OSU Mechanical Engineering

Smart Products Laboratory

Software III | Three

ME Course Number
Spring 2019

Table of Contents

List of Figures

1 Overview

This lab will work on the use memory manipulation, bitwise operations, and advanced functionality of C++.

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

- **Pointer:** variable whose value is the memory address of another function or variable:

```
int* Pointer;// empty pointer of type int
int x = 4;// variable of type int with a value of 4
Pointer = &x;// Address of x is accessed by &x, assign it to the Pointer
std::cout<<*Pointer;//access the value of x with a pointer by *Pointer
```

- **Address:** a binary based key that encodes the location of a register in the hardware.
- **Register:** is a row of physical elements that store data in terms of a 1 or 0. Each register usually has three capabilities
  - Reading the value of the data

```
int value_at_x_is = *Pointer;// reading the data with a pointer
```

  - Writing new data to the register

```
*Pointer = 5;//writing new data to x
```

  - Accessing the address of the register

```
std::cout<< &x;//printing the address of x
```

- **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

These concepts and specific raspberry pi GPIO information are further presented next, feel free to use this as a helpful reference when creating your code.

2.2 Simple Code Examples

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
// Demonstration of a pointer
//-----------------------------------
// say we have a variable x of type integer
   int x;
// then we create a pointer
   int* ptr;
// this pointer must be of the same type as x for
// it to be able to point to the address of x.
   ptr = &x;
// the '&' is the reference operator and provides
// a pointer the address of the variable x
// so that it can access the value of x.
   printf("Address of x is %p \n", (void*)&x);
   printf("Address of ptr is %p \n", (void*)ptr);
// Now we assign x a value
   x = 1000;
// We show that the value of x is the same as dereferencing the
// pointer.
   printf("Value of x is %d \n", x);
   printf("Value of ptr is %d \n",*ptr);
// Now we can change the value of x through the use of the ptr
   *ptr = 54;
   printf("Value of x is %d \n", x);
   printf("Value of ptr is %d \n",*ptr);
}
```

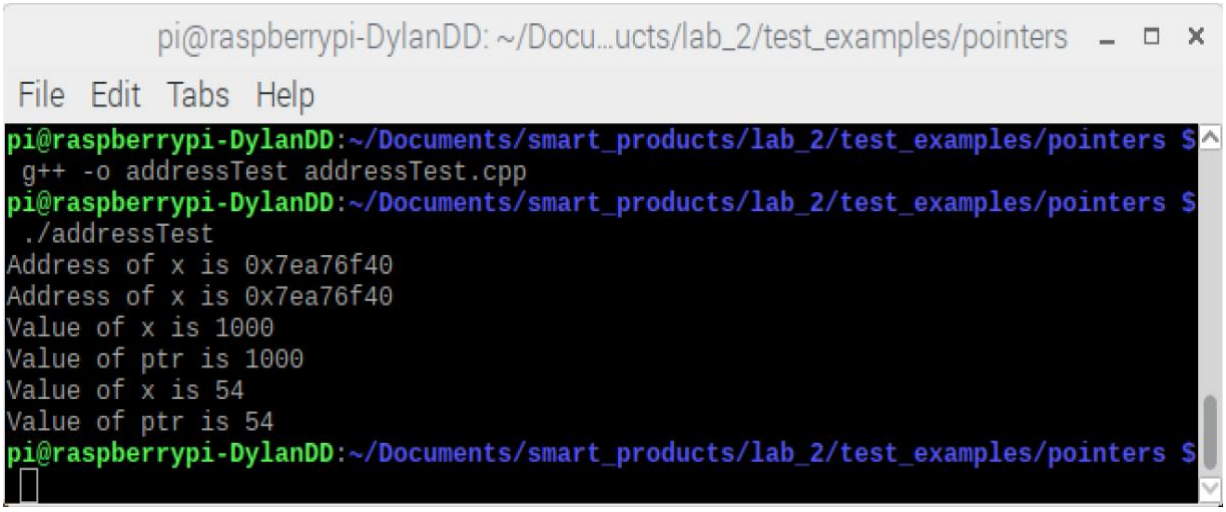


Figure 1 – Demonstration of a pointer in C++

```cpp
#include <iostream>
int main()
{
    char *base;//creating a pointer
    char x = 'A';//creating a variable
    base = &x;//assigning the variable's address to the base pointer

    int register_offset = 0x01;//creating an offset
    //creating a pointer to the register after the base
    char *register1 = base+register_offset;
    //creating a pointer to the register after the first register.
    char *register2 = register1+register_offset;

    *register1 = 'B';//assigning value to register 1
    *register2 = 'C';//assigning value to register 2
    std::cout<<"value at the base = " <<*base<<std::endl;
    base = base + 0x01; //increment the base address by 1;
    std::cout<<"value of the register1 = " <<*register1<<std::endl;
    std::cout<<"value at the incremented base = " <<*base<<std::endl;
    base = base + 0x01; //increment the pointers address by 1;
    std::cout<<"value of the register2 = " <<*register2<<std::endl;
    std::cout<<"value at the incremented base = " <<*base<<std::endl;
    return 0;
}
```

```
Example addressing and offsets
#Terminal print out
value at the base = A
value of the register1 = B
value at the incremented base = B
value of the register2 = C
value at the incremented base = C
```

Figure 2 – Demonstration of addressing and offsets in C++

```cpp
#include <iostream>
class Base {
    public:
   virtual void f()=0;
   int num() {return 1;}
};
class Derived : public Base {
    public:
    void f() override { // 'override' is required
        std::cout << "derived: "<< Base::num()<<std::endl;
    }
};
int main()
{
    Derived d;
    d.f();
    return 0;
}
```

Compiler:

```
derived: 1
```

Figure 3 – Demonstration of a virtual function in C++

2.3 GPIO manipulation

The general-purpose input output (GPIO) hardware on the raspberry pi allows you to control and read digital signals. You can access the individual pins by accessing various registers which control the different settings of the GPIO hardware. Selection registers (SR0-SR5) contain the pin modes for each of the 54 GPIO pins on the pi. If you only want to read the digital state of a pin, you would set its pin mode to INPUT. If you want to read and write the digital state of a pin, you would set it to OUTPUT. There also exists alternative modes, but we don't use them in this lab. In order to set the pin mode, you must write to the correct register for the specific pin, these registers are listed in Table 1. Since there are multiple pin modes, each one is specified with 3 bits. Instead of using one 32-bit register per pin, the designers of the pi have designed the hardware such that each selection register controls the pin mode for 10 pins. This concept is shown in Figure 5. Similarly, there are separate registers which allow you to read and write to the individual pins. Since there are 54 pins, and a register can only hold 32 bits, there are two registers for each of the read and write functions. The registers which read the digital state are ready only. There are four registers for writing the digital state of the pins, two for writing the state to HIGH and two for writing the digital state to LOW. In either case, if you want to change the state of a pin, you must write 0x1 to its corresponding bit, writing 0x0 won't do anything.

In the provided code, the base address of the GPIO hardware is held by the member pointer named mMemPtr. To access a register that is offset from the base use the function named `getPtr(GPIOregisters gpio_register)`as shown in Figure 4. See the appendix for further details on the GPIO registers.

Table 1 – GPIO register information and functionality

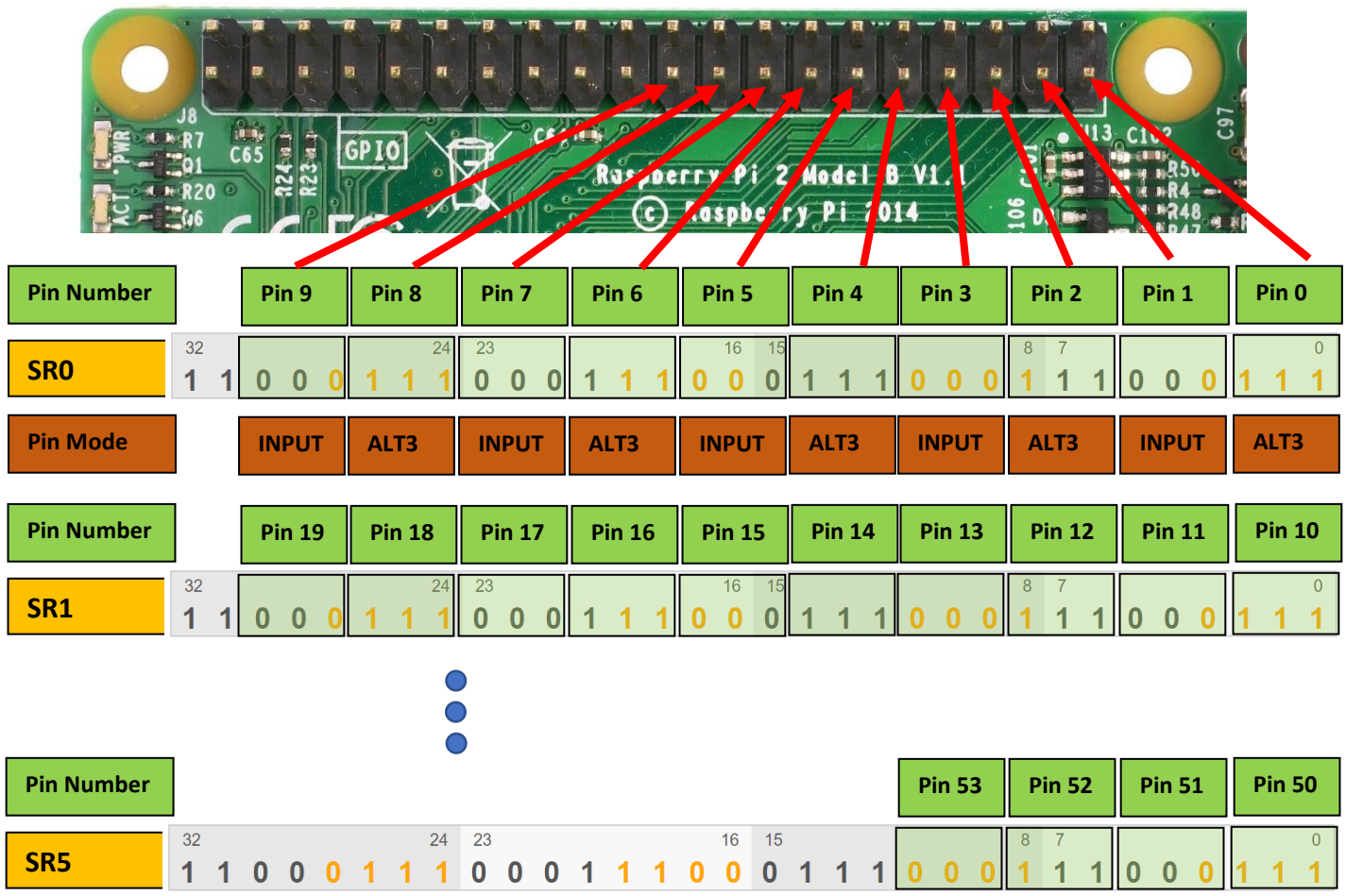
Register function (Range of Pins effected)	Short Name	Offset	Bits of Info/ pin	Writing 1 to a bit will	Writing 0 to a bit will
Pin Mode Selection Register 0 (0-9)	SR0	0x0000	3	-	-
Pin Mode Selection Register 1 (10-19)	SR1	0x0004	3	-	-
Pin Mode Selection Register 2 (20-29)	SR2	0x0008	3	-	-
Pin Mode Selection Register 3 (30-39)	SR3	0x000C	3	-	-
Pin Mode Selection Register 4 (40-49)	SR4	0x0010	3	-	-
Pin Mode Selection Register 5 (50-53)	SR5	0x0014	3	-	-
Read Pin State Register (0-31)	Read0	0x0034	1	Read-only	Read-only
Read Pin State Register (32-53)	Read1	0x0038	1	Read-only	Read-only
Turn Pin State to High Register (0-31)	WriteHigh0	0x001C	1	Turn pin to high	Do nothing
Turn Pin State to High Register (32-53)	WriteHigh1	0x0020	1	Turn pin to high	Do nothing
Turn Pin State to Low Register (0-31)	WriteLow0	0x0028	1	Turn pin to low	Do nothing
Turn Pin State to Low Register (32-53)	WriteLow1	0x002C	1	Turn pin to low	Do nothing

```
/*
    The offsets from GPIO Base for different memory registers ne
eded
    to:
        change the pin modes (SRx),
        read the digital state of a pin (Readx),
        change the digital output of a pin to 1 (WriteHighx),
        change the digital output of a pin to 0 (WriteLowx)
*/
enum class GPIOregisters : std::uint32_t {
    SR0 = 0x0000,
    SR1 = 0x0004,
    SR2 = 0x0008,
    SR3 = 0x000C,
    SR4 = 0x0010,
    SR5 = 0x0014,
    Read0 = 0x0034,
    Read1 = 0x0038,
    WriteHigh0 = 0x001C,
    WriteHigh1 = 0x0020,
    WriteLow0 = 0x0028,
    WriteLow1 = 0x002C
};
```

```
void* getPtr(GPIOregisters gpio_register)
{
    return (mMemPtr +  static_cast<off_t>(gpio_register));
}
```

Name	Address = base +*offset*	Value = Binary data with a maximum amount of 32 individual bits
SR0	mMemPtr + *0x0000*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
SR1	mMemPtr + *0x0004*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
SR2	mMemPtr + *0x0008*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
SR3	mMemPtr + *0x000C*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
SR4	mMemPtr + *0x0010*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
WLow0	mMemPtr + *0x0028*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1
WLow0	mMemPtr + *0x002C*	32 24 23 16 15 8 7 0 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1 0 0 0 1 1 1

Figure 4 – RPI GPIO Registers



Pin Mode	Bits	Hex
INPUT	000	0x0
OUTPUT	001	0x1
ALT0	100	0x4
ALT1	101	0x5
ALT2	110	0x6
ALT3	111	0x7
ALT4	011	0x3
ALT5	010	0x2

```
1.          // All the different PinModes that can be set for GPIO
2.          enum class PinModes : std::uint32_t {
3.              INPUT = 0x0,
4.              OUTPUT = 0x1,
5.              ALT0 = 0x4,
6.              ALT1 = 0x5,
7.              ALT2 = 0x6,
8.              ALT3 = 0x7,
9.              ALT4 = 0x3,
10.             ALT5 = 0x2
11.         };
```

Figure 5 – Pin mode in the selection registers

Example: Setting the pin mode for pin 6 to OUTPUT

1. First read the correct memory register which holds the pin modes for the first 10 pins
2. Move the desired pin mode for pin 6 over to the 18th bit (the first bit belonging to pin 6)
3. Remove bits 18, 19, 20 in the current pin mode configuration.
4. Merge the desired pin mode for pin 6 with the modified pin configuration
5. Write the merged configuration back to the register.

```
1.  std::uint32_t pinmode_now = *(reinterpret_cast<std::uint32_t *>(pin_setting.PtrPinMode));
2.  std::uint32_t bitmask = 0x1 << 18
3.  std::uint32_t filtered_config = pinmode_now & (~(0x7<< 18));
4.  std::uint32_t new_pinmode = filtered_config| bitmask;
5.  *(reinterpret_cast<std::uint32_t *>(pin_setting.PtrPinMode) )= new_pinmode;
```

Operator	Symbol	Form	Operation
left shift	<<	x << y	all bits in x shifted left y bits
right shift	>>	x >> y	all bits in x shifted right y bits
bitwise NOT	~	~x	all bits in x flipped
bitwise AND	&	x & y	each bit in x AND each bit in y
bitwise OR	|	x | y	each bit in x OR each bit in y
bitwise XOR	^	x ^ y	each bit in x XOR each bit in y

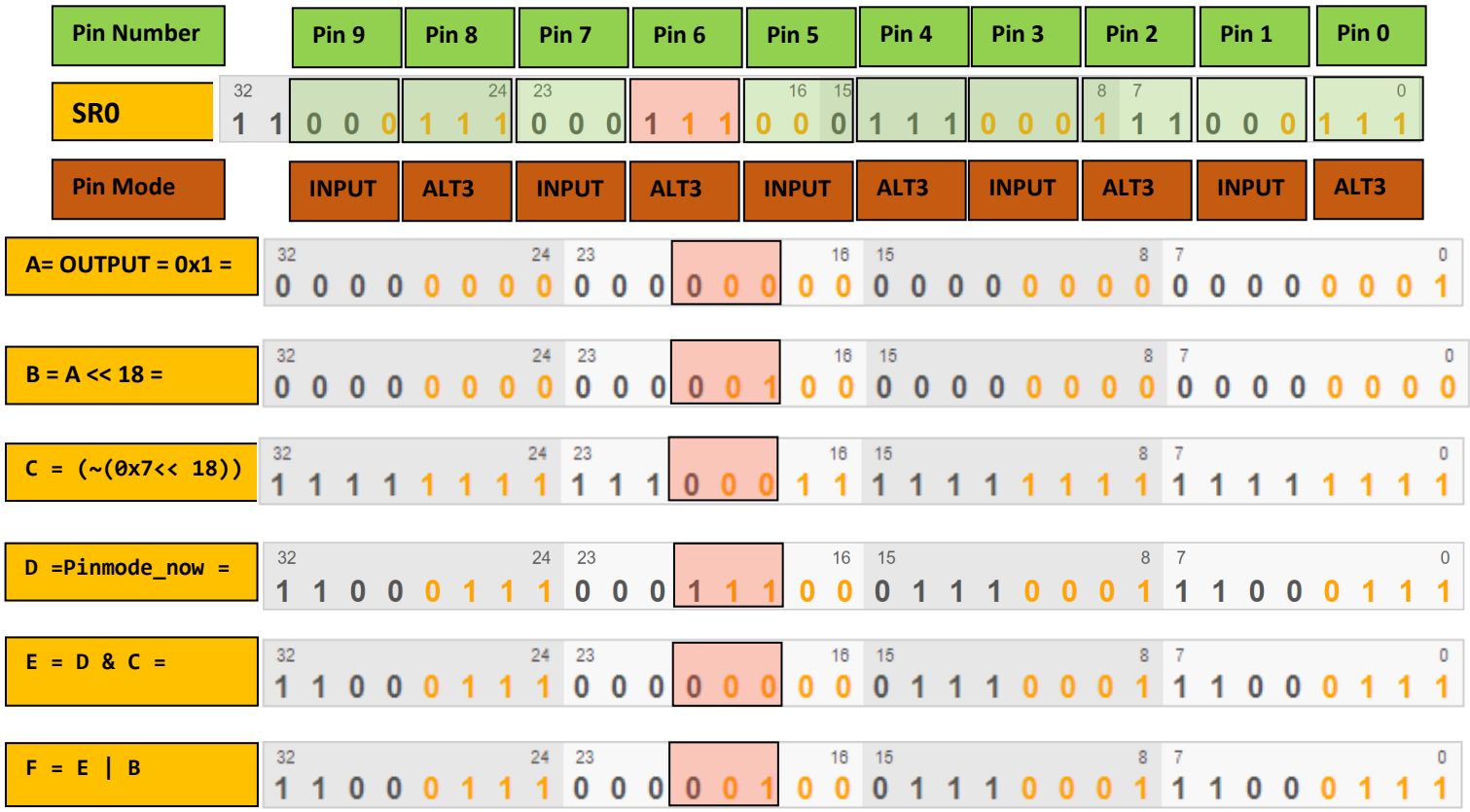


Figure 6 – Changing Pin Mode

```
1.  //when reading from the register, use this notation
2.  std::uint32_t current_pin_mode_configuration =
3.          *(reinterpret_cast<std::uint32_t *>(pin_setting.PtrPinMode));
4.
5.  //when writing to the registers, use this notation
6.  *(reinterpret_cast<std::uint32_t *>(pin_setting.PtrPinMode) )=
7.           merged_pin_mode_configuration;
```

```
1.  struct PinSettings {
2.      /*
3.          the position of the first bit in the memory register
4.          which corresponds to the value of the pin mode for
5.          the given pin number
6.          - (ranges from 0 to 31) -
7.      */
8.      int PinPositionPM;
9.      /*
10.     the position of the first bit in the memory register
11.     which corresponds to the state (HIGH OR LOW) of a given pin
12.     which will be either written to or read from
13.     (ranges from 0 to 31)
14.     */
15.     int PinPositionRW;
16.     /*
17.         Pointer to the pin mode register
18.         use reinterpret_cast<std::uint32_t *>(PtrPinMode) later on to read and write
19.     */
20.     void* PtrPinMode;
21.     /*
22.         Pointer to the digital write register
23.         use  reinterpret_cast<std::uint32_t *>(PtrWrite) later on to read and write
24.     */
25.     void* PtrWrite;
26.     /*
27.         Pointer to the digital read register
28.         use  reinterpret_cast<std::uint32_t *>(PtrRead) later on to read
29.     */
30.     void* PtrRead;
31. };
```

Figure 7 – Conventions in the code

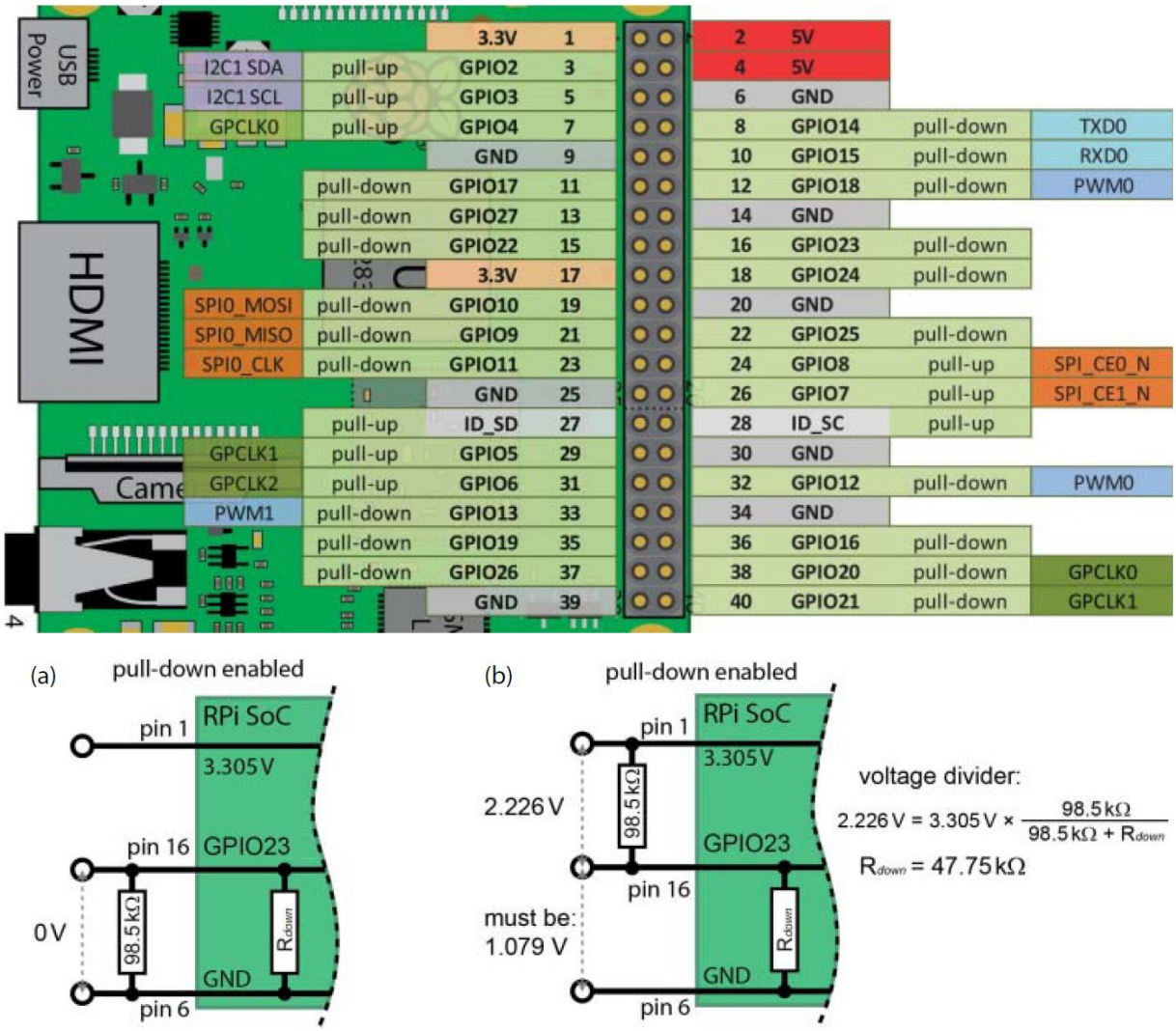


Figure 6-6: Internal pull-down resistor value determination, using a 100 kΩ resistor connected (a) from the GPIO pin to GND, and (b) from the GPIO pin to the 3.3 V supply

Figure 8 – Raspberry Pi 2/3 GPIO and pull-up/pull-down pinouts

3 References & Resources

3.1 References

The following references may be helpful to complete the lab.

1. https://en.cppreference.com/w/cpp/language/types
2. http://calc.50x.eu/
3. https://en.wikipedia.org/wiki/Processor_register
4. https://www.learncpp.com/cpp-tutorial/38-bitwise-operators/

4 Pre-lab

This section should be worked on prior to arriving in lab.

4.1 Items needed to complete lab two

Make sure you read this entire lab.

5 Laboratory

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

5.1 Requirements

Bring your raspberry pi (RPI) so that you can develop your code and test your GPIO class.

5.2 Procedure

1. Download the header file that defines the abstract GPIO class.
2. Make sure you enable GPIO access in the raspberry pi interface settings.
3. When creating your GPIO class, create a header file (.h) for your declarations and an implementation file (.cpp) for your definitions.
4. To include the provided abstract GPIO class, all that is needed is the header file *SPGPIO.h* which you will import by including it at the top of your class declaration.
5. Define your main function in *labthree.cpp*.
6. You can test out your code using my example main function and the terminal printout of its results.
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_lab03_Lastname_dotnumber.zip
    b. Inside the zip files should be only the following
        i. Three programming files which are (what you call them doesn't matter, just make sure I can tell what each file does)
            1. *labthree.cpp* – main implementation file for lab three
            2. *MyGPIO.h* – header file you created (call it whatever you like)
            3. *MyGPIO.cpp* – implementation file you created

ii. One report in PDF format which will contain only the following in 3 pages or less,
    1. Date, Name, Class, and Lab number(s) at the top.
    2. Section one:
        a. Describe the algorithms you developed for manipulating the different register's values for reading and writing the state of the pins as well as for the setting the pin mode.
        b. Did you take into consideration the efficiency of your algorithms when developing your code? If so, briefly explain how you optimized your code. If not, then describe what you would do differently to optimize your algorithms if you have a chance to redo them.
        c. Very, very briefly describe any function you created other than the ones required.
        d. Describe how you used bitwise operators to solve the exercises, briefly.
        e. How did you incorporate error checking in your software, if at all?
    3. Section three
        a. What information do you think would have helped you better understand and complete the exercises in this lab?
        b. Include a picture or two of the terminal printouts from the main implementation file created in this lab, displaying your results.

5.3 Exercises

Create a GPIO class which is derived from the base GPIO class (I called my class **MyGPIO,** name yours whatever you'd like) which satisfies the following requirements

1. The class is derived from the sp::GPIO abstract class.
2. The class overrides the following functions
    a. `int pinMode(int pin_number, PinModes p_mode)`
        i. This function's inputs are the pin number and the pin mode you wish to set it to. The function outputs a -1 if the pin mode failed to be changed and 1 otherwise.
        ii. The **PinModes** type is an enumerator class structure as shown in Figure 5 and should be casted as `static_cast<std::uint32_t>(p_mode).`
        iii. Generalize the process shown in Figure 5 to be able to set any pin to any pin mode.
    b. `int digitalRead(int pin_number)`
        i. This function's input is the desired pin number from which you wish to read its digital state. The function outputs 1 if the digital state is HIGH, 0 is the digital state is LOW, -1 if the function failed to read the digital state.
    c. `int digitalWrite(int pin_number, DigitalOut out_value)`
        i. This function's inputs are the pin number and the state that you wish to change it to. The function outputs a -1 if the pin state failed to be changed and 1 otherwise.
        ii. Make sure you incorporate a way to see if you have set your pin mode to OUTPUT.
        iii. The **DigitalOut** type is an enumerator class structure whose options are either HIGH or LOW.
    d. `PinSettings getPinSettings(int pinNumber, DigitalOut outValue = DigitalOut::LOW)`
        i. This function's inputs are the pin number and the state that you wish to change it to, if at all. The function outputs a struct type call PinSettings, which is displayed in Figure 7.
        ii. This function is **OPTIONAL** and may be used to optimize your code. However, how or if it is used is left up to the student.

You may demonstrate that your code is working using the main file code as shown in Figure 10. 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	/ 100	/ 30	/ 10	/ 10	/ 150
Report	-	/ 50	-	-	/ 50
Total	**/ 100**	**/ 80**	**/10**	**/10**	**/200**

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 three
```

```
1.  #include "SPGPIO.h"
2.  #include<iostream>
3.
4.  int main()
5.  {
6.      // starting GPIO session
7.      sp::MyGPIO gpio; // enables the constructor
8.      int pin_num = 27; // pin to control
9.      std::cout<< "Toggling Pin Mode ...";
10.     gpio.pinMode(pin_num, sp::GPIO::PinModes::INPUT);
11.     gpio.pinMode(pin_num, sp::GPIO::PinModes::OUTPUT);// set the pin mode
12.     std::cout << "toggle complete" << std::endl;
13.     std::cout<< "Writing Low : "<< std::endl;
14.     gpio.digitalWrite(pin_num, sp::GPIO::DigitalOut::LOW); //  set the pin to low
15.     int pin_val = gpio.digitalRead(pin_num); // read the value of the pin
16.     std::cout << "Value of pin " << pin_num << " is  " << pin_val << std::endl;
17.     std::cout<< "Writing High : "<< std::endl;
18.     gpio.digitalWrite(pin_num, sp::GPIO::DigitalOut::HIGH); //write high to pin 27
19.     pin_val = gpio.digitalRead(pin_num);// read the pin
20.     std::cout << "Value of pin " << pin_num << " is  " << pin_val << std::endl;
21.     return 0;
22. }
```

```
Example solutions to lab three
#Terminal print out
Toggling Pin Mode ...toggle complete
Writing Low :
Value of pin 27 is  0
Writing High :
Value of pin 27 is  1
```

Figure 9 – Example main file "labthree.cpp" for executing the GPIO class operations

Appendix

	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 10 – 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 11 – BCM2835 peripherals function descriptions