10 C Language Tips for Hardware Engineers

Jacob Beningo - March 5, 2013

On its own, the software development process has numerous hazards and obstacles that require navigation in order to successfully launch a product. The last thing that any engineer wants is challenges resulting from the language or tool that is being used. It often makes sense or is necessary for the hardware designer to write code to test that the hardware is working or in resource constrained cases, develop both hardware and embedded software. The language of choice is still C and despite the advances in tools and structured programming, time and again basic mistakes occur that lead to bugs and maintenance nightmares. In an attempt to avoid these C programming pitfalls, here are 10 C language tips for hardware engineers.

Tip #1 – Don't use "goto" statements
A couple decades or so ago when computer programming was in its' infancy, a programs flow was controlled by "goto" statements. These statements allowed a programmer to break the current line of code and literally go to a different section of code. A simple example of this can be seen in Listing 1.

```
void main(void)
{
    int Count = 0;

    // Do Something

    // Wait for a while
    WAIT:
    if (Count < 5000)
    {
      goto WAIT;
    }

    // Do more stuff
}
```

Listing 1: Use of goto statement

Programming languages eventually began to incorporate the idea of a function, which allows the program to break off to a section of code but rather than requiring another goto statement, when completed the function returns to the next instruction after the function call. An example can be seen in Listing 2. The result was improved program structure and readability and ever since this has been considered the appropriate way to write a program. The very sight or thought of goto statements cause software engineers to cringe and shudder in distaste. One of the main reasons is that a program with goto's sprinkled throughout is very difficult to follow, understand and maintain.

```
void main(void)
{
    // Do Something

    // Wait for while
    Delay(50);

    // Do more stuff
}
```

Listing 2: Using a function to control flow

Tip #2 – Use for(;;) or while(1)
If goto's are out, some hardware engineers may wonder how an infinite loop can be created for the program. After all, this may have been done before by creating a goto statement that returns back to the top of main. The answer is to take advantage of the looping statements that are already built into the C language; for and while. Listing 3 and Listing 4 show examples of using for and while loop in this regard.

```
// Using a for loop to create the infinite program loop
void main(void)
{
   // Initialize the system
   Sys_Init();

   // Run the program forever
   for(;;)
   {
       // Run the cool application
   }
}
```

Listing 3 – Using an infinite for loop

```
// Using a while loop to create the infinite program loop
void main(void)
{
   // Initialize the system
   Sys_Init();

   // Run the program forever
   while(1)
   {
       // Run the cool application
   }
}
```

Listing 4 – Using an infinite while loop

The loop conditionals in the listings are relatively straight forward. The for loop is nothing more than the for conditional with no conditions. The while loop on the other hand will execute as long as the statement is true which is the same as having any non-zero value for the condtion.

Tip #3 – Use the appropriate conditional statement for the job

Program execution time can be highly dependent on the type of conditional structure that is selected for making a decision in addition to the readability of the code. Many hardware engineers are familiar with the use of the simple if statement; however, sometimes the engineer doesn't realize that if the first condition isn't correct, an else or else if statement can be used. This can save the processor time by not having to evaluate another conditional statement. An example of this can be seen in Listing 5. In the before code, if Var is equal to one it will still check to see if the Var is equal to zero; however, in the after code that uses the else, only the first statement is evaluated and then the code moves on, thereby saving clock cycles and making the code clearer.

Before

```
if(Var == 1U)
{
    // Do something neat
}

if(Var == 0U)
{
    // Do something cooler
}
```

After

```
if(Var == 1U)
{
    // Do something neat
}
else
{
    // Do something cooler
}
```

Listing 5 – Using if/else instead of just if

The if/else if/else statements still may not always be appropriate. If there are a number of possible conditions that need to be checked, a switch statement may be more appropriate. This allows the processor to evaluate the statement and then select from a list of answers what it should do next rather than continually evaluating a bunch of conditions. An example can be seen in Listing 6 that corresponds to the same type of example shown in Listing 5.

```
switch(Var)
{
    case 1:

        // Do something neat

    break;

    case 0:

        // Do something cooler

    break;

    default:

        // Should never get here
        // Error or something

    break;
}
```

Listing 6 – Using switch statements

The moral of the story is simply to keep alternative conditional statement options open and select the most appropriate for the job at hand. This will ease in understanding the flow of the program by making the structure straight forward and could squeeze extra clock cycles out of the processor.

Tip #4 – Avoid using assembly language

The natural language for a microprocessor is assembly language instructions. Writing a program in the low level machine language can result in more efficient code for the processor; however, humans don't naturally speak this language and as experience has shown writing assembly language results in misunderstanding. Misunderstanding then leads to improper maintenance or worse and the result is a system overridden with bugs. Because of this, the general tip is to avoid the use of assembly language. The detailed truth of the matter is that most compilers now a day compile very efficient code. Developing in the higher languages like C/C++ results in a more organized structure which is easier to understand and maintain, the result of which is overall better code. A simple example can be seen in Listing 7 which compares the assembly to C code to increment a 32 bit variable.

```
INC [_Sys_Tick+3]          ;Increment Byte LSB
JNZ .GoReturn              ;Jump is no overflow

INC [_Sys_Tick+2]          ;Increment Byte 1
JNZ .GoReturn              ;Jump is no overflow

INC [_Sys_Tick+1]          ;Increment Byte 2
JNZ .GoReturn              ;Jump is no overflow

INC [_Sys_Tick]            ;Increment Byte MSB
```

Assembly

```
Sys_Tick++;
```

C Code

Listing 7 – Incrementing a variable in Assembly vs C

With that said, there are still occasions when it is appropriate to use assembly language, but those times are scarce. The first recommended time may be when developing a boot-loader. In this instance, during start-up it may be necessary to optimize how quickly a decision is made to boot the application or the boot-loader. In this case assembly code for the branch decision may make sense. Another, is when developing a control loop that has tight timing requirements on a DSP. In order to squeeze every clock cycle out of the device it may make sense to code the control loop in assembly. If the task at hand is appropriate for using assembly, make sure that it is well documented so that future developers (or future versions of yourself) can understand what the code is doing.

Tip #5 – Take advantage of modularity

One of the most common experiences the author has when taking on a new project that was started by hardware engineers is the atrocious organization of the code. It isn't uncommon to find that code consists of a single main module with in excess of 25,000 lines. In these applications everything is global, functions are sparse and goto statements rule the organization of the code. Fifteen years ago this was the norm but not anymore! Programming in C/ has given engineers the ability to break up their code into separate functional modules. This eases navigation of the code but also starts to allow an engineer to use object oriented techniques such as encapsulation. Where it makes sense, organize code into logical modules. It'll take a little bit more time up front (a few minutes!) but in the long run it will save many long nights and headaches of debugging!

Tip #6 – Write Lasagna not Spaghetti code

Beningo is an Italian name, and like any good Italian, a love of pasta is given. When comparing pasta to software, spaghetti and lasagna come to mind. Spaghetti is chaotic; noodles intertwine going this way and that way resulting in a complete lack of any type of structure. Writing unstructured code is exactly like spaghetti, with each bite you have no clue what you are going to get!

On the other hand there is lasagna! The noodles are layered, giving the meal structure! Code developed using layers is not only easier to understand, it has the potential to have one layer removed and a new layer added, basically allowing for reuse and ease in maintainability. Figure 1 shows an example of a simple software model that uses the lasagna model

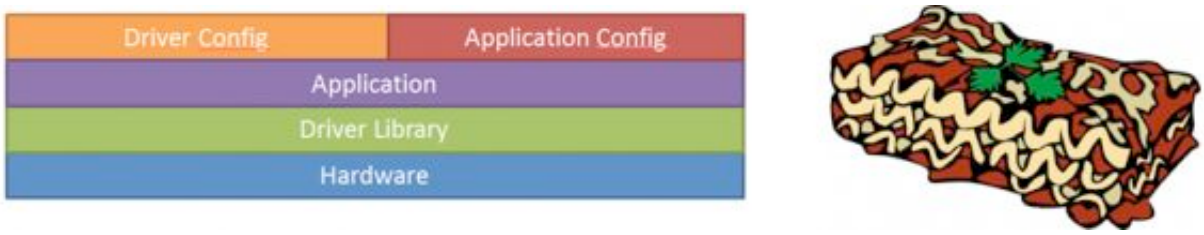

Figure 1 – Lasagna Software Model

Tip #7 – Use Descriptive variable names

One of the barriers to writing great software that is understandable and easy to maintain is the naming convention of variables. In an effort to keep variable names short, it is common for developers to create shortened, cryptic mnemonics that only they can understand once in a blue moon. Modern languages allow for 100's of characters to be included in a variable name. To keep things clear, "call a spade a spade" as the phrase goes rather than something else. This will make the variable name obvious not only to the developer but also future maintenance teams. An example can be seen in Listing 8.

```
int Frq;                int Frequency;
int Btn;                int Button;
int MtrState;           int MotorState;
int Spd;                int Speed;
```

Listing 8 – Variable Naming

Tip #8 – Use #pragma statements sparingly
In the C language there is a special type of statement known as #pragma. These statements often handle non-standard syntax and features. They should be avoided as much as possible because of the fact that they are non-standard and will not port from one processor to the next. Some compilers may require them for tasks such as defining an interrupt service routine. In these instances there may be no way around using a #pragma. If possible, keep the #pragma statements all together in one module or in a couple of modules. This will help to ensure that when the code is ported, there are only a few places to update the code rather than places sprinkled throughout the code base. This will help prevent a nightmare when the ported code is compiled for the first time.

Tip #9 – Errors aren't always as they seem
One of the gotchas to look out for when debugging a C program is compiler errors. Depending on the sophistication of the compiler, when an error is detected, more often than not the error lays somewhere else in the program than where the compiler is indicating. The reason for this has to do with the steps that the compiler takes to generate the program. The types of errors are generally pretty consistent so there are a couple errors an engineer can look for that nine times out of ten is the culprit.

a. Watch out for missing #include files. This can result in the developer looking at a perfectly good line of code but because the necessary header files aren't included the compiler flags it as error, indicating that something is not defined.
b. Watch for missing semicolons. One of the most common errors when writing C code is to forget the semicolon (;) at the end of a statement.
c. Watch for missing brackets (}). Brackets are another common mistake that are either left out on accident or from mistyping turns into a different character.
d. Watch for missing commas (,). In complex definitions it's easy to forget the comma!

In general when a strange compiler error pops up, look around at what might have been compiled immediately before that line. Odds are that is where the mistake is! It may be one line up, half a page or in a completely different file. Don't give up though! With some experience, finding the difficult ones eventually becomes second nature.

Tip #10 Good programmers don't necessarily write fewer lines of code
It is a common misconception that a good programmer can write fewer lines of code to do something than an average programmer. Don't get sucked into this idea! A good programmer has a well thought out and structured code base. Variables are nicely names and encapsulated, few global variables exist in the system. Functions should be short and concise. If the code looks confusing and it would be clearer to write more lines of code then do so! Check out the online awards for writing the most confusing C code! A good programmer writes clean code that is easy to understand and maintain not the fewest lines of code!

```
\
int
_,l;\
char*I,
*0[]={"",
"gjstu","t"
"fdpoe","uij"
"se","gpvsui",\
"gjgui","t"
"jyui","tfwf"
"oui","fjhiui",
"ojoui","ufoui",\
"fmfwfoui","uxfmgu"
"i","b!qbsusjehf!jo!"
"b!qfbs!usff/\xb\xb",""
"uxp!uvsumf!epwf"
"t-\xb","uisff!gsf"
"odi!ifot-!","gpvs!d"
"bmmjoh!cjset-!","gjwf"
"!hpme!sjoht<\xb","tjy!h"
"fftf!b.mbzjoh-!","tfwfo!t"
"xbot!b.txjnnjoh-\xb","fjhiu"
"!nbjet!b.njmljoh-!","ojof!mbe"
"jft!ebodjoh-!","ufo!m"
"pset!b.mfbqjoh-\xb","fm"
"fwfo!qjqfst!qjqjoh-!","ux"
"fmwf!esvnnfst!esvnnjoh-!",""
"Po!uif!","!ebz!pg!Disjtunbt!n"
"z!usvf!mpwf!hbwf!up!nf\xb","boe"
"!"};int putchar(int);int main(void
){while(l<(sizeof 0/sizeof*0-2)/2-1){
I=0[_=!_?sizeof 0/sizeof*0-
3:_<(sizeof(0)/sizeof*0-2)/2?
sizeof 0/sizeof*0-2:_==(sizeof(
0)/sizeof*0-2)/2?++l,0:_<(sizeof(
0)/sizeof(*0))-3?(_-1)==(sizeof(0)/
sizeof*0-2)/2?sizeof 0/sizeof*0-1:_-1
:_<sizeof(0)/sizeof*0-2?l+1:_<sizeof(0)
/sizeof*0-1?l+(sizeof 0/sizeof(*0)-2)/2:(
sizeof(0)/sizeof*0-2)/2];while(*I){putchar(
*I++-1);}}
return 0;}
```

Figure 2 – A short program

Jacob Beningo is a lecturer and consultant on embedded system design. He works with companies to develop quality and robust products and overcome their embedded design challenges. Feel free to contact him at jacob@beningo.com or at his website www.beningo.com.
