C32 Impressions (3/23/2012)

Posted on **March 21, 2012**

In my PIC32 programming I have three primary development tools: The MPLAB-X IDE, the MPLAB ICD-3 programmer/debugger, and the C32 V2.02 "C" compiler. In this post we are finally going to take a look at the compiler. Like many compiler projects, C32 is a subset of the open source GCC 4.5.1 compiler and the non-proprietary bits are available in source format. Some major features absent from C32 include: C++ support, 128 bit integers, decimal floating types, 16 bit float types, and fixed point (DSP oriented) types. Still, many useful extensions to "C" are included and the user would do well to download the GCC 4.5.3 documentation (the closest to 4.5.1 that I could find). I especially like the binary constants feature; for example: **i = 0b101010;**

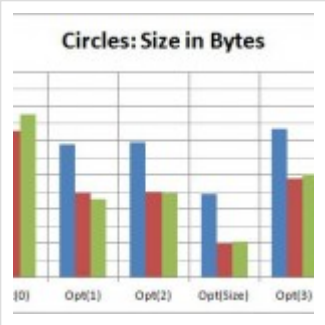
C32 is available in essentially two major versions: A free, 60 day trial and Full. The Free version is a free download from the we site and has the the following restrictions specified in the compiler release notes:

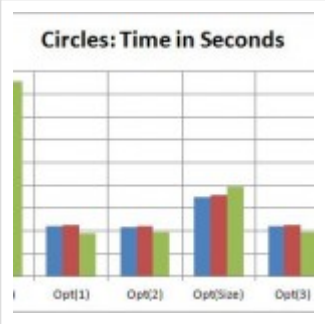
"Microchip provides a free Standard Evaluation edition of the MPLAB C Compiler for PIC32 MCUs. The standard evaluation edition of the compiler provides additional functionality for 60 days. After the evaluation period has expired, the compiler becomes reverts to the lite edition, and optimization features associated with levels -O2, -O3, and -Os are disabled. In addition, MIPS16 code generation is disabled. The compiler continues to accept the -O1 and -O0 optimization levels indefinitely."

The full version is available for purchase from Microchip Direct for a list price of $895. This price is routinely discounted for attendees of the annual Master's Conference, but even so it it is still a rather large chunk of change. To gain a better understanding of compiler performance, I undertook to perform a series of benchmarks using C32 V1.11, V1.12, and V2.02 at the four levels of optimization measuring performance in space (size in bytes) and time (execution in seconds). To test with I created four programs. Here are my findings for each program:

Test #1 Circles

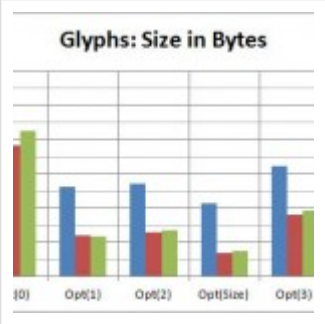
Draw 100,000 random circles on a graphical display (no H/W acceleration)

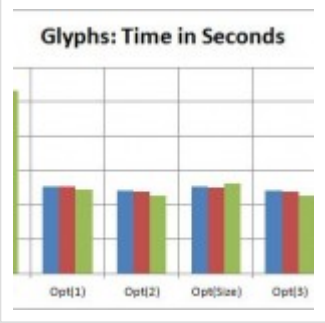




Test #2 Glyphs

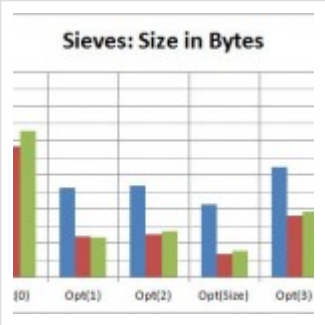
Draw 200,000 (8 by 13) Character Glyphs on a graphical display (no H/W acceleration)





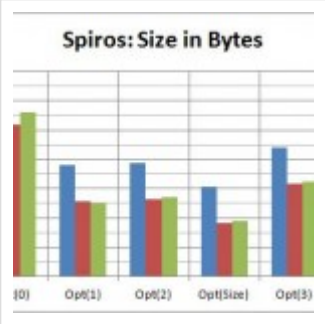
Test #3 Prime Number Sieves

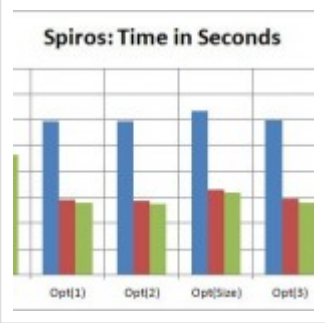
Find all prime numbers less than 20,000 and do this 5000 times.



Test #4 Spirographic Patterns

Draw 200 repetitions of a complex polar plot diagrams reminiscent of the old Spirograph toy.





Benchmarks Summary

For almost all debugging, it is advised to use O(0) or non-optimized code. The code re-arranging done by the optimizer makes debugging very difficult. At least, optimization should be turned off for the source file being debugged. As for production code, there does not seem to be a great deal of difference in optimization setting 1, 2, and 3. Setting “s” produces consistently smaller and slower code. Thus the main advantage of the full compiler versus the free one seems to lie mainly in that it can produce more compact code.

MIP16 Mode

There are options to force the compiler to use MIPS16 mode throughout a project, however given that most projects contain various routines not allowed to 16 bit, this approach will seldom work well. I prefer to use the attribute ability to mark individual functions as 16 bit. This gives me fairly fine grained control over where I favor space over time. For brevity I define:

```
#define mCode16 __attribute__((mips16)) __attribute__((noinline))
#define mCode32 __attribute__((nomips16))
```

Note that the mCode16 macro selects mips16 mode and the noinline option. The noinline is required as a work around to a minor compiler bug, but the avoidance of inline code makes sense if you are trying to save space. A simple example where space savings may be desired is:

```
void mCode16 emQryTouchRaw(emPTOUCHRAW rd)
{
    rd->x = xf;
    rd->y = yf;
}
```

Even in this trivial case, MIPS 16 mode saved 16 bytes of program space. This option is NOT available in the free compiler after the 60 day trial period expires.

Conclusions

For the most part, the C32 compiler is a like a good car engine. It does a good job and you don't have to waste time fussing over it. Like any good tool, it lets you focus on the job at hand rather than constantly tweaking things. I run the FULL version and I am glad to have the flexibility it offers. Nevertheless, a great many projects can be accommodated by the powerful free version of C32, and tool cost should not be an impediment to trying out new ideas.

Command Line Options

The compiler command line includes many many options, most of which are *not* documented. They are listed in this PIC32 GCC Usage file for your convenience. Use at your own risk!

Peter Camilleri (aka Squidly Jones)

PS: While Microchip is proceeding with caution, there are indications that they may yet unleash the C++ capabilities inherited from GCC! And that too would be a good thing!

UPDATE (3/23/2012) On further reflection, I just remembered that I left out a major method of reducing code bloat that should work in all versions of C32. When creating code modules, it is not uncommon for some functions to not be called. Now, these could just be deleted, but that often means maintaining multiple versions of the code which can be messy. Two options, work together to automatically remove un-called code. In the picc32-gcc there is "Isolate each function is a

section” and in the picc32-ld there is “Remove unused sections”. Together these options can result in substantial space savings with no loss of speed or debugging capability.

On another unrelated note, the image to the left is a screen shot of my spiro-graph test. The subtle color changes are on purpose. I added them so I could see the colors shift as I drew 200 copies of the pattern for my testing. Many many years ago, I was tasked with writing a CALCOMP pen plotter driver and scientific graphing package in FORTRAN. For my test data, I used spiro-graphic patterns back then too and I have been intrigued by them ever since.

This entry was posted in **Dev Tools**, **Embedded Development** by **Peter Camilleri**. Bookmark the **permalink [http://teuthida-technologies.com/?p=555]** .