Getting the Least Out of Your C Compiler

Class #508, Embedded Systems Conference San Francisco 2001

Jakob Engblom
IAR Systems
Box 23051
SE-750 23 Uppsala
Sweden
email: `jakob.engblom@iar.se`

Using only the internal program and data memory of a microcontroller can save large costs in embedded systems design. This requires, however, that the program fits into the memory—which is not always easy to accomplish.

This article discusses how to help a modern, highly optimizing C compiler generate small code, while maintaining the portability and readability advantages offered by C. In order to facilitate an understanding of what a compiler likes and does not like, we will give an inside view on how a compiler operates.

Many established truths and tricks are invalidated when using modern compilers. We will demonstrate some of the more common mistakes and how to avoid them, and give a catalog of good coding techniques. An important conclusion is that code that is easy for a human to understand is usually also compiler friendly, contrary to hacker tradition.

1 Introduction

A C compiler is a basic tool for most embedded systems programmers. It is the tool by which the ideas and algorithms in your application (expressed as C source code) are transformed into machine code executable by your target processor. To a large extent, the C compiler determines how large the executable code for the application will be.

The C language is well suited for low-level programming. It was designed for coding operating systems, which has left the imprint of powerful pointer handling, bit manipulation power not found in other high-level languages, and target-dependent type sizes to generate the best possible code for a certain target.

The semantics of C are specified by the ISO/ANSI C Standard [1]. The standard makes an admirable job of specifying the language without unduly constraining an implementation of the language. For example, compared to Java, the C standard gives the compiler writer some flexibility in the size of types and the precise order and implementation of calculations. The result is that there are C compilers for all available processors, from the humblest 8-bitter to the proudest supercomputers.

A compiler performs many transformations on a program in order to generate the best possible code. Examples of such transformations are storing values in registers instead of memory, removing code which does nothing useful, reordering computations in a more efficient order, and replacing arithmetic operations by cheaper operations.

To most programmers of embedded systems, the case that a program does not *quite* fit into the available memory is a familiar phenomenon. Recoding parts of an application in assembly language or throwing out functionality may seem to be the only alternatives, while the solution could be as simple as rewriting the C code in a more compiler-friendly manner.

In order to write code that is compiler friendly, you need to have a working understanding of compilers. Some simple changes to a program, like changing the data type of a frequently-accessed variable, can have a big impact on code size while other changes have no effect at all. Having an idea of what a compiler can and cannot do makes such optimization work much easier.

This paper (and associated presentation) will try to convey a feeling for how a modern C compiler works and how you can help it compile your code in the best possible way.

2 Modern C Compilers

Assembly programs specify both what, how, and the precise order in which calculations should be carried out. A C program, on the other hand, only specifies the calculations that should be performed. With some restrictions, the order and the technique used to realize the calculations are up to the compiler.

The compiler will look at the code and try to understand what is being calculated. It will then generate the best possible code, given the information it has managed to obtain, locally within a single statement and also across entire functions and sometimes even whole programs.

2.1 The Structure of a Compiler

In general, a program is processed in six main steps in a modern compiler (not all compilers follow this blueprint completely, but as a conceptual guide it is sufficient):

- **Parser**: The conversion from C source code to an intermediate language.
- **High-level optimization**: Optimizations on the intermediate code.
- **Code generation**: Generation of target machine code from the intermediate code.
- **Low-level optimization**: Optimizations on the machine code.
- **Assembly**: Generation of an object file that can be linked from the target machine code.
- **Linking**: Linking of all code for a program into an executable or downloadable file.

The parser parses the C source code, checking the syntax and generating error messages if syntactical errors are found in the source. If no errors are found, the parser then generates intermediate code (an internal representation of the parsed code), and compilation proceeds with the first optimization pass.

The high-level optimizer transforms the code to make it better. The optimizer has a large number of transformations available and will perform them in various passes, possibly repeating some passes.

Note that we use the word “transformation” and not “optimization”. “Optimization” is a bit of a misnomer. It conveys the intuition that a change always improves a program and that we actually find optimal solutions, while in fact optimal solutions are very expensive or even impossible to find (undecidable, in computer science lingo). To ensure reasonable compilation times and termination of the compilation process, the compiler has to use heuristic methods (“good guesses”). Transforming a program is a highly non-linear activity, where different orderings of transformations will yield different results, and some transformations may actually make the code worse. Piling on more “optimizations” will not necessarily yield better code.

When the high-level optimizer is done, the code generator transforms the intermediate code to the target processor instruction set. This stage is performed, piece-by-piece, on the intermediate code from the optimizer, and the compiler will try to do smart things on the level of a single expression or statement, but not across several statements.

The code generator will also have to account for any differences between the C language and the target processor. For example, 32-bit arithmetic will have to be broken down to 8-bit arithmetic for a small embedded target (like an Intel 8051, Motorola 68HC08, Samsung SAM8, or Microchip PIC).

A very important part of code generation is allocating registers to variables. The goal is to keep as many values as possible in registers, since register-based operations are typically faster and smaller than memory-based operations.

After the code generator is done, another phase of optimization takes place, where transformations are performed on the target code. The low-level optimizer will clean up after the code generator (which sometimes makes suboptimal choices), and perform more transformations. There are many transformations which can only be applied on the target code, and some which are repeated from the high-level phase, but on a lower level. For example, transformations like removing a “clear carry” instruction if we already know that the carry flag is zero are only possible at the target code level, since the flags are not visible before code generation.

After the low-level optimizer is finished, the code is sent to an assembler and output to an object file.

All the object files of a program are then linked to produce a final binary executable ROM image (in some format appropriate for the target). The linker may also perform some optimizations, for example by discarding unused functions.

Thus, one can see that the seemingly simple task of compiling a C program is actually a rather long and winding road through a highly complex system. Different transformations may interact, and a local improvement may be worse for the whole program. For example, an expression can typically be evaluated more efficiently if given more temporary registers. Taking a local view, it thus seems to be a good idea to provide as many registers as necessary. A global effect, however, is that variables in registers may have to be spilled to memory, which could be more expensive than evaluating the expression with fewer registers.

2.2 The Meaning of a Program

Before the compiler can apply transformations to a program, it must analyze the code to determine which transformations are legal and likely to result in improvements. The legality of transformations is determined by the semantics laid down in the C standard.

The most basic interpretation of a C program is that only statements that have side effects or compute values used for performing side-effects need to be kept in the program. Side effects are any effects of an expression that change the global state of the program. Examples which are generally considered to be side effects are writing to a screen, accessing global variables, reading volatile variables, and calling unknown functions.

The calculations between the side-effects are carried out according to the principle of “do what I say, not what I mean”. The compiler will try to rewrite each expression into the most efficient form possible, but a rewrite is only possible if the result of the rewritten code is the same as the original expression. The C standard defines what is considered “the same”, and sets the limits of allowable optimizations.

2.3 Basic Transformations

A modern compiler performs a large number of basic transformations that act locally, like folding constant expressions, replacing expensive operations by cheaper ones (“strength reduction”), finding and removing redundant calculations, and moving invariant calculations outside of loops. The compiler can do most mechanical improvements just as well as a human programmer, but without tiring or making a mistake. The table below shows (in C form for readability) some typical basic transformations performed by a modern C compiler.

Before Transformation	After Transformation
`unsigned short int a;` `a /= 8;`	`unsigned short int a;` `a >>= 3;  /* shift replaced divide*/`
`a *= 2;`	`a += a;  /* multiply to add  */` `a <<= 1;  /* or using a shift */`

`a = b + c * d;`
`e = f + c * d;`
	`temp = c * d;  /* common expression */`
	`a = b + temp;  /* saves one multiply */`
	`e = f + temp;`

`a = b * 1;`	`a = b;  /* x*1 == x */`

`a = 17;`
`b = 56 + ( 2 * a );`
	`a = 17;`
	`b = 90;  /* constant value evaluated */`
	`/* at compile time  */`

`#define BITNO 4` `port |= ( 1 << BITNO);`	`port |= 0x10;`

`if(a > 10)`
`{`
`  b = b * c + k;`
`  if(a < 5)`
`  a+=6;`
`}`
	`if(a > 10)`
	`{`
	`  b = b * c + k;`
	`  /* unreachable code removed */`
	`}`

`a = b * c + k;` `a = k + 7;`	`/* useless computation removed */` `a = k + 7;`

`for(i=0; i<10; i++)`
`{`
`  b = k * c;`
`  p[i] = b;`
`}`
	`b = k * c;  /* constant code moved */`
	`/* outside the loop  */`
	`for(i=0; i<10; i++)`
	`{`
	`  p[i] = b;`
	`}`

All code that is not considered useful—according to the definition in the previous section—is removed. This removal of unreachable or useless computations can cause some unexpected effects. An important example is

that empty loops are completely discarded, making “empty delay loops” useless. The code shown below stopped working properly when upgrading to a modern compiler that removed useless computations:

Code that Stopped Working	… After Compiler Optimizations

`void delay(unsigned int time)`
`{`
`  unsigned int i;`
`  for (i=0; i<time; i++)`
`  ;`
`  return;`
`}`
 	
`void InitHW(void)`
`{`
`  /* Highly timing-dependent code */`
`  OUT_SIGNAL (0x20);`
`  delay(120);`
`  OUT_SIGNAL (0x21);`
`  delay(121);`
`  OUT_SIGNAL (0x19);`
`  delay(120);`
`  OUT_SIGNAL (0x38);`
`  delay(147);`
`  OUT_SIGNAL (0x0C);`
`}`
	`void delay(unsigned int time)`
	`{`
	`  /* loop removed */`
	`  return;`
	`}`
	
	`void InitHW(void)`
	`{`
	`  /* Delays do not last long here… */`
	`  OUT_SIGNAL (0x20);`
	`  delay(120);`
	`  OUT_SIGNAL (0x21);`
	`  delay(121);`
	`  OUT_SIGNAL (0x19);`
	`  delay(120);`
	`  OUT_SIGNAL (0x38);`
	`  delay(147);`
	`  OUT_SIGNAL (0x0C);`
	`}`

Note that a compiler cannot in general make function calls into common subexpressions. Two subsequent calls to the same function with the same argument will generate two function calls, since the compiler does not in general know what happens inside the called function (for example, it might perform side-effects). If you intend to evaluate a function only once, write it once!

Bad Example	Good Example

`void bad()`
`{`
`  /* Two calls to foo */`
`  if (foo(12) && SomeCondition)`
`  {`
`  ...`
`  }`
`  if (!foo(12) || SomeOtherCondition)`
`  {`
`  ...`
`  }`
`}`
	`void good()`
	`{`
	`  /* One call to foo */`
	`  r = foo(12)`
	`  if ( r && SomeCondition)`
	`  {`
	`  ...`
	`  }`
	`  if ( !r || SomeOtherCondition)`
	`  {`
	`  ...`
	`  }`
	`}`

2.4 Register Allocation

Processors usually get better performance and smaller code when calculations are performed using registers, instead of memory. This means that the compiler will try to assign the variables in a function to registers. A local variable or parameter will not need any RAM allocated at all if the variable can be kept in registers for the duration of the function.

If there are more variables than registers available, the compiler needs to decide which of the variables to keep in registers, and which to put in memory. This is the problem of *register allocation*, and it cannot be solved optimally. Instead, heuristic techniques are used. The algorithms used can be quite sensitive, and even small changes to a function may considerably alter the register allocation.

Note that a variable needs only occupy a register while it is being used. If a variable is only used in part of a function, it will be register allocated in that part, but it will not exist in the rest of the function. This explains that a debugger sometimes tells you that a variable is “optimized away at this point”.

The register allocator is limited by the language rules of C—for example, global variables have to be written back to memory when calling other functions, since they can be accessed by the called function, and all changes must be visible to all functions. Between the function calls, the variables can be kept in registers. The

same applies to local variables that have their address taken (for instance as parameters to functions like `scanf()`, which require pointers to local variables to store the values read).

Note that there are times when you *do not* want variables to be register allocated. For example, reading an I/O port or spinning on a lock, you want each read in the source code to be made from memory, since the variable can be changed outside the control of your program. This is where the `volatile` keyword is to be used. It signals to the compiler that the variable should not ever be allocated in registers, but read from memory (or written) each time it is accessed.

In general, only simple values like integers, floats, and pointers are considered for register allocation. Arrays have to reside in memory since they are designed to be accessed through pointers, and structures are usually too large. Also, on small processors, large values like 32-bit integers and floats may be hard to allocate to registers.

2.5 Function Calls

As assembly programmers well know, calling a function written in a high-level language can be rather burdensome. The calling function must save global variables back to memory, make sure to move local variables to the registers that survive the call (or save to the stack), and some parameters may have to be pushed on the stack. Inside the called function, registers will have to be saved, parameters taken off the stack, and space allocated on the stack for local variables. For large functions with many parameters and variables, the effort required for a call can be quite large.

Modern compilers do their best, however, to reduce the cost of a function call, especially the use of stack space. A number of registers will be designated for parameters, so that short parameter lists will most likely be passed entirely in registers. Likewise, the return value will be put in a register, and local variables will only be put on the stack if they cannot be allocated to registers.

The number of register parameters will vary wildly between different compilers and architecture. Note, however, that research has found that passing just 3 bytes in registers would cover about 87% of all functions for a set of embedded 8- and 16-bit programs [2].

Note also that just like for register allocation, only small parameter types will be passed in registers. Arrays are always passed as pointers to the array (C semantics dictate that), and structures are usually copied to the stack and the structure parameter changed to a pointer to a structure. That pointer might be passed in a register, however. It is a good rule to always use pointers to structures as parameters and not the structures themselves.

2.6 Function Inlining

It is good programming practice to break out common pieces of computation and accesses to shared data structures into (small) functions. This, however, brings with it the cost of calling a function each time something should be done. In order to mitigate this cost, the compiler transformation of *function inlining* has been developed. Inlining a function means that a copy of the code for the function is placed in the calling function, and the call is removed.

Inlining is a very efficient method to speed up the code, since the function call overhead is avoided but the same computations carried out. Many programmers do this manually by using preprocessor macros for common pieces of code instead of functions, but macros lack the type checking of functions and produce harder-to-find bugs. The executable code will often grow as a result of inlining, since code is being copied into several places.

Inlining may also help shrink the code: for small functions, the code size cost of a function call might be bigger than the code for the function. In this case, inlining a function will actually save code size (as well as speeding up the program).

The main problem when inlining for size is to estimate the gains in code size (when optimizing for speed, the gain is almost guaranteed). Since inlining in general increases the code size, the inliner has to be quite conservative. The effect of inlining on code size cannot be exactly determined, since the code of the calling function is disturbed, with non-linear effects.

To reduce the code size, the ideal would be to inline all calls to a function, which allows us to remove the function from the program altogether. This is only possible if all calls are known, i.e. are placed in the same source file as the function, and the function is marked `static`, so that it cannot be seen from other files. Otherwise, the function will have to be kept (even though it might still be inlined at some calls), and we rely on the linker to remove it if it is not called. However, since this decreases the likely gain from inlining, we are less likely to inline such a function.

2.7 Low-Level Code Compression

A common transformation on the target code level is to find common sequences of instructions from several functions, and breaking them out into subroutines [3]. This transformation can be very effective at shrinking the executable code of a program, at the cost of performing more jumps (note that this transformation only introduces machine-level subroutine calls and not full-strength function calls). Experience shows a gain from 10 to 30% for this transformation.

2.8 Linker

The linker should be considered an integral part of the compilation system, since there are some optimizations that are performed in the linker. The most basic embedded-systems linker should remove all unused functions and variables from a program, and only add those parts of the standard libraries that are actually used. The granularity at which program parts are discarded varies, from files or library modules down to individual functions or even snippets of code. The smaller the granularity, the better the linker.

Some linkers also perform post-compilation transformations on the program. A favorite is to extend the low-level transformation of breaking out common code sequences to work across the whole program, with corresponding potential gains in code compression.

2.9 Controlling Compiler Optimization

Compiler writers are hard-working engineers with the ambition to make the best possible compiler. They try to find the combination and sequence of transformations that will perform best across a wide range of inputs. Since they cannot be certain that the compiler will work optimally for every program thrown at it, they leave some room for control to the user.

A compiler can be instructed to compile a program with different goals, usually speed or size. For each setting, a set of transformations has been selected that tend to work towards the goal—maximal speed (minimal execution time) or minimal size. The settings should be considered approximate. To give better control, some compilers allow individual transformations to be enabled or disabled.

For size optimization, the compiler uses a combination of transformations that tend to generate smaller code, but it might fail in some cases, due to the characteristics of the compiled program. This means that one should always explore the effects of the optimization settings on a given program.

As an example, the fact that an inliner is more aggressive for speed optimization makes some programs smaller on the speed setting than on the size setting. The following example data demonstrates this; the two programs were compiled with the same version of the same compiler, using the same memory and data model settings, but optimizing for speed or size:

Program 1	Speed optimization	1301 bytes
	Size optimization	1493 bytes
Program 2	Speed optimization	20432 bytes
	Size optimization	16830 bytes

Program 1 gets slightly smaller with speed optimization, while program 2 is considerably larger, an effect we traced to the fact that the inliner was lucky on program 1.

The conclusion is that one should always try to compile a program with different optimization settings and see what happens. Some compilers allow you to adjust the aggressiveness of individual optimizations—explore that possibility, especially for inlining.

It is often worthwhile to use different compilation settings for different files in a project: put the code that must run very quickly into a separate file and compile that for minimal execution time (maximum speed), and the rest of the code for minimal code size. This will give a small program, which is still fast enough where it matters. Some compiler allow different optimization settings for different functions in the same source file using `#pragma` directives.

2.10 Memory Model

An embedded micro is usually available in several variants ("derivatives"), each with a different amount of program and data memory. For smaller chips, the fact that the amount of memory that can be addressed is limited can be exploited by the compiler to generate smaller code. An 8-bit pointer uses less code memory than a 24-bit banked pointer where software has to switch banks before each access. This goes for code as well as data.

For example, some Atmel AVR chips have a code area of only 8 kB, which allows a small jump with an offset of +/- 4 kB to reach all code memory, using wrap-around to jump from high addresses to low addresses. Taking advantage of this yields smaller and faster code.

The capability of the target chip can be communicated to the compiler using a *memory model* option. There are usually several different memory models available, ranging from "small" up to "huge". In general, function calls get more expensive as the amount of code allowed increases, and data pointers get bigger and more expensive as the amount of accessible data increases.

Use the smallest model that fits your target chip and application—this might give you large savings in code size. Bigger is not always better…

3 Selecting Data Types

The previous section discussed how a compiler works. Now, let's have a look at how a programmer can help the compiler generate good assembly code by writing compiler-friendly C code.

The first item to consider is data type selection. This can have a large impact on code size and code speed, and must be considered carefully. In the concept of data types, we also include pointers to different memory spaces, and placement of variables in memory.

3.1 Use the Right Data Size

The semantics of C state that all calculations should have the same result as if all operands were cast to `int` and the operation performed on `int`[1]. If the result is to be stored in a variable of a smaller type like `char`, the result is cast down. On any decent 8-bit micro compiler, this process is short-circuited where appropriate, and the entire expression calculated using `char` or `short`.

Thus, the size of a data item to be processed should be appropriate for the CPU used. If an unnatural size is chosen, the code generated might get much worse. For example, on an 8-bit micro, accessing and calculating 8-bit data is very efficient. Working with 32-bit values will generate much bigger code and run more slowly, and should only be considered when the data being manipulated need all 32 bits. Using big values also increases the demand for registers for register allocation, since a 32-bit value will require four 8-bit registers to be stored.

On a 32-bit processor, working with smaller data might be inefficient, since the registers are 32 bits. The results of calculations will need to be cast down if the storing variable type is smaller than 32 bits, which introduces shift, mask, and sign-extend operations in the code (depending on how smaller types are represented). On such machines, 32-bit integers should be used for as many variables as possible. `char`s and `short`s should only be used when the precise number of bits are needed (like when doing I/O), or when big types would use too much memory (for example, an array with a large number of elements).

The "`int`" type in C is intended to be the best choice when one does not care about the range of a variable. However, since the C language defines `int` to be at least 16 bits, `int` might not be the best choice for a

[1] Unless some operand has a larger range than `int`, this might be the case for `long int`.

small 8-bit micro. A portable solution to the data size problem is to define a set of types with minimal guaranteed range, and then change the definition depending on the target. For example, "`best_int8_t`" is guaranteed to cover the range –128 to + 127, but might be bigger than 8 bits if that is more efficient:

For 8-bit machine	For 32-bit machine

`/* char is best type for 8 bits */`
`typedef signed char` **`best_int8_t`**`;`
`typedef signed short` **`best_int16_t`**`;`
	`/* int is best for calculations */`
	`typedef int` **`best_int8_t`**`;`
	`typedef int` **`best_int16_t`**`;`

This solution should not be used when the actual size of the data is important (reading and writing I/O, for example).

3.2 (Un)signed When You Mean It

The "signedness" of a variable will affect the code generated. For operations like division, which may not be directly supported in the instruction set, there might be significant differences in the code generated. The language rules make it implementation-dependent how division involving negative numbers and right-shifting of signed values work, but everybody expects division to truncate towards zero and right-shift to be arithmetic shift (preserving the sign), which in effect locks the semantics.

Considering this, when replacing constant division by shift, special consideration has to be given to signed variables. The result is an extra test and jump, which would not be necessary for an unsigned variable, as in the following example:

Source code	Optimized Code

`int a;`
`a /= 2;`
	`int a;`
	`if( a < 0)  /* compensate for */`
	`  a++;  /* sign of a  */`
	`a >>= 1;`

`unsigned int a;` `a /= 2;`	`unsigned int a;` `a >>= 1;`

The conclusion is that, if you think about a value as never going below zero, make it unsigned. If the purpose of a variable is to manipulate it as bits, make it unsigned. Otherwise, operations like right shifting and masking *might* do strange things to the value of the variable.

3.3 Watch Out for Library Functions

As discussed above, the linker has to bring in all library functions used by a program with the program. This is obvious for C standard library functions like `printf()` and `strcat()`, but there are also large parts of the library which are brought in implicitly when certain types of arithmetic are needed, most notably floating point. Due to the way in which C performs implicit type conversions inside expressions, it is quite easy to inadvertently bring in floating point, even if no floating point variables are being used.

For example, the following code will bring in floating point, since the `ImportantRatio` constant is of floating point type—even if its value would be `1.95*20==39`, and all variables are integers:

Example of Accidental Floating Point

`#define ImportantRatio (1.95*Other)`
`…`
`int temp = a * b + CONSTANT * ImportantRatio;`

If a small change to a program causes a big change in program size, look at the library functions included after linking. Especially floating point and 32-bit integer libraries can be insidious, and creep in due to C implicit casts.

Another way to shrink the code of your program is to use limited versions of standard functions. For instance, the standard `printf()` is a very big function. Unless you really need the full functionality, you should use a limited version that only handles basic formatting or ignores floating point. Note that this should be done at

link time: the source code is the same, but a simpler version is linked. Because the first argument to `printf()` is a string, and can be provided as a variable, it is not possible for the compiler to automatically figure out which parts of the function your program needs.

3.4 Use the Best Pointer Types

A typical embedded micro has several different pointer types, allowing access to memory in variety of ways, from small zero-page pointers to software-emulated generic pointers. It is obvious that using smaller pointer types are better than using bigger, since both the data space required to store them and the manipulating code is smaller for smaller pointers.

However, there may be several pointers of the same size but with different properties, for example two banked 24-bit pointers `huge` and `far`, with the sole difference that `huge` allows objects to cross bank boundaries. This difference makes the code to manipulate `huge` pointers much bigger, since each increment or decrement must check for a bank boundary. Unless you really require very large objects, using the smaller pointer variant will save a lot of code space.

For machines with many disjoint memory spaces (like Microchip PIC and Intel 8051), there might be "generic" pointers that can point to all memory spaces. These pointers might be tempting to use, since they are simple to use, but they carry a cost in that special code is needed before each pointer access to check which memory a pointer points to and performing appropriate actions. Also note that using generic pointers typically brings in some library functions (see Section 3.3).

In summary: use the smallest pointers you can, and avoid any form of generic pointers unless necessary. Remember to check the compiler default pointer type (used for unqualified pointers). In many cases it is a rather large pointer type.

3.5 Try to Avoid Casting

C performs some implicit casts (for example, between floating point and integers, and different sizes of integers), and C programs often contain explicit casts. Performing a cast is in general not free (except for some special cases), and casts should be avoided wherever possible.

Casting from a smaller and a bigger signed integer type will introduce sign extend operations, and casting to and from floating point will force calls to the floating point library. These types of casts can be introduced by mistake, for example by mixing variables of different sizes or mixing floating point and integers in the same expression.

A more acute problem is involved when using function pointers. People used to desktop systems often consider `int` and function pointers interchangeable, which they are not. If you want to store a function pointer, use a variable of function pointer type. It is not uncommon that function pointers are larger than `int`—on a 16-bit machine, `int` will be 16 bits, while code pointers may well be 24 bits. Casting an `int` to a function pointer will lose information in this case. The code below shows an example of this mistake.

Bad Example of Mixing Function Pointers and Integers

```
typedef char (* FuncPtr)(void);     /* pointer to function "char f()" */
typedef unsigned short uint16_t;    /* 16-bit value */

  /*     Macro to call a function… really?                */
  /* The argument (function name) is cast to an integer   */
  /* and then back to a function pointer, and then called */
#define Call(pF) ((FuncPtr)(uint16_t)(*(&pF)))();

char foo( void );   /* a function to call */
void main( void )
{
  Call(foo);   /* can jump to strange places if FuncPtr is > 16 bits */
}
```

A good way to avoid implicit casts is to use consistent `typedef`s for all types used in the application. Otherwise, it is easy to start mixing different-size integers or other types. Some lint-type tools can also be used to check for type consistency.

4 Facilitating Optimizations

The previous section listed some items about selecting and using data types that should be quite obvious. This section will deal with the somewhat more complex issue of how to write your code so that it is easy to understand for the compiler, and so that optimization is allowed to the greatest possible extent.

The basic principle is that "the compiler is your friend". But it is not an AI system that can understand what you are doing… and it is a rather dumb tool many times. If you understand the strengths and weaknesses of the compiler, you will write much better code.

4.1 Use Parameters

Some programmers use global variables to pass information between functions. As discussed above, register allocation has a hard time with global variables. If you want to improve register allocation, use parameters to pass information to a called function. They will often be allocated to registers both in the calling and called function, leading to very efficient calls.

4.2 Do Not Take Addresses

If you take the address of a local variable (the "`&var`" construction), it is less likely to be allocated to a register, since it has to have an address and thus a place in memory (usually on the stack). It also has to be written back to memory before each function call, just like a global variable, since some other function might have gotten hold of the address and is expecting the latest value. Taking the address of a global variable does not hurt as much, since they have to have a memory address anyway.

Thus, you should only take the address of a local variable if you really must (it is very seldom necessary). If the taking of addresses is used to receive return values from called functions (for instance, from `scanf()`), introduce a temporary variable to receive the result, and then copy the value from the temporary to the real variable[2]. This should allow the real variable to be register allocated.

Making a global variable `static` is a good idea (unless it is referred to in another file), since this allows the compiler to know all places where the address is taken, potentially leading to better code.

An example of when not to use the address-of operator is the following, where the use of addresses to access the high byte of a variable will force the variable to the stack. The good way is to use shifts to access parts of values.

Bad example	Good example

`#define highbyte(x) (*((char *)(&x)+1))`
 	
`short a;`
`char  b = highbyte(a);`
	`#define highbyte(x) ((x>>8)&0xFF)`
	
	`short a;`
	`char  b = highbyte(a);`

4.3 Use Function Prototypes

Function prototypes were introduced in ANSI C as a way to improve type checking. The old style of calling functions without first declaring them was considered unsafe, and is also a hindrance to efficient function calls.

If a function is not properly prototyped, the compiler has to fall back on the language rules dictating that all arguments should be promoted to `int` (or `double`, for floating-point arguments). This means that the function call will be much less efficient, since type casts will have to be inserted to convert the arguments. For a desktop machine, the effect is not very noticeable (most things are the size of `int` or `double` already), but for small embedded systems, the effect is potentially great. Problems include ruining register parameter passing (larger values use more registers) and lots of unnecessary type conversion code.

In many cases, the compiler will give you a warning when a function without a prototype is called. Make sure that no such warnings are present when you compile!

---

[2] Note that in C++, reference parameters ("`foo(int &)`") can introduce pointers to variables in a calling function without the syntax of the call showing that the address of a variable is taken.

The old way to declare a function before calling it (Kernighan & Ritchie or "K&R" style) was to leave the parameter list empty, like "`extern void foo()`". This is not a proper ANSI prototype and will not help code generation. Unfortunately, few compilers warn about this by default.

4.4 Read Global Variables into Temporaries

If you are accessing a global variable several times over the life of a function, it might pay to copy the value of the global into a local temporary. This temporary has a much higher chance of being register allocated, and if you call functions, the temporary might remain in registers while a global variable would have to be written to memory. Note that this assumes that you know that the functions called will not modify your variables.

Example

```
unsigned char gGlobal;  /* global variable */

void foo(int x)
{
  unsigned char  ctemp;
  ctemp = gGlobal;  /* should go into register */
  ...
  /* Calculations involving ctemp, i.e. gGlobal */
  bar(z);  /* does not read or write gGlobal, otherwise error */
  /* More calculations on ctemp */
  ...
  gGlobal = ctemp;   /* make sure to remember the result */
}
```

4.5 Group Function Calls

Function calls are bad for register allocation, since they force write-back of global variables to memory and increase the demand for registers (since registers are used for parameters and return values, and the called function is allowed to scratch certain registers). For this reason, it is a good idea to avoid function calls for as long stretches of code as possible. To minimize the effects, try to group function calls together.

For example, the code below shows two different ways of initializing a structure. The second variant will most likely generate better code, since the function calls are grouped together, making the simple values assigned to the other fields much more likely to survive in registers.

Bad example

```
void foo(char a, int b, char c, char d)
{
  struct S s;

  s.A = a;
  s.B = bar(b);
  s.C = c;
  s.D = c;
  s.E = baz(d);
}
```

Good example

```
void foo(char a, int b, char c, char d)
{
  struct S s;

  s.A = a;
  s.C = c;
  s.D = c;
  s.B = bar(b);
  s.E = baz(d);
}
```

Note that grouping function calls has no effect if the functions become inlined.

4.6 Make Local Functions Static

Function inlining is a very effective optimization for reducing the run time and sometimes code size of a program. To facilitate inlining, structure your code so that small functions are located in the same source code file (module) as the callers, and make the functions `static`. Making a function `static` tells the compiler that the function will not be called from outside the module, giving more information to the inliner, enabling it to be more aggressive and make better decisions about when to inline.

Note that only functions located within the same module can be inlined (otherwise, the source code of the function is not available).

A risky technique to take advantage of inlining is to declare a number of small helper functions in a header file and making them `static`. This will give each compiled module its own copy of every function, but since they are small and static, they are quite likely to be inlined. If the inlining succeeds, you might save a lot of code space and gain speed. This approach is similar to declaring `inline` functions in C++ header files, which is also just a hint to the compiler.

4.7 Make File-Global Variables Static

By the same logic as local functions, variables that are global but only used within one file should be made static. This has the double gain of explicitly hiding the variable from other source files and giving the compiler more information about the variable. Since the compiler sees a whole module (file) at the same time, it will know all accesses to the variable, including whether its address is taken, and can thus optimize the code accessing the variable more efficiently.

If a variable is not marked as `static`, the compiler has to assume that someone outside the current file accesses it, forcing it to be more conservative in optimizing code working with the variable.

4.8 Do Not Use Inline Assembly Language

Using inline assembly is a very efficient way of hampering the optimizer. Since there is a block of code that the compiler knows nothing about, it cannot optimize across that block. In many cases, variables will be forced to memory and most optimizations turned off. Instruction scheduling (especially important on DSPs) has a hard time coping with inline assembly, since the hand-written assembly should not be rescheduled—the programmer has made a very strong statement about the code he/she wants to be generated.

The output of a function containing inline assembly should be inspected after each compilation run to make sure that the assembly code still works as intended. In addition, the portability of inline assembly is very poor, both across machines (obviously) and across different compilers for the same target.

If you need to use assembler, the best solution is to split it out into assembly source files, or at least into functions containing only inline assembly. Do not mix C code and assembly code in the same function!

4.9 Do Not Write Clever Code

Some C programmers believe that writing fewer source code characters and making clever use of C constructions will make the code smaller or faster. The result is code which is harder to read, and which is also harder to compile. Writing things in a straightforward way helps both humans and compilers understand your code, giving you better results. For examples, conditional expressions gain from being clearly expressed as conditions.

For example, consider the two ways to set the lowest bit of variable `b` if the lower 21 bits of another (32-bit) variable are non-zero as illustrated below. The clever code uses the `!` operator in C, which returns zero if the argument is non-zero ("true" in C is any value except zero), and one if the argument is zero.

The straightforward solution is easy to compile into a conditional followed by a set bit instruction, since the bit-setting operation is obvious and the masking is likely to be more efficient than the shift. Ideally, the two solutions should generate the same code. The clever code, however, may result in more code since it performs two `!` operations, each of which may be compiled into a conditional.

"Clever" solution	Straightforward solution

`unsigned long int a;`
`unsigned char  b;`
 	
`/* Move bits 0..20 to positions 11..31`
` * If non-zero, first ! gives 0  */`
`b |= !!(a << 11);`
	`unsigned long int a;`
	`unsigned char  b;`
	
	`/* Straight-forward if statement */`
	`if( (a & 0x1FFFFF) != 0)`
	`  b |= 0x01;`

Another example is the use of conditional values in calculations. The "clever" code will result in larger machine code, since the generated code will contain the same test as the straightforward code, and adds a

temporary variable to hold the one or zero to add to `str`. The straightforward code can use a simple increment operation rather than a full addition, and does not require the generation of any intermediate results.

"Clever" solution	Straightforward solution

`int bar(char *str)`
`{`
`  /* Calculating with result of */`
`  /* comparison. */`
`  return foo(str+(*str=='+'));`
`}`
	`int bar(char *str)`
	`{`
	`  if(*str=='+')`
	`  str++;`
	`  return foo(str);`
	`}`

Since clever code almost never compiles better than straightforward code, why write clever code? From a maintenance standpoint, writing simpler and more understandable code is definitely the method of choice [4].

4.10 Write Compiler-Friendly Loops

There are three loop constructs in C, the `while`, `do-while`, and `for`-loops. The `for` loop is the standard for writing loops, with `do-while` used for special cases (the loop body must be executed before the loop test is performed).

For a straightforward loop, the compiler is quite capable of figuring out that a loop body is always executed at least once, skipping the initial test of the for-loop. There is no need to explicitly rewrite the loops in the less readable `do-while`. Another common optimization done by progammers is to change from counting up ("`for(i=0;i<100;++i)`") to counting down to zero ("`for(i=100;i>0;--i)`"), since this is more efficient on some machines. This can also be done automatically by a compiler.

To enable the compiler to perform such optimizations, you should write your loops so that they are easy to understand by the compiler: change the value of the index variable only in the loop header. Do not use the loop index variable after the loop unless this is the purpose of the loop, since this will make it harder to turn the counting.

Sometimes the compiler will get confused, and then you might have no choice but to rewrite a loop—if it is critical. But in general, write easy-to-understand loops, and leave the compiler to optimize the code for a certain target.

4.11 Map Peripherals to Variables

Accessing special hardware registers and peripheral units is a fundamental feature of embedded software. How you access these registers may have a big impact on the code quality of your application; the best approach is usually to use the special features available in your compiler for hardware access, since this will generate better code. This breaks portability across compilers, but only for the small pieces of the code that interact with hardware which are project- and hardware-specific anyway.

The best approach is to place a variable on top of the I/O register. This will allow the variable to be type-checked by the compiler, including `const`, `volatile`, and keywords for memory attributes (`__near`, etc.). The common idiom to cast a constant address to a pointer "`*((char *)0x4000)`" is bad, since it makes the code harder to read and may generate worse target code. Also, using a variable allows you to compile the code on your PC and try it there, simulating the peripheral using the variable—with an explicit pointer deference, you can be almost assured to get a segmentation fault.

Placing a variable at a particular address is done using compiler-specific syntax or by declaring an extern variable, never providing a definition, and then setting the value of the symbol in the linker file. The last solution is portable across compilers and will enable optimization of the source, but will require work on the link file for each target.

Compiler-specific features	Linker use

`#pragma location=0x4000`
`volatile char Port;`
	`/* C-Source: */`
	`  extern volatile char Port;`
	`/* Linker file: */`
	`  DEFINE Port=0x4000`

For standard derivatives, there are usually header files available describing the available I/O-ports. Use them, as they save you some work. They are also written in the most appropriate way for your compiler.

4.12 Use Switch for Jump Tables

If you want a jump table, see if you can use a `switch` statement to achieve the same effect. It is quite likely that the compiler will generate better and smaller code for the switch rather than a series of indirect function calls through a table. Also, using the `switch` makes the program flow explicit, helping the compiler optimize the surrounding code better. It is very likely that the compiler will generate a jump table, at least for a small dense switch (where all or most values are used).

Using a switch is also more reliable across machines; the layout that may be optimal on one CPU may not be optimal on another, but the compiler for each will know how to make the best possible jump table for both. The `switch` statement was put into the C language to facilitate multiway jumps: use it!

4.13 Access Structures in Order

On machines with post-increment modes for memory access (when a memory access is performed, an address register may be incremented simultaneously), the compiler will be helped if you try to order your memory accesses consecutively. Stepping through an array in order is an obvious case, but since structures are usually placed in memory (as discussed previously), they should also be accessed in the correct order. Jumping around inside a structure will force more address calculations to be generated.

The compiler might be able to reorganize some accesses, but if there are side effects involved, the compiler cannot reorder the accesses.

Original Code	Rewritten Code

`char x, y;`
`volatile char port;`
 	
`s.D = port;`
`s.B = s.D + 4;`
`s.A = bar(x,y);`
`s.C = x;`
	`char x, y;`
	`volatile char port;`
	
	`s.A = bar(x,y);`
	`s.B = port;`
	`s.C = x;`
	`s.D = s.B – 4;`

Note that this conflicts with some other advice in this paper about grouping function calls, so check and see which change gives the best effects. Accessing in order is probably less important, but it all depends on your particular program, compiler, and platform.

4.14 Split Structures

Structures are hard to allocate to registers due to their copy semantics and large size (compared to simple values like integers and pointers). If you are using a structure which is just an aggregate of smaller values for convenience, consider splitting it up into several variables. This will make the pieces much easier to allocate to registers, and might improve your code. Note that it is especially difficult to allocate structures or structure members to registers if the address of the struct or one of its members has been taken (just like for ordinary variables as discussed above).

This is to be considered a desperate measure, since it is considered good programming style to group related values into structures.

4.15 Investigate Bit Fields Before Using Them

Bitfields is one of the more obscure features of the C language. They offer a very readable way to address small groups of bits as integers, but the bit layout is implementation defined, which makes it a problem for portable code. Since bitfields are seldom used, the code generated for bitfields will be of very varying quality. Some compilers will generate incredibly poor code since they do not consider them worth optimizing, while others will optimize the operations so that there is no difference to manual masking and shifting.

The advice is to test a few bitfield variables and check that the bit layout is as expected, and that the operations are efficiently implemented. If several compilers are being used, check that they have the same bit layout. In general, using explicit masks and shifts will generate more reliable code across more targets and compilers.

4.16 Try Another Compiler

Different compilers for the same chip are different. Some are better at generating fast code, other at generating small code, and some may be no good at all.

To evaluate a compiler, the best way is to use a demo version to compile small portions of your own "typical" code. Some chip vendors also provide benchmark tests of various compilers for their chips, usually targeted towards the intended application area for their chips. The compiler vendor's own benchmarks should be taken with some skepticism; it is (almost) always possible to find a program where a certain compiler performs better than the competition.

5 Summary

This paper has tried to give an idea of how a modern C compiler works. Based on this, we have also given practical tips for how you can write code that is easy to compile and that will allow your executable code to be made smaller.

A compiler is a very complex system with highly non-linear behavior, where a seemingly small change in the source code can have big effects on the assembly code generated.

The basis for the compilation process is that the compiler should be able to understand what your code is supposed to do, in order to perform the operations in the best possible way for a given target. As a general rule, code that is easy to understand for a fellow human programmer—and thus easy to maintain and port—is also easier to compile efficiently.

References

[1] *ISO/IEC 9899:1999 Programming languages – C* or *American National Standard for Information Systems - Programming Language - C*, ANSI X3.159-1989.

[2] Jakob Engblom: *Why SpecINT95 Should Not Be Used To Benchmark Embedded Systems Tools*, in Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES) 1999, ACM Press, May 1999.

[3] Johan Runeson, Sven-Olof Nyström, and Jan Sjödin: *Optimizing Code Size through Procedural Abstraction*, Poster presented at the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES) 2000, Vancouver, Canada, June 2000.

[4] Steve McGuire: *Writing Solid Code*, Microsoft Press, Redmond, Washington, 1993.