nesdoug

2. how cc65 works

All NES assembers are command line programs. What does that mean? It has no graphic user interface. You don't type code into it. You have to write your code in a separate program (Notepad++) and save it. And, then open a command prompt, and run the assembler by typing into the command prompt.

Cc65 is much more complicated, it requires several directives to be typed in, with multiple steps, so we're not going to use the command prompt. But DON'T WORRY! I'm going to simplify it for you… we're going to write a batch file (compile.bat) to automate the process. Once it's written, all you should have to do is double click on compile.bat, and if all goes well, it will make your NES file. (Write the .bat file with Notepad++.)

How cc65 compiles..first, you will write your source code in C (with Notepad++). cc65 will compile it into 6502 assembly code. Then ca65 will assemble it into an object file. Then ld65 will link your object files (using a configuration file .cfg) into a completed .nes file. All these steps will be written in the batch file, so in the end, you will be double clicking on a .bat file to do all these steps for you.

More about cc65…

The 6502 processor that runs the NES is an 8 bit system. It doesn't have an easy way to access variables larger than 8 bit, so it is preferred to use 'unsigned char' for most variables. Addresses are 16 bit, but nearly everything else is processed 8 bit. And, the only math it knows is adding, substraction, and bit shift multiplication/division (ie. x 2 or / 2).

What this means, is you may want to write your C code very differently, due to the system limitations.

1. Most variables should be defined as unsigned char (8 bit, assumed to have a value 0-255).
2. I try to either pass no values to functions…or use a fastcall, which stores the passed variables in the A,X,Y registers.
3. Arrays, ideally, should have a max of 256 bytes.
4. Many of the things you're used to (printf) won't work.
5. use ++g instead of g++ (it's faster)
6. cc65 can't pass structs by value, nor return a struct.
7. Variables declared globally will compile much faster than local ones. Even structs declared globally will work much faster than local.

And when compiling, use the -O directive to optimize the code. There are also i,r,s directives, which are sometimes combined as -Oirs, which can add another level of optimization. However, each of these can also introduce bugs (for example, reading data from a hardware register, and the data isn't used by the program, it will optimize away the hardware read).

Here's some more suggestions for cc65 coding…

http://www.cc65.org/doc/coding.html (http://www.cc65.org/doc/coding.html)

Why are we so concerned with optimization? Because timing is very important and resources are so slim for an old 8-bit processor. And, clean unaltered C code will compile into very slow code that takes up too much of the limited memory space.

Sharing variables between files…the asm modules (ca65) can share variables/labels with each other with import, export, importzp, exportzp definitions. cc65 can access variables and arrays from asm modules by declaring a variable "extern unsigned char foo;" (and if it's a zeropage symbol, add the line #pragma zpsym ("foo");. When it compiles the C code, it will add an import definition for it. I hardly use "extern". This may be the only time I mention it, except maybe…if you have a large binary file…it's easiest to include it in the asm code like this…

```
.export _foo
_foo:
.incbin "foo.bin"
```

Then to access it from the C code, do this "extern unsigned char foo[];". Note the underscore. For some reason, when cc65 compiles, it adds an underscore before every symbol. So, on the asm side you have to add an underscore to every exported label/variable.

You can call functions in cc65 written in asm with a __fastcall__. This will store the passed variables in the A,X,Y registers, rather than the C stack. A few bits of code can't be done in C, so we might use a library of special ASM instructions that can be imported and called by the C code. (like the startup code). I tend to pass no values, if possible. You can, but it will produce very slow asm code. Lets compare these 2 functions…

(Note, test and A are globals)

```
void Test (char A) {
 test = A;
}
// one passed variable...compiles to 19 lines of code
// 20, if you count the 'lda Foo' just before we jumped here

_Test:

 jsr pusha
 ldy #$00
 lda (sp),y
 sta _test  ; test = A;

 jmp incsp1

pusha: ldy sp
 beq @L1
 dec sp
 ldy #0
 sta (sp),y
 rts

@L1: dec sp+1
 dec sp
 sta (sp),y
 rts

incsp1:

 inc sp
 bne @L1
 inc sp+1
@L1: rts
```

```
void Test (void) {
 test = A;
}
// no passed variables, compiles to 3 lines of code

_Test:
 lda _A
 sta _test
 rts
```

It’s also possible to inline asm code into the C code (I almost never do, maybe I should 🙂). It would look like this…

```
asm ("Z: bit $2002") ;
asm ("bpl Z") ;
```

Another note. I’ve changed the startup code, was crt0.s, now called reset.s (my philosophy is ‘keep it simple’). Also I’ve changed the .cfg file. These files will change from lesson to lesson. I’m using the standard nes.lib file that comes with cc65. I’ve added –add-source to the command line, so recompiling will generate a .s (asm) file with c code included, in case you want to look at the generated asm code.

Going back to sharing variables between modules…my preference is to write all the variables into the C code. Any time you need to reference them in the ASM code, you add and underscore before it, and add the line .import _Foo (or .importzp). You could, alternatively, make a file with all the variables, and declare them all to be global. (Then you won’t have to type .import/.export on any of the ASM files). Mentally, for me, I prefer that things originate with the C code, and flow downhill to the ASM. I am NOT the cc65 expert, so…you don’t have to listen to me.

November 15, 2015April 16, 2017 dougfraker

2 thoughts on “2. how cc65 works”

1. **Matt** says:
   April 23, 2018 at 4:06 am Edit
   Did I miss where we together write the compile.bat file? It seems other commentors have it…

   Reply

   dougfraker says:
   April 23, 2018 at 11:31 am Edit
   I didn’t include any lessons on (.bat) batch file writing. You may have to read the cc65 and ca65 and ld65 documentation (included with cc65 in an html folder) to see how the ‘command line options’ work.

   Reply

Create a free website or blog at WordPress.com.