Introduction to C programming for C8051F120 mixed signal processor

Sometimes when more complex algorithms need to be implemented in real-time application, assembler language programming can lead to cryptic and difficult to understand programs. A lot of programmers nowadays tend to use high-level languages rather than assembler, to program microcontrollers or digital signal processors. Although programming in high-level language is generally much easier, this commodity does not come for free. Programs written in high-level language are larger so more program memory might be needed to store them. They can also be less efficient when it comes to the speed of execution – a high-level language program is typically slower compared to a similar program written in assembler language. Most often used high-level language for microcontroller programming is C language. One of the advantages of using C language compared to some other high-level languages is that various C compilers, i.e. programs that translate C programs into machine code, are usually very efficient. A lot of manufacturers claim their compiler to be 80% efficient. This basically means that the corresponding assembler language program will only be 20% smaller in size on average compared to our program written in C. This trade-off between the ease of programming and the program size is usually acceptable in most situations, although you might find in practice this figure of 80% efficiency is difficult to achieve. C language efficiency is usually somewhere in the 50-70% region.

This section will introduce some basics of C programming for C8051F120 mixed signal processor. Even though C is a standard programming language, there are some specific issues to consider when programming any processor in C. For a more general and detailed treatment of C language, the student is advised to have a look at some standard C books, while the main aim of this handout remains the introduction of basics of C language. Enough information will be given to start with some simple projects like the one given in the first laboratory exercise in this course. Students are then expected to build their knowledge further as they attempt more and more complex projects and start to work on their coursework project.

Template C program for PIC16F27A

C is a function-based language and in fact, any C program itself is merely a function. Functions are sections of code that perform a single, well-defined task. Functions are the C equivalent to assembly-level subroutines. A function in C can take any number of parameters and return a maximum of one value. A parameter (or argument) is a value passed to a function that is used to alter its operation or indicate the extent of its operation. Function consists of a name followed by the parentheses enclosing arguments, or a keyword “void” if the function requires no arguments. If there are several arguments to be passed, the arguments are separated by commas. A simple C function is detailed in Figure 1.

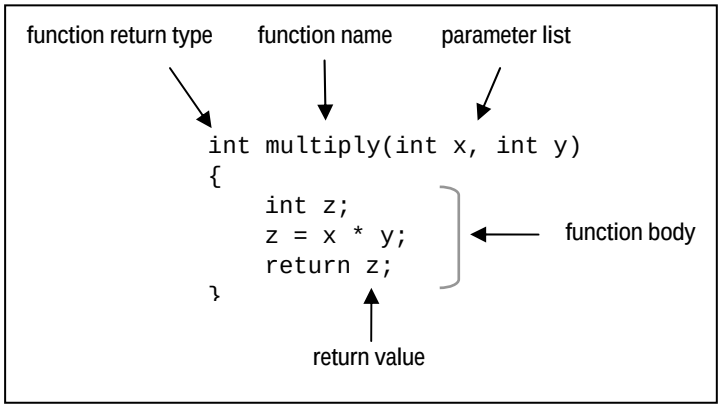

Figure 1. Simple C function

Every C program must at least have a function named ‘main’, and this function is the one that executes first when the program is run. We can here write the simplest possible version of the main function that can run on the PIC16F27A microcontroller. Program given in Listing 1 will do almost nothing – it finishes as soon as it starts.

However, it represents a useful template for further development of more complex C programs on our PIC microcontroller.

Listing 1. Template C program for C8051F120 microcontroller

```
1 // template C program – does nothing useful – starts and stops
2 #include <c8051f120.h>
3 void main(void)
4 {
5
6 // body of the program goes here
7
8 }
```

The second line in the Listing 1 is the so called include statement. This line tells the C compiler that during the compilation it should look into header file (i.e. c8051f120.h) for any symbols not defined within the program. This file holds definitions of all SFRs in the microprocessor and their addresses. Different models of 8051 processor usually require different .h header files with definitions specific for that particular device. Various C compiler manufacturers usually provide their own versions of each header file, specific and compatible with their own compiler. Care therefore needs to be taken to include proper .h file in your code.

Line 3 from the Listing 1 officially declares the main function. The prefix void in this line means that this function returns no argument and the second void means that the function also requires no arguments to be supplied to it. This is almost always the case with 8051 C programs, so you will use this construction very often when programming it using C language. Line 4 announces the beginning of the main function and line 8 is the end of the function. C functions are always enclosed in curly braces and the main function is no exception. The fact that there is no real code between those two braces only means that this C program will do nothing useful – it will start and soon after that it will end. Lines 1 and 6 are program comments.

Everything starting with a double forward slash is a program comment. Similarly to comments in assembler language programs comments in C programs are ignored by C compiler and are used to clarify some sections of the code to whoever needs to read, use or modify the program.

The following C program is slightly more meaningful.

Listing 2. Simple C program

```
1 // this program disables the watchdog timer and stops
2 #include <c8051f120.h>
3 void main(void)
4 {
5
6    WDTCN = 0xDE;                    // disable watchdog timer
7    WDTCN = 0xAD;
8
9 }
```

This program will disable the watchdog timer in C8051F120. A watchdog timer (WDT) is a timer within the C8051F120 which runs off the system clock. An overflow of the WDT forces the C8051F120 into the reset state. Before the WDT overflows, the application program must restart it. WDT is useful in preventing the system from running out of control, especially in critical applications. If the system experiences a software or hardware malfunction which prevents the software from restarting the WDT, the WDT will overflow and cause a reset. After a reset, the WDT is automatically enabled and starts running at the default maximum time interval which is 524 ms for a 2 MHz system clock. The WDT may be enabled or disabled by software. To disable the watchdog timer a sequence of two instructions, from the above listing is used.

The body of the main program in the above listing consists of just those two statements. The common way to terminate statements in C code is to use semi-column at the end of each line, as it has been done with statements from lines 6 and 7 (there are some exceptions not discussed here).

To complete this brief review of C functions we can rewrite the program from the Listing 2 by introducing a new function called wdt_off. That function will have a task of disabling the watchdog timer. This is of course not the very clever thing to do – program itself is so short that there is no point in splitting it even further. We just do it here in order to demonstrate how a C function can be called by the main program, or how in fact to call a function from any other function in C language program.

Lisitng 3. Calling another function from the main function in C

```
1 // this program disables the watchdog timer and stops
2 // it calls a function wdt_off to accomplish this task
3 #include <c8051f120.h>
4 void main(void)
5 {
6    wdt_off();
7 }
8
9 void wdt_off(void)
10 {
11     WDTCN = 0xDE;                 // disable watchdog timer
12   WDTCN = 0xAD;
13}
```

In the program from the Lisitng 3, the main function contains just one statement. It is a call to the function myfunc which contains a further two statement that can disable the watchdog timer.

Variables

Variables in C are data objects that may change in value during the program execution. Variables must be declared immediately after the curly bracket marking the beginning of a function by specifying the name and data type of the variable. Variable names in C may consist of a single letter or a combination of a number of letters and numbers. Usually up to 52 characters can be used for the variable name. Starting variable name with a number is not permitted in C and spaces and punctuation are not allowed as part of a variable name. C is a case sensitive language and two variables 'A' and 'a' are therefore treated as two separate and different variables. In C, a data type defines the way the program stores information in memory. Basic data types with corresponding specifiers (proper ways of declaring data types in C), allocation sizes and ranges are given in Table 1 below. You should understand that this is not a universal table and it only applies to our C8051F120 model. It might also change slightly depending on the C compiler. Table 1 describes the data types for C51 compiler available from Silicon Laboratories and designed specifically for C8051F120 model.

Table 1. Basic C data types

INTEGER DATA TYPES

Data Type	Bits	Bytes	Value Range
signed char	8	1	-128 to +127
unsigned char	8	1	0 to 255
signed int	16	2	-32768 to +32767
unsigned int	16	2	0 to 65535
signed long	32	4	-2147483648 to 2147483647
unsigned long	32	4	0 to 4294967295

FLOAT DATA TYPES

Data Type	Bits	Bytes	Value Range
float	32	4	±1.175494E-38 to ±3.402823E+38

SPECIAL DATA TYPES

Data Type	Bits	Bytes	Value Range
bit	1	-	0 to 1

sbit	1	-	0 to 1
sfr	8	1	0 to 255
sfr16	16	2	0 to 65535

Special data types available on C8051F120 include bit-valued, bit-addressable and special function registers type data.

Bit-valued data and bit-addressable data must be stored in the bit-addressable memory space on the C8051F120. This means that bit- valued data and bit-addressable data must be labelled as such using the bit, sbit and bdata label.

Bit-addressable data must be identified with the bdata language extension:

```
1 int bdata X;
```

The integer variable X declared above is bit-addressable. Any bit valued data must be given the bit data type, this is not a standard C data type:

```
1 bit flag;
```

The bit-valued data flag is declared as above. The sbit data type is used to declare variables that access a particular bit field of a previously declared bit-addressable variable.

```
1 bdata X;
2 sbit X7flag = X^7;      // bit 7 of X
```

X7flag declared above is a variable that references bit 7 of the integer variable X. You cannot declare a bit pointer or an array of bits. The bit valued data segment is 16 bytes or 128 bits in size, so this limits the amount of bit-valued data that a program can use.

As can be seen in the include files c8051f020.h, the special function registers are declared as a sfr data type. The value in the declaration specifies the memory location of the register:

```
1 sfr P0 = 0x80;
2 sfr P1 = 0x90;
```

Extensions of the 8051 often have the low byte of a 16 bit register preceding the high byte. In this scenario it is possible to declare a 16 bit special function register, sfr16, giving the address of the low byte:

```
1 sfr16 TMR3RL = 0x92;    // Timer3 reload value
2 sfr16 TMR3 = 0x94;      // Timer3 counter
```

The memory location of the register used in the declaration must be a constant rather than a variable or expression.

Definition of type of a variable in a C program is an important factor in the efficiency of a program. Depending on the type of a variable the compiler will reserve a memory space sufficient to save that variable. Larger data type will usually mean longer computation time so if the speed is more important than range you should try and make most of the variables in your program of 'char' type. While doing this you need to be careful and keep in mind the range of the 'char' type variables. If, for example the 'unsigned char' variable exceeds 0-255 range your program will give an incorrect results. Since most of the registers on C8051F120 are 8 bit registers, char is usually sufficient for most variables in the program. To give you an idea how to declare your data in C programs some examples are given below:

```
1  int goals;           // integer variable
2  float x,y,z;         // three float variables
3  char p = 0xFF;       // char variable is declared and
4                       // initialised at the same time
5  double f = 56.3;     // double variable
```

1.1.1 Arrays

Arrays in C are specific data structures used to store multiple variables of the same data type. An array of 10 character variables named A can be defined as:

```
char A[10];
```

The above declaration actually declares the array with 10 elements: A[0], A[1], A[2], ... , A[9], where the value within brackets is called subscript. In the C language the array subscript starts from 0, so to access all elements of the array A, subscript needs to take values from 0 to 9. Accessing A[10] is not legal, and is a fruitful source of error for many programmers!

If we want to assign the initial values to elements of an array, we need to enter these values between curly braces:

```
char days[7] = {1, 2, 3, 4, 5, 6, 7};
```

To access any value from the array days, we need to use [ ] separators.

```
Tuesday = days[1];     // second value from the array days is assigned
                       // to a variable named Tuesday
Sunday = days[6];      // seventh value from the array days is
                       // assigned to variable named Sunday
Monday = days[7];      // error!!!, days[7] does not exist
                       // some random value will be assigned
                       // to a variable named Monday
```

Constants

C also allows declaration of constants. When you declare a constant, it is a bit like a variable declaration except the value cannot be changed later in the program. The attribute 'const' is used to declare constant as shown below:

```
int const a = 1;
const int b = 4;
```

C Operators

C has a full set of arithmetic, relational and logical operators. It also has some useful operators for direct bit-level manipulations on data, which makes it similar to assembler language. Most important operators are listed in Table 2. Those operators can be used to form expressions. Mathematical operators follow standard precedence rules – multiplication and division will be executed before any addition or subtraction. Sub-expressions surrounded with parentheses have the highest precedence and are always evaluated first. Statements may optionally be grouped inside pairs of curly braces { }, and as such are called compound statements.

Table 2. Common C operators

	Operator	Action	Example
Assignment Operators	=	Assignment	x = y;
Mathematical Operators	+	Addition	x = x + y;
	-	Subtraction	x = x - y;
	*	Multiplication	x = x * y;
	/	Division	x = x / y;
	%	Modulus	x = x % y;
Logical Operators	&&	Logical AND	x = true && false;
	||	Logical OR	x = true || false;
	&	Bitwise AND	x = x & 0xFF;
	|	Bitwise OR	x = x | 0xFF;
	~	Bitwise NOT	x = ~x;
	!	Logical NOT	false = !true
	>>	Shift bits right	x = x >> 1;
	<<	Shift bits left	x = x << 2;

Equality Operators	==	Equal to	if(x == 10) {…}
	!=	Not equal to	if(x != 10) {…}
	<	Less than	if(x < 10) {…}
	>	Greater than	if(x > 10) {…}
	<=	Less then or equal to	if(x <= 10) {…}
	>=	Greater than or equal to	if(x >= 10) {…}

A common mistake in writing the C statements is to use assignment operator '=' instead of '==' equality operator 'i = j' and 'i == j' are both perfectly legal C statements but the first one will copy the value of variable j into i while the second compares the values of two statements and results in logical value ('1' if those two variables are equal, '0' if i and j are different).

Conditional statements and iteration

To control the flow of execution in C program we usually use a conditional 'if' statement as well as 'for' and 'while' loops.

The 'if' statement is used to allow decisions to be made about parts of the program that will be executed depending on some conditions tested by the 'if' statement in the program. If the condition given to the 'if' statement is true then a section of code is executed as outlined below.

```
if(condition)
{
     execute this code if condition is true
}
```

Example of simple usage of 'if' statement is given below:

```
if(x<0)
{
     x = -1 * x;
}
```

Sometimes you might want to perform one action when the condition is true and another action when the condition is false. Here, an 'else' statement needs to be combined with the 'if' statement:

```
if(condition)
{
     execute this code if condition is true
}
else
{
     execute this code if condition is false
}
```

It is easy to chain a lot of 'if-else' statements:

```
if(condition 1)
{
     execute this code if condition 1 is true
}
else if(condition 2)
{
     execute this code if condition 2 is true and condition 1 is false
}
else   // optional
{
     execute this code if condition 1 and condition 2 are false
}
```

It is also possible to nest 'if' statements:

```
if(condition 1)
     if(condition 2)
     {
```

```
       execute this code if condition 1 and condition 2 are true
    }
else
{
    execute this code if condition 1 or condition 2 are false
}
```

While the 'if' statement allows branching in the program flow, 'for', 'while' and 'do' statements allow the repeated execution of code in 'loops'. It has been proven that the only loop needed for programming is the 'while' loop but sometimes it is more convenient to use 'for' loop instead. While the condition specified in the 'while' statement is true a statement or group of statements (compound statements) are executed as outlined below.

```
while(condition)
{
    execute this code while condition is true
}
```

Example of simple usage of 'while' statement is given below:

```
while(x>0)
{
    result = result * x;
    x = x – 1;
}
```

The other type of loop is the 'for' loop. It takes three parameters: the starting number, the test condition that determines when the loop stops and the increment expression.

```
for(initial; condition; adjust)
{
    code to be executed while the condition is true
}
```

Example of simple usage of 'for' statement is given below:

```
for(counter = 1; counter <= x; counter = counter +1)
{
    result = result * x;
}
```

EXAMPLE:

What would each of two C functions defined below return if called from the main program with the lines:

```
k = compl1(9);
m = compl2(9);
```

Variable "k" is declared at the beginning of the main program as unsigned char type while variable "m" is defined as signed char type.

```
unsigned char compl1(unsigned char i)
{
      unsigned char j;
      j = ~i;
      return j;
}
```

```
signed char compl2(signed char i)
{
     signed char j;
     j = ~i;
     return j;
}
```

Function **`compl1()`** calculates the "bitwise not" of the argument passed to the function. Both, argument and returned value are of unsigned char type so,

k = ~k = ~(9) = ~(0x09) = ~(0b00001001) = 0b11110110 = 0xF6 = 246

Function **`compl2()`** also calculates the "bitwise not" of the argument passed to the function but argument and returned value of this function are of signed char type so,

m = ~m = ~(9) = ~(0x09) = ~(0b00001001) = 0b11110110 = 0xF6 = -10