PROPELLER MICROCONTROLLER

PROPELLER MICROCONTROLLER
Parallel Microcontroller Programming

Nikita Ermoshkin

CONTENTS

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, understand basic binary logic and arithmetic, and have possibly used a microcontroller before. This book is written such that those new to microcontrollers and those with years of experience can benefit and learn about Propeller.

The end of each chapter will have a series of exercises to help review the topics covered in each section. Some examples and exercises will require some extra hardware to be built. This are all very simple circuits build mostly out of resistors, capacitors, or potentiometers. Appendix **??** contains a complete Bill of Materials necessary to build all the examples at once, but parts can often be reused between examples.

In addition to examples and exercises, some chapters will be dedicated to walking through larger projects.

I.1 Notation

This book makes extensive use various forms of representing numbers in digital systems. Specifically, different prefixes will represent numbers with different bases, summarized below:

- No prefix implies a base-10 (decimal) number
- The 0x prefix implies a base-16 (hexadecimal) number
- The 0b prefix implies a base-2 (binary) number

Logical and bitwise operators will follow the operators found in C, summarized in Table I.1.

Operator	Symbol
NOT	!
AND	&
OR	||
EQUAL	==
NOT EQUAL	!=

Table I.1 Summary of Logical Operators

Memory addresses will always be expressed in hexadecimal unless otherwise noted. Each memory address corresponds to the byte address, but all read/write operations are 32-bit, so 4-byte words (ints) are read at a time.

I.2 A Note on Parallax's Simple Tools Library

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, we won't be getting too in the weeds too early.

CHAPTER 1

HELLO WORLD

Hello World is the start of every programmer's journey. Before diving into any details, the first thing every reader should do is set up a development environment and run a simple example program to make sure everything function. This chapter is dedicated to that and will provide the first look into how a Propeller program is structured.

1.1 Choice of Environment and Project Setup

The first step in setting up a development environment is choosing an IDE or choosing to simply use a command line and a text editor to build project. This book encourages the latter, as it is a more portable approach and avoids issues created by IDEs such a issues with file references, IDE bugs, configuration problems, etc. This book will focus on a simple text editor and Make files to build projects. The build tools provided with this book require either macOS (OS X) or Linux. The simplest option for Windows users is to install VirtualBox and your favorite Linux distribution, such as Ubuntu or Fedora.

To install the compiler and loader tools onto the system, run the setup script provided. [provide details once script exists] Advanced users can modify it to change the install location or build the compiler from scratch.

1.2 Blinking an LED

Every introduction to microcontrollers begins with a blinking LED program, so this book will too. Open Example 1.1. This program will blink an LED on P16 at 1Hz forever.

1.2.1 Compiling and Running on a QuickStart Board

All Propeller programs are loaded over UART, a dated but extremely useful communication protocol. Due to UART's age, most computers do not have serial ports anymore, so companies like FTDI make USB-UART bridges that allow operating systems to send and receive data over a serial port that is established through USB. The QuickStart board has all of this circuitry already, so all you need is a single USB cable to interface with the Propeller on the board. Before we can actually load the example program, we need to identify which serial port on the computer has a Propeller attached to it. In a terminal, run:

```
propeller-load -P
```

This will scan all available serial ports and return the one that has a Propeller connected. On Linux, it will be something like `/dev/ttyUSB0` and on macOS it will be something like `/dev/cu.usbserial-ABCDEFGH` (the string of letters and numbers will be different for every board). Open up the Makefile in the example, and edit the `PORT` variable to be the serial port found above. Then, in the project directory, run:

```
make
```

The code should compile and load into RAM and the P16 LED will blink!

```
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 1.1 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 GPIO 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 in this program is to set the direction of the GPIO pin to output. This is done via the `DIRA` register, which is the **DIR**ection register for Port **A**.

1.3 What is a GPIO pin?

A **G**eneral **P**urpose **I**nput/**O**utput pin is every microcontroller's simplest interface to the outside world. It can be an input by reading the state of the pin (high or low) or an output

by setting the state. A high (or 1) state means that the voltage on the pin is above some threshold (about 2V) and a low or 0 state means that the voltage is below some threshold (about 1V). In between the state is not well defined and can be either one, or jump between the two. In general, all pins will always be in one of those two states, either due to the program's instructions or due to the outside world. Whether a pin is an input or an output is determined by its direction state. A low direction means it is an input and a high direction means it will be an output. From now on, since all pins on Propeller are GPIOs, they will simply be refered to as "pins".

In the example, we want the pin to be an output so that the program sets the state of the pin and therefore the LED. This is done by modifying the DIRA register.

1.3.1 The DIRx Register

The two direction registers, DIRA and DIRB, control the direction of the Port A and Port B pins. On the P8X32A, 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. Like all registers on this chip, DIRx is a 32-bit register, with each bit corresponding to a pin in the Port. On the P8X32A, 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 register 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.

1.3.2 The OUTx Register

The two output registers, OUTA and OUTB, control the output state of the Port A and Port B 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 its state, toggling the output state 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.

1.3.3 The CNT Register

The CNT register is a 32-bit unsigned integer that increments by one every clock cycle. Some useful properties about the CNT register and about free-running counters in general:

1. Its maximum value is $2^{32} - 1$ = 0xFFFFFFFF = 4,294,967,295.
2. With an 80MHz system clock, it will overflow and reset to 0 approximately every 53.7 seconds.

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.

1.3.4 The INx Register

One more register that isn't used in this program is the **IN**put register. Like the others, there is one for Port A and one for Port B and Port B is currently unused on the P8X32A. The input register contains the state of each pin's voltage, 1 for high and 0 for low. These values are always valid, regardless of the state of the OUTA or DIRA bits. For example, to read the state of P7, the following code can be used:

```
1 //...
2 if ((INA >> 7) & 1) {
3     // P7 is high
4 } else {
5     // P7 is low
6 }
7 //...
```

In this snippet, the INA register is shifted seven places to the right, so that the lowest bit corresponds to P7. That results is AND'd with 1 to isolate the lowest bit. This is a quick and efficient way to get the value of a bit in any number.

There are other registers (16 total) built into each cog of Propeller, most of which will be discussed throughout this book.

1.4 Exercises

1. Modify Example 1.1 to blink the LED 5Hz.
2. Modify Example 1.1 to blink both the P16 and P17 LED at the same rate.
3. Modify Example 1.1 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 the point that this is a tedious task in a single-threaded system.
4. With a QuickStart board, write a program that will turn on an LED when the corresponding button pad it touched.

CHAPTER 2

THE PROPELLER ARCHITECTURE

The Propeller is an eight-core (called cogs), 32-bit microcontroller featuring 32 general purpose I/O pins, 2KB of RAM per cog, 32KB of shared hub RAM, and 32KB of shared hub ROM for built-in look-up tables. The chip operates at 3.3V and consumes less than 400mW of power, depending on how many cogs are running. Each I/O pin is capable of sourcing or sinking 40mA with total current draw from the power supply limited to 300mA. The system clock source can be the internal RC oscillator, an external crystal multiplied by the internal PLL, or an external clock source. The datasheet is extremely well written and explains the chip and its architecture very clearly. This section is dedicated to providing the most important details that every Propeller developer should know. Some topics are only briefly covered here as they have entire chapters dedicated later on.

The chip is available in three packages, a 28 pin DIP, a 44 pin QFP, and a 44 pin QFN lead-less package.

2.1 The Cog

Each core, called a cog, runs PASM (Propeller Assembly) code using a five-stage pipeline with four cycles/instruction throughput. The only exception is when an instruction requires access to main memory, which can take anywhere between eight and 23 clock cycles, depending on the state of the hub. Each cog has 2KB of RAM available, with the last 16 longs reserved for special purpose registers for memory-mapped hardware. Each of these registers is described in Table 2.1 and in detail in the sections following the table. The

descriptions here are for reference. Their uses will be described in detail throughout the chapters of this book.

Register	Address	Access	Description
PAR	0x1F0	Read Only	Parameter given to the cog at boot
CNT	0x1F1	Read Only	Free-running system counter
INA	0x1F2	Read Only	Input states for Port A
INB	0x1F3	Read Only	Input states for Port B (reserved)
OUTA	0x1F4	Read/Write	Output states for Port A
OUTB	0x1F5	Read/Write	Output states for Port B (reserved)
DIRA	0x1F6	Read/Write	Direction states for Port A
DIRB	0x1F7	Read/Write	Direction states for Port B (reserved)
CTRA	0x1F8	Read/Write	Counter A control
CTRB	0x1F9	Read/Write	Counter B control
FRQA	0x1FA	Read/Write	Counter A frequency
FRQB	0x1FB	Read/Write	Counter B frequency
PHSA	0x1FC	Read/Write	Counter A phase
PHSB	0x1FD	Read/Write	Counter B phase
VCFG	0x1FE	Read/Write	Video generator configuration
VSCL	0x1FF	Read/Write	Video generator scale

Table 2.1 Cog Registers

PAR When a new cog is started, the cog starting it can set the new cog's PAR register. This is most often a memory address of some block of main memory that the cog will use as a stack. These details will be discussed later, in Chapter 3.

CNT The CNT register is teh value of a global, 32-bit, free-running counter. It increments at the frequency of the system clock, typically 80MHz. Its value is the same across all cogs and is used for timing and synchronizing events.

INx The INA and INB registers describe the input states of each pin. Each bit of INA corresponds to P0-P31. A set bit means the pin is high and a cleared bit means the pin is low. INB is reserved for future use and currently has no effects.

OUTx The OUTA and OUTB registers describe the set output state of each pin. Each bit of OUTA corresponds to P0-P31. A set bit means the pin will be driven high and a cleared bit means the pin will be driven low. This is true only if the corresponding DIRA bit is also set. OUTB is reserved for future use and currently has no effects

DIRx The DIRA and DIRB registers describe the set direction state of each pin. Each bit of DIRA corresponds to P0-P31. A set but means the pin will be an output and a cleared bit means the pin will be an input. DIRB is reserved for future use and currently has no effects

CTRx The CTRA and CTRB registers describe the configuration of Counter A and Counter B, respectively. Section 2.4 and Chapter 8 describe counters and their modes of operation in detail.

FRQx The FRQA and FRQB registers describe the accumulation amount for each respective counter. Section 2.4 and Chapter 8 describe counters and their modes of operation in detail.

PHSx The PHSA and PHSB register hold each respective counter's accumulated value. Section 2.4 and Chapter 8 describe counters and their modes of operation in detail.

VCFG The VCFG register is one of two video generator configuration registers. The other is VSCL.

VSCL The VSCL register is one of two video generator configuration registers. The other is VCFG.

There are also two registers at the beginning of the main memory: CLKFREQ at address 0x0000 and CLK at 0x0004. The CLK register configures the system clock source, the PLL, and a chip reset. The CLKFREQ register contains the resulting system clock frequency configured by CLK. This value must be set by the user and it not set by the system. These registers are typically set by the board configuration file during the program load step, but can be set by the program to change power consumption at runtime.

2.2 The Hub and Main Memory

The hub is an access system that gives each cog access to the main memory, in a round-robin fashion. The main memory is 64KB total, 32KB of RAM and 32KB of ROM. RAM is used to store the program and its dynamically allocated variables. ROM is used to store Propeller's character and math look-up tables, boot loader, and Spin interpreter. Note that there is no non-volatile flash or EEPROM built into the chip, so programs must be loaded during boot from either a host computer or an I2C EEPROM connected to P28 and P29, as described in the Propeller datasheet and later in this section.

2.3 Input and Output

The P8X32A has 32 GPIO pins, enumerated P0 through P31. Each of the 32 bits of the Port A I/O registers corresponds to an I/O pin. Port B does not exist on this chip, but the register names and memory mappings are reserved for future use. Each cog has its own I/O registers (INx, OUTx, and DIRx), but there is only one set of pins corresponding to these registers. To avoid conflict between cogs setting different states, the direction state of a pin P is given by OR'ing together the Pth bit of every cog's DIRA register. This means that if any cog sets the pin to be an output, it will be an output. If the pin is set to output, the output state will be the result of OR'ing together the Pth bit of every cog's OUTA register and output state of the counters and the video generator and then AND'ing that with the pin's direction state. This means that if any cog's hardware sets the pin to output high, the pin will output high.

2.4 Counters

Each cog has two counters that can independently be configured to one of 32 modes of operation (including a disabled mode). These include clock generation, PWM, edge detection, and logic operations, operating on two pins for input or output. In general, every counter mode accumulates the value in its frequency register to its phase register under some condition.

Table 2.2 shows the configuration register's (CTRx) bit mapping.

Bit Number	31	30-26	25-23	22-15	14-9	8-6	5-0
Description	-	Mode	PLL divider	-	Pin B	-	Pin A

Table 2.2 CTRx Bit Map

Mode There are 32 different modes of operation each counter can use. The 5-bit number in the Mode field sets the counter to operate in one of these 32 modes.

PLL divider When using PLL mode, the output of the PLL can be divided down by different factors set by the PLL divider field

Pin B For modes that use it, pin B will be the pin described by the logical pin number in this field.

Pin A For modes that use it, pin A will be the pin described by the logical pin number in this field

2.5 Video Generator

Each cog also features a video generator capable of VGA or Composite (NTSC or PAL) video generation. This can be very useful for larger projects that require a large display (such as a computer monitor) that accepts VGA input. Through some clever configuration, the video generator hardware can also be used for many other tasks, such as PWM generation (when counters aren't enough) and parallel communication protocols. This book won't discuss Composite video generation as it is pretty dated and easily replaced by VGA. However, the Propeller datasheet has a full explanation of how to use the video generator to create composite video output.

2.6 Locks

There are many applications where two or more cogs will want access to the same block of memory or function. It will become possible for data or process corruption if multiple cogs try to access the same resource. For example, let's say there is a function that will output a stream of data on a pin and multiple cogs try to call this function at the same time. If this happens, each cog will be writing its own DIRA and OUTA registers, so one cog might try to set the pin low and the other will try to set the pin high, resulting in the pin being high. As a result, the bit stream will be corrupted and not transmit the data properly.

The way to solve this is to allow this function to introduce some kind of flag, known as a lock, that doesn't let function proceed another cog has already called the function and it has not completed. Propeller has eight of these hardware-based locks that can be checked out by cogs to help coordinate access to mutually exclusive resources.

2.7 Boot-up Procedure

After power-up or reset, Propeller performs the following steps to begin operation:

1. The internal RC oscillator starts and Cog 0 begins executing the boot loader stored in ROM.
2. The boot loader looks for communication from a host computer. If found, it will communicate with the host computer to load a program into RAM and optionally, the EEPROM.
3. If no communication is found, the boot loader will attempt to read a program from the external EEPROM into RAM.
4. If a program is successfully loaded, Cog 0 is reset to execute the program.
5. If no program is loaded, Cog 0 stops and the chip enters shutdown mode, which can be exited by either power-cycle or reset, which will restart this boot procedure.

2.8 A Typical Propeller Board

A typical Propeller board will contain the following:

1. P8X32A Propeller Microcontroller
2. 5MHz or 6.25MHz crystal to generate a 80MHz or 100MHz system clock
3. 3KB or 64KB EEPROM
4. USB-UART bridge (for program loading and serial interfacing)

Propeller can generate its own system clock using an internal RC, which can be used when low power is necessary and accurate timing is not, but is most commonly used with a 5MHz crystal oscillator, which is multiplied by the internal PLL to 80MHz to generate the system clock. High-power Propeller boards will use a 6.25MHz crystal to boost this up to 100MHz. An active oscillator can also be used to provide a precise clock as desired. As mentioned earlier, an EEPROM is used to store programs that can be reloaded in the event of a power cycle or reset. This EEPROM must be at least 32KB in size, but often a 64KB EEPROM is installed and the upper 32KB are used as non-volatile storage for static data. Finally the programming circuitry (which can also be used as a UART interface) allows a host computer to program the chip's RAM or external EEPROM over UART to load programs at boot and communicate with the host during runtime.

This book will focus on using the P8X32A QuickStart, shown in in Figure 2.1. It features Propeller, a 64KB EEPROM, a USB-UART bridge for serial interfacing and program loading, eight LEDs connecting to P16-P23, eight touch pads connected to P0-P7, and all

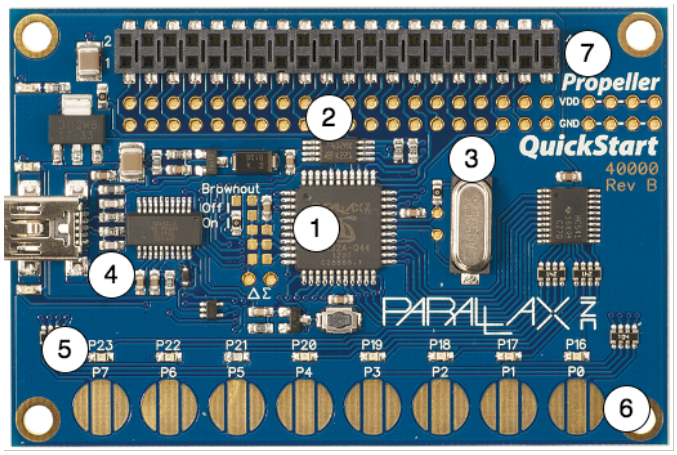

Figure 2.1 P8X32A QuickStart with Propeller (1), EEPROM (2), 5MHz crystal (3), USB-UART circuit (4), LEDs (5), touch pads (6), and GPIO breakout (7). Image taken from the QuickStart Product Page on parallax.com

pins broken out on a 2x20 header. The board is USB powered, so it very handy for quick development and testing when not a lot of extra hardware is needed.

It is often useful to disassemble the ELF file and examine the PASM code that was generated, both for debugging and optimization purposes.

2.9 Exercises

1.

2. Given that the hub gives cogs access to main memory in a round-robin fashion, explain why the best case to access main memory is eight clock cycles and the worst case is 23.

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 to allocate space for 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 longs 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.

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, using the cog's PAR register. 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 must do two things: initialize three instances of the LED mailbox and start the 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. If time must ever be expressed in seconds, simply divide the time in clock ticks by the clock frequency, CLKFREQ 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 run in the background.

[Add more about the cog]

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

4.1 Serial Printing (Hello World for Real)

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)

4.1.1 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).

4.2 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.

4.2.1 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.

4.2.2 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).

4.3 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 lies 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 Memory Model

The Cog Memory Model is the simplest to understand. The compiled PASM code simply 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 (512 longs), and some of that is reserved by the chip for the built-in registers. This means that the maximum size of the program with this memory model is only 496 instructions. 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 very 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. This is discussed later in Section 5.5.

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 main 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 23 clock cycles (refer to Chapter 2 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 format, it will break down the sizes of different

sections of the program. This can be useful for catching large memory usage and determining what causes it.

Table 5.1 compares Example 1.1 program sizes when compiled using different memory models. It's clear the difference in sizes between COG and hub-based memory models.

Memory Model	Size [Bytes]
COG	180
LMM	2624
CMM	2484

Table 5.1 Program Size Comparison

The size difference between LMM and CMM is much more noticeable for large programs. Project [Example Project] is X bytes when compiled in LMM, but only Y bytes when compiled in CMM.

5.5 Mixed-Model Programs

In many projects, the main program is written in CMM or LMM, with supporting small COG programs to perform fast functions, such as servo control. This can be accomplished by compiling multiple programs using the different models and linking them together. Then, the main program starts the subprograms in different cogs.

A COG program is defined just like a C program, shown below:

```
1  #include <propeller.h>
2  #include "pwm.h"
3
4  _NAKED int main(struct pwm_mailbox **ppmailbox) {
5
6      struct pwm_mailbox *par = *ppmailbox;
7      //...
8
9      while(1) {
10     //...
11     }
12 }
```

Example 5.1 COG Program

The main function has a `_NAKED` attribute (this helps with performance) and a parameter. Recall that when a cog is started, its PAR register can be set by the starting cog. Unless the COG program does not use a stack, this PAR register must contain a pointer to the start of a stack defined in main memory. The top of the (empty) stack can contain a pointer to a mailbox to the COG program. This means the PAR register will be a double pointer of the mailbox type. Then the first thing that needs to be done is to deference the double pointer and copy the result to a local variable. See Figure 5.1 for a visual explanation of this process.

5.5.1 Mailboxes

Mailboxes are a tool that can be used for cogs to communicate with each other. A struct, such as the one below, is defined as the mailbox type. An instance of this mailbox is

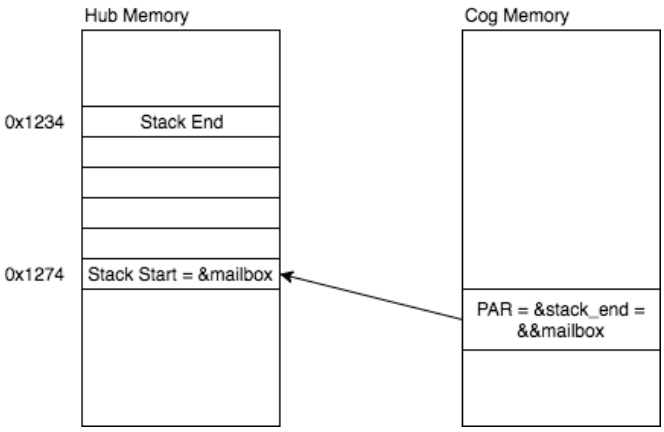

Figure 5.1 PAR Register Pointer Setup

declared and both cogs have a pointer to that instance (using the process started early), so they can read and write data. It is often important to use semaphores (discussed later) for writing multiple ints of data.

```
1 struct pwm_mailbox {
2     uint8_t pin;
3     uint32_t duty_cycle;
4     uint32_t freq;
5 };
```

The COG program can then do whatever its purpose in life is, communicating with the rest of the program through the mailbox. The main program can setup and start the COG program using the following method:

```
1  extern unsigned int _load_start_pwm_cog[];
2  static unsigned stack_pwm[PWM_STACK_SIZE + 1];
3  volatile struct pwm_mailbox pwm_par;
4
5  //...
6
7  void main() {
8  //...
9      stack_pwm[PWM_STACK_SIZE] = (unsigned)&pwm_par;
10
11     cognew(_load_start_pwm_cog, &stack_pwm[PWM_STACK_SIZE]);
12 //...
13 }
```

`_load_start_pwm_cog` is a symbol defined by the compiler when compiling and linking a COG program. This is a pointer to the block of hub memory containing the COG program, which will be copied, with startup code, to the cog.

A stack is defined for the COG program to use. It's size will depend on how much the stack is used by the COG program. The stack is used when functions are called and when variables are defined in hub memory. 64 ints is typically a good starting point.

Lastly, a mailbox is defined. It must be defined as `volatile` so that the compiler's optimizer doesn't try to remove it when it sees it has no effect on the main program, as it doesn't know that the COG program exists. Recall that stacks work by moving downwards

through memory, so the first position in the stack is the end of the array. The first stack entry is set to a pointer to the mailbox.

We start the COG program by calling `cognew()`. The first parameter is the pointer to the program, defined earlier. The second is a pointer to the start of the stack (the end of the stack's array). This is also a double pointer to the mailbox, as described earlier.

5.5.2 Compilation Differences

The COG program object is compiled slightly differently, as shown below. Note that this will look different if compiling using a Makefile. The details of program compilation are discussed in Chapter 6.

```
propeller-elf-gcc -r -mcog -o pwm.cog pwm.c
propeller-elf-objcopy --localize-text --rename-section .text=pwm.cog pwm
    .cog
```

Note that the name of the object, in this case `pwm.cog`, will determine the name of the pointer variable generated by the compiler. The name of the pointer symbol will be `_load_start_[name]_cog`, where `[name]` is the file name of the object, dropping the extension. The .cog object is linked into the main program as before.

5.6 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

PROJECT ARCHITECTURE

[Note, all the info in this chapter is how I compile most of my projects. there may easily be some redundancies and improper use of GCC or explanations of compiler options. I'm still working on experimenting and figuring out exactly how applications must be compiled, but it works by doing what is described here]

Now is a good time to explain how a large Propeller project can be set up to be built. This chapter will describe the details of each step of the build process and how to piece these steps together into a complete Propeller application

6.1 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

6.1.1 The PropGCC Compiler

When working with C, you have most likely used GCC to compile code and create executable. PropGCC is a port of GCC designed to compile C into PASM code. In general, each C source file is compiled using the following command:

```
propeller-elf-gcc -c -o out.o -mcmm main.c
```

This will compile main.c into an object file called out.o with using the CMM. This is repeated for every C source file in the application.

After all objects have been compiled, they are linked together (along with any libraries) into the actual executable application. This is done in the following way:

```
propeller-elf-gcc -o out.elf -mcmm out.o pwm.o -li2c
```

This will take the out.o file generated in the previous step, and along with pwm.o and a library named i2c, link and create the ELF executable.

Often, the code created is not very efficient in speed or space. The GCC compiler provides options that will reduce code size and speed up execution very well. For both speed and size performance, the following flags are often also used (especially when compiling in C++ mode, discussed in Chapter **??**. Read the GCC documentation for a detailed explanation of how each of these works.

- `-Os`
- `-Wl,--gc-sections`
- `-s`
- `-static`

The first two are perhaps the most important. `-Os` tells the compiler to optimize the code by making the compiled object as small as it can be. It will also perform all speed optimizations that don't significantly increase object size. `-Wl,--gc-sections` tells the linker to remove compiled code that is never used or reference, or otherwise has no effect on the system. This is especially useful when linking large libraries where only small portions are used. The other options perform other small tasks that remove unneeded tables and dynamically linked libraries.

After the generation of the ELF executable, the application must be loaded into Propeller's RAM and optionally, it's EEPROM. This is done with the following step:

```
propeller-load -Dreset=dtr -S5 -t -b QUICKSTART -p /dev/ttyUSB0 -r -v
    out.elf
```

This will load out.elf onto the board. The important flags here are:

- `-t`: Enter terminal mode after loading. This will open the load program's terminal to quickly interface with the application over serial.
- `-b`: This specifies the type of board that is being loaded to. Specifically, this is to specify the clock configuration and any other options that should be set at boot.
- `-p`: This specifies the serial port the board is connected to. If not specified, the loader will scan all available ports and determine if any have a Propeller connected.
- `-e`: This can be optionally specified to also load to program into EEPROM.

6.1.2 Other Important GCC Flags

Below is a short list of other flags that will likely be used in most (if not all) projects.

- `-I[DIRECTORY/]`: Add DIRECTORY/ to the compiler's search path for included header files. Ex: `-I/opt/propellerlibs/include/`.
- `-L[DIRECTORY/]`: Add DIRECTORY/ to the linker's search path for static libraries. Ex: `-I/opt/propellerlibs/lib/`.
- `-l[name]`: Include library called libname in the linking process. Ex: `-li2c`.

Typically, we don't want to manually run these steps for every C source file because that will make testing every code change very tedious. So, we turn to Make to automate this process. Example 6.1 is a full blank project that can be used as a template for almost any project. The Makefile for this project is shown below, with comments.

This is a placeholder, the real one comes soon.

```
1  PROPGCC = /opt/parallax
2  CC = $(PROPGCC)/bin/propeller-elf-gcc
3  LOAD = $(PROPGCC)/bin/propeller-load
4
5  MMODEL = cog
6  BOARD = QUICKSTART
7
8  TARGET = scratch_pad
9
10 PORT = /dev/cu.usbserial-*
11 OBJDIR = ../build/$(TARGET)
12
13 LOAD_MODE = load
14
15 INCLUDES =
16 LFLAGS =
17
18 CFLAGS = -Os -n -m32bit-doubles -mfcache -fno-exceptions -std=c99
19 LIBS =
20
21 LOADFLAGS = -Dreset=dtr -S50 -t
22
23 .PHONY: all setup load load_eeprom clean
24
25 all: setup $(OBJDIR)/$(TARGET).elf $(LOAD_MODE)
26
27 setup:
28     $(shell pkill propeller-load)
29     mkdir -p $(OBJDIR)
30
31 $(OBJDIR)/$(TARGET).elf: $(OBJDIR)/$(TARGET).o
32         $(CC) $(INCLUDES) $(LFLAGS) -o $@ -m$(MMODEL) $(CFLAGS) $^ $(
               LIBS)
33         $(PROPGCC)/bin/propeller-elf-size $(OBJDIR)/$(TARGET).elf
34
35 $(OBJDIR)/$(TARGET).o: main.c
36         $(CC) -c $(INCLUDES) -o $@ -m$(MMODEL) $(CFLAGS) $(DEADCODESTRIP
               ) $<
37
38 load: $(OBJDIR)/$(TARGET).elf
39         $(LOAD) $(LOADFLAGS) -b $(BOARD) -p $(PORT) -I $(PROPGCC)/
               propeller-load -r -v $<
```

```
40
41 load_eeprom: $(OBJDIR)/$(TARGET).elf
42         $(LOAD) $(LOADFLAGS) -b $(BOARD) -p $(PORT) -I $(PROPGCC)/
               propeller-load -e -r -v $<
43
44 clean:
45     rm -rf $(OBJDIR)/*
```

Example 6.1 Sample Project Makefile

At this point, you are prepared to create simple, yet powerful, applications using Propeller. The next few chapters will cover more specific topics, such as motors, counters, video generation, and communication protocols.

CHAPTER 7

PROJECT: WIRELESS COMMUNICATION WITH XBEES

Now that all of the major hardware of Propeller has been covered, this chapter will guide you through developing a simple wireless communication system using XBee wireless modules.

CHAPTER 8

COUNTERS

Each cog has two hardware counters. Counters are circuits that, well, count. More specifically, they will accumulate values to a register under some condition or drive pins high or low under some condition. Recall that there are three registers that control each counter, CTRx, FRQx, and PHSx, where x is A or B, depending on the counter to control. In one sentence, CTRx configures the counter to add the value in FRQx to PHSx every clock cycle under some condition.

Please read Section 8.3 for a detailed explanation of PWM and duty cycles, if you are unfamiliar with the topic.

8.1 CTRx

The CTRx register configures the counter's different options. These are the mode of operation, the PLL divider (if in PLL mode), pin B, and pin A. Repeated below is the bit mapping of the CTRx register from Chapter 2.

Bit Number	31	30-26	25-23	22-15	14-9	8-6	5-0
Description	-	Mode	PLL divider	-	Pin B	-	Pin A

Table 8.1 CTRx Bit Map

Mode There are 32 different modes of operation each counter can use. The 5-bit number in the Mode field sets the counter to operate in one of these 32 modes. Table 8.2 shows these different modes, condition for accumulation, and output states.

PLL divider When using PLL mode, the output of the PLL can be divided down by different factors set by the PLL divider field.

Pin B For modes that use it, pin B will be the pin described by the logical pin number in this field.

Pin B For modes that use it, pin A will be the pin described by the logical pin number in this field.

8.1.1 NCO Modes

[Add some figured to illustrate how NCO mode works, especially for non 50% duty cycle]

The simplest counter mode to is the **N**umerically **C**ontrolled **O**scillator mode. In this mode, the FRQx register is always added to the PHSx register. Pin A will be equal to the most significant bit of the PHSx register. The result is a square wave with 50% duty cycle with its frequency determined by the value of FRQx. The general formula for frequency is given by Equation 8.1. In differential mode, pin B will output the inverted pin A signal.

$$f = \frac{\text{FRQx}}{2^{32}} \cdot \text{CLKFREQ} \tag{8.1}$$

It is also possible to generate waves of constant frequency and any duty cycle using NCO mode, which is extermely useful for any PWM application. This is accomplished by creating a loop that runs at the desired frequency and at the start of every loop iteration, setting the PHSx register to the negative of the desired "ON" time, in clock ticks. Call this desired time t. By setting PHSx to a negative number, the MSB becomes a 1, setting the output high immediately. After t clock ticks have passed, the PHSx value will overflow to 0, setting the output low. The loop can be used to set up the next iteration, perform tasks that fit within the loop period, or simply wait until the next iteration. Example 8.1 demonstrates using a counter in NCO mode to slowly pulse an LED.

```
1  #include <propeller.h>
2  #include <stdio.h>
3
4  #define p_LED 16
5  #define NCO_SINGLE (0x04 << 26)
6  #define LOOP_T (CLKFREQ/1000)
7
8  void main() {
9      DIRA |= 1 << p_LED;
10
11     PHSA = 0;
12     FRQA = 1;
13     CTRA = NCO_SINGLE | p_LED;
14
15     uint32_t t = CNT;
16     uint32_t duty = 0;
17     int32_t sign = 1;
18     while(1) {
19
20         PHSA = -duty*(LOOP_T/1000);
```

Mode	Description	Accumulation Condition	Pin A State	Pin B State
0x00	Disabled	0 (never)	0	0
0x01	PLL internal	1 (always)	0	0
0x02	PLL single-ended	1	PLLx	0
0x03	PLL differential	1	PLLx	!PLLx
0x04	NCO single-ended	1	PHSx[31]	0
0x05	NCO differential	1	PHSx[31]	!PHSx[31]
0x06	Duty cycle single-ended	1	PHSx Carry	0
0x07	Duty cycle differential	1	PHSx Carry	!PHSx Carry
0x08	Positive detector	A (A is high)	0	0
0x09	Positive detector with feedback	A	0	!A
0x0A	Rising edge detector	A & A$(n-1)$	0	0
0x0B	Rising edge detector with feedback	A & A$(n-1)$	0	!A
0x0C	Negative detector	!A	0	0
0x0D	Negative detector with feedback	!A	0	0
0x0E	Falling edge detector	!A & A$(n-1)$	0	0
0x0F	Falling edge detector with feedback	!A & A$(n-1)$	0	0
0x10	Logic: 0	0	0	0
0x11	Logic: !A & !B	!A & !B	0	0
0x12	Logic: A & !B	A & !B	0	0
0x13	Logic: !B	!B	0	0
0x14	Logic: !A & B	!A & B	0	0
0x15	Logic: !A	!A	0	0
0x16	Logic: A != B	A != B	0	0
0x17	Logic: !A ‖ !B	!A ‖ !B	0	0
0x18	Logic: A & B	A & B	0	0
0x19	Logic: A == B	A == B	0	0
0x1A	Logic: A	A	0	0
0x1B	Logic: A ‖ !B	A ‖ !B	0	0
0x1C	Logic: B	B	0	0
0x1D	Logic: !A ‖ B	!A ‖ B	0	0
0x1E	Logic: A ‖ B	A ‖ B	0	0
0x1F	Logic: 1	1	0	0

Table 8.2 Counter Modes. Note: A$(n-1)$ refers to a measurement one clock cycle before the current measurement

```
21
22          duty += sign;
23          if (duty == 1000 || duty == 0) {
24              sign *= -1;
25          }
```

```
26
27          waitcnt(t += LOOP_T);
28      }
29  }
```

Example 8.1 Counter using NCO Mode

In this example, we write 0x04 to the 26th bit (the mode field) of CTRA and a 16 to the pin A field. We write a 1 to FRQA, which gives us the highest duty cycle resolution. Within the loop, PHSA is set to the negative of the computed "ON" time, and the time for the next iteration is computed. Finally, a call to `waitcnt()` pauses execution until the desired period of time has passed since the last loop iteration. Chapter **??** will go into the details of creating accurately timed loops.

8.1.2 PLL Modes

The PLL mode is similar to NCO mode, except instead of setting pin A to be equal to the MSB of PHSx, the clock signal is sent to a PLL, which can be configured to eight different multipliers by the PLL divider field of CTRx. This allows Propeller to generate clocks in the range of 500 kHz to 128 MHz. Frequencies higher than 128 MHz are outside of the stable range of the PLL and are not recommended.

There are three PLL modes: internal, which is connected to the video generator; single-ended, which outputs the PLL on pin A, and differential, which outputs the PLL on pin A and the inverse of the clock on pin B.

8.1.3 Duty Cycle Modes

Duty Cycle modes operate in a similar fashion to NCO modes, except instead of setting pin A to the MSB of PHSx, pin A is set to the carry bit of the PHSx adder. This means that pin A will be high whenever adding FRQx to PHSx causes an overflow. When taking a time average of the output waveform, the duty cycle will be defined by Equation 8.2.

$$\mathrm{DC}[\%] = 100\% \cdot \frac{\mathrm{FRQA}}{2^{32}} \tag{8.2}$$

[Add a figure showing different wave forms]

This is extremely useful when building a simple Digital to Analog Converter (DAC). Figure **??** shows the simple circuit necessary to filter the generated waveform into an analog voltage. This method of DA conversion is known as Delta-modulation, often written as Δ-modulation. Section 8.2 will provide a detailed DAC example.

8.1.4 State and Edge Detection Modes

The pin state and edge detection modes are extremely useful for counting and timing events. For stability, the value of pin A is buffered, so it will always be one clock cycle behind. Positive detection means the accumulation occurs whenever pin A is high, negative means accumulation occurs whenever pin A is low. This is useful for measuring pulse widths and event lengths. Similarly, the edge detectors accumulate FRQx to PHSx whenever a rising or falling edge is detected, depending on the counter configuration.

Each mode also has a feedback option, which will output on pin B the inverse of pin A. One application of this is a Sigma-Delta Analog to Digital Converter ($\Sigma\Delta$ ADC). A $\Sigma\Delta$

ADC allows Propeller to measure analog voltages using purely digital pins. Section 8.2 provides an example and explanation of implementing a $\Sigma\Delta$ ADC on Propeller.

8.1.5 Logic Modes

The last set of modes are the logic modes. Each mode has an associated logic equation of pin A and pin B that must be satisfied for accumulation to occur. One application of this mode of operation is a long duration event timer, an example of which is provided in Section 8.2.

8.2 Applications of Counters

8.2.1 RC Servo Control

8.2.2 Δ Modulation DAC

8.2.3 $\Sigma\Delta$ ADC

[This is better explanation of app note: https://www.parallax.com/sites/default/files/downloads/AN008-SigmaDeltaADC-v1.0.pdf]

A $\Sigma\Delta$ ADC operates by using a 1-bit DAC (a comparator) to sample and integrate an analog signal at a very high frequency. This method of oversampling allows a digital circuit to read an analog voltage. Figure **??** shows the block diagram of a theoretical $\Sigma\Delta$ ADC. The general principle of operating is that we can use a 1-bit ADC and a 1-bit DAC in feedback to measure an analog voltage.

[insert figure]

This example won't go into the details of how $\Sigma\Delta$ works and why this is a $\Sigma\Delta$ ADC. However, it will try to motivate that the end result is the same as a $\Sigma\Delta$ ADC. Now take a look at Figure **??**. Similar to how an inverting operational amplifier functions,

8.3 Pulse Trains and Duty Cycles

CHAPTER 9

PROJECT: CLOSED-LOOP STEPPER MOTOR CONTROL

Now that all of the major hardware of Propeller has been covered, this chapter will guide you through developing a closed-loop stepper motor controller.

CHAPTER 10

COMMUNICATION PROTOCOLS

This chapter will cover I2C, SPI, and UART in detail.

CHAPTER 11

LCDS AND DISPLAYS

This chapter will using character LCDs, I2C OLED displays, and a VGA monitor to disconnect data feedback from a computer.

CHAPTER 12

TIPS AND TRICKS

This chapter is dedicated to describing some common code structures and other tricks used in many Propeller programs to improve performance or take advantage of Propeller's unique architecture.

12.1 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 exactly a half second period (not counting the 8 clock cycles it takes to return to the top of the loop and save the CNT register again).

12.2 Shared Resources and Mutual Exclusion

There are a number of shared resources that each cog can have access to. These include the I/O pins, main memory, system counter, clock configuration, and locks. I/O pins and the system counter are known as Common Resources—all cogs can access them at the same time. All other shared resources are known as Mutually Exclusive Resources—they must be accessed through the hub. Recall that the hub is the mechanism that gives each cog access to Mutually Exclusive Resources one at a time in a round-robin fashion. This guarantees that when reading or writing data in elementary units (bytes or words), the entire piece of data will be read and no other cog can read or write that piece of data. However, problems can arise when trying to read or write blocks of data with different cogs. Cogs must coordinate access to that block of data to maintain mutual exclusion.

The Propeller hardware provides eight locks that can be used to help cogs coordinate access to a resource. [Write an example showing two cogs misbehaving and then how locks fix it]

CHAPTER 13

WHERE TO GO FROM HERE