The Zero Tutorial – Getting Started

Welcome to the Parallax Propeller and the PropGCC programming environment! Congratulations! You've loaded up SimpleIDE and are looking at the hello.c program loaded into the IDE. If you know C, you may want to skip over the C programming part of this tutorial and get to a brief discussion of memory models. If you are new to C and the Propeller, then start paying attention!

Hello World! (of course)

Hello.c is a trivial programming example but we can learn a surprising amount from it. Here's the code in case you aren't near your computer:

```
/*
 * This is a non-traditional hello demo for Propeller-GCC.
 * The demo repeats printing every 100ms with the iteration.
 * It uses waitcnt instead of sleep so it will fit in a COG.
 */
#include <stdio.h>
#include <propeller.h>

int main(void)
{
  int n = 1;
  while(1) {
    waitcnt(CLKFREQ/10+CNT);
    printf("Hello World %d\n", n);
    n++;
  }
  return 0;
}
```

We'll play with this program a little bit for this tutorial, just to give you a feel for C on the Propeller and using SimpleIDE to control how your program is built and loaded into the Propeller.

Comments, comments, comments….use comments to document your code: end of story.

You or those who follow in your footsteps will appreciate them later on. In C, you can denote a block of comments by enclosing them between /* and */ - what the code does, who wrote it and when, what the current version is, etc. These are all good things to comment. If you want a single line comment, use two back-to-back slashes, '//', and the compiler will ignore everything following those up to the end of the current line. This is very handy when you want to comment out a single line of code.

The power of the "#include"

C itself is an uncomplicated language with a small set of keywords a simple syntax and a small set of static data types. The power of C comes from the libraries; both standard system provided libraries and user created libraries. The way to access the code in these libraries is by using the #include directive to bring the library declarations into your source code during compilation allowing you to use the functions from that library. For more information about how #include works, check out a C programming book or tutorial.

Just for fun, comment out the #include <propeller.h> by putting two slashes '//' in front of it and rebuild the program.

Good job! You've just created your first bug! SimpleIDE should have just complained about a few things it couldn't find while it tried to compile your program:

```
propeller-elf-gcc.exe -o a.out -Os -mcog -I . -fno-exceptions hello.c
hello.c: In function 'main':

hello.c:13:17: error: 'CLKFREQ' undeclared (first use in this function)

hello.c:13:17: note: each undeclared identifier is reported only once for each function it appears in

hello.c:13:28: error: 'CNT' undeclared (first use in this function)

Done. Build Failed!
```

The first line is how SimpleIDE called the compiler. For now, we can ignore this but later on, you may use this information during debugging.

Next, the compiler starts to tell you about the problems in your code (get used to this!) In the hello.c, in the function 'main', at line 13, column 17 (hello.c:13:17), an error (versus a warning) was found – CLKFREQ was undeclared ALSO, at line 13, column 28 (hello.c:13:28), CNT was undeclared.

BEFORE you try and use a variable or a function, C requires everything be declared by you (or by someone else in a file that you include in your program). That way, C knows the name, what it is and has some idea how it can be used. By commenting out the #include of propeller.h, you took away those two declarations – now C can't find them and isn't able to compile your program. You'll see these errors in your programs and it usually means you've forgotten to include something or you've spelled something incorrectly or used the incorrect case.

If you didn't know it already, you better learn it now: C IS CASE SENSITIVE!!! In C, a rose is not a ROSE is not a Rose! YES, you WILL make this error!

Time to move on, there's a lot more to talk about! Remove the comment from your #include so your program compiles again.

int main(void)

This is where your program starts. From now until the end of your glorious C programming career, you'll be seeing and writing some variation of this line with each program you write. You must have a function named 'main'. In this example, the compiler now knows that main will return an integer and isn't expecting any parameters as input. For more details on this, it's time to consult your C books and tutorials.

Programming with style

A sensitive topic of debate is coding style and how you format your source code. If the way something is written or formatted doesn't affect the way the program compiles or runs, I won't talk about style in any tutorial. You may see a function declared like this:

```
int main(void)
{
}
```

Or

```
int main(void) {
}
```

They both work. There have been papers written as to why one is better than the other and why the other is better than the one. We're not getting into that debate. Pick what works for you and be tolerant of the choices others make. I certainly won't take points off for style. When you write your compiler, you can enforce your will on others if you so desire.

Programming with Scope

One important thing that a function declaration does is determine the scope or visibility of a variable declaration. For now, we'll keep this simple: anything declared outside of a function declaration is considered GLOBAL and can be seen by every function in a program and also can be made visible outside of your program; anything declared inside a function can only be seen by that function or any function declared inside that function.

I predict this will come back to haunt you and cause at least one stressful debugging session!

If you look at the next line of hello.c, you see the variable declaration and assignment for n:

```
int n = 1;
```

This is pretty straight forward: n is being declared as an integer (int) and the value is being set to 1 (= is the assignment operator in C). Because n is being declared within the braces of main, it is visible in main

and in any function declared within main. That is considered its scope. Comment out the declaration inside main and insert a new declaration above main, so your code looks like this:

```
int n = 1;
int main(void)
{
//int n = 1;
```

You haven't changed the way your program runs but you have changed the scope of n from being local to main to being global to the entire hello.c program. Scope is important and will be the subject of future tutorials.

while (1)

In the embedded world, once you get things in your program set up, you often just sit back and loop and loop and loop forever. The while (1) is a common way to do this – basically, you are saying that while 1 is true (1 is always true), do the following block of code. The Propeller will do whatever is in the braces following the while. This is like the loop function in your sketch if you are coming from the world of Arduino.

To sleep(), perchance to dream, or maybe just waitcnt?

If you know C, you know that there are sleep() and usleep() standard functions in the C library. So, why use waitcnt(), a decidedly Propeller way to make your program pause? As the comment says, if you use waitcnt, your program will fit into a single COG. If you use the sleep() or usleep() library routines, your program will not compile with the COG memory model.

If you want to see this for yourself (and more importantly, the error message you might see if you try this with some other library function, you need to insert

```
#include <sys/unistd.h>
```

After your #includes and then comment out the

```
//waitcnt(CLKFREQ/10+CNT);
```

line like so and insert

```
sleep(1);
```

after it.

Make sure your memory model is set to something other than COG and compile and load your program into the Propeller. It should work….it will just print the lines out much slower than before (1 second between lines).

Now, change your memory model to COG and compile and load again…..GREAT, you broke your program, again!!

```
propeller-elf-gcc.exe -o a.out -Os -mcog -I . -fno-exceptions hello.c
C:\Users\Rick\AppData\Local\Temp\ccmAva09.o: In function `.L2':
(.text+0x34): undefined reference to `_sleep'
collect2: ld returned 1 exit status
Done. Build Failed!
Check source for bad function call or global variable name `_sleep'
```

Without getting into ugly details, the compiler couldn't find anything to match the _sleep reference it generated when you used the sleep() function. The COG memory model libraries don't have the code to support sleep() in them. By using the waitcnt() your program will compile with any memory model.

Finishing out the program

The final two lines of the while(1) block are pretty basic C:

```
printf("Hello World %d\n", n);
n++;
```

printf() prints out the classic "Hello World" and the value of n and the final line of the block increments n using C's post-increment operator.

return(0) to where we came from

In the embedded world in general and the Propeller world in particular, the return isn't required at the end of the main() function. If your program was being started by some sort of supervisor or OS, the final return would allow your program to return a status to whatever started it. It's good coding practice to provide the return to complete your program. If you notice, the value returned must agree in type with the type you specified when declaring your main() function.

Memory Models

If you've gotten this far, you have either suffered through the description of the C program or skipped ahead to get to the good stuff due to your expert command of C. Congratulations, either way!!

Now's the time to break out your copy of the Propeller P8X32A Datasheet and read all about the memory organization of the Propeller's memory so we can briefly discuss the memory models available in the PropGCC environment.

Stemming from the organization of the Propeller memory, PropGCC implements three memory models: COG, LMM and XMM. The compiler generates and loads code based on the memory model you choose to use. Which memory model you choose depends on the hardware available on the Propeller board

you are programming as well as the requirements of your application. As with most things, there are trade-offs made with the various memory models and you need to weigh those trade-offs as they apply to your application.
Each memory model has different hardware requirements and each offers different features and capabilities to your application. There will be future tutorials and documentation getting more in depth with memory models and the impact they have to your programming and how to provide different hardware to support them.

Memory Model Basics

COG - your executable code is loaded into a Cog and must fit within the 2K byte memory of the Cog. This model offers the fastest execution speeds since your code is running as native Propeller instructions in a Cog. Obviously, your code size is seriously limited by the 2K bytes of Cog memory. Due to the program size limits, you may not be able to use many standard library functions. The COG model is supported by ANY Propeller board and does not require any additional hardware. This is only useful for very small programs.

LMM - Cog memory is actually loaded with a LMM (Large Memory Model) Kernel. The kernel does slow down your program's execution speed but it also open up the entire 32Kb of HUB memory for use as program and data space. The LMM model is supported by ANY Propeller board and does not require any additional hardware. This is useful for small to medium sized programs. If you are using the Propeller and PropGCC (and/or SimpleIDE) to learn C programming, this is a good memory model. It allows you to use most C functionality, loads to the propeller quickly and runs on any Propeller board.

XMM - Cog memory is loaded with the XMM Kernel. All program code is loaded into external memory (64K EEPROM, SRAM, flash or SD). The XMM cache and program stack reside in HUB memory. The program's data can reside in either HUB memory or in external memory depending on which XMM option you choose. This is the slowest memory model by far but it does open the door to very large programs. To support this memory model, your hardware needs to have either 64K EEPROM, SRAM, flash or an SD card.

Going Forward

At this point, if you are in the position of needing or wanting to learn C by working through more traditional exercises from books or tutorials, you can do most anything you need to do from a programming standpoint on the Propeller using the LMM memory model.

It's always a good time to review the Propeller datasheet if you are still learning your way around the chip's hardware or look through the standard header files from the library to see what code is provided for you by the PropGCC package.

There are also a growing number of demo programs provided with the PropGCC and SimpleIDE distribution packages that are excellent ways to learn more about programming the Propeller in C.

Hopefully, there will even be more tutorials to help walk you through those examples and many other features of the Propeller and C programming.

Revision History
Rev 1.0 – June 25, 2012