OPTIMIZATION OF C CODE IN A REAL-TIME ENVIRONMENT

Mark B. Kraeling
Cummins Engine Company
1460 National Road, MS C7004
Columbus, Indiana 47201
email: kraeling@cel.cummins.com

ABSTRACT

The process of developing software varies from company to company, and often project to project. Often lost in the argument of which process to use is coding for optimization of throughput and resources.

This paper will explore different ways of optimizing C source code. Topics presented include, but are not limited to, the importance of selecting a compiler and understanding its options at the beginning of a project, analysis of fixed-point versus floating point operations, and ways to conserve stack and memory resources. Comparisons between compilers and processors will not be addressed, but data and examples will be given to show improvements on multiple platforms.

PROGRAMMING IN C

At the start of a project, it is important to evaluate which programming language to use. Below are a few fundamental ideas about using the C programming language:

- C is a general-purpose programming language.
- C is a relatively "low-level" language; programmers deal with addresses, storage sizes, and logical operators much in the same way as real computers do.
- C provides fundamental control-flow constructs such as if-else, switch, for, and do-while, allowing well-structured programs.
- C code can be developed to run very fast, even though in doing so it may not be very portable across different platforms and processors.
- C does not lend itself well to data abstraction or object-oriented programming.
- C trusts the programmer, and does not prevent the programmer from accessing any data locations.

SELECTING A COMPILER/OPTIONS

At the beginning of a project, the platform is normally selected based on current and anticipated processing needs. In the course of a project, the processing time and throughput may creep upward, eventually causing problems to occur. It then becomes vital to optimize the system for fast execution. Similarly, when RAM or other memory begins to increase above forecasted use, it becomes necessary to optimize for code size and memory use.

Early C compilers were not very good at looking for improvements which often caused the generated machine code to be far from optimum. Today, high-quality compilers normally include an optimizer which examines the generated assembly code and looks for ways to improve it. The compilers will generate machine instructions based on the C code written, and then make multiple passes through the code to look for improvements. The compiler will update the machine code with improvements and continue to analyze the code until no further enhancements can be made. Ideally, these compilers will generate the same assembly code independent of coding style used within the source code. This allows programmers to code with clarity and readability in mind.

ISBN# 0-7803-3274-1

One compiler option is eliminating redundant code. This involves looking for operations that produce no useful result when performed. When the compiler finds these statements while optimizing, it will remove them from the compiled machine code.

```
a = a + 0;   <-- no code generated
b + c;       <-- no code generated
d &= d;      <-- no code generated
```

Another compiler option is replacing operations with equivalent but faster operations that do not effect the final result. Costly multiply and divide operators are replaced with left and right shift operators wherever possible. The modulus (%) operator is replaced with the bit-and operator wherever possible. If your compiler does not make these optimizations for you, then it may become necessary to put the equivalent operations directly in the source code.

```
    x = y * 16    -- replaced with -->  x = y << 4
(unsigned)y / 64  -- replaced with -->  y >> 6
(unsigned)z % 8   -- replaced with -->  z & 7
```

Many of the newer C compilers are also very good at making effective use of the processor's registers that are available. Even on small 8-bit processors, arguments that are supposed to be passed to and from functions are placed in fast registers instead. On larger 32- and 64-bit platforms, more registers are available to take advantage of this optimization. This helps to keep pushing and pulling of data off the stack to a minimum, again helping to increase speed. The example below shows a C segment compiled on a 32-bit platform. Notice that the function assembler output below places the result in register d0 as opposed to pushing the result onto the stack.

Code:

```
static char j, k;
char Example( void )
{
    return ( j + k );
}
```

Assembler Output:

```
move.b  _j, d0
add.b   _k, d0
rts
```

Compiler optimizations can also remove code segments that it knows cannot be executed based on the code conditions. This unreachable code occurs through preprocessor conditional compiles or other optimizations that the compiler has made. The compiler will normally generate a warning message if the unreachable code is a result of preprocessor conditional compiles or programmer's error. Two examples of unreachable code are shown below.

```
    goto L1:
    j = 20;   --> no code generated
    k = 40;   --> no code generated
L1:
    f = 30;


if ( a != a ) b = 10;  --> no code generated
```

When optimizing for speed, the compiler can also "unroll" loops. This normally involves looking for less than a fixed number of iterations of a loop. If a small number is found, instead of having a compare and a conditional branch being performed, it places the particular operations repetitively. This helps a developer write well-written code without worrying about manually unrolling the smaller loops in the code. For a small number of iterations, it may also save on the code size.

There are some things to look out for when having the compiler do optimizations for you. The code executed on the processor will change when the compiler optimizer options change. This makes it important to have all the programmers on a project use the same set of options. Perform unit and systems tests using options intended for the final product. Comparing the source code and generated assembly output can reveal errors in the code or errors when the compiler made improvements. Looking at the assembly output can also help you to become a better programmer in general.

FIXED AND FLOATING POINT OPERATIONS

Most microprocessors and microcontrollers used for embedded or small systems today do not have hardware-assisted floating-point math support. Having a math coprocessor is an added expense that some platforms do not need. The C compilers of today implement floating-point used in C source code with floating-point emulation libraries. These libraries are written by the compiler companies to perform fixed-point conversions and operations to handle floating-point math.

A common way to execute code faster when hardware-assisted floating-point support is not available is to use fixed-point math instead of floating-point emulation math. This allows the compiler to use machine instructions for the math operations as opposed to the floating-point emulation libraries. This will cut down the amount of processing time needed to do mathematical operations. An even better alternative is to use fixed-point representation of base two numbers so the optimizer on the compiler can determine that multiplying by eight is the same as left bit shifting by three. Floating-point emulation libraries also take up additional RAM and fixed memory, so using fixed-point will also decrease code size.

To determine the differences between fixed-point and floating-point emulation multiplies, an emulator with time measurement capability was used. On an 8-bit platform, an 8-bit fixed multiply took 12 microseconds. On the same 8-bit platform, a floating-point multiply took 250 microseconds. On a 32-bit platform, a 32-bit fixed multiply took 3 microseconds. On the same 32-bit platform, a floating-point multiply took 40 microseconds. This shows there are definite advantages in avoiding floating-point emulation in time-critical applications.

As mentioned in the previous section on compiler optimizations, wherever base-2 fixed-point constants are used, left and right bit shift operations can be implemented. This is one of the added benefits of using fixed-point math. Below is an example where bit shifting is used in place of a machine multiply.

Code:

```
static unsigned int i, j, k;

static void example( void )
{
   i = j * k / 32;
}
```

Assembler Output:

```
move.l  _.S13_j, d0    ; places j in register
mulu.l  _.s14_k, d0    ; machine multiply
lsr.l   #5, d0         ; performs right shift
move.l  d0, _.S12_i
rts
```

Floating-point emulation causes the generated code to make a function call to the floating-point libraries. Many operations are performed to set up the registers before the floating-point library function is called. The setting up of these registers is compiler specific. Shown below is an example of floating-point multiply being performed and the assembly code it generates.

Code:

```
static float f, g, h;

void Example( void )
{
   f = g * h;
}
```

Assembler Output (30 clk + fmul$ call ~450 clk):

```
ldy  h     ; compiler-specific loading registers
ldd  h+2
ldx  #g
jsr  fmul$  ; floating-point library call
sty  f      ; compiler-specific saving result
std  f+2
rts
```

Using fixed-point operations causes the compiler to execute the built-in machine instructions instead of making special library calls. The number of clock cycles necessary to perform the multiply is much less than the floating-point example above. Shown below is an example of the same multiply code as above except the operation is performed using non floating-point numbers. Notice the time difference

in clock cycles compared to the previous floating-point example.

Code:

```
static unsigned int j, k, n;

void Example( void )
{
   j = k * n;
}
```

Assembler Output (27 clk):

```
ldab  n
ldaa  k
mul
stab  j
rts
```

Fixed-point math involves representing floating-point numbers with integers. For instance, if a 16-bit integer is used to represent engine speed, and scaling for engine speed was 1/4 RPM, a number of 22 in the integer engine speed would represent 5.5 RPM. Wherever engine speed is used in the code, its scaling of 1/4 would have to be considered in calculations and comparisons. If you were to compare it to an engine speed limit that had a scaling of 1/8 RPM, then you would have to adjust the scaling of the engine speed to 1/8 by multiplying the engine speed integer by 2. The example below shows a simple speed multiplied by time operation. The calculation becomes complex because of the scaling conversion to put the distance in the proper scaling of 1/128 miles. Even though the calculation appears to be very complex, the compiler optimizer evaluates the constant expression and converts it to a single number.

Code:

```
#define SPD_SCALE   4.0  /* Speed 1/4 MPH */
#define TIME_SCALE  2.0  /* Time 1/2 second */
#define DIST_SCALE 128.0 /* Dist 1/128 mile */
#define SEC_IN_HR 3600.0 /* 3600 sec/hour */

extern unsigned int dist, spd, time;
void Example( void )
{
  dist = spd * time /
  (unsigned int)(SPD_SCALE * TIME_SCALE *
                 SEC_IN_HR / DIST_SCALE);
}
```

Assembler Output:

```
move.w  _spd, d0
mulu.w  _time, d0
divul.l #225, d0      ; preprocessor evaluates
move.w  d0, dist
rts
```

There are some things to watch out for if you decide to use fixed-point instead of floating-point. The equations and code doing comparisons between unlike scaled numbers will become complicated. The scaling of the fixed-point numbers has to be considered wherever they are used. It is best to come up with a system of defining the scaling of widely-used system variables and stick with it. For instance, if the measurement being done to determine engine speed is only accurate to 1/4 RPM, define ENG_SPD_SCALE as 4.0 through the use of a #define and make it accessible to any source file that needs to manipulate anything around engine speed. Use this scale factor for all engine speed related parameters to help keep the math complexity to a minimum.

Another item to watch out for is casting of fixed-point numbers. If two 16-bit integers are multiplied and placed in a 32-bit result, it may be necessary to cast the 16-bit numbers to a 32-bit number. This will let the compiler know to keep the 32-bit result instead of chopping off the first 16 bits. The ANSI C standard defines that the result is based on the numbers being multiplied, added, or subtracted, not the size of the result.

The last item to watch out for is making sure that you are truly using fixed-point numbers. Since the pre-compiler will evaluate constant expressions, constants can be floating-point numbers. Remember, though, to cast the result of these constant expressions to fixed-point numbers. Failure to do so will cause the compiler to call a floating-point library function. Many compilers allow the removal of the floating-point libraries from being pulled in by the linker. Removing the floating-point libraries will generate linker errors when the libraries are not found, so you can quickly see which module is using floating-point math and correct it.

GENERAL USER OPTIMIZATIONS

This section discusses techniques a programmer can use to improve coding efficiency. These items directly effect the way the compiler handles your code, and helps it to make further optimizations. The optimizations are listed as single entities, so that different ones can be used or not used. Some of the optimizations listed are specific to the size of platform, with a "small" platform being 8-bit or smaller, and a "large" platform being 32-bit or larger. A 16-bit platform could fit into either category, so it is important to try the recommendations out to see what works best on your platform.

One user optimization is to set a direction for development, as far as optimizing for speed or for code size, and to what degree. This decision will depend on the current capabilities of your processor and platform, and the ease to expand the processor or memory. After a direction is chosen, then all programmers on the project can program with the same end result in mind.

On smaller size microprocessors, it is important to choose the right data definition for the job if optimizing for speed. Try to use the base unit or smaller size of the microprocessor wherever possible, so the compiler can take advantage of using fast registers and built-in machine opcodes. The data listed below gives a relative sample of the length of time necessary to do multiplication and division operations on an 8-bit platform. This will be, of course, compiler dependent since the compiler will have to pull in special libraries to do the operations. This optimization can only be used if the accuracy and range of the data used is acceptable.

8-bit multiply - 15 microseconds
16-bit multiply - 160 microseconds
32-bit multiply - 297 microseconds

8-bit divide - 85 microseconds
16-bit divide - 277 microseconds
32-bit divide - 6685 microseconds

If optimizing for speed, look for the opportunity to inline functions if the compiler doesn't already optimize for it. Inlining functions places the code block of the function directly in place of the called function. This eliminates the process of pushing and pulling data off the stack. Local functions can then be added to a file to help organize the code without worrying about the extra time it takes to call a function. The negative impact of performing this optimization is when the inlined function is called more than once, resulting in increased code size.

If optimizing for speed, avoid using pointers or taking the address of variables. This limits the compiler in using fast registers for the manipulation of data. Using pointers and de-referencing also uses extra steps in the compiled code to determine the location of the data before accessing it. The example below shows an example of data accessed through the use of a pointer, and data being accessed without a pointer. This will increase code size, and may also drive duplication of code. If optimizing for code size, it becomes more important to use pointers when performing operations on like sets of data. This will keep duplication of code at a minimum.

Code using pointer:

```
extern char x, y;
extern char *ptr[20];

void Example( void )
{
    x = *ptr[y];
}
```

Assembler Output (34 clk):

```
ldab  'y'
clra
asld
addd  #ptr
xgdx
ldx   0,x
ldab  0,x
stab  'x'
rts
```

Code without pointer:

```
extern char x, data;

void Example( void )
{
    x = data;
}
```

Assembler Output (13 clk):

```
ldab  data
stab  'x'
rts
```

On smaller size microprocessors, when optimizing for speed, avoid the use of function arguments wherever possible. Use of function arguments causes parameters to be pushed and pulled off the stack wherever they are used. Try

re-structuring the code so that static variables are shared between the two functions if local to a file, or use global variables if not within a file. Creation of global variables may be considered a poor structured design practice, but this may be an appropriate trade off.

On larger size microprocessors, when optimizing for speed, it is not necessary to avoid function arguments all together. Depending on the compiler used, it may allocate registers specifically to function argument passing. Data to be passed will be placed in these registers, for quicker access by the called function. This eliminates the need to use the stack for the arguments. The compiler will only be able to use these registers if the data being passed is equal to or smaller than the register size of the microprocessor. If optimizing for speed, remember there are also only a limited number of these registers available, so it still may become necessary to use static file variables or global variables for data sharing between functions as discussed in the previous paragraph.

On smaller size microprocessors, where the number of registers is limited, declaring local variables as "static" may improve the execution speed of the code as well as decrease the size of the code. When the number of registers is limited, declaring variables as "static" gives them a fixed location in memory. This helps eliminate the variable being used in the stack space, and speeds up access to the variable. If it is non-static, then the code may have to push and pull items from the stack to access it. The example below shows how the assembly code is generated with a local variable as static and the other as non-static. This will have a definite negative impact to the amount of RAM used, since you are now using a fixed memory location.

Code with local var: Assembler Output (23 clk):

```
unsigned char k, n;          des
                             ldab n
void Example( void )         addb k
{                            tsx
    unsigned char j;         stab 0,x
    j = k + n;               ins
}                            rts
```

Code with static var: Assembler Output (14 clk):

```
unsigned char k, n;          ldab n
                             addb k
void Example( void )         stab j
{                            rts
    static unsigned char j;
    j = k + n;
}
```

On larger size microprocessors, it may not make sense to declare items as static. Larger size microprocessors have a greater availability of fast registers. The compiler will allocate a number of these for local variables. These larger microprocessors can then perform the required operations without going out to the RAM space and placing this data on the stack. The number of local variables that can be declared using these registers is compiler and microprocessor specific. Declaring a number of local variables and then checking the assembly output will show the number available. Based on this number, you may wish to have the most often used local variables declared as non-static to use the registers, and the others to be declared as static. This will have a negative impact to the RAM size as discussed in the previous paragraph, but may improve speed and overall ROM code size.

If optimizing for code size, it is important to develop a standard set of base library functions that can be used by all the programmers. This would possibly include a standard filter algorithm, a standard searching algorithm, etc. Write one algorithm that fits all the needs of the programmers. If a programmer needs a filtering routine for 32-bit numbers, and another needs one for 16-bit numbers, write a single routine that processes both 16- and 32-bit numbers. This would have a negative impact to code speed if your microprocessor cannot perform 32-bit operations as quickly as 16-bit operations.

Another user optimization is loop unrolling. This is unnecessary if your compiler automatically does loop unrolling when optimizing for speed. Loop unrolling is the process of taking looped code and repeating the number of iterations of that code without looping. This would increase code size but helps to increase the speed. If you

have to manually unroll the loops in your code it also makes the code less readable and harder to maintain. The best way to unroll loops is have your compiler take care of it for you, so the code can be left with the "for" and "do-while" loops left in. There is usually an iteration count the optimizer uses to determine if the loop should be unrolled, such as three or less iterations. This number is sometimes a user specified option.

The last user optimization helps the compiler optimizer directly. Consider making use of the keyword type-qualifiers "volatile" and "const" in front of declared variables wherever possible. This helps the compiler to determine the type of data when optimizing. Using the keyword "volatile" will tell the compiler to re-access the data type wherever it is used in the code. Leaving this keyword out may drive the compiler to access the data once at the start of the code block and place the data in a register. The compiled code will then reference this register wherever it is used in the code instead of the actual data location. Using the keyword "const" will help the compiler to make memory optimizations. Many compilers will keep declared constants in read only memory, rather than RAM. Pointers declared as constant will also help the compiler to not continually re-evaluate where a pointer is pointing before it is used. The keywords "volatile" and "const" can be interpreted differently depending on the compiler used, so check how they are used on your platform.

CONCLUSIONS

The user optimizations that were discussed in the previous section should be considered as separate optimizations to make based on the platform being used. They should be utilized in conjunction with having a good compiler optimizer. It is also important to determine if fixed-point math can be used as opposed to floating-point emulation math. Evaluating all three of these issues will get you on your way writing C code with optimizing principles in mind.

Since C is considered a relatively "low-level" language, it is important to look at the assembler output from the C compiler and look for bottlenecks. Sometimes the bottlenecks and delays may not be require rewriting the code, but may provide valuable information for the next code iteration. Sometimes restructuring of the code will also aid the compiler's optimization algorithm, producing very positive results.

The suggestions and ideas presented in this paper should be tried out on your particular platform and compiler. It is also important to temper some of the recommendations with the way your project's code is structured. By far the best and easiest way to optimize your code is having your compiler do it for you. It is also important to determine if fixed-point can be used as opposed to floating-point emulation. Last of all, do your best while coding to keep code size or speed in mind and make efficient use of the stack, registers, and CPU bandwidth.