



Programming for Embedded Systems



Even if C is losing its position as the mainstream programming language for general application development, embedded programming is still its stronghold. Students and programmers new to embedded programming, though proficient in general C programming, are often clueless about C programming for embedded systems. This article offers an introduction.



Every embedded system has software and hardware elements that are very closely dependent, and this affects how we program an embedded device. Based on this fact, we'll look at the hardware characteristics of embedded devices, how the hardware affects the language chosen to write programs for it, how C is used differently for embedded programming and, finally, we'll cover how GCC is used for embedded systems.

Characteristics of embedded devices

Programming for embedded systems is

quite different from general programming (for PCs). The main difference is due to the underlying hardware. The important characteristics of embedded systems when compared to regular PCs are:

- Embedded devices have significant resource constraints; typically, less memory, less processing power and limited hardware devices compared to regular desktop computers. [But this is slowly changing; the latest embedded devices have considerably more resources (such as a huge memory) and are more powerful than ever before. Nowadays, with smart mobile phones, we can

browse the Internet, play games, make phone calls, and even do programming—so, handheld devices are becoming as powerful as a PC, blurring the difference between embedded systems programming and general programming.]

- The hardware components used are not the same as the ones used for a PC: the components in an embedded device are lighter, cooler, smaller and less power-consuming. To be specific, the processor used in an embedded system is rarely the same used in a PC; and heavy I/O devices such as keyboards and monitors are not used in embedded systems.
- Embedded systems are more tied to the hardware/OS vendor than the PCs. PCs are general-purpose machines and we can take the availability of applications and programming tools for granted. However, embedded systems are typically more tied to the hardware or OS vendor and, hence, the common applications or tool-chains need not be widely or freely available.
- Embedded systems should be more reliable and efficient. If a program in a PC fails, we can restart the program and nothing serious happens. However, for many embedded system programs, if they fail, they can make the device useless. And sometimes they can cause serious harm -- imagine what could happen if an embedded program in a pacemaker device fails.

Programming for embedded devices

An embedded hardware device, depending on its size and capabilities, can have an operating system—such as embedded Linux—with limited or minimal functionality compared to a desktop version. For very small embedded devices, an OS might be entirely absent: it is not possible to write programs, compile, run and debug the code in such small devices. In such a situation, it is necessary to use cross compilers (or assemblers), which compile programs written in a high-level language on a host system (typically a PC) and generate code for a target system (for example, an embedded device). If we write assembly programs and use an assembler running on a host to generate code for a target device, it is a cross assembler. So, we can write programs on our PC, generate code for the embedded device and run it there. This solves the problem of creating executable code for embedded systems, but testing, debugging or tracing embedded programs is difficult.

One problem is that these programs usually don't have input devices such as keyboards and a mouse, or output devices like full-fledged monitors or display screens that we usually take for granted in regular programming for PCs. So, debugging, fixing and testing them is often more difficult. Fortunately, there are tools available to help an embedded programmer. For example, an in-circuit emulator is a hardware device used to help debug the program that runs in an embedded system. It plugs on top of an embedded device; it can be even used to emulate a processor that is yet to be developed! With help from this device, we can

set breakpoints, visualise the processing of the embedded hardware or monitor the signals in the pins. In this way, it is possible to debug programs, though it is not as convenient as regular debugging. There are also other aids such as simulators. For those new to embedded systems, understanding such domain-specific details takes more time than actually writing the programs.

Languages for programming embedded devices

C is the language of choice for most of the programming done for embedded systems. Why? Why not assembly language, C++, Java or any other language?

It might appear that assembly language is intuitively the most obvious choice, since embedded programming is all about programming hardware devices such as micro-controllers. It is true that micro-controllers were initially programmed mostly in assembly language as with other embedded devices. It is not that difficult to write an assembly program since the assembly language produces the tightest code, making it possible to squeeze every possible byte of memory usage. However, the problem is that it becomes difficult to use for any reasonably-sized program, and even a slightly complicated device. The difficulties are in getting assembly programs to work correctly; and understanding, debugging, testing and, most importantly, maintaining them in the long run. Also, high quality C compilers can often generate code that is comparable to the speed of programs written in assembly. So, the benefits of using assembly for efficiency are negligible compared to the ease with which programmers can write C code. However, if performance is the key to make or break a device, then it is hard to beat assembly. For example, DSP (digital signal processing) devices are mostly programmed in assembly even today, because performance is the most important requirement in these devices.

Languages such as C++ have features that are often bulky, inefficient or inappropriate for use in resource-constrained environments such as embedded devices. In particular, virtual functions and exception handling are two language features that are not efficient in terms of space and speed in embedded systems. Sometimes, C++ programming is used as 'Safe C', where only a small subset of C++ features is included. However, for convenience, most embedded projects pragmatically use C itself.

Languages with 'managed runtimes', such as Java, are mostly heavyweight. Running Java programs requires a Java Virtual Machine, which can take up a lot of resources. Though Java is popular in high-end mobile phones because of the portability it provides and for browsing the Web, it is rarely suitable for use in small embedded devices.

There are numerous special purpose or proprietary languages meant to be used in embedded systems such as B# and Dynamic C. Others, like Forth, are also well suited for the purpose. However, C is widely used and familiar to programmers worldwide, and its tools are easily available.

How C is used differently for embedded programming

If you are new to embedded C programming, you will notice that there are only subtle differences between a regular C program and an embedded C program.

The following is a list of the most important differences between C programming for embedded systems and C programming for PCs.

- *Writing low-level code:* In embedded programming, it is necessary to directly access the underlying hardware. For example, we might need to access a port, timer or a memory location. Similarly, we might need to do low-level programming activities like accessing the job queue, raise some signals, interrupts, etc. So, we need to write low-level programs that directly access, modify or update hardware; this is an important characteristic of embedded C programs. How do we write low-level code? C features such as pointers and bit-manipulation facilities enable us to program at the hardware level directly; so these features are used extensively in embedded programming.
- *Writing in-line assembly code:* The C language provides a limited set of features, so it is not possible to use high-level C code to perform a specific function. For example, the device might have some instructions for which there is no direct equivalent in C code (for example, bit-wise rotation). In such cases, we can write assembly code embedded within C programs called 'inline assembly'. The exact syntax depends on the compiler; here is an example for GCC:

```
int a=10, b;
asm ("movl %1, %%eax;
    movl %%eax, %0;"
    : "=r" (b)      /* output */
    : "r" (a)       /* input */
    : "%eax"        /* clobbered register */
    );
```

This code just assigns a to b (!), but this is just an example to show how in-line assembly code looks; usually it will have the *asm* keyword prefixed and/or suffixed by underscores. The assembly code is written in C program itself and we can use C variables to refer to data; the register allocation and other details will be taken care of by the compiler. We can write high-level functionality such as looping in C syntax, and only when we require processor specific functionality, we can write in-line assembly code. In this way, writing in-line assembly is more convenient than writing full-fledged assembly code.

- *No recursion, no heap:* Many of the devices are small, so the heap area (where dynamic memory segments will be allocated) might be limited or might not even exist. So, we would need to program without using *malloc* or its variations. This poses difficulties for those new to embedded systems: how to use a linked list or a tree

data structure which we conventionally program using dynamic memory allocation?

Fortunately, many of the data structures can be implemented by using static allocation itself. Here is a simple illustration of a doubly linked list made up of three nodes that are allocated statically (instead of dynamic memory allocation):

```
struct node {
    struct node * prev;
    int data;
    struct node * next;
};

int main(){
    struct node one, two, three;
    one.prev = &three; one.next = &two; one.data = 100;
    two.prev = &one; two.next = &three; two.data = 200;
    three.prev = &two; three.next = &one; three.data = 300;
    struct node* temp = &one;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while(temp != &one);
}
```

This program prints '100 200 300'. Using the basic idea from this program, if we pre-allocate the number of expected nodes statically, then we can write our own *my_node_malloc()* function that will return a node that was allocated statically. Using such techniques, it is possible to implement dynamic data structures such as linked lists and trees that still work under the limitations of embedded devices.

Similarly, features such as recursion are not allowed in most of the embedded devices. One reason is that the recursion is inefficient in terms of space and time compared to iterative code. Fortunately, it is possible to write any recursive code as iterative code (though writing or understanding iterative versions of the code is not usually intuitive).

To put it simply, costly language features (in terms of space and time) are either not available, nor recommended in embedded programming. As programmers, we should find alternative ways of achieving the same functionality.

- *Using limited C features or new language extensions:* Some of the language features that we take for granted in regular programming might simply not be available for embedded programming. For example, if the device does not support floating point numbers, then we cannot use floats or doubles in the programs.

Problems frequently occur the other way also: often, the underlying devices have hardware features that don't have direct support in the C programming language. For example, a device might support fixed-point numbers; however, C language doesn't have any support for this data

type, so there is no direct way of programming it.

Embedded C compilers differ from regular C compilers in many ways; one important difference is the language features they support. Most of the embedded C compilers will not have full conformance to the ANSI C standard (because it is heavy-weight); rather, they will support the embedded C specification (check the 'Reference' section for more details). This specification is meant for use in embedded systems, and the most common extensions and features to enhance performance and convenience for accessing underlying hardware resources are provided in it. The advantage in using this specification is that a large number of compilers implement the specification, and thus it is easy to port embedded programs with it.

Using GCC for embedded programming

GCC is a very valuable tool for embedded engineering. Though there are quite a few commercial embedded compilers available (such as the compilers from byte craft, ACE, EDG and CoWare), GCC is the most widely used compiler for embedded programming.

Why, you may ask?

It is not just that GCC is open source and free, it is also available for almost all the processors released so far. The GCC tool chain and libraries also have variants that are meant for use in embedded systems. For example, the *uClibc* is a smaller, lightweight implementation of the C library. With little pain, it is possible to build a compiler tool chain for a new embedded platform, though that topic is beyond the scope of this article (you can refer to www.linuxjournal.com/article/9904 to get started with it); we'll just cover what we can do with GCC.

A compiler tool-chain is a collection of programs that is used for compiling and building programs. It consists of three parts: the compiler, binary utilities and standard libraries. For example, the compiler, linker and assembler are part of the compiler tool chain. Binary utilities (*binutils*) are low-level programs that are necessary for working on a new platform. A few important *binutils* are given here:

- *nm*: reads a binary file and displays the (name of)

symbols in that file


- *size*: displays the size of various components of a binary file
- *strip*: removes optional parts of a binary file (such as sections for debugging info)
- *objdump*: reads the object file and displays the various sections and information from that file
- *ar*: stands for 'archiver'; it can collect object files and store them as a single file (code archives)

The debugger is not necessarily part of this, but it is often bundled with the rest of such binary utilities.

The *glibc* is a comprehensive, complex, portable implementation of the C standard library; but it is not very suitable for embedded systems. *uClibc* is a configurable, small C library meant for use in embedded systems. Since it is widely used and actively maintained, support is available for it and, hence, it is often the No 1 choice in the embedded compiler tool chain. By porting the GCC compiler tool chain, you need not leave the tools and techniques that we are already familiar with and can use them for embedded programming.

Where to from here?

This article provided only an overview of C programming for embedded systems. Low-level programming in C for hardware is one of the most interesting jobs one can get; so go ahead, explore and enjoy programming for your embedded device!

1. www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s4
2. JTC1/SC22/WG14. Programming languages – C – Extensions to support embedded processors. Technical report, ISO/IEC, 2004 (www.open-std.org/jtc1/sc22/wg14)
3. www.linuxjournal.com/article/9904 

By: S.G. Ganesh is a research engineer at Siemens (Corporate Technology). His latest book, "60 Tips on Object Oriented Programming", was published by Tata McGraw-Hill in December last year. You can reach him at sgganesh@gmail.com