLK	GPCLK1	

Type	Linux DT	Description
GPIO	sysfs	general purpose input/output
SPI	spi	serial peripheral interface
I2C	i2c0/i2c1	I^2C Bus
UART	uart0	UART
PWM	pwm	Pulse Width Modulation
GPCLK	gp_clk	General purpose clock (GPCLK1 is reserved)
PCM	pcm	PCM audio
SA	smi	Secondary Memory Interface

Note 1: The data in this table was created from the www.eLinux.org web pages, system information, and datasheets where available.

Note 2: On early models of the RPi, Pin 3 is GPIO0 and Pin 5 is GPIO1. Also, these pins have permanent on-board 1.8 KΩ pull-up resistors attached (for the I^2C bus).

Note 3: ID_SD and ID_SC pins are reserved for the ID EEPROM (for different HATs). This is an I^2C interface that is probed at boot time in order to detect attached boards. This allows Linux to load the correct drivers for a HAT. See Chapter 8.

Register	Offset	MSB	32-bits for each register	LSB

Offset is from the virtual address 0x3F000000 on the RPi 2/3, and 0x20000000 on all other RPi models.

GPFSELn	The Function Select mode for each GPIO (3 bits for each GPIO and 54 GPIOs in total)											Read/Write
Bits		[31-30]	[29-27]	[26-24]	[23-21]	[20-18]	[17-15]	[14-12]	[11-9]	[8-6]	[5-3]	[2-0]
GPFSEL0	0000	X	FSEL9	FSEL8	FSEL7	FSEL6	FSEL5	FSEL4	FSEL3	FSEL2	FSEL1	FSEL0
GPFSEL1	0004	X	FSEL19	FSEL18	FSEL17	FSEL16	FSEL15	FSEL14	FSEL13	FSEL12	FSEL11	FSEL10
GPFSEL2	0008	X	FSEL29	FSEL28	FSEL27	FSEL26	FSEL25	FSEL24	FSEL23	FSEL22	FSEL21	FSEL20
GPFSEL3	000C	X	FSEL39	FSEL38	FSEL37	FSEL36	FSEL35	FSEL34	FSEL33	FSEL32	FSEL31	FSEL30
GPFSEL4	0010	X	FSEL49	FSEL48	FSEL47	FSEL46	FSEL45	FSEL54	FSEL43	FSEL42	FSEL41	FSEL40
GPFSEL5	0014	[32-12] X							FSEL53	FSEL52	FSEL51	FSEL50

GPSETn	The Output Set register - use this to set a GPIO high (1 bit for each of the 54 GPIOs)	Read/Write
GPSET0	001C	[31-0] mapped to GPIO31 to GPIO0 (1 = set GPIO)(0 = no effect)
GPSET1	0020	[31-22] X [21-0] mapped to GPIO53 to GPIO32 (1 = set GPIO)(0 = no effect)

GPCLRn	The Output Clear register - use this to set a GPIO low (1 bit for each of the 54 GPIOs)	Read/Write
GPCLR0	0028	[31-0] mapped to GPIO31 to GPIO0 (1 = clear GPIO)(0 = no effect)
GPCLR1	002C	[31-22] X [21-0] mapped to GPIO53 to GPIO32 (1 = clear GPIO)(0 = no effect)

GPLVLn	The Level Read register - use this to read the value of a GPIO (1 bit for each of the 54 GPIOs)	Read Only
GPLVL0	0034	[31-0] mapped to GPIO31 to GPIO0 (1 = level is high)(0 = level is low)
GPLVL1	0038	[31-22] X [21-0] mapped to GPIO53 to GPIO32 (1 = level is high)(0 = level is low)

GPPUD	The Pull-up/Pull-down Enable register - use this to define the configuration	Read/Write
GPPUD	0094	[31-2] X [1-0]

GPPUDCLKn	The Pull-up/Pull-down Enable Clock register - use this to apply GPPUD to a particular GPIO	Read/Write
GPPUDCLK0	0098	[31-0] mapped to GPIO31 to GPIO0 (1 = assert clock)(0 = no effect)
GPPUDCLK1	009C	[31-22] X [21-0] mapped to GPIO53 to GPIO32 (1 = assert clock)(0 = no effect)

Function	GPFSELn
Bits	Meaning
000	input
001	output
100	ALT0
101	ALT1
110	ALT2
111	ALT3
011	ALT4
010	ALT5

Example: Set GPIO17 to be an output and set it high.
Solution: Write bits 001 to FSEL17, which is bits 21, 22, and 23 of the GPFSEL1 register to set the pin up as an output.
Then write a 1 to bit 17 of the GPSET0 register to set the output high.

Function	PUD
Bits	Meaning
00	no pull-up/down
01	pull-down
10	pull-up
11	X

GPFSEL1

[31]	[30]	...	[24]	[23]	[22]	[21]	[20]	...	[0]
X	X	...	?	0	0	1	?	...	?

FSEL17

Do not change the "?" bits as this will affect other GPIO select modes!

Figure 24 – Raspberry Pi 2/3 GPIO memory addressing information

Name	Function	See section
SDA0	BSC[5] master 0 data line	BSC
SCL0	BSC master 0 clock line	BSC
SDA1	BSC master 1 data line	BSC
SCL1	BSC master 1 clock line	BSC
GPCLK0	General purpose Clock 0	<TBD>
GPCLK1	General purpose Clock 1	<TBD>
GPCLK2	General purpose Clock 2	<TBD>
SPI0_CE1_N	SPI0 Chip select 1	SPI
SPI0_CE0_N	SPI0 Chip select 0	SPI
SPI0_MISO	SPI0 MISO	SPI
SPI0_MOSI	SPI0 MOSI	SPI
SPI0_SCLK	SPI0 Serial clock	SPI
PWMx	Pulse Width Modulator 0..1	Pulse Width Modulator
TXD0	UART 0 Transmit Data	UART
RXD0	UART 0 Receive Data	UART
CTS0	UART 0 Clear To Send	UART
RTS0	UART 0 Request To Send	UART
PCM_CLK	PCM clock	PCM Audio
PCM_FS	PCM Frame Sync	PCM Audio
PCM_DIN	PCM Data in	PCM Audio
PCM_DOUT	PCM data out	PCM Audio
SAx	Secondary mem Address bus	Secondary Memory Interface
SOE_N / SE	Secondary mem. Controls	Secondary Memory Interface
SWE_N / SRW_N	Secondary mem. Controls	Secondary Memory Interface
SDx	Secondary mem. data bus	Secondary Memory Interface
BSCSL SDA / MOSI	BSC slave Data, SPI salve MOSI	BSC ISP slave
BSCSL SCL / SCLK	BSC slave Clock, SPI slave clock	BSC ISP slave
BSCSL - / MISO	BSC <not used>,SPI MISO	BSC ISP slave
BSCSL - / CE_N	BSC <not used>, SPI CSn	BSC ISP slave
Name	**Function**	**See section**
SPI1_CEx_N	SPI1 Chip select 0-2	Auxiliary I/O
SPI1_MISO	SPI1 MISO	Auxiliary I/O
SPI1_MOSI	SPI1 MOSI	Auxiliary I/O
SPI1_SCLK	SPI1 Serial clock	Auxiliary I/O
TXD0	UART 1 Transmit Data	Auxiliary I/O
RXD0	UART 1 Receive Data	Auxiliary I/O
CTS0	UART 1 Clear To Send	Auxiliary I/O
RTS0	UART 1 Request To Send	Auxiliary I/O
SPI2_CEx_N	SPI2 Chip select 0-2	Auxiliary I/O
SPI2_MISO	SPI2 MISO	Auxiliary I/O
SPI2_MOSI	SPI2 MOSI	Auxiliary I/O
SPI2_SCLK	SPI2 Serial clock	Auxiliary I/O
ARM_TRST	ARM JTAG reset	<TBD>
ARM_RTCK	ARM JTAG return clock	<TBD>
ARM_TDO	ARM JTAG Data out	<TBD>
ARM_TCK	ARM JTAG Clock	<TBD>
ARM_TDI	ARM JTAG Data in	<TBD>
ARM_TMS	ARM JTAG Mode select	<TBD>

Figure 25 – BCM2835 peripherals function descriptions