Abstraction in C

Posted on **April 3, 2012**

I am going to go out on a limb and state that in my view, the quest from greater Abstraction has been one of the key driving forces in the field of computer science. Software engineers have long been envious of our electrical engineering cousins who have the luxury of using integrated circuits to contain and hide complexity. On many occasions, I have heard calls for the creation of “software ICs” that would be just as easy to use as hardware ICs.

The first step in this direction was the use of high level languages. Rather than messing about with loading registers, adding and storing results you could simply write A=B+C; A tangle of branches, jumps and labels was replaced with far more manageable structured constructs; and the concept of assembly language subroutines became the high level language concept of functions. With algebraic notation and structured programming a function could wrap up a complex algorithm into a nice easy to use package. Mission Accomplished!

Except, it doesn’t really work though. The problem is that a function is too limiting. Just as a real IC often has multiple functions, to be useful, so will software ICs. Yes you can make a function do multiple things with an extra “command” parameter, but that is a rather slow, complex, and ugly approach often called “Swiss Army Knife” syndrome. We need to take a number of functions, plus supporting code and data, and tie them together somehow into a larger unit. That unit was embodied in the concept of a module.

The fact that you are reading this article means that you probably already know the basics of Modular programming in “C”. Modular programming is so essential to “C” programming that even the super silly “Hello World!” program demonstrates it by utilizing the stdio module of the standard “C” library.

```
#include <stdio.h>
int main()
{
    printf("Hello World!\n");
```

```
    return 0;
}
```

Note that the discipline to organize files along these lines is not enforced by the language and it is possible to create some really horrible messes, but it is not difficult to follow either, so most programmers do so. Now within the header file consider these three declarations:

```
/* 1 */ extern int FontCount;
/* 2 */ int CountFonts(FONT_ROOT fontRoot);
/* 3 */ extern int TestCode(int testType);
```

Line 1 is used to declare a variable being exported and line 2, a function prototype which also serves to export the function and finally line 3, declares that the coder does not understand that the “extern” keyword does not apply to function prototypes. Unfortunately, the error in line 3 is so common, that it is allowed, but the spurious “extern” serves no purpose.

More modern languages make modular programming an integral part of the language; the keep-it-simple “C” language provides just enough resources to make it possible. The header file plays the vital role as the Interface to the module. Sometimes, if the abstracted code is of a large scope, we call it a Library or a Framework and the interface gets the fancier title of Application Programming Interface or API.

In the world of computing, many themes often repeat with endless variation. Consider the simple serial port. Most embedded systems have them. The addresses and register details vary, but the basic capabilities are similar. Removing the fiddly distinctions and creating a reusable interface is a perfect example of abstraction in action. In the old days this sort of thing was called a BIOS, but today it goes by the fancier name of Hardware Abstraction Layer or HAL.

In the ideal world, there would be a standardized HAL interface and each vendor would code to that interface. While abstraction is not perfect, it would go a long way toward improving portability. The problem is, such a universal HAL spec does not exist, and even if it did, most vendors do not want it to lead to true portability as this would make it easier for their customers to switch away to other brands.

A most infamous case of this is from the PC programming world. Microsoft created an abstraction called MFC to encapsulate the Windows API and make it easier to write programs. The problem was that Microsoft was in competition with other vendors offering similar systems and did not desire to make it easy to write portable applications. By “coincidence”, the MFC library has so many places where raw Windows programming is required that one pundit stated that “The MFC framework is not so much an encapsulation as a leaky diaper!”

Another example of this sort of problem is the Cortex Microcontroller Software Interface Standard or CMSIS which is supposed to make it easier to move among ARM processors but is plagued with vendor specific deviations and non-standard definitions thus making the task of porting from one ARM core to another about as hard as porting from ARM to an unrelated core. Don’t get me

wrong, I think CMSIS is a great idea, it's just that there needs to be more oversight and compliance testing to make it work.

In the mean time, the best that can be done is to establish HAL standards at the project, team or corporate level and use these, non-vendor based models for creating a sensible, level playing field. With some luck, an open source project might even propagate such a fair model to the point that it becomes a defacto standard. It has happened before. Where would the world of networking be without the historic development of TCP/IP? We can only hope.

Abstraction and Modular programming are powerful tools in the armory of the embedded programmer. While not complete, "C" supports them well enough to create very sophisticated applications. In upcoming articles, I hope to examine some examples of modular abstraction in action!

As always, comments and input are welcome!

Peter Camilleri (aka Squidly Jones)

This entry was posted in **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=552]** .