Lab 0

Introduction to C Programming

Authors:

Naveen Shamsudhin, Jonas Lussi, Alexandre Mesot and Prof. Bradley J. Nelson

Multiscale Robotics Lab, Institute of Robotics and Intelligent Systems

Date: 2020

Version: 1.1

Summary: The objective of this lab is to review the basics of the C programming language. We will learn:

- Basic C syntax
- How to compile a program using the GNU C compiler
- Hexadecimal and binary number systems
- Reading user inputs from the terminal
- Writing a function library

0.1 Background

0.1.1 C Language Basics

C will be used extensively throughout this class, and this week will be devoted specifically to the C language. A number of good resources are available for the C programming language:

- Online guide to programming in C (http://www.cs.cf.ac.uk/Dave/C/)
- Numerical Recipes in C (http://apps.nrbook.com/c/index.html)
- Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language (Second Edition)*, Prentice Hall, 1988.
- Gyron S. Gottfried, *Schaum's Outline of Theory and Problems of Programming in C*, McGraw Hill.
- William. H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press.

0.1.1.1 C compilation under Linux

If you are not familiar with C compilation under Unix, take a look at the additional Unix Commands and C Program Compilation material in the Appendix.

0.1.2 Number systems

In our daily life, we represent and work with numbers in a decimal or base-10 numbering system. Computers on the other hand deal with binary numbers (1s and 0s, i.e. ON/OFF states), hence, knowing how to represent numbers in binary (base-2) and converting between different numbering systems is important. Instead of dealing directly with binary numbers, programmers prefer to use hexadecimal (base-16) numbers. The following sections outline procedures for converting from decimals to hex and for representing negative numbers in hex.

0.1.2.1 Binary numbers

If you are not familiar with arithmetic in binary, take a look at the additional Binary Numbers material in the Appendix.

0.1.2.2 Hexadecimal numbers

The procedure for converting decimal integers to two's complement hexadecimal integers is illustrated by an example below.

Example 1: Convert the decimal number +18435 to hexadecimal. From the following table:

```
Quotient        Integer part    Remainder

18435/16            1152            3
 1152/16            72              0
   72/16            4               8
    4/16            0               4
```

Read the reminder column from bottom to top, i.e. 4803. Therefore, 4803 is the hexadecimal representation of the decimal number +18435. In C, hexadecimal numbers are typically represented with a "0x" in front of them. For example 4803 is written as: 0x4803.

Note: This procedure also works when converting to other bases. For example, to convert to binary (base-2) divide by 2 instead of 16 in the quotient column above.

0.1.2.3 Representing negative decimal integers

This involves two steps. First find the hex equivalent of the magnitude of the number. Then find its two's complement. The procedure for finding the two's complement of a number is explained below using an example.

Example 2: Convert the decimal number -18435 to hexadecimal. First, find the hexadecimal value of +18435, then take two's complement of that value. The result is the hexadecimal value of -18435. Note: Taking the two's complement of a number gives the negative of that number. From Example 1, the hexadecimal value of +18435 is 0x4803. Now, take the two's complement of 0x4803.

Convert the hexadecimal number to binary.

```
0x4803 =       4    8    0    3
            (0100 1000 0000 0011) in binary (this conversion only works between hex and binary)
```

Complement the binary number by changing the 1's to 0's and the 0's to 1's:

```
(0100 1000 0000 0011)
(1011 0111 1111 1100)
```

Add one to the complement:

```
 (1011 0111 1111 1100)
+                   1
----------------------
 (1011 0111 1111 1101)
```

Convert back to hexadecimal. Note that the letters A,B,C... stand for 10,11,12.. in hexadecimal format.

```
(1011 0111 1111 1101)
  B    7    F    D = 0xB7FD
```

0xB7FD is the two's complement of 0x4803. Thus, 0xB7FD = -18435. As a check, verify that 0x4803 + 0xB7FD equals zero:

```
  0x4803                 18435
+ 0xB7FD               + (-18435)
---------              ----------
  0x0000                 0x0000
```

When adding two's complement numbers, ignore the remainder of the most significant digit if there is one, i.e., ignore the overflow bit.

Example 3: What is the two's complement of 0xB7FD? This example below shows a faster method of obtaining the two's complement of a hexadecimal number: Subtract each hexadecimal digit from 15.

```
 15      15      15      15
- B     - 7     - F     - D
---     ---     ---     ---
  4       8       0       2
```

Add one:

```
  0x4802
+      1
--------
  0x4803
```

This is the same result as Example 2.

For more information about the 2's complement you can visit:
http://www.mathcs.emory.edu/~cheung/Courses/561/Syllabus/1-Intro/2-data-repr/signed.html

0.1.2.4 Declaring hex numbers in C

Often times you will need to input hexadecimal numbers in C. This is typically used as an intermediate step when inputing binary numbers because hexadecimal is easier for humans to read than binary.

Example 4: How do you write hexadecimal numbers in C?

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

main()
{
    uint16_t a = 0x4803;
    uint16_t b = 0xb7fd;
    uint16_t c;
    c = a + b;
    printf("A =%x, B =%x, C =%x \n",a,b,c);
}
```

The value of "c" should be zero in the output. Here the "%x" command tells the printf statement to print hexadecimal numbers.

Attention:

You may notice that the above section of code does not use the built in data types of C. This is because of some ambiguity in the length of the data types on different systems. For instance, on some machines an `int` is 32-bit while on others it is 64. To avoid this ambiguity, a header named `stdint.h` was introduced by the American National Standards Institute (ANSI) in March 2000. In short, this file makes the following data types available:

Type Declaration	Bits	Type	Range
int8_t	8-bit	integer	-128 – 127
uint8_t	8-bit	unsigned integer	0 – 255
int16_t	16-bit	integer	-32768 – 32767
uint16_t	16-bit	unsigned integer	0 – 65535
int32_t	32-bit	integer	-2,147,483,648 – 2,147,483,647
uint32_t	32-bit	unsigned integer	0 – 4,294,967,295
int64_t	64-bit	integer	-9.22 x 10^18 – 9.22 x 10^18
uint64_t	64-bit	unsigned integer	0 – 1.84 x 10^19

0.1.2.5 Bit-wise operators

Bitwise operation	Symbol
AND	&
OR	|
XOR	^
NOT	~
Shift right	>>
Shift left	<<

These operators are frequently used for manipulating data. We will be using these operators while reading digital IO, so they will be useful later. Also take a look at the section entitles "Formatters for printf() and scanf(), bit operations" in the online tutorial.

"<<" Left shift operator:

```
Assume A is a uint16_t (a 16bit unsigned integer)

A              =   0xa234          =   1010 0010 0011 0100
A = A<<4       =   0x2340          =   0010 0011 0100 0000
A = A<<2       =   0x8d00          =   1000 1101 0000 0000
```

Vacancies in the right most bits are filled with zeroes.

">>" Right shift operator:

```
A              =   0xa234          =   1010 0010 0011 0100
A = A>>4       =   0x0a23          =   0000 1010 0010 0011
```

"&" AND Operator:

```
A   =   0xa2; B = 0x34; C = A&B;

A   =   1010 0010  =   0xa2
B   =   0011 0100  =   0x34
A&B =   0010 0000  =   0x20
```

"|" OR operator:

```
A   =   1010 0010  =   0xa2
B   =   0011 0100  =   0x34
A|B =   1011 0110  =   0xb6
```

0.2 Prelab Procedure

Note: Prelab assignments must be done before reporting to the lab and must be turned in to the lab instructor at the beginning of your lab session. Additionally, you also have to upload your solution as a single PDF-file to moodle. Make sure to upload the file before your lab session starts, late submissions will not be corrected. Since this is the first prelab, each group member has to hand in her/his solution separately.

These are some basic tasks that we ask you to finish before the session in the lab. This will help familiarize you with the material, and speed up progress in the lab. Make sure that your approach is clearly documented. Do not just write the final result. All the prelab tasks are marked with **PreLab Qx**.

1. Select a group for the lab sessions. This can be done on moodle by clicking on "IRM Lab Groups". The groups have different time slots and consist of three people. The time for each group can be displayed by clicking on "Show descriptions".
2. Go through the Appendix.
3. Perform the following calculations. Calculate the sum in hexadecimal numbers. **(PreLab Q1)**

```
 0x0004          0x0034          0x2001          0x3401
+0x0006         +0x0056         +0x35FA         +0x25EE
-------         -------         -------         -------
     ?               ?               ?               ?
```

4. Convert the numbers in the above problem to binary, then calculate the binary sum for each problem. **(PreLab Q2)**
5. Convert the numbers in the above problems to decimal, then calculate the sum for each problem. **(PreLab Q3)**
6. Write a subroutine (function) called `bit_merge`, that accepts two four-digit hexadecimal numbers (16 bits each) as input and combines their bits into a single 32 bit number as follows:

```
void main()
{
    uint16_t hn1 = 0x1234;
    uint16_t hn2 = 0xabcd;
    uint32_t  result;

    result = bit_merge(hn1,hn2);
    printf("\nmerging 0x%x and 0x%x results in 0x%x\n",hn1, hn2, result);
}
```

   The output should look like:

```
merging 0x1234 and 0xabcd results in 0xabcd1234
```

   To do this requires a left shift of hn2 by 16 bits and then a bitwise OR of hn1 with the left-shifted hn2. Remember that hn2 needs to be left shifted into a 32 bit variable, or all the data will be lost when shifted. Be careful of data types, and think about what is going on at a bit level? **(PreLab Q4)**

7. What is the difference between the output of section A and section B? Show your calculations and explain. **(PreLab Q5)**

```
uint16_t i = 25000;
int16_t j = 25000;
int16_t l;
int32_t m;
```

```
/* Section A */
l = i + j;
m = i + j;
printf("%d %d\n",l,m);

/* Section B */
l = i + j;
m = l;
printf("%d %d\n",l,m);
```

8. What is the difference between the output of Section A and Section B? Why? **(PreLab Q6)**

```
int16_t a = 20;
int16_t b = 10;
int16_t c = 100;
int32_t m;

/* Section A */
m = (a*b)/c;
printf("%d\n",m);

/* Section B */
m = a*(b/c);
printf("%d\n",m);
```

9. At the end of the main function, what are the values of i,j,k and l? For more information on pointers in C, refer to the Appendix. **(PreLab Q7)**

```
int16_t f1(int16_t j)
{
    j = 5;
    return j*j;
}

int16_t f2(int16_t *i)
{
    *i = 6;
    return *i;
}

main()
{
    int16_t i=0;
    int16_t j=1;
    int16_t k,l;

    k = f1(i);
    l = f2(&j)+ k;
}
```

10. Read through the entire lab procedure. If you don't understand something, prepare questions for the lab assistants.

0.3 Lab procedure

Note: The PostLab needs to be done and handed in as a group. The relevant PostLab tasks are marked with **PostLab Qx**.

0.3.1 Creating your first program

We will initially be working mainly in the console mode of the system. This is to help familiarize you with the underlying modes of operation of this system. Once you better understand working in Unix on the command line, the GUI level is much easier to learn independently.

Once you log in, you can find the terminal icon on the desktop. You can open the console window and you'll find a command line prompt that should look something like this:

```
ubuntu@udooubuntu:~$
```

Now we will create a directory to work in. Type the following commands:

```
ubuntu@udooubuntu:$ cd ~/Desktop
ubuntu@udooubuntu:~/Desktop$ mkdir irm
ubuntu@udooubuntu:~/Desktop$ cd irm
ubuntu@udooubuntu:~/Desktop/irm$ mkdir lab00
ubuntu@udooubuntu:~/Desktop/irm$ cd lab00
```

With these commands first we change into the `Desktop` directory.

Hint: The folder `Desktop` is located in the `/home/` directory, the `/` at the beginning is for that. Also, the part up to the semi-colon does not have to be the same on your system. That part tells us the respective combination of `username` `computername` and will be different according to your system setup. The part between the semi-colon and the `$` character tells you your current directory.

After changing to that directory, we create a subdirectory called `irm` and change into it. We finally create the directory `lab00` and change into it. If you know the number of items in a series, it is usually good to pad the single digit entries with 0s to make the structure more readable in the future.

Attention:

It is often good to build a hierarchical directory system to keep your files better organized. This makes it much easier to navigate through your files in the future. Also, starting from Lab01, it will be essential that the directories are labelled according to our instructions for the program files to work properly.

Now, we will create our first C program using Geany. Geany is a basic Integrated Development Environment (IDE) featuring many useful tools for programming and debugging on a single window. Geany contains a typical text editor that can automatically highlight the syntax of your code and a terminal for compiling, running, and interacting with it. To start Geany, you can go through the programs menu (click on the Udoo logo, located on the top left corner) and find it in "Programming", or use the terminal as follows

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ geany hello_world.c
```

You will notice that when the program starts, you are unable to use the original console. This is because Geany has control of the console until it finishes. Alternatively, you may start Geany like

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ geany hello_world.c &
```

The & symbol indicates that the program should run in the background and you will be able to type commands again. Now type in the sample program which just outputs "Hello world!" to the console.

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

main()
```

```
{
    printf("Hello world!\n");
}
```

To save the file, go to the file menu and click "Save". To compile this code, you can use the built in terminal located among the vertical tabs at the bottom section of the Geany window. On the terminal, first thing you have to make sure is being in the folder that contains the code that you have written. To make sure type

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ ls
```

This will list the files in the current directory of the terminal. If you can see "hello_world.c", you can go ahead; otherwise, you should navigate to the directory that you have created in previous steps. In any case, typing

```
ubuntu@udooubuntu:$ cd ~/Desktop/irm/lab00
```

will place you into your directory. Once you make sure that you are in the directory of your code file, type

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ gcc hello_world.c
```

This will create an executable called `a.out`. To run it, type

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ ./a.out
```

The ./ is necessary to tell the system that we are running the program in the current directory. `a.out` is the filename that gcc assigns a program when no other filename is provided. This is not a very good approach though, because if you have multiple programs named `a.out` you will never know which is which. To have gcc assign a different filename, type:

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ gcc -o hello_world hello_world.c
```

Now, although this allows us to create a program with the correct filename, it is too cumbersome to continually compile our program in this manner. To alleviate this problem, a tool called `make` is used. We will only be making a simple example and do not need to concern ourselves too heavily with the syntax right now. If you would like more information, we do have a section on Makefiles in the Appendix. In it's simplest form, a Makefile has the following syntax:

```
target: source file(s)
     command (must be preceded by a tab)
```

The files for this lab can be found on moodle. Copy these files to your local folder:

```
/Desktop/irm/lab00
```

For this lab, a Makefile has already been created. Since we will be creating our own function library, we need to link files. The Makefile already contains commands to compile the function library (`functions.c` and `functions.h`) and the program `sum_numbers` and to link them. For the mean time, start Geany to edit the Makefile (run `geany Makefile`). Modify the Makefile to also compile your program `hello_world.c`. Since we do not need the function library for the program `hello_world.c`, we do not need to link it to `functions.o`. Make a new entry in the Makefile for your program. The Makefile should look something like this:

```
# when running make, all programs are generated
all: sum_numbers hello

# compile hello_world.c and make an executable program
```

```
hello:
     gcc -o hello_world hello_world.c

# linking of sum_numbers.o with functions.o
sum_numbers: sum_numbers.o functions.o
     gcc sum_numbers.o functions.o -o sum_numbers

# compile sum_numbers.c
sum_numbers.o: sum_numbers.c
     gcc -c sum_numbers.c

# compile functions.c
functions.o: functions.h functions.c
     gcc -c functions.c

# remove generated files and programs
clean:
     rm functions.o sum_numbers.o sum_numbers hello_world
```

Make sure to use a tab before gcc and rm because this is the syntax of the Makefile. You will get an error if you do not do this. This will create two executable programs called hello_world and sum_numbers, and provisions for removing them. When you run this, it should look like the following:

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ make
gcc -c sum_numbers.c
gcc -c functions.c
gcc sum_numbers.o functions.o -o sum_numbers
gcc -o hello_world hello_world.c
```

To remove the compiled files, type:

```
ubuntu@udooubuntu:~/Desktop/irm/lab00 $ make clean
rm functions.o sum_numbers.o sum_numbers hello
```

You have now written and compiled your first program in Unix.

0.3.2 Function library

It will be necessary to make new entries in the makefile. Make sure to link files correctly and to also compile the source files. In this lab we will write our own function library. This way we can easily reuse the functions in multiple programs by simply including the header file `functions.h` in the source file of the programs (e.g `sum_numbers.c`).

1. Write a function to output the individual bits of a 16 bit variable. Every four bits should be separated by a space. Refer to Declaring hex numbers in C if you need help with the syntax. The function should be defined as:

   ```
   void print_bits(uint16_t arg_word);
   ```

   If `arg_word` is 0x0123, the output should look similar to:

   ```
   hex: 0x0123, bin: 0000 0001 0010 0011
   ```

   The functions has already been declared in the header file `functions.h`. Write the implementation in the corresponding source file `functions.c`. **(PostLab Q1)**

2. Write a program (`sum_numbers`) to check the addition of hexadecimal numbers that you have done in the prelab. As input it should take 2 hexadecimal numbers and output the sum printed in hexadecimal and binary formats (use the function `print_bits`). The source file has already been created for you and it is named `sum_numbers.c`. Write your code in this file. The Makefile already includes commands to compile and link `sum_numbers`, so you do not need to change the Makefile. **(PostLab Q2)**

3. Implement the bit merging function from the Prelab Procedure. The function needs to be implemented in the source file `functions.c`. **(PostLab Q3)**

4. Implement a program (`manipulate_two_numbers`) to read two numbers from the terminal and output the merged result in hexadecimal (use function `bit_merge`), and the sum in hexadecimal and binary formats (use function `print_bits`). The output should look as follows:

```
merging 0x1234 and 0x2343 results in 0x23431234
the sum is hex: 0x3577, bin: 0011 0101 0111 0111
```

   Write your code in the file `manipulate_two_numbers.c`. Here you need to make a new entry to the Makefile in order to generate an executable program. **(PostLab Q4)**

   Hint: This is pretty easy. Assume we would like to print three variables `int var1=0x0001, var2=0x0abc, var3=0x8767;` on the console, comma separated in hexadecimal format, `printf("%x,%x,%x",var1,var2,var3);` would be the command to use, and the output would be `0x0001,0x0abc,0x8767`. Similarly, when we wish to read in three values from the console in the same format we would use `scanf("%x,%x,%x",&var1,&var2,&var3);`. The reason why we used `&` is to give the function `scanf` the addresses of the variables to manipulate and the associated format specifier `%x` tells the number of bits to manipulate at that address, once it gets the value from the user (for more information on this, please refer to the introductory chapter on pointers in the appendix). Just one more thing, since the format specifier- `%x` in this case- specifies the number of bits, you might want to use `%hx` to tell that you are expecting 16-bit values. To ease the learning you can think of the `scanf` as almost exactly the same as the `printf`, **just don't forget the address operators!!!!!** As a general advice, placing a `printf` before a `scanf` to explain the number of inputs and their limits, ideally providing a sample input, is a good practice.

5. **Bonus:** Doing the previous steps, you build yourself a simple calculator capable of merging and summing two hexadecimal numbers and outputting the result in binary and/or hexadecimal formats. Now, what is not so good about this is that, everytime you want to perform the operation you need to run the code from the scratch. An ideal program would be started for once and would keep performing its task until it is explicitly told to quit. In this case, modify the program from the previous step to run until the user inputs a predefined pair of numbers. This task will only add to your total points if you reached less than maximum amount **(PostLab Q5 - Bonus)**

   Hint: You can use `while(1)` to enter an infinite loop. In the loop, check the inputs and upon receiving the predefined pair stop the loop with `break;`.

0.4 Postlab and lab report

- Show your working programs to the teaching assistant.
- Upload a single PDF-file with your solution to moodle. The file should contain the code you wrote to solve tasks 1, 2, 3, 4 and eventually 5 (PostLab Q1 - Q5). In particular, make sure to include the files `functions.c`, `functions.h`, `sum_numbers.c`, `manipulate_two_numbers.c`, your source file from task 5 (if you did it in a separate file) and also the Makefile. Please upload your solution in time (before next lab session), late submissions will not be corrected.
- Print the PDF-file and hand it in at the beginning of the next lab session.
- Come prepared with the prelab procedure for the next lab.