12 Embedded C

12.1 Introduction

C is a high level language, developed in the 1970's at Bell Labs. Used in developing the Unix operating system, it was optimized for system development. Unix was made readily available for the academic community. Along with that, C came along and became very popular. This original version is popularly known as the "K&R" version, after its authors, Brian Ritchie and Kernighan. As various vendors implemented the language, they added in vendor specific extensions, to correct deficiencies in the language. This had the effect of losing the advantages of portability. The American National Standards Institute (ANSI) formed a committee to standardize the language and include in many new features. This was finalized in 1989 and this was known as ANSI C. From that time, the standardization task took on a global nature under the International Standards Organization and this was known as Standard C. Much of this effort involves providing support for multinational applications. Of course, there is now C++ and Java, later developments.

12.1.1 Advantages

Portability: same program source can run across variety of processors - 4/8/16/32 bit with little modifications. Quick upgrade without much s/w change!

Efficient: program code can be as small or fast as an assembler program

Productivity: As in all high level languages, details of memory spaces, hardware resources are taken care of by compiler. For example, there is no need to worry about location of data, whether ROM or RAM.
Concentrate on problem, not on machine characteristics!
May be possible to write program on PC, test using PC compilers, then bring over to target processor.

12.1.2 Disadvantages

cryptic: program can be hard to read (often called a write-only language!) many non intuitive operators and their order of evaluation can cause confusion

protection: indiscriminate use and management of pointers can corrupt system very easily

debug: difficult to on development tools that use assembler only

12.2 Using C on an embedded system

Various standards groups like ANSI have specified how the C language is to be used. These specifications are oriented towards desktop computers, workstations and larger systems. In these systems, processor storage consists of RAM for the operating system and user applications. This is available in terms of several megabytes. Secondary storage runs into the thousands of megabytes. The processor runs at several hundred Millions of Instructions per Second (MIPs).

12.2.1 Cross compilers

Modern C compilers which run on desktop machines, but which produce code for another processor are called cross compilers. Cross compilers for embedded systems provide various modifications to standard C called extensions This helps in using the processor more efficiently. They also provide various header and include files to access processor resources more easily, like predefined register names.

This may also make the program less portable. But the objective is to use the lowest cost processor for the project. However the programming job may overall be more efficient than using assembler.

12.2.2 Design constraints for embedded systems

A typical embedded system has much more modest resources. For lower end processors, a few kilobytes of nonvolatile program memory, a few hundred bytes of RAM and a processor speed of up to 1 MIP. Of course, there are embedded systems which have as much computing power and resources as a desktop the only difference being the lack of a keyboard, screen and mouse.

In order to make best use of these resources, the programmer must take note of the machine architecture when writing programs. The embedded system typically has a few kilobytes of ROM for program, but more crucial, only a few hundred bytes of RAM on chip. If we exceed this, extra memory chips have to be added to the hardware and possibly decoding hardware. This also increases the size of the printed circuit board. For a mass produced device, this can drive up the cost significantly. The overall product may be bigger as well. The goal here is to use all the on-chip facilities as efficiently as possible.

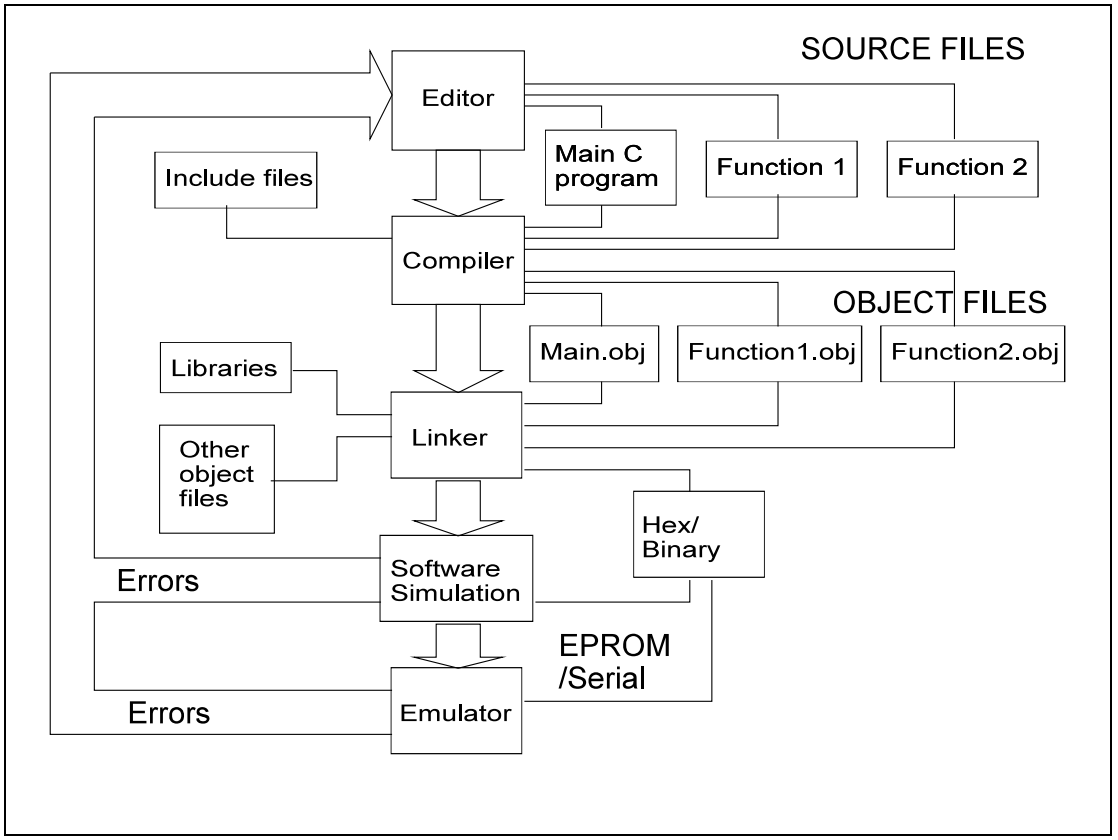
12.3 Development of an embedded C program

An embedded system is part of another product. For example, the processor in an air conditioning unit is part of, or “embedded” in the main air conditioning unit. The function of the embedded system is fixed, because of its embedded software.

There is more than one way to implement a software function. Because of this, we need to look at how the cross compiler generates assembly or machine code, in order to make better use of the processor. This has important consequences when debugging a system.

12.3.1 Differences in normal C and assembler programming

Writing an embedded C program is very similar to that of a typical C program. The main difference is the conversion and transferring of code to the system, which is through an EPROM or serial communications.



Cross Development Cycle of an Embedded C Program

When compared to standard assembler programming, the main difference here is the debugging method.

12.3.2 Debug facilities

Some considerations to take note of are the choice of debugging languages. It is preferable to debug in C. But the processor executes machine instructions. One line of C code may generate several bytes of machine code. It is a very useful feature to be able to debug in C, assembler, or a combination of them, in both the simulator and emulator.

The simulator and emulator however, work mainly at machine level. These systems need to know the variable names in the source file and where they are in the processor memory. Also they need to know which C source line corresponds to which machine code it is executing. All these information are produced by the compiler and has to be transferred somehow to the simulator or emulator. But because of lack of standards in embedded C compilers, it may not be possible to debug at source level.

12.4 Some features of using embedded C

For general purpose desktop microprocessor system such as PC's, all program and data functions are in RAM. But for microcontrollers we have to be concerned about:

CODE (Program) is normally in EPROM - read only, not writable
DATA (Variables) is normally in RAM - read/write, volatile
limited RAM space
C function and their use of stack space for local variables

Since a microcontroller has to control devices, there are several features in this type of programming that differs from normal C programming. For example, accessing processor registers that control the timer or interrupts. These registers have standard names and it is convenient to use them. Also the standard input/output devices which are taken for granted like the graphics display, keyboard, mouse and hard disk, are not present. Later we will look at ways of optimizing C programs.

12.4.1 Accessing processor resources

The definitions for the Input/Output registers are normally kept in header files, and should be included in your C program. Thus it is possible to refer to these registers by just referring to them appropriately.

Standard library functions for desktop machines like printf, scanf and so on, may not be available. Input and output may be to serial devices. It is common for compiler vendors who provide libraries based on this fact. For other ports and other kinds of facilities like string routines, multiple precision arithmetic and floating point calculations, users have to write their own routines. It is important to see what features are available and not make assumptions.

The C compiler generates code that is memory location independent. However when linking, you have to specify where the code (ROM) and data (RAM) memory locations are. This is highly dependent on the system design. Also note that the generated code is not memory location independent.

Uninitialized variables are not guaranteed to be zero. You need to do it explicitly, or modify the C startup routine.

12.4.2 Accessing I/O devices

Type casting for memory mapped I/O

Since the task of a microcontroller is to control devices, interfacing plays an important role. For I/O devices which are addressed as normal memory, (this is called memory mapped) we treat as external memory and a normal read from an address is:

```
data = *((int *) 0x1000);    /* 0x1000 is address $1000 */
```

However this is a 16 bit operation - we have to make the result fit into 8 bits. This is done through a “cast” which is the purpose of (char *).

to read: `data = *((char *) 0x1000);`
to write: `*((char  *) 0x1000) = data;`

I/O mapped I/O

In the PC/104, since we are using I/O mapped operations, extensions to the C language allow the use of input and output statements.

to read: `data = _inp(0x332);`
to write:
```
retcode = _outp(0x331, data); /* retcode for status */
                              /* of I/O op */
```

Optimising compilers

Another consideration are optimising compilers. These try to make the user program smaller or faster. They do this by examining the program to remove what it considers redundant code. For example, a common situation with external devices is that values present in their registers change without the processor taking any action. For example, in a keypad - the value read from it changes depending on whether someone has pressed a key, without intervention from the processor.

As discussed before, we will output a column scan value and then immediately read back to see if someone has pressed a key.

```
#define KbdPort  *((char *)0xA000)/* ptr to keypad port */

KbdPort = Col7Lo;                /* (1) output a value  */
ScanCode = KbdPort;              /* (2) read is ignored */
```

Here the value retrieved into ScanCode comes from the keypad port.

If this is compiled it will not work. The compiler's optimiser assumed that, because no

WRITE occurred between (1) and (2), KbdPort cannot have changed. Hence the code generated to make the second access to the keypad port is optimised out and we will get ScanCode being equal to Col7Lo. This is shown in the equivalent assembler code:

```
KbdPort = Col7Lo;
     ldab 0FF
     stab 0A000
ScanCode = KbdPort;
     stab  _ScanCode
```

The solution is declare KbdPort as "volatile" thus:

```
#define KbdPort  *((volatile char *)0xA000)
```

Now the optimiser will not try to remove subsequent accesses to the register, as shown by the “ldab”.

```
KbdPort = Col7Lo;
     ldab 0FF
     stab 0A000
ScanCode = KbdPort;
     ldab  0A000
     stab  _ScanCode
```

12.5 Generating efficient code

As mentioned earlier, because of the limited resources available to an embedded system, we need to look at various considerations to make code more efficient in terms of speed and code size.

These typically depend on the word size of the processor - be it 8, 16 or 32 bits. Accessing and processing variables of longer length than this will generate more instructions and use up more memory. In this section, we show some examples of the code generated by a typical compiler. You will see that the choice of variables will affect the code generated.

12.5.1 Use short and/or unsigned variables

The objective is to conserve use of scarce internal RAM. Try to use 'unsigned char' types for unsigned quantities or ‘short int’ as they are one byte. Note that 'int' (integer) is 16 bit.

For a one line statement involving addition, the size of code produced is shown. Variables with s,i,u below are short (8 bit signed), integer (16 bit signed), unsigned integers (8 bit unsigned), respectively. Mixed variables use the most code as they have to convert from one form to another.

short	int	mixed	mixed unsigned
`sl=s2+ s1*s3;`	`i1=i2+ i1*i3;`	`i1=s2+ s1*i3;`	`ui=us2+ us1*ui3;`
9 bytes	12 bytes	24 bytes	18 bytes

12.5.2 Use bit flags

Use bit flags for variables that take on values of only true or false, or 1 and 0. Thus instead of using eight bytes for eight variables, you only need a byte.

There are six operations that can be performed and they work on a bit by bit basis, in the same way as their assembler equivalents. The following summarizes the operation of bit variables. The variable A here has the value 0x55 or %01010101 binary. In the 6811 equivalent, assume accumulator A contains the value.

Operator	C	Result	Binary	Comment
AND	`A & 0x0F`	0x0A	0101 0101 0000 1111 --------- 0000 0101	Bitwise AND
OR	`A | 0x0F`	0x5F	0101 0101 0000 1111 --------- 0101 1111	Bitwise OR
XOR	`A ^ 0x0f`	0x5A	0101 0101 0000 1111 --------- 01011010	Bitwise XOR
NOT	~A	0xAA	0101 0101 --------- 1010 1010	One's complement
Left Shift	`A << 1`	0xAA	0101 0101 --------- 1010 1010	shift multiple, ignore CY
Right Shift	`A >> 1`	0x2A	0101 0101 --------- 0010 1010	shift multiple, ignore CY

Bit flags can be set up in one of two ways:

1. Use defines

```
#define STATIC 01
#define EXTERNAL 02
```

```
        unsigned short flags1;
```

2. Use a structure

```
struct {
   unsigned short static:1;
   unsigned short external:1;
   }flags2;
```

To set a bit flag, look at the following examples:

```
   flags1 |= (EXTERNAL | STATIC);
   flags2.external = 1;
   flags2.static = 1;
```

Note that in the structure, the most significant bits are assigned first.
To branch on bit flag values, here are some other examples

```
if((flags1 & (EXTERNAL | STATIC))==0) s++;
if((flags2.extenal==0) && (flags2.static==0)) s++;
```

12.5.3 Use ROM space for variables

Tables, messages, and other non volatile data should not be kept in RAM. Typically, they use up a lot of space and should be kept in ROM always. Be careful to see that they do not get copied to RAM by the compiler's code. Use the 'const' directive to store in EPROM. For example:

```
const unsigned char ScanTable [12] =
{0x7D, 0xEB, 0xED, 0xEE, 0xDB, 0xDD, 0xDE, 0xBB, 0xBD,
0xBE, 0x7B, 0x7E};
```

12.5.4 Arrays

If an array element or set of elements is frequently used, then efficiency will be increased by setting a pointer to the element and using the pointer to refer to it. If possible, use one dimensional arrays. Higher orders will involve multiplication, which is always slow.

```
example()
{
   short arr[20];
   short I,*p;

   t = arr[ I ];
   p = arr + I;
```

```
t = *p;
p = arr + I;
t = *p;
```

12.5.5 Functions

If function is non-recursive (does not call itself) and does not require local (automatic) variables, then the stack allocation / deallocation overhead on function entry and exit can be avoided by using:

```
function ( args )
/* argument declarations here */
{
      /* a non-recursive function with no local variables */
}
```

For functions, try not to pass data in functions:

```
function (argument1, argument2, argument3);
```

This takes up space on the stack

12.5.6 Use global variables

Function parameters take up stack space. Also, local variables in a function take up extra RAM. One way to save on this is to use global variables. Note that this is not recommended in C, because modules may modify a global variable by accident. But they may save RAM space and improve speed.

```
unsigned char      argument1, argument2

      function()
      result1 = argument1 ...
      result2 = argument2 ...
```

12.5.7 Interfacing to assembler

As we have seen in an earlier section, one C statement can generate many bytes of assembly code which takes time to execute. If certain parts of a program need to be speeded up, we may need to write that part of the program in assembler.

Of course this assumes one is able to come up with a more efficient algorithm to achieve the task! This is why it is useful to examine the code produced by a compiler, to see if the compiler has done a good job of generating the machine code. We can interface to the main program using standard linking techniques and inline code.

12.5.7.1 Linking object modules

Most cross compilers allow you to assemble program modules separately and combine them at link time.

12.5.7.2 Inline code

This inserts actual assembler instructions into the C program. The compiler needs to generate assembly language code from the C program, then invoke the assembler to convert the intermediate program into the object file.

Use asm ("CLI") directive to embed an assembler portion

If the assembler program needs to access C program variables, you need to be careful of what its 'real' name actually is:

EG: variable is 'ScanCode' - internally it becomes '_ScanCode'

Advantages
You do not have to maintain a separate file of assembler routines. Everything is in the C program.

If we are linking in an external assembler routine, it is not possible to know the absolute locations of program code and variables when debugging. Using inline code, this is possible as all addresses refer to the same source program.

Disadvantage
As the compiler has to generate an intermediate assembler file, it is slower. Also the assembler has to be called up after that.

12.6 Other Considerations

Macros and functions. For faster execution, use macros; to minimize memory, use functions.