Experiment 2

C Language Programming

During the infancy years of microprocessor based systems, programs were developed using assemblers and fused into the EPROMs. There used to be no mechanism to find what the program was doing. LEDs, switches, etc. were used to check correct execution of the program. Some 'very fortunate' developers had In-circuit Simulators (ICEs), but they were too costly and were not quite reliable as well. As time progressed, use of microprocessor-specific assembly-only as the programming language reduced and embedded systems moved onto C as the embedded programming language of choice. C is the most widely used programming language for embedded processors/controllers. Assembly is also used but mainly to implement those portions of the code where very high timing accuracy, code size efficiency, etc. are prime requirements.

Embedded Systems Programming

Embedded programs must work closely with the specialized components and custom circuitry that makes up the hardware. Unlike programming on top of a full-function operating system, where the hardware details are removed as much as possible from the programmer's notice and control, most embedded programming acts directly with and on the hardware. This includes not only the hardware of the CPU, but also the hardware which makes up all the peripherals (both on-chip and off-chip) of the system. Thus an embedded programmer must have a good knowledge of hardware, at least as it pertains to writing software that correctly interfaces with and manipulates that hardware. This knowledge will often extend to specifying key components of the hardware (microcontroller, memory devices, I/O devices, etc.) and in smaller organizations will often go as far as designing and laying out the hardware as a printed circuit board. An embedded programmer will also need to have a good understanding of debugging equipment such as multimeters, oscilloscopes and logic analyzers.

Another difference from more general purpose computers is that most embedded systems are quite limited as compared to the former. The microcomputers used in embedded systems may have program memory sizes of a few thousand to a few hundred thousand bytes rather than the gigabytes in the desktop machine, and will typically have even less data (RAM) memory than program memory.

There are many factors to consider when developing a program for embedded systems. Some of them are:

- Efficiency - Programs must be as short as possible and memory must be used efficiently.
- Speed - Programs must run as fast as possible.

- Ease of implementation.
- Maintainability
- Readability

C Programming Language for Embedded Systems

Embedded systems are commonly programmed using C, assembly and BASIC. C is a very flexible and powerful programming language, yet it is small and fairly simple to learn. C gives embedded programmers an extraordinary degree of direct hardware control without sacrificing the benefits of high-level languages so its compilers are available for almost every processor in use today and there is a very large body of experienced C programmers.

C used for embedded systems is slightly different as compared to C used for general purpose programming (under a PC platform). Programs that are developed for embedded systems are usually expected to monitor and control external devices and directly manipulate and use the internal architecture of the processor such as interrupt handling, timers, serial communications and other available features. C compilers for embedded systems must provide ways to examine and utilize various features of the microcontroller's internal and external architecture; this includes interrupt service routines, reading from and writing to internal and external memories, bit manipulation, implementation of timers/counters and examination of internal registers etc. Standard C compiler, communicates with the hardware components via the operating system of the machine but the C compiler for the embedded system must communicate directly with the processor and its components. For example, consider the following C language statements:

```
printf("C Programming for Embedded Systems\n");
c = getchar();
```

In standard C running on a PC platform, the *printf* statement causes the string inside the quotation to be displayed on the screen. The same statement in an embedded system causes the string to be transmitted via the serial port pin (i.e., TxD) of the microcontroller provided the serial port has been initialized and enabled. Similarly, in standard C running on a PC platform *getchar()* causes a character to be read from the keyboard on a PC. In an embedded system the instruction causes a character to be read from the serial pin (i.e., RxD) of the microcontroller.

Template for Embedded C Program

```
#include "lm4f120h5qr.h"

void main(void)
{
// body of the program goes here
}
```

Template for Embedded C Program

- The first line of the template is the C directive. This tells the compiler that during compilation, it should look into this file for symbols not defined within the program.
- The next line in the template declares the beginning of the body of the main part of the program. The main part of the program is treated as any other function in C program. Every C program should have a main function.
- Within the curly brackets you write the code for the application you are developing.

Preprocessor Directives

Preprocessor directives are lines included in the code of our programs that are not program statements but directives for the preprocessor. Preprocessor directives begin with a hash symbol (#) in the first column. As the name implies, preprocessor commands are processed first.i.e., the compiler parses through the program handling the preprocessor directives.

These preprocessor directives extend only across a single line of code. As soon as a newline character is found, the preprocessor directive is considered to end. No semicolon (;) is expected at the end of a preprocessor directive. The only way a preprocessor directive can extend through more than one line is by preceding the newline character at the end of the line by a backslash (\).

The preprocessor provides the ability for the inclusion of header files, macro expansions, conditional compilation, and line control etc. Here we discuss only two important preprocessor directives:

Macro Definitions (#define)

A macro is a fragment of code which has been given a name. Whenever the name is used, it is replaced by the contents of the macro. There are two kinds of macros. Object-like macros resemble data objects when used, function-like macros resemble function calls.

Object-like Macros

An object-like macro is a simple identifier which will be replaced by a code fragment. It is called object-like because it looks like a data object in code that uses it. They are most commonly

used to give symbolic names to numeric constants. To define preprocessor macros we can use #define. Its format is:

```
#define identifier replacement
```

Syntax to define macros in C

When the preprocessor encounters this directive, it replaces any occurrence of *identifier* in the rest of the code by *replacement*. This *replacement* can be an expression, a statement, a block or simply anything. The preprocessor does not understand C, it simply replaces any occurrence of *identifier* by *replacement*.

```
#define DELAY 20000
```

Wherever, DELAY is found as a token, it is replaced with 20000.

Function-like Macros

Macros can also be defined which look like a function call. These are called function-like macros. To define a function-like macro, the same '#define' directive is used, but with a pair of parentheses immediately after the macro name. For example:

```
#define SUM(a, b, c)  a + b + c
#define SQR(c)  ((c) * (c))
```

Advantages Of Using A Macro

- The speed of the execution of the program is the major advantage of using a macro.
- It saves a lot of time that is spent by the compiler for invoking / calling the functions.
- It reduces the length of the program

Including Files (#include)

It is used to insert the contents of another file into the source code of the current file. There are two slightly different ways to specify a file to be included:

```
#include "filename"
```

Syntax to define macros in C

```
#include "lm4f120h5qr.h"
```

This include directive will include the file named "lm4f120h5qr" at this point in the program. This file will define all the I/O port names for LM4F120 microcontroller.

Structures

Structures provide a way of storing many different values in variables of potentially different types under the same name. This makes it a more modular program, which is easier to modify because its design makes things more compact. Structs are generally useful whenever a lot of data needs to be grouped together–for instance, they can be used to hold records from a database or to store information about contacts in an address book. In the contacts example, a struct could be used that would hold all of the information about a single contact–name, address, phone number, and so forth.

Defining a Structure

```
struct [structure tag]
{
   member definition;
   member definition;
   ...
   member definition;
} [one or more structure variables];
```

Syntax to define structure in C

The structure tag is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

Example 2.1.

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the member access operator (.). The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use struct keyword to define variables of structure type. Following is the example to explain usage of structure:

Example 2.2.

```
struct Books
{
   char  title[50];
   char  author[50];
   char  subject[100];
   int   book_id;
};

int main( )
{
   struct Books Book1;        /* Declare Book1 of type Book */

   strcpy( Book1.title , "C Programming");
   strcpy( Book1.author , "Richard C. Dorf");
   strcpy( Book1.subject , "C Programming Tutorial");
   Book1.book_id = 6495407;


   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming
Book 1 author : Richard C. Dorf
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
```

Type Casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a long value into a simple integer then you can type cast long to int. You can convert values from one type to another explicitly using the cast operator as follows:

```
(type_name) expression
```

Syntax to typecast a variable

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation:

Example 2.3.

```
#include <stdio.h>

main()
{
   int sum = 17, count = 5;
   double mean;

   mean = (double) sum / count;
   printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value. Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Variable Type	Keyword	Bytes Required	Range
Character	char	1	-128 to 127
Unsigned Character	unsigned char	1	0 to 255
Integer	int	Depends upon architecture	
Short Integer	short int	2	-32678 to 32767
Long integer	long int	4	-2147483648 to 2147483647
Unsigned Integer	unsigned int	2	0 to 65535
Unsigned Short Integer	unsigned short int	2	0 to 65535
Unsigned Long Integer	unsigned long int	4	0 to 4294967295
Float	float	4	1.2E-38 to 3.4E38
Double	double	8	2.2E-308 to 1.8E308
Long Double	long double	10	3.4E-4932 to 1.1E+4932

Type Specifiers

Type Qualifiers

Although the idea of const has been borrowed from C++. Let us get one thing straight: the concepts of const and volatile are completely independent. A common misconception is to imagine that somehow const is the opposite of volatile and vice versa. The table 2.1 provides a list of various type specifiers along with information regarding their memory allotment. Notice that the keywords 'volatile' and 'const' are not present. This is because these are type qualifiers and not type specifiers. We will now look at these two type qualifiers in some detail now.

Const Keyword

A data object that is declared with const as a part of its type specification must not be assigned to, in any way, during the run of a program. It is very likely that the definition of the object will contain an initializer (otherwise, since you can't assign to it, how would it ever get a value?), but this is not always the case. For example, if you were accessing a hardware port at a fixed memory address and promised only to read from it, then it would be declared to be const but not initialized.

Volatile Keyword

The reason for having this type qualifier is mainly to do with the problems that are encountered in real-time or embedded systems programming using C. What volatile keyword does is that it tells the compiler that the object is subject to sudden change for reasons which cannot be predicted from a study of the program itself, and forces every reference to such an object to be a genuine reference. It is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time - without any action being taken by the code the compiler finds nearby.

To declare a variable volatile, include the keyword volatile before or after the data type in the variable definition. For instance, both of these declarations will declare foo to be a volatile integer:

```
volatile int foo;
int volatile foo;
```

Now, it turns out that pointers to volatile variables are very common, especially with memory-mapped I/O registers. Both of these declarations declare pReg to be a pointer to a volatile unsigned 8-bit integer:

```
volatile uint8_t * pReg;
uint8_t volatile * pReg;
```

Proper Use of Volatile

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

- Memory-mapped peripheral registers
- Global variables modified by an interrupt service routine
- Global variables accessed by multiple tasks within a multi-threaded application

Pointers

Pointers are extremely powerful programming tool. They can make some things much easier, help improve your program's efficiency, and even allow you to handle unlimited amounts of

data. For example, using pointers is one way to have a function modify a variable passed to it. It is also possible to use pointers to dynamically allocate memory, which means that you can write programs that can handle nearly unlimited amounts of data on the fly - you don't need to know, when you write the program, how much memory you need. As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

Example 2.4.

```
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

How to declare a pointer?

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration:

```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to use Pointers?

There are few important operations, which we will do with the help of pointers very frequently. (a) we define a pointer variable (b) assign the address of a variable to a pointer and (c) finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations:

Example 2.5.

```c
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Double Pointers

Double pointer is a pointer to a pointer. In general, pointers store the address of a variable. Whereas pointer to pointer is a pointer stores the address of another pointer, and this second pointer has the address of the variable. Double pointers are declared as:

```c
int     **ip;    /* pointer to a pointer to an integer */
```

Consider the following example.

Example 2.6.

```c
#include <stdio.h>

int main ()
{
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   int ** ptr;  /* double pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/
   ptr = &ip;  /* store address of ip in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   /* access the value using double pointer */
   printf("Value of **ptr variable: %d\n", **ptr );

   return 0;
}
```

When the above code is compiled and executed, it produces result something as follows:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
Value of **ptr variable: 20
```

Functions

A function is a group of statements that together perform a task. Every C program has at least one function which is main(), and all the most trivial programs can define additional functions. You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division usually is so each function performs a specific task. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. A function is known with various names like a method or a sub-routine or a procedure etc.

The general form of a function definition in C programming language is as follows:

```
return_type function_name( parameter list )
{
   body of the function
}
```

Syntax to define functions in C

Following is the source code for a function called max(). This function takes two parameters num1 and num2 and returns the maximum between the two:

Example 2.7.

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{   /* local variable declaration */
   int result;

   if (num1 > num2)
      result = num1;
   else
      result = num2;

   return result;
}
```

Passing Arguments by Value

By default, arguments in C are passed by value. When arguments are passed by value, a copy of the argument is passed to the function. Therefore, changes made to the formal parameter by the called function have no effect on the corresponding actual parameter. Consider the following code:

Example 2.8.

```
#include <stdio.h>

void foo(int y)
{
    printf("y = %d\n", y);
}

int main()
{
    foo(5); // first call

    int x = 6;
    foo(x); // second call
    foo(x+1); // third call
```

```
    return 0;
}
```

In the first call to foo(), the argument is the literal 5. When foo() is called, variable y is created, and the value of 5 is copied into y. Variable y is then destroyed when foo() ends. In the second call to foo(), the argument is the variable x. x is evaluated to produce the value 6. When foo() is called for the second time, variable y is created again, and the value of 6 is copied into y. Variable y is then destroyed when foo() ends. In the third call to foo(), the argument is the expression x+1. x+1 is evaluated to produce the value 7, which is passed to variable y. Variable y is once again destroyed when foo() ends. Thus, this program prints:

```
y = 5
y = 6
y = 7
```

Because a copy of the argument is passed to the function, the original argument can not be modified by the function. This is shown in the following example:

Example 2.9.

```
#include <stdio.h>

void foo(int y)
{
    printf("y = %d\n", y);

    y = 6;
    printf("y = %d\n", y);
} // y is destroyed here

int main()
{
    int x = 5;
    printf("x = %d\n", x);

    foo(x);
    printf("x = %d\n", x);

    return 0;
}
```

The output of the program is:

```
x = 5
y = 5
y = 6
x = 5
```

At first, x is 5. When foo() is called, the value of x (5) is passed to variable y inside foo(). y is assigned the value of 6, and then destroyed. The value of x is unchanged, even though y was changed.

Arguments passed by value can be variables (e.g., x), literals (e.g., 6) or expressions (e.g., x+1). There are two reasons to call a function by value: side effects and privacy. Unwanted side effects are usually caused by inadvertent changes made to a call by reference parameter. Mostly data is required to be private and, if allowed, only someone calling the function is permitted to change it. However, passing large structures or data by value can take a lot of time and memory to copy, and this can cause a performance penalty, especially if the function is called many times. So, it is better to use a call by value by default and only use call by reference if data changes are expected.

Passing Arguments by Reference

When passing arguments by value, the only way to return a value back to the caller is using the return statement. While this is suitable in many cases, there are a few cases where better options are available. One such case is when writing a function that needs to modify the values of an array (e.g., sorting an array). In this case, it is more efficient and more clear to have the function modify the actual array passed to it, rather than trying to return something back to the caller. One way to allow functions to modify the value of argument is by using pass by reference. In pass by reference, we declare the function parameters as references rather than normal variables:

```
void AddOne(int *y)
{
    *y = *y + 1;
}
```

When the function is called, y will become a reference to the argument. Since a reference to a variable is treated exactly the same as the variable itself, any changes made to the reference are passed through to the argument! The following example shows this in action:

Example 2.10.

```
#include <stdio.h>

void foo(int *y)
{
    printf("y = %d\n", *y);

    *y = 6;
    printf("y = %d\n", *y);
}

int main()
{
```

```
    int x = 5;
    printf("x = %d\n", x);

    foo(&x);
    printf("x = %d\n", x);

    return 0;
}
```

This program is the same as the one we used for the pass by value example, except foo's parameter is now a reference instead of a normal variable. When we call foo(&x), *y becomes a reference to x. This example produces the output:

```
x = 5
y = 5
y = 6
x = 6
```

Note that the value of x was changed by the function.

Bitwise Operations in C

The byte is the lowest level at which we can access data; there's no "bit" type, and we can't ask for an individual bit. In fact, we can't even perform operations on a single bit – every bitwise operator will be applied to, at a minimum, an entire byte at a time. This means we will be considering the whole representation of a number whenever we talk about applying a bitwise operator. Table 2.2 summarizes the bitwise operators available in C.

Operator	Description	Example	Result
&	Bitwise AND	0x88 & 0x0F	0x08
$\wedge$	Bitwise XOR	0x0F $\wedge$ 0xFF	0xF0
|	Bitwise OR	0xCC | 0x0F	0xCF
$\ll$	Left Shift	0x01 $\ll$ 4	0x10
$\gg$	Right Shift	0x80 $\gg$ 6	0x02

Bitwise Operators in C

Bit Masking

Bitwise operators treat every bit in a word as a Boolean (two-value) variable, apply a column-wise Boolean operator, and generate a result. Unlike binary math, there is no carry or borrow and every column is independent.

Turning Off Bits

Bit masking is using the bits in one word to "mask off" or select part of the range of bits in another word, using the bitwise Boolean AND operator. The 1 bits in the "mask" select

which bits we want to keep in the other word, and the zero bits in the mask turn all the other corresponding bits to zeroes. In other words, the 1 bits are the "holes" in the mask that let the corresponding bits in the other word flow through to the result.

Turning On Bits

The opposite of masking (turning off bits) is "ORing in" bits, where we use the bitwise Boolean OR to "turn on" one or more bits in a word. We select a value that, when ORed with some other value, "turns on" selected bits and leaves the other bits unchanged.

Toggling Bits

Sometimes it does not really matter what the value is, but it must be made the opposite of what it currently is. This can be achieved using the XOR (Exclusive OR) operation. XOR returns 1 if and only if an odd number of bits are 1. Therefore, if two corresponding bits are 1, the result will be a 0, but if only one of them is 1, the result will be 1. Therefore inversion of the values of bits is done by XORing them with a 1. If the original bit was 1, it returns 1 XOR 1 = 0. If the original bit was 0 it returns 0 XOR 1 = 1. Also note that XOR masking is bit-safe, meaning that it will not affect unmasked bits because Y XOR 0 = Y, just like an OR.