# Lab 2

**Parallax Propeller**

Robert Wortman III  
CpE 185, Section 2 | Monday  
Dennis Dahlquist  
November 7, 2011

## Contents

1. **Introduction**  
2. **Part 1: Introduction to the Propeller Microcontroller**  
   2.1 Data  
   2.2 Summary  
3. **Part 2: Propeller I/O and Methods**  
   3.1 Data: Lab 4  
   3.1.1 Questions  
   3.1.2 Exercises  
   3.1.3 Project 1  
   3.1.4 Project 2  
   3.1.5 Project 3  
   3.1.6 Project 4  
   3.1.7 Project 5  
   3.2 Data: Lab 5  
   3.2.1 Questions  
   3.2.2 Exercises  
   3.2.3 Project 1  
   3.2.4 Project 2  
   3.2.5 Project 3  
   3.3 Summary
4 Part 3: Propeller Objects and Counters

4.1 Data: Lab 6 ............................................................................. 17
4.1.1 Questions ........................................................................... 19
4.1.2 Exercises ........................................................................... 21
4.1.3 Project 1 ............................................................................. 21
4.1.4 Project 2 ............................................................................. 23
4.1.5 Project 3 ............................................................................. 24

4.2 Data: Lab 7 ............................................................................. 25
4.2.1 Questions ........................................................................... 26
4.2.2 Project 1 ............................................................................. 28
4.2.3 Project 2 ............................................................................. 29

5 Conclusion .............................................................................. 30

Listings

1 Test program for the Propeller Education Kit .............................. 2
2 Do Nothing; place Propeller in safe state ...................................... 2
3 Structural Constructs in Spin ......................................................... 3
4 ButtonToLed.spin – Led mirrors pushbutton state ......................... 3
5 ShiftRightWithInput.spin ............................................................ 5
6 TrafficDumb.spin ....................................................................... 9
7 TrafficSmart.spin ...................................................................... 10
8 BlinkMultiple.spin ..................................................................... 11
9 SetClock.spin .......................................................................... 11
10 SetTimer.spin .......................................................................... 12
11 SquareWave.spin ...................................................................... 16
12 SquareWaveExp.spin ................................................................. 16
13 HelloPST.spin ......................................................................... 17
14 HelloPST.spin after corrections .................................................. 19
15 Blinker.spin modified to test stack usage .................................... 23
16 TickTock.spin .......................................................................... 24
17 DAC2Channel.spin ................................................................... 28
18 TestFourPWM.spin ................................................................... 29
1 Introduction

The purpose for this lab is to become familiar with the features of the Parallax Propeller microcontroller and with the tools used to program for it. I will study the programming model of the Spin language, to include objects and methods. I will also study the architecture of the Propeller, to include Cogs, the hub/distributor model, external interfacing, and internal components.

2 Part 1: Introduction to the Propeller Microcontroller

The purpose for this part is to construct the Propeller Education Kit platform and to verify its functioning.

2.1 Data

Listing 1: Test program for the Propeller Education Kit

```
'' File: PushbuttonLedTest.spin
'' Test program for the Propeller Education Lab "PE Platfrm SEtup"

CON

_clkmode = xtal1 + pll16x ' Feedback and PLL multiplexer
_xinfreq = 5.000.000 ' External oscillator = 5 MHz

LEDs_START = 00 ' Start of I/O pin group for signals
LEDs_END = 15 ' End of I/O pin group for signals
PUSHBUTTON = 27 ' Pushbutton Input Pin

PUB ButtonBlinkSpeed ' Main Method

'' Sends on/off (3.3 V / 0 v) signals at approximately 2 Hz.

dira[LEDs_START..LEDs_END]~~ ' Entire pin group to output
repeat ' Endless loop

! outa[LEDs_START..LEDs_END] ' Change the state of pin group

if ina[PUSHBUTTON] == 1 ' If pushbutton pressed
    waitcnt(clkfreq / 4 + cnt) ' Wait 1/4 second -> 2 Hz
else ' If pushbutton not pressed
    waitcnt(clkfreq / 20 + cnt) ' Wait 1/20 second -> 10 Hz
```

Listing 2: Do Nothing; place Propeller in safe state

```
'' File: DoNothing.spin

PUB main ' empty main
```
2.2 Summary

I got to troubleshoot my first bug during Lab 3. While testing the Propeller with PushbuttonLedTest.spin, I found that pins 17 and 27 failed to function as input. I discovered that when the selector wire was directly connected to the pin, rather than inserted into the breadboard, the pins worked fine as input. On closer inspection, I noticed that the Propeller chip was not fully inserted into the breadboard. After correcting this error, the problem disappeared. No further issues were noted.

3 Part 2: Propeller I/O and Methods

The purpose for this section is to become familiar with writing Methods in the Spin language and with utilizing the interfacing features of the Propeller microcontroller.

3.1 Data: Lab 4

Listing 3: Structural Constructs in Spin

```spin
repeat <numeric_expression>
  [block]

repeat until <boolean_expression>
  [block]

repeat while <boolean_expression>
  [block]

if <boolean_expression>
  [block]
elseif <boolean_expression>
  [block]
else
  [block]
```

Listing 4: ButtonToLed.spin – Led mirrors pushbutton state

```
' File: ButtonToLed.spin
' Led mirrors pushbutton state.

PUB ButtonLed
  ' Pushbutton/Led Method

dira[4] := 1 ' P4 -> output
dira[5] := 1 ' P5 -> output
dira[6] := 1 ' P6 -> output
dira[7] := 1 ' P7 -> output
dira[8] := 1 ' P8 -> output
dira[9] := 1 ' P9 -> output
dira[21] := 0 ' P21 -> input
```
<table>
<thead>
<tr>
<th>operator</th>
<th>operation</th>
<th>usage</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:=</td>
<td>simple assignment</td>
<td>infix</td>
<td>assignment</td>
</tr>
<tr>
<td>clear</td>
<td>postfix arithmetic &amp; assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>postfix arithmetic &amp; assignment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>binary</td>
<td>prefix</td>
<td>numeric constant</td>
</tr>
<tr>
<td>.</td>
<td>separator</td>
<td>postfix</td>
<td>numeric constant</td>
</tr>
<tr>
<td>=</td>
<td>constant declaration</td>
<td>infix</td>
<td></td>
</tr>
<tr>
<td>++</td>
<td>increment</td>
<td>pre/post</td>
<td>arithmetic &amp; assignment</td>
</tr>
<tr>
<td>-</td>
<td>decrement</td>
<td>pre/post</td>
<td>arithmetic &amp; assignment</td>
</tr>
<tr>
<td>==</td>
<td>equality</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>inequality</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>=&gt;</td>
<td>greater than or equal</td>
<td>infix</td>
<td>relational</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>//</td>
<td>modulus</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
<td>infix</td>
<td>arithmetic</td>
</tr>
<tr>
<td>+=, -=, *=, /=, //=, &lt;&lt;=, &gt;&gt;=</td>
<td>combination</td>
<td>infix</td>
<td>arithmetic &amp; assignment</td>
</tr>
<tr>
<td>and</td>
<td>and</td>
<td>infix</td>
<td>boolean</td>
</tr>
<tr>
<td>or</td>
<td>or</td>
<td>infix</td>
<td>boolean</td>
</tr>
<tr>
<td>not</td>
<td>not</td>
<td>prefix</td>
<td>boolean</td>
</tr>
</tbody>
</table>
repeat  
' endless loop 
' copy P23 to P4 and P7 
' copy P22 to P5 and P8 
' copy P21 to P6 and P9

Listing 5: ShiftRightWithInput.spin

'' File: ShiftRightWithInput.spin
'' Demonstrates the right shift operator and input from switch

PUB ShiftLedsLeft ' Or ... y'know ... right

dira[9..4]~

repeat

   waitcnt(clkfreq/10 + cnt)

   outa[9..4] >>= 1

3.1.1 Questions

1. 8 processors (p7)

2. 32 kb main RAM (p9)

3. The Propeller chip supply voltage ($V_{DD}$) is 3.3 VDC. (p 21) This is also the output voltage of an output pin set to 1. (p 48) An input pin which sees greater than 1.65 VDC will read a 1; otherwise it will read a 0. (p 50)

4. Spin code is compiled into tokens and loaded into the Propeller where it is either stored in RAM or on the EEPROM. If stored on the EEPROM, it is then loaded into RAM.

5. Spin code is compiled into tokens, whereas assembly is assembled into machine code. (p 11) An interpreter stored in ROM is loaded into RAM. It will fetch each token, interpret it, and execute it. (pp 8-9) Machine code is executed directly by the cog at 20 MIPS (p 9)

6. An object is a conceptual model comprised of an internal state, external capabilities, and external interfaces. A method is an interface to an object which can access the objects state and capabilities. (pp 46-47)

7. A top object is analogous to a main function or routine; it determines the first routine to be executed. (p 46)

8. dira determines whether each I/O pin is configured for Input or Output (0 and 1, respectively). outa determines the value output on each I/O pin, if configured for output. (p 48)
9. The **repeat** command may be given a numeric constant argument, in which case it will perform that many loops. It may be given a boolean argument, in which case, it will loop until the argument is false. And the **repeat** command may be used as a for-loop: 

```assembler
repeat <variable> from <start> to <stop> [step <increment>]
```

(pp 57, 59-60)

10. **clkfreq** returns the frequency of the system clock in Hz (p54)

11. Commands in a **repeat** block are indented (p 52)

12. The sloppy method of calculating a target for **waitcnt** is 

```assembler
(clkfreq * <period> + cnt)
```

for <period> in seconds. (p 51-52) A more precise method is to prime two variables 

```assembler
dT := clkfreq * <period> and T := cnt
```

and calculate the target with 

```assembler
(T += dT).
```

(p 64)

13. **_xinfreq** is defined to the frequency of the external oscillator. **_clkmode** is defined to various flags that configure the input frequency orange from the crystal and the output multiplier from the PLL. (p 53)

14. The PLL multiplies teh signal from the crystal. (p 53)

15. A constant value for delays will not produce the same delay if a different clock frequency is selected. (p 54)

16. The external signal will be more accurate (±0.003%) than the internal clock (+66% − 33%). (p 54)

17. The **dira** and **outa** registers determine the direction and output of the I/O pins. (p 48) For a pin configured as input, its corresponding bit in the **ina** register will vary with the input. (p 50)

18. Single bits are indicated by a single subscript (e.g. [5]). Groups of bits are indicated by a ranged subscript (e.g. [5..9]). (p 49)

19. The binary number indicator is %. (p 49)

20. The pin is configured as input; its **outa** bit does not matter. (p 48)

21. Default to 0 at startup. (p 58)

22. See Table 1, page 4

23. See Table 1, page 4

24. See Table 1, page 4

25. Comparison operators are not required for if and repeat tests. Integers decay to boolean in the conventional way. (p 63)
26. Variables can either be global (declared in a \texttt{VAR} block) or local (declared in the method declaration) (p 63-64).

27. Global variables can be declared as byte, word, or long. Local variables can be long. (p 64)

28. Local variables are declared separated from the method declaration with a pipe and from each other with commas.

3.1.2 Exercises

1. \texttt{dira[8..12] := outa[8..12]~}

   \hspace{1em} \texttt{outa[13..15]~}
   \hspace{1em} \texttt{dira[13..15]~}

3. \texttt{dira[0..8] := %111000000}

4. repeat
   \hspace{1em} \texttt{outa[9] := outa[8]}
   \hspace{1em} \texttt{!outa[8]}
   \hspace{1em} \texttt{waitcnt(clkfreq*100 + cnt)}

5. repeat
   \hspace{1em} \texttt{outa[0..7] != ina[8..15]}

6. \texttt{CON}
   \hspace{1em} \texttt{_xinfreq = 5_000_000}
   \hspace{1em} \texttt{_clkmode = xtal1 + pll2x}

7. \texttt{waitcnt(clkfreq/5+cnt)}

8. \texttt{outa[5..11]~}
   \hspace{1em} \texttt{waitcnt(clkfreq/3 + cnt)}
   \hspace{1em} \texttt{outa[6..10] ^= %10101}

9. \texttt{PUB LightsOn}
   \hspace{1em} \texttt{dira[9..4]~}
   \hspace{1em} \texttt{repeat 6}
   \hspace{1em} \hspace{1em} \texttt{outa[9..4] := outa[9..4] << 1 + 1}
   \hspace{1em} \hspace{1em} \texttt{waitcnt( clkfreq + cnt )}
   \hspace{1em} \hspace{1em} \texttt{repeat}

10. \texttt{CON}
    \hspace{1em} \texttt{PinIn = 0}
PinOut = 27
PUB TimedLight
dira[PinOut]~~
dira[PinIn]~
repeat
  if ina[PinIn]
    outa[Pinout]~~
    waitcnt(clkfreq*5+cnt)
    outa[PinOut]~

11. PUB Countdown
dira[9..4]~~
outa[9..4] := 60
repeat while outa[9..4]-- > 1
  waitcnt(clkfreq/5+cnt)
repeat

12. PUB Countdown
dira[9..4]~~
repeat
  outa[9..4] := 60
  repeat while outa[9..4]-- > 1
    waitcnt(clkfreq/5+cnt)
repeat

13. PUB PushTwoStart
dira[21]~
dira[23]~
outa[9]~
dira[9]~~
repeat until ina[21] and ina[23]
outa[9]~~
repeat

14. PUB PushTwoCountdown
dira[21]~
dira[23]~
outa[9..4]~
dira[9..4]~~
repeat until ina[21] and ina[23]
outa[9..4] := 60
repeat while outa[9..4]-- > 1
  waitcnt(clkfreq/10+cnt)
repeat
3.1.3 Project 1

Listing 6: TrafficDumb.spin

' File: TrafficDumb.spin
' This program operates a pair of traffic lights in a dumb manner.
' It simply cycles through, without regard to traffic.

CON

    TimeYellow = 1
    TimeGreen = 2

    NSRed = 4
    NSYellow = 5
    NSGreen = 6

    EWRed = 7
    EWYellow = 8
    EWGreen = 9

PUB TrafficDumb

    outa[4..9]~
    dira[4..9]~~

    outa[NSRed]~~
    outa[EWRed]~~

    repeat

        if outa[EWRed]
            outa[NSRed]~
            outa[NSGreen]~~
            waitcnt(clkfreq*TimeGreen + cnt)
            outa[NSGreen]~
            outa[NSYellow]~~
            waitcnt(clkfreq*TimeYellow + cnt)
            outa[NSYellow]~
            outa[NSRed]~~

        if outa[NSRed]
            outa[EWRed]~
            outa[EWGreen]~~
            waitcnt(clkfreq*TimeGreen + cnt)
            outa[EWGreen]~
            outa[EWYellow]~~
            waitcnt(clkfreq*TimeYellow + cnt)
            outa[EWYellow]~
            outa[EWRed]~~
3.1.4 Project 2

Listing 7: TrafficSmart.spin

' ' File: TrafficSmart.spin
' ' This program operates a pair of traffic lights in a less-dumb manner.
' ' It normally leaves the high traffic road with a green light.

CON

  TimeYellow = 1
  TimeGreen = 2

  NSRed = 4
  NSYellow = 5
  NSGreen = 6

  EWRed = 7
  EWYellow = 8
  EWGreen = 9

  Trigger = 21

PUB TrafficDumb

  outa[4..9]~
  dira[4..9]~~
  dira[Trigger]~

  outa[NSRed]~~
  outa[EWRed]~~

repeat

  if outa[EWRed]
  outa[NSRed]~
  outa[NSGreen]~~
  repeat until ina[Trigger]
  waitcnt(clkfreq*TimeGreen + cnt)
  outa[NSGreen]~
  outa[NSYellow]~~
  waitcnt(clkfreq*TimeYellow + cnt)
  outa[NSYellow]~
  outa[NSRed]~~

  if outa[NSRed]
  outa[EWRed]~
  outa[EWGreen]~~
  waitcnt(clkfreq*TimeGreen + cnt)
  outa[EWGreen]~
  outa[EWYellow]~~
  waitcnt(clkfreq*TimeYellow + cnt)

  outa[EWYellow]~
  outa[EWGreen]~~
  waitcnt(clkfreq*TimeYellow + cnt)
3.1.5 Project 3

Listing 8: BlinkMultiple.spin

'' File: BlinkMultiple.spin
'' This program blinks 6 LEDs at different frequencies

CON

_xinfreq = 5_000_000
_clkmode = xtal1 + pll16x

Freq4 = 1
Freq5 = 2
Freq6 = 3
Freq7 = 7
Freq8 = 12
Freq9 = 13

PUB BlinkMultiple
    dira[4..9]~
    repeat
        outa[8] := (cnt * 2 * Freq8 / clkfreq) // 2
    In retrospect, I realize that I should have moved most of the calculation out of the
    loop. Also, a bitwise mask to select the last bit would have been better than a modulo
    operation. Those lines should read something more like outa[X] := (cnt / DelayX) & 1
    with something like DelayX := clkfreq / 2 / FreqX preceding the loop.

3.1.6 Project 4

Listing 9: SetClock.spin

'' File: SetClock.spin
'' This program sets a clock, with repeat by button hold.

CON

_xinfreq = 5_000_000
_clkmode = xtal1 + pll16x

FreqSlow = 2
FreqFast = 12
SlowClicks = 5
VAR
  byte Count ' How many times the slow counter has incremented the clock
  byte LastButn ' The state of the buttons during the last loop
  byte ThisButn ' The state of the buttons at the start of this loop
  byte Dir ' The direction requested by the pressed button
  long LastClick ' The time of the last moment the clock was clicked

PUB SetClock
  dira[4..9]~~
dira[21..23]~
  Count := 0

  repeat
    if ina[23] and not ina[21]
      ' Up
      ThisButn := TRUE
      Dir := 1
    elseif ina[21] and not ina[23]
      ' Down
      ThisButn := TRUE
      Dir := -1
    else
      ' Neither, treating "both" as equivalent to neither
      ThisButn := FALSE
      Count := 0

    if ThisButn
      if not LastButn
        ' Fresh button press
        outa[9..4] += Dir ' adjust the clock
        LastClick := cnt ' record the time
      elseif ( Count > SlowClicks ) and ( cnt - LastClick > clkfreq / FreqFast )
        ' Held button press, long enough to go fast
        outa[9..4] += Dir ' adjust the clock
        LastClick := cnt ' record the time
      elseif cnt - LastClick > clkfreq / FreqSlow
        ' Held button press
        outa[9..4] += Dir ' adjust the clock
        LastClick := cnt ' record the time
        Count++ ' one step closer to going fast
    LastButn := ThisButn

3.1.7 Project 5

Listing 10: SetTimer.spin

' File: SetTimer.spin
' This program sets a timer, with repeat by button hold.
' When triggered, it counts down to zero.

CON
_xinfreq = 5.000_000
_clkmode = xtall + pll16x

FreqSlow = 2
FreqFast = 12
SlowClicks = 5

VAR
  byte Count ' How many times the slow counter has incremented the clock
  byte LastButn ' The state of the buttons during the last loop
  byte ThisButn ' The state of the buttons at the start of this loop
  byte Dir ' The direction requested by the pressed button
  long LastClick ' The time of the last moment the clock was clicked

PUB SetClock
  dira[4..9]~
  dira[21..23]~
  Count := 0

repeat while not ina[22]
  if ina[23] and not ina[21]
    ' Up
    ThisButn := TRUE
    Dir := 1
  elseif ina[21] and not ina[23]
    ' Down
    ThisButn := TRUE
    Dir := -1
  else
    ' Neither, treating "both" as equivalent to neither
    ThisButn := FALSE
    Count := 0

  if ThisButn
    if not LastButn
      ' Fresh button press
      outa[9..4] += Dir ' adjust the clock
      LastClick := cnt ' record the time
    elseif ( Count > SlowClicks ) and ( cnt - LastClick > clkfreq / FreqFast )
      ' Held button press, long enough to go fast
      outa[9..4] += Dir ' adjust the clock
      LastClick := cnt ' record the time
    elseif cnt - LastClick > clkfreq / FreqSlow
      ' Held button press
      outa[9..4] += Dir ' adjust the clock
      LastClick := cnt ' record the time
      Count++ ' one step closer to going fast
    LastButn := ThisButn

repeat while outa[9..4]-- > 1
3.2 Data: Lab 5

3.2.1 Questions

1. The method returns the value of the `result` variable, which may be aliased to another local variable in the method declaration. (pp 77-78)

2. A method may be passed however many parameters it was declared to accept. (pp 71-72)

3. A method returns exactly one value. (pp 77-78)

4. Within the method, its return value is determined by assigning to its `result` variable, or alternate alias. In the calling method, the called method’s return value is obtained by using the method as an rvalue in an expression. (pp 77-78)

5. The `cognew` command needs the method call to launch and a pointer to some stack space for the method to use. (pp 73-74)

6. Cog 0’s stack is the unused global RAM after the program code. Other Cogs use stack space allotted by their calling cog. (p 73)

7. `cognew` launches a method in the first available Cog, and returns the number of the Cog selected. `coginit` launches a method in the first available Cog, and doesn’t return anything.

8. A Cog will stop when its running method runs out of code to execute, or another method may call `cogstop`. (pp 74-75)

9. When a method is called, space is allocated on the stack for the return address to the calling method, the called method’s `result` variable and any other local variables, including parameters and scratch space for evaluating expressions. (p 75)

10. As a method is executed, part of the stack space is used for scratch space.

11. During nested or recursive method calls, each call uses the appropriate amount of stack space.

12. To avoid trouble with stack space, allocate far more than needed during prototyping, and trim it down later.

13. The `cognew` command returns the number of the Cog that was launched.

14. It is possible to launch successive cogs in a loop by relating the index of the cog to the stack space reserved for that cog. (pp 78-80)
3.2.2 Exercises

1. PUB SquareWave( pin, tHigh, tCycle) : success | tC, tH

2. yesNo := SquareWave( 24, clkfreq/2000, clkfreq/100 )

3. VAR
   long swStack[40]

4. VAR
   long swCog

5. swCog := cognew( SquareWave( 24, clkfreq/2000, clkfreq/100 ), @swStack[0] )

6. coginit( SquareWave( 5, 24, clkfreq/2000, clkfreq/100 ), @swStack[0] )

7. VAR
   long swStack[40 * 3]

8. VAR
   long swCog[3]

9. PUB LaunchCogs
   swCog[0] := cognew( SquareWave( 5, clkfreq/20, clkfreq/10 ), @swStack[40 * 0] )
   swCog[1] := cognew( SquareWave( 6, clkfreq/100, clkfreq/9 ), @swStack[40 * 1] )
   swCog[2] := cognew( SquareWave( 9, clkfreq/1000, clkfreq/2000 ), @swStack[40 * 2] )

3.2.3 Project 1

CON

_xinfreq = 5_000_000
_clkmode = xtal1 + pll16x

PUB SquareWave( pin, tHigh, tCycle) : success | tC, tH
dira[p]~
tC := tH := cnt
repeat
   outa[p]~
   waitcnt(tH := tC + tHigh)
   outa[p]~
   waitcnt(tC += tCycle)

3.2.4 Project 2
Listing 11: SquareWave.spin

'' File: SquareWave.spin

CON

_xinfreq = 5_000_000
_clkmode = xtal1 + pll16x
NumWaves = 3

VAR

long swCog[NumWaves]
long swStack[40 * NumWaves]

PUB Main
swCog[0] := cognew ( SquareWave (5, clkfreq/20, clkfreq/10), @swStack[40 * 0] )
swCog[1] := cognew ( SquareWave (6, clkfreq/100, clkfreq/9), @swStack[40 * 1] )
swCog[2] := cognew ( SquareWave (9, clkfreq/2000, clkfreq/1000), @swStack[40 * 2] )
repeat

PUB SquareWave ( pin , tHigh , tCycle ) : success | tC , tH , T
    dira[pin]~~
    tC := tH := cnt
    repeat
        outa[pin]~~
        waitcnt ( tH := tC + tHigh )
        outa[pin]~
        waitcnt ( tC += tCycle )

3.2.5 Project 3

Listing 12: SquareWaveExp.spin

'' File: SquareWaveExp.spin
'' Experimentation with the square wave generator
'' Makes the six LEDs slowly fade on at different speeds

CON

_xinfreq = 5_000_000
_clkmode = xtal1 + pll16x
NumWaves = 6

VAR

long swCog[NumWaves]
long swStack[40 * NumWaves]

PUB Main | Duty[NumWaves], DutySpeed[NumWaves], i
    repeat i from 0 to NumWaves-1
        Duty[i] := 0
        DutySpeed[i] := i+1
    repeat
        repeat i from 0 to NumWaves-1
            coginit ( i+1, SquareWave ( i+4, clkfreq/100000*Duty[i], clkfreq/1000 ), @swStack[40 * i] )
            swCog[i] := i+1
            Duty[i] += DutySpeed[i]
            Duty[i] //= 99
        waitcnt(clkfreq/20 + cnt)
    repeat i from 0 to NumWaves-1
        cogstop( swCog[i] )

17
```plaintext
PUB SquareWave (pin, tHigh, tCycle) : success | tC, tH, T
dira[pin]--
tC := tH := cnt
repeat
  outa[pin]--
  waitcnt (tH := tC + tHigh)
  outa[pin]--
  waitcnt (tC += tCycle)
```

3.3 Summary

In Chapter 4, I became more familiar with the Parallax Propeller timer, including configuration of the external crystal timer, and use of the `cnt` buffer and `clkfreq` function. I learned the procedural constructs for writing in Spin, and some of the operators. I became more familiar with the interfacing capabilities of the Propeller, including the `dira`, `outa`, and `ina` registers. I also learned that I have a rotten habit of causing bugs by absentmindedly specifying the wrong register for what I’m trying to do. This will be a point on which I can focus for future improvement.

In Chapter 5, I became more familiar with Parallax Propeller architecture elements used to invoke a method on a new cog, the associated memory management techniques, and the bookkeeping required to keep track of which process is running on which cog.

4 Part 3: Propeller Objects and Counters

The purpose of this part is to learn the object-oriented features of the Spin language and the use of the counter modules on the Propeller microcontroller.

4.1 Data: Lab 6

I had some difficulties with the `HelloPST.spin` exercise. I was able to load the program into the Propeller, and it linked as expected with the Parallax Serial Terminal object, but I only got garbage from the serial line.

Listing 13: HelloPST.spin

```
'' HelloPST.spin
'' Test message to Parallax Serial Terminal.

CON

  _clkmode = xtal1 + p1l16x
  _xin = 5_000_000

OBJ

  pst : "Parallax Serial Terminal"

PUB TestMessages
```
' Send test messages to Parallax Serial Terminal

```
pst.Start(115_200)

repeat
    pst.Str(string("This is a test message!"))
    Pst.NewLine
    waitcnt(clkfreq + cnt)
```

The garbage was as follows. It came in bursts with one-second pauses interspersed, as the text was expected.

```
00000000  c0 c0 ce 3e 80 3e 0e 3e 3e fc 3e f0 0e 3e 3e fc |...>.>.>>>>.>>>.
00000010  3e f0 0e f0 3e f0 c0 fe 7e 0e 7e f0 7e 70 7f 1f |>.>...>....`````````````````p...
00000020  c3 ce 3e ce f0 3e f0 7e f0 3e 3e f0 7e 0e 3e f0 |>.>...>..`````..``````````p...
00000030  ce c0 ce 3e 80 3e 0e 3e 3e fc 3e f0 0e 3e 3e fc |...>.>>>>.>>>.
00000040  7e f0 0e f0 3e f0 c0 fe 3e 0e 7e f0 7e 70 7f 1f |`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
The cause of my problem was that I had incorrectly set the crystal frequency of the controller. Once I corrected that, I was able to correctly read text from the serial line. One remaining observation is that the `NewLine` method actually sends a carriage return character, rather than a line feed. I corrected this by adding a line feed character to the end of the string.

Listing 14: HelloPST.spin after corrections

```
'' HelloPST.spin
'' Test message to Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfraq = 5_000_000

OBJ

  pst : "Parallax Serial Terminal"

PUB TestMessages

  '' Send test messages to Parallax Serial Terminal

  pst.Start(115_200)

  repeat
    pst.Str(string("This is a test message!", $0a))
    pst.NewLine
    waitcnt(clkfreq + cnt)
```

4.1.1 Questions

1. Methods in the same object are called by using the method name. Methods in other objects are called by using the object’s nickname and the method name, joined with a period. (p 85)

2. Methods are given arguments and their return values are obtained using the same notation whether or not the method is from another object. (p 86)

3. The object file must be in the same folder as the calling object, or in the same folder as the `Propeller Tool.exe` file. (p 86)

4. Object hierarchy can be viewed in the upper-left pane of the Object Info Window. (p 87)

5. Documentation comments are either prefixed with a pair of single-quotes (’’ like this) or surrounded by two pairs of curly-braces ({{ like this }}). (p 103)
6. Code can be filtered out, leaving documentation comments only, by using the Documentation View of the Object Window. (pp 103-104)

7. Objects, by convention, use the **Start** and **Stop** methods for launching new cogs and stopping running cogs. (pp 90, 92)

8. Give the object multiple nicknames in the **OBJ** block. (p 96) Alternately, give the object a nickname with a subscript, as an array. (?)

9. The net effect of the **Start** method calling the **Stop** method is that, if the user of that object mistakenly calls **Start** twice, it won’t end up running in two different cogs.

10. Custom characters from the wonky Parallax character set are inserted in a documentation block by mouse-typing them from the Character Chart. (p 94)

11. An object may call its own **PUBlic** and **PRIvate** methods, but only the **PUBlic** methods in other objects.

12. Give the object multiple nicknames in the **OBJ** block. (p 96)

13. Propeller Library objects are stored in the Library folder where the Propeller Tool.exe file is stored. (p 86)

14. Object Interface information is viewed in the same way as documentation comments, by selecting the Documentation View in the Object Info window. (p 87)

15. The **String** directive stores a string in the program memory. (p 104)

16. The **Str** method of the Parallax Serial Terminal object expects zero-terminated strings. (pp 104-105)

17. An object’s documentation should explain what arguments a method requires, what the method returns, and any effects caused by invoking the method.

18. Character strings may be stored in a **DAT** block (**ExampleString byte "like this", 0**). (p 112)

19. The **long**, **word**, and **byte** keywords are used to declare variables in a **VAR** block, to declare the size of sequence elements in a **DAT** block, and to identify the size of a value when dereferencing a pointer.

20. The value 1.5 is pre-assigned to the variable **a**, whatever that means. (p 114)

21. The address of a variable is obtained by prefixing it with the @@ operator, **Pointer := @LikeThis** (p 117)
22. A values may be accessed through a pointer by dereferencing it using the long, short, or byte keywords (Value := byte[LikeThis] or Value := byte[LikeThis][0]). Values after the pointed-to address may be accessed using the second form and incrementing the index (Value := byte[LikeThis][1] and Value := byte[LikeThis][2]).

23. Yes. See above.

24. Yes. See above.

4.1.2 Exercises

1. OBJ
   led : "MyLedObject"

2. led.on(4)

3. I have no idea how to write \( f = T^{-1} \) in the wonky Parallax font, but I can write it in \LaTeX{}.

4. PRI calcArea(height, width) : area

5. OBJ

6. uart[2].str("Hello!!!")

7. DAT
   Hi byte "Hello!!!", 0

8. PUB circumference(d) : c
   c := f.FMul(d, PI)

9. address := fst.FloatToString(c)

4.1.3 Project 1

'' File: Bs2IoLite.spin
'' implements methods with effects similar to PBASIC commands

{{
  high:
    A specified pin outputs high when this method returns.

  Arguments:
    pin: the pin to set high
}}
Return: nothing
}
PUB high(pin)
   outa[pin]~
   dira[pin]~

{{
   low:
   A specified pin outputs low when this method returns.

   Arguments:
   pin: the pin to set low

   Return: nothing
}}
PUB low(pin)
   outa[pin]~
   dira[pin]~

{{
toggle:
   If the pin outputs high when this method is called, it outputs low when the method returns. If the pin outputs low when this method is called, it outputs high when the method returns. If the pin does not output when the method is called, this method has no effect.

   Arguments:
   pin: The pin to toggle.

   Return: nothing
}}
PUB toggle(pin)
   if dira[pin]
      !outa[pin]

{{
in:
   Returns the value input to a pin.

   Arguments:
   pin: the pin to read

   Return: The value currently applied to the pin.
}}
PUB in(pin)
   dira[pin]~
   result := ina[pin]

{{
pause:
Returns after a delay, specified in microseconds.

Arguments:
  delay: The time to wait, in microseconds.

Return: Nothing.
}}

PUB pause(delay)
  waitcnt(clkfreq/1000*delay + cnt)

4.1.4 Project 2

Listing 15: Blinker.spin modified to test stack usage

{{ File: Blinker.spin
Example cog manager for a blinking LED process.
Modified for examining memory usage with the Stack Length object
}}

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

VAR

  long stack[10] 'Cog stack space
  byte cog 'Cog ID

OBJ

  Stk : "Stack Length"
  pst : "Parallax Serial Terminal"

PUB TestStack(pin, rate, reps)
  pst.Start(115_200)
  Stk.Init(@stack,10)
  Start(pin, rate, reps)
  waitcnt(clkfreq/2 + cnt)
  pst.Str( String("Stack Usage: ") )
  pst.Dec( Stk.GetLength(0,0) )
  pst.Str( String( pst#NL, pst#LF ) )

PUB Start(pin, rate, reps) : success

  {{Start new blinking process in new cog; return True if successful.
Parameters:
  pin - the I/O connected to the LED circuit; see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles
}}

Stop
  success := (cog := cognew(Blink(pin, rate, reps), @stack) + 1)
PUB Stop
'' Stop Blinking process, if any.

    if cog
    cogstop(cog ~ - 1)

PUB Blink(pin, rate, reps)
{(Blink an LED circuit connected to pin at a given rate for reps repetitions.

Parameters:
  pin - the I/O connected to the LED circuit; see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles}
)
  dira[pin]~
  outa[pin]~

  repeat reps * 2
    waitcnt(rate/2 + cnt)
    !outa[pin]

4.1.5 Project 3

Listing 16: TickTock.spin

'' File: TickTock.spin

CON

_xinfreq = 5.000.000
_clkmode = xtal1 + pll1x

VAR

long seconds, minutes, hours, days, dT, T
long stack[42]
byte cog

PUB Start( inSeconds, inMinutes, inHours, inDays ): success
  Stop
  seconds := inSeconds
  minutes := inMinutes
  hours := inHours
  days := inDays

  success := (cog := cognew(TickTock, @stack) +1)

PUB Stop
  if cog
  cogstop(cog ~ - 1)

PUB TickTock
dT := clkfreq / 500
T := cnt

repeat
    waitcnt(T += dT)
    seconds++
    if seconds == 60
        minutes++
        seconds~
    if minutes == 60
        hours++
        minutes~
    if hours == 24
        days++
        hours~

PUB Get( ptrDays , ptrHours , ptrMinutes , ptrSeconds )
long[ptrDays] := days
long[ptrHours] := hours
long[ptrMinutes] := minutes
long[ptrSeconds] := seconds

4.2 Data: Lab 7

In the light detection project, TestRcDecay.spin, I found that I could vary the time returned by the counter between 48 ticks at the one end of the potentiometer to 6400 ticks at the other end.

In the modified light detection project, I found that I could vary the time returned by the counter by varying the illumination.

<table>
<thead>
<tr>
<th>light source</th>
<th>ticks</th>
</tr>
</thead>
<tbody>
<tr>
<td>spotlight</td>
<td>1</td>
</tr>
<tr>
<td>flashlight, close</td>
<td>6,000</td>
</tr>
<tr>
<td>ambient light</td>
<td>15,000</td>
</tr>
<tr>
<td>darkened room</td>
<td>1,000,000</td>
</tr>
<tr>
<td>covered sensor</td>
<td>30,000,000</td>
</tr>
</tbody>
</table>

In the IrObjectDetection.spin project, I found that I could detect my hand at a distance of about a foot, and my PE Kit manual at a distance of about a cubit.

In the IrDetector.spin distance detection project, I found that my hand read a distance of 25 when about an inch from the detector and a distance of about 175 when about a cubit from the detector. I only achieved this after correcting several hardware bugs and one very elusive software bug. Apparently, when you screw up the ranged subscript operator with just one dot (LikeThis[30.26]), that’s perfectly acceptable in the Spin language, and no error is raised.
The special purpose registers can be accessed through their \textit{spr[i]} name, or through their bit-addressable aliases.

<table>
<thead>
<tr>
<th>index</th>
<th>alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>ctra</td>
</tr>
<tr>
<td>9</td>
<td>ctrb</td>
</tr>
<tr>
<td>10</td>
<td>frqa</td>
</tr>
<tr>
<td>11</td>
<td>frqb</td>
</tr>
<tr>
<td>12</td>
<td>phsa</td>
</tr>
<tr>
<td>13</td>
<td>phsb</td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

4.2.1 Questions

1. The Propeller has two counter modules: Counter A and Counter B (p 125)

2. Each counter module has three registers. These are the phase register, the frequency register, and the control register, generically referred to as PHS, FRQ, and CTR, respectively. (p 125)

3. Counter A’s registers are specifically called \texttt{phsa}, \texttt{frqa}, and \texttt{ctra}. (p 125)

4. Counter B’s registers are specifically called \texttt{phsb}, \texttt{frqb}, and \texttt{ctrb}. (p 125)

5. The FRQ register is conditionally added to the PHS register every tick. (p 126)

6. The mode in the CTR register determines whether the FRQ register is added to the PHS register. (p 126)

7. The mode in the CTR register determines whether the PHS register affects the output pins. (p 126)

8. The time required for a measured decay in voltage relates to values in the RC decay circuit. If only one value is an independent variable, it can be calculated from the other values. (p 127)

9. The current-limiting resistor is required by convention, for equipment safety. (p 127)

12. CTRMODE bits are in \texttt{ctrx[30..26]} (p 129)

13. CTRMODE bits control how FRQ is added to PHS (p 129)

14. The CTRMODE field must be set to \%00100 to enable the “POS detector” mode. The APIN (and possibly BPIN) field(s) must be set to the pin(s) to monitor (and possibly affect). (pp 128-129)
15. Most practically, the FRQ register should be set to 1. (p 130)

16. Set the CTR reg with the desired CTRMODE and APIN settings. (p 128) Set the FRQ register to the desired rate: 1. (p 130) Set the pin high, and wait for voltage to stabilize. (p 130)

17. To start the measurement, clear the PHS register and the pin. (p 130)

18. RC measurements can be performed concurrently because they do not require the processor. (p 131)

19. In RC decay applications, the counter monitors an I/O pin. In DA conversion applications, the counter outputs a signal on the pin. (pp 128, 135)

20. The duty cycle is determined by the ratio of the FRQ register to its maximum value ($2^{32}$, or 4,294,967,296). (p 135)

21. The carry bit from the phase adder controls the value on the pin in DUTY mode. (pp 135-136)

22. The scale constant provides a conversion factor from the number of desired levels of brightness and the full duty setting of the FRQ register. (p 138)

23. Special Purpose Registers may be addressed with spr[index]. (p 140-141)

24. \texttt{ctrb} is also \texttt{spr[9]} (p 140)

25. \texttt{frqa} is also \texttt{spr[10]} (p 140)

26. \texttt{spr[13]} is also \texttt{phsb} (p 140)

27. \texttt{ctrb} := myVar or \texttt{spr[9]} := myVar (p 140)

28. \texttt{spr[8]} and \texttt{spr[9]} are \texttt{ctra} and \texttt{ctrb}, respectively. (p 140)

29. Bit 31 of the PHS register controls the selected I/O pin in NCO mode. (p 143)

30. Whenever the counter is configured in NCO mode, FRQ is added to PHS every clock cycle.

31. \[
\text{FRQ} = \text{freq} \times \frac{2^{32}}{\text{clkfreq}}
\]

(p 143)

32. \[
\text{freq} = \text{FRQ} \times \frac{\text{clkfreq}}{2^{32}}
\]

(p 144)
33. Set the pin to input, set the counter to some other mode, or set the frequency to 0. (p 146)

34. Yes, run one on each counter. (p 151-152)

35. Change the FRQ register.

36. The POSEDGE and NEGEDGE modes can count the number of cycles in a given period of time. The number of cycles divided by the time gives the frequency. (p 158)

37. while spr[13] & < 31

4.2.2 Project 1

Listing 17: DAC2Channel.spin

{{
DAC2Channel.spin

Object interface:

PUB Init(channel, ioPin, bits, level)
PUB Update(channel, level)
PUB Remove(channel)
}}

VAR

    long scale[2]
    byte pin[2]

PUB Init(channel, ioPin, bits, level)

    ' calculate scale
    scale = 2^32 / 2^bits
    scale = |< (32 - bits)
    scale[channel] := |< (32 - bits)

    ' store pin
    pin[channel] := ioPin

    ' set ctr reg to DUTY mode
    ' ctr reg is spr[ 8 + channel ]
    ' DUTY mode is %00110
    ' set ctr reg to APIN
    spr[ 8 + channel ] := (%00110 << 26) + ioPin

    ' set pin to output

while spr[13] & < 31
dira[ioPin]~

' set frm reg to obtain duty period
' frm reg is spr[10 + channel ]
spr[10 + channel ] := level * scale[channel]

PUB Update(channel, level)
' set frm reg to obtain duty period
' frm reg is spr[10 + channel ]
spr[10 + channel ] := level * scale[channel]

PUB Remove(channel)
' pin off
dira[ pin[channel] ]~
' counter off
spr[ 8 + channel ]~

4.2.3 Project 2

Listing 18: TestFourPWM.spin

{{
TestFourPWM.spin
Taken from TestDualPWM.spin, as provided
Modified to handle four servos
}}

CON

_clkmode = xtal1 + pll16x
_xinfreq = 5.000.000

PUB TestPwm | tc, tHa, tHb, t, us

us := clkfreq/1.000.000

ctra[30..26] := ctrb[30..26] := %00100  ' NCO single-ended
frqa := frqb := 1  ' increment by 1

dira[4]~
  ' Four pins now
dira[5]~
dira[6]~
dira[7]~

tC := 200.000 * us

tHa := 700 * us
tHb := 2200 * us

t := cnt

repeat tHa from (700 * us) to (2200 * us)
ctra[5..0] := 4  ' first two pins
ctrb[5..0] := 5
phsa := -tHa
phsb := -tHb

waitcnt(2200 * us + cnt) ' wait for first two to finish

ctra[5..0] := 6 ' first two pins
ctrb[5..0] := 7
phsa := -tHa
phsb := -tHb

t += tC ' and wait for the next loop
waitcnt(t)

5 Conclusion

During this lab, I learned the features of the Parallax Propeller microcontroller, and a few
of the capabilities of the official Propeller IDE. Mostly, however, I used vim for editing,
bstc for compiling and loading, and cu from the Taylor UUCP package for serial terminal
connections.

There are features of the Spin language I dislike. The operators that may or may not be
assignment operators, depending on context, are uncomfortable to use in an rvalue context. I
keep thinking they’re going to change the variable when they’re not supposed to. Or, maybe
they are supposed to, and my concerns stem from confusion rather than paranoia.

And speaking of error-prone operators, it bit me pretty hard once when Spin accepted a
floating-point number in a context where only an integer made sense. That is, when I meant
to use the .. range operator, but I left out a ‘.’. The result was that I gave a floating-point
index to an array, which no sane language should have accepted.

I have never liked the use of indentation as a meaningful construct. It is all the more
disconcerting because I do not often program in languages with that feature, but I hold that
it’s a misfeature in principle. Blocks of code should have some syntactical delimation.
Brackets are great. Even a begin / end set would be better than nothing at all.

Still, there are features of the language I really like. The decode and encode operators (<| arg
and |> arg) are handy. The former can readily be duplicated with (1 << arg), but
the latter can only be accomplished with a loop and a spare variable to catch the result
( unsigned var=0; while ( (var <<= 1)+, arg >>= 1) ; (var >>= 1)-). I can’t tell
if I like maxvalue and minvalue operators. On one hand, they seem like a poor man’s assert
or exception. On the other hand, they’re still a shorthand, nifty way to make sure that values
don’t wander out of range. The post-set and post-clear operators ( and , respectively)
seem superfluous, but they concisely represent an action that is done every other line when
programming for the Propeller.

Programming for an individual Cog, launching code on a separate Cog, handling the
Hub/Distributor model, and directly interfacing with the I/O pins were all straightforward.
The use of the Counters and their SPRs to handle input and output was considerably more
abstract. I will need to play with these features a bit longer before I am comfortable with
them. Overall, this was a great introduction to a new programming platform, and I wish I’d had the time to explore it more thoroughly.