PROPELLER MICROCONTROLLER

PROPELLER MICROCONTROLLER
Truly Parallel Microcontroller Programming

Nikita Ermoshkin

CONTENTS IN BRIEF

CONTENTS

LIST OF FIGURES

LIST OF TABLES

INTRODUCTION

The Propeller microcontroller is cool, you should use it. This book won't make you an expert on every corner of PropGCC, but you should be able to do most projects. Complete code examples will be on the GitHub repository. This book assumes you know the basics of programming and of C.

Parallax provides the Simple Tools Library to provide easy access to the Propeller's functionality. While it is great for prototyping ideas and is super easy to use, it is inefficient and by nature, hides the Propeller's inner workings, making it a poor tool for learning the Propeller architecture. So, we will avoid using the Simple Tools Library while exploring the Propeller. That being said, to avoid getting too in the weeds too early, we won't dig into the nitty gritty details, at first.

The end of each chapter will have a series of exercises (with potential solutions avialble on GitHub) to help review the topics covered in each section.

CHAPTER 1

THE PROPELLER ARCHITECTURE

Describe each COG, its registers, memory, shared memory, general IO stuff, clock/counter. basically first part of propeller manual.

CHAPTER 2

FIRST PROGRAM

2.1 Choice of IDE and Project Setup

SimpleIDE, Eclipse, make, etc. All viable options, but this book will focus on a text editor and Make files to build projects. The simplest option for Windows is to install VirtualBox and your favorite Linux distribution, such as Ubuntu or Fedora.

2.1.0.1 How a Propeller Program is Built

In general, Propeller programs are built in the following way, just like any other C program (with one extra step to load it onto the actual hardware):

1. Compile each .c file into a .o file
2. Link together all .o files and .a (static libraries) into a .elf file
3. Load the .elf file onto the chip and run the program

2.2 Blinking an LED

Every introduction to microcontrollers begins with a blinking LED program, so we will too. Open Example 2.1. This program will blink an LED on P16 at 1Hz forever.

2.2.1 Compiling and Running on a Quickstart

Open up the Makefile, and edit the `PORT` variable to be the serial port corresponding to the Propeller. Open up the terminal, enter the directory containing the project and run:

```
make
```

The code should load into RAM and the P16 LED and blink!
We will explore the structure of this program in detail.

```
1  #include <propeller.h>
2
3  #define p_LED 16
4
5  void main() {
6      DIRA |= 1 << p_LED;
7
8      while (1) {
9          OUTA ^= 1 << p_LED;
10         waitcnt(CNT + CLKFREQ/2);
11     }
12 }
```

Example 2.1 LED Blink

Let's examine this program line by line. First, we include `propeller.h`, which defines all of our system functions, keywords, and registers. Next, we define a constant for the pin number of our LED, P16. Note that this is a logical IO number, not a physical chip pin number. Next, we define our main function. This is what gets called first after all the C setup code generated by the compiler. The main function does some setup, then starts an infinite while loop that will never exit to blink our LED forever.

The only set up necessary is to set the direction of the IO pin to output. This is done via the `DIRA` register, which is the **DIR**ection register for Port **A**.

2.2.2 The Direction Register

The two direction registers, DIRA and DIRB, control the direction of the Port A and Port B IO pins. On the Propeller 1, only Port A exists. Port B is reserved for future use on a larger chip. From now on, we will only ever use Port A and DIRA, but if DIRB is ever implemented, the same principles will apply.

DIRx is a 32-bit register, with each bit corresponding to a pin in the Port. On the Propeller 1, bit 0 corresponds to P0, bit 1 to P1, and so on. A 0 bit sets the corresponding pin to an input and a 1 sets the pin to an output.

In the above code listing, DIRA is OR'ed with a number that has a 1 in 16th bit, corresponding to P16. This sets bit 16 of the DIRA regsiter to a 1, which is what we want to drive an LED!

Next, we enter the while loop and toggle the state of the output pin via the OUTA register, the Port A OUTput register.

2.2.3 The Output Register

The two output registers, OUTA and OUTB, control the output state of the Port A and Port B IO pins. Just like with DIRA, we will only use OUTA. Also like DIRA, each bit of the register corresponds to a pin. Writing a 1 to a bit sets the pin to high and a 0 sets the pin

to low, if the corresponding DIRA bit is set to 1. If the DIRA bit is set to 0, writing the OUTA register has no effect.

In the above code listing, we XOR the 16th bit of the OUTA register to toggle it's state, toggling the output of the pin.

Finally, we wait for half a second before looping back to the toggle the pin again. Waiting in a Propeller program is done using the `waitcnt()` function. The `waitcnt()` will pause execution until the system counter register, CNT, is equal to the value passed into the function.

2.2.4 The CNT Register

The CNT register is a 32-bit unsigned integer that increments by one every clock cycle.

So to wait for half a second, we need to determine what the CNT register will be in half a second. To know this, we need to know out clock speed, and this is given by the CLKFREQ register. This is a constant set during compilation based on the board configuration and clock settings. In almost all cases, it will be either 80 Mhz or 100 Mhz. Regardless of what it is, we know that there are CLKFREQ number of clock cycles in one second, so we add half of that to the current CNT register and pass that into the `waitcnt()` function.

The result is half a second on, half a second off.

2.3 Tips and Tricks for IO Pins and Timing

The OUTA register bits only take effect when the corresponding DIRA register bits are set. However, you can build an open-source or open-drain style output by writing to the OUTA register once and using the DIRA register to toggle the state of the output–driven or released. This is super useful when building drivers for communication protocols such as I2C.

2.3.1 Accurate Timing

Take a moment to think about the mechanism behind executing the line `waitcnt(CNT + CLKFREQ/2);`. How does it compare to `waitcnt(CLKFREQ/2 + CNT);` and why should you use the former instead of the latter? The answer is simple: CNT is always changing. [Reference prop manual for an explanation on this].

Additionally, the above code does not run at exactly 1Hz, but slightly less, due to the extra time in the loop caused by executing the actual commands in the loop. So, instead of waiting for half a second after executing the loop, we save the system counter at the beginning of the loop, and then tell `waitcnt()` to wait until half a second after that time has elapse. [Reference propeller manual for better explanation]

```
1 //...
2 uint32_t t = 0;
3
4 while (1) {
5     t = CNT
6     OUTA ^= 1 << p_LED;
7     waitcnt(t + CLKFREQ/2);
8 }
9 //...
```

In the above modification, the loop will run at almost exactly a half second period. The ”almost” comes from the fact that saving the CNT to t, setting up and calling `waitcnt()`, and returning to the top of the loop all take time.

2.4 Exercises

1. Modify Example 2.1 to blink the LED 5Hz.
2. Modify Example 2.1 to blink both the P16 and P17 LED at the same rate.
3. Modify Example 2.1 to blink the LEDs on P16 at 1Hz and P17 at 3Hz. Then add the P18 LED to blink at 1.3 Hz. You’ll notice this is increasingly difficult and frustrating. Don’t worry too much about completing this exercise, it serves to prove a point.

CHAPTER 3

THE PROPELLER COG

Now, we will look into what makes the Propeller special: the cog. Recall how the cog works. It's a core of the propeller with it's own RAM, video generator, I/O interface, etc, with access to the hub memory in a round-robin fashion. In our first program, everything ran on cog 0. Now, we will explore a program that utilizes multiple cogs at once (rather inefficiently).

Lets take a look at Example 3.1, shown below.

```
1  #include <propeller.h>
2
3  #define p_LED 16
4
5  unsigned stack1[64];
6  unsigned stack2[64];
7  unsigned stack3[64];
8
9  struct led_param_t {
10     uint8_t pin;
11     uint32_t delay_time;
12 };
13
14 void blink_led(void* par) {
15     struct led_param_t *led = par;
16
17     DIRA |= 1 << led->pin;
18
19     while (1) {
20         OUTA ^= 1 << led->pin;
```

```
21          waitcnt(CNT + led->delay_time);
22      }
23  }
24
25  void main() {
26      struct led_param_t led1 = {16, CLKFREQ/2};
27      struct led_param_t led2 = {17, CLKFREQ/3};
28      struct led_param_t led3 = {18, CLKFREQ/5};
29
30      cogstart(blink_led, &led1, stack1, sizeof(stack1));
31      cogstart(blink_led, &led2, stack2, sizeof(stack2));
32      cogstart(blink_led, &led3, stack3, sizeof(stack3));
33
34      while(1);
35
36  }
```

Example 3.1 LED Blink with Cogs

When you run it, you'll see the three LEDs blink at different rates. Let's examine this program section by section. First, we define some arrays that will function as stacks for each cog. Each stack is used by each cog to store local variables and allow for function calls, just like a normal processor's stack. The stack must be at least 40 ints large plus space for any local variables and regular stack use. In general, 64 is a good starting point, but larger stacks will be required for more complicated programs. The Simple Tools Library has a tool for determining a cog's memory usage.

Next, we define a function that our cogs will run. It must always have the prototype: `void function_name(void *par)`. The par parameter is a value that can be passed into the cog once it is started. Usually, this is an address to a memory location for a mailbox to pass data between cogs. In this example, the cog function is similar to Example 1.1's main function and the mailbox is a struct containing the pin number of the LED to blink and the delay time of the main loop (to set the blink rate). The main difference is how we use the par pointer. When the cog is started, we pass in a pointer to an instance of the led_param_t mailbox, which is held by par. So we cast the par pointer to an led_param_t and then we can read out the data was set before the cog was called. Pretty cool, right?

The main function does two things: initialize three instances of our LED mailbox and start our cogs. Note that the delay time is in clock tick units, not seconds or milliseconds. It is important to think about time in terms of clock ticks rather than seconds. Starting a new cog is done with the `cogstart()` function, which takes four parameters: the function to run, a value to be passed to the par register (can be NULL, if no parameter is needed), a pointer to the start of the cog's stack and the size of that stack, in bytes. The function returns the ID of the started cog, but that only needs to be saved if the cog needs to be stopped later. Finally, we enter an empty loop while our LED blinking cogs start in the background.

3.1 Exercises

1. Repeat Exercise 2.3, but this time using cogs. Much easier than the first time, right?

CHAPTER 4

DEBUGGING AND THE SERIAL INTERFACE

The most useful way to debug a program is by printing to a serial console. This is done with the standard C `printf()` function. Example 4.1 shows simple printing from a propeller program over a serial port.

```
1 #include <propeller.h>
2 #include <stdio.h>
3
4 void main() {
5
6     printf("Hello World!\n");
7
8     while(1);
9 }
```

Example 4.1 Hello World

This looks just like any "Hello World" program out there. Below is the default configuration for the serial port, but it can be customized (described later).

- 115200 baud
- 8 bits, no parity, 1 stop bit
- No hardware or software flow control (so disable, CTS, DTR, etc)

4.0.0.1 Some Info on Serial Printing

When using the `printf()` function, the compiler includes some code to initialize P30 and P31 for UART communication (more on this

later). This allows the printing function to work function in cog 0. Note that it can't be used in any other cog. The driver runs in the same cog and does not buffer data, so input cannot be read unless actively checking for input. All the basic format specifiers, such as %d, %s, %c, %x, etc, can be used to format the output as usual. Note that %f is not supported by default (we will get to floating point later in this book, for now, just assume it doesn't exist).

4.0.1 Device Drivers

The Propeller Library includes two device drivers: the Simple Serial driver and the Full Duplex Serial driver. The Simple Serial driver runs in the same cog and is "half-duplex", meaning it can only send or receive at a given time, but not both. Additionally, the Simple Serial driver is not buffered, meaning input can only be read when the program is asking for it and anything received at any other time won't be detected. It would be nice to buffer data as it comes in, while the processor is doing another task, so how would you do this on a Propeller? Run the receiver in a cog! While I encourage you to write your own as an exercise, the Propeller Library includes the Full Duplex Serial driver to do this for you.

The Full Duplex Serial driver runs in a separate cog and will buffer received data as it comes in, regardless of what else the program is doing. This means that even if the main cog isn't checking for input, anything received will be saved until the program is ready to read it.

4.0.1.1 Installing a Driver

Example 4.2 shows how a driver can be "installed" and used by the program. This example installs and uses the Full Duplex Serial driver.

```
1  #include <propeller.h>
2  #include <stdio.h>
3
4  extern _Driver _FullDuplexSerialDriver;
5
6  _Driver *_driverlist[] = {
7    &_FullDuplexSerialDriver,
8    NULL
9  };
10
11 void main() {
12     stdin->_flag |= _IONONBLOCK;
13     stdin->_flag &= ~_IOCOOKED;
14
15     char buf[64] = {0};
16     while(1) {
17         waitcnt(CNT + CLKFREQ*2);
18         fgets(buf, sizeof(buf), stdin);
19         printf("while waiting, I received: %s\n", buf);
20     }
21 }
```

Example 4.2 Device Drivers

We've done a few new things in this program. First, we defined a custom driver list. This tells the compiler and linker what drivers we want to use in our program. Then we set and clear some special flags on the standard input file. Setting the `_IONONBLOCK` flag tells the driver to not block when performing a read–if no data is available, don't wait for it, just return immediately. Clearing the `_IOCOOKED` flag tell the driver to use "raw" mode instead of "cooked" mode. In "cooked" mode, the driver will echo back any received characters,

as well as interpret special characters like backspaces to remove data from the buffer. This is useful for creating a shell-like interface where a user is entering data into a terminal. However, when using raw mode, none of this is done and data is presented exactly as it's received, special characters and all. Figuring out what the rest of the program does is left as an exercise for the reader–it is a combination of topics we have already covered and some standard C code.

4.0.1.2 *Interfacing Using FILEs*

All of these drivers work by reading and writing standard C `FILES`. This may seem confusing, since there is no actual file system on the Propeller. PropGCC takes advantage of the `FILE` structure provided by the C Standard Library to perform it's serial interfacing. There are two special "files" on the propeller. One for the Simple Serial Driver, one for the Full Duplex Serial driver. When opening one these files, it is actually doing all the setup necessary to use that driver and the file name specifies the parameters to open the serial port with. The Simple Serial driver file is `SSER:b,r,t`, where b is the baud rate to use, r is the receive pin to use, and t is the transmit pin to use. Similarly, the Full Duplex Serial driver file is `FDS:b,r,t`, where b, r, and t are the same as before. For example, let's say you want to use the Full Duplex Serial Driver, transmitting on P15 and receiving on P16, with 57600 baudrate. Simply open a new file to create this interface:

```
1 //...
2 FILE *f = fopen("FDS:57600,16,15", "r+");
3 fprintf(f, "Hello World!");
4 //...
```

When a C program starts, it opens three files and assigns them to stdin, stdout, and stderr (that's "standard in", "standard out" and "standard error"). On a normal Linux system, these are the file structures that interface with the terminal shell that started the program. On the Propeller, this "terminal shell" is a serial interface. By default, the Propeller opens the file `SSER:115200,31,30`. That means it's the Simple Serial driver with 115200 baudrate, receiving on pin 31 and transmitting on pin 30 (these are the port parameters used to load programs onto the Propeller). It is opened three times, once for reading (stdin) and twice for writing (stdout and stderr).

4.1 Exercises

1. Write a buffered serial driver to receive input in a background cog, without using the Full Duplex Serial driver.

2. Use the Full Duplex Serial driver to write a simple program that prints the integer representation of the received character, while blinking an LED at 1Hz, without using any extra cogs.

CHAPTER 5

MEMORY MODELS

If you look at the Makefiles of the previous examples, you'll see that there is a variable `MMODEL` set to `cmm`. What this is doing is telling the compiler to generate our program using the **C**ompact **M**emory **M**odel. There are several memory models available, this chapter will focus on exploring the following three models:

1. Cog Memory Model (COG)
2. Large Memory Model (LMM)
3. Compact Memory Model (CMM)

The difference between these models lie in how the compiled code is stored on the Propeller, each with different resulting memory usage and execution speed. There are other memory models as well, but they are not as useful as they require external hardware and are pretty slow.

5.1 Cog Memeory Model

The Cog Memory Model is the simplest to understand. The compiled ASM code simple lives in the cog's RAM and is executed directly. Super straight forward, with one (rather big) caveat. Recall that each cog has only 2KB of RAM, and some of that is reserved by the chip for the built-in registers and look-up tables. This means that the maximum size of the program with this memory model is only 2KB. You'll find that it's very easy to make

a program larger than that. As a result, you can't do things like serial printing or floating point math, as those libraries take up way more than 2KB.

There is a light at the end of the tunnel, though. The benefit of the cog model is that it is FAST. It is the fastest of all the memory models. Since the code lives in the cog's RAM (instead of the hub RAM, as you will see later with other memory models), there's no overhead of loading an instruction before executing. This makes it a great choice for anything that requires very precise and very fast timing, such as servo control or communication protocols. While most projects won't solely be in the cog memory model, chances are you'll have some kind of driver that will require it.

Example 5.1 shows an example of how a cog program is written and compiled. There are some small difference to note.

Example 5.1 Cog Memory Model

5.2 Large Memory Model

The 2KB memory limit really puts a damper on the kinds of programs we can write. This is where the Large Memory Model comes in. The Large Memory Model, or LMM, stores the compiled code in the main hub RAM of the Propeller, and a small kernel runs on cog 0. This kernel grabs code from the Hub RAM and executes it. Our kernel is less than 2KB, so it lives in the cog just fine. The actual compiled program lives in the 32KB hub RAM, giving us a lot more breathing room. However, I'm sure you can already see the major hit taken when using this memory model. For the kernel to get the next piece of code to execute, it must access hub RAM, which takes longer than a typical instruction, as long as 28 clock cycles (refer to Chapter 1 to remind yourself exactly why this is the case). As a result, the program executes slower. The communication protocols with fast clocks might not survive this speed downgrade, but it's still pretty fast and on par with a typical microcontroller of this caliber.

5.3 Compact Memory Model

When using the LMM, we gain a lot of extra room for programs. But even this will eventually not be enough, especially when we start linking the math library and using `printf()`. This is where the Compact Memory Model, or CMM, comes in. The CMM is similar to the LMM in that there is a small kernel running in the cog and the actual program is stored in hub RAM, but the stored code is actually compressed from the original assembly generated by the compiler. So there is less data to read per instruction and we can fit more instructions in the 32KB hub RAM, which sounds great! However, like all the memory models, we trade speed for program size. The kernel in cog 0 needs to decompress the stored code back into assembly instructions Propeller can understand, which takes time. As a result, this is the slowest of the three models examined here, but provides quite a bit of extra program space. As a result, this is often the most common model for Propeller programs.

5.4 Determining Program Size

While developing large programs, it is often useful to be able to see how big the program actually is. The Propeller GCC package comes with a binary called `propeller-elf-size`. When passed a compiled program in .elf form, it will break down the sizes of different sections of the program. This can be useful for catching large memory usag and determining what causes it.

5.5 Exercises

1. Convince yourself of the speed/size trade-offs. Write programs using each memory model that toggle a pin as fast as possible. Use a oscilloscope to measure the rate at which this pin is toggled. You should see that it is fastest with the cog model and slowest with the CMM. Look at the size of each of these programs to see which is the largest and which is the smallest.

CHAPTER 6

COUNTERS

CHAPTER 7

PROJECT ARCHITECTURE

CHAPTER 8

WHERE TO GO FROM HERE