CSCE 312 Lab manual

Lab-1 - Introduction to Digital System Design

Instructor: Dr. Ki Hwan Yum

Prepared by

Dr. Rabi N Mahapatra.
Suneil Mohan & Amitava Biswas

Fall 2016

Department of Computer Science & Engineering
Texas A&M University

Chapter 1: General lab instructions

1. Learning method:

You will learn by doing exercises and by designing systems. You will also learn "how to learn from additional sources" other than text books. System design is more of an art than science, which is best learned by observing work of other designers and looking over reference designs. Software often documents and reflects the design of the system. For the lab assignments you will be specifically asked to focus on the subset of the topics that you can learn from programs that will be provided to you by the instructors and the TA. However additional topics can be learnt by completely analyzing these given programs and progressively developing advanced skills by adopting and applying those through out the rest of the course. As the lab class proceeds, we will progressively guide you towards those additional topics.

Therefore read the complete programs that are provided even though you may not understand them entirely at this point of time. By doing so you would at least become familiar how code segments and functions look like, and when time permits you should do additional research using "Google" and "man" (from Linux/Unix environment) to understand and rationalize them. Instructions regarding these are provided in the next section. Your level of programming skill and knowledge (and future job prospects) will depend on your motivation to go this extra mile to learn additional topics beyond the immediate need of doing the assignments.

2. Resource that you may need:

The CS help desk web-pages are a great resource. Remember that learning to use these software tools is your responsibility.

1. CS department help pages at https://wiki.cse.tamu.edu/index.php/Special:AllPages
2. How to compile C programs, can be found at – https://wiki.cse.tamu.edu/index.php/Compiling_C_C%2B%2B_UNIX
3. About off campus CS VPN access (for completing assignments from home) - https://wiki.cse.tamu.edu/index.php/Category:Computer_Science_and_Engineering_VPN
4. How to use Linux machine from Windows systems using "putty" and FTP clients - https://wiki.cse.tamu.edu/index.php/Logging_into_UNIX_from_PuTTY https://wiki.cse.tamu.edu/index.php/File_Transfer_Clients
5. To access "man" pages, in the Linux/Unix console you have to type "man" followed by the Linux system call name. For example to learn about the "gettimeofday" system call function, type "man gettimeofday". Remember only the Linux system calls are documented under the linux Manual pages, not arbitrary C functions. You can also find the "man" pages on the web, as they are also published in html format. To find man pages on the web type "man gettimeofday" into Google. Ask your TA and develop an understanding what are system call functions and how are they different from standard C library functions.

5. Specific instructions

1. **Submit** a **soft copy** of all answers that you submit as lab assignments. Hard copies may be requested for specific assignments, you will be informed in this case. The **soft copies** should be submitted online on **CSNET before the deadline**. **For homework assignments, you must work out the problems on paper** and then submit to instructor. The **late penalty policy** is as put in the course syllabus.
2. For each problem, answer all the sub problems under "Activities to do". This should be turned in as a **PDF format**. For some problems you may have to submit text and C program files in addition to the responses to the questions. When submitting a lab assignment, combine all the files as a single zip file and upload the zip file. If you have questions about submission, please talk to your TA.
3. You may use your own machines to compile and test code, however to ensure fair grading, all assignments must be shown to work on the CS department Linux/Unix systems.
4. Assignments might come with associated program code, text, etc. files. Those will be available inside a single zip file meant for that particular lab class. These files will be available in the lab page for downloading. A lab topic might span more than one lab class.

Chapter 2: Introduction to digital system design

In this chapter you learn about importance of low level system design for solving real-life problems. This chapter will immerse you in a problem context, which will require you to apply the theoretical concepts taught in the class. This will help you to develop basic skills needed to become a device level system developers and digital system designer. While going through this chapter concentrate on developing a good understanding of the problem context, as it is a fundamental requirement for becoming a skilled system designer or application developer.

1. Learning duration: 2 weeks. **Required Tools:** gcc

2. Objective:

To learn -

Primary topics

1. How numbers are actually stored in a computer system.
2. How to relate computer programs to a given real-life applications.
3. How to design C code to solve real-life problems using Boolean algebra basics (specifically how to construct the right logical expression and use them).
4. How to optimize code for a real-life embedded system with programming tricks that applies Boolean logic concepts, tools (truth table) and low level C programming features.
5. To appreciate the need to learn more about hardware. This hardware related knowledge is necessary to develop useful systems and real-world applications.
6. How to design digital system especially for safety critical applications.

Secondary topics

7. Useful C functions that will help to understand a platform's data representation.
8. How to use "gcc" compiler in Linux environment.
9. Coding patterns, styles, jargons and terms used in the trade (computing profession).
10. What are standard C library and Linux specific system functions, difference and relationship between them.

3. Exercises to do

3.1 Problem 1:

You are provided with the following C program –

```
#include <stdio.h> //For input/output
#include <sys/time.h> //For gettimeofday() function

int main()
{
	int int_var; //Tag 1

	struct timeval this_instant;
	double time_stamp;

	FILE *my_file_pointer;
	if ( (my_file_pointer = fopen("lab1_prob1_out.txt", "w")) == NULL)
	{
		printf("Error opening the file, so exiting\n");
		exit(1);
	}

	gettimeofday(&this_instant,0);
	time_stamp = this_instant.tv_sec;

	//Code segment for file I/O
	fprintf(my_file_pointer, "This program was executed at time : %d secs\n", time_stamp);

	fprintf(my_file_pointer, "The sizes of different data type for this machine and compiler are -\n");
	fprintf(my_file_pointer, "int data type is %d bytes or %d bits long\n",sizeof(int_var), sizeof(int_var)*8 );
	fprintf(my_file_pointer, "double data type is %d bytes or %d bits long\n",sizeof(double), sizeof(double)*8 );

	//Code segment for console I/O, this can be used instead of the file I/O
	printf("This program was executed at time : %d secs\n", time_stamp);

	printf("The sizes of different data type for this machine and compiler are -\n");
	printf("int data type is %d bytes or %d bits long\n",sizeof(int_var), sizeof(int_var)*8 );  //Tag 2
	printf("double data type is %d bytes or %d bits long\n",sizeof(double), sizeof(double)*8 );
	fclose(my_file_pointer); //To close the output file, mandatory to actually get an output !

	return 0;
`
```

This code (file name "lab1_prob1.c") is available in the zipped package for download from the lab webpage.

Activities to do-

a) Using less than 4 sentences, explain what function/action, the two statements marked with "Tag 1" and "Tag 2" perform.

b) Compile and run this program with "gcc" on the CS Linux machine (`linux.cse.tamu.edu`).

c) Compile and run this program with "gcc" on the CS Sun machine. (`sun.cse.tamu.edu`).

d) When you ran your code, did you get the time executed to be negative? If yes, why did that happen? (Since time cannot be negative). How could you fix this? Figure out how to fix it and do the necessary modifications.

e) Change the data type of the variable time_stamp from `double` to `long int`. Re-run the program on both machines. Is there a change in the values reported? If so, which of the values is the correct value? Why is there a difference?

f) Find out the structure of type "timeval". Is it a standard C data type or is it platform specific?

g) **Submit your output files for all four runs**. (and the fixed versions if you made changes above). *Note: If you submit an output file with negative time, you must submit the corrected version also. Failure to do so will lead to loss of 50% of the points for this problem.*

Hint – See the instructions on how to compile with gcc as indicated under section named "Resource that you may need".

3.2 Problem 2:

Write a C program to get bit and byte lengths for all the following C numerical data types -

`unsigned int, double, long, long long, char, float, struct timeval.`

The C code file should have name "lab1_prob2.c" and it should generate an output file named "lab1_prob2_out.txt".

Activities to do-

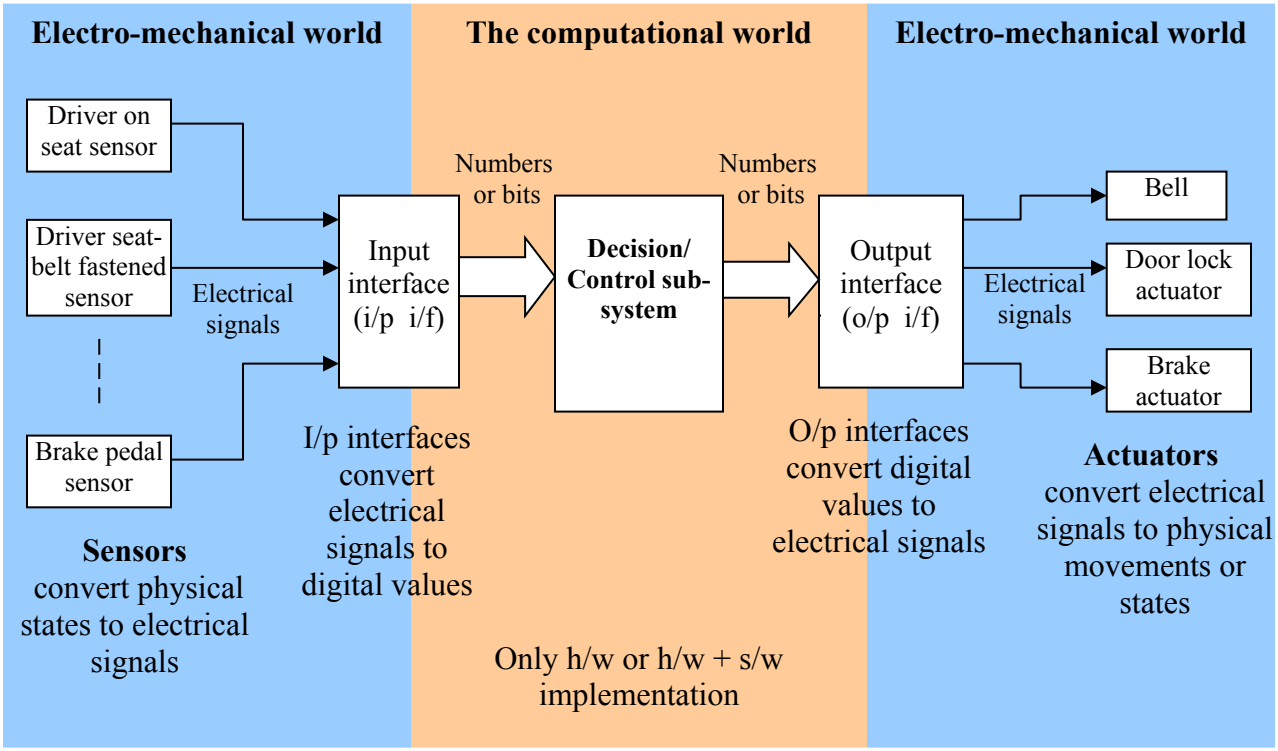
a) Submit your C code file and also your output file.

Hint – You can add codes to the C program given for problem 1 to do the problem 2.

3.3 Problem 3:

Assume you are hired by Ford Motor Company to develop the embedded software for their new model of cars. This software executes on the main computer that sits inside the car. Your manager wrote down the "Software Requirement Specification (SRS)" document and gave it to you. Based on the "requirements" listed in this document you have to develop a C program that materializes these requirements/specifications. Given below is a portion of the SRS document which includes the schematic of the car's electronics and computer systems –

Fig. 1: Block diagram of the embedded control system for the car



Explanation of Fig. 1: The **sensors** convert the physical states (pressure induced by the driver on the seat when he is seating on it, etc.) and mechanical motions (driver pressing the brake pedal) to electrical signals (voltage, current). The **input interface** sub-system (commonly denoted as "i/p i/f", "i/p" for input, "i/f" for interface) converts electrical signal into digital form (number, bits etc.) in the way as explained in during the lecture sessions. (you can find more about this in the suggested text book and the class slides). The **decision/control sub-system** is the main functional block which provides the intelligence to the car so that it can take the right action based on the given situation. The decision logic provides digital output (numbers, bits), which is converted back to electrical signals by the **Output interface** (o/p i/f). These electrical signals are used to activate the electro-mechanical actuators (motors, electro-magnets etc.). For this particular lab class, your job is to **design the decision control logic**. You will design the i/p

i/f and o/p i/f sub-systems in later lab classes. You have choice to use a combination of hard-software (h/w+s/w) or hardware (h/w) alone to design this sub-system.

Eight Available sensors

Each hardware sensor provides a "high" (1) or "low" (0) output. The input interface sub-system sets the value of the corresponding global integer variable to the respective value. The decision/ control logic sub-system will read this integer to take the right decision. It is assumed that this integer is available for C programming. The i/p i/f sub-system keeps monitoring the sensor hardware outputs and takes action (changes value of the integer variable) as soon as the sensor output state changes.

1. **DOS** – driver on seat. This sensor indicates whether a driver is present. This sensor provides logical "high" (1) as output when a person is sitting on the driver's seat and "low" (0) if he is not on the seat.
2. **DSBF** – driver seat belt fastened. This sensor indicates whether the driver seat belt is fastened or not. The sensor hardware provides "high" when driver's seat belt is fastened, "low" otherwise. The corresponding integer variable that reflects the physical states inside the computer and programming world is "driver_seat_belt_fastened". The i/p i/f sub-system sets this integer to 1 when DSBF output is high, and to 0 when DSBF is low, and the decision/control sub-system code reads/uses it.
3. **ER** – engine running. This sensor indicates whether engine is running or not. It provides "true" when engine is running, false otherwise. The corresponding integer variable to read and use is "engine_running".
4. **DC**- doors closed. Indicates whether all doors are closed or not. The corresponding integer is "doors_closed".
5. **KIC** – key in car. Indicates that the keys are still inside the key hole, the corresponding integer variable is "key_in_car".
6. **DLC** – door lock lever. This indicates whether the door lock lever is closed or not. To close the electronic door locks the driver has to close this door lock lever. When the car's computer finds that door lock lever is closed it checks all other variables to asses the situation and finally decides whether to activate the electronic door locks to lock the doors or not. For example if the car keys are still inside but the driver is not on seat (has gone out of car) then the doors should never be locked even though the driver has closed the door lock lever.
7. **BP** – brake pedal. This indicates that the brake pedal is pressed by the driver.
8. **CM** – car moving. This sensor indicates the car is moving and atleast one of its wheels are turning. The corresponding integer variable "car_moving" has value 1 when the car is moving, and has value 0 if the car is not moving.

Three Available actuators

1. **BELL** - A beeper/chime that sounds/plays to alert the driver of any abnormal/hazardous situation (as found in your car). A global integer variable named "bell" is provided in the computer, if your code that implements the decision/control sub-system sets this variable to 1, then the output interface sub-system will read this value and turn on the voltage on the electrical wire that feeds the beeper/chime. As a result the beeper/chime will start beeping. The beeper will stop when the decision/control subsystem code sets the value of "bell" to 0, because then the o/p i/f will turn down the electrical voltage feeding the beeper hence it will stop. The o/p i/f keeps on monitoring the integer variable and takes action (change the voltage) when the integer changes its value.
2. **DLA** – door lock actuator. This actuator locks the doors. A corresponding global integer variable named "door_lock" is provided in the computer, if you set this integer to 1 all the doors are locked, it unlocks all doors when you set the value of "door_lock" to 0.
3. **BA** – brake actuator. This actuator will actually activate the disk brakes in each of the four wheels if the global integer variable "brake" is set to 1. The brake will be released when this variable is set to 0 by the code that implements the decision/control logic sub-system.

Five Requirements

1. The BELL should chime/sound when the driver starts the engine without fastening his seatbelt.
2. The BELL should sound when the driver starts the car without closing all the doors.
3. The BELL should be off as soon as the conditions change to normal.
4. The doors should not lock when the driver has got out of the car but the keys are still inside the engine, even though the driver has closed the door lock lever. **Note**: If the driver is on the seat and requests the doors to be locked, the doors must lock.
5. The brake should be engaged when the driver presses the brake pedal. Brakes should disengage when the brake pedal is released. The brake should engage only when the car is moving, when the car is stationary the brake should not unnecessarily engage to reduce mechanical wear and tear of the brake's hydraulic system.

Activities to do-

a) For each requirement, separately provide **the Boolean expressions** that you decided to use.

b) Create a single combined **truth table** with all the available **sensor inputs and actuator outputs** for all the five requirements together. Some of the truth table entries will be don't-care states (represented by an X) instead of true or false. For this sub-problem assume that these five requirements together constitute a complete system. Note: if you don't use the "don't care" conditions, you could end up with a large 255 row truth table.

c) Write a C program using the Boolean logic concepts and tools learnt in class to materialize these five requirements. Use if-then-else structure to do this**. Submit the C code and your executable.** A basic code framework (file name "lab1_prob3_framework.c") for a general control system has been provided. This is available in the lab files that you downloaded. Use this code to learn how to develop C program for a real-life control system.

d) Submit your version of lab1_probl3.c with the code that you wrote, along with sample input and output. (named: lab1_prob3_input.txt , lab1_prob3_output.txt)

For those who choose to use console instead of file I/O - you can simply copy the input and output traces from Linux console and paste into a text file for submission.

3.4 Problem 4:

Next day after writing the code (for problem 3), you realized that it was a waste of hardware and memory space to use separate integers to represent sensor output and actuator input states. As you get more experienced and confident, you decide to improve your code to save space. You realized that saving space is essential because the cars computer is an "embedded system[1]" (you were quick to pick up the industry jargons) which has very small memory (say only 256 bytes compared to gigabytes of RAM in a traditional desktop system). You decided that all these sensor output states can fit inside a single 32 bit integer, and the actuator output states can be fit into another 32 bit integer. So instead of using 10 integers of 4 bytes each (total 40 bytes), you decided to use only 4 + 4 = 8 bytes. You decided the following integers with following bit formats–

An global integer variable named "sensor_inputs" with the following sensor input encoding format.

Bit 31	Bit 30	…	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
			CM	BP	DLC	KIC	DC	ER	DSBF	DOS

The above table means – when the DOS sensor input is high, the bit position 0 of integer "sensor_inputs" is set to 1 by the i/p i/f sub-system, the bit position is set to 0 when DOS sensor output is low.

Bit 1 corresponds to DSBF sensor input, Bit 2 corresponds to ER output and so on. Reserving a single 32 bit/4 byte integer saves space but it also means you accommodate only 32 sensors (which ok as you are only asked to work with 5 sensors at this moment).

For the actuators you decided to use a global integer variable "actuator_outputs" with the following format -

[1] Search the web to find out what is an embedded system, ask your instructor and TA to explain you anything that you didn't understand from your search on "embedded system".

Bit 31	Bit 30	…	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
								BA	DLA	BELL

The above table means – to activate the beeper/chime your decision/control sub-system logic has to set the bit position 0 of integer "actuator_outputs" to 1, to stop the beeper this bit position 0 should be set to 0. The o/p i/f will then read this bit and excite/ deactivate the corresponding actuator. To activate the door lock the bit position 1 should be set to 1 and to deactivate it should be set to 0 respectively. This scheme can only accommodate 32 actuators.

Hint:

The method of setting individual bits in a given variable is termed bit-masking. **Bit masking** uses a combination of the logical operators of C (& and |) as well as the shift operators << and >> to allow the user to set (turn to 1) or unset (turn to 0) the corresponding bit. Look at the files bitmasking_sample.c and lab1_prob4_hint.c in Lab1Files.zip to see examples of this.

Activities to do-

a) Write a C program for the car control system (as asked by the SRS) which only uses 8 bytes (as explained above). This program should work in exactly the same manner as the code written for Problem-3. You must use **bit-masking** to be able to accomplish this. Use the framework that you used in problem-3 (lab1_prob3_framework.c) as the basis for this program.

   Ref) When you wrote the code for the above activity, you would have written multiple bitmask patterns such as 0x03 or 0x1f. Keeping track of these patterns can become confusing quickly. To help replace such patterns, you could use an **enumerated data-type** and **assign a name to each pattern**. (You could also do the same, using a C macro). Explain in a couple of sentences (with specific references to the code you wrote), where and how *enum* could help improve the readability and maintainability of your code.

3.5 Problem 5:

At the end of the day after re-writing the code (for problem 4), you "released[2]" this code to the system testers. Unfortunately the system tester rejected this nice code on the grounds that it is not safe[3]". You discussed with them to understand why the code is "unsafe". After a long debate you realized that your code does not activate the brakes fast enough. The car - System Test Engineer wants the brakes to activate within 50 nano seconds of pressing the brake pedal. But your

[2] In software engineering, "releasing" a code means giving the final tested version of the software for testing, validation or use by users.

[3] A code is termed safe when it can be used in a safety critical system without any risk of potential damage or injury. Developers have the ethical and professional responsibility to create codes that are safe to use in safety critical systems like cars, elevators, etc. If the car brake doesn't engage within certain time, the car may not stop causing accidents. Hence is the need for safety assessment during testing.

program is unable to respond that fast. So you decided to measure the execution time (or speed) of the code to estimate how much faster it has to run.

Activities to do-

a) **Measure the execution time of the code** that you developed for problem 4 on both the **CS Linux system** (linux.cse.tamu.edu) & the **CS Sun system** (sun.cse.tamu.edu) A execution time measuring framework is provided for this purpose (lab1_prob5_framework.c). Insert your code within this framework to measure time in the place as indicated in the code file.

   i. Submit the modified code file which has both your code segment and time measuring instrumentation portions. (The same file should be run on both machines.)

   ii. Submit the outputs generated by both machines.

Instructions to measure the execution time of your code using the execution time measuring framework file -

   i. Insert your codes in the places as indicated in the "lab1_prob5_framework.c" framework file.

   ii. Compile the "lab1_prob5.c" file using gcc and the realtime library using the *–lrt* option. `[user@linux ~/]gcc file.c -lrt`

   iii. Run the executable created in the previous step.

   iv. The executable will report the execution time of the code.

b) Similarly measure the execution time of the code developed for problem 3.

   i. Compare the execution time for the codes developed for problem 3 and for problem 4.

   ii. Are you convinced that code developed for problem 4 is faster than the code developed for problem 3? Why or why not?