**EMBEDDED C**

**Embedded C** is a set of language extensions for the [C Programming language](http://en.wikipedia.org/wiki/C_Language) by the [C Standards committee](http://en.wikipedia.org/wiki/SC22) to address commonality issues that exist between C extensions for different [embedded systems](http://en.wikipedia.org/wiki/Embedded_systems). Historically, embedded C programming requires nonstandard extensions to the C language in order to support exotic features such as [fixed-point arithmetic](http://en.wikipedia.org/wiki/Fixed-point_arithmetic), multiple distinct [memory banks](http://en.wikipedia.org/wiki/Memory_bank), and basic [I/O](http://en.wikipedia.org/wiki/Input/output) operations.

In 2008, the C Standards Committee extended the C language to address these issues by providing a common standard for all implementations to adhere to. It includes a number of features not available in normal C, such as, fixed-point arithmetic, named address spaces, and basic I/O hardware addressing.

Embedded C use most of the syntax and semantics of standard C, e.g., main() function, variable definition, datatype declaration, conditional statements (if, switch. case), loops (while, for), functions, arrays and strings, structures and union, bit operations, macros, unions,etc.

The C programming language is perhaps the most popular programming language for programming embedded systems. (Earlier [Embedded Systems/Embedded Systems Introduction#Which Programming Languages Will This Book Use?](http://en.wikibooks.org/wiki/Embedded_Systems/Embedded_Systems_Introduction#Which_Programming_Languages_Will_This_Book_Use.3F) we mentioned other popular programming languages).

Most C programmers are spoiled because they program in environments where not only is there a standard library implementation, but there are frequently a number of other libraries available for use. The cold fact is, that in embedded systems, there rarely are many of the libraries that programmers have grown used to, but occasionally an embedded system might not have a complete standard library, if there is a standard library at all. Few embedded systems have capability for dynamic linking, so if standard library functions are to be available at all, they often need to be directly linked into the executable. Oftentimes, because of space concerns, it is not possible to link in an entire library file, and programmers are often forced to "brew their own" standard c library implementations if they want to use them at all. While some libraries are bulky and not well suited for use on microcontrollers, many development systems still include the standard libraries which are the most common for C programmers.

C remains a very popular language for micro-controller developers due to the code efficiency and reduced overhead and development time. C offers low-level control and is considered more readable than assembly. Many free C compilers are available for a wide variety of development platforms. The compilers are part of an IDEs with ICD support, breakpoints, single-stepping and an assembly window. The performance of C compilers has improved considerably in recent years, and they are claimed to be more or less as good as assembly, depending on who you ask. Most tools now offer options for customizing the compiler optimization. Additionally, using C increases portability, since C code can be compiled for different types of processors.

**Const**

A "const" in a variable declaration is a promise by the programmer who wrote it that the program will not alter the variable's value.

There are 2 slightly different reasons "const" is used in embedded systems.

One reason is the same as in desktop applications:

Often a structure, array, or string is passed to a function using a pointer. When that argument is described as "const", such as when a header file says

void print\_string( char const \* the\_string );

, it is a promise by the programmer who wrote that function that the function will not modify any items in the structure, array, or string. (If that header file is properly #included in the file that implements that function, then the compiler will check that promise when that implementation is compiled, and give an error if that promise is violated).

On a desktop application, such a program would compile to exactly the same executable if all the "const" declarations were deleted from the source code -- but then the compiler would not check the promises.

When some other programmer has an important piece of data he wants to pass to that function, he can be sure simply by reading the header file that that function will not modify those items. Without that "const", he would either have to go through the source code of the function implementation to make sure his data isn't modified (and worry about the possibility that the next update to that implementation might modify that data), or else make a temporary copy of the data to pass to that function, keeping the original version unmodified.

**Storing data in ROM**

Another reason to use "const" is specific to embedded systems:

On many embedded systems, there is much more program Flash (or ROM) than RAM. A ".c" file that uses a definition such as

char \* months[] = {

"January", "February", "March",

"April", "May", "June",

"July", "August", "September",

"October", "November", "December",

};

forces the compiler to store a all those strings in program Flash, then on boot-up, to copy those values to a location in RAM. That wastes precious RAM if, as is often the case, the program never actually modifies those strings. By modifying the declaration to

char const \* const months[] = { ... };

, we inform the compiler that we promise to never modify those strings (or their order in the array), and so the compiler is free to store all those strings in program Flash, and fetch the original value from Flash whenever it is needed. That saves RAM for variables that really do change.

(Some compilers, if you use definitions such as

static char \* months[] = { ... };

, are smart enough to work out for themselves whether or not that the program ever actually modifies those strings. If the program does modify those strings, then of course the compiler must put them in RAM. But if not, the compiler is free to store those strings only once, in program Flash).

**Storing data in ROM on a Princeton architecture microcontrollers**

Princeton architecture microcontrollers use exactly the same instructions to access RAM as program Flash.

C compilers for such architectures typically put all data declared as "const" into program Flash. Functions neither know nor care whether they are dealing with data from RAM or program Flash; the same "read" instructions work correctly whether the function is given a pointer to RAM or a pointer to program Flash.

**Storing data in ROM on a Harvard architecture microcontrollers**

Unfortunately, Harvard architecture microcontrollers use completely different instructions to access RAM than program Flash.(Often they also have yet another set of instructions to access EEPROM, and another to access external memory chips). This makes it difficult to write a subroutine ( such as puts() ) that can be called from one part of the program to print out a constant string (such as "November") from ROM, and called from another part of the program to print out a variable string in RAM.

Unfortunately, different C compilers (even for the same chip) require different, incompatible techniques for a C programmer to tell a C compiler to put data in ROM. There are at least 3 ways for a C programmer to tell a C compiler to put data in ROM.

(1) Some people claim that using the "const" modifier to indicate that some data is intended to be stored in ROM is an abuse of notation. [[1]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-1) Such people typically propose using some non-standard attribute or storage specifier, such as "PROGMEM" or "rom"[[2]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-2), on variable definitions and function parameters, to indicate a "typed pointer" of type "value resides in program Flash, not RAM". Unfortunately, different compilers have different, incompatible ways of specifying that data may be placed in ROM. Typically such people use function libraries that 2 copies of functions that deal with strings (etc.); one copy is used for strings in RAM, the other copy is used for strings in ROM. This technique uses the minimum amount of RAM, but it usually requires more ROM than other techniques.

(2) Some function libraries assume the data is in RAM. When a programmer wants to call such functions with data that is actually in ROM, the programmer must make sure the data is first temporarily copied to a buffer in RAM, and then call that function with the address of that buffer. This technique uses the minimum amount of ROM to hold the library, but it uses more ROM and RAM than the other techniques at every function call that involves data in ROM.

(3) Some function libraries use functions that can handle being called from one place with a string in RAM, and from other places with a string in ROM. This typically requires "fat pointers" aka "generic pointers" that have extra bits that indicate whether the pointer is pointing to something in RAM or ROM. Every time such a library uses a pointer, the executing code checks those bits to see whether to execute the "read from RAM" or "read from ROM" instructions.[[3]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-3)[[4]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-4)[[5]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-5) This is a special case of "fat pointers" and "tagged pointers" used in other systems that execute different code depending on the type of the pointed-to object, where the "pointer" includes both type information and the destination address.[[6]](http://en.wikibooks.org/wiki/Embedded_Systems/C_Programming#cite_note-6)

**Volatile**

A "volatile" in a variable declaration tells us and the compiler that the value of that variable may change at any time, by some means outside the normal flow of this section of code. These changes may be caused by hardware i.e. a peripheral, another processor in a multiprocessor system, or an interrupt service routine.

The "volatile" keyword tells the compiler not to make certain optimizations that only work with "normal" variables stored in RAM or ROM that are completely under the control of this C program.

The entire point of embedded programming is its communications with the outside world -- and both input and output devices require the "volatile" keyword.

There are at least 3 types of optimizations that "volatile" turns off:

* "read" optimizations -- without "volatile", C compilers assume that once the program reads a variable into a register, it doesn't need to re-read that variable every time the source code mentions it, but can use the cached value in the register. This works great with normal values in ROM and RAM, but fails miserably with input peripherals. The outside world, and internal timers and counters, frequently change, making the cached value stale and irrelevant.
* "write" optimizations -- without "volatile", C compilers assume that it doesn't matter what order writes occur to different variables, and that only the last write to a particular variable really matters. This works great with normal values in RAM, but fails miserably with typical output peripherals. Sending "turn left 90, go forward 10, turn left 90, go forward 10" out the serial port is completely different than "optimizing" it to send "0" out the serial port.
* instruction reordering -- without "volatile", C compilers assume that they can reorder instructions. The compiler may decide to change the order in which variables are assigned to make better use of registers. This may fail miserably with IO peripherals where you, for example, write to one location to acquire a sample, then read that sample from a different location. Reordering these instructions would mean the old/stale/undefined sample is 'read', then the peripheral is told to acquire a new sample (which is ignored).

Depending on your hardware and compiler capabilities, other optimizations (SIMD, loop unrolling, parallelizing, pipelining) may also be affected.

**Const volatile**

Many people don't understand the combination of "const" and "volatile". As we discussed earlier in [Embedded Systems/Memory](http://en.wikibooks.org/wiki/Embedded_Systems/Memory), embedded systems have many kinds of memory.

Many input peripherals -- such as free-running timers and keypad interfaces -- must be declared "const volatile", because they both (a) change value outside by means outside this C program, and also (b) this C program should not write values to them (it makes no sense to write a value to a 10-key keypad).

**Compiled and interactive**

The vast majority of the time, when people write code in C, they run that code through C compiler on some personal computer to get a native executable. People working with embedded systems then download that native executable to the embedded system, and run it.

However, a few people working with embedded systems do things a little differently.

* Some use a C interpreter such as [PicoC](http://code.google.com/p/picoc/) or [Wikipedia: Interactive C](http://en.wikipedia.org/wiki/Interactive_C) or [Extensible Interactive C (EiC)](http://sourceforge.net/projects/eic/). They download the C source code to the embedded system, then they run the interpreter in the embedded system itself.
* Some people have the luxury of working with "large" embedded systems that can run a standard C compiler (it runs the standard GCC on Linux or BSD; or it runs the [Wikipedia: DJGPP](http://en.wikipedia.org/wiki/DJGPP) port of GCC on FreeDos; or it runs the [Wikipedia: MinGW](http://en.wikipedia.org/wiki/MinGW) port of GCC on Windows; or it runs the [Wikipedia: Tiny C Compiler](http://en.wikipedia.org/wiki/Tiny_C_Compiler) on Linux or Windows; or some other C compiler). They download the C source code to the embedded system, then they run the compiler in the embedded system itself.

**C compilers for embedded systems**

Perhaps the biggest difference between C compilers for embedded systems and C compilers for desktop computers is the distinction between the "platform" and the "target". The "platform" is where the C compiler runs -- perhaps a laptop running Linux or a desktop running Windows. The "target" is where the executable code generated by the C compiler will run -- the CPU in the embedded system, often without any underlying operating system.

The GCC compiler is the most popular C compiler for embedded systems. GCC was originally developed for 32-bit Princeton architecture CPUs. So it was relatively easily ported to target ARM core microcontrollers such as XScale and Atmel AT91RM9200; Atmel AVR32 AP7 family; MIPS core microcontrollers such as the Microchip PIC32; and Freescale 68k/ColdFire processors.

The people who write compilers have also (with more difficulty) ported GCC to target the Texas Instruments MSP430 16-bit MCUs; the Microchip PIC24 and dsPIC 16-bit Microcontrollers; the 8-bit Atmel AVR microcontrollers; the 8-bit Freescale 68HC11 microcontrollers.

Other microcontrollers are very different from a 32-bit Princeton architecture CPU. Many compiler writers have decided it would be better to develop an independent C compiler rather than try to force the round peg of GCC into the square hole of 8-bit Harvard architecture microcontroller targets:

SDCC - Small Device C Compiler for the Intel 8051, Maxim 80DS390, Zilog Z80, Motorola 68HC08, Microchip PIC16, Microchip PIC18

There are some highly respected companies that sell commercial C compilers. You can find such a commercial C compiler for practically every microcontroller, including the above-listed microcontrollers. Popular microcontrollers not already listed (i.e., microcontrollers for which the only known C compiler is a commercial C compiler) include the Cypress M8C MCUs; Microchip PIC10 and Microchip PIC12 MCUs; etc.

The **Arduino** environment performs some minor pre-processing to turn your sketch into a C++ program. It then gets passed to a **compiler** (avr-gcc), which turns the human readable code into machine readable instructions (or object files).