Chapter 2 Bit Manipulation and Data Representation

In many embedded systems, the usual variable and data types may not directly make sense. Instead, one must figure out what each individual bit represents. The purpose of this chapter is to get familiar with such scenarios which often require bitwise operations. We will see an example about a room controller on the host platform (e.g., lab's Linux Mint).

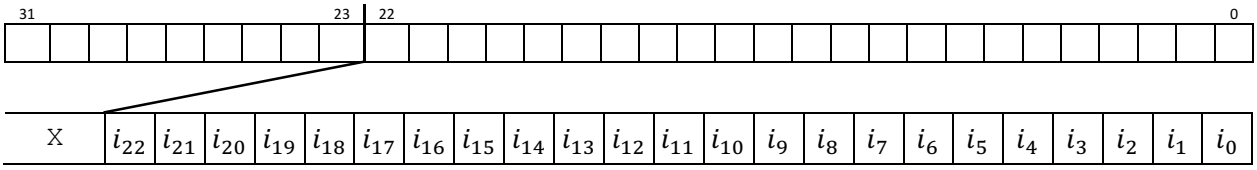
Note that a number of functionalities are already implemented in the source code, but a few are left for you to fix and complete (marked with "**`// Assignment`**" in the source code). You can start by reading the following explanations quickly by skipping the details. Then, study the source code together with the information given below to understand how it works.

1.1 Room Controller

Let us assume that there is a room with a lockable door, a red/green signal light, a fan for ventilation, a temperature sensor, a heater, a cooler, a humidity sensor, a warning light for low humidity, and a warning light for high humidity. Assume that there is a room controller which reads status of the lock, the fan speed switch, the air conditioning (AC) switch, the temperature, and the humidity. Depending on these, the embedded system that works as room controller sends proper commands to the signal light, fan, heater, cooler, and humidity warning lights. It also logs the readout values and the outputs.

1.1.1 Data Acquisition and Format

All the input and sensor data are connected to pins (on the Integrated Circuit (IC)) that can be read collectively from a single input register. Such input registers are often physically read-only. It will appear as these data are concatenated in one unsigned integer variable:



Letter "X" implies that these other bits are don't-care and i_n represents $n-th$ bit in input register, I, which is shown above. Although the input data bits come in a variable, their meaning and values must be interpreted as follows:

Status of lock is captured by bit i_0 as:

Table [1]

i_0	Status
0	Door is unlock
1	Door is locked

Fan speed switch affects bits i_3, i_2, i_1 and should be interpreted as:

$i_3\ i_2\ i_1$	Value set by room user
1 1 1	Off
0 1 1	Low Speed
1 0 1	Medium Speed
0 0 0	Full Speed

Table [2]

AC switch affects bits i_9, …, i_4 and should be interpreted as:

i_4	AC on/off
0	Off
1	On

Table [3]

In this system we will have a desired temperature that room user decides and communicate it to the system through a temperature selector switch. Moreover, the actual room temperature is measured by a temperature system and read by the system.

This particular temperature selector, accepts temperatures from 15.0 ℃ to 25.0 ℃ with a resolution of 0.5 ℃. The binary representation starts at "`00001`":

$i_9\ i_8\ i_7\ i_6\ i_5$	Temperature set by room user
0 0 0 0 1	15.0 ℃
0 0 0 1 0	15.5 ℃
…	…
1 0 1 0 1	25.0 ℃

Table [4]

The above table suggests that the desired temperature can be calculated based on selector value as:

$$temperaturDesired = 14.5 + (i_9 i_8 i_7 i_6 i_5) \times 0.5 \text{ ℃} \quad (1)$$

The room temperature measured by the sensor is captured by i_{17}, …, i_{10} as follows:

$i_{17}\ i_{16}\ i_{15}\ i_{14}\ i_{13}\ .\ i_{12}\ i_{11}\ i_{10}$	Room temperature
0 0 0 0 0 . 0 0 0	0.0 ℃
0 0 0 0 0 . 0 0 1	0.2 ℃
…	…
1 0 0 1 1 . 0 1 1	19.6 ℃
…	…
1 1 1 1 0 . 0 0 0	30.0 ℃
1 1 1 1 1 . 0 0 0	$<$ 0.0 ℃ (for convenience show -1)
1 1 1 1 1 . 1 1 1	$>$ 30.0 ℃ (for convenience show 31)

Table [5]

This particular temperature sensor, reports temperatures in a range between 0.0 ℃ to 30.0 ℃ with a resolution of 0.2 ℃. If the temperature is lower than 0.0 ℃, the sensor reports "`11111000`" and if higher

than 30.0 ℃ the sensor reports "11111111". The above table suggests that the measured temperature (assuming in-range values) can be calculated as:

$$temperaturMeasured = (i_{17}i_{16}i_{15}i_{14}i_{13})\times 1.0\ ℃ + (i_{12}i_{11}i_{10})\times 0.2\ ℃ \quad (2)$$

Note that if the temperature is outside the valid range, the above equation will not work correctly.

The room humidity percentage measured by the sensor is captured by i_{23}, …, i_{18} as follows:

$i_{22}\ i_{21}\ i_{20}\ i_{19}\ i_{18}$	Room humidity	$i_{22}\ i_{21}\ i_{20}\ i_{19}\ i_{18}$	Room humidity
0 0 0 0 0	0 %	…	…
0 0 0 0 1	4 %	1 1 0 1 0	80 %
0 0 0 1 0	8 %	1 1 0 1 1	83 %
0 0 0 1 1	11 %	1 1 1 0 0	87 %
0 0 1 0 0	14 %	1 1 1 0 1	91 %
0 0 1 0 1	17 %	1 1 1 1 0	95 %
…	…	1 1 1 1 1	100 %

Table [6]

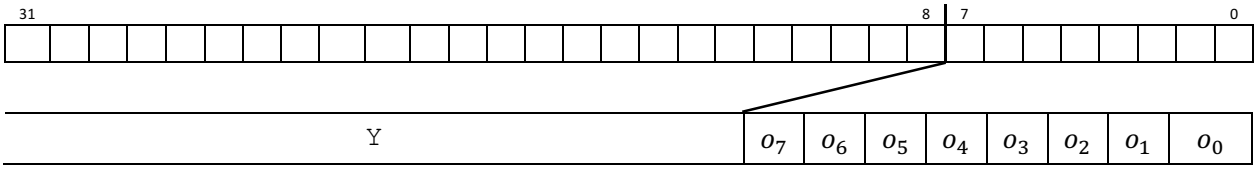
This particular sensor, reports humidity percentage in a range between 0 % to 100 % in a nonlinear manner. The resolution is 3 % in the middle of the range and 4 % for low and high values, except that it takes a 5 % step from 95 % to 100 %. The above table suggests that the measured humidity can be calculated as:

$$humidity = \begin{cases} (i_{22}i_{21}i_{20}i_{19}i_{18})\times 4, & (i_{22}i_{21}i_{20}i_{19}i_{18}) < 3 \\ 8 + (i_{22}i_{21}i_{20}i_{19}i_{18} - 2)\times 3, & 3 \le (i_{22}i_{21}i_{20}i_{19}i_{18}) \le 27 \\ 83 + (i_{22}i_{21}i_{20}i_{19}i_{18} - 27)\times 4, & 28 \le (i_{22}i_{21}i_{20}i_{19}i_{18}) \le 30 \\ 100, & (i_{22}i_{21}i_{20}i_{19}i_{18}) = 31 \end{cases} \quad (3)$$

As implied by letter "X", the rest of the bits (i_{31} … i_{23}) are don't-care. We can ignore them.

1.1.2 Command and Actuation

Depending on the acquired data and their interpretation and information that they provide, a number of commands must be sent to actuators. All these are put together in an integer variable which is then enforced on an output register which is connected to the output pins of the IC.



Letter "Y" implies that these other bits must not be touched and o_n represents $n-th$ bit in the output register, O, which is shown above. Their meaning and values are interpreted by the actuators as follows:

The signal lights' inputs are defined as:

o_1	Red light
0	Off
1	On

o_0	Green light
0	Off
1	On

Table [7]

The fan inputs are defined as:

o_3 o_2	Fan speed
0 0	Off
0 1	Low Speed
1 0	Medium Speed
1 1	Full Speed

Table [8]

The cooler input is defined as:

o_4	Cooler
0	Off
1	On

Table [9]

The heater input is defined as:

o_5	Heater
0	Off
1	On

Table [10]

The humidity warning lights are defined as (note that they are active low):

o_7	Low humidity warning
1	Off
0	On

o_6	High humidity warning
1	Off
0	On

Table [11]

As implied by letter "Y", the rest of the bits (o_{31} … o_8) are don't-touch. These must not be changed, as they might be connected to other devices that we do not want to influence. We can ignore them.

1.1.3 System Operations

Let us use a polling scheme to acquire input data. The main application runs in an infinite loop continuously reading the input register "I" (see section 2.1.1). Based on these inputs and laws described in this section, the outputs are determined and continuously updated by writing into output register "O" (see section 2.1.1). Apart from "O" related information must be logged into a file.

If the door is locked the red light must be on. If the door is unlocked the green light must be on.

The fan speed must be set according to the fan switch. For example if the fan switch indicates the off status ($i_3 i_2 i_1 = 111$), the fan must be turned off by writing the output $o_3 o_2 = 00$. Refer to tables 2 and 8.

The cooler can be turned on only if the AC is turned on by user. If AC is on, the condition for switching the cooler on from an off state is

$$temperaturMeasured > temperaturDesired + 0.3℃\,. \qquad (4)$$

The heater can be turned on only if the AC is turned on by user. If AC is on, the condition for switching the heater on from an off state is

$$temperaturMeasured < temperaturDesired - 0.3℃ . \quad (5)$$

The heater continues to work until it is warmer than or equal to the specified desired temperature. The cooler continues to work until it is colder than or equal to the specified desired temperature. If the AC switch is turned off by the user, both heater and cooler must be turned off.

If the humidity is less than 25% low humidity warning light must be on. If the humidity is more than 60% then the high humidity warning light must be turned on.

1.1.4 Hazardous System Behavior

Embedded systems are often used to govern a system consisting of a number of subsystems. These subsystems must work in coordination in order to assure a safe and trouble-free operation. Therefore, it must be assured that some hazardous situations will never happen. Considering the room controller, what do you think about the following?

- Keeping both cooler and heater on at the same time!
- Keeping both red and green light on!
- Keeping both red and green light off!
- The room temperature is outside the sensor's operational range!
- Keeping both low and high humidity lights on at the same time!

1.1.5 Temperature Control Approach

One of common roles of embedded systems is to work as a controller in a "control theory" sense. In this chapter we used a closed-loop approach to control the temperature. The feedback is provided using the temperature sensor. The controller scheme that we used is a Bang-bang controller, also known as hysteresis controller. A Bang-bang controller is particularly useful here since the only change that we can make is to switch on or off the cooler or the heater.

An open-loop approach would be to switch the heater/cooler on and off regularly without reading the actual temperature. The period and the duty cycle (borrowing terminology from Pulse-Width Modulation) must be calculated and fixed beforehand and will not change when the system is working. For example, for keeping the temperature at $24℃$ the heater is turned on for 1 second and then turned off for 3 seconds. This repeats over and over. This open-loop approach is useful if we assume that almost everything is fixed. For example, the following must not change: (1) room's thermal characteristics, (2) the outside ambient temperature, and (3) heating/cooling output power of the heater/cooler. However, in reality these will change and therefore an open-loop controller will not be able to keep the temperature at the desired level. Examples for factors that affect room's thermal characteristics include number of people in the room as well as the windows being open or closed.

1.2 Work Flow

Copy "room controller" from "skeleton" (/home/TDDI11/lab/skel) directory to your local directory (we assume: userID/TDDI11/ room_controller)

```
cp -r /home/TDDI11/lab/skel/room_controller /home/userID/TDDI11
```

Please note that userID is the user name that you use to login to the lab computers. Now let us change to the new directory and check it:

```
cd /home/userID/TDDI11/room_controller
ls
```

Check to see if “room_controller” directory contains the following:

- main.c
- inputData.txt
- outputLog_Correct.txt
- displayLog_Correct.txt

The source code is in “main.c”. Some inputs for simulation are provided in “inputData.txt”. The correct outputs for these inputs are logged in “outputLog_Correct.txt”. The corresponding display log is in “displayLog_Correct.txt”. In the assignment, you will use “inputData.txt” for simulation and compare your results with correct results that are provided in “outputLog_Correct.txt” and “displayLog_Correct.txt”.

1.2.1 The Source Code

Please note that the area in the source code marked with “**`// Assignment`**” are not correct. These are areas left to be completed as part of the assignment.

The acquisition of inputs is simulated by a function called “readInput”. In an embedded system such a function will read the values from an input register. In this lab, we read from a file, instead. Each line in “inputData.txt” contains a sample which is a simulated readout of the input register. Please ignore this function:

```
unsigned int readInput(unsigned int sampleIndex)
```

The enforcing of output is simulated by a function called “writeOutput”. In an embedded system such a function will write the outputs determined by the program to the output register. In this lab, we write to a file, instead. Each line in “outputLog.txt” contains an output value that is determined based on a corresponding input value. Please ignore this function:

```
int writeOutput(unsigned int outputCurrent, unsigned int sampleIndex)
```

The main part of the code is in an infinite loop. This loops has three main sections: (1) Read inputs and sensor data; then convert them to meaningful information. (2) Decide on outputs. (3) Write outputs.

Extracting Inputs

For extracting and isolating the related bits out of the input register, we use shift and mask operations. Masking means that we change the bits that we do not need to zero while keeping the important bits unchanged.

For example, humidity sensor data in the input variable can be visualized as “`XXXX XXXX XIII IIXX XXXX XXXX XXXX XXXX`” assuming that "`I`" is an input bit from the humidity sensor and "`X`" is a don't-care bit. To extract humidity value, we use a mask that is all 0 but at bits 18 to 22. The mask is “`0000 0000 0111 1100 0000 0000 0000 0000`” in binary. The hex equivalent is `0x007C0000`. We

use bitwise AND operation to filter out all irrelevant bits in the "inputRegister". Bitwise AND operation with 0 (false) results in 0, wiping out the unrelated bits. For example:

```
inputRegister: "XXXX XXXX XIII IIXX XXXX XXXX XXXX XXXX" &
mask:          "0000 0000 0111 1100 0000 0000 0000 0000"
Results in:    "0000 0000 0III II00 0000 0000 0000 0000"
```

Bitwise AND operation with 1 will not change the value of the other bit, keeping the needed data intact.

For example:

```
inputRegister: "XXXX XXXX X101 01XX XXXX XXXX XXXX XXXX" &
Mask:          "0000 0000 0111 1100 0000 0000 0000 0000"
Results in:    "0000 0000 0101 0100 0000 0000 0000 0000"
```

Then we shift the remaining bits to end up with sensor data.

```
humiditySensor = (inputRegister & 0x007C0000) >> 18

For example:   "0000 0000 0101 0100 0000 0000 0000 0000" >> 18
Results in:    "0000 0000 0000 0000 0000 0000 0001 0101"
```

The isolated bits must be then interpreted using tables and equations given in section 2.1. For example "1 0101" is the humidity sensor data value which must be converted to humidity percentage using table 6 and equation 3.

Based on information gained in section (1) of the code, the outputs are determined. For example if it is colder than desired, the heater must be turned on, if it is too dry, the warning light for low humidity must be lit, and so on.

Combining Outputs

Now that the outputs are determined, they must be combined into a single variable. Section (3) in the code combines these to be written to the output register. To combine these, shift and bitwise OR operations are used. For example low humidity warning light control bit (o_7) can be visualized as:

"YYYY YYYY YYYY YYYY YYYY YYYY OYYY YYYY" , assuming that "O" is the output bit and "Y" is a don't-touch bit that its value must not change. The output value "O" must be first shifted:

```
               "0000 0000 0000 0000 0000 0000 0000 000O" << 7
Results in:    "0000 0000 0000 0000 0000 0000 O000 0000"
```

Bitwise OR operation with 0 (false) will not change the value of the other bit, keeping the outputs written by other parts of the program intact. For example:

```
outputRegister:"1001 1001 1001 1001 1001 1001 O101 1001" |
Shifted bit:   "0000 0000 0000 0000 0000 0000 O000 0000"
Result in:     "1001 1001 1001 1001 1001 1001 O101 1001"
```

Since before writing anything else in the output variable, we clear it and since the other parts of the program do not change bits that are not for their use, the bits related to the this part of the program are 0. Therefore, a bitwise OR operation with the determined value will correctly place them in the output variable.

```
outputRegister:"YYYY YYYY YYYY YYYY YYYY YYYY 0YYY YYYY" |
Shifted bit:   "0000 0000 0000 0000 0000 0000 O000 0000"
Result in:     "YYYY YYYY YYYY YYYY YYYY YYYY OYYY YYYY"
```

Pay attention that in this case, as define in table 11, writing 1 actually turns off the light.

Another aspect of bit operations is implementing simple multiplications with shift and summations. On some platforms, this approach might be faster or consumes less power. We utilize the fact that a single shift left is multiplication by 2. For examples: $(a \times 2) = (a << 1)$; $(a \times 3) = ((a << 1) + a)$; $(a \times 4) = (a << 2)$; and so on.

Compile and Run

Compile and run the code with:

```
gcc main.c -std=c99
./a.out > displayLog.txt
```

For debugging purposes, we have the possibility to stop the loop by "getchar()" and see the output on the terminal. Alternatively we can redirect the display output to "displayLog.txt" by "./a.out > displayLog.txt". To be rigorous you can use "Wall", "Wextra", and "pedantic" while compiling.

Please note that the c code can be understood only by referring to tables and equations given in section 2.1.

In each of the loop, a new sample from the input file is read and processed. When we reach to the end of file, the program ceases execution. In reality, it will run as long as the system is powered and "on". There will be no end of file in real life.

1.2.2 Outputs

The input and output register value are not human readable. We can compare the output values with the correct ones in "outputLog_Correct.txt". The terminal log, "displayLog_Correct.txt", is however human readable. The following is displayed for a single input

inputCurrent=246192

inputs: Lock=0, Fan=0, AC=1, Desired=21.000000, Measured=30.000000, Humidity=0

outputs: Green=1, Red=0, Fan=3, Cooler=1, Heater=0, Humidity2H_n=1, Humidity2L_n=0

first line, "inputCurrent", shows the input sample as a decimal value. The interpreted inputs are displayed in the next line. The open locks means green signal as shown in the outputs printed in the third line. It is warmer than desired, therefore, the cooler is on. It is too dry, therefore, dryness warning is on (pay attention that the signal is active low and "`0`" means "`true`").

1.3 Evaluation

1.3.1 Assignments

Compile and run the source code and then compare the results (displayLog) with the correct one. Read and understand the source code. Pay attention that the area marked by "**`// Assignments`**" indicates missing code or incorrect code that you need to update, change, or add new lines to it, in order to correct the program. It is needed to go back and forth between the lab manual section 2.1, the source code, its output, and the correct output.

Identify the correct behavior and correct the source code by adding missing code and correcting the wrong existing code. The corrected code must generate correct outputs similar to the "_Correct" files.

Discussions

Please briefly discuss the answers to the following questions. Write just a few sentences.

1. We discussed combining outputs with shift and bitwise `OR` in section 2.2.1. Can we "add" (+) different outputs instead of performing bitwise `OR` operations (section 3 of code inside the main loop)? Why?

2. (Optional): Do a brief research (perhaps on the web) on Bang-bang (Hysteresis) controllers in relation to equations 4 and 5. Answer the following questions in a few words:

3. (Optional): Which part of equations 4 and 5 represents "hysteresis" in our system? How much is the value of "hysteresis"?

4. (Optional): Why do we need hysteresis in our system? What will happen if its value is zero?

1.3.2 Demonstrations

Run the program and show the code to the lab assistant. Briefly discuss the answers to the above questions.

1.3.3 Deliverables

- The corrected source code
- outputLog.txt
- displayLog.txt
- Answers to the above questions
- Feedback questionnaire

Email them to your lab assistant. Write in the subject: TDDI11 Chapter 2. Make sure that you have used the given inputData.txt and compare your output files against the "_Correct" files.