INTRODUCTION TO EMBEDDED C

By
D.BALAKRISHNA,

Research Assistant, IIIT-H

CHAPTER 1: ASSEMBLY Vs HIGH LEVEL LANGUAGE:

1.1 ASSEMBLY LANGUAGE:

- The computation in assembly language program is less than machine language.
- This program runs faster to produce the desired result as compared to high lavel language.
- This program contains more instruction than High level language.
- The program written on one computer can't be used on any other computer.
- Assembly language is a low level language. Programmers of assembly language must be well versed in architecture of the processor for which that particular program is written.
- For example, an assembly program written for Intel 8080 will not run on Zilog Z80.
- Assemblers are dependent on the instruction sets of a particular microprocessor.

1.2 HIGH LEVEL LANGUAGE:

- The computation in high level language program is less than assembly language.
- This program runs slower to produce the desired result as compared to Assembly language.
- This program Contains less instruction than Assembly Language.
- The program written on one computer can be used on other computer.
- High level languages are nearer to human languages.
- They are independent of the instruction sets of the particular processor.
- A program in Basic written for Intel 8080 will also run on Zilog Z80.
- The operating system will take care of the job of converting the basic program in to the machine code of that particular processor.

1.3 ADVANTAGE OF USING HIGH LEVEL LANGUAGE (HLL) FOR PROGRAMMING:

- Code reusability- a function or routine can be repeatedly used in a program.
- Standard library functions─ for examples, the mathematical functions and delay ( ), wait ( ), sleep ( ) functions.
- Use of the modular building blocks.

CHAPTER 2: EMBEDDED C:

2.1 Why use C?

- Procedure oriented language (No objects).
- Provision of inserting the assembly language codes in between (called in- line assembly) to obtain a direct hardware control.
- It is a 'mid-level', with 'high-level' features (such as support for functions and modules), and 'low-level' features (such as good access to hardware via pointers).
- Can control many machine level functions without resorting to assembly language.
- Application can be written in C more easily than assembly language because the development software manages the details because of modularity; reusable codes can be developed and maintained.
- The programmer need not be very thorough with the architecture of the processor.
- Code developed in C is more portable.
- It is very efficient.
- It is popular and well understood.
- Even desktop developers who have used only Java or C++ can soon understand C syntax.
- Good, well-proven compilers are available for every embedded processor (8-bit to 32-bit or more).
- Books, training courses, code samples and WWW sites discussing the use of the language are all widely available.

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

Example:

```
printf(" C - Programming for 8051\n");
```

In standard C running on a PC platform, the 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.

2.1.1 Language Extensions:

C51 provides number of extensions of ANSI standard C. Most of these provide direct support for elements of the 8051 architecture.

C51 includes extensions for

- Memory types and memory areas on the 8051.
- Memory models.
- Memory type specifiers.
- Bit variables and bit addressable data.
- Special function registers.
- Pointers.
- Function attributes

2.2 MEMORY MODALS:

Small:

- All variables default to the internal data memory of the 8051.
- Same as if they were declared.
- Explicitly using the data memory type specifier variable access is very efficient.
- All data objects, as well as the stack must fit into the internal RAM.

Compact:

- All variables default to one page of external data memory.
- Same as if they were explicitly declared using the pdata memory type specifier.
- Can accommodate a maximum of 256 bytes of variables as it accessed indirectly through R0 and R1.
- Variable access is not as fast as small model.

Large:

- All variables default to external data memory.
- Same as if they were explicitly declared using the xdata memory type specifier.
- Memory access through this data pointer is inefficient, especially for variables with a length of two or more bytes.
- This type of data access generates more code than the small or compact models.

2.3 DIRECTIVES:

C compilers incorporate a pre-processing phase that alters the source code in various ways before passing it on for compiling. Four capabilities are provided by this facility in C.

They are:

- Inclusion of text from other files
- Macro processing
- Conditional compiling
- In-line assembly language

Inclusion of text from other files:

Header, configuration and other available source files are made the part of an embedded system program source file by this directive.

The pre-processor also recognizes directives to include source code from other files. The two directives

```
#include "Filename"

#include <Filename>
```

The **<filename>** version will search for the file in the standard include directory, while the **"filename"** version will search for the file in the same directory as the original source file.

Example:

```
# include "VxWorks.h" //Include VxWorks functions

# include "semLib.h" //Include Semaphore functions Library

# include "taskLib.h" //Include multitasking functions Library
```

Macro Processing:

We use macros for three reasons.

1) To save time we can define a macro for long sequences that we will need to repeat many times.

2) To clarify the meaning of the software we can define a macro giving a symbolic name to a hard-to-understand sequence. The I/O port #define macros are good examples of this reason.

3) To make the software easy to change, we can define a macro such that changing the macro definition, automatically updates the entire software.

```
#define Name CharacterString
```

Global Variables — `# define volatile boolean IntrEnable`

Constants — `# define false 0`

Strings— `# define welcomemsg "Welcome To ABC Telecom"`

Conditional Compiling:

This pre-processing feature lets us designate parts of a program which may or may not be compiled depending on whether or not certain symbols have been defined. In this way it is possible to write into a program optional features which are chosen for inclusion or exclusion by simply adding or removing #define directives at the beginning of the program.

When the pre-processor encounters

```
#ifdef    Name
```

It looks to see if the designated name has been defined. If not, it throws away the following source lines until it finds a matching

```
#else
```

(or)

```
#endif
```

directive.

Nesting of these directives is allowed; and there is no limit on the depth of nesting. It is possible, for instance, to write something like

```
#ifdef ABC      /* ABC */
#ifndef DEF     /* ABC and not DEF */
#else           /* ABC and DEF */
#endif          /* ABC */
#else           /* not ABC */
#ifdef HIJ      /* not ABC but HIJ */
#endif          /* not ABC */
#endif
```

Interrupt Software:

We use the interrupt handler pragma to specify a function as an interrupt handler. The compiler will then use the RTI instruction rather than the RTS instruction to return from ExtHan.

```
#pragma interrupt_handler ExtHan()
void ExtHan(void)
{
     KWIFJ=0x80; // clear flag
     PutFifo(PORTJ&0x7F);
}
```

2.4 DATA TYPES:

A good understanding of C data types for 8051 can help programmers to create smaller hex files.

- unsigned char
- signed char
- unsigned int
- signed int

- sbit (single bit)
- bit and sfr

unsigned char:

- It is the most natural choice.
- 8-bit data type in the range of 0 – 255 (00 – FFH).
- C compilers use the **signed** char as the **default** if we do not put the keyword unsigned.

Example:

Write an 8051 C program to send values 00 – FF to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
      unsigned char z;
      for (z=0;z<=255;z++)
            P1=z;
}
```

signed char:

8-bit data type.

- Use the MSB D7 to represent – or +
- Give us values from –128 to +127

We should stick with the unsigned char unless the data needs to be represented as signed numbers.

Ex: Temperature

Example:

Write an 8051 C program to send values of –4 to +4 to port P1.

Solution:

```
#include <reg51.h>
void main(void)
{
      char mynum[]={+1,-1,+2,-2,+3,-3,+4,-4};
      unsigned char z;
      for (z=0;z<=8;z++)
            P1=mynum[z];
}
```

unsigned int:

- It is a 16-bit data type.
- Takes a value in the range of 0 to 65535 (0000 – FFFFH).
- Define 16-bit variables such as memory addresses.
- Set counter values of more than 256.
- Since registers and memory accesses are in 8-bit chunks, the misuse of int variables will result in a larger hex file.

Signed int:

- It is a 16-bit data type.
- Use the MSB D15 to represent – or +.
- We have 15 bits for the magnitude of the number from –32768 to +32767.

Sbit:

sbit keyword allows access to the single bits of the SFR registers .

Example:

Write an 8051 C program to toggle bit D0 of the port P1 (P1.0) 50,000 times.

Solution:

```
#include <reg51.h>
```

```
sbit MYBIT=P1^0;

void main(void)

{

      unsigned int z;

      for (z=0;z<=50000;z++)

      {

            MYBIT=0;

            MYBIT=1;

      }

}
```

bit and sfr :

The bit data type allows access to single bits of bit-addressable memory spaces 20 – 2FH.

To access the byte-size SFR registers, we use the sfr data type.

Data Type	Size in Bits	Data Range/Usage
unsigned char	8-bit	0 to 255
signed char	8-bit	-128 to +127
unsigned int	16-bits	0 to 65535
signed int	16-bits	-32768 to +32767
sbit	1-bit	SFR bit-addressable only
bit	1-bit	RAM bit-addressable only
sfr	8-bit	RAM addresses 80 – FFH only

2.5 MODIFIERS:

auto:

- Modifier 'auto' or No modifier, if outside a function block, means that there is ROM allocation for the variable by the locator if it is initialised in the program. RAM is allocated by the locator, if it is not initialised in the program.
- Modifier 'auto' or No modifier, if inside the function block, means there is ROM allocation for the variable by the locator if it is initialised in the program. There is no RAM allocation by the locator.

unsigned:

Modifier 'unsigned' is modifier for a short or int or long data type. It is a directive to permit only the positive values of 16, 32 or 64 bits, respectively.

static:

Modifier 'static' declaration is **inside a function block**. Static declaration is a directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it. It then does not save on a stack on context switching to another task. When several tasks are executed in cooperation, the declaration static helps.

The private declaration here means that there are no other instances of that method in any other object. It then does not save on the stack. There is ROM allocation by the locator if it is initialised in the program.

There is RAM allocation by the locator if it is not initialised in the program.

Modifier static declaration is **outside a function block**. It is not usable outside the class in which declared or outside the module in which declared.

There is ROM allocation by the locator for the function codes.

const:

Modifier const declaration is outside a function block. It must be initialised by a program.

Example:

```
#define const Welcome_Message "There is a mail for you"
```

There is ROM allocation by the locator.

register:

Modifier register declaration is inside a function block. It must be initialised by a program.

Example:

`'register CX'`. A CPU register is temporarily allocated when needed.

There is no ROM or RAM allocation.

interrupt:

It directs the compiler to save all processor registers on entry to the function codes and restore them on return from that function. [This modifier is prefixed by an underscore, `'_interrupt'` in certain compilers.]

extern:

Modifier extern. It directs the compiler to look for the data type declaration or the function in a module other than the one currently in use.

volatile:

Modifier volatile outside a function block is a warning to the compiler that an event can change its value or that its change represents an event. An event example is an interrupt event, hardware event or inter-task communication event.

volatile static:

Modifier volatile static declaration is inside a function block.

Examples:

- `'volatile static boolean RTIEnable = true;'`
- `'volatile static boolean RTISWTEnable;'`
- `'volatile static boolean RTCSWT_F;'`

The static declaration is for the directive to the compiler that the variable should be accessible outside the function block also, and there is to be a reserved memory space for it; and volatile means a directive not to optimise as an event can modify. It then does not save on the stack on context switching to another task.

When several tasks are executed in cooperation, the declaration static helps. The compiler does not optimise the code due to declaration volatile. There is no ROM or RAM allocation by the locator.

2.6 MEMORY TYPES:

code:

- Program (code) memory
- Addresses are from 0x0000 to 0xFFFF
- 8051— 64 kB of code memory

data:

- Directly addressable internal RAM memory for data
- Gives fast access 8051—128 byte of data memory

idata:

- Indirectly addressable internal RAM memory for data

- 8051/52— has full internal space 256 byte of idata memory

bdata:

- Bit addressable internal RAM memory for data
- 8051—128 bits data memory (16 bytes between 0x20 and 0x2F byte addresses)

xdata:

- Indirectly addressable external RAM memory for data
- 8051— full internal space 64 kB byte of external data memory

pdata:

- Paged external data memory of 256 bytes
- Accessed by simple instruction MOVX @R1 or MOVX @R2.

2.7 INFINITE LOOPS:

Infinite loops- Never desired in usual programming. Why? The program will never end and never exit or proceed further to the codes after the loop.

Infinite loop is a feature in embedded system programming!

Example:

A telephone is never switching off. The system software in the telephone has to be always in a waiting loop that finds the ring on the line. An exit from the loop will make the system hardware redundant.

Embedded system might be required to continuously execute a section of a program indefinitely.

To achieve this indefinite loop (loop without any exit condition) can be used the statement that performs this is:

```
for(;;)
{

}
```

Part of the program to be repeated indefinitely must then be placed in between the curly brackets after the `for(;;)` statement.

The "super loop" software architecture:

Example:

```
void main(void)
{
    /* Prepare for task X */
    X_Init();
    while(1) /* 'for ever' (Super Loop) */
    {
        X(); /* Perform the task */
    }
}
```

Crucially, the 'super loop', or 'endless loop', is required because we have no operating system to return to, our application will keep looping until the system power is removed.

Strengths:

- The main strength of Super Loop systems is their simplicity. This makes them (comparatively) easy to build, debug, test and maintain.
- Super Loops are highly efficient: they have minimal hardware resource implications.
- Super Loops are highly portable.

Drawbacks:

- This framework will not provide the accuracy or flexibility you require.
- Operates at 'full power' (normal operating mode) at all times. Impacts on system power consumption.

2.8 TIME DELAYS:

For various reasons it might be necessary to include some sort of time delay routine in most of the embedded system programs.

One simple approach (not involving timers) is to let the processor count for a while before it continues.

Example:

```
for(j=0; j<=255; j=j+1)
{
      ;
}
```

Once the loop counter reaches the value of 255 program will exit the loop and continue execution with the first statement following the loop section.

For a longer delays we can use a nested loop structure, i.e. Loop within the loop:

Example:

```
for(i=0; i<=255; i=i+1)
{
      for(j=0; j<=255; j=j+1)
      {
            ;
      }
}
```

Note that with this structure the program counts to 255 x 255.