The Toggle Tutorials

“If you control the pins, you can control the world” – Napoleon Bonaparte

Introduction

This guide is intended to develop an understanding of Propeller GCC C programming on the Parallax Propeller micro-controller. It is not a C programming guide and is not a detailed introduction to micro-controllers or the Propeller hardware architecture. It is a way for a programmer, hopefully with C experience and some Propeller experience to learn how to leverage the special features of the Propeller through C code.

You can use C and the Propeller to run small C example programs as a general purpose computer. It allows you to do I/O to a terminal emulator and perform numeric and symbolic manipulation of data and perform most of the functions of a small personal computer. This is fine for learning C but misses out on the real reason for having a microcontroller as pointed out in Napoleon’s quote above (ok, maybe it wasn’t Napoleon).

Physical computing is about connecting to the physical world and the Propeller does that through its 32 I/O pins. If you can control those pins through a program, you truly can control the physical computing world.

Through a series of progressively more sophisticated programs that basically toggle I/O pins, we will learn how to control the physical I/O on the Propeller and also how to leverage the multi-processing power of the Propeller through C in various ways. Along with that, we’ll learn how to take advantage of the various memory models that are made available through the Propeller GCC implementation.

Each section starts with the program that will be used to develop the concepts of that section. I try to heavily comment the code so if you read through it and understand what is going on, please skip to the next section. You can always come back and I certainly don’t want anybody to get bored going over things they already know.

By the end of the tutorial, you should be able to write simple LMM programs, LMM programs with COGs running C code, LMM programs with COGs running GAS or PASM assembler code, LMM programs with multiple threads and XMM programs using pseudo-threads (pthreads) and just about any other memory or COG configuration that’s supported. If some of the terms sound a little confusing or daunting, then you are in the right place.

Most of all have fun. If programming at this level isn’t enjoyable, then I seriously suggest trying some other language on the Propeller. You can always come back at another time when you have the need or desire.

Tutorial #0 – Assumptions and preparations

#1) You have some programming experience – preferably with C. If you have SPIN and PASM experience, that's even better!

#2) You have experience with the Parallax Propeller and it's hardware architecture. It's not your typical micro-controller and has features that this tutorial will highlight and exploit through C.

#3) You have enough computer experience to create and edit program source files, navigate directories, and work in the command prompt or terminal environment on your computer.

#4) You are comfortable with decimal, hexadecimal and binary numbers and have some knowledge of logical and bitwise operators and bit manipulation.

#5) You have installed the latest Propeller GCC (PROPGCC) release for your operating system on your computer. The programs for this tutorial are contained in the toggle subdirectory under the demos subdirectory – you should be able to find this after you install PROPGCC.

#6) I'm using a 64KB QuickStart board for testing examples unless stated otherwise in a particular section. For most tutorials, any Propeller board should work as long as it has an LED to blink.

#7) This document and all code examples are covered under the MIT license and follow the terms of use outlined in that license.

Tutorial #1 – lmm_toggle

The Code:

```
int main(int argc, char* argv[])
{
   int mask = 0x3fffffff;           // Create an integer variable mask and
                                    // initialize it as a bit mask with 1's in
                                    // each place except for bit 30 and 31

   int freq = CLKFREQ>>1;            // Initialize the integer variable freq
                                     // to the system clock freq divided by 2
   DIRA = mask;                     // Set the Propeller direction register to
                                    // the bitmask - in this example, all bits except
                                    // 30 and 31 will be set as outputs.

   for(;;) {                        // This is an empty (infininte) for loop
      OUTA ^= DIRA;                 // Set the Propeller OUTA register to iteself
                                    // XOR'd with the bit mask.
                                    // XOR is a common way to toggle bits or 'flip'
                                    // bit values.

      waitcnt(freq+CNT);            // Cause the COG to wait (idle) until the system
                                    // counter reaches a value of freq+CNT.
                                    // CNT is the current value of the system counter.
   }
}
```

Questions:

So what is LMM and why is it important?

What's in propeller.h

What does **int mask = 0x3fffffff** do?

What does **int freq = CLKFREQ>>1** do?

What do **DIRA = mask** and **OUTA ^= mask** do?

What does **waitcnt(freq+CNT)** do?

How do I compile a C program in LMM mode to run on the Propeller?

If you feel comfortable with these questions, then feel free to skip to the next tutorial. If not, you may want to read a bit further in this tutorial.

What is LMM?

The Propeller port of GCC supports a number of memory models, or ways of storing the program in memory. Basically these models provide a trade off of speed for code space. By far the fastest model is the native “cog” model (-mcog) in which machine instructions are executed directly. However, in that model only the 2K of internal COG memory (actually slightly less) is available for code. In the other models (LMM, XMMC, and XMM) code is stored in RAM external to the cog and is loaded in by a small kernel. This makes more space available for the code (and, in the XMM case, the data) but at the cost of having kernel overhead on each instruction executed (See Appendix A for memory model details)

The default memory model is LMM. In this model, your program is loaded into HUB ram and an LMM interpreter is loaded into COG 0. The LMM interpreter fetches your program instructions from the HUB memory and executes them in COG memory. LMM programs are still bound by the 32K memory of the propeller. LMM suffers a speed penalty since the entire program is not In COG memory but it is significantly faster than interpreted SPIN code. LMM programs should run on any Propeller and any of the current production boards.

What’s in propeller.h?

Propeller.h is one of the GCC implementation specific header files supporting the propeller libraries. It contains hardware definitions and function prototypes for a portion of the Propeller specific C library. The C code implementing the functions in propeller.h can be found in propeller.c in the libpropeller directory. You need to include propeller.h in your program so you can reference the Propeller registers and other hardware structures.

What does   int mask = 0x3fffffff do?

In the Propeller GCC port, integers are stored as 32 bit values. This piece of code loads the variable **mask** with a very LARGE number but we aren’t going to be using it as a number, we are going to be using it as bits to load into Propeller registers. If you look at this number as a string of bits in a 32 bit register, you see:

Bit 31	Bit 30	Bit29	Bit 28	Bit 27	… …	Bit 3	Bit 2	Bit 1	Bit 0
0	0	1	1	1	1	1	1	1	1

If this doesn’t makes sense, please review binary and hexadecimal number systems and bit manipulation (online reference?)

By using this value as the bit mask, you are saying you will be working with ALL pins except for 30 and 31. Those we don’t touch since they are used for serial I/O with your computer in these examples. If the board you are working with has problems with the other 30 pins toggling from HIGH to LOW, then the bit mask needs to be adjusted. If you have an LED on pin #0, then your bit mask would be 0x01, if you are using a QuickStart and just want the LED pins to be toggled, you could use 0x0ff0000, which just selects bits 16 to 23.

What does int freq = CLKFREQ>>1 do?

This line makes the contents of freq equal to the Propeller clock frequency (80,000,000 in this example) shifted to the right by 1 bit. ‘>>’ is the C bitwise operator for Shift Right. By shifting a number to the right by one bit you actually divide it by 2 (2 bits = /4, 3 bits = /8, etc.). CLKFREQ contains the number of clock ticks per second which in this examples defaults to 80,000,000 for an 80MHz clock. By dividing it by two, you get the clock ticks in ½ second. This will be the duty cycle for the flashing LEDs.

What do DIRA = mask and OUTA ^= mask do?

DIRA and **OUTA** are more of the Propeller registers that are defined in the propeller.h header file.

The first piece of code loads the contents of **mask** into DIRA, the direction register. Setting the bits in that register to 1 make them outputs, 0's are inputs.

The second piece of code sets the Propeller OUTA register equal to exclusive OR result of the OUTA register and the variable mask.

Say what??

'^=' is the C compound operator for a bitwise exclusive or. We are doing an exclusive OR of **OUTA** and **mask**.

Say what??

A compound operator in C takes the left side operator, performs the indicated operation on it and the right side operator and then places the result back into the left side operator. It's a shorthand way of writing **OUTA = OUTA ^ mask**.

The exclusive OR (XOR) is a logical operator that compares the bits in each variable. If both bits are 0 or both bits are 1, the result is 0, if EITHER bit is 1 then the result is 1. This has the effect of toggling the bits in OUTA 0 to 1 or 1 to 0 each time through the loop. If you are uncomfortable with logical operators and bitwise operators, please review them. Bit manipulation is used extensively in programming micro-controllers. (online reference)

What does waitcnt(freq+CNT) do?

Waitcnt is a direct translation of the SPIN **waitcnt** instruction. You are telling the COG to wait until **CNT** (the system counter) to reach the value **freq+CNT**. **CNT** is the system counter made available through its definition in propeller.h and **freq** is the time we want to wait that was calculated earlier. In this case, the COG will wait idle for 0.5 seconds and then resume instruction execution where it paused.

How do I compile a C program in LMM mode to run on the Propeller?

Assuming you have a Propeller board connected to your computer and you know what port # it is on, this is pretty easy for the examples.

The good news is that each example has a make file in its directory. Make files are beyond the scope of this tutorial. For now, for any of the tutorial examples, if you are in the directory for that example, you can type the following at the operating system prompt to rebuild an executable Propeller GCC binary:

```
>make clean
>make
```

To load the executable into the Propeller, type:

```
>propeller-load –r –pn toggle.elf
```

Where *n* is the port number the Propeller is connected to.

If you've done everything right, you should see some blinking LEDs.

If I've done everything right, you should have some idea WHY they are blinking.

On to Tutorial #2!!

Appendix A. – Memory Models

The Propeller port of GCC supports a number of memory models, or ways of storing the program in memory. Basically these models provide a trade off of speed for code space. By far the fastest model is the native "cog" model (`-mcog`) in which machine instructions are executed directly. However, in that model only the 2K of internal memory (actually slightly less) is available for code. In the other models (LMM, XMMC, and XMM) code is stored in RAM external to the cog and is loaded in by a small kernel. This makes more space available for the code (and, in the XMM case, the data) but at the cost of having kernel overhead on each instruction executed.

COG

As previously mentioned, in the COG memory model code is stored directly in the internal memory of the Propeller's cogs. This memory is 512 words, or 2048 bytes, long, but 16 words are reserved for hardware registers. The GNU C compiler reserves an additional 17 words for compiler registers. These registers, named `r0-r14`, `lr`, and `sp`, are used by the compiler for various purposes. `r0-r7` are temporary registers within a function (not preserved across function calls). `r8-r14` are working registers within a function and are preserved across function calls. `lr` is the "link register" which holds the return address for functions (except for functions declared with the "native" attribute, for which the return address is stored directly in the `ret` instruction). `sp` is the stack pointer.

Assembly programming in COG mode

Inline assembly (or writing external C callable functions in assembly) in COG mode is very straightforward – the Propeller instruction set is directly usable. The GNU assembler gas accepts input that is very similar to PASM. The major differences are: (1) gas does not understand local labels (labels that start with `:`), and (2) by default addresses are considered to be byte addresses rather than word addresses; you can tell gas to treat them as words by using the `.cog_ram` directive. The instruction set is the same, although for convenience gas offers a `mova` ("move address") instruction that works just like `mov` but which divides any immediate source value by 4, thus converting it from a byte address to a word address. `mova` is not needed in .cog_ram mode, but is convenient in the default mode.

LMM

In the Large Memory Model (LMM mode), the program code is stored in hub memory, which is 32K in size. The program's data and stack are also stored there. A small kernel program (the "LMM Kernel") runs in COG memory and loads and executes program instructions from hub memory. In this mode the same 17 compiler registers are available as in hub mode (`r0-r14`, `lr`, and `sp`), but there is an additional register `pc` which is a pointer to the next instruction to be fetched.

Assembly programming in LMM mode

In LMM mode most PASM instructions can be used, except for any jump or call instructions (including `djnz`, `tjnz`, and `jmpret`). Instead of doing direct jumps, we have to modify the value of the `pc` register used by the LMM interpreter instead. This may be done either by directly adding an offset to `pc`, or by loading a new value into it. gas provides a pseudo-instruction `brs` ("branch short") which translates into an immediate `add` or `sub` of the `pc`; this may be used to branch to a destination within 508 bytes (plus or minus) of the instruction following the `brs`. For longer branches we can use the `__LMM_JMP` routine built into the kernel, or else move a different register into `pc` (for an indirect jump). After the `jmp #__LMM_JMP` instruction comes a 4 byte address indicating the new value for the `pc`.

Calls are handled by the `__LMM_CALL` and `__LMM_CALL_INDIRECT` kernel functions. `__LMM_CALL` is like `__LMM_JMP` except that before it loads the new `pc` it saves the old value into the `lr` register. The called function can thus return with a simple `mov pc,lr` instruction. `__LMM_CALL_INDIRECT` uses a special register, `__TMP0`, as the address to call; thus, an indirect call via compiler register `r6` would be coded as:

```
mov __TMP0, r6
jmp #__LMM_CALL_INDIRECT
```

`__LMM_CALL_INDIRECT` also saves the return address in `lr`.

XMMC

In the eXtended Memory Model – Code (XMMC mode) the program code is stored in external memory, either flash or ram. The exact size of this memory depends on the board used, but it is generally quite a bit larger than hub memory. The program's data and stack reside in hub memory as in LMM mode.

Assembly programming in XMMC mode is exactly the same as in LMM mode.

XMM

In the eXtended Memory Model both the code and data are placed in external memory (in the case of data it must be an external RAM). This allows for the largest possible programs, but at a considerable cost in execution time, since all data accesses must go through functions in the XMM kernel. As in all other modes, the stack remains in hub memory (and hence is limited to 32K bytes less the cache size used by the XMM kernel, which is typically 8K bytes).

Assembly programming in XMM mode is very similar to LMM mode, except that data accesses which may point into external memory must be done via the appropriate kernel functions instead of directly with `rdlong` and `wrlong` instructions.

Kernel APIs

The LMM, XMM, and XMMC kernels all provide some common functions which may be used by programs (and which the compiler takes advantage of).

Data movement

```
jmp #__LMM_MVI_rn
long val
```

This is a "move immediate" instruction to move the value "val" into compiler register `rn` (or the link register `lr`). After the instruction is complete the LMM interpreter resumes automatically.

```
mov __TMP0, #(count<<4)|regnum
call #__LMM_PUSHM
```

Push multiple compiler registers onto the stack. "count" is the number of registers to push, and "regnum" is the number of the first register to push (0 for `r0`, 1 for `r1`, and so on; register number 15 is `lr`). Note that "count+regnum" should be less than or equal to 16. The registers count up, so if regnum is 12 and count is 3 then `r12`, `r13`, and `r14` will be pushed (in that order).

```
mov __TMP0, #(count<<4)|regnum
call #__LMM_POPM
```

Pop multiple compiler registers from the stack. "count" is the number of registers to pop, and "regnum" is the number of the first register to pop, which should be the last register pushed (0 for `r0`, 1 for `r1`, and so on). Note that "count+regnum" should be less than or equal to 16. The registers count down, so if regnum is 14 and count is 3 then `r14`, `r13`, and `r12` will be popped (in that order).

Branches and calls

```
jmp #__LMM_JMP
long addr
```

Moves the immediate value `addr` into the `pc` register (so the LMM interpreter will begin to execute at address `addr`, thus performing an unconditional jump).

```
jmp #__LMM_CALL
long addr
```

Moves the address of the next instruction into the `lr` register, and moves `addr` into the `pc` register (so the LMM interpreter will begin execution at address `addr`, which is typically a subroutine).

```
mov __TMP0,rn
jmp #__LMM_CALL_INDIRECT
```

Moves the address of the next instruction into the `lr` register, and moves `__TMP0` into the `pc` register (so the LMM interpreter will begin execution at address, it contained, which is typically a subroutine).

Math functions

```
call #__UDIVSI
```

Performs 32 bit unsigned division of the value in compiler register `r0` by the one in `r1`. Returns the quotient in `r0`, and the remainder in `r1`.

```
call #__DIVSI
```

Performs a 32 bit signed division of `r0` by `r1`. Returns the quotient in `r0`, and the remainder in `r1`. The sign of the remainder is chosen so as to be consistent with the C standard.

```
call #__MULSI
```

Multiplies the two 32 bit numbers in `r0` and `r1`, and returns the (low order) 32 bits of the result in `r0`.

```
call #__CLZSI
```

Counts the number of leading 0 bits in register `r0`, and returns the result in `r0`. For example, if `r0` = 0x00008100, then the result will be 16. This is useful for normalizing fixed and floating point numbers.

```
call #__CTZSI
```

Counts the number of trailing 0 bits in register `r0`, and returns the result in `r0`. For example, if `r0` = 0x00008100, then the result will be 8.

Miscellaneous functions

```
jmp #__LMM_FCACHE_LOAD
long nbytes
```

Loads some code into the fast cache and executes it. The code to be loaded and executed immediately follows the load sequence, and is `nbytes` bytes long (which must be a multiple of 4). Code executing in the fast cache must use only COG instructions, and must not refer to the program counter `pc` (`jmp` instructions must be used instead, with labels referring to the COG memory region starting at `__LMM_FCACHE_START`, which is where it will be executed from. At the end of the fastcached block should come a `jmp __LMM_RET` instruction (note that this is an indirect jump).

For example, the C strcpy function:

```
char *strcpy(char *dst_orig, const char *src) {
    char *dst = dst_orig;
    while ((*dst++ = *src++) != 0)
        ;
    return dst_orig;
}
```

is translated by the compiler into code using the fast cache like:

```
        .global _strcpy
_strcpy
        mov     r7, r0
        jmp     #__LMM_FCACHE_LOAD
        long    .L6-.L5
.L5
.L2
        rdbyte  r6, r1
        cmp     r6, #0 wz
        add     r1, #1
        wrbyte  r6, r7
        add     r7, #1
        IF_NE   jmp     #__LMM_FCACHE_START+(.L2-.L5)
        jmp     __LMM_RET
.L6
        mov     pc,lr
```