Guide Book for Programming on Internet of Things with Raspberry Pi

Zhiyi Huang
Department of Computer Science
University of Otago

July 12, 2018

Contents

Chapter 1

Handling Raspberry Pi hardware

Before you power on a Raspberry Pi by plugging in the cable of the power adapter, plug in other devices like keyboard, mouse, and monitor first.

When you want to shutdown a Raspberry Pi, make sure to click on the raspberry icon on the top left corner and find and click `Shutdown` from the drop-down menu. Then wait until the monitor becomes black, and the red LED light in the Raspberry Pi box becomes solid (no flash) and the green LED light is off. Finally, unplug the power adapter cable first before unplugging other devices.

Before you touch any hardware inside the Raspberry Pi box, make sure the Raspberry Pi is shutdown as instructed above, especially it is powered off, i.e., the power adapter cable is unplugged. However, try not to touch any metal in the box.

Chapter 2

A general model of computers

A computer can be seen as two components: CPU (central processing unit) and memory. CPU can do many operations such as addition, division, etc. It can also do logical operations like `x<1` and `y==3` etc. These operations are on the variables like x and y and are executed by ALU (Arithmetic Logic Unit) inside the CPU. Each variable has a type such as integer (int), float, character (char) or a combination of them (called structure). All variables reside in memory (usually called main memory). If we have a statement like `x = 4`, we basically ask the CPU to store a value 4 into the variable `x` which has a location in memory. By using such assignment statement (`x=y+2`), we can move data around the memory. Note that a variable has two attributes: value and memory location. They are very important to understand the pointers in the C language. For the input and output of data, we move data to the memory of an I/O device like a monitor (usually called device memory). A program can achieve a particular purpose through a number of CPU operations and data movements (which are collectively called algorithm). For example, for our add.c program, we takes the two numbers from the device memory of keyboard, which are then moved to the main memory by the system software called operating system. Finally the program gets the two numbers in character format through `argv[]`. After the computation, the addition result is moved to the device memory of the monitor via the function `printf()`. In summary, a program is just a sequence of instructions for the CPU to transform and move data around in memory. Keeping this understanding in mind will help you understand programming better.

Chapter 3

Instructions of Github

I suggest everyone to register an account at github so that you can upload your code to the github repository instead of on the raspberry pi or your laptop because they are not stable storage for your code. Also having a github account is a good sign of a professional programmer. It is easy to register a github account if you google "github registration". After registration, login to your account, follow the online instruction, and create your first code repository.

You can find all my code from my github account using the following command:

```
$ git clone https://github.com/zhiyihuang/C-examples.git
```

Below are some resources for learning C programming:

- learn-c.org
- https://www.programiz.com/c-programming/examples
- https://www.programiz.com/c-programming#tutorial

Chapter 4

Instructions on text editors and command terminal

First, I want to say a few words about text editor. You can use `nano` as a start. If you feel its function is limited, you can choose more advanced text editors like `vim` (use the command `$ sudo apt-get install vim` to install if you wish to use it on Raspberry Pi and then use `vimtutor` to learn how to use it). However, even with `nano`, there are a lot of functions we could do with combined keys such as `CTRL+k` etc. For example, if you want to delete multiple lines, you can follow the instructions below:

1. use `CTRL+Shift+6` to mark the beginning of your block. 2. move cursor with arrow keys to end of your block, the text will be highlighted. 3. use `CTRL+k` to cut/delete block.

I just found the instructions by googling "how to delete multiple lines with nano editor in Linux". Likewise, you should be able to find instructions for other operations of `nano`. For example, if you want to show line numbers in `nano`, try to google "how to show line numbers in nano linux" and see what you will get.

Second, when you type a command or a file name on a terminal, you can use the `tab` key to complete the rest of the command or file for you if it exists. If you double click the `tab` key quickly, it will show you the options of the commands or file names that match your currently typed characters, so that you don't need to type the full command or file name. It saves your time in typing. If you just want to repeat a command you typed in the past, just use the `UP` key to find out the commands of the past. These skills are essential for competent use of the command terminal. If you are not sure what I am talking about here, ask me and I will show you what I meant.

Chapter 5

Instructions for the Calculator project

We have known functions which are the basic building blocks of our programs. Inside each function, we have variables which could be `int`, `float`, `char` representing integer, decimal or real numbers, and a character like 'a', 'b', '$' etc. Every variable should be declared before you can use it in a statement like `r= x+y;` where `r`, `x`, `y` should be declared as below at the very beginning of a function.

```
int x, y, r;
```

A function is delimited by curly bracelets like { and }. If you declare variables outside any function, they are called global variable which can be used by any function; otherwise they are called local variable which can only be used by the function where it is declared.

There is a simple addition program called add.c in my repository under the directory called calculator.

Before moving to add.c, let me introduce a few commands that you should use often with the terminal:

`$ ls`

to show the files in the current directory;

`$ cd`

to change to a particular directory, depending on the directory name following the command;

`$ nano`

the editor to change your program;

`$ gcc`

the compiler you use to compile your program. If there is only one file, you can compile your program with:

```
$ gcc -o add add.c
```

here *a*dd.c represents any C program file you want to compile, and the compiled machine executable code is in add which again you can choose any name.

Once the compilation is successful, you can run your program using the executable code:

`$ ./add`

here `./` means to find the command (executable file) in the current directory.

Based on the *a*dd.c program, we can easily create programs like *s*ub.c, *m*ul.c, *d*iv.c to do other operations. However, this is not convenient for users to use our program as they have to use different programs each time. Could we combine these programs into one program so that we can simply use a command like: `$ calculate + 3 4`? Also we would like to find out if the users use our programs in a correct way? How can we achieve that? What possible errors could the users make with using our programs?

If you have done the above already, I ask you to think about a calculator program that can repeatedly take input string like "345 + 564" and print the result out. So the program should work as below:

```
$ calculator
3 + 5
= 8
```

```
7 - 5
= 2
3*6
= 18
```

Basically every time the program should take the whole string e.g. `3 + 5` using the function `fgets(buf, 100, stdin)`.

The following code snippet allows you to read strings repeatedly from the terminal:

```
while(fgets(buffer, BUFFERSIZE , stdin) != NULL)
    {
        printf("%s\n", buffer);
    }
```

What you need to add is a parser for the expressions like `3 + 5` which can separate the numbers `3`, `5` and the operator `+`. Then you can use `atoi()` to convert numbers into integers and the rest of it should look like:

```
while(fgets(buffer, BUFFERSIZE , stdin) != NULL)
    {
        printf("%s\n", buffer);
        parser(buffer, num1, num2, operator);
        x = atoi(num1);
        y = atoi(num2);
        if(operator == '+') r = add(x, y);
        else if(operator == '-') r = sub(x, y);
        else if(operator == '*') r = mul(x, y);
        else if(operator == '/') r = div(x, y);
        else {
              printf("Invalid operator %c\n", operator);
              continue;
        }
        printf("= %d\n", r);
 }
```

If you have done the above, the new question is, can your program handle real numbers like `3.4 + 5.6`? Can you change it to handle them?

Chapter 6

Instructions for the GPIO project

The GPIO programs are in my C-examples repository. If you haven't downloaded the code yet, use the following command to download:

```
$ git clone https://github.com/zhiyihuang/C-examples.git
```

The GPIO programs are used to work with two circuits which are drawn in the file called *c*ircuits.pdf under the directory `doc` in the above repository.

One circuit in the file is a LED light circuit, which uses an output pin (pin 8, but could be any pin you set as output) to set the LED light on/off. The other is a switch circuit, which uses an input pin (pin 7, but could be any pin you set as input) to get the on/off state of the switch.

The code which controls the LED light circuit is in the directory `setgpio` and the code which gets the state of the switch in the switch circuit is in the directory `getgpio`.

There are three important functions I provided to you in the gpio.c file in those directories.

Now type the following commands to get into the code directory and compile the code in `setgpio`:

```
$ cd C-examples
$ git pull
$ cd setgpio
$ make
$ sudo ./setgpio 1
```

You should see the LED light is on if the circuit is connected to the Raspberry Pi.

The command `make` is to compile the code into machine code and the resulting executable code file is called `setgpio`. Note that, if there are many files, we use a Makefile script to describe what files we would like to compile and link together. There are a lot of details of Makefile. At the moment, I skip the details for the sake of simplicity, but you are encouraged to have a look at the `Makefile` file which is not too difficult to understand. Usually if there is a `Makefile` in a directory of programs, simply use:

```
$ make
```

to compile and create the executable machine code.

In the code, the three important functions are defined in *g*pio.c and invoked in *s*etgpio.c:

```
void setup_io(void);
```

is used to prepare the gpio pins for memory access. This is compulsory if you want to access the gpio hardware in your code.

```
void setgpiofunc(unsigned , unsigned );
```

is used to set the function of a particular pin. GPIO means "general purpose input/output", so each pin can be set for input or output or other functions. In our code, we set the pin as output so that we can output a high voltage to power the LED light or a low voltage to switch it off.

For example, we set the pin 8 as OUTPUT in our code using:

`setgpiofunc(8, 1);`
Then you can use the function
`void write_to_gpio(char , int );`
to change of output voltage of the pin 8.
For example,
`write_to_gpio(1, 8);`
set the pin to high voltage so the LED light is on; and
`write_to_gpio(0, 8);`
set the pin to low voltage and the LED light is off.
Apart from the existing code, actually you can set a pin (e.g. 7) as INPUT by:
`setgpiofunc(7, 0);`
Once you set a pin as INPUT, you can read the value of the pin using a function:
`int read_from_gpio(int pin);`
For example you can get the value of pin 7 using:
`v = read_from_gpio(7);`
If the pin has an input of high voltage, `v` will be 1; otherwise the pin has a low voltage and `v` is 0;

This function could be used to find out if a switch is on or off if the switch circuit is connected to pin 7 and the 5v pin on Raspberry Pi, as suggested by the switch circuit.

The code for reading the state of the switch circuit is in the directory `getgpio`. You can compile and test the code using the following commands:

```
$ cd ~/C-examples/getgpio
$ make
$ sudo ./getgpio
```

You should be able to tell if the switch is on or off from the displayed message.

More details about gpio pins can be found from *B*CM2835 ARM Peripherals which I uploaded to the C-examples repository.

Now the question is: can you use the gpio functions along with some circuit to control something in real life? This could be a brainstorming session.

Now I want to introduce two constructs that are very useful in programming: `for` loop and `while` loop.

If we want to repeat similar statements, loop constructs can help shorten our program. For example, suppose you have an `int` array defined like:
`int buf[5];`
This definition of array means you have 5 integers grouped together in an array variable `buf`. Suppose you want to add all values of the elements in the array into a variable `sum`, you would have to do this without using a loop construct:

```
sum = 0;
sum = sum + buf[0];
sum = sum + buf[1];
sum = sum + buf[2];
sum = sum + buf[3];
sum = sum + buf[4];
```

However, using a `for` loop, you can achieve the same purpose with the code below:

```
sum = 0;
for(i=0; i<5; i++) sum = sum + buf[i];
```

You can see that we use `i` as an index for the elements of the array. The index `i` starts from 0, increases by one after each loop, and stop when `i` is not smaller than 5, which is decided by the expression

`(i=0;i<5;i++)`, where `i=0;` is the initial condition set before the loop starts, `i<5;` is the condition that should be satisfied for the loop to continue, and `i++` is the operation after each loop (`i++` means `i` plus 1). To understand `for` loop better, try the following code and guess what result you will get:

```
int main() {
int buf[5];
int sum;

for(i=0; i<5; i++) buf[i] = i;

sum = 0;
for(i=0; i<5; i++) sum = sum + buf[i];

printf("The␣sum␣is␣%d\n", sum);
}
```

If you don't quite understand `for` loop, don't worry. There are plenty of material to help you understand `for` loop on the Internet. Just google "how to understand for loop in C" and you will find much better explanation (even cartool) than mine.

The next loop construct is `while` loop. It looks like

```
while (condition) {
statements;
}
```

The condition could be any logical expression like `x>4`, or `y==3`, or simply true. The loop repeats until the condition is false.

For example, if you want make our LED lights flash three times, i.e., on and off three times, you can add the following code at the end of the `main()` function of the file *s*etgpio.c under the `setgpio` directory in our `C-examples` code repository.

```
i = 0;
while (i<3){
  write_to_gpio(1, 8);
  sleep(1);
  write_to_gpio(0, 8);
  sleep(1);
  i++;
}
```

The `sleep()` function adds some delay between the on and off of the LED light. Here we make the delay of 1 second by calling `sleep(1);`.

If you want to make the LED light flash forever, you can use the code below to replace the above code:

```
while (true){
  write_to_gpio(1, 8);
  sleep(1);
  write_to_gpio(0, 8);
  sleep(1);
}
```

Now let us have a close look at the GPIO programs to get a holistic view.

Now let's look at *s*etgpio.c in the `setgpio` directory first.

```
$ cd setgpio
$ nano setgpio.c
```

Then you will see the following code:

```
int main(int argc, char **argv) {
  int n;

  if(argc != 2) {
        printf("Usage: setgpio <numbers either 0 or 1>\n");
        exit(0);
  }

  setup_io();
  setgpiofunc(8, 1);

  n = atoi(argv[1]);
  if(n < 0 || n > 1) {
        printf("The number is invalid\n");
        exit(0);
  }

    write_to_gpio(n, 8);

  return EXIT_SUCCESS;
}
```

In this file, the `main()` function is the entry point of the program. It has two arguments: `argc` and `argv`. `argc` is the number of arguments at the command line and `argv` is the argument strings. For example, for the command `$ setgpio 1`, `argc` is set 2 and `argv[0]` is `setgpio` and `argv[1]` is `"1"`, which are sent to the `main()` function.

A program is a collection of functions invoked like dorminos. For example, after `main()` is invoked, functions `printf()`, `setup_io()`, `setgpiofunc()`, `atoi()`, `write_to_gpio()`, etc are invoked one by one. These functions are defined either by the system software or by me in the file *g*pio.c (e.g. `setup_io()`, `setgpiofunc()`, `write_to_gpio()`). At the moment, you don't need to worry about how these functions are defined but just to know how to invoke them. Also you should clearly know how a function is defined and how function is invoked. For example, `setup_io()` has no argument in its definition, so we invoke it with no argument. It is a function that sets up the memory space of the GPIO pins so that they could be controlled by reading/writing memory addresses.

The function `setgpiofunc()` has two arguments, one for the pin number and one for the direction of the pin, INPUT (0) or OUTPUT (1). This function set the direction of a pin. In *s*etgpio.c, we set pin 8 as an OUTPUT pin which could be used to turn on the LED light or turn it off depending on the voltage of pin 8 we set with the function `write_to_gpio()`.

The function `write_to_gpio()` has two arguments: an integer value (1 or 0) and pin number. In our case, we invoke it like `write_to_gpio(n, 8)` which means we set pin 8 to the value `n`. If `n` is 0, pin 8 will be set to low voltage; if `n` is 1, pin 8 will be set to high voltage. From the code we see that `n` is from a command line argument `argv[1]` and we use the function call `atoi(argv[1])` to change the string `argv[1]` into a number which is then stored in `n`.

To summarize, *s*etgpio.c can change the voltage of pin 8 depending on the command line argument after `setgpio`. If we use the command `$ ./setgpio 1`, then `n` will be 1 and pin 8 is set to high voltage. If we use the command `$ ./setgpio 0`, pin 8 is set to low voltage.

There are some other lines of code like:

```
  if(argc != 2) {
        printf("Usage: setgpio <numbers either 0 or 1>\n");
        exit(0);
  }
```

which is used for error prevention. If the number of arguments is not equal to 2, then we will print error message and exit the program.

Likewise, the code below:

```
  if(n < 0 || n > 1) {
        printf("The number is invalid\n");
        exit(0);
  }
```

prevents the wrong values of `n` which has to be 1 or 0.

Now that we had a closer look at the *s*etgpio.c, I hope you understand the program better.

Let's move to *g*etgpio directory with the command:

`$ cd ../getgpio`

where `../` means the parent directory.

Open the file *g*etgpio.c with `nano`:

`$ nano getgpio.c`

You will see the code like below:

```
int main(int argc, char **argv) {
  int n;

  if(argc != 1) {
        printf("Usage: setgpio \n");
        exit(0);
  }

  setup_io();
  setgpiofunc(7, 0);

  n = read_from_gpio(7);

  printf("The pin value is %d\n", n);

  return EXIT_SUCCESS;
}
```

You will see the code is very similar to *s*etgpio except different arguments for the function call `setgpiofunc(7, 0)` which sets the direction of pin 7 to INPUT. Also we have a new function `read_from_gpio()` which is also defined in *g*pio.c. The function call `read_from_gpio(7)` returns the voltage level of pin 7 which depends on the switch status of the switch circuit.

Though the two circuits are simple, they are very powerful and the LED light or switch could be replaced with any other more advanced devices. Now the question is: can you connect the two circuits together by combining the code of *g*etgpio and *s*etgpio? Basically can you change the code so that the switch circuit could be used to control the on/off of the LED light?

If you have done the above task, there is a new question for you: can you make the two circuits connect remotely with Internet? This is the task in the next chapter.

Chapter 7

The networking programs

You have programmed with two circuits: the LED circuit and the switch circuit. Now let's try to control the LED light on/off through the switch circuit via the Internet. Basically, with network programming, we can use the switch circuit to send a control signal to the *g*etgpio program on a Raspberry Pi, which then sends a command to the *s*etgpio program on another remote Raspberry Pi with the LED circuit. The two Raspberry Pi computers are connected via the Internet.

To achieve the above goal, we need to learn some networking functions.

First, use the following commands to update your C-examples directory.

```
$ cd ~/C-examples
$ git pull
```

Then you will find two new sub-directories: *c*lient and *s*erver. Go to the two directories one by one and compile the programs as below.

```
$ cd ~/C-examples/client
$ make
```

and then

```
$ cd ~/C-examples/server
$ make
$ ./server 9000
```

Then open a new terminal and type the following commands.

```
$ cd ~/C-examples/client
$ ./client 127.0.0.1 9000
```

Then you will see the message `Hello Client!` shown at the new terminal.

What happens is that the server program has sent a message `Hello Client!` to the client after the client connects to the server through the network. The string `"127.0.0.1"` is the local (loopback) Internet (IP) address of the Raspberry Pi, and the number 9000 is the port number of the server which is used to identify the server program. Basically you can use an IP address and a port number to uniquely identify and connect a program on the Internet, no matter where it is.

To understand these programs better, you can work in pairs and use two Raspberry Pi computers to test the client and server programs. To do that, you should first know how to find out the IP address of your computer. Below is the command I use.

```
$ ifconfig | grep inet
```

After this command, you should see the following information.

```
inet addr:192.168.1.46  Bcast:192.168.1.255  Mask:255.255.255.0
inet addr:127.0.0.1  Mask:255.0.0.0
```

As I said, `127.0.0.1` is the local loopback address. To allow remote connection, we should use the other IP address: `192.168.1.46`. Mind you that you may see a different address which you should use in the following commands.

On the same Raspberry Pi computer, run the following command.

```
$ ./server 9000
```

This server program is identified and connectable on the Internet through the IP address `192.168.1.46` and the port number 9000.

Now move to the other Raspberry Pi computer, type the following command (assuming you have updated and compiled the *c*lient/*s*erver programs as described previously on this new Raspberry Pi).

```
$ cd ~/C-examples/client
$ ./client  192.168.1.46  9000
```

You should see the same `Hello Client!` message appears at the terminal.

Your task today is to modify the *c*lient and *s*erver programs to send messages continuously in a while loop. The messages should be read from the standard input via the terminal using the function call `fgets(buf, 100, stdin)`.

Of course, to be able to modify the *c*lient and *s*erver programs, you should understand them first. Modifying existing programs to fit your own purpose is a well-sought ability in the open-source software world, given that there are many programs with source code available on the Internet.

Now let's look at the *c*lient/*s*erver programs one by one.

First, open *s*erver.c under the *s*erver directory using `nano` or any text editor you feel comfortable.

First you will see a long list of header files we included as below. They defines the functions including the network functions we will use in the program.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <time.h>
#include "server_api.h"
```

Note that I defined some network functions for you to use in server_api.h.

Then, as usual, we have the `main()` function and error detection code as below.

```
int main(int argc, char *argv[])
{
int port_num, sockfd, connfd;

  if(argc != 2) {
        fprintf(stderr, "Usage: %s <port_num>\n", argv[0]);
        exit(1);
  }

  port_num = atoi(argv[1]);
  if(port_num < 2048) {
        fprintf(stderr, "Invalid port number: %d\n", port_num);
        exit(1);
```

```
}
```

The above code also gets the port number from the command line argument `argv[1]` which is converted into an integer stored in the variable `port_num`.

Then the program creates a socket (a connection point on the Internet) with the function `setup_server_socket(port_num)` as below.

```
sockfd = setup_server_socket(port_num);
```

The socket descriptor is stored in `sockfd` which will be used to accept connections from the client. The variable `port_num` combined with the IP address of the local Raspberry Pi uniquely defines the connecting point of the socket on the Internet.

Then the server is waiting for connection requests from the client as below.

```
connfd = accept_client_connection(sockfd);
```

If the connection between the server and the client is established (actually now you should have run the client program on another Raspberry Pi), `accept_client_connection(sockfd)` will return another socket descriptor to `connfd`. We will use `connfd` to exchange messages between the client and the server.

The server program sends a message using the `write(connfd, "Hello Client!\n", 15)` function call, where 15 is the length of the message.

```
write(connfd, "Hello Client!\n", 15);
```

Finally the server sleeps for one second to allow the message to be sent before it closes the socket and exit.

```
sleep(1);
close(connfd);
```

Similarly the *c*lient.c program has a long list of header files for functions it will use including the network functions. Then we have the `main()` function and the error detection code as below.

```
int main(int argc, char *argv[])
{
  int sockfd, port_num, n;
  char buf[1000];

  if(argc != 3){
        printf("Usage: %s <ip of server> <port number> \n",argv[0]);
        exit(1);
  }

  port_num = atoi(argv[2]);
  if(port_num < 2048) {
        printf("Invalid port number: %d\n", port_num);
        exit(1);
  }
```

Note that the client program has three command line arguments: client, IP address, and port number.

Then we set the initial values of the `buf` char array to 0.

```
memset(buf, 0, 1000);
```

where 1000 is the size of the array `buf` and 0 is the value to set.

Then we connect to the server with the server IP address and the port number acquired from the command line arguments.

```
sockfd = connect_server(argv[1], port_num);
```

where the socket descriptor is returned and stored in `sockfd`.

Then we use `read()` function to get the message sent by the server

```
  n = read(sockfd, buf, 999);
```

where the message is read into `buf` whose size is 999 (to allow one byte as the terminator `\0` of the string). The number of bytes (characters) is returned to `n`.

If `n` is not 0, then we print the message as below.

```
  if(n > 0) printf("%s\n", buf);
```

Then we close the socket and return.

```
close(sockfd);
return 0;
```

Now, modify the above programs to allow the client and server to exchange messages continuously in a while loop.

After doing this, integrate the code from *g*etgpio and *s*etgpio into the *c*lient/*s*erver programs in order to make the switch circuit to remotely control the LED light in the LED circuit on and off. Note that the two circuits should connect to two different Raspberry Pi computers respectively. Also, instead of sending "Hello" messages, "0" or "1" should be sent depending on the state of the switch.

Chapter 8

I2C interface and temperature sensor

In this chapter, we will learn how to interact with the I2C interface which is commonly used to connect sensors such as the temperature sensor we are going to use in our programs.

First, we need to enable the I2C interface on the Raspberry Pi:

```
$ sudo raspi-config
```

Then, under the prompted menu, go to *A*dvanced Options and then enable I2C.

Next, install the I2C related library and tools:

```
$ sudo apt-get install libi2c-dev
$ sudo apt-get install i2c-tools
```

Finally, edit the file /etc/modules:

`$ sudo nano /etc/modules`

Add in /etc/modules a line: `i2c-dev` if it is not there.

Then shutdown the Raspberry Pi and unplug the power cable of the Pi.

Plug in the temperature sensor board (called *M*PU9255) as shown at:

- http://www.cs.otago.ac.nz/staffpriv/hzy/iot.php

In this way, we plug the sensor board to the I2C interface of the Raspberry Pi. Actually, apart from the temperature sensor, this sensor board (*M*PU9255) also has 3D accelerometer, 3D gyro, and 3D magnetometer.

Now check the I2C interface is working with the sensor board using the following command:

`$ sudo i2cdetect -y 1`

You should see some non-zero numbers shown on the terminal window.

For more details of how to use the sensors, you may refer to:

- https://learn.sparkfun.com/tutorials/mpu-9250-hookup-guide
- https://github.com/sparkfun/SparkFun_MPU-9250_Breakout_Arduino_Library/tree/master/src

At the moment, you should be alright by following my instructions to work with the temperature sensor.

To understand the I2C protocol, you may refer to:

- https://learn.sparkfun.com/tutorials/i2c
- http://www.robot-electronics.co.uk/i2c-tutorial

Now let us look at the code for getting the data from the temperature sensor.

First, go to the i2c directory and compile the code:

```
$ cd ~/C-example/i2c
$ make
```

Then, run the command:

```
$ ./get_temp
```

You should be able to see the temperature data. Put your thumb on the sensor board and run the above command again and again. You should see the temperature numbers increasing.

Now let us understand how the programs work by reading the source code below.

First, the header files are included as usual.

```
#include <stdio.h>
#include <stdlib.h>
#include "i2c_api.h"
#include "MPU9255.h"
```

You may notice the two header files, *i*2c_api.h and *M*PU9255.h. The first header file is for the declaration of I2C related functions defined in *i*2c_api.c and the second header file defines the register addresses of the sensors on the sensor board called *M*PU9255. You will need to be comfortable with these hardware related information, as each hardware vendor will provide you such information in their product manual or their library code.

Then, in the `main()` function, you will see the code below.

```
int main(int argc, char** argv)
{
        int fd, serversocket, clientsocket, n;
        float temperature;

        fd = open_i2c();

        temperature = getTemperature(fd);

        printf("The temperature is %f degrees.\n", temperature);
}
```

The `main()` function basically opens the I2C device with `open_i2c()` and the get the temperature data with the function `getTemperature(fd)`. Note `fd` is the file descriptor representing the I2C device which is connecting to our *M*PU9255 sensor board. The function `open_i2c()` is defined in *i*2c_api.c. The file *i*2c_api.c also includes two other functions `read_byte()` and `write_byte()`, which read from and write to the registers of the sensor board.

So the key business of getting the temperature is done in the following function.

```
float getTemperature(int fd)
{
        u8 byte1 = read_byte(fd, DEVICE_ADDRESS, TEMPERATURE_DATA_REG);
        u8 byte2 = read_byte(fd, DEVICE_ADDRESS, TEMPERATURE_DATA_REG + 1);

        if(byte1 < 0 || byte2 < 0)
        {
                printf("error reading temperature.\n");
                exit(1);
```

```
        }

        short result = (byte1 << 8) | byte2;

        return ((float) result - 21.0) / 333.87 + 21.0;
}
```

In this function, we use `read_byte()` to get the two bytes of the temperature data stored at `TEMPERATURE_DATA_REG` and `TEMPERATURE_DATA_REG + 1` respectively. We also need to provide the sensor device address `DEVICE_ADDRESS` to the function. You might be curious about the exact values of the addresses. Check the file *M*PU9255.h you will find the following.

```
#define         DEVICE_ADDRESS                  0x68
#define         TEMPERATURE_DATA_REG            0x41
```

As I said before, these information are provided by the vendor. For *M*PU9255, you can find them from:

- https://learn.sparkfun.com/tutorials/mpu-9250-hookup-guide

Back to the `getTemperature()` function, after the two bytes of temperature data are read, they form a 16-bit integer by shifting the first byte by 8 bits, which are then stored in the variable `result`.

Finally, we need to convert the value in `result` into temperature in degrees Celsius following the manual of *M*PU9255:

`((float) result - 21.0) / 333.87 + 21.0`

This is an interesting part of dealing with sensor data. Data usually have no meaning. To make them meaningful, we should convert them into a meaningful format such as degrees Celsius. Also, to make the measurement accurate, we need to calibrate the measured temperature with other more accurate temperature meters. This is usually done by the vendor with a given formula like the above. However, if we think the temperature is not accurate enough, we need to get it calibrated. From this example, you see that computer science is closely related to other sciences. Everything connects!

Now that you have understood the I2C programs regarding how to get temperature from the temperature sensor, can you combine this program with the `setgpio` program so that you can use the temperature sensor to control the LED light on/off? That is, if the temperature is higher than 25 degree, turn the LED light on; otherwise, it should be off.