**ECE 150 Final Project Report**

**Group Name: Smart Plant**

**Group 40**

**Xiaohan Ma**

**Xueyan Sun**

**Yuan Xie**

**University of Waterloo**

**December 4, 2017**

**Project Overview**

**What is the project about?**

* Our project is an application which can measure and analyze the soil moisture of plants and display necessary information.

**What would you have liked it to do if it could do everything?**

* According to our original plan, we expected our project could also visualize and display data on the OLED, send a twitter notification message using Losant platform to the users according to different soil moisture conditions of plants.

**What subset does it do?**

* Our project allows people to monitor soil moisture statue of their plants, and send detailed reminders when the soil moisture level is below or above certain values, or stay in an abnormal condition for too long.

**System design:**

**What is the complete system design: hardware and software?**

HARDWARE

1. Connected the soil moisture sensor to the Arduino Dock with 3 Male-to Female wires

* Connecting the GND Pin of Soil Moisture Sensor with GND Pin of Arduino Dock
* Connecting the VCC Pin of Soil Moisture Sensor with 5V Pin of Arduino Dock
* Connecting the A0 Pin of Soil Moisture Sensor with A0 Pin of Arduino Dock

1. Connected the Onion Omega 2 directly onto the Arduino Dock
2. Connected the USB key with the USB port in the Arduino Dock
3. Once everything has been set up, we can run our ‘Project.cpp’ code on the omega 2, and check the processing result by opening the ‘output.txt’ and ‘logfile.txt’ from a different terminal.

SOFTWARE

1. Uploading a sketch to Arduino Dock through Arduino IDE to connect the Soil Moisture Sensor with Arduino Dock, and then send the moisture data to the ttyS1 UART port.
2. In our C++ source code, we used the open () function in <fcntl.h> library to open the ‘/dev/ttyS1’ port, and then we used the read() function in <unistd.h> library to read the data from the ‘/dev/ttyS1’ port, and then store data to an unsigned char array. Next, we transform the char-typed data to int-typed data, which can be used for our following processing data. We also used the <errono.h> library which includes the variable “errno” that gets set to whatever number the error was when any of the above functions return -1.
3. In our main function, because the data is continuously passed into our program, we created a while loop with condition (1) to iteratively process the data. Each time we enter the loop, the code will call the “read\_serial” function, which would return a int typed data, and we would do the rest of our processing within the while loop.

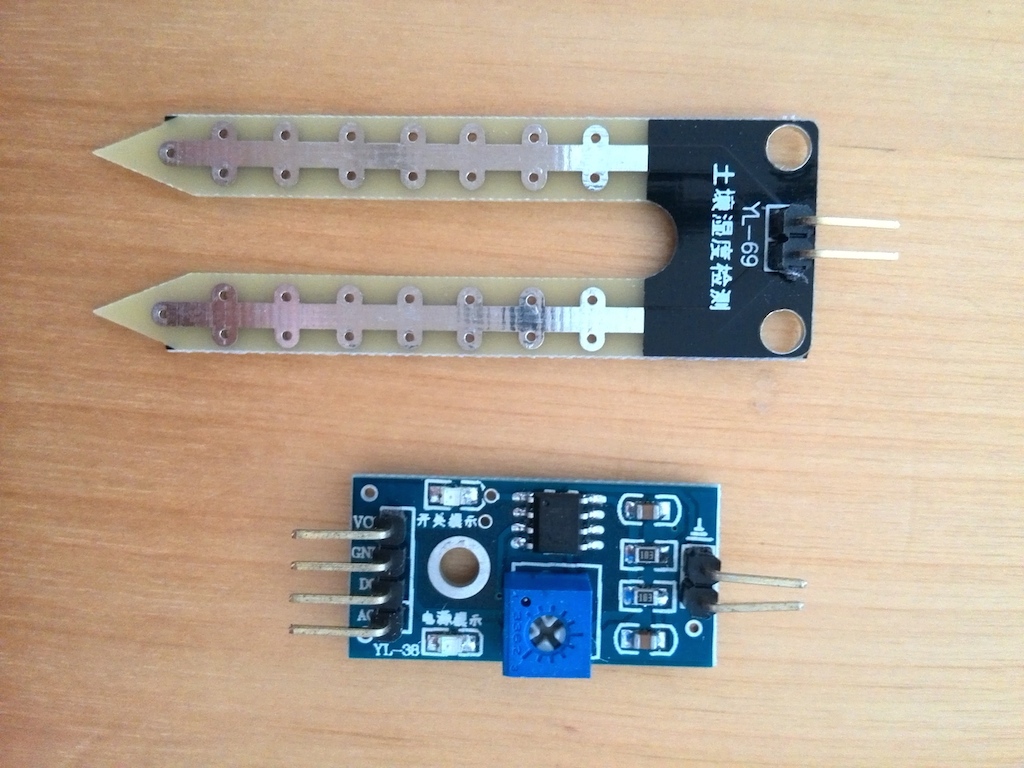
**All components we had used:**

* **Hardware:**

1. Arduino Dock 2
2. Onion Omega 2
3. Soil Moisture Sensor YL-69, YL 38
4. Male-to-Female and Female-to-female wires
5. USB wire
6. An USB

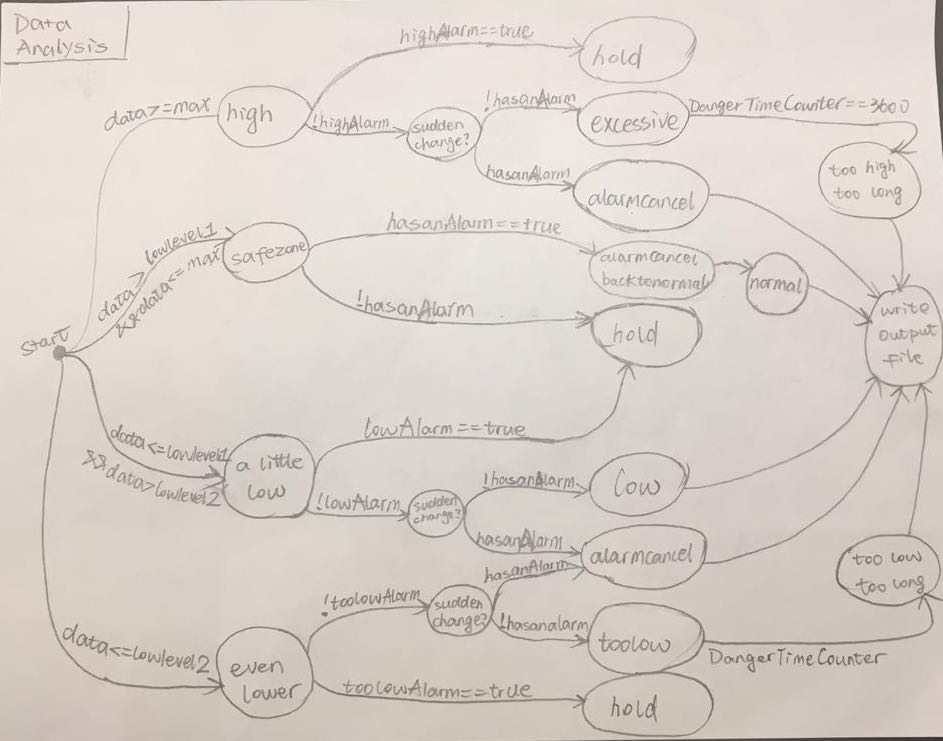
* **Software:**

1. Arduino IDE
2. Virtual Machine (Ubuntu)
3. Project.cpp

Sensor Picture: 

**Software design:**

State machine diagrams:



(State machine diagram of our data analyzing process.)

(NOTE: the repetitive cases “hold” and “alarmcancel” represent exactly the same and contain the identical operations. They are put in different position just for the arrows to be easily connected to each other.)

**Function call tree:**

Main

while loop

Open\_serial

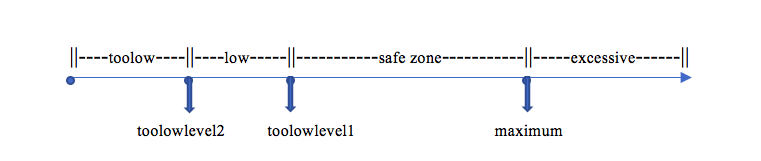
Read\_serial

Set\_interface\_attribs

writelog

writeoutput

Our moisture level could be generally divided into three ranges, roughly displayed as the following axis:



|  |  |
| --- | --- |
| FLOAT Variables:  the crucial values divide the range into four states. | |
| maximum: | the maximum suitable moisture level for plants,  above which current condition will be regarded as "excessive" state. |
| lowlevel1: | the minimum suitable moisture level for plants,  below which current condition will be regarded as "low" state. |
| lowlevel2: | dangerously low moisture level,  below which current condition will be regarded as "toolow" state. |
| BOOL Variables:  help to determine whether or not reminding info has been sent (i.e. wrote to the output file), since the user will not want the same repetitive message every time a new data is gained. | |
| highAlarm: | the alarm (i.e. signal) for first-time entrance of "excessive" state,  TRUE only if current data exceeds maximum. |
| hightoolongAlarm: | the alarm for three-hour consecutive stay in "excessive" state,  TRUE only if current data has been in "excessive" state for three hours or more (multiplicity of three hours, i.e. 6,9 hours). |
| lowAlarm: | the alarm for first-time entrance of "low" state.  TRUE only if current data exceeds lowlevel1, but not yet exceeds lowlevel2. |
| toolowAlarm: | the alarm for first-time entrance of "toolow" state,  TRUE only if current data exceeds lowlevel2. |
| toolowtoolongAlarm: | the alarm for three-hour consecutive stay in "toolow" state,  TRUE only if current data has been in "toolow" state for three hours or more (multiplicity of three hours, i.e. 6,9 hours). |
| hasanAlarm:  (i.e. has another alarm other than existing alarm type) | the signal indicating whether or not data is currently in an abnormal state (i.e. one of the alarms above is true). This variable helps to check if there is sudden change between two following data (for instance, moisture level changed from "low" state to "excessive" state, in which case the original alarm should be canceled first).  TRUE if any abnormal state is true;  FALSE only if all the alarm has been canceled, or current data not yet exceeds any abnormal moisture level. |
| Enum mystate {excessive, normal, low, toolow, alarmcancel, toolowtoolong, hightoolong, hold, ignore};  (indicating current moisture state, switching to execute corresponding operations) | |
| Excessive: | 1) "highAlarm" set to be true;  2) use the integer "DangerTimeCounter" to check how long consecutively moisture level stayed in this state.  In such case, hightoolongAlarm is set to be true, and new corresponding output should be saved to output file.  (note: DangerTimeCounter is reset to be 0 only if moisture level state has changed or 3 hours maintained.) |
| normal | This state could not be entered. It can only be passed into "writeoutput" function to output a "back-to-normal" statement if any kind of alarm is reset to be false (i.e. alarm canceled). |
| low | 1) "lowAlarm" set to be true.  2) passed into "writeoutput" function to output a corresponding statement. |
| toolow | 1) "toolowAlarm" set to be true;  3) passed into "writeoutput" function to output a corresponding statement.  2) use the integer "DangerTimeCounter" to check how long consecutively moisture level stayed in this state.  In this case, toolowtoolongAlarm is set to be true, and new corresponding output should be saved to output file.  (note: DangerTimeCounter is reset to be 0 only if moisture level state has changed or 3 hours maintained.) |
| alarmcancel | 1) set all the alarm to be false;  2) set the DangerTimeCouner to be 0;  This case indicates that current moisture level has been changed (not necessarily back to normal). |
| toolowtoolong | corresponding output file writing operation |
| hightoolong | corresponding output file writing operation |
| hold | This case indicates the situation when some kind of abnormal moisture level is maintained. (i.e. Alarm was turnt on before and is still satisfied to be on). No operation is needed. |
| ignore | This case is set up to deal with a special return value of function "read\_serial". After the data (of char type still initially) is artificially converted into int type, there is a line return after the true value, resulting a mess when we try to use atoi. We identify whether or not line returns occur and return -2 in such a case. Back in main function, whenever read==-2, we are getting a line return, and thus set "currmystate" to be in case ignore. |
| Enum type { error, trace }  (indicating type of info appearing in log structured file) | |
| error | Error occurs and needs to be put in log file. |
| trace | Updating current operation taken by program. |
| Enum logstate {recorded, empty, null}  (indicating content hint for log file's corresponding output message) | |
| recorded | Current moisture level state has been saved to output file. |
| empty | Current moisture data is not gained. |
| null | Indicating the program is currently switching to determine the state. |

* **What each function represents**

(“set\_interface\_attribs” and “open\_serial” functions are not accomplished by ourselves alone and references are at the end of report)

|  |  |
| --- | --- |
| int set\_interface\_attribs(int fd, int speed)  From line 25 to line 58 | Establish the serial communication with the UART serial port |
| int open\_serial(const char \* portname)  From line 62 to line 75 | Using the open() function to open the ‘/dev/ttyS1’ UART port. |
| int read\_serial(int fd, int\* buffer)  From line 81 to line 105 | Using the read() function to read data from ‘/dev/ttyS1’ UART Port, and save the data to the “buffer” char array. Then, convert the data from char to int, then return the data. |
| int main() {  From line 108 to line 289 | 1) analyzing all the data in an infinite loop to determine which state is the moisture level currently in'  2) call writelog function to output corresponding information to "logfile.txt";  3)call writeoutput function to output corresponding information to "output.txt"; |
| int writelog(type currtype, logstate currlogstate){  From line 292to line 328 | 1) use currtype to determine which type of information needs to be written to logfile;  use currlogstate to determine what content needs to be written to logfile;  2) output current time (when outfile) as a "timestamp" at the beginning of every line |
| int writeoutput(mystate currstate, const float data)  From line 330 to line 393 | 1)use currstate to determine which state the moisture level is currently in;  use data to output current data to output file.  2)output corresponding reminding messages to output file to inform the user with current condition and necessary actions. |

**System-independent components:**

* Parts of code that you could have written without any knowledge of the hardware

The system-independent components basically are the codes we wrote for analyze the data as well as write them into a new file, which is from line 117 to line 411 in Project.cpp

**System-dependent components:**

* Parts of code that rely on the hardware that you are using specifically (Reading and writing values to and from the pins on the Omega board would be an example of code that is system-dependent)

The system-dependent components are the codes we used for connecting the pins of Arduino Dock with pins of Soil Moisture Sensor and sending the data to the ‘/dev/ttyS1’ port. Also, in the Project.cpp, from line 23 to line 114 is also a system-dependent component. In this part, it requires the information based on what hardware we used in order to identify the which serial port we are going to use. Thus, this part is also a system-dependent component.

**Logging infrastructure:**

**We use the writelog function to implement this part.as following:**

1)We use the currtype, which is of the "type" enum type, to determine what type of info is currently being written to logfile.txt:

a) If currtype is error, it indicates that some error occurs:

If currlogstate is empty: SENSOR FAILED TO GET DATA FROM SENSOR. NO DATA BEING ANALYZE...

Otherwise: FAILED TO OPEN THE OUTPUT FILE.

b) If currtype is trace, it is tracing the running process:

If currlogstate is null: SWITCHING TO ANALYZE CRRENT DATA......

Otherwise: REMINDING MESSAGE SUCCESSFULLY RECORDED TO OUTPUT FILE.

2) We use the currlogstate, which is of the "logstate" enum type, to determine what content needs to be output (as stated above in the examples).

**Additional aspects**

Identify all source code files (.cpp, .h, makefile, anything else required to build the source code)

|  |  |
| --- | --- |
| Source file | Project.cpp |
| Libraries | * #include <errno.h> * #include <fcntl.h> * #include <stdio.h> * #include <stdlib.h> * #include <string.h> * #include <termios.h> * #include <unistd.h> * #include <time.h> * #include <fstream> |

**Testing**

**Identify how you know your system does what it claims to do**

In order to ensure our program will complete tasks as expectation, each function in our source code was tested separately before we test our program in the virtual machine. In order to test whether our program would output the correct information in the file, we first use terminal to see if we can get the correct output message or not. In total, we have 4 different scenarios, which are "excessive" (when the soil moisture level above 72%), "normal" (when the soil moisture level between 72% and 40%), "low" (when the soil moisture level between 40% and 25%), and “too low” (when the soil moisture level below 25%).

To test them, we used different printf to output the data we originally get, the case current moisture level is in, and the function we entered all to the onion console directly, so that we do not have to open the logfile to check every time. According to output information, we first determine whether current moisture level state matches the data or not. If yes, we then open the output file looking for the corresponding reminding message. If no, we use the printed information to navigate the problem and modify our code. Finally, we check the logfile to see if every "trace" is recorded or errors if any.

After modifying our code, we mixed four cups of soil with different amount of water to create different states the soil could possibly have, and then use those distinct values to set our maximum, lowlevel1 and lowlevel2 value. Then we use these four cups of soil, switch our sensor between them and see if the change of output is correct.

**Limitations**

**What doesn’t your project do (that it might be expected to do?)**

* In our proposal, we indicated that when the soil moisture level below a certain point, we expect to notify the users with a twitter message. However, we did not achieve this application in our project. When we attempted to include this application, we found out that we can use the Losant IoT platform instead of C++ code to send the notifications. Since this is a programming course, we think this function is not that relevant. Therefore, we gave up this application and came up with a new idea which is to output different messages into a file according to different conditions to replace the twitter notification. To be more specifically, when the soil moisture level is below or higher than a certain point, instead of sending a twitter notification to users, we will write a message into a new file to reminder the users that their plants are either with too much water or there is not enough water in the soil. (Actually, we did create a Twitter application on the Losant website)

**Lesson learned**

**What we learned from this project?**

* Basic knowledge of cross-compilation, for example, what syntax we have to type into the virtual machine in order to cross-compile our code
* Basic knowledge of how to design, develop and start a small embedded-system project
* How to refer to documentations effectively, comprehend it and implement it so that it can match our code and purposes
* How to debug our program without a debugger and deal with different situations using limited knowledge
* How to read data from UART Serial Port
* How to write log files
* How to connect hardware component correctly

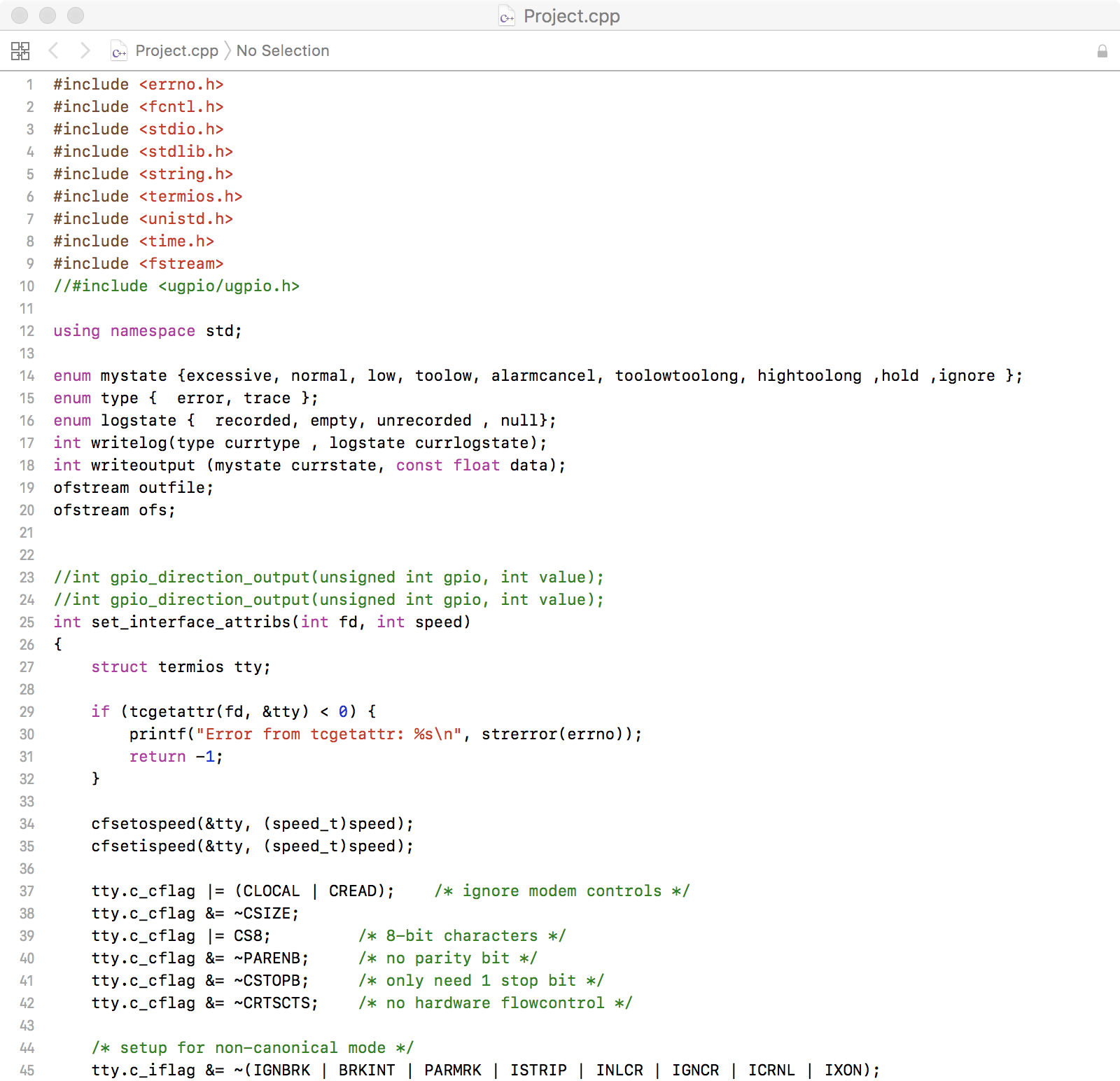
**Reflect on your project: what would you do differently if you were starting it again?**

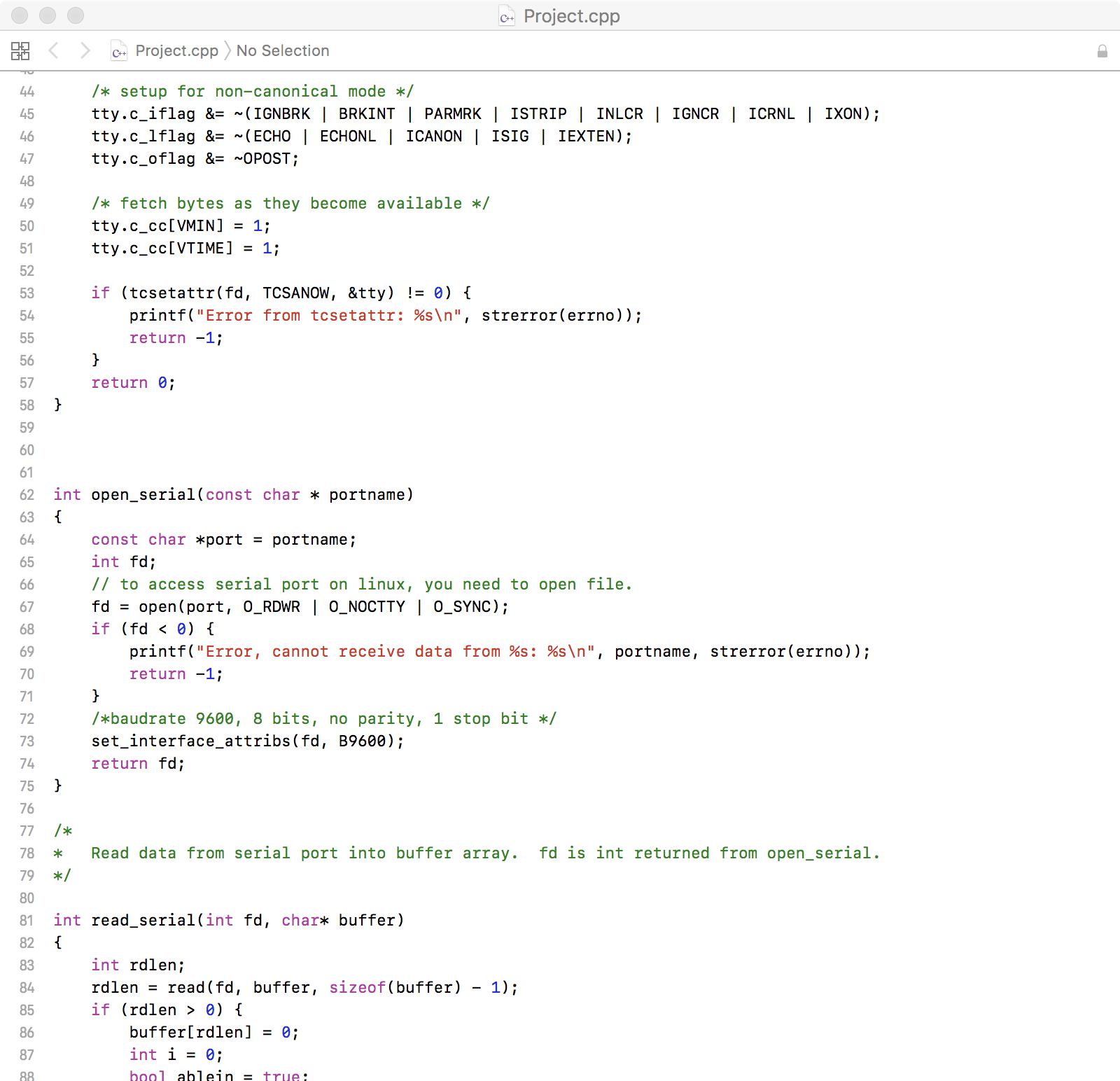
* Firstly, we would come up with a more specific schedule and implementation process for our project, including the anticipated required time, and the maximum acceptable delay for each part.
* Secondly, we would like to have multiple plans, for example, if the Plan A does not work within certain period, we will use Plan B instead.
* Thirdly, we would spend more time on reading the documentations for unfamiliar libraries, because we can learn the syntax and methods provided by the library and implement them in our code later. We would not waste too much time on Googling things any more. Reading the documentations is a far more efficient learning way.

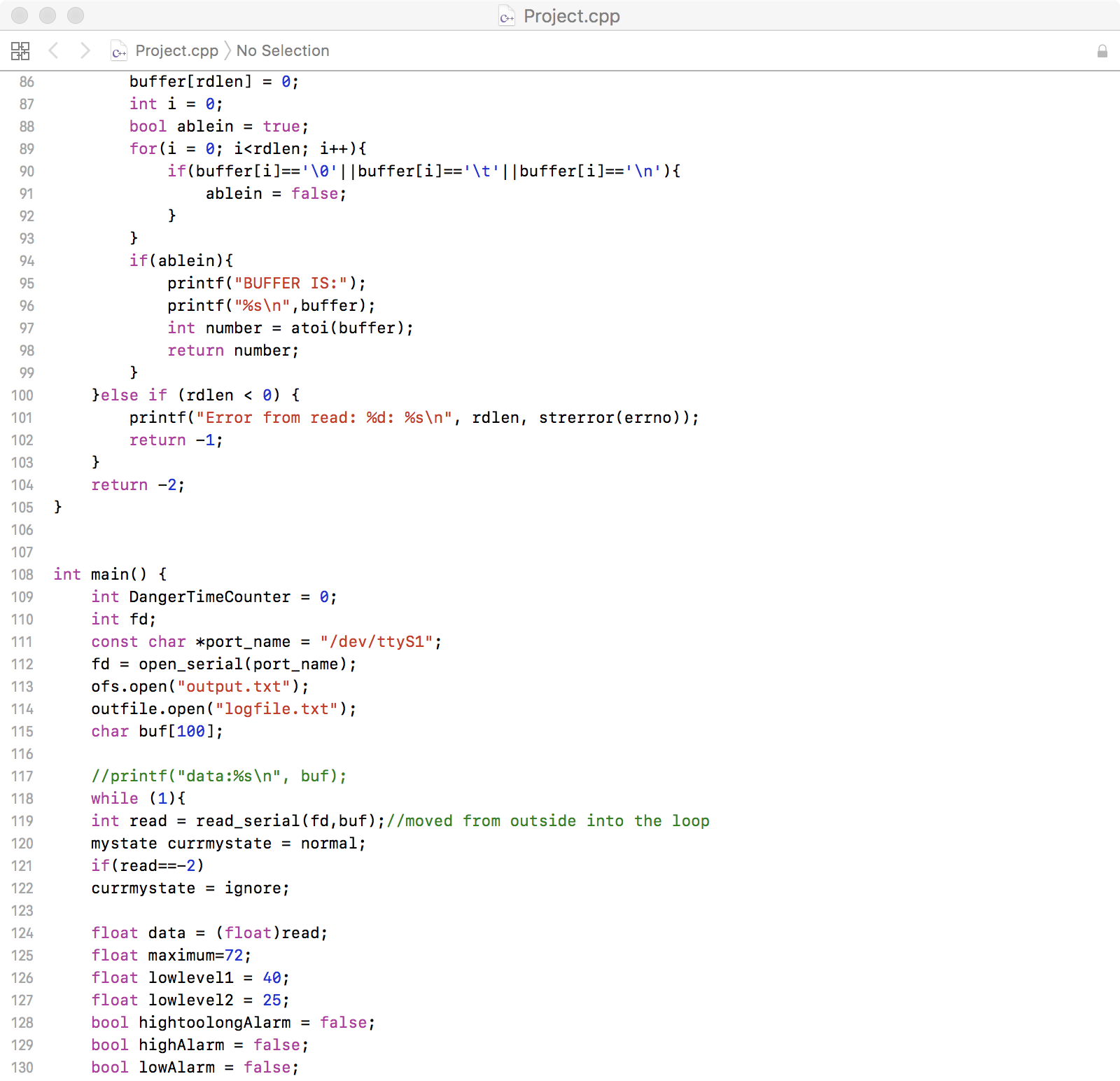
**Appendix**

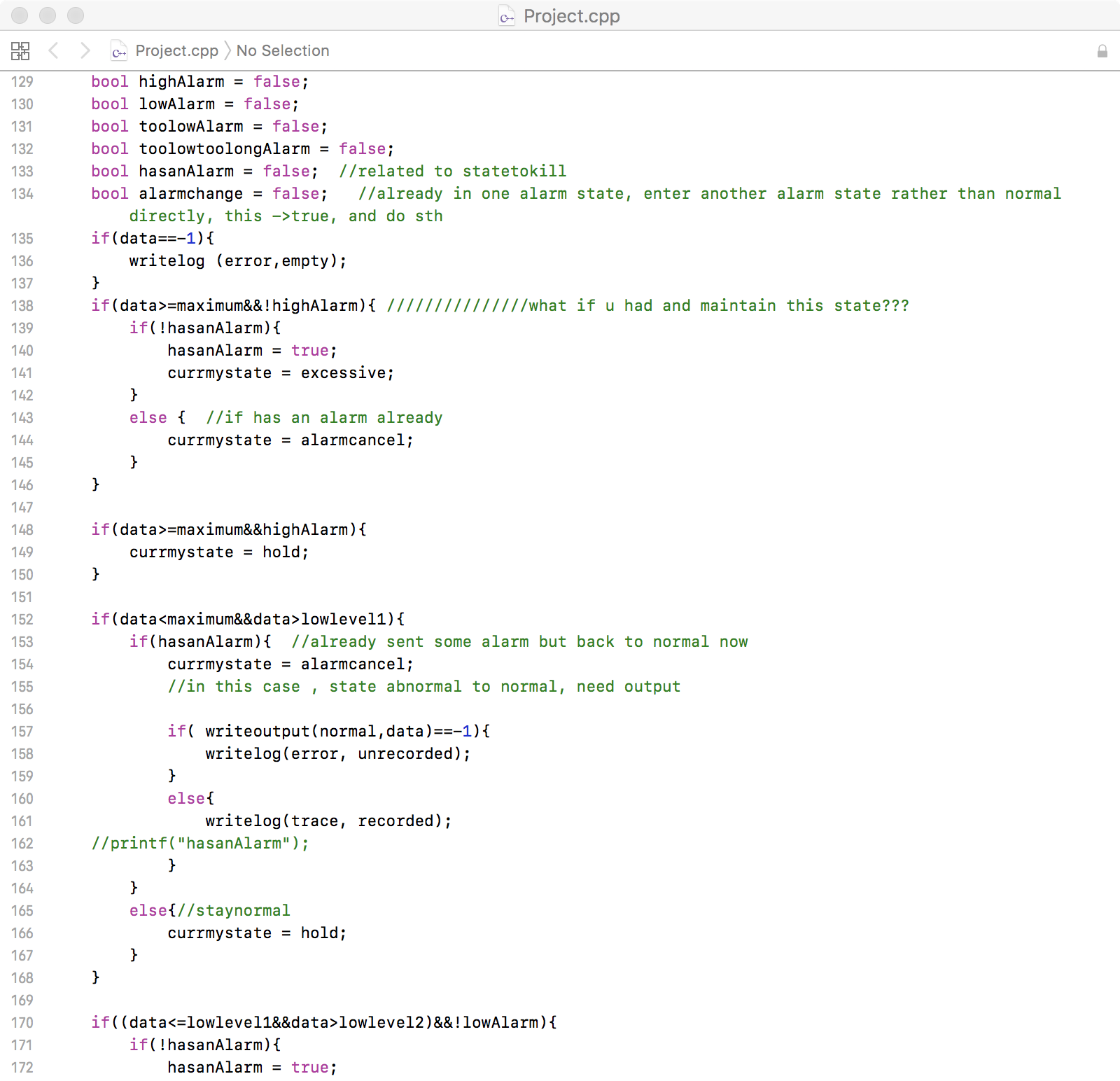
**Source code**

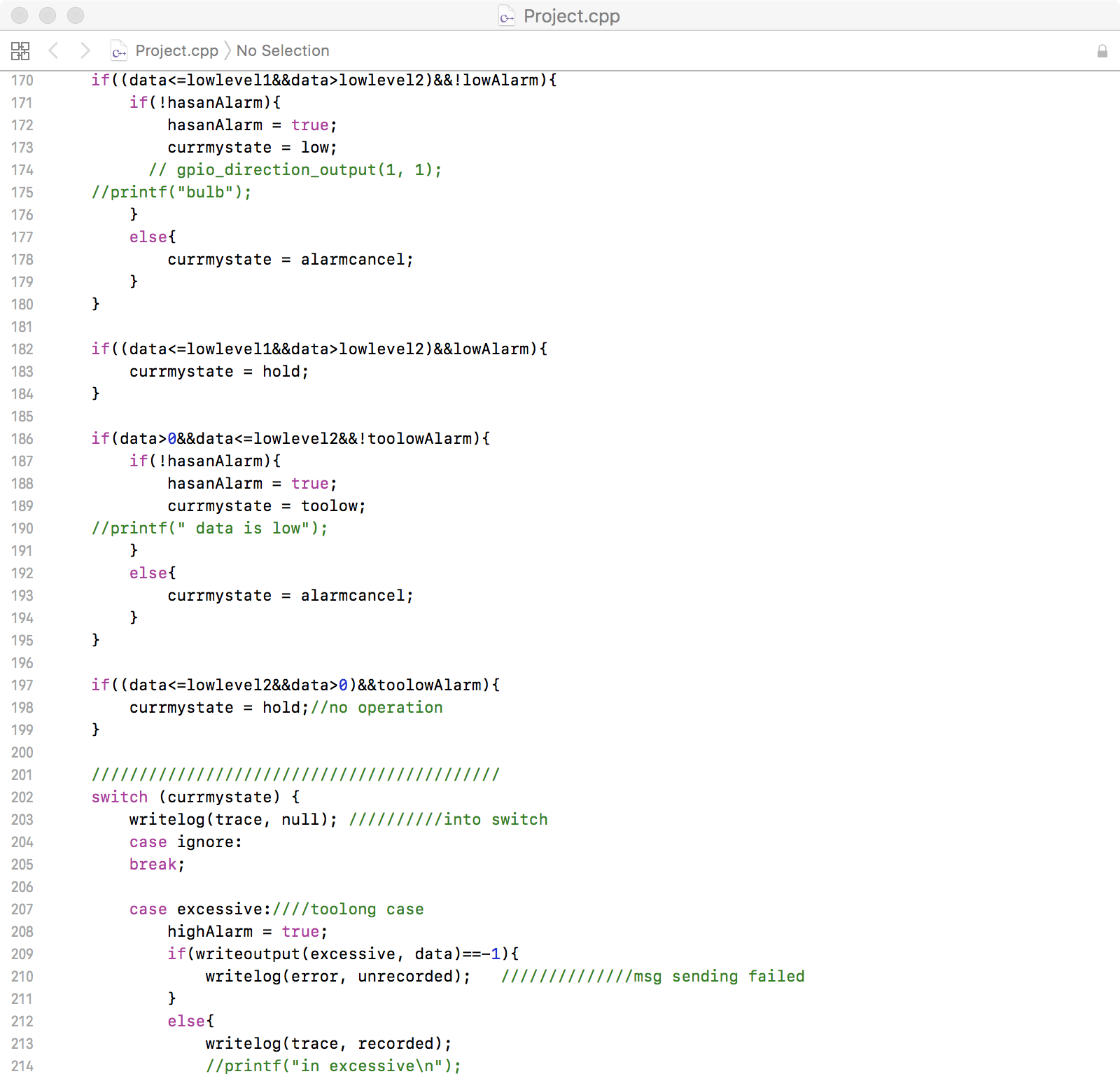
(note: for explanation, refer to variable and cases explanation before. Comments in the code below may not be updated.)

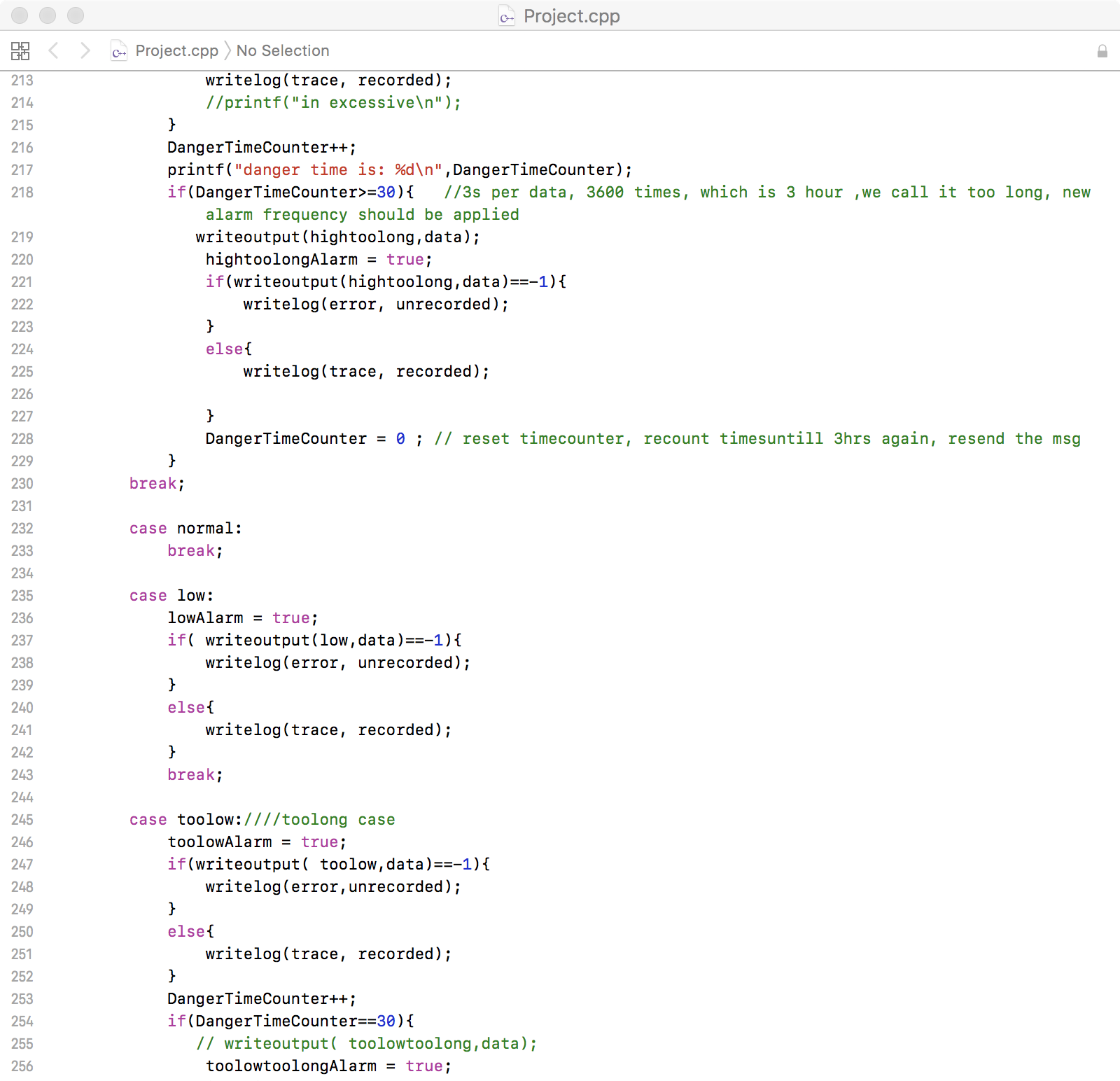


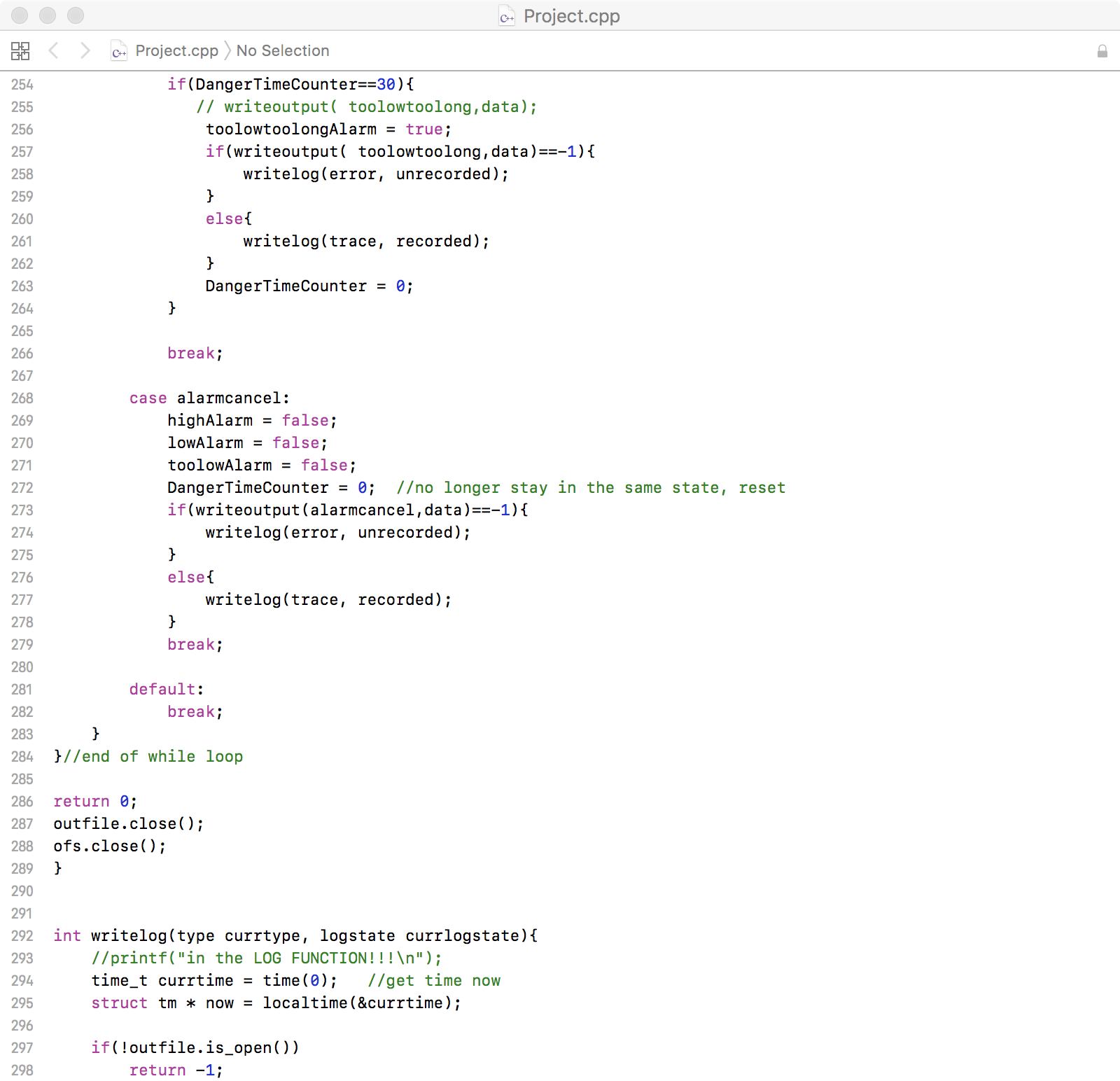


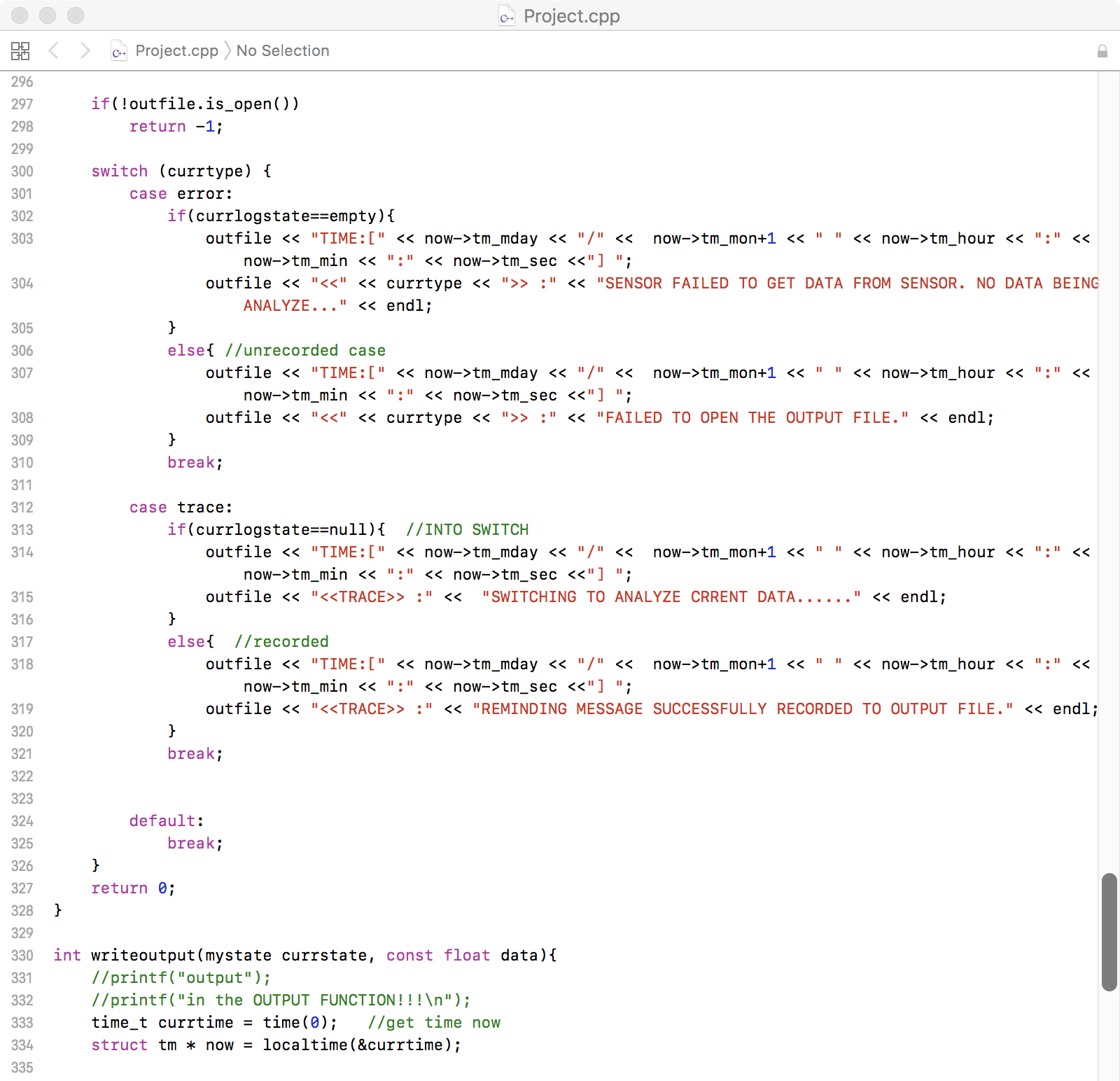


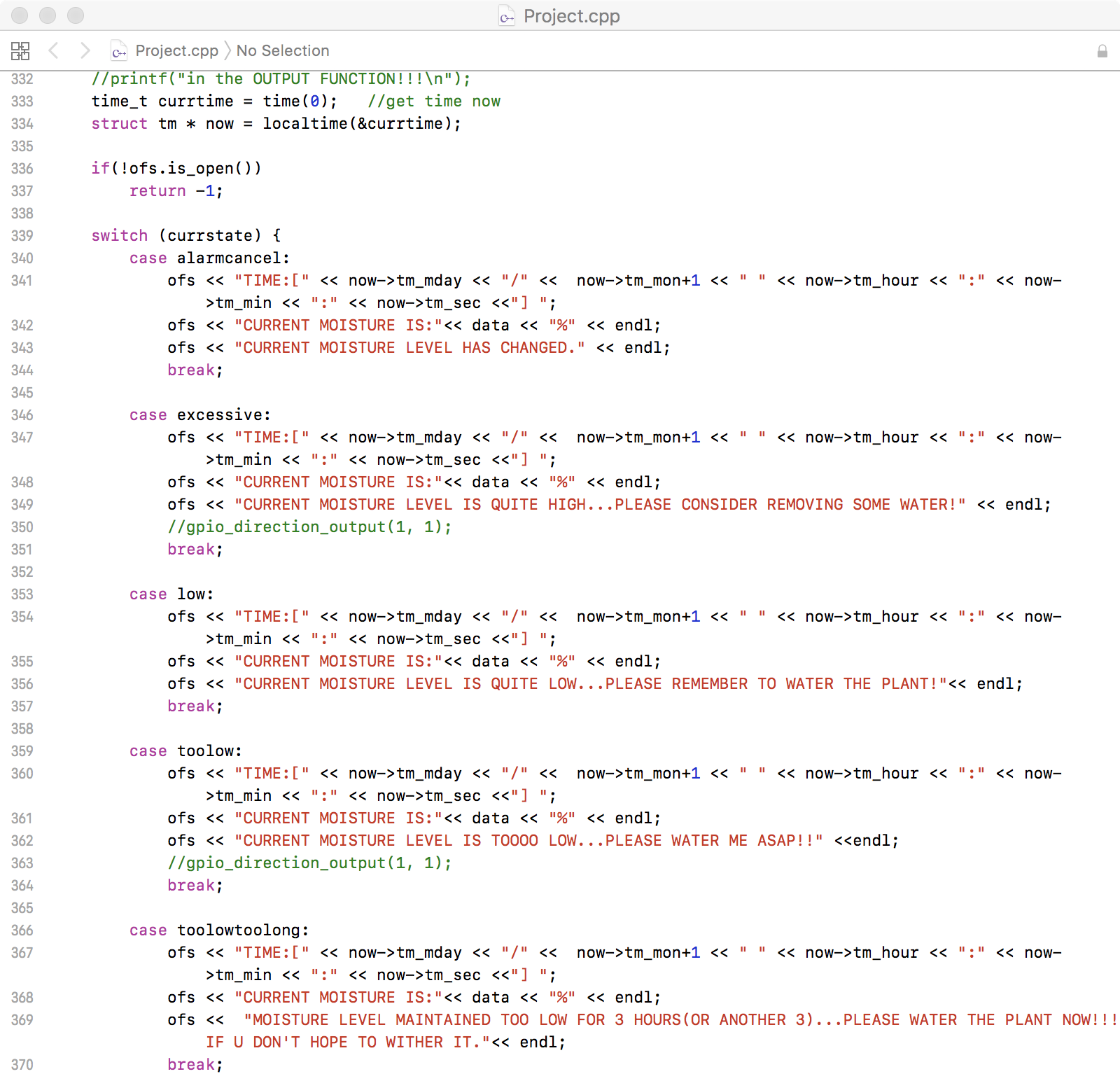


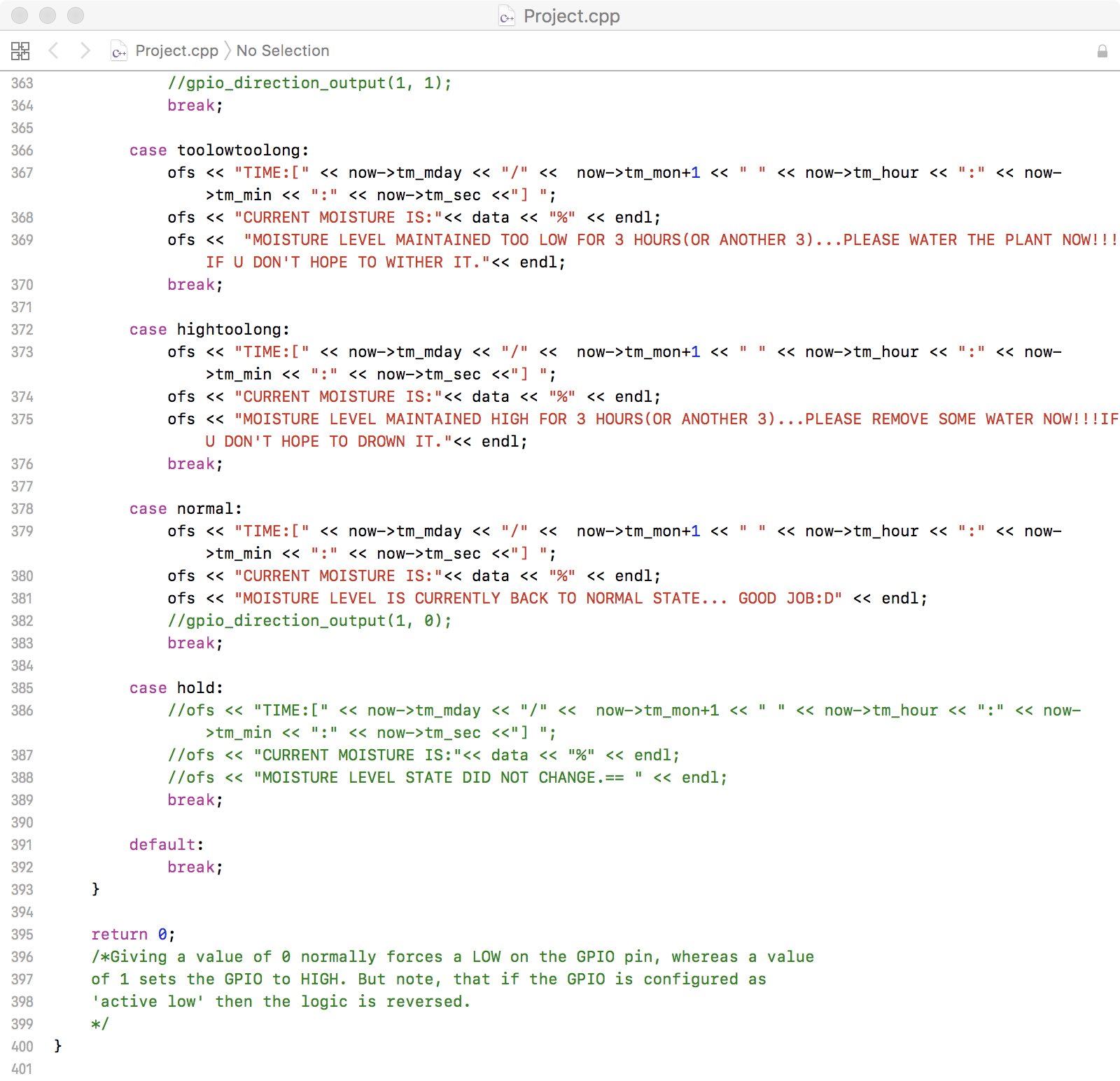












**Peer contribution**

Xueyan Sun is in charge of the data analyzing’s code part and corresponding report writing;

Xiaohan Ma and Yuan Xie are in charge of data gaining and converting and corresponding report writing;

**Reference:**

Documentation for <fcntl.h>:

<http://pubs.opengroup.org/onlinepubs/7908799/xsh/fcntl.h.html>

<https://docs.oracle.com/cd/E36784_01/html/E36873/fcntl.h-3head.html>

documentation for read() in <unistd.h>:

<http://pubs.opengroup.org/onlinepubs/009695399/functions/read.html>

Merry Christmas!☺