## Scope

In the Basic Concepts section, the concept of scope was introduced. It is important to revisit the distinction between local types and global types, and how to declare variables of each. To declare a local variable, you place the declaration at the beginning (i.e. before any non-declarative statements) of the block to which the variable is intended to be local. To declare a global variable, declare the variable outside of any block. If a variable is global, it can be read, and written, from anywhere in your program.

Global variables are not considered good programming practice, and should be avoided whenever possible. They inhibit code readability, create naming conflicts, waste memory, and can create difficult-to-trace bugs. Excessive usage of globals is usually a sign of laziness and/or poor design. However, if there is a situation where local variables may create more obtuse and unreadable code, there's no shame in using globals.

## [[edit](http://en.wikibooks.org/w/index.php?title=C_Programming/Variables&action=edit&section=16)] Other Modifiers

Included here, for completeness, are more of the modifiers that standard C provides. For the beginning programmer, *static* and *extern* may be useful. *volatile* is more of interest to advanced programmers. *register* and *auto* are largely deprecated and are generally not of interest to either beginning or advanced programmers.

**static** is sometimes a useful keyword. It is a common misbelief that the only purpose is to make a variable stay in memory.  
When you declare a function or global variable as *static* it will become internal. You cannot access the function or variable through the extern (see below) keyword from other files in your project.  
When you declare a local variable as *static*, it is created just like any other variable. However, when the variable goes out of scope (i.e. the block it was local to is finished) the variable stays in memory, retaining its value. The variable stays in memory until the program ends. While this behaviour resembles that of global variables, static variables still obey scope rules and therefore cannot be accessed outside of their scope.  
Variables declared static are initialized to zero (or for pointers, NULL) by default.

You can use static in (at least) two different ways. Consider this code, and imagine it is in a file called jfile.c:

#include <stdio.h>

static int j = 0;

void up(void)

{

/\* k is set to 0 when the program starts. The line is then "ignored"

\* for the rest of the program (i.e. k is not set to 0 every time up()

\* is called)

\*/

static int k = 0;

j++;

k++;

printf("up() called. k= %2d, j= %2d\n", k , j);

}

void down(void)

{

static int k = 0;

j--;

k--;

printf("down() called. k= %2d, j= %2d\n", k , j);

}

int main(void)

{

int i;

/\* call the up function 3 times, then the down function 2 times \*/

for (i= 0; i < 3; i++)

up();

for (i= 0; i < 2; i++)

down();

return 0;

}

The j var is accessible by both up and down and retains its value. the k vars also retain their value, but they are two different variables in each their scopes. static vars are a good way to implement encapsulation, a term from the object-oriented way of thinking that effectively means not allowing changes to be made to a variable except through function calls.

Running the program above will produce the following output:

up() called. k= 1, j= 1

up() called. k= 2, j= 2

up() called. k= 3, j= 3

down() called. k= -1, j= 2

down() called. k= -2, j= 1

**extern** is used when a file needs to access a variable in another file that it may not have #included directly. Therefore, *extern* does not actually carve out space for a new variable, it just provides the compiler with sufficient information to access the remote variable.

**volatile** is a special type modifier which informs the compiler that the value of the variable may be changed by external entities other than the program itself. This is necessary for certain programs compiled with optimizations - if a variable were not defined volatile then the compiler may assume that certain operations involving the variable are safe to optimize away when in fact they aren't. *volatile* is particularly relevant when working with embedded systems (where a program may not have complete control of a variable) and multi-threaded applications.

**auto** is a modifier which specifies an "automatic" variable that is automatically created when in scope and destroyed when out of scope. If you think this sounds like pretty much what you've been doing all along when you declare a variable, you're right: all declared items within a block are implicitly "automatic". For this reason, the *auto* keyword is more like the answer to a trivia question than a useful modifier, and there are lots of very competent programmers that are unaware of its existence.

**register** is a hint to the compiler to attempt to optimize the storage of the given variable by storing it in a register of the computer's CPU when the program is run. Most optimizing compilers do this anyway, so use of this keyword is often unnecessary. In fact, ANSI C states that a compiler can ignore this keyword if it so desires -- and many do. Microsoft Visual C++ is an example of an implementation that completely ignores the *register* keyword.

Bottom of Form

MACROS are substituted whole wherever they are used in a program: this is potentially a huge amount of code repetition. The advantage of a macro over an actual function, however, is speed. No time is taken up in passing control to a new function, because control never leaves the home function; the macro just makes the function a bit longer.

There are a few more directives for macro definition besides #define:

#undef

This undefines a macro, leaving the name free.

#ifdef

This is a kind of #if that is followed by a macro name. If that macro is defined then this directive is true. #ifdef works with #else in the same way that #if does.

#ifndef

This is the opposite of #ifdef. It is also followed by a macro name. If that name is not defined then this is true. It also works with #else.

Here is a code example using some macro definition directives from this section, and some conditional compilation directives from the last section as well.

#include <stdio.h>

#define CHOICE 500

int my\_int = 0;

#undef CHOICE

#ifdef CHOICE

void set\_my\_int()

{

my\_int = 23;

}

#else

void set\_my\_int()

{

my\_int = 17;

}

#endif

int main ()

{

set\_my\_int();

printf("%d\n", my\_int);

return 0;

}

The above code example displays 17 on the screen.

#define STRING1 "A macro definition\n"

#define STRING2 "must be all on one line!\n"

#define EXPRESSION1 1 + 2 + 3 + 4

#define EXPRESSION2 EXPRESSION1 + 10

#define ABS(x) (((x) < 0) ? -(x) : (x))

#define MAX(a,b) ((a < b) ? (b) : (a))

#define BIGGEST(a,b,c) ((MAX(a,b) < c) ? (c) : (MAX(a,b)))

Guidelines for writing efficient C/C++ code

Simple [source code](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=source%20code&x=&y=) changes can often result in substantial performance enhancements using modern optimizing compilers on high-end embedded processors. But, why is performance necessary? After all, the capabilities of modern microprocessors dwarf the capabilities of 1980-era supercomputers.

First, the average case response time of a real-time system is irrelevant. It is the worst-case response time that guards against a dropped call, a misprint, or other error conditions in a product. In other words, performance is necessary to minimize the response time of one's system, thereby achieving product reliability.

Second, increased performance allows for the implementation of more features without compromising the [integrity](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=integrity&x=&y=) of the system. Conversely, performance might be used to select a cheaper microprocessor or one that consumes less power.

There are many factors that determine the performance of a system. The choice of hardware can mean the difference between a few MIPS and a few hundred. Good data structures and algorithms are essential, and bookshelves have been written on this topic. A good [compiler](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=compiler&x=&y=) is also essential. One should evaluate the features and optimization capabilities of a compiler before spending too much time working with it.   
  
The purpose here is to explore an often-overlooked aspect of achieving maximum performance. No matter what hardware is chosen, which data structures and algorithms are employed, and which compiler is used, proper coding guidelines can dramatically impact the efficiency of one's code.

Nonetheless, developers are often unaware of the consequences of their programming habits. And unlike fundamental design decisions, many of these improvements can be made at a late stage of a project.

In the following examples, efficient coding guidelines will be illustrated using the [C](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=C&x=&y=) and [C++](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=C++&x=&y=) programming languages and, at times, PowerPC, ARM, and x86 assembly code. Many of these concepts, however, apply to other programming languages. Most of them are also processor independent, although there are some exceptions, which will be noted later.

**Choice of Data Type Sizes**   
The most fundamental data type in the C language is the integer, which is normally defined as the most efficient type for a given processor. The C language treats this type specially in that the language does not operate on smaller types, at least conceptually speaking.

If a [character](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=character&x=&y=) is incremented, the language specifies that the character is first promoted to an integer, the addition is performed as an integer, and the truncated result is stored back into the character.

In practice, using a smaller type for a local variable is computationally inefficient. Consider the following snippet of code:

**int m;   
char c;   
m += ++c;**

In PowerPC assembly code, this looks like:   
http://i.cmpnet.com/embedded/insights/2006/03/GHS422Code1.jpg  
On a CISC chip like the 80386, it looks like:   
http://i.cmpnet.com/embedded/insights/2006/03/GHS422Code2.jpg

On a RISC chip, one pays the price after writing to a sub-integral variable " the value must be sign or zero extended in register to ensure that the high bits of the register are consistently set. On a CISC chip, one pays the price when mixing the short variable with a larger integer variable. And unless you're using an 8 or [16-bit](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=16-bit&x=&y=) microprocessor, the microprocessor does not save on the shorter add instruction, so there is nothing to make up for the cost of the sign or zero extension after the increment.

Variables in memory are sometimes different. Because memory is scarce, it often makes sense to conserve by using a smaller data type. Most microprocessors can load and extend the short value in a single instruction, which means that the extension is free. Also, the truncation that is necessary when storing the value can often be avoided because sub-integral store instructions generally ignore the high bits.

These choices can be simplified by using the C99 header **stdint.h**. Even if your compiler does not support C99, it might provide stdint.h, or you may write it yourself. This file defines types that fit into a few categories:

**1)** i**nt***size***\_t** " Used when a type must be exactly a given length.   
**2)  int\_least***size***\_t** " Used when a type must be at least a given size and when the type should be optimized for space. This is preferred for data structures.   
**3)**  **int\_fast***size***\_t** " Used when a type must be at least a given size and when the type should be as fast as possible. This is preferred for local variables.

Larger data types should also be used sparingly. For example, C99 adds the long long type. This type has been available as an extension to C89 in most compilers for years. On a [32-bit](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=32-bit&x=&y=) microprocessor, these variables take up two registers and require multiple operations to perform arithmetic on them. An addition or a subtraction can usually be done inline in a few instructions.

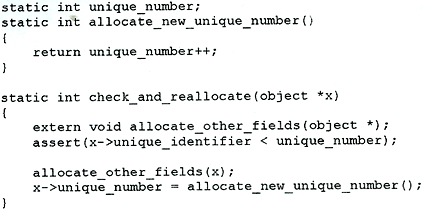
However, an innocuous looking shift can turn into quite a few instructions. And you can forget about division! Of course, these variables might be necessary or convenient when coding, but do not use them unless you need them and understand the corresponding code size and speed impact.

**Variable signedness**   
Unsigned variables can result in more efficient generated code for certain operations than using signed variables. For example, unsigned division by a power of two can be performed as a right shift.

Most architectures require a few instructions to perform a signed divide by a power of two. Likewise, computing an unsigned modulus by a power of two can be implemented as an AND operation (or a [bit](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=bit&x=&y=) extract). Computing a signed modulus is more complicated. In addition, it is sometimes useful that unsigned variables can represent just over twice as many positive values as their signed counterparts.

Characters deserve special mention here. Many architectures, such as the ARM, and PowerPC, are more efficient at loading unsigned characters than they are at loading signed characters. Other architectures, such as the V800 and SH, handle signed characters more efficiently. Most architectures' ABIs define the plain "char" type as whichever type is most efficient. So, unless a character needs to be signed or unsigned, use the plain char type.

**Use of access types**   
For global data, use the **static** keyword (or C++ anonymous namespaces) whenever possible. In some cases, **static** allows a compiler to deduce things about the access patterns to a variable. The **static** keyword also "hides" the data, which is generally a good thing from a programming practices standpoint. Declaring a function as **static** is also helpful in many cases.   
Example:



This code can be optimized by the compiler in a couple of ways because the functions were declared as **static**. First, the compiler can inline the call to *allocate\_new\_unique\_number()* and delete the out-of-line copy, because it is not called from any other place.

Second, if the compiler has basic information on the external *allocate\_other\_fields()* function, the compiler can sometimes tell that the function will not call back into this module. This knowledge allows eliminating the second load of unique\_number embedded in the inlined *allocate\_new\_unique\_number()* function.

<>On the other hand, **static** data should be avoided whenever possible for function-local data. The data's value must be preserved between calls to the function, making it very expensive to access the data, and requiring permanent RAM storage. The static keyword should only be used on function-local data when the value must be preserved across calls or for large data allocations (such as an array) when the programmer prefers to tradeoff consumption of stack space for permanent RAM.

**Global Variable Definition**   
Compilers can sometimes optimize accesses to global variables if they are defined and used together in the same module. In that case, the compiler can access one variable as an offset from another variable's address, as if they were both members of the same structure. Therefore, all other things being equal, it is worthwhile to define global variables in the modules where they are used the most.

For example, if the global variable "**glob**" is used the most in file.c, it should be defined there. Use a definition such as:

**int glob;**   
or:   
**int glob = 1;**

in addition to declaring glob (with **extern int glob;**) in a header file so that other modules can reference it.

Some programmers get confused about uninitialized variable definitions, so it is worth clarifying how they are often implemented. Most C compilers support a feature, called common variables, where uninitialized global variable definitions are combined and resolved at link-time. For example, a user could place the definition:

**int glob;**

in a header file and include this header file in every file in the project. While a strict reading of the C language implies that this will result in link-time errors, under a common variable model, each module will output a special common reference, which is different from a traditional definition.

The linker will combine all of the common references for a variable, and if the variable was not already defined elsewhere, allocate space for the variable in an uninitialized data section (such as **.bss**) and point all of the common references to this newly allocated space. On the other hand, if the user employs a definition such as:

**int glob = 1;**

in one module, all other uninitialized, common references would resolve to this definition.

It is best to write code without defining the same uninitialized global variable in multiple modules. Then, if you turn off the common variable feature in the compiler, the compiler is able to perform more aggressive optimizations because it knows that uninitialized global variable definitions are true definitions.

**Volatile Variables**   
The volatile keyword tells the compiler not to optimize away accesses to the variable or type. That is, the value will be loaded each time it is needed, and will be stored each time it is modified. Volatile variables have two major uses:

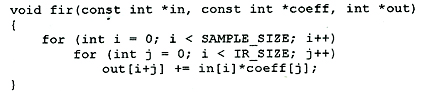
**1)** The variable is a memory mapped hardware device where the value can change asynchronously and/or where it is critical that all writes to the variable are respected.   
**2)** The variable is used in a multi-threading environment where another thread may modify the variable at any time.

The alternative to careful use of the volatile keyword is to disable so-called "memory optimizations". Effectively, all variables are treated as volatile when this option is chosen. Because disabling memory optimizations are important for efficient code, developers are encouraged to choose a less conservative approach.

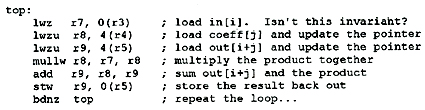
Separate threads often perform separate functions. As a result, they may have many variables and data structures that are not accessed by other tasks. Good software engineering practices might be able to minimize the overlap to a few shared header files. If such practices have reduced the scope of the code to a few files, it is more feasible to find the variables and data structures that are shared between threads.

**Const**   
The const keyword is helpful in a couple of ways. First, const variables can usually be allocated to a read-only data section, which can save on the amount of RAM required if the read-only data is allocated to flash or ROM. Second, the const keyword, when applied to a pointer or reference parameter, might allow the compiler to deduce that the call will not result in the value being modified.

**Restrict**  
The restrict keyword tells a compiler that the specified pointer is the only way to access a given piece of data. This can allow the compiler to optimize more aggressively. Consider the following example, a simple finite impulse response code:



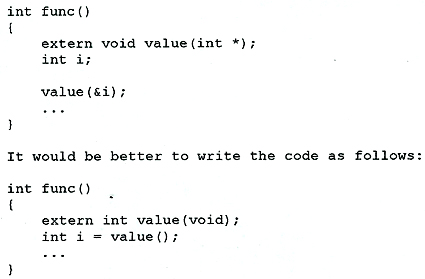
Consider the inner loop for a PowerPC target:



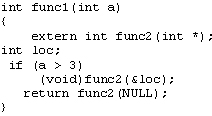
Better code can be generated by pulling the first load out of the loop, since **in[i]** does not change within the inner loop. However, the compiler cannot tell that **in[i]** does not change within the loop. The compiler is unable to determine this because the in and out arrays could overlap. If the function declaration is changed to:

**void fir\_and\_copy(int \*in, const int \*coeff, int \*restrict out)**

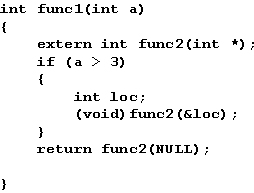
the compiler knows that writes through **\*out** cannot change the values in **\*in**. The restrict keyword is a bit confusing because it applies to the pointer itself rather than the data the pointer points to (contrast const int **\*x** and **int \*restrict x**).

**Pointers and the & operator**   
It is usually more efficient to have a function return a scalar value than to pass the scalar value by reference or by address. For example, often times a value is returned from a function by passing the address of an integer to the function. For example:   


Taking the address of a variable forces the compiler to allocate the variable on the stack, all but assuring less efficient code generation. Passing an argument as a C++ reference parameter has the same effect.

**Declaration scope of variables**   
Declare a variable in the inner-most scope possible, particularly when its address must be taken. In such cases, the compiler must keep the variable around on the stack until it goes out of scope. This can inhibit certain optimizations that depend on the variable being dead. For example, consider the variable "loc" in the following function:   
  
The compiler could potentially perform a tail call for the call to *func2()*. This is an optimization where the frame for *func()*, if any, is popped, and the last call instruction to *func2()* is replaced by a branch to func2().

This saves the need to return to *func1()*, which would immediately return to its caller. Instead, *func2()* returns directly to *func1()*'s caller. Unfortunately, the compiler cannot employ this optimization because it cannot determine that *loc* is not used in the second call to *func2()* (which is possible if its address was saved in the first call). The following code allows for better optimization:

  
In this case, the compiler knows that the lifetime of *loc* ends before the final call " and the tail call, at least in principle, can happen. Another benefit of using inner scopes is that variables from non-overlapping scopes can share the same space on the stack. This helps to minimize stack use and can result in smaller code size on some architectures.

However, it is usually not worthwhile to create artificially small scopes simply to bound the lifetimes of variables.

**Floating Point Arithmetic**   
Understanding the rules of arithmetic promotion can help you avoid costly mistakes. Many embedded architectures do not implement floating point arithmetic in hardware. Some processors implement the single precision "**float**" type in hardware, but leave the "**double**" type to floating point software emulation routines.

Unless doubles are implemented in hardware, it is more efficient to do arithmetic with the single precision type. If this is the case and if single precision arithmetic is sufficient, follow these rules:

**1**. Write single precision floating point constants with the **F** suffix. For example, write **3.0F** instead of **3.0**. The constant **3.0** is a double precision value, which forces the surrounding arithmetic expressions to be promoted to double precision.   
**2.** Use single precision math routines, such as **sinf()** instead of **sin()**, the double precision version.   
**3.** Avoid old-style prototypes and function declarations because these force floats to be promoted to double. So, instead of:

**float foo(f)**   
**float f;**

do:

**float foo(float f);**

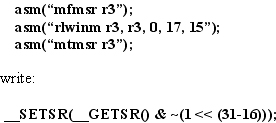
This is probably only a concern for old code bases.

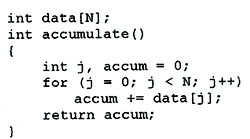
**Variable Length Arrays**The variable length array feature, which is included in C99, might result in less efficient array access. In cases where the array is multi-dimensional and subscripts other than the first are of variable lengths, the resulting code may be larger and slower. The feature is useful, but be aware that code generation can suffer.

**Low Level Assembly Instructions**Sometimes it is helpful or necessary to use specific assembly instructions in embedded programming. Intrinsic functions are the best way to do this. Intrinsic functions look like function calls, but they are inlined into specific assembly instructions. Refer to your compiler vendor's documentation to determine which intrinsics are available.

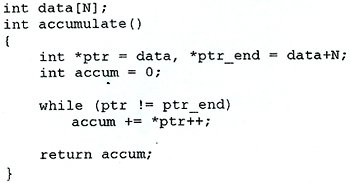
Inlined assembly code uses non-portable syntax and compilers generally make over-conservative assumptions when encountering inlined assembly, thus affecting code performance. Intrinsics can be #define'd into other names if switching from one compiler to another. This can be done once in a header file rather than going through the code to see every place where inlined assembly was used.

For example, instead of writing the following code to disable interrupts on the PowerPC:

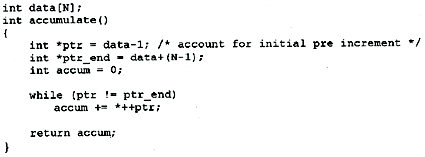
****

**Manual Loop Tricks**Sometimes programmers feel compelled to manually unroll loops, perform strength reduction, or use other transformations that a standard compiler optimizer would handle. For example:  ****

is sometimes manually transformed into:

****

Such transformations are usually only effective under fairly specific architectures and compilers. For example, the 32-bit ARM [architecture](http://www.embedded.com/encyclopedia/defineterm.jhtml?term=architecture&x=&y=) supports the post-increment addressing mode used above, but the PowerPC architecture only includes the pre-increment addressing mode. So, for the PowerPC, this loop could be written as:

****As a general rule, only do manual transformations for time critical sections of your code where your compiler of choice has not been able to perform adequate optimizations on its own, even after adjusting compilation options. Write simple code for most cases and let the compiler do the optimization work for you.

**Conclusion**The performance impact of some decisions that programmers make when writing their code can be significant. While efficient algorithmic design is of the highest importance, making intelligent choices when implementing the design can help application code perform at its highest potential.

**Introduction to the Volatile Keyword**

**The use of volatile is poorly understood by many programmers. This is not surprising, as most C texts dismiss it in a sentence or two.**

Have you experienced any of the following in your C/C++ embedded code?

* Code that works fine-until you turn optimization on
* Code that works fine-as long as interrupts are disabled
* Flaky hardware drivers
* Tasks that work fine in isolation-yet crash when another task is enabled

If you answered yes to any of the above, it's likely that you didn't use the C keyword **volatile**. You aren't alone. The use of **volatile** is poorly understood by many programmers. This is not surprising, as most C texts dismiss it in a sentence or two.

**volatile** is a qualifier that is applied to a variable when it is declared. It tells the compiler that the value of the variable may change at any time-without any action being taken by the code the compiler finds nearby. The implications of this are quite serious. However, before we examine them, let's take a look at the syntax.

**Syntax**

To declare a variable volatile, include the keyword **volatile** before or after the data type in the variable definition. For instance both of these declarations will declare foo to be a volatile integer:

**volatile int foo;  
int volatile foo;**

Now, it turns out that pointers to volatile variables are very common. Both of these declarations declare foo to be a pointer to a volatile integer:

**volatile int \* foo;   
int volatile \* foo;**

Volatile pointers to non-volatile variables are very rare (I think I've used them once), but I'd better go ahead and give you the syntax:

**int \* volatile foo;**

And just for completeness, if you really must have a volatile pointer to a volatile variable, then:

**int volatile \* volatile foo;**

Incidentally, for a great explanation of why you have a choice of where to place volatile and why you should place it after the data type (for example, **int volatile \* foo**), consult Dan Sak's column, "Top-Level cv-Qualifiers in Function Parameters" (February 2000, p. 63).

Finally, if you apply **volatile** to a struct or union, the entire contents of the struct/union are volatile. If you don't want this behavior, you can apply the volatile qualifier to the individual members of the struct/union.

**Use**

A variable should be declared volatile whenever its value could change unexpectedly. In practice, only three types of variables could change:

* Memory-mapped peripheral registers
* Global variables modified by an interrupt service routine
* Global variables within a multi-threaded application

**Peripheral registers**

Embedded systems contain real hardware, usually with sophisticated peripherals. These peripherals contain registers whose values may change asynchronously to the program flow. As a very simple example, consider an 8-bit status register at address 0x1234. It is required that you poll the status register until it becomes non-zero. The nave and incorrect implementation is as follows:

**UINT1 \* ptr = (UINT1 \*) 0x1234;**

**// Wait for register to become non-zero.  
while (\*ptr == 0);  
// Do something else.**

This will almost certainly fail as soon as you turn the optimizer on, since the compiler will generate assembly language that looks something like this:

**mov    ptr, #0x1234     mov    a, @ptr loop     bz    loop**

The rationale of the optimizer is quite simple: having already read the variable's value into the accumulator (on the second line), there is no need to reread it, since the value will always be the same. Thus, in the third line, we end up with an infinite loop. To force the compiler to do what we want, we modify the declaration to:

**UINT1 volatile \* ptr =   
    (UINT1 volatile \*) 0x1234;**

The assembly language now looks like this:

**mov     ptr, #0x1234  
loop    mov    a, @ptr          
    bz    loop**

The desired behavior is achieved.

Subtler problems tend to arise with registers that have special properties. For instance, a lot of peripherals contain registers that are cleared simply by reading them. Extra (or fewer) reads than you are intending can cause quite unexpected results in these cases.

**Interrupt service routines**

Interrupt service routines often set variables that are tested in main line code. For example, a serial port interrupt may test each received character to see if it is an ETX character (presumably signifying the end of a message). If the character is an ETX, the ISR might set a global flag. An incorrect implementation of this might be:

int etx\_rcvd = FALSE;

**void main()  
{  
    ...  
    while (!ext\_rcvd)  
    {  
        // Wait  
    }  
    ...  
}**

**interrupt void rx\_isr(void)  
{  
    ...  
    if (ETX == rx\_char)  
    {  
        etx\_rcvd = TRUE;  
    }  
    ...  
}**

With optimization turned off, this code might work. However, any half decent optimizer will "break" the code. The problem is that the compiler has no idea that **etx\_rcvd** can be changed within an ISR. As far as the compiler is concerned, the expression **!ext\_rcvd** is always true, and, therefore, you can never exit the while loop. Consequently, all the code after the while loop may simply be removed by the optimizer. If you are lucky, your compiler will warn you about this. If you are unlucky (or you haven't yet learned to take compiler warnings seriously), your code will fail miserably. Naturally, the blame will be placed on a "lousy optimizer."

The solution is to declare the variable **etx\_rcvd** to be **volatile**. Then all of your problems (well, some of them anyway) will disappear.

**Multi-threaded applications**

Despite the presence of queues, pipes, and other scheduler-aware communications mechanisms in real-time operating systems, it is still fairly common for two tasks to exchange information via a shared memory location (that is, a global). When you add a pre-emptive scheduler to your code, your compiler still has no idea what a context switch is or when one might occur. Thus, another task modifying a shared global is conceptually identical to the problem of interrupt service routines discussed previously. So all shared global variables should be declared **volatile**. For example:

**int cntr;**

**void task1(void)  
{  
    cntr = 0;  
    while (cntr == 0)  
    {  
        sleep(1);  
    }  
    ...  
}**

**void task2(void)  
{  
    ...  
    cntr++;  
    sleep(10);  
    ...  
}**

This code will likely fail once the compiler's optimizer is enabled. Declaring cntr to be volatile is the proper way to solve the problem.

**Final thoughts**

Some compilers allow you to implicitly declare all variables as volatile. Resist this temptation, since it is essentially a substitute for thought. It also leads to potentially less efficient code.

Also, resist the temptation to blame the optimizer or turn it off. Modern optimizers are so good that I cannot remember the last time I came across an optimization bug. In contrast, I come across failures to use **volatile** with depressing frequency.

If you are given a piece of flaky code to "fix," perform a grep for **volatile**. If grep comes up empty, the examples given here are probably good places to start looking for problems.

## Introduction

During a project for developing a light JPEG library which is enough to run on a mobile device without compromising quality graphics on a mobile device, I have seen and worked out a number of ways in which a given computer program can be made to run faster. In this article, I have gathered all the experiences and information, which can be applied to make a C code optimized for speed as well as memory.

Often, speeding up a program can also cause the code's size to increase. This increment in code size can also have an adverse effect on a program's complexity and readability. It will not be acceptable if you are programming for small device like mobiles, PDAs etc., which have strict memory restrictions. So, during optimization, our motto should be to write the code in such a way that memory and speed both will be optimized.

## Declaration

Actually, during my project, I have used the tips from [this](http://www.arm.com/pdfs/DAI0034A_efficient_c.pdf ) for optimization ARM because my project was on ARM platform, but I have also used many other articles from Internet. All tips of every article do not work well, so I collect only those tips together, which are very useful and very efficient. Also, I have modified some of them in such a way that they are almost applicable for all the environments apart from ARM.

What I did is just make a collection of the information from various sites but mostly from that PDF file I mentioned above. I never claimed that these are my own discoveries. I have mentioned all information sources in the *References* section at the end of this article.

## Where it is needed?

Without this point, no discussion can be started. First and the most important part of optimizing a computer program is to find out where to optimize, which portion or which module of the program is running slow or using huge memory. If each part is separately being optimized then the total program will be automatically faster.

The optimizations should be done on those parts of the program that are run the most, especially those methods which are called repeatedly by various inner loops that the program can have.

For an experienced programmer, it will usually be quite easy to find out the portions where a program requires the most optimization attention. But there are a lot of tools also available for detecting those parts of a program. I have used Visual C ++ IDE's in-built profiler to find out where the program spends most click tricks. Another tool I have used is Intel Vtune, which is a very good profiler for detecting the slowest parts of a program. In my experience, it will usually be a particular inner or nested loop, or a call to some third party library methods, which is the main culprit for running the program slow.

## Integers

We should use unsigned int instead of int if we know the value will never be negative. Some processors can handle unsigned integer arithmetic considerably faster than signed (this is also good practice, and helps make for self-documenting code).

So, the best declaration for an int variable in a tight loop would be:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

register unsigned int variable\_name;

although, it is not guaranteed that the compiler will take any notice of register, and unsigned may make no difference to the processor. But it may not be applicable for all compilers.

Remember, integer arithmetic is much faster than floating-point arithmetic, as it can usually be done directly by the processor, rather than relying on external FPUs or floating point math libraries.

We need to be accurate to two decimal places (e.g. in a simple accounting package), scale everything up by 100, and convert it back to floating point as late as possible.

## Division and Remainder

In standard processors, depending on the numerator and denominator, a 32 bit division takes 20-140 cycles to execute. The division function takes a constant time plus a time for each bit to divide.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

Time (numerator / denominator) = C0 + C1\* log2 (numerator / denominator)

= C0 + C1 \* (log2 (numerator) - log2 (denominator)).

The current version takes about 20 + 4.3N cycles for an ARM processor. As an expensive operation, it is desirable to avoid it where possible. Sometimes, such expressions can be rewritten by replacing the division by a multiplication. For example, (a / b) > c can be rewritten as a > (c \* b) if it is known that b is positive and b \*c fits in an integer. It will be better to use unsigned division by ensuring that one of the operands is unsigned, as this is faster than signed division.

## Combining division and remainder

Both dividend (x / y) and remainder (x % y) are needed in some cases. In such cases, the compiler can combine both by calling the division function once because as it always returns both dividend and remainder. If both are needed, we can write them together like this example:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int func\_div\_and\_mod (int a, int b) {

return (a / b) + (a % b);

}

## Division and remainder by powers of two

We can make a division more optimized if the divisor in a division operation is a power of two. The compiler uses a shift to perform the division. Therefore, we should always arrange, where possible, for scaling factors to be powers of two (for example, 64 rather than 66). And if it is unsigned, then it will be more faster than the signed division.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

typedef unsigned int uint;

uint div32u (uint a) {

return a / 32;

}

int div32s (int a){

return a / 32;

}

Both divisions will avoid calling the division function and the unsigned division will take fewer instructions than the signed division. The signed division will take more time to execute because it rounds towards zero, while a shift rounds towards minus infinity.

## An alternative for modulo arithmetic

We use remainder operator to provide modulo arithmetic. But it is sometimes possible to rewrite the code using if statement checks.

Consider the following two examples:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

uint modulo\_func1 (uint count)

{

return (++count % 60);

}

uint modulo\_func2 (uint count)

{

if (++count >= 60)

count = 0;

return (count);

}

The use of the if statement, rather than the remainder operator, is preferable, as it produces much faster code. Note that the new version only works if it is known that the range of count on input is 0-59.

## Using array indices

If you wished to set a variable to a particular character, depending upon the value of something, you might do this:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

switch ( queue ) {

case 0 : letter = 'W';

break;

case 1 : letter = 'S';

break;

case 2 : letter = 'U';

break;

}

Or maybe:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

if ( queue == 0 )

letter = 'W';

else if ( queue == 1 )

letter = 'S';

else

letter = 'U';

A neater (and quicker) method is to simply use the value as an index into a character array, e.g.:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

static char \*classes="WSU";

letter = classes[queue];

## Global variables

Global variables are never allocated to registers. Global variables can be changed by assigning them indirectly using a pointer, or by a function call. Hence, the compiler cannot cache the value of a global variable in a register, resulting in extra (often unnecessary) loads and stores when globals are used. We should therefore not use global variables inside critical loops.

If a function uses global variables heavily, it is beneficial to copy those global variables into local variables so that they can be assigned to registers. This is possible only if those global variables are not used by any of the functions which are called.

For example:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int f(void);

int g(void);

int errs;

void test1(void)

{

errs += f();

errs += g();

}

void test2(void)

{

int localerrs = errs;

localerrs += f();

localerrs += g();

errs = localerrs;

}

Note that test1 **must load and store** the global errs value each time it is incremented, whereas test2 stores localerrs in a register and needs only a single instruction.

## Using Aliases

Consider the following example -

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

void func1( int \*data )

{

int i;

for(i=0; i<10; i++)

{

anyfunc( \*data, i);

}

}

Even though \*data may never change, the compiler does not know that anyfunc () did not alter it, and so the program must read it from memory each time it is used - it may be an alias for some other variable that is altered elsewhere. If we know it won't be altered, we could code it like this instead:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

void func1( int \*data )

{

int i;

int localdata;

localdata = \*data;

for(i=0; i<10; i++)

{

anyfunc ( localdata, i);

}

}

This gives the compiler better opportunity for optimization.

## Live variables and spilling

As any processor has a fixed set of registers, there is a limit to the number of variables that can be kept in registers at any one point in the program.

Some compilers support live-range splitting, where a variable can be allocated to different registers as well as to memory in different parts of the function. The live-range of a variable is defined as all statements between the last assignment to the variable, and the last usage of the variable before the next assignment. In this range, the value of the variable is valid, thus it is alive. In between live ranges, the value of a variable is not needed: it is dead, so its register can be used for other variables, allowing the compiler to allocate more variables to registers.

The number of registers needed for register-allocatable variables is at least the number of overlapping live-ranges at each point in a function. If this exceeds the number of registers available, some variables must be stored to memory temporarily. This process is called spilling.

The compiler spills the least frequently used variables first, so as to minimize the cost of spilling. Spilling of variables can be avoided by:

* Limiting the maximum number of live variables: this is typically achieved by keeping expressions simple and small, and not using too many variables in a function. Subdividing large functions into smaller, simpler ones might also help.
* Using register for frequently-used variables: this tells the compiler that the register variable is going to be frequently used, so it should be allocated to a register with a very high priority. However, such a variable may still be spilled in some circumstances.

## Variable Types

The C compilers support the basic types char, short, int and long (signed and unsigned), float and double. Using the most appropriate type for variables is very important, as it can reduce code and data size and increase performance considerably.

## Local variables

Where possible, it is best to avoid using char and short as local variables. For the types char and short, the compiler needs to reduce the size of the local variable to 8 or 16 bits after each assignment. This is called sign-extending for signed variables and zero extending for unsigned variables. It is implemented by shifting the register left by 24 or 16 bits, followed by a signed or unsigned shift right by the same amount, taking two instructions (zero-extension of an unsigned char takes one instruction).

These shifts can be avoided by using int and unsigned int for local variables. This is particularly important for calculations which first load data into local variables and then process the data inside the local variables. Even if data is input and output as 8- or 16-bit quantities, it is worth considering processing them as 32-bit quantities.

Consider the following three example functions:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int wordinc (int a)

{

return a + 1;

}

short shortinc (short a)

{

return a + 1;

}

char charinc (char a)

{

return a + 1;

}

The results will be identical, but the first code segment will run faster than others.

## Pointers

If possible, we should pass structures by reference, that is pass a pointer to the structure, otherwise the whole thing will be copied onto the stack and passed, which will slow things down. I've seen programs that pass structures several Kilo Bytes in size by value, when a simple pointer will do the same thing.

Functions receiving pointers to structures as arguments should declare them as pointer to constant if the function is not going to alter the contents of the structure. As an example:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

void print\_data\_of\_a\_structure ( const Thestruct \*data\_pointer)

{

...printf contents of the structure...

}

This example informs the compiler that the function does not alter the contents (as it is using a pointer to constant structure) of the external structure, and does not need to keep re-reading the contents each time they are accessed. It also ensures that the compiler will trap any accidental attempts by your code to write to the read-only structure and give an additional protection to the content of the structure.

## Pointer chains

Pointer chains are frequently used to access information in structures. For example, a common code sequence is:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

typedef struct { int x, y, z; } Point3;

typedef struct { Point3 \*pos, \*direction; } Object;

void InitPos1(Object \*p)

{

p->pos->x = 0;

p->pos->y = 0;

p->pos->z = 0;

}

However, this code must reload p->pos for each assignment, because the compiler does not know that p->pos->x is not an alias for p->pos. A better version would cache p->pos in a local variable:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

void InitPos2(Object \*p)

{

Point3 \*pos = p->pos;

pos->x = 0;

pos->y = 0;

pos->z = 0;

}

Another possibility is to include the Point3 structure in the Object structure, thereby avoiding pointers completely.

## Conditional Execution

Conditional execution is applied mostly in the body of if statements, but it is also used while evaluating complex expressions with relational (<, ==, > and so on) or boolean operators (&&, !, and so on). Conditional execution is disabled for code sequences which contain function calls, as on function return the flags are destroyed.

It is therefore beneficial to keep the bodies of if and else statements as simple as possible, so that they can be conditionalized. Relational expressions should be grouped into blocks of similar conditions.

The following example shows how the compiler uses conditional execution:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int g(int a, int b, int c, int d)

{

if (a > 0 && b > 0 && c < 0 && d < 0)

// grouped conditions tied up together//

return a + b + c + d;

return -1;

}

As the conditions were grouped, the compiler was able to conditionalize them.

## Boolean Expressions & Range checking

A common boolean expression is used to check whether a variable lies within a certain range, for example, to check whether a graphics co-ordinate lies within a window:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

bool PointInRectangelArea (Point p, Rectangle \*r)

{

return (p.x >= r->xmin && p.x < r->xmax &&

p.y >= r->ymin && p.y < r->ymax);

}

There is a faster way to implement this: (x >= min && x < max) can be transformed into (unsigned)(x-min) < (max-min). This is especially beneficial if min is zero. The same example after this optimization:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

bool PointInRectangelArea (Point p, Rectangle \*r)

{

return ((unsigned) (p.x - r->xmin) < r->xmax &&

(unsigned) (p.y - r->ymin) < r->ymax);

}

## Boolean Expressions & Compares with zero

The Processor flags are set after a compare (i.e. CMP) instruction. The flags can also be set by other operations, such as MOV, ADD, AND, MUL, which are the basic arithmetic and logical instructions (the data processing instructions). If a data processing instruction sets the flags, the N and Z flags are set the same way as if the result was compared with zero. The N flag indicates whether the result is negative, the Z flag indicates that the result is zero.

The N and Z flags on the processor correspond to the signed relational operators x < 0, x >= 0, x == 0, x != 0, and unsigned x == 0, x != 0 (or x > 0) in C.

Each time a relational operator is used in C, the compiler emits a compare instruction. If the operator is one of the above, the compiler can remove the compare if a data processing operation preceded the compare. For example:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int aFunction(int x, int y)

{

if (x + y < 0)

return 1;

else

return 0;

}

If possible, arrange for critical routines to test the above conditions. This often allows you to save compares in critical loops, leading to reduced code size and increased performance. The C language has no concept of a carry flag or overflow flag, so it is not possible to test the C or V flag bits directly without using inline assembler. However, the compiler supports the carry flag (unsigned overflow). For example:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int sum(int x, int y)

{

int res;

res = x + y;

if ((unsigned) res < (unsigned) x) // carry set? //

res++;

return res;

}

## Lazy Evaluation Exploitation

In a if(a>10 && b=4) type of thing, make sure that the first part of the AND expression is the most likely to give a false answer (or the easiest/quickest to calculate), therefore the second part will be less likely to be executed.

## switch() instead of if...else...

For large decisions involving if...else...else..., like this:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

if( val == 1)

dostuff1();

else if (val == 2)

dostuff2();

else if (val == 3)

dostuff3();

It may be faster to use a switch:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

switch( val )

{

case 1: dostuff1(); break;

case 2: dostuff2(); break;

case 3: dostuff3(); break;

}

In the if() statement, if the last case is required, all the previous ones will be tested first. The switch lets us cut out this extra work. If you have to use a big if..else.. statement, test the most likely cases first.

## Binary Breakdown

Break things down in a binary fashion, e.g. do not have a list of:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

if(a==1) {

} else if(a==2) {

} else if(a==3) {

} else if(a==4) {

} else if(a==5) {

} else if(a==6) {

} else if(a==7) {

} else if(a==8)

{

}

Have instead:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

if(a<=4) {

if(a==1) {

} else if(a==2) {

} else if(a==3) {

} else if(a==4) {

}

}

else

{

if(a==5) {

} else if(a==6) {

} else if(a==7) {

} else if(a==8) {

}

}

Or even:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

if(a<=4)

{

if(a<=2)

{

if(a==1)

{

/\* a is 1 \*/

}

else

{

/\* a must be 2 \*/

}

}

else

{

if(a==3)

{

/\* a is 3 \*/

}

else

{

/\* a must be 4 \*/

}

}

}

else

{

if(a<=6)

{

if(a==5)

{

/\* a is 5 \*/

}

else

{

/\* a must be 6 \*/

}

}

else

{

if(a==7)

{

/\* a is 7 \*/

}

else

{

/\* a must be 8 \*/

}

}

}

|  |  |
| --- | --- |
| **http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png** [**Copy Code**](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  **Slow and Inefficient** | **http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png** [**Copy Code**](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  **Fast and Efficient** |
| http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  c=getch();  switch(c){  case 'A':  {  do something;  break;  }  case 'H':  {  do something;  break;  }  case 'Z':  {  do something;  break;  }  } | http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  c=getch();  switch(c){  case 0:  {  do something;  break;  }  case 1:  {  do something;  break;  }  case 2:  {  do something;  break;  }  } |

#### Compare between the two Case statements

## Switch statement vs. lookup tables

The switch statement is typically used for one of the following reasons:

* To call to one of several functions.
* To set a variable or return a value.
* To execute one of several fragments of code.

If the case labels are dense, in the first two uses of switch statements, they could be implemented more efficiently using a lookup table. For example, two implementations of a routine that disassembles condition codes to strings:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

char \* Condition\_String1(int condition) {

switch(condition) {

case 0: return "EQ";

case 1: return "NE";

case 2: return "CS";

case 3: return "CC";

case 4: return "MI";

case 5: return "PL";

case 6: return "VS";

case 7: return "VC";

case 8: return "HI";

case 9: return "LS";

case 10: return "GE";

case 11: return "LT";

case 12: return "GT";

case 13: return "LE";

case 14: return "";

default: return 0;

}

}

char \* Condition\_String2(int condition) {

if ((unsigned) condition >= 15) return 0;

return

"EQ\0NE\0CS\0CC\0MI\0PL\0VS\0VC\0HI\0LS\0GE\0LT\0GT\0LE\0\0" +

3 \* condition;

}

The first routine needs a total of 240 bytes, the second only 72 bytes.

## Loops

Loops are a common construct in most programs; a significant amount of the execution time is often spent in loops. It is therefore worthwhile to pay attention to time-critical loops.

### Loop termination

The loop termination condition can cause significant overhead if written without caution. We should always write count-down-to-zero loops and use simple termination conditions. The execution will take less time if the termination conditions are simple. Take the following two sample routines, which calculate n!. The first implementation uses an incrementing loop, the second a decrementing loop.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int fact1\_func (int n)

{

int i, fact = 1;

for (i = 1; i <= n; i++)

fact \*= i;

return (fact);

}

int fact2\_func(int n)

{

int i, fact = 1;

for (i = n; i != 0; i--)

fact \*= i;

return (fact);

}

As a result, the second one fact2\_func" will be more faster than the first one.

### Faster for() loops

It is a simple concept but effective. Ordinarily, we used to code a simple for() loop like this:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for( i=0; i<10; i++){ ... }

[ i loops through the values 0,1,2,3,4,5,6,7,8,9 ]

If we needn't care about the order of the loop counter, we can do this instead:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for( i=10; i--; ) { ... }

Using this code, i loops through the values 9,8,7,6,5,4,3,2,1,0, and the loop should be faster.

This works because it is quicker to process i-- as the test condition, which says "Is i non-zero? If so, decrement it and continue". For the original code, the processor has to calculate "Subtract i from 10. Is the result non-zero? If so, increment i and continue.". In tight loops, this makes a considerable difference.

The syntax is a little strange, put is perfectly legal. The third statement in the loop is optional (an infinite loop would be written as for( ; ; )). The same effect could also be gained by coding:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for(i=10; i; i--){}

or (to expand it further):

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for(i=10; i!=0; i--){}

The only things we have to be careful of are remembering that the loop stops at 0 (so if it is needed to loop from 50-80, this wouldn't work), and the loop counter goes backwards. It's easy to get caught out if your code relies on an ascending loop counter.

We can also use register allocation, which leads to more efficient code elsewhere in the function. This technique of initializing the loop counter to the number of iterations required and then decrementing down to zero, also applies to while and do statements.

### Loop jamming

Never use two loops where one will suffice. But if you do a lot of work in the loop, it might not fit into your processor's instruction cache. In this case, two separate loops may actually be faster as each one can run completely in the cache. Here is an example.

|  |  |
| --- | --- |
| http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  //Original Code :  for(i=0; i<100; i++){  stuff();  }    for(i=0; i<100; i++){  morestuff();  } | http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  //It would be better to do:  for(i=0; i<100; i++){  stuff();  morestuff();  } |

### Function Looping

Functions always have a certain performance overhead when they are called. Not only does the program pointer have to change, but in-use variables have to be pushed onto a stack, and new variables allocated. There is much that can be done then to the structure of a program's functions in order to improve a program's performance. Care must be taken though to maintain the readability of the program whilst keeping the size of the program manageable.

If a function is often called from within a loop, it may be possible to put that loop inside the function to cut down the overhead of calling the function repeatedly, e.g.:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for(i=0 ; i<100 ; i++)

{

func(t,i);

}

-

-

-

void func(int w,d)

{

lots of stuff.

}

Could become....

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

func(t);

-

-

-

void func(w)

{

for(i=0 ; i<100 ; i++)

{

//lots of stuff.

}

}

### Loop unrolling

Small loops can be unrolled for higher performance, with the disadvantage of increased code size. When a loop is unrolled, a loop counter needs to be updated less often and fewer branches are executed. If the loop iterates only a few times, it can be fully unrolled, so that the loop overhead completely disappears.

This can make a **big** difference. It is well known that unrolling loops can produce considerable savings, e.g.:

|  |  |
| --- | --- |
| http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  for(i=0; i<3; i++){  something(i);  }    //is less efficient than | http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)  something(0);  something(1);  something(2); |

because the code has to check and increment the value of i each time round the loop. Compilers will often unroll simple loops like this, where a fixed number of iterations is involved, but something like:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

for(i=0;i< limit;i++) { ... }

is unlikely to be unrolled, as we don't know how many iterations there will be. It is, however, possible to unroll this sort of loop and take advantage of the speed savings that can be gained.

The following code (Example 1) is obviously much larger than a simple loop, but is much more efficient. The block-size of 8 was chosen just for demo purposes, as any suitable size will do - we just have to repeat the "loop-contents" the same amount. In this example, the loop-condition is tested once every 8 iterations, instead of on each one. If we know that we will be working with arrays of a certain size, you could make the block size the same size as (or divisible into the size of) the array. But, this block size depends on the size of the machine's cache.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

//**Example 1**

#include<STDIO.H>

#define BLOCKSIZE (8)

void main(void)

{

int i = 0;

int limit = 33; /\* could be anything \*/

int blocklimit;

/\* The limit may not be divisible by BLOCKSIZE,

\* go as near as we can first, then tidy up.

\*/

blocklimit = (limit / BLOCKSIZE) \* BLOCKSIZE;

/\* unroll the loop in blocks of 8 \*/

while( i < blocklimit )

{

printf("process(%d)\n", i);

printf("process(%d)\n", i+1);

printf("process(%d)\n", i+2);

printf("process(%d)\n", i+3);

printf("process(%d)\n", i+4);

printf("process(%d)\n", i+5);

printf("process(%d)\n", i+6);

printf("process(%d)\n", i+7);

/\* update the counter \*/

i += 8;

}

/\*

\* There may be some left to do.

\* This could be done as a simple for() loop,

\* but a switch is faster (and more interesting)

\*/

if( i < limit )

{

/\* Jump into the case at the place that will allow

\* us to finish off the appropriate number of items.

\*/

switch( limit - i )

{

case 7 : printf("process(%d)\n", i); i++;

case 6 : printf("process(%d)\n", i); i++;

case 5 : printf("process(%d)\n", i); i++;

case 4 : printf("process(%d)\n", i); i++;

case 3 : printf("process(%d)\n", i); i++;

case 2 : printf("process(%d)\n", i); i++;

case 1 : printf("process(%d)\n", i);

}

}

}

### Population count - counting the number of bits set

This example 1 efficiently tests a single bit by extracting the lowest bit and counting it, after which the bit is shifted out. The example 2 was first unrolled four times, after which an optimization could be applied by combining the four shifts of n into one. Unrolling frequently provides new opportunities for optimization.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

//**Example - 1**

int countbit1(uint n)

{

int bits = 0;

while (n != 0)

{

if (n & 1) bits++;

n >>= 1;

}

return bits;

}

//**Example - 2**

int countbit2(uint n)

{

int bits = 0;

while (n != 0)

{

if (n & 1) bits++;

if (n & 2) bits++;

if (n & 4) bits++;

if (n & 8) bits++;

n >>= 4;

}

return bits;

}

### Early loop breaking

It is often not necessary to process the entirety of a loop. For example, if we are searching an array for a particular item, break out of the loop as soon as we have got what we need. Example: this loop searches a list of 10000 numbers to see if there is a -99 in it.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

found = FALSE;

for(i=0;i<10000;i++)

{

if( list[i] == -99 )

{

found = TRUE;

}

}

if( found ) printf("Yes, there is a -99. Hooray!\n");

This works well, but will process the entire array, no matter where the search item occurs in it. A better way is to abort the search as soon as we've found the desired entry.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

found = FALSE;

for(i=0; i<10000; i++)

{

if( list[i] == -99 )

{

found = TRUE;

break;

}

}

if( found ) printf("Yes, there is a -99. Hooray!\n");

If the item is at, say position 23, the loop will stop there and then, and skip the remaining 9977 iterations.

## Function Design

It is a good idea to keep functions small and simple. This enables the compiler to perform other optimizations, such as register allocation, more efficiently.

### Function call overhead

Function call overhead on the processor is small, and is often small in proportion to the work performed by the called function. There are some limitations up to which words of arguments can be passed to a function in registers. These arguments can be integer-compatible (char, shorts, ints and floats all take one word), or structures of up to four words (including the 2-word doubles and long longs). If the argument limitation is 4, then the fifth and subsequent words are passed on the stack. This increases the cost of storing these words in the calling function and reloading them in the called function.

In the following sample code:

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

int f1(int a, int b, int c, int d) {

return a + b + c + d;

}

int g1(void) {

return f1(1, 2, 3, 4);

}

int f2(int a, int b, int c, int d, int e, int f) {

return a + b + c + d + e + f;

}

ing g2(void) {

return f2(1, 2, 3, 4, 5, 6);

}

the fifth and sixth parameters are stored on the stack in g2, and reloaded in f2, costing two memory accesses per parameter.

### Minimizing parameter passing overhead

To minimize the overhead of passing parameters to functions:

* Try to ensure that small functions take four or fewer arguments. These will not use the stack for argument passing.
* If a function needs more than four arguments, try to ensure that it does a significant amount of work, so that the cost of passing the stacked arguments is outweighed.
* Pass pointers to structures instead of passing the structure itself.
* Put related arguments in a structure, and pass a pointer to the structure to functions. This will reduce the number of parameters and increase readability.
* Minimize the number of long parameters, as these take two argument words. This also applies to doubles if software floating-point is enabled.
* Avoid functions with a parameter that is passed partially in a register and partially on the stack (split-argument). This is not handled efficiently by the current compilers: all register arguments are pushed on the stack.
* Avoid functions with a variable number of parameters. Those functions effectively pass all their arguments on the stack.

### Leaf functions

A function which does not call any other functions is known as a leaf function. In many applications, about half of all function calls made are to leaf functions. Leaf functions are compiled very efficiently on every platform, as they often do not need to perform the usual saving and restoring of registers. The cost of pushing some registers on entry and popping them on exit is very small compared to the cost of the useful work done by a leaf function that is complicated enough to need more than four or five registers. If possible, we should try to arrange for frequently-called functions to be leaf functions. The number of times a function is called can be determined by using the profiling facility. There are several ways to ensure that a function is compiled as a leaf function:

* Avoid calling other functions: this includes any operations which are converted to calls to the C-library (such as division, or any floating-point operation when the software floating-point library is used).
* Use \_\_inline for small functions which are called from it (inline functions discussed next).

### Inline functions

Function inlining is disabled for all debugging options. Functions with the keyword \_\_inline results in each call to an inline function being substituted by its body, instead of a normal call. This results in faster code, but it adversely affects code size, particularly if the inline function is large and used often.

http://www.codeproject.com/images/minus.gifCollapsehttp://www.codeproject.com/images/copy_16.png [Copy Code](http://www.codeproject.com/KB/cpp/C___Code_Optimization.aspx)

\_\_inline int square(int x) {

return x \* x;

}

#include <MATH.H>

double length(int x, int y){

return sqrt(square(x) + square(y));

}

There are several advantages to using inline functions:

* No function call overhead.

As the code is substituted directly, there is no overhead, like saving and restoring registers.

* Lower argument evaluation overhead.

The overhead of parameter passing is generally lower, since it is not necessary to copy variables. If some of the parameters are constants, the compiler can optimize the resulting code even further.

The big disadvantage of inline functions is that the code sizes increase if the function is used in many places. This can vary significantly depending on the size of the function, and the number of places where it is used.

It is wise to only inline a few critical functions. Note that when done wisely, inlining may decrease the size of the code: a call takes usually a few instructions, but the optimized version of the inlined code might translate to even less instructions.

## Using Lookup Tables

A function can often be approximated using a lookup table, which increases performance significantly. A table lookup is usually less accurate than calculating the value properly, but for many applications, this does not matter.

Many signal processing applications (for example, modem demodulator software) make heavy use of sin and cos functions, which are computationally expensive to calculate. For real-time systems where accuracy is not very important, sin/cos lookup tables might be essential. When using lookup tables, try to combine as many adjacent operations as possible into a single lookup table. This is faster and uses less space than multiple lookup tables.

## Floating-Point Arithmetic

Although floating point operations are time consuming for any kind of processors, sometimes we need to used it in case of implementing signal processing applications. However, when writing floating-point code, keep the following things in mind:

* Floating-point division is slow.

Division is typically twice as slow as addition or multiplication. Rewrite divisions by a constant into a multiplication with the inverse (For example, x = x / 3.0 becomes x = x \* (1.0/3.0). The constant is calculated during compilation.).

* Use floats instead of doubles.

Float variables consume less memory and fewer registers, and are more efficient because of their lower precision. Use floats whenever their precision is good enough.

* Avoid using transcendental functions.

Transcendental functions, like sin, exp and log are implemented using series of multiplications and additions (using extended precision). As a result, these operations are at least ten times slower than a normal multiply.

* Simplify floating-point expressions.

The compiler cannot apply many optimizations which are performed on integers to floating-point values. For example, 3 \* (x / 3) cannot be optimized to x, since floating-point operations generally lead to loss of precision. Even the order of evaluation is important: (a + b) + c is not the same as a + (b + c). Therefore, it is beneficial to perform floating-point optimizations manually if it is known they are correct.

However, it is still possible that the floating performance will not reach the required level for a particular application. In such a case, the best approach may be to change from using floating-point to fixed point arithmetic. When the range of values needed is sufficiently small, fixed-point arithmetic is more accurate and much faster than floating-point arithmetic.

## Misc tips

In general, savings can be made by trading off memory for speed. If you can cache any often used data rather than recalculating or reloading it, it will help. Examples of this would be sine/cosine tables, or tables of pseudo-random numbers (calculate 1000 once at the start, and just reuse them if you don't need truly random numbers).

* Avoid using ++ and -- etc. within loop expressions. E.g.: while(n--){}, as this can sometimes be harder to optimize.
* Minimize the use of global variables.
* Declare anything within a file (external to functions) as static, unless it is intended to be global.
* Use word-size variables if you can, as the machine can work with these better (instead of char, short, double, bit fields etc.).
* Don't use recursion. Recursion can be very elegant and neat, but creates many more function calls which can become a large overhead.
* Avoid the sqrt() square root function in loops - calculating square roots is very CPU intensive.
* Single dimension arrays are faster than multi-dimension arrays.
* Compilers can often optimize a whole file - avoid splitting off closely related functions into separate files, the compiler will do better if it can see both of them together (it might be able to inline the code, for example).
* Single precision math may be faster than double precision - there is often a compiler switch for this.
* Floating point multiplication is often faster than division - use val \* 0.5 instead of val / 2.0.
* Addition is quicker than multiplication - use val + val + val instead of val \* 3. puts() is quicker than printf(), although less flexible.
* Use #defined macros instead of commonly used tiny functions - sometimes the bulk of CPU usage can be tracked down to a small external function being called thousands of times in a tight loop. Replacing it with a macro to perform the same job will remove the overhead of all those function calls, and allow the compiler to be more aggressive in its optimization..
* Binary/unformatted file access is faster than formatted access, as the machine does not have to convert between human-readable ASCII and machine-readable binary. If you don't actually need to read the data in a file yourself, consider making it a binary file.
* If your library supports the mallopt() function (for controlling malloc), use it. The MAXFAST setting can make significant improvements to code that does a lot of malloc work. If a particular structure is created/destroyed many times a second, try setting the mallopt options to work best with that size.

Last, but definitely not least - turn compiler optimization on! Seems obvious, but is often forgotten in that last minute rush to get the product out on time. The compiler will be able to optimize at a much lower level than can be done in the source code, and perform optimizations specific to the target processor.