Custom Computer Services, Inc.

OPTIMIZATION AND OVERHEAD

Using the CCS C Compiler for PICmicro® MCU Targets

C vs. Assembly

Since writing in C takes a fraction of the time it takes to write the same code in Assembly. The intention of this paper is to analyze the overhead of memory used, while employing a C compiler as opposed to Assembly for PIC® MCU projects. The example code and the numbers used in this paper are for an 8-bit PIC® MCU with 14-bit targets. Keep in mind that while 12-bit targets are going to be a little less optimized, 16-bit targets will be even more optimized, given the same scenarios. ROM location counts are given for 14-bit words. For estimation purposes, one can assume that there are 5 ROM locations per C source line. This means that about 200 lines are used for a 1K part, 800 lines for a 4K part, or 1600 lines for an 8K part. The analysis presented herein is presented for Version 3.073 of the CCS C compiler.

Overhead

All programs require nine ROM locations and two RAM locations for compiler overhead. Additional overhead may be incurred, depending on the libraries used. For example, while using a PIC16F84 chip and requesting a *putc()*, 36 locations will be used for a library routine to perform the software *putc()*. On the other hand, using a PIC16C74 chip with a built-in USART allows the *putc()* to be implemented as inline code. The overhead used, to initialize the USART, is now just eight locations . On the PIC16F84 processor, each call to *putc()* takes three locations, and on the PIC16C74, because the code is inline, it takes four locations.

Only functions that are actually used will be included in the final output code. For example, when including the RS232 library, several functions are incorporated such as *putc(), getc(), kbhit(), printf()*, and so on. If only *putc()* is used, the other functions are not included in the ROM. Furthermore, some functions are customized depending on their specific use. For instance, using *printf()*s with only %U for unsigned numbers will prevent code for handling signed numbers from being generated. The same rule applies to user functions, statements, and constants intended for ROM. Functions never called and statements that could never be executed, due to program logic, are simply not put into the final ROM image.

Assembly lines generated for each C line

Counting C lines can be subjective, since lines such as *comments* and *#defines* do not generate code. Even a C statement can have a wide range of outputs. For example:

```
printf("This is a printf with a long constant string generating a ton of code");
```

will require 85 ROM locations; and

```
i++;
```

will take only one location.

Data types can affect the efficiency as well. For example:

```
i=j;
```

If *i* and *j* are unsigned eight-bit integers, this will require two ROM locations. However, if *i* and *j* are 32-bit integers or floats, then eight locations are required.

Floating point can be very space expensive and should only be used when absolutely necessary. For example, when doing the first floating point "add", 321 locations are required for the library routine. Additionally, each "add" takes 26 locations just to move the data to the appropriate spot and to make the call.

For example, an 8K, real life application that includes simple user input, LCD output, RS485 communication, E^2 memory management, some floating point math, and a couple of custom hardware interfaces have the following characteristics:

C Source Lines:	1726	(all comments removed)
ROM Locations Used:	7988	
Average ROM used per line:	4.6	

Examining a number of real application programs of differing sizes the same way, we find the following ratios:

Most Efficient:	3.4	ROM locations per line
Overall Average:	5.2	ROM locations per line
Least Efficient:	7.5	ROM locations per line

The above analysis counted all ROM locations used in the final program and all lines, including blank lines, except comments.

Comparison to hand generated Assembly

The compiler is excellent with basic operations involving unsigned bytes and bits. This is the type of work generally done on target microprocessors.

For example, consider the C source line:

if(alarm_triggered && !override_active && (seconds_timer>60))

The generated code from the compiler will appear as the following:

```
0013: BTFSS 26,0
0014: GOTO 01B
0015: BTFSC 2B,4
0016: GOTO 01B
0017: MOVLW 3D
0018: SUBWF 2E,W
0019: BTFSC 03,0
```

Generating this Assembly by hand is not going to improve the ROM usage, however the code will be less readable and less maintainable. On the other hand, using the C compiler will allow for easy to read and easy to understand source code. The compiler is most effective when data types are well chosen, for the particular application. In the previous example, the variables *alarm_triggered* and *override_active* are one-bit variables. If they were declared as eight-bit integers instead, then two more ROM locations would be required for the *#if* statement. The variable *seconds_timer* is an unsigned eight-bit integer. If it had instead, been declared as a signed eight-bit integer, then three more ROM locations would have been used. This demonstrates the importance of carefully choosing the most effective data type.

The compiler handles automatic switching between RAM banks, when accessing RAM. In order to minimize the bank switching, the compiler will attempt to group local variables used by a specific function, within the same bank. Using the above example, assume the second variable accessed is forced to be in a different RAM bank than the first and third variables. Then, the resulting code takes four more ROM locations as follows:

```
004D: BTFSS 14,1
004E: GOTO 059
004F: BSF 03,5
0050: BTFSS 20,2
0051: GOTO 054
0052: BCF 03,5
0053: GOTO 059
0054: BCF 03,5
0055: MOVLW 3D
0056: SUBWF 21,W
0057: BTFSC 03,0
```

When coding in Assembly, the aforementioned inefficiency becomes obvious and the programmer may attempt to move the variables around to make the code smaller. Although the compiler does some variable grouping, it will not be as good as if it were done by hand. Note that when programming in C, the same effort may be made to rearrange the variable allocation; however, the need may not be as obvious unless the list file is reviewed.

Example code generation

```
.....               for(i=1;i<=10;i++) {
0004: MOVLW 01
0005: MOVWF 51
0006: MOVLW 0B
0007: SUBWF 51,W
0008: BTFSC 03,0
0009: GOTO 018
.....               if(table[i]!=0)
000A: MOVLW 25
000B: ADDWF 51,W
000C: MOVWF 04
000D: MOVF 00,F
000E: BTFSS 03,2
.....                break;
000F: GOTO 018
.....               if(i>a)
0010: MOVF 51,W
0011: SUBWF 4D,W
0012: BTFSC 03,0
0013: GOTO 016
.....                a=i;
0014: MOVF 51,W
0015: MOVWF 4D
.....               }
0016: INCF 51,F
0017: GOTO 006
.....
.....               do {
.....               table[i]=1;
0018: MOVLW 25
0019: ADDWF 51,W
001A: MOVWF 04
001B: MOVLW 01
001C: MOVWF 00
.....               } while (--i!=0);
001D: DECFSZ 51,F
001E: GOTO 018
.....
001F: MOVF 50,W
0020: ANDLW 7F
0021: IORLW 02
0022: MOVWF 4F
.....
.....               i16 = i16 + 9;
0023: MOVLW 09
0024: ADDWF 52,F
0025: BTFSC 03,0
0026: INCF 53,F
.....
.....               i16 = (i16*2)+3;
0027: BCF 03,0
0028: RLF 52,W
0029: MOVWF 54
002A: RLF 53,W
002B: MOVWF 55
002C: MOVLW 03
002D: ADDWF 54,W
002E: MOVWF 52
002F: MOVF 55,W
0030: MOVWF 53
0031: BTFSC 03,0
0032: INCF 53,F
.....
.....                if(i16>a)
0033: MOVF 53,F
0034: BTFSS 03,2
0035: GOTO 039
0036: MOVF 52,W
0037: SUBWF 4D,W
0038: BTFSS 03,0
.....                a=0;
0039: CLRF 4D
```

Conclusion

To recap, the CCS C compiler generates extremely efficient, optimized code. Efficiency is going to be even more greatly improved through additional efforts made by the programmer when choosing data types, and when applicable, locating the data in memory. Even in situations where complex expressions generate more code than is desired, additional effort can usually result in highly optimized code generation. For critical code, where efficiency is not as important as the timing of the code, exact timing can be achieved by using inline Assembly within the C source code. One can always generate code in C as optimized as Assembly by using only simple C constructs and data types. If Assembly code already exists for a project, consider converting the code to C to enhance readability and maintainability.

About CCS

Established in 1996, CCS is a leading worldwide supplier of embedded software, and hardware development tools, that enable companies to develop premium products based on Microchip PIC®MCU and dsPIC® DSC devices. CCS C Compilers are the most advanced, highly developed and most widely used compiler in the industry. These compilers include a generous library of built-in functions, pre-processor commands, and ready-to-run example programs to quickly jump-start any project. CCS IDE C compiler products provide a unique Profiler Tool to track time and usage information for use on functions, code blocks, as well as receiving live data from running programs. Complete proven tool chains include a full line of programmers and debuggers, application specific hardware prototyping boards, and software development kits. CCS is also a leading provider of electronic engineering services for embedded software development, R&D support, hardware design, and custom electronic products that adhere to our client's high-quality standards.