599 Menlo Drive, Suite 100
Rocklin, California 95765, USA
Office: (916) 624-8333
Fax: (916) 624-8003

Sales:sales@parallax.com
Technical: support@parallax.com
Web Site: www.parallax.com

Counter Modules and Circuit Applications

PROPELLER EDUCATION KIT LAB SERIES

Introduction

Each Propeller cog has two *counter modules*, and each counter module can be configured to independently perform repetitive tasks. So, not only does the Propeller chip have the ability to execute code simultaneously in separate cogs, each cog can also orchestrate up to two additional processes with counter modules while the cog continues executing program commands.

Counters can provide a cog with a variety of services; here are some examples:

- Measure pulse and decay durations
- Count signal cycles and measure frequency
- Send numerically controlled oscillator (NCO) signals, i.e. square waves
- Send phase-locked loop (PLL) signals, which can be useful for higher frequency square waves
- Signal edge detection
- Digital to analog (D/A) conversion
- Analog to digital (A/D) conversion
- Provide internal signals for video generation

Since each counter module can be configured to perform many of these tasks in a "set it and forget it" fashion, it is possible for a single cog to execute a program and at the same time do things like generate speaker tones, control motors and/or servos, count incoming frequencies, and transmit and/or measure analog voltages.

This lab provides examples of how to use ten of the thirty-two different counter modes to perform variations of eight different tasks:

- RC decay time measurement for potentiometers and photoresistor
- D/A conversion to control LED brightness
- NCO signals to send speaker tones
- NCO signals for modulated IR for object and distance detection
- Count speaker tone cycles
- Detect a signal transition
- Pulse width control
- Generate high frequency signals for metal proximity detection

A cog doesn't necessarily have "set and forget" a counter module. It can also dedicate itself to processes involving counter modules to do some amazing things, including a number of audio and video applications. This lab also includes an example that demonstrates this kind of cog-counter relationship, applied to sending multiple PWM signals.

Please note that the majority of the code examples in this lab are top level objects that demonstrate various details and inner workings of counter modules. If you plan on incorporating these concepts into library objects that are designed to be used by other applications, make sure to pay close attention to the section entitled: Probe and Display PWM – Add an Object, Cog and Pair of Counters that begins on page 39.

Prerequisites

Please complete the following labs first before continuing here:

- Setup and Testing
- I/O and Timing
- Methods and Cogs
- Objects

How Counter Modules Work

Each cog has two counter modules, Counter A and Counter B. Each cog also has three 32-bit special-purpose registers for each of its counter modules. The Counter A special purpose registers are `phsa`, `frqa`, `ctra`, and Counter B's are `phsb`, `frqb` and `ctrb`. Note that each counter name is also a reserved word in Spin and Propeller assembly. If this lab is referring to a register generally, but it doesn't matter whether it's for Counter A or Counter B, it will use the generic names PHS, FRQ, and CTR.

Here is how each of the three registers works in a counter module:

- PHS – the "phase" register gets updated every clock tick. A counter module can also be configured make certain PHS register bits affect certain I/O pins.
- FRQ – the "frequency" register gets conditionally added to the PHS register every clock tick. The counter module's mode determines what conditions cause FRQ to get added to PHS. Mode options include "always", "never", and conditional options based on I/O pin states or transitions.
- CTR – the "control" register configures both the counter module's mode and the I/O pin(s) that get monitored and/or controlled by the counter module. Each counter module has 32 different modes, and depending on the mode, can monitor and/or control up to two I/O pins.

Measuring RC Decay with Positive Detector Mode

Resistor-Capacitor (RC) decay is useful for a variety of sensor measurements. Some examples include:

- Dial or joystick position with one or more potentiometers
- Ambient light levels with either a light-dependent resistor or a photodiode
- Surface infrared reflectivity with an infrared LED and phototransistor
- Pressure with capacitor plates and a compressible dielectric
- Liquid salinity with metal probes

RC Decay Circuit

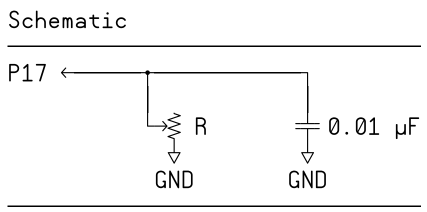
RC decay measurements are typically performed by charging a capacitor (C) and then monitoring the time it takes the capacitor to discharge through a resistor (R). In most RC decay circuits, one of the values is fixed, and the other varies with an environmental variable. For example, the circuit in Figure 1 is used to measure a potentiometer knob's position. The value of C is fixed at 0.01 µF, and the value of R varies with the position of the potentiometer's adjusting knob (the environmental variable).

✓ Build the circuit shown in Figure 1 on your PE Platform. This circuit and all others in this lab are in addition to the basic Propeller circuit built in the Setup and Testing lab.

Figure 1: RC Decay Parts and Circuit

Parts List

(1) Potentiometer 10 kΩ
(1) Capacitor – 0.01 µF
(misc) Jumper wires



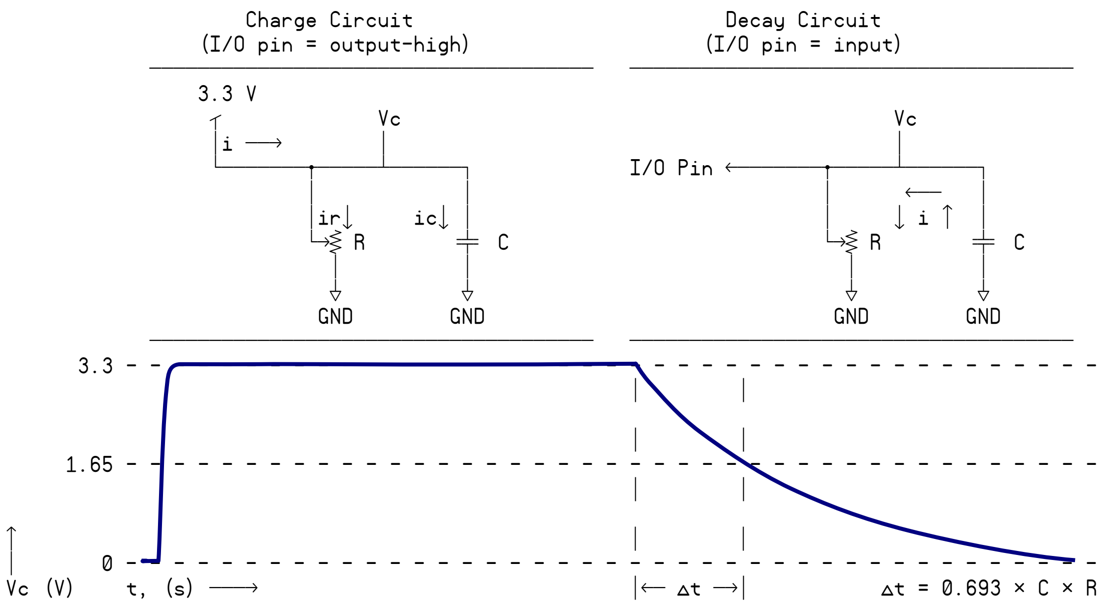
Measuring RC Decay

Before taking the RC decay time measurement, the Propeller chip needs to set the I/O pin connected to the circuit to output-high. This charges the capacitor up to 3.3 V as shown on the left side of Figure 2. Then, the Propeller chip starts the RC decay measurement by setting the I/O pin to input, as shown on the right side of Figure 2. When the I/O pin changes to input, the charge built up in the capacitor drains through the variable resistor. The time it takes the capacitor to discharge from 3.3 V down to the I/O pin's 1.65 V threshold is:

$$\Delta t = 0.693 \times C \times R$$

Since 0.693 and C are constants, the time **Δt** it takes for the circuit to decay is directly proportional to R, the variable resistor's resistance.

Figure 2: RC Charge and Decay Circuits and Voltages



Where is the current-limiting series resistor?

The Propeller chip's I/O pin driver circuits do not need to be protected from the sudden initial current spike that results when the I/O pin is taken from either output-low or input to output-high. The I/O pins' output capacity and current-limiting characteristics prevent any damage from occurring.

If you try to use this circuit with a different microcontroller, you will probably need to include a current-limiting resistor between the I/O pin and the RC circuit. Make sure that it is large enough to prevent the I/O pin from getting damaged. The decay time won't be linear because the voltage divider created by the second resistor causes the RC decay measurement's starting voltage to vary. Choosing an R in the RC circuit that is very large compared to the series resistor will help the decay time more closely resemble a linear behavior.

Positive Detector Mode

In positive detector mode, the Propeller chip's counter module monitors an I/O pin, and adds FRQ to PHS for every clock tick in which the pin is high. To make the PHS register accumulate the number of clock ticks in which the pin is high, simply set the counter module's FRQ register to 1. For measuring RC decay, the counter module should start counting (adding FRQ = 1 to PHS) as soon as the I/O pin is changed from output-high to input. After the signal level decays below the I/O pin's 1.65 V logic threshold, the module no longer adds FRQ to PHS, and what's stored in PHS is the decay time measurement in system clock ticks.

One significant advantage to using a counter module to measure RC decay is that the cog doesn't have to wait for the decay to finish. Since the counter automatically increments PHS with every clock tick in which the pin is high, the program is free to move on to other tasks. The program can then get the value from the PHS register whenever it's convenient.

Configuring a Counter Module for "POS detector" Mode

Figure 3 shows excerpts from the Propeller Library's CTR object's Counter Mode Table. The CTR object has counter module information and a code example that generates square waves. The CTR object's Counter Mode Table lists the 32 counter mode options, seven of which are shown below. The mode we will use for the RC decay measurement is positive detector, shown as "POS detector" in the table excerpts.

Figure 3: Excerpts from the CTR.spin's Counter Mode Table

CTRMODE	Description	Accumulate FRQ to PHS	APIN output*	BPIN output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
.				
.				
%01000	POS detector	A^1	0	0
%01001	POS detector w/feedback	A^1	0	!A^1
%01010	POSEDGE detector	A^1 & !A^2	0	0
%01011	POSEDGE detector w/feedback	A^1 & !A^2	0	!A^1
.				
.				
%11111	LOGIC always	1	0	0

```
* must set corresponding DIR bit to affect pin
```

A^1 = APIN input delayed by 1 clock
A^2 = APIN input delayed by 2 clocks
B^1 = BPIN input delayed by 1 clock

Notice how each counter mode in Figure 3 has a corresponding 5-bit CTRMODE code. For example, the code for "POS detector" is %01000. This value has to be copied to a bit field within the counter module's CTR register to make it function in "POS detector" mode. Figure 4 shows the register map for the `ctra` and `ctrb` registers. Notice how the register map names bits 31..26 CTRMODE. These are the bits that the 5 bit mode code from Figure 3 have to be copied to to make a counter module operate in a particular mode.

Like the `dira`, `outa` and `ina` registers, the `ctra` and `ctrb` registers are bit-addressable, so the procedure for setting and clearing bits in this register is the same as it would be for a group I/O pin operations with `dira`, `outa`, or `ina`. For example, here's a command to make Counter A a "POS detector":

```
  ctra[30..26] := %01000
```

Figure 4: CTRA/B Register Map from CTR.spin

bits	31	30..26	25..23	22..15	14..9	8..6	5..0
Name	—	CTRMODE	PLLDIV	———	BPIN	——	APIN

The Counter Mode Table and CTRA/B Register Map appear in the Propeller Library's CTR object, and also in the *Propeller Manual's* CTRA/B section, located in the Spin Reference chapter. APIN and BPIN are I/O pins that the counter module might control, monitor, or not use at all, depending on the mode.

Notice also in Figure 4 how there are bit fields for PLLDIV, BPIN, and APIN. PLLDIV is short for "phase-locked loop divider" and is only used for PLL counter modes, which can synthesize high-frequency square waves (more on this later). APIN (and BPIN for two-pin modes) have to store the I/O pin numbers that the counter module will monitor/control. In the case of the Counter A module set to positive detector mode, `frqa` gets added to `phsa` based on the state of APIN during the previous clock. (See the A^1 reference and footnote in Figure 3.) So the APIN bit field needs to store the value 17 since P17 will monitor the RC circuit's voltage decay. Here's a command that sets bits 5..0 of the `ctra` register to 17:

```
  ctra[5..0] := 17
```

Remember that `frqa` gets added to `phsa` with every clock tick where APIN was high. To make the counter module track how many clock ticks the pin is high, simply set `frqa` to 1:

```
  frqa := 1
```

At this point, the `phsa` register gets 1 added to it for each clock tick in which the voltage applied to P17 is above the Propeller chip's 1.65 V logic threshold. The only other thing you have to do before triggering the decay measurement is to clear the `phsa` register.

In summary, configuring the counter module to count clock ticks when an I/O pin is high takes three steps:

1) Store %01000 in the CTR register's mode bit field:

```
ctra[30..26] := %01000
```

2) Store the I/O pin number that you want monitored in the CTR register's APIN bit field:

```
ctra[5..0] := 17
```

3) Store 1 in the FRQ register so that the `phsa` register will get 1 added to it for every clock tick that P17 is high:

```
frqa := 1
```

1 isn't the only useful FRQ register value. Other FRQ register values can also be used to prescale the sensor input for calculations or even for actuator outputs. For example, FRQ can instead be set to `clkfreq/1_000_000` to count the decay time in microseconds.

```
frqa := clkfrq/1_000_000
```

This expression works for Propeller chip system clock frequencies that are common multiples of 1 MHz. For example, it would work fine with a 5 MHz crystal input, but not with a 4.096 MHz crystal since the resulting system clock frequency would not be an exact multiple of 1 MHz.

One disadvantage of larger FRQ values is that the program can not necessarily compensate for the number of clock ticks between clearing the PHS register and setting the I/O pin to input. A command that compensates for this source of error can easily be added after the clock tick counting is finished, and it can be followed by a second command that scales to a convenient measurement unit, such as microseconds.

Measure input or output signals. This counter mode can be used to measure the duration in which an I/O pin sends a high signal as well as the duration in which a high signal applied to the I/O pin. The only difference is the direction of the I/O pin when the measurement is taken.

"Counting" the RC Decay Measurement

Before the RC decay measurement, the capacitor should be charged. Here's a piece of code that sets P17 to output-high, then waits for 10 µs, which is more than ample for charging the capacitor in the Figure 1 RC network.

```
dira[17] := outa[17] := 1
waitcnt(clkfreq/100_000 + cnt)
```

To start the decay measurement, clear the PHS register, and then set the I/O pin that's charging the capacitor to input:

```
phsa~
dira[17]~
```

After clearing `phsa` and `dira`, the program is free to perform other tasks during the measurement. At some later time, the program can come back and copy the `phsa` register contents to a variable. Of course, the program should make sure to wait long enough for the decay measurement to complete. This can be done by polling the clock, waiting for the decay pin to go low, or performing a task that is known to take longer than the decay measurement.

To complete the measurement, copy the `phsa` register to another variable and subtract 624 from it to account for the number of clock ticks between `phsa~` and `dira[17]~`. The result of this subtraction can also be set to a minimum of 0 with `#> 0`. This will make more sense than -624 when the resistance is so low that it pulls the I/O pin's output-high signal low.

```
time := (phsa – 624) #> 0
```

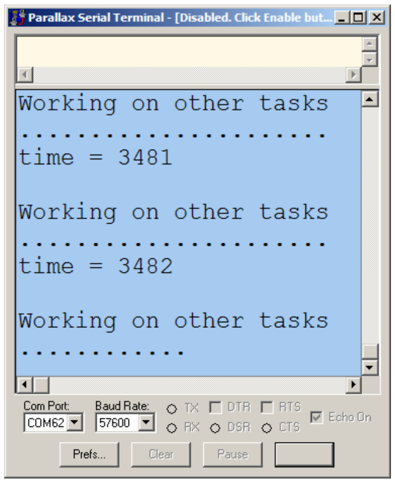
Where did 624 come from?

The number of clock ticks between `phsa~` and `dira[17]~` was determined by replacing the 0.01 μF capacitor with a 100 pF capacitor and finding the lowest value before zero was returned. In the test program, `time := phsa` replaces `time := (phsa – 624) #> 0`, and the lowest measurable value was 625.

Example Object Measures RC Decay Time

The TestRcDecay object applies the techniques just discussed to measure RC decay in a circuit with variable resistance controlled by the position of a potentiometer's adjusting knob. As shown in Figure 5, the program displays a "working on other tasks" message after starting the RC decay measurement to demonstrate that the counter module automatically increments the `phsa` register until the voltage applied to P17 decays below the Propeller chip's 1.65 V I/O pin threshold. The program can then check back at a later time to find out the value stored in `phsa`.

Figure 5: RC Decay Times



```
Working on other tasks
......................
time = 3481

Working on other tasks
......................
time = 3482

Working on other tasks
............
```

- ✓ Open the TestRcDecay.spin object. It will call methods in FullDuplexSerialPlus.spin, so make sure they are both saved in the same folder.
- ✓ Open Parallax Serial Terminal and set its Com Port field to the same port the Propeller Tool software uses to load programs into the Propeller chip.
- ✓ Use the Propeller tool to load TestRcDecay.spin into the Propeller chip.
- ✓ Immediately click the Parallax Serial Terminal's Enable button. (Don't wait for the Program to finish loading. In fact, you can click the Parallax Serial Terminal's Enable button immediately after you have pressed F10 or F11 in the Propeller Tool software.)
- ✓ Try adjusting the potentiometer knob to various positions and note the time values. They should vary in proportion to the potentiometer's adjusting knob's position.

```
'' TestRcDecay.spin
'' Test RC Decay circuit decay measurements.

CON

  _clkmode = xtal1 + pll16x                 ' System clock → 80 MHz
  _xinfreq = 5_000_000

  CR = 13

OBJ

  Debug: "FullDuplexSerialPlus"             ' Use with Parallax Serial Terminal to
                                            ' display values

PUB Init

  'Start serial communication, and wait 2 s for connection to Parallax Serial Terminal.

  Debug.Start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)

  ' Configure counter module.

  ctra[30..26] := %01000                    ' Set mode to "POS detector"
  ctra[5..0] := 17                          ' Set APIN to 17 (P17)
  frqa := 1                                 ' Increment phsa by 1 for each clock tick
  main                                        ' Call the Main method

PUB Main | time
'' Repeatedly takes and displays P17 RC decay measurements.
  repeat

     ' Charge RC circuit.

     dira[17] := outa[17] := 1                ' Set pin to output-high
     waitcnt(clkfreq/100_000 + cnt)           ' Wait for circuit to charge

     ' Start RC decay measurement.  It's automatic after this...

     phsa~                                    ' Clear the phsa register
     dira[17]~                                ' Pin to input stops charging circuit

     ' Optional - do other things during the measurement.

     Debug.str(String(CR, CR, "Working on other tasks", CR))
     repeat 22
       Debug.tx(".")
       waitcnt(clkfreq/60 + cnt)

     ' Measurement has been ready for a while.  Adjust ticks between phsa~ & dira[17]~.

     time := (phsa - 624) #> 0

     ' Display Result

     Debug.Str(String(13, "time = "))
     Debug.Dec(time)
     waitcnt(clkfreq/2 + cnt)
```

Two Concurrent RC Decay Measurements

Since the counter module keeps track of high time after the decay starts, it is possible to take two concurrent RC decay measurements on different pins. Let's do so with P25 and a light-dependent resistor (abbreviated LDR and also called a photoresistor) instead of a potentiometer. The second measurement will start later than the first since the `phsb~` and `dira[25]~` commands will follow `dira[17]~`. However, the decays can occur in parallel, and while the decays last, the cog continues to execute other commands.

- ✓ Build the circuit shown in Figure 6.
- ✓ Modify a copy of TestRcDecay.spin so that it can measure the circuits from Figure 1 and Figure 6 concurrently.

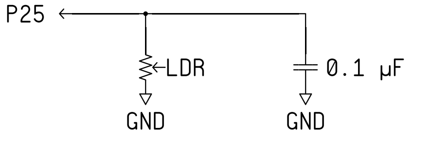
Be careful, you'll need to add commands that set the values of `ctrb`, `frqb`, and `phsb`, but the DIR register should be `dira`, not `dirb`. `dirb` is reserved for I/O pins 32..63 in a module with 64 I/O pins. Also, `phsb~` and `dira[25]~` should come immediately after `dira[17]~`.

Figure 6: Second RC Decay Parts and Circuit

Parts List

(1) Photoresistor
(1) Capacitor - 0.1 µF
(misc) Jumper wires

Schematic



D/A Conversion – Controlling LED Brightness with Duty Mode

A counter module in duty mode allows you to control a signal that can be used for digital to analog conversion with the FRQ register. Although the signal switches rapidly between high and low, the average time it is high (the duty) is determined by the ratio of the FRQ register to 2^{32}.

$$\text{duty} = \frac{\text{pin high time}}{\text{time}} = \frac{\text{FRQ}}{4\_294\_967\_296} \qquad \textbf{Eq. 1}$$

For D/A conversion, let's say the program has to send a 0.825 V signal. That's 25% of 3.3 V, so a 25% duty signal is required. Figuring out the value to store in the FRQ register is simple. Just set duty = 0.25 and solve for FRQ.

$$0.25 = \frac{\text{FRQ}}{4\_294\_967\_296} \quad \rightarrow \quad \text{FRQ} = 1\_073\_741\_824$$

You can also use Eq. 1 to figure out what duty signal an object is sending. Let's say 536_870_912 is stored in a counter module's FRQ register, and its CTR register has it configured to duty mode.

$$\text{duty} = \frac{536\_870\_912}{4\_294\_967\_296} = 0.125$$

On a 3.3 V scale, that would resolve to 0.375 V. Again, the great thing about counters is that they can do their jobs without tying up a cog. So, the cog will still be free to continue executing commands while the counter takes care of maintaining the D/A conversion duty signal.

How Duty Mode Works

Each time FRQ gets added to PHS, the counter module's phase adder (that adds FRQ to PHS with every clock tick) either sets or clears a carry flag. This carry operation is similar to a carry operation in decimal addition. Let's say you are allowed 3 decimal places, and you try to add two values that add up to more than 999. Some value would normally be carried from the hundreds to the thousands slot. The binary version of addition-with-carry applies when the FRQ register gets added to the PHS register when the result is larger than $2^{32} - 1$. If the result exceeds this value, the PHS adder's carry flag (think of it as the PHS registers "bit 32") gets set.

The interesting thing about this carry flag is that the amount of time it is 1 is proportional to the value stored in the FRQ register divided by 2^{32}. In single-ended duty mode, the counter module's phase adder's carry bit controls an I/O pin's output state. Since the time in which the phase adder's carry bit is 1 is proportional to FRQ/2^{32}, so is the I/O pin's output state. The I/O pin may rapidly switch between high and low, but the average pin high time is determined by the FRQ-to-2^{32} ratio shown in Eq. 1 above.

Parts and Circuit

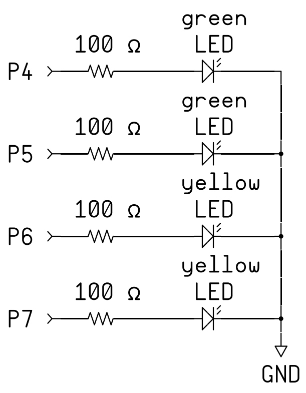
Yes, it's back to LEDs for just a little while, and then we'll move on to other circuits. Previous labs used LEDs to indicate I/O pin states and timing. This portion of this lab will use duty mode for D/A conversion to control LED brightness.

Figure 7: LED Circuit for Brightness Control with Duty Signals

Parts List

(4) Resistors 100 Ω
(2) LEDs - green
(2) LEDs - yellow
(misc) Jumper wires

Schematic



✓ Add the circuit shown in Figure 7 to your PE Platform, leaving the RC decay circuit in place.

Configuring a Counter for Duty Mode

Figure 8 shows more entries from the CTR object's and Propeller Manual's Counter Mode Table. There are two types of duty modes, single-ended and differential. With single-ended, the APIN mirrors the state of the phase adder's carry bit. So, if FRQ is set to the 1_073_741_824 value calculated earlier, the APIN will be high ¼ of the time. An LED circuit receiving this signal will appear to glow at ¼ of its full brightness. In differential mode, the APIN signal still matches the phase adder's carry bit, while the PBIN is the opposite value. So whenever the phase adder's carry bit (and APIN) are 1, BPIN is 0, and vice-versa. If FRQ is set to 1_073_741_824, APIN would still cause an LED to glow at ¼ brightness while BPIN will glow at ¾ brightness.

Figure 8: More Excerpts from the CTR.spin's Counter Mode Table

CTRMODE	Description	Accumulate FRQ to PHS	APIN output*	BPIN output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
. . .				
%00110	DUTY single-ended	1	PHS-Carry	0
%00111	DUTY differential	1	PHS-Carry	!PHS-Carry
. . .				
%11111	LOGIC always	1	0	0

```
* must set corresponding DIR bit to affect pin

 A¹ = APIN input delayed by 1 clock
 A² = APIN input delayed by 2 clocks
 B¹ = BPIN input delayed by 1 clock
```

Figure 9 is a repeat of Figure 3. This time, the counter module will be configured to duty mode instead of positive detector mode. From Figure 8, we know that the value stored in the CTR register's CTRMODE bit field has to be either %00110 (single-ended) or %00111 (differential). Then, the APIN (and optionally BPIN) bit fields have to be set to the I/O pins that will transmit the duty signals.

Figure 9: CTRA/B Register Map from CTR.spin

bits	31	30..26	25..23	22..15	14..9	8..6	5..0
Name	—	CTRMODE	PLLDIV	———	BPIN	——	APIN

The RC decay application set the FRQ register to 1, and the result was that 1 got added to PHS for every clock tick in which the pin being monitored was high. In this application, the FRQ register gets set to values that control the high time of the duty signal applied to an I/O pin. There is no condition for adding with duty mode; FRQ gets added to PHS every clock tick.

Setting up a Duty Signal

Here are the steps for setting a duty signal with a counter:

(1) Set the CTR register's CTRMODE bit field to choose duty mode.
(2) Set the CTR register's APIN bit field to choose the pin.
(3) (Optional) set the CTR register's BPIN field if CTRMODE is differential.
(4) Set the I/O pin(s) to output.
(5) Set the FRQ register to a value that gives you the percent duty signal you want.

Example – Send a 25% single-ended duty signal to P4 Using Counter A.

(1) Set the CTR register's CTRMODE bit field to choose duty mode. Remember that bits 30..26 of the CTR register (shown in Figure 9) have to be set to the bit pattern selected from the CTRMODE list in Figure 8. For example, here's a command that configures the counter module to operate in single-ended duty mode:

```
    ctra[30..26] := %00110
```

(2) Set the CTR register's APIN bit field to choose the pin. Figure 9 indicates that APIN is bits 5..0 in the CTR register. Here's an example that sets the `ctra` register's APIN bits to 4, which will control the green LED connected to P4.

```
    ctra[5..0] := 4
```

We'll skip step (3) since the counter module is getting configured to single-ended duty mode and move on to:

(4) Set the I/O pin(s) to output.

```
    dira[4]~~
```

(5) set the FRQ register to a value that gives you the duty signal you want. For ¼ brightness, use 25% duty. So, set the `frqa` register to 1_073_741_824 (calculated earlier).

```
    frqa := 1_073_741_824
```

Tips for Setting Duty

Since the special purpose registers initialize to zero, `frqa` is 0, so 0 is repeatedly added to the PHS register, resulting on no LED state changes. As soon as the program sets the FRQ register to some fraction of 2^{32}, the I/O pin, and the LED, will start sending the duty signal.

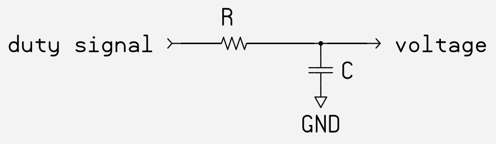
Having 2^{32} different LED brightness levels isn't really practical, but 256 different levels will work nicely. One simple way to accomplish that is by declaring a constant that's $2^{32} \div 256$.

```
CON

  scale = 16_777_216                          ' 2^32 ÷ 256
```

Now, the program can multiply the `scale` constant by a value from 0 to 255 to get 256 different LED brightness levels. Now, if you want ¼ brightness, multiply `scale` by ¼ of 256:

```
  frqa := 64 * scale
```

Time Varying D/A and Filtering: When modulating the value of `frqa` to send time varying signals, an RC circuit typically filters the duty signal. It's better to use a smaller fraction of the useable duty signal range, say 25% to 75% or 12.5% to 87.5%. By keeping the duty in this middle range, the D/A will be less noisy and smaller resistor R and capacitor C values can be used for faster responses. This is especially important for signals that vary quickly, like audio signals, which will be introduced in a different lab.



Duty Code Example

The LedDutySweep object demonstrates the steps for configuring a counter for duty mode and transmitting a duty signal with an I/O pin. It also sweeps a `duty` variable from 0 to 255 repeatedly, causing the P4 LED to gradually increase in brightness and then turn off.

- ✓ Load the LedDutySweep object into the Propeller chip and observe the effect.

```
''LedDutySweep.spin
''Cycle P4 LED from off, gradually brighter, full brightness.


CON

  scale = 16_777_216                          ' 2³²÷ 256

PUB TestDuty | pin, duty, mode

  'Configure counter module.

  ctra[30..26] := %00110                      ' Set ctra to DUTY mode
  ctra[5..0] := 4                             ' Set ctra's APIN
  frqa := duty * scale                        ' Set frqa register

  'Use counter to take LED from off to gradually brighter, repeating at 2 Hz.

  dira[4]~~                                   ' Set P5 to output

  repeat                                      ' Repeat indefinitely
    repeat duty from 0 to 255                 ' Sweep duty from 0 to 255
      frqa := duty * scale                    ' Update frqa register
      waitcnt(clkfreq/128 + cnt)              ' Delay for 1/128th s
```

Duty – Single Ended vs. Differential Modes

The LedDutySweep object uses the single-ended version of a counter module's duty mode. Differential is a second option for duty and several other counter modes. Differential signals are useful for getting signals across longer transmission lines, and are used in wired Ethernet, RS485, and certain audio signals.

When a counter module functions in differential mode, it uses one I/O pin to transmit the same signal that single-ended transmits, along with a second I/O pin that transmits the opposite polarity signal. For example, a counter module set to duty differential mode can send the opposite signal that P4 transmits on P5 or any other I/O pin. Whenever the signal on P4 is high, the signal on P5 is low, and visa versa. Try modifying a copy of LedDutySweep.spin so that it sends the differential signal on P5. Then, as the P4 LED gets brighter, the P5 LED will get dimmer. Here are the steps:

- ✓ Save a copy of the LedDutySweep object that you will modify.
- ✓ To set the counter module for "DUTY differential" mode, change `ctra[30..26] := %00110` to `ctra[30..26] := %00111`.
- ✓ Set the `ctra` module's BPIN bit field by adding the command `ctra[14..9] := 5`
- ✓ Set P5 to output so that the signal gets transmitted by the I/O pin with the command `dira[5]~~`.

Using Both A and B Counter Modules

Using both counter modules to display different LED brightnesses is also a worthwhile exercise. To get two counter modules sending duty signals on separate pins, try these steps:

- ✓ Save another copy of the original, unmodified LedDutySweep object.
- ✓ Add `ctrb[30..26] := %00110`.
- ✓ Assuming `ctrb` will control P6, add `ctrb[5..0] := 6`.
- ✓ Also assuming `ctrb` will control P6, add `dira[6]~~`.
- ✓ In the `repeat duty from 0 to 255` loop, make `frqb` twice the value of `frqa` with the command `frqb := 2 * frqa`. This will cause the P6 LED to get bright twice as fast as the P4 LED.

Inside Duty Mode

Let's take a closer look at how this works by examining the 3-bit version. Since the denominator of the fraction is 2 raised to the number of bits in the register, a 3-bit version of FRQ would be divided by $2^3 = 8$:

$$\text{duty} = \frac{\text{pin high time}}{\text{time}} = \frac{\text{frq}}{8} \qquad \text{(3-bit example)}$$

Let's say the carry bit needs to be high $^3/_8$ of the time. The 3-bit version of the FRQ register would have to store 3. The example below performs eight additions of 3-bit-FRQ to 3-bit-PHS using long-hand addition. The carry bit (that would get carried into bit-4) is highlighted with the ↓ symbol whenever it's 1. Notice that out of eight PHS = PHS + FRQ additions, three result in set carry bits. So, the carry bit is in fact set $^3/_8$ of the time.

```
carry flag set                          ↓                       ↓              ↓
                            11         11       11            111     1      111

3-bit frq             011    011     011     011    011    011    011    011
3-bit phs(previous)  +000   +011    +110    +001   +100   +111   +010   +101
                     ----   ----    ----    ----   ----   ----   ----   ----
3-bit phs(result)     011    110     001     100    111    010    101    000
```

Binary Addition works just like decimal addition when it's done "long hand". Instead of carrying a digit from 1 to 9 when digits in a particular column add up to a value greater than 9, binary addition carries a 1 if the result in a column exceeds 1.

		Binary Result	
0 + 0	=	0	
0 + 1	=	1	
1 + 0	=	1	
1 + 1	=	10	(0, carry the 1)
1 + 1 + 1	=	11	(1, carry the 1)

Special Purpose Registers

Each cog has a special purpose register array whose elements can be accessed with `spr[`*index*`]`. The index value lets you pick a given special purpose register. For example, you can set the value of `ctra` by assigning a value to `spr[8]`, or `ctrb` by assigning a value to `spr[9]`. Likewise, you can assign values to `frqa` and `frqb` by assigning values to `spr[10]` and `spr[11]`, or `phsa` and `phsb` by assigning values to `spr[12]` and `spr[13]`. A full list of the `spr` array elements can be found in the Propeller Manual.

- ✓ Look up `SPR` in the Spin Language reference section of the Propeller Manual, and review the `SPR` explanation and table of SPR array elements.

The advantage to using SPR array elements is that they are accessible by index values. Also, ctrb, frqb, and phsb are all one array element above ctra, frqa, and phsa. This makes it possible to choose between A and B counter registers by simply adding 1 to (or subtracting 1 from) the index value used to access a given SPR register. This in turn makes it possible to eliminate condition statements for deciding which counter module to use and it also makes it possible to initialize and update counter modules within looping structures.

One drawback to special purpose registers is that they are not bit-addressable. For example, the commands ctra[30..26] := %00110 and ctra[5..0] := 4 have to be coded differently for spr[8], which is the ctra special purpose array element. The most convenient way to accomplish these two commands in Spin language with the SPR array is like this:

```
    spr[8] := (%00110 << 26) + 4
```

In the command above, the bit pattern %00110 is shifted left by 26 bits, which accomplishes the same thing as ctra[30..26] := %00110, and adding 4 to it without any shifting has the same effect as ctra[5..0] := 4. Here is the equivalent addition:

```
   %00110 << 26        %00011000000000000000000000000000
   + 4                 %00000000000000000000000000000100
   ────────────        ─────────────────────────────────
   spr[8]              %00011000000000000000000000000100
```

Let's say that the application will send duty signals on P4 and P5. A loop that could set up these I/O pins for duty signals might look like this:

```
  repeat module from 0 to 1                        ' 0 is A module, 1 is B.
    spr[8 + module] := (%00110 << 26) + (4 + module)
    dira[4 + module]~~
```

The first time through the loop, module is 0, so the value 4 gets stored in bits 5..0 of spr[8] and dira[4 + module]~~ becomes dira[4]~~. The second time through the loop, module is 1, so 5 gets stored in bits 4..0 of spr[9], and dira[4 + module]~~ becomes dira[5]~~.

When using counters in objects, the pins will probably get passed as parameters. If the parameters hold the pin values, they might not be contiguous or linked by some mathematical relationship. A handy way to keep a list of non-contiguous pins if you're not expecting them to come from elsewhere would be a lookup or lookupz command. Given an index value, both lookup and lookupz return an element in a list. For example the command value := lookup(index: 7, 11, 13, 1) will store 7 in value if index is 1, 11 in value if index is 2, and so on. If index exceeds the length of the lookup table, the lookup command stores 0 in value. The same command with lookupz will store 7 in value if index is 0, or 11 in value if index is 1, and so on. Like lookup, lookupz returns 0 if index exceeds the list length.

Here is a version of the repeat loop that uses lookupz to store a list of non-contiguous pins and load them into the 5..0 bits of the cog's A and B CTR special purpose registers (spr[8] and spr[9]). Notice how the lookupz command stores 4 and 6. The first time through the loop, module is 0, so 4 gets stored in apin, which in turn gets stored in bits 5..0 of spr[8] and sets bit 4 in the dira register. The second time through the loop, module is 1, so 6 gets stored in apin, which in turn gets stored in bits 5..0 of spr[9] and bit 6 of dira gets set.

```
  repeat module from 0 to 1                        ' 0 is A module, 1 is B.
    apin := lookupz (module: 4, 6)
    spr[8 + module] := (%00110 << 26) + (apin)
    dira[apin]~~
```

The LedSweepWithSpr object does the same job as the LedDutySweep code you modified in the "Using Both A and B Counter Modules" section. The difference is that it performs all counter module operations using the SPR array instead of referring to the A and B module's CTR, FRQ and PHS registers.

- ✓ Compare your copy of LedDutySweep that sweeps both counters against the code in LedSweepWithSpr.
- ✓ Run LedSweepWithSpr and use the LEDs to verify that it controls two separate duty signals.

```
''LedSweepWithSpr.spin
''Cycle P4 and P5 LEDs through off, gradually brighter, brightest at different rates.

CON

  scale = 16_777_216                                ' 2³²÷ 256

PUB TestDuty | apin, duty[2], module

  'Configure both counter modules with a repeat loop that indexes SPR elements.

  repeat module from 0 to 1                         ' 0 is A module, 1 is B.
    apin := lookupz (module: 4, 6)
    spr[8 + module] := (%00110 << 26) + apin
    dira[apin]~~

  'Repeat duty sweep indefinitely.

  repeat
    repeat duty from 0 to 255                       ' Sweep duty from 0 to 255
      duty[1] := duty[0] * 2                        ' duty[1] twice as fast
      repeat module from 0 to 1
        spr[10 + module] := duty[module] * scale    ' Update frqa register
      waitcnt(clkfreq/128 + cnt)                    ' Delay for 1/128th s
```

Modifying LedSweepWithSpr for Differential Signals

Try updating the LedSweepWithSpr object so that it does two differential signals, one on P4 and P5, and the other on P6 and P7.

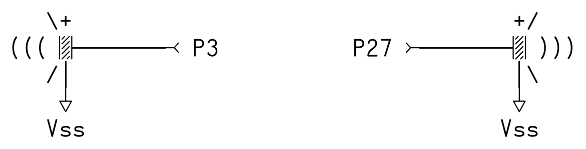
- ✓ Make a copy of LedSweepWithSpr.spin.
- ✓ Add a `bpin` variable to the `TestDuty` method's local variable list.
- ✓ Add the command `bpin := lookupz(module: 5, 7)` just below the command that assigns the `apin` value with a `lookup` command.
- ✓ Change `spr[8 + module] := (%00110 << 26) + apin` to `spr[8 + module] := (%00111 << 26) + (bpin <<9) + apin`.
- ✓ Add `dira[bpin]~~` immediately after `dira[apin]~~`.
- ✓ Load the modified copy of LedSweepWithSpr.spin into the Propeller chip and verify that it sends two differential duty signals.

Generating Piezospeaker Tones with NCO Mode

NCO stands for *numerically controlled oscillator*. If a counter module is configured for the single-ended version of this mode, it will make an I/O pin send a square wave. Assuming `clkfreq` remains constant, the frequency of this square wave is "numerically controlled" by a value stored in a given cog's counter module's FRQ register.

✓ Assemble the parts list and build the schematic shown in Figure 10.

Figure 10: Audio Range NCO Parts List and Circuits

Parts List	Schematic
(2) Piezospeakers (misc) Jumper wires	Piezospeakers 

Counter Module in NCO Mode

When configured to single-ended NCO mode, the counter module does two things:

- The FRQ register gets added to the PHS register every clock tick.
- Bit 31 of the PHS register controls the state of an I/O pin.

When bit 31 of the PHS register is 1, the I/O pin it controls sends a high signal, and when it is 0, it sends a low signal. If `clkfreq` remains the same, the fact that FRQ gets added to PHS every clock tick determines the rate at which the PHS register's bit 31 toggles. This in turn determines the square wave frequency transmitted by the pin controlled by bit 31 of the PHS register.

Given the system clock frequency and an NCO frequency that you want the Propeller to transmit, you can calculate the necessary FRQ register value with this equation:

$$\text{FRQ register} = \text{PHS bit 31 frequency} \times \frac{2^{32}}{\text{clkfreq}} \qquad \textbf{Eq. 2}$$

Example:
What value does `frqa` have to store to make the counter module transmit a 2093 Hz square wave if the system clock is running at 80 MHz? (If this were a sine wave, it would be a C7, a C note in the 7th octave.)

For the solution, start with Eq. 2. Substitute 80_000_000 for `clkfreq` and 2093 for `frequency`.

```
frqa = 2_093 × 2^32 ÷ 80_000_000
frqa = 2093 × 53.687
frqa = 112_367
```

Table 1 shows other notes in the 6th octave and their FRQ register values at 80 MHz. The sharp notes are for you to calculate. Keep in mind that these are the square wave versions. In another lab, we'll use objects that digitally synthesize sine waves for truer tones.

Table 1: Notes, Frequencies, and FRQA/B Register Values for 80 MHz

Note	Frequency (Hz)	FRQA/B Register	Note	Frequency (Hz)	FRQA/B Register
C6	1046.5	56_184	G6	1568.0	84_181
C6#	1107.8		G6#	1661.2	
D6	1174.7	63_066	A6	1760.0	94_489
D6#	1244.5		A6#	1864.7	
E6	1318.5	70_786	B6	1975.5	105_629
F6	1396.9	74_995	C7	2093.0	112_367
F6#	1480.0				

Eq. 3 can also be rearranged to figure out what frequency gets transmitted by an object given a value the object stores in its FRQ register:

$$\text{PHS bit 31 frequency} = \frac{\text{clkfreq} \times \text{FRQ register}}{2^{32}} \qquad \textbf{Eq. 3}$$

Example: An object has its cog's Counter B operating in single-ended NCO mode, and it stores 70_786 in its `frqb` register. The system clock runs at 80 MHz. What frequency does it transmit?

We already know the answer from Table 1, but here it is with Eq. 3

$$\text{PHS bit 31 frequency} = \frac{80\_000\_000 \times 70\_786}{2^{32}} = 1318 \text{ Hz}$$

Configuring a Counter Module for NCO Mode

Figure 11 shows the NCO mode entries in the CTR object's Counter Mode table. Note that it is called NCO/PWM mode in the table, you may see that occasionally. PWM is actually an application of NCO mode that will be explored in the PWM section on page 36. Like DUTY mode, NCO mode has a single-ended and differential options. Single-ended causes a signal that matches bit 31 of the PHS register to be transmitted by the APIN. Differential mode sends the same signal on APIN along with an inverted version of that signal on BPIN.

Figure 11: NCO Excerpts from the CTR Object's Counter Mode Table

CTRMODE	Description	Accumulate FRQ to PHS	APIN output*	BPIN output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
. . .				
%00100	NCO/PWM single-ended	1	PHS[31]	0
%00101	NCO/PWM differential	1	PHS[31]	!PHS[31]
. . .				
%11111	LOGIC always	1	0	0

```
* must set corresponding DIR bit to affect pin
```

A^1 = APIN input delayed by 1 clock
A^2 = APIN input delayed by 2 clocks
B^1 = BPIN input delayed by 1 clock

The steps for configuring the counter module for NCO mode are similar to the steps for DUTY mode. The CTR register's CTRMODE, APIN (and BPIN in differential mode) bit fields have to be set. Then, the FRQ register gets a value that sets the NCO frequency. As with other output examples, the I/O pins used by the counter module have to be set to output.

Remember that the steps for configuring the counter module are similar to the previous two modes (POS detector and DUTY) but the functional result is again very different. In DUTY mode, the phase adder's carry flag ("bit 32" of the PHS register) determined the I/O pin's state, which in turn resulted in a duty signal that varied with the value stored by the FRQ register. In NCO mode, bit 31 of the PHS register controls the I/O pin, which results in a square wave whose frequency is determined by the value stored in the FRQ register.

Here are the steps for configuring a counter module to NCO mode.

(1) Configure the CTRA/B register
(2) Set the FRQA/B register
(3) Set the I/O pin to output

1) Configure the CTRA/B register:

Here is an example that sets Counter A to "NCO single-ended" mode, with the signal transmitted on P27. To do this, set `ctra[30..26]` to %00100, and `ctra[5..0]` to 27.

```
ctra[30..26] := %00100
ctra[5..0] := 27
```

2) Set the FRQA/B register:

Here is an example for the square wave version of the C7 note:

```
frqa := 112_367
```

3) Set the I/O pin to output:

Since it's P27 that's sending the signal, make it an output:

```
dira[27]~~
```

After starting the counter module, it runs independently. The code in the cog can forget about it and do other things, or monitor/control/modify the counter's behavior as needed.

Square Wave Example

The SquareWaveTest object below plays the square wave version of C in the 7th octave for 1 second.

- ✓ Examine the SquareWaveTest object and compare it to steps 1 through 4 just discussed.
- ✓ Load the SquareWaveTest object into the Propeller chip. Run it and verify that it plays a tone.
- ✓ Change `frqa := 112_367` to `frqa := 224_734`. That'll be C8, the C note in the next higher octave.
- ✓ Load the modified object into the Propeller chip. This time, the note should play at a higher pitch.

```
''SquareWaveTest.spin
''Send 2093 Hz square wave to P27 for 1 s with counter module.

CON

  _clkmode = xtal1 + pll16x            ' Set up clkfreq = 80 MHz.
  _xinfreq = 5_000_000

PUB TestFrequency

  'Configure ctra module
  ctra[30..26] := %00100               ' Set ctra for "NCO single-ended"
  ctra[5..0] := 27                     ' Set APIN to P27
  frqa := 112_367                      ' Set frqa for 2093 Hz (C7 note) using:
                                       ' FRQA/B = frequency × (232 ÷ clkfreq)
  'Broadcast the signal for 1 s
  dira[27]~~                           ' Set P27 to output
  waitcnt(clkfreq + cnt)               ' Wait for tone to play for 1 s
```

Stopping (and restarting) the Signal

In the SquareWaveTest object, the cog runs out of commands, so the tone stops because the program ends. In many cases, you will want to stop and restart the signal. The three simplest ways to stop (and resume) signal transmission are:

(1) ***Change the Direction of the I/O pin to input.*** In the SquareWaveTest object, this could be done with either `dira[27] := 0` or `dira[27]~` when the program is ready to stop the signal. (To restart the signal, use either `dira[27] := 1` or `dira[27]~~`.)

(2) ***Stop the counter module by clearing CTR bits 30..26.*** In the SquareWaveTest object, this can be accomplished with `ctra[30..26] := 0`. Another way to do it is by setting all the bits in the `ctra` register's CTRMODE bitfield to zero with `ctra[30..26]~`. In either case, the I/O pin is still an output, and its output state might be high or low. Later, we'll examine a way to make sure the signal ends when the I/O pin is transmitting a low signal. (To restart the signal, copy %00100 back into `ctra[30..26]`.)

(3) ***Stop adding to PHS by setting FRQ to 0.*** In the SquareWaveTest object, this could be done with either `frqa := 0` or `frqa~`. The counter would keep running, but since it would add zero to `phsa` with each clock tick, bit 31 of `phsa` wouldn't change, so the I/O pin would also stop toggling. Like stopping the counter, the I/O pin would hold whatever output state it had at the instant `frqa` is cleared. (To restart the signal, use `frqa := 112_367`.)

The Staccato object toggles the I/O pin between output and input to cause the 2.093 kHz tone to start and stop at 15 Hz for 1 s. It uses approach (1) for stopping and restarting the signal. Your job will be to modify two different copies of the code to use approaches 2 and 3.

- ✓ Load Staccato.spin into the Propeller chip and verify that it chirps at 15 Hz for 1 s.
- ✓ Make two copies of the program.
- ✓ Modify one copy so that it uses approach 2 for starting and stopping the signal.
- ✓ Modify the other copy so that it uses approach 3 for starting and stopping the signal.

```
''Staccato.spin
''Send 2093 Hz beeps in rapid succession (15 Hz for 1 s).

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestFrequency

  'Configure ctra module
  ctra[30..26] := %00100                     ' Set ctra for "NCO single-ended"
  ctra[8..0] := 27                           ' Set APIN to P27
  frqa := 112_367                            ' Set frqa for 2093 Hz (C7 note):

  'Ten beeps on/off cycles in 1 second.
  repeat 30
    !dira[27]                                 ' Set P27 to output
    waitcnt(clkfreq/30 + cnt)                 ' Wait for tone to play for 1 s

  'Program ends, which also stops the counter module.
```

Use F10 and F11 to easily compare programs:

It is convenient to put the original Staccato.spin into the EEPROM with F11, then use F10 when you test your modifications. After running your new program, you can then press and release the PE Platform's reset button to get an instant audio comparison.

Playing a List of Notes

DoReMi.spin is an example where the counter module is used to play a series of notes. Since it isn't needed for anything else in the meantime, the I/O pin that sends the square wave signal to the piezospeaker is set to input during the ¼ stops between notes. bit 31 of the `phsa` register still toggles at a given frequency during the quarter stop, but the pseudo-note doesn't play.

The `frqa` register values are stored in a `DAT` block with the directive:

```
DAT
  ...
  notes long 112_367, 126_127, 141_572, 149_948, 168_363, 188_979, 212_123, 224_734
```

A `repeat` loop that sweeps a variable named `index` from 0 to 7 is used to retrieve and copy each of these notes to the `frqa` register. The loop copies each successive value from the notes sequence into the `frqa` register with this command:

```
  repeat index from 0 to 7
    'Set the frequency.
    frqa := long[@notes][index]
    ...
```

✓ Load the DoReMi object into the Propeller chip and observe the effect.

```
''DoReMi.spin
''Play C6, D6, E6, F6, G6, A6, B6, C7 as quarter notes quarter stops between.

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestFrequency | index

  'Configure ctra module
  ctra[30..26] := %00100                     ' Set ctra for "NCO single-ended"
  ctra[8..0] := 27                           ' Set APIN to P27
  frqa := 0                                  ' Don't play any notes yet

  repeat index from 0 to 7


    frqa := long[@notes][index]              'Set the frequency.

    'Broadcast the signal for 1/4 s
    dira[27]~~                               ' Set P27 to output
    waitcnt(clkfreq/4 + cnt)                 ' Wait for tone to play for 1/4 s


    dira[27]~                                '1/4 s stop
    waitcnt(clkfreq/4 + cnt)

DAT
'80 MHz frqa values for square wave musical note approximations with the counter module
'configured to NCO:
'          C6      D6      E6      F6      G6      A6      B6       C7
notes long 56_184, 63_066, 70_786, 74_995, 84_181, 94_489, 105_629, 112_528
```

Counter NCO Mode Example with bit 3 Instead of bit 31

In NCO mode, the I/O pin's output state is controlled by bit 31 of the PHS register. However, the on/off frequency for any bit in a variable or register can be calculated using Eq. 4 and assuming a value is repeatedly added to it at a given rate:

$$\text{frequency} = (\text{value} \times \text{rate}) \div 2^{\text{bit} + 1} \qquad \textbf{Eq. 4}$$

Here is an example that can be done on scratch paper that may help clarify how this works.

bit 3 Example
At what frequency does bit 3 in a variable toggle if you add 4 to it eight times every second?

Value is 4, rate is 8 Hz, and bit is 3, so

$$
\begin{aligned}
\texttt{frequency} &= (\texttt{value} \times \texttt{rate}) \div 2^{\texttt{bit + 1}} \\
&= (4 \times 8\ \texttt{Hz}) \div 2^{3\ +\ 1} \\
&= 32\ \texttt{Hz} \div 16 \\
&= 2\ \texttt{Hz}
\end{aligned}
$$

Table 2 shows how this works. Each 1/8 second, the value 4 gets added to a variable. As a result, bit 3 of the variable gets toggled twice every second, i.e. at 2 Hz.

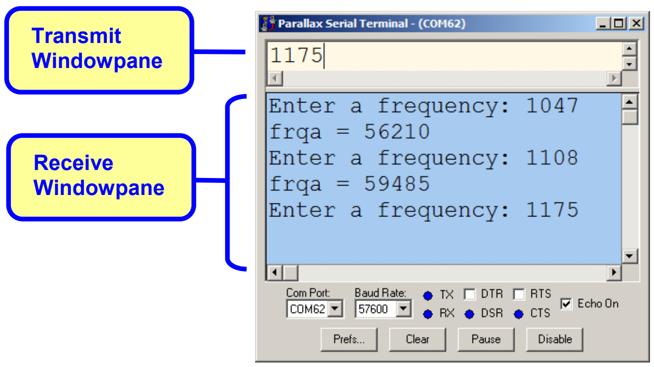
Table 2: Bit 3 Example

Time (s)	Value	Variable	Bit 3 in Variable							
			7	6	5	4	3	2	1	0
0.000		0	0	0	0	0	0	0	0	0
0.125	4	4	0	0	0	0	0	1	0	0
0.250	4	8	0	0	0	0	1	0	0	0
0.375	4	12	0	0	0	0	1	1	0	0
0.500	4	16	0	0	0	1	0	0	0	0
0.625	4	20	0	0	0	1	0	1	0	0
0.750	4	24	0	0	0	1	1	0	0	0
0.875	4	28	0	0	0	1	1	1	0	0
1.000	4	32	0	0	1	0	0	0	0	0
1.125	4	36	0	0	1	0	0	1	0	0
1.250	4	40	0	0	1	0	1	0	0	0
1.375	4	44	0	0	1	0	1	1	0	0
1.500	4	48	0	0	1	1	0	0	0	0
1.625	4	52	0	0	1	1	0	1	0	0
1.750	4	56	0	0	1	1	1	0	0	0
1.875	4	60	0	0	1	1	1	1	0	0

NCO FRQ Calculator Method

The TerminalFrequencies object allows you to enter square wave frequencies into Parallax Serial Terminal, and it calculates and displays the FRQ register value and plays the tone on the P27 piezospeaker. (See Figure 12.) The object's NcoFrqReg method is an adaptation of the Propeller Library CTR object's `fraction` method. Given a square wave frequency, it calculates `frqReg` = frequency × (2^{32} ÷ `clkfreq`), and returns `frqReg`. So, for a given square wave frequency simply set the FRQ register equal to the result returned by the `NcoFrqReg` method call.

Figure 12: Calculating frqa Given a Frequency in Hz



```
1175
```

```
Enter a frequency: 1047
frqa = 56210
Enter a frequency: 1108
frqa = 59485
Enter a frequency: 1175
```

Com Port: COM62 Baud Rate: 57600 [✓ Echo On]

The NcoFrqReg method uses a binary calculation approach to come up with frqReg = frequency × (2^{32} ÷ clkfreq). It would also have been possible to use the FloatMath library to perform these calculations. However, the NcoFrqReg method takes much less code space than the FloatMath library. It also takes less time to complete the calculation, so it makes a good candidate for a counter math object.

✓ Use the Propeller Tool to load TerminalFrequencies.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button. (Remember, you don't even have to wait for the program to finish loading.)
✓ When prompted, enter the integer portion of each frequency value (not the FRQ register values) from Table 1 on page 18 into the Parallax Serial Terminal's Transmit Windowpane, shown in Figure 12 on page 23.
✓ Verify that the NcoFrqReg method's calculations match the calculated FRQ register values in the table.
✓ Remember to click Parallax Serial Terminal's Disconnect button before loading the next program.

```
''TerminalFrequencies.spin
''Enter frequencies to play on the piezospeaker and display the frq register values
''with Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

  CLS = 16, CR = 13                          ' Parallax Serial Terminal control characters

OBJ

  Debug   : "FullDuplexSerialPlus"           ' Parallax Serial Terminal display object


PUB Init

  'Configure ctra module.
  ctra[30..26] := %00100                     ' Set ctra for "NCO single-ended"
  ctra[8..0] := 27                           ' Set APIN to P27
  frqa := 0                                  ' Don't send a tone yet.
  dira[27]~~                                 ' I/O pin to output

  'Start FullDuplexSerialPlus and clear the Parallax Serial Terminal.
  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)
  Debug.tx(CLS)

  Main


PUB Main | frequency, temp

  repeat

    Debug.Str(String("Enter a frequency: "))
    frequency := Debug.getDec
    temp := NcoFrqReg(frequency)
    Debug.Str(String("frqa = "))
    Debug.Dec(temp)
    Debug.tx(CR)
    'Broadcast the signal for 1 s
    frqa := temp
    waitcnt(clkfreq + cnt)
    frqa~


PUB NcoFrqReg(frequency) : frqReg
{{
Returns frqReg = frequency × (2³² ÷ clkfreq) calculated with binary long
division.  This is faster than the floating point library, and takes less
code space.  This method is an adaptation of the CTR object's fraction
method.
}}

  repeat 33
    frqReg <<= 1
    if frequency => clkfreq
      frequency -= clkfreq
      frqReg++
    frequency <<= 1
```

Use Two Counter Modules to Play Two Notes

The TwoTones object demonstrates how both counters can be used to play two different square wave tones on separate speakers. In this example, all the program does is wait for certain amounts of time to pass before adjusting the `frqa` and `frqb` register values. The program could also perform a number of other tasks before coming back and waiting for the CLK register to get to the next time increment.

- ✓ Load the TwoTones object into the Propeller chip.
- ✓ Verify that it plays the square wave approximation of C6 on the P27 piezospeaker for 1 s, then pauses for ½ s, then plays E6 on the P2 piezospeaker, then pauses for another ½ s, then plays both notes on both speakers at the same time.

```
''TwoTones.spin
''Play individual notes with each piezospeaker, then play notes with both at the
''same time.

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000


OBJ

  SqrWave : "SquareWave"


PUB PlayTones | index, pin, duration

  'Initialize counter modules
  repeat index from 0 to 1
    pin := byte[@pins][index]
    spr[8 + index] := (%00100 << 26) + pin
    dira[pin]~~

  'Look tones and durations in DAT section and play them.
  repeat index from 0 to 4
    frqa := SqrWave.NcoFrqReg(word[@Anotes][index])
    frqb := SqrWave.NcoFrqReg(word[@Bnotes][index])
    duration := clkfreq/(byte[@durations][index])
    waitcnt(duration + cnt)
```

```
DAT
pins       byte  27, 3

'index           0        1        2        3        4
durations  byte  1,       2,       1,       2,       1
anotes     word  1047,    0,       0,       0,       1047
bnotes     word  0,       0,       1319,    0,       1319
```

Inside TwoTones.spin

The TwoTones object declares the SquareWave object (see Appendix A) in its `OBJ` block and gives it the nickname `SqrWave`. This object has a method with the same name and function as `NcoFrqReg` in the TerminalFrequencies object, but the coding relies on methods adapted from the Propeller Library's CTR object to perform the calculation.

The first `repeat` loop in the `PlayTones` method initializes the `counter` method by setting `SPR` array elements 8 and 9, which are the `ctra` and `ctrb` registers. The `index` variable in that loop is also used to look up the pin numbers listed in the `DAT` block's `Pins` sequence using `pin := byte[@pin][index]`. The second `repeat` loop looks up elements in the `DAT` block's `durations`, `anotes` and `bnotes` sequences. Each sequence has five elements, so the `repeat` loop indexes from 0 to 4 to fetch each element in each sequence.

Take a look at the command `frqa := SquareWave.NcoFrqReg(word[@Anotes][index])` in the TwoTones object's second `repeat` loop. First, `word[@Anotes][index]` returns the value that's `index` elements to the right of the `anotes` label. The first time through the loop, `index` is 0, so it returns 1047. The second, third and fourth time through the loop, `index` is 1, then 2, then 3. It returns 0 each time. The fifth time through the loop, `index` is 4, so it returns 1047 again. Each of these values returned by `word[@Anotes][index]` becomes a parameter in the `SquareWave.NcoFrqReg` method call. Finally, the value returned by `SquareWave.NcoFrqReg` gets stored in the `frqa` variable. The result? A given frequency value in the `anotes` sequence gets converted to the correct value for `frqa` to make the counter module play the note.

Counter Control with an Object

If you examined the SquareWave object, you may have noticed that has a `Freq` method that allows you to choose a counter module (0 or 1 for Counter A or Counter B), a pin, and a frequency. The `Freq` method considerably simplifies the TwoTones object.

- ✓ Compare TwoTonesWithSquareWave (below) against the TwoTones object (above).
- ✓ Load TwoTonesWithObject into the Propeller chip and verify that it behaves the same as the TwoTones object.

```
''TwoTonesWithSquareWave.spin
''Play individual notes with each piezospeaker, then play notes with both at the
''same time.

CON

  _clkmode = xtal1 + pll16x               ' System clock → 80 MHz
  _xinfreq = 5_000_000

OBJ

  SqrWave : "SquareWave"
```

```
PUB PlayTones | index, pin, duration

  'Look tones and durations in DAT section and play them.
  repeat index from 0 to 4
    SqrWave.Freq(0, 27, word[@Anotes][index])
    SqrWave.Freq(1, 3, word[@Bnotes][index])
    duration := clkfreq/(byte[@durations][index])
    waitcnt(duration + cnt)

DAT
  pins       byte  27, 3

  'index           0        1        2        3        4
  durations  byte  1,       2,       1,       2,       1
  anotes     word  1047,    0,       0,       0,       1047
  bnotes     word  0,       0,       1319,    0,       1319
```

Applications – IR Object and Distance Detection

When you point your remote at the TV and press a button, the remote flashes an IR LED on/off rapidly to send messages to the IR receiver in the TV. The rate at which the IR LED flashes on/off is matched to a filter inside the TV's IR receiver. Common frequencies are 36.7, 38, 40, and 56.9 kHz. This frequency-and-filter system is used to distinguish IR remote messages from ambient IR such as sunlight and the 120 Hz signal that is broadcast by household lighting.

The wavelength of IR used by remotes is typically in the 800 to 940 nm range.

The remote transmits the information by modulating the IR signal. The amount of time the IR signal is sent can contain information, such as start of message, binary 1, binary 0, etc. By transmitting sequences of signal on/off time, messages for the various buttons on your remote can be completed in just a few milliseconds.

The IR LED and receiver that are used for beaming messages to entertainment system components can also be used for object detection. In this scheme, the IR LED and IR receiver are placed so that the IR LED's light will bounce off an object and return to the IR receiver. The IR LED still has to modulate its light for the IR receiver's pass frequency. If the IR LED's light does reflect off an object and return to the IR receiver, the receiver sends a signal indicating that it is receiving the IR signal. If the IR does not reflect off the object and return to the IR receiver, it sends a signal indicating that it is not receiving IR.

This detection scheme uses very inexpensive parts, and has become increasingly popular in hobby robotics.

The PE Kit's IR receiver shown on the right side of Figure 13 has a 38 kHz filter. A Propeller chip cog's counter module can be used to generate the 38 kHz signal for the IR LED to broadcast for either IR object detection or entertainment system component control. This section of the lab will simply test object detection, but the same principles will apply to remote decoding and entertainment system component control.

✓ Build the circuit shown in Figure 13 – Figure 15, using the photo as a parts placement guide.

Figure 13: IR Detection Parts and Schematic

Parts List

(1) Resistor 1 kΩ
(1) Resistor 10 kΩ
(1) IR LED
(1) IR detector
(1) LED shield
(1) LED standoff
(misc) Jumper wires

Schematic

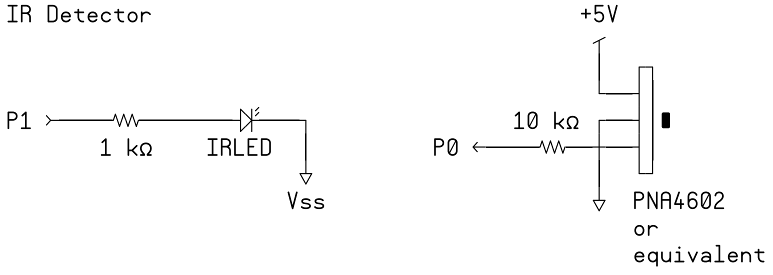
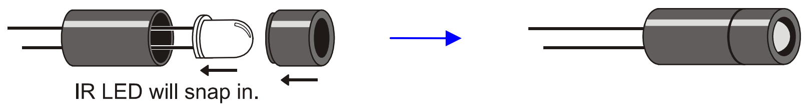


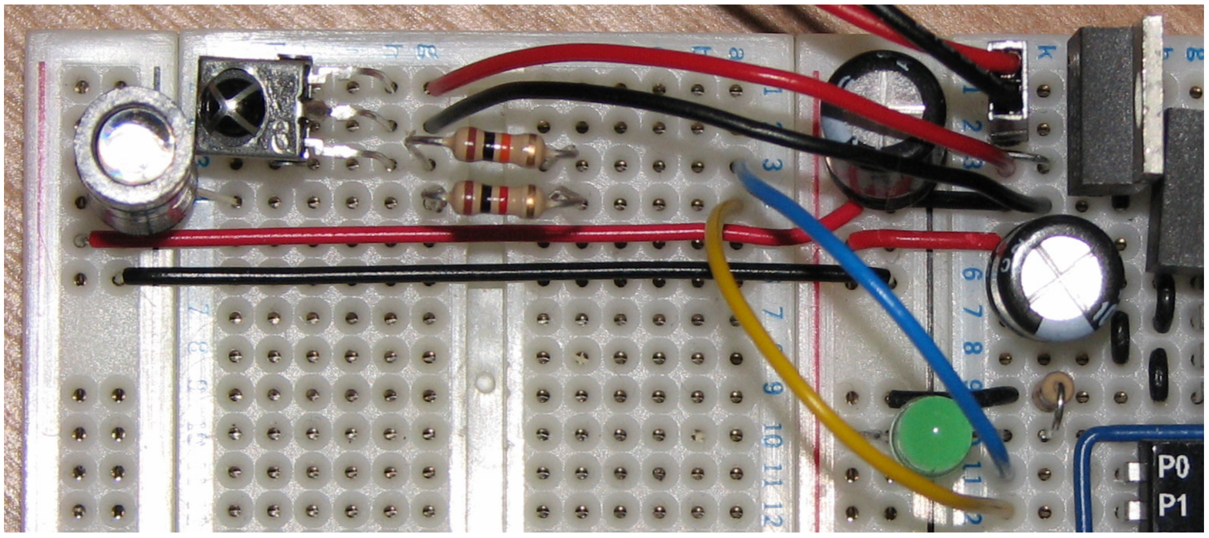
Figure 14 shows how to assemble the IR LED for object detection. First, snap the IR LED into the LED standoff. Then, attach the light shield to the standoff.

Figure 14: IR LED Assembly



A breadboard arrangement that works well for the IR LED and receiver is shown in Figure 15. Notice how the IR receiver's 5 V source is jumpered from the center breadboard's socket (l, 3) to the left breadboard's socket (g, 1). The IR receiver's ground is jumpered from the center breadboard's socket (k, 4) to the left breadboard's (g, 2) socket. The IR LED's shorter cathode pin is connected to the left vertical ground rail (black, 4). A 1 kΩ resistor is in series between the IR LED's anode and P1. A large resistor is important for connecting a 5 V output device to the Propeller chip's 3.3 V input; a 10 kΩ resistor is used between the IR receiver's 5 V output and the Propeller chip's P0 I/O pin. A 1 to 2 kΩ resistor is useful in series with the IR LED to reduce the detection range. A small resistor like 100 Ω can cause phantom detections of far away objects, such as the ceiling.

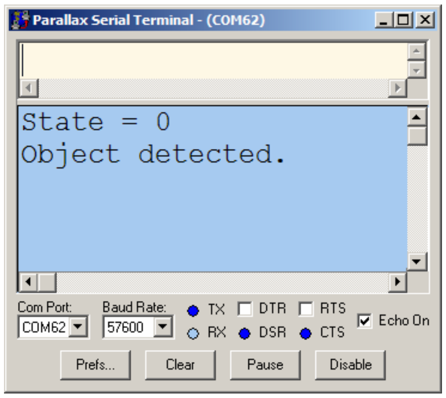
Figure 15: IR LED and Detector Orientation



IR Object Detection with NCO

The IrObjectDetection object sets up the 38 kHz signal using NCO mode. Whenever the I/O pin connected to the IR LED is set to output, the 38 kHz transmits. In a `repeat` loop, the program allows the IR LED to transmit the 38 kHz infrared signal for 1 ms, then it saves `ina[0]` in a variable named `state` and displays it on Parallax Serial Terminal (Figure 16).

Figure 16: Object Detection Display



```
State = 0
Object detected.
```

✓ Use the Propeller Tool to load IrObjectDetection.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button.

✓ The state should be 1 with no obstacles visible, or 0 when you place your hand in front of the IR LED/receiver.

```
'' IrObjectDetection.spin
'' Detect objects with IR LED and receiver and display with Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x                     ' System clock → 80 MHz
  _xinfreq = 5_000_000

  ' Constants for Parallax Serial Terminal.
  HOME = 1, CR = 13, CLS = 16, CRSRX = 14, CLREOL = 11

OBJ

  Debug     : "FullDuplexSerialPlus"
  SqrWave   : "SquareWave"

PUB IrDetect | state

  'Start 38 kHz square wave
  SqrWave.Freq(0, 1, 38000)                     ' 38 kHz signal → P1
  dira[1]~                                      ' Set I/O pin to input when no signal needed

  'Start FullDuplexSerialPlus
  Debug.start(31, 30, 0, 57600)                 ' Start FullDuplexSerialPlus
  waitcnt(clkfreq * 2 + cnt)                    ' Give user time to click Enable.
  Debug.tx(CLS)                                 ' Clear screen

  repeat

    ' Detect object.
    dira[1]~~                                   ' I/O pin → output to transmit 38 kHz
    waitcnt(clkfreq/1000 + cnt)                 ' Wait 1 ms
    state := ina[0]                             ' Store I/R detector output
    dira[1]~                                    ' I/O pin → input to stop signal

    ' Display detection (0 detected, 1 not detected)
    Debug.str(String(HOME, "State = "))
    Debug.Dec(state)
    Debug.str(String(CR, "Object "))
    if state == 1
      Debug.str(String("not "))
    Debug.str(String("detected.", CLREOL))
    waitcnt(clkfreq/10 + cnt)
```

IR Distance Detection with NCO and Duty Sweep

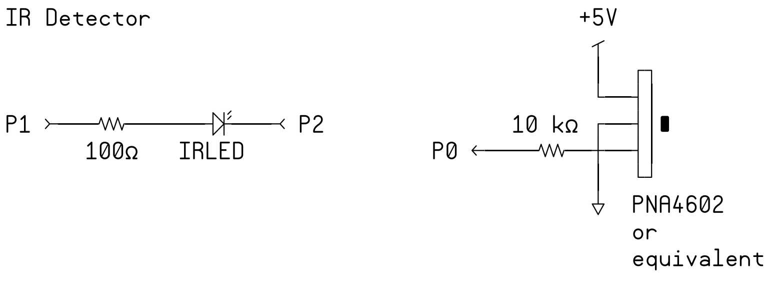
If the IR LED shines more brightly, it makes the detector more far-sighted. If it shines less brightly, it makes it more near-sighted. Recall that a counter module's Duty mode can be used to control LED brightness and even sweep the LED's brightness from dim to bright (see page 9.) This same duty sweep approach can be combined with the NCO signal from the IR object detection example to make the IR LED flash on/off at 38 kHz, sweeping from dim to bright. With each increase in brightness, the IR detector's output can be rechecked in a loop. The number of times the IR detector reported that it detected an object will then be related to the object's distance from the IR LED/detector.

Although the circuit from Figure 13 can be used for distance detection with a combination of NCO and duty signals, the circuit in Figure 17 makes it possible to get better results from the IR receiver. Instead of tying the IR LED's cathode to GND, it is connected to P2. The program can then sweep the voltage applied to IR LED's cathode from 0 to 3.3 V via P2 while the signal from P1 transmits the 38 kHz NCO signal to the anode end of the circuit. Since an LED is a 1-way valve, the low portion of the 38 kHz signal does not get transmitted since it is less than the DC voltage that the duty signal synthesizes on P2. During the high portions of the 38 kHz signal, the increased voltages applied to P2 reduce the voltage across the LED circuit, which in turn reduces its brightness. So, it's the same 38 kHz signal, just successively less bright.

Figure 17: IR Detection Parts and Schematic

```
Parts List

(1) Resistor 100 Ω
(1) Resistor 10 kΩ
(1) IR LED
(1) IR detector
(1) LED shield
(1) LED standoff
(misc) Jumper wires
```

Schematic



The IR Detector object below performs the distance detection just discussed. The parent object has to call the `init` method to tell it which pins are connected to the IR LED circuit's anode and cathode ends and the IR receiver's outputs. When the `distance` method gets called, it uses the duty sweep approach just discussed and the pin numbers that were passed to the `init` method to measure the object's distance.

The IrDetector object's `distance` method uses the SquareWave object to start transmitting the 38 kHz signal to the IR LED circuit's anode end using Counter B. Then, it configures Counter A to duty mode and initializes **`frqa`** and **`phsa`** to 0, which results in an initial low signal to the IR LED circuit's cathode end. Next, a **`repeat`** loop very rapidly sweeps duty from 0/256 to 255/256. With each iteration, the voltage to the IR LED circuit's cathode increases, making the IR LED less bright and the IR detector more nearsighted. Between each duty increment, the loop adds the IR receiver's output to the `dist` return parameter. Since the IR receiver's output is high when it doesn't see reflected IR, `dist` stores the number of times out of 256 that it did not see an object. When the object is closer, this number will be smaller; when it's further, the number will be larger. So, after the loop, when the method returns the `dist` parameter, it contains a representation of the object's distance.

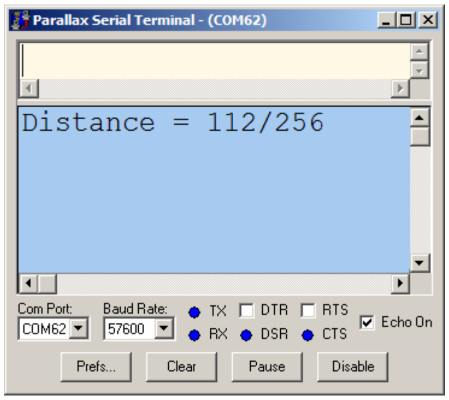
Keep in mind that this distance measurement will vary with the surface reflecting the IR.

For example, if the distance method returns 175, the measured distance for a white sheet of paper might be five times the distance of a sheet of black vinyl. Reason being, the white paper readily reflects infrared, so it will be visible to the receiver much further away. In contrast, black vinyl tends to absorb it, and is only visible at very close ranges.

```
''IrDetector.spin

CON

  scale = 16_777_216                         ' 2³²÷ 256

OBJ

  SquareWave  : "SquareWave"                 ' Import square wave cog object

VAR

  long anode, cathode, recPin, dMax, duty


PUB init(irLedAnode, irLedCathode, irReceiverPin)

  anode := irLedAnode
  cathode := irLedCathode
  recPin := irReceiverPin

PUB distance : dist
{{
Performs a duty sweep response test on the IR LED/receiver and returns dist, a zone value
from 0 (closest) to 256 (no object detected).
}}
  'Start 38 kHz signal.
  SquareWave.Freq(1, anode, 38000)           ' ctrb 38 kHz
  dira[anode]~~

  'Configure Duty signal.
  ctra[30..26] := %00110                     ' Set ctra to DUTY mode
  ctra[5..0] := cathode                      ' Set ctra's APIN
  frqa := phsa := 0                          ' Set frqa register
  dira[cathode]~~                            ' Set P5 to output

  dist := 0

  repeat duty from 0 to 255                  ' Sweep duty from 0 to 255
    frqa := duty * scale                     ' Update frqa register
    waitcnt(clkfreq/128000 + cnt)            ' Delay for 1/128th s
    dist += ina[recPin]                      ' Object not detected?  Add 1 to dist.
```

The TestIrDutyDistanceDetector object gets distance measurements from the IrDetector object and displays them in Parallax Serial Terminal (Figure 18). With the 100 Ω resistor in series with the IR LED, whether or not the system detects your ceiling from table height depends on how high and how reflective your ceiling is and how sensitive your particular detector is. If the system detects no object, it will return 256. Daylight streaming in through nearby windows may introduce some noise in the detector's output, resulting in values slightly less than 256 when no object is detected. As a target object is brought closer to the IR LED/receiver, the measurements will decrease, but not typically to zero unless the IR LED is pointed directly into the IR receiver's phototransistor (the black bubble under the crosshairs).

Figure 18: Distance Detection Display



```
Distance = 112/256
```

- ✓ Make sure IrDetector.spin is saved to the same folder as TestIrDutyDistanceDetector.spin and FullDuplexSerialPlus.spin.
- ✓ Use the Propeller Tool to load TestIrDutyDistanceDetector.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Experiment with a variety of targets and distance tests to get an idea of what such a system might and might not be useful for.

```
'' TestIrDutyDistanceDetector.spin
'' Test distance detection with IrDetector object.

CON

  _xinfreq = 5_000_000
  _clkmode = xtal1 | pll16x

  CLS = 16, CRSRX = 14, CLREOL = 11

OBJ

  ir     : "IrDetector"
  debug  : "FullDuplexSerialPlus"


PUB TestIr | dist

  'Start serial communication, and wait 2 s for Parallax Serial Terminal connection.

  debug.Start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)
  debug.tx(CLS)
  debug.str(string("Distance = "))
  'Configure IR detectors.
  ir.init(1, 2, 0)

  repeat
    'Get and display distance.
    debug.str(string(CRSRX, 11))
    dist := ir.Distance
    debug.dec(dist)
    debug.str(string("/256", CLREOL))
    waitcnt(clkfreq/3 + cnt)
```

Counting Transitions

Counter modules also have positive and negative edge detection modes (see Figure 19). In POSEDGE mode, a counter module will add FRQ to PHS when it detects a transition from low to high on a given I/O pin. NEGEDGE mode makes the addition when it detects a high to low transition. Either can be used for counting the cycles of signals that pass above and then back down below a Propeller I/O pin's 1.65 V logic threshold.

These counter modes can be used to either count the transitions of a signal applied to the I/O pin or the transitions of a signal the I/O pin is transmitting.

Figure 19: Edge Detector Excerpts from the CTR Object's Counter Mode Table

CTRMODE	Description	Accumulate FRQ to PHS	APIN output*	BPIN output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
. . .				
%01010	POSEDGE detector	A^1 & !A^2	0	0
. . .				
%01110	NEGEDGE detector	!A^1 & A^2	0	0
. . .				
%11111	LOGIC always	1	0	0

* must set corresponding DIR bit to affect pin

A^1 = APIN input delayed by 1 clock
A^2 = APIN input delayed by 2 clocks
B^1 = BPIN input delayed by 1 clock

Notice from the notes in the Counter Mode Table excerpt in Figure 19 that the addition of FRQ to PHS occurs one clock cycle after the edge. This could make a difference in some assembly language programs where the timing is tight, but does not have any significant impact on interpreted Spin language programs.

The steps for setting up a counter still involve setting the CTR register's MODE bit field (bits 30..26) and its APIN bit field (bits 5..0) along with setting the FRQ register to the value that should be added to the PHS register when an edge is detected. Before the measurement, they can be set to zero.

```
ctrb[30..26] := %01110
ctrb[8..0] := 27
frqb~
phsb~
```

Here's an example from the next program that demonstrates one way of using NEGEDGE detector to control the duration of a tone played on the piezospeaker. The Counter A module is set to transmit a 2 kHz square wave with NCO mode on the same I/O pin the Counter B register will monitor with NEGEDGE mode. The `frqb` register is set to 1, so that with each negative clock edge, 1 gets added to `frqb`. To play a 2 kHz tone for 1 second, it takes 2000 cycles. The `repeat while phsb < 2000` command only allows the program to move on and clear `frqa` to stop playing the tone after 2000 negative edges have been detected.

```
        frqb := 1
        frqa := SquareWave.NcoFrqReg(2000)

        repeat while phsb < 2000

        frqa~
```

Polling: This example polls the **phsb** register, waiting for the number of transitions to exceed a certain value, but it doesn't necessarily need to poll for the entire 2000 cycles. This will free up the cog to get a few things done while the signal is transmitting and check periodically to find out how close **phsb** is to 2000.

✓ Load CountEdgeTest.spin into the Propeller chip and verify that counting edges can be used to control the duration of the tone.

```
{{
CountEdgeTest.spin
Transmit NCO signal with Counter A
Use Counter B to keep track of the signal's negative edges and stop the signal
after 2000.
}}

CON

  _clkmode = xtal1 + pll16x                    'System clock → 80 MHz
  _xinfreq = 5_000_000

OBJ

  SqrWave      : "SquareWave"

PUB TestFrequency

  ' Configure counter modules.

  ctra[30..26] := %00100                      'ctra module to NCO mode
  ctra[8..0] := 27

  ctrb[30..26] := %01110                      'ctrb module to NEGEDGE detector
  ctrb[8..0] := 27
  frqb~
  phsb~

  'Transmit signal for 2000 NCO signal cycles

  outa[27]~                                   ' P27 → output-low
  dira[27]~~

  frqb := 1                                   ' Start the signal
  frqa := SqrWave.NcoFrqReg(2000)

  repeat while phsb < 2000                    ' Wait for 2 k reps

  frqa~                                       ' Stop the signal
```

Faster Edge Detection

The next example program can stop frequencies up to about 43.9 kHz on the falling clock edge. For controlling the number of pulses delivered by faster signals, an assembly language program will be way more responsive, and can likely detect the falling edge and stop it within a few clock cycles.

BetterCountEdges.spin monitors a 3 kHz signal transmitted by Counter A. Instead of monitoring negative edges, it configures Counter B to monitor positive edges on P27 with `ctrb[30..26] := 01010` and `ctrb[5..0] := 27`. Next, it sets `frqb` to 1 so that 1 gets added to the PHS register with each positive edge. Instead of clearing the PHS register and waiting for 3000 positive edges, it sets `phsb` to -3000. Next, it sets bit 27 in a variable named `a` to 1 with the command `a |< 27`.

- ✓ Look up the bitwise decode `|<` operator in the Propeller Manual.

When the `frqa := SquareWave.CalcFreqReg(3000)` command executes, P27 starts sending a 3 kHz square wave. Since `phsb` is bit-addressable, the command `repeat while phsb[31]` repeats while bit 31 of the `phsb` register is 1. Recall that the highest bit of a variable or register will be 1 so long as the value is negative. So `phsb[31]` will be 1 (non zero) while `phsb` is negative. The `phsb` register will remain negative until `frqb = 1` is added to `phsb` 3000 times.

When the `repeat` loop terminates, the signal is high because it was looking for a positive edge. The goal is to stop the signal after it goes low. The command `waitpeq(0, a, 0)` waits until P27 is zero. The command `waitpeq(0, |< 27, 0)` could also have been used, but the program wouldn't respond as quickly because it would have to first calculate `|< 27`; whereas `waitpeq(0, a, 0)` already has that value calculated and stored in the `a` variable. So the `waitpeq` command allows the program to continue to `frqa~`, which clears the `frqa` register, and stops the signal at output-low after the 3000th cycle.

- ✓ Look up and read about `waitpeq` in the Propeller Manual.
- ✓ Load BetterCountEdges.spin into the Propeller chip and verify that it plays the 3 kHz signal for 1 s.
- ✓ If you have an oscilloscope, set the signal for ten cycles instead of 3000. Then, try increasing the frequency, and look for the maximum frequency that will still deliver only 10 cycles.

```
''BetterCountEdges.spin

CON

  _clkmode = xtal1 + pll16x                'System clock → 80 MHz
  _xinfreq = 5_000_000


OBJ

  SquareWave      : "SquareWave"

PUB TestFrequency | a, b, c

  ' Configure counter modules.

  ctra[30..26] := %00100                  'ctra module to NCO mode
  ctra[8..0] := 27
  outa[27]~                               'P27 → output-low
  dira[27]~~

  ctrb[30..26] := %01010                  'ctrb module to POSEDGE detector
  ctrb[8..0] := 27
  frqb := 1                               'Add 1 for each cycle
  phsb := -3000                           'Start the count at -3000

  a := |< 27                              'Set up a pin mask for the waitpeq command

  frqa := SquareWave.NcoFrqReg(3000)      'Start the square wave
  repeat while phsb[31]                          'Wait for 3000th low→high transition
  waitpeq(0, a, 0)                               'Wait for low signal
  frqa~                                          'Stop the signal
```

PWM

PWM stands for pulse width modulation, which can be useful for both servo and motor control. A counter module operating in NCO mode can be used to generate precise duration pulses, and a `repeat` loop with a `waitcnt` command can be used to maintain the signal's cycle time.

Let's first take a look at sending a single pulse with a counter module. This is a very precise method is good down to the duration of a Propeller chip's system clock tick. After setting up the counter in NCO mode, simply set the PHS register to the duration you want the pulse to last by loading it with a negative value. For example, the command `phsa :=  -clkfreq` in the next example program sets the `phsa` register to -80_000_000. Remember that bit 31 of a register will be 1 so long as it's negative, and also remember that in NCO mode bit 31 of the PHS register controls an I/O pin's output state. So, when the PHS register is set to a negative value (and FRQ to 1), the I/O pin will send a high signal for the same number of clock ticks as the negative number stored in PHS.

The example programs in this PWM section will send signals to the LED circuits in Figure 7 on page 10.

✓ If you removed the circuit from in Figure 7 on page 10 from your board, rebuild it now.

Sending a Single Pulse

The SinglePulseWithCounter object uses this technique to send a 1 second pulse to the LED on P4. Even thought the program can move on as soon as it has set the PHS register to `-clkfreq`, it can't ignore the PHS register indefinitely. Why? Because, $2^{31} – 1$ = 2_147_483_647 clock ticks later, the PHS register will roll over from a large positive number to a large negative number and start counting down again. Since bit 31 of the PHS register will change from 0 to 1 at that point, the I/O pin will transition from low to high for no apparent reason.

✓ Load SinglePulseWithCounter.spin into the Propeller chip and verify that it sends a 1 second pulse. This pulse will last exactly 80_000_000 clock ticks.
✓ With the Propeller chip's clock running at 80 MHz, the pin will go high again about 26.84 seconds later. Verify this with a calculator and by waiting 27 seconds after the 1 s high signal ended.
✓ If you have an oscilloscope, try setting the PHS register to -1 and see if you can detect the 12.5 ns pulse the propeller I/O pin transmits. Also try setting `phsa` to `clkfrq/1_000_000` for a 1 µs pulse.

```
''SinglePulseWithCounter.spin
''Send a high pulse to the P4 LED that lasts exactly 80_000_000 clock ticks.

CON

  _clkmode = xtal1 + pll16x                      ' System clock → 80 MHz
  _xinfreq = 5_000_000

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


  ctra[30..26] := %00100                         ' Configure Counter A to NCO
  ctra[8..0] := 4
  frqa := 1
  dira[4]~~

  phsa := - clkfreq                          ' Send the pulse

  ' Keep the program running so the pulse has time to finish.
  repeat
```

Pulse Width Modulation

For a repeating PWM signal, the program has to establish the cycle time using `waitcnt`. Then, the pulse duration is determined each time through the loop by setting the PHS register to a negative value at the beginning of the cycle.

The 1Hz25PercentDutyCycle object blinks the P4 LED every second for 0.25 seconds. The `repeat` loop repeats once every second, and the counter sends a high signal to the P4 LED for ¼ s with each repetition. The command `tC := clkfreq` sets the variable that holds the cycle time to the number of clock ticks in one second. The command `tHa := clkfreq/4` sets the high time for the A counter module to ¼ s. The command `t := cnt` records the `cnt` register at an initial time.

Next, a `repeat` loop manages the pulse train. It starts by setting `phsa` equal to `-tHa`, which starts the pulse that will last exactly `clkfrq`/4 ticks. Then, it adds `tC`, the cycle time of `clkfreq` ticks, to `t`, the target time for the next cycle to start. The `waitcnt(t)` command waits for the number of ticks in 1 s before repeating the loop.

- ✓ Run the program and verify the ¼ s high time signal every 1 s with the LED connected to P4.
- ✓ If you have an oscilloscope, try a signal that lasts 1.5 ms, repeated every 20 ms. This would be good to make a servo hold its center position.

```
''1Hz25PercentDutyCycle.spin
''Send 1 Hz signal at 25 % duty cycle to P4 LED.

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, t

  ctra[30..26] := %00100                     ' Configure Counter A to NCO
  ctra[8..0] := 4
  frqa := 1
  dira[4]~~

  tC := clkfreq                              ' Set up cycle and high times
  tHa := clkfreq/4
  t := cnt                                   ' Mark counter time

  repeat                                     ' Repeat PWM signal
    phsa := -tHa                             ' Set up the pulse
    t += tC                                  ' Calculate next cycle repeat
    waitcnt(t)                               ' Wait for next cycle
```

This is another good place to examine differential signals. The only differences between this example program and the previous one are:

- The mode is set to NCO differential using `ctra[30..26] := %00101` (differential) instead of `ctra[30..26] := %00100` (single-ended)
- A second I/O pin is selected for differential signals with `ctra[14..9] := 5`

- Both P4 and P5 are set to output with `dira[4..5]~~` instead of just `dira[4]~~`

✓ Try the program and verify that P5 is on whenever P4 is off.

```
''1Hz25PercentDutyCycleDiffSig.spin
''Differential version of 1Hz25PercentDutyCycle.spin

CON

  _clkmode = xtal1 + pll16x                   ' clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, t

  ctra[30..26] := %00101                      ' Counter A → NCO (differential)
  ctra[8..0] := 4                             ' Select I/O pins
  ctra[14..9] := 5
  frqa := 1                                   ' Add 1 to phs with each clock tick

  dira[4..5]~~                                ' Set both differential pins to output

  ' The rest is the same as 1Hz25PercentDutyCycle.spin

  tC := clkfreq                               ' Set up cycle and high times
  tHa := clkfreq/4
  t := cnt                                    ' Mark counter time

  repeat                                      ' Repeat PWM signal
    phsa := -tHa                              ' Set up the pulse
    t += tC                                   ' Calculate next cycle repeat
    waitcnt(t)                                ' Wait for next cycle
```

The TestDualPwm object uses both counters to transmit PWM signals that have the same cycle time but independent high times (1/2 s high time with Counter A and 1/5 s with Counter B). The duty cycle signals are transmitted on P4 and P6.

✓ Try making both signals differential, using I/O pins P4..P7.
✓ Again, if you have an oscilloscope, try making one signal 1.3 ms and the other 1.7 ms. This could cause a robot with two continuous rotation drive servos to either go straight forward or straight backwards.

```
{{
TestDualPWM.spin
Demonstrates using two counter modules to send a dual PWM signal.
The cycle time is the same for both signals, but the high times are independent of
each other.
}}

CON

  _clkmode = xtal1 + pll16x                   ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, tHb, t

  ctra[30..26] := ctrb[30..26] := %00100      ' Counters A and B → NCO single-ended
  ctra[8..0] := 4                             ' Set pins for counters to control
  ctrb[8..0] := 6
  frqa := frqb := 1                           ' Add 1 to phs with each clock tick

  dira[4] := dira[6] := 1                     ' Set I/O pins to output
  tC := clkfreq                              ' Set up cycle time
  tHa := clkfreq/2                           ' Set up high times for both signals
  tHb := clkfreq/5
  t := cnt                                   ' Mark current time.

  repeat                                     ' Repeat PWM signal
    phsa := -tHa                             ' Define and start the A pulse
    phsb := -tHb                             ' Define and start the B pulse
    t += tC                                  ' Calculate next cycle repeat
    waitcnt(t)                               ' Wait for next cycle
```

A variable or constant can be used to stores a time increment for pulse and cycle times. In the example below, the `tInc` variable stores `clkfreq/1_000_000`. When `tC` is set to `50_000 * tInc`, it means that the cycle time will be 500_000 µs. Likewise, `tHa` will be 100_000 µs.

```
''SinglePwm with Time Increments.spin

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, t, tInc

  ctra[30..26] := %00100                     ' Configure Counter A to NCO
  ctra[8..0] := 4                            ' Set counter output signal to P4
  frqa := 1                                  ' Add 1 to phsa with each clock cycle
  dira[4]~~                                  ' P4 → output

  tInc := clkfreq/1_000_000                  ' Determine time increment
  tC := 500_000 * tInc                       ' Use time increment to set up cycle time
  tHa := 100_000 * tInc                      ' Use time increment to set up high time

  ' The rest is the same as 1Hz25PercentDutyCycle.spin

  t := cnt                                   ' Mark counter time

  repeat                                     ' Repeat PWM signal
    phsa := -tHa                             ' Set up the pulse
    t += tC                                  ' Calculate next cycle repeat
    waitcnt(t)                               ' Wait for next cycle
```

Probe and Display PWM – Add an Object, Cog and Pair of Counters

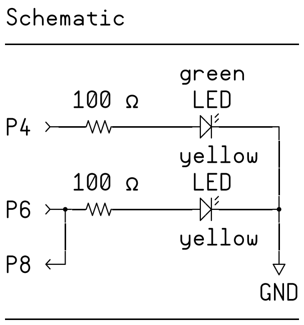
Since the Propeller chip has multiple processors, some of them can be running application code while others are running monitoring and diagnostic code. In this example, we'll incorporate the MonitorPWM and FullDuplexSerialPlus objects (monitoring/diagnostic) into the TestDualPwm object (application) we tested in the previous section. The MonitorPWM object is important because it uses counters in a second cog to monitor the pulse trains transmitted by the cog executing the TestDualPwm code (which is also using two counters).

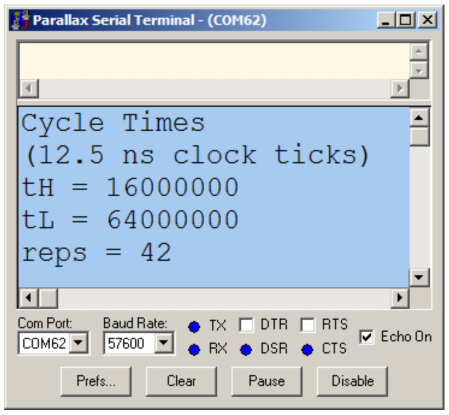
NOTE: After demonstrating an example of using the MonitorPWM object in the TestDualPwmWithProbes object, the MonitorPWM object is examined in detail.

The TestDualPwmWithProbes application below is a modified version of TestDualPwm that makes it possible to monitor the pulse trains sent on P4 or P6 by probing them with P8. The probe information is then displayed on the Parallax Serial Terminal shown in Figure 20. The schematic in Figure 20 shows the P8 I/O pin probing P6. In other words, there is a jumper wire connecting P6 to P8. To probe P4, simply disconnect the P6 end of the P8→P6 jumper and connect it to P4. The measurements are displayed in the Parallax Serial Terminal in terms of 12.5 ns clock ticks; however, the application can easily be modified to display them in terms of ms, µs, duty cycle, etc. A second instance of MonitorPWM can also be declared and used to simultaneously monitor a second channel.

Figure 20: Use P8 to Measure PWM Signal from P6

```
Parts List

(2) Resistors 100 Ω
(1) LED - green
(1) LED - yellow
(misc) Jumper wires
```





```
Cycle Times
(12.5 ns clock ticks)
tH = 16000000
tL = 64000000
reps = 42
```

The code added to the TestDualPwm object to make it monitor and display the pulse trains is highlighted in the TestDualPwmWithProbes object below. Most of the code that was added is for displaying the values in the Parallax Serial Terminal. All that is needed to incorporate the MonitorPWM object is:

- Three variable declarations – `tHprobe`, `tLprobe`, and `pulseCnt`
- An object declaration – `probe : "MonitorPWM"`
- A call to the MonitorPWM object's `start` method that passes the addresses of `tHprobe`, `tLprobe` and `pulseCnt` – `probe.start(8, @tHprobe, @tLprobe, @pulseCnt)`

After that, the MonitorPWM object automatically updates the values stored by **`tHprobe`**, **`tLprobe`**, and `pulseCnt` with each new cycle. These measurements are displayed in the Parallax Serial Terminal with **`debug.dec`**`(tHprobe)`, **`debug.dec`**`(tLprobe)`, and **`debug.dec`**`(pulseCnt)`.

✓ Make sure TestDualPwmWithProbes.spin object is saved to the same folder as MonitorPwm.spin and FullDuplexSerialPlus.spin.

✓ Use the Propeller Tool to load TestDualPwmWithProbes.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button.

✓ Disconnect the end of the P8 → P6 jumper wire that is connected to P6 and connect it to P4. The display should update to reflect the different high and low times.

```
{{
TestDualPwmWithProbes.spin
Demonstrates how to use an object that uses counters in another cog to measure (probe) I/O
pin activity caused by the counters in this cog.
}}

CON

  _clkmode = xtal1 + pll16x                      ' System clock → 80 MHz
  _xinfreq = 5_000_000

  ' Parallax Serial Terminal constants
  CLS = 16, CR = 13, CLREOL = 11, CRSRXY = 2

OBJ
```

```
  debug : "FullDuplexSerialPlus"
  probe : "MonitorPWM"

PUB TestPwm | tc, tHa, tHb, t, tHprobe, tLprobe, pulseCnt

  ' Start MonitorServoControlSignal.
  probe.start(8, @tHprobe, @tLprobe, @pulseCnt)

  'Start FullDuplexSerialPlus.
  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)
  Debug.str(String(CLS, "Cycle Times", CR, "(12.5 ns clock ticks)", CR))

  Debug.str(String("tH = ", CR))
  Debug.str(String("tL = ", CR))
  Debug.str(String("reps = "))

  ctra[30..26] := ctrb[30..26] := %00100     ' Counters A and B → NCO single-ended
  ctra[8..0] := 4                            ' Set pins for counters to control
  ctrb[8..0] := 6
  frqa := frqb := 1                          ' Add 1 to phs with each clock tick

  dira[4] := dira[6] := 1                    ' Set I/O pins to output

  tC := clkfreq                              ' Set up cycle time
  tHa := clkfreq/2                           ' Set up high times for both signals
  tHb := clkfreq/5
  t := cnt                                   ' Mark current time.

  repeat                                     ' Repeat PWM signal
    phsa := -tHa                             ' Define and start the A pulse
    phsb := -tHb                             ' Define and start the B pulse
    t += tC                                  ' Calculate next cycle repeat

    ' Display probe information
    debug.str(String(CLREOL, CRSRXY, 5, 2))
    debug.dec(tHprobe)
    debug.str(String(CLREOL, CRSRXY, 5, 3))
    debug.dec(tLprobe)
    debug.str(String(CLREOL, CRSRXY, 7, 4))
    debug.dec(pulseCnt)

    waitcnt(t)                               ' Wait for next cycle
```

Monitoring PWM – Example of an Object that Uses Counters in another Cog

The MonitorPWM object below can be used by other objects to measure the characteristics of a pulse train (its high and low times). Code in some other cog can be transmitting pulses, and the application can use this object to measure the high and low times of the pulses. Up to this point, all objects have been using the counter modules in cog 0. In contrast, the MonitorPWM object launches a new cog and uses that new cog's counter modules to measure the pulse high and low times. It then makes its measurements available to the other objects by storing them at mutually agreed upon locations in main RAM.

Here are three important tips for writing objects that launch new cogs and use those cogs' counter modules. Keep them in mind as you examine the MonitorPWM object.

1) If the object is launching a new cog, it should have `start` and `stop` methods and global variables named `cog` and `stack`. This is a convention introduced by Parallax that is used in the Propeller Library and the Propeller Object Exchange. The object should also declare any global variables required by the process that gets launched into the new cog. (This was all introduced in the Objects lab.)
2) The `start` method should copy any parameters it receives to global variables before launching the method that manages the process into a new cog.
3) **The method that gets launched into a new cog should make counter configurations and I/O pin assignments.**

Regarding tip 3: Let's say that cog 0 calls your object's `start` method, and the `start` method launches a counter-using method into cog 1 with the **`cognew`** command. You have to put code that does counter configuration and I/O pin assignments into the method that gets launched by **`cognew`** if you want the counter modules in cog 1 to work. If you try to configure the counters or I/O pins in the `start` method (before the cog gets launched), those configurations affect cog 0 instead of cog 1. This would in turn create program bugs because the counter modules in cog 1 will not be able to access the I/O pins.

```
{{
MonitorPWM.spin

Monitors characteristics of a probed PWM signal, and updates addresses in main RAM
with the most recently measured pulse high/low times and pulse count.

How to Use this Object in Your Application
------------------------------------------
1) Declare variables for high time, low time, and pulse count.  Example:

   VAR
     long tHprobe, tlprobe, pulseCnt

2) Declare the MonitorPWM object.  Example:

   OBJ
     probe : MonitorPWM

3) Call the start method and pass the I/O pin used for probing and the variable addresses
   from step 1.  Example:

   PUB MethodInMyApp
     '...
     probe.start(8, tHprobe, tLprobe, pulseCnt)

4) The application can now use the values of tHprobe, tLprobe, and pulseCnt to monitor
   the pulses measured on the I/O pin passed to the start method (P8 in this example).
   In this example, this object will continuously update tHprobe, tLprobe, and pulseCnt
   with the most recent pulse high/low times and pulse count.

See Also
--------
TestDualPwmWithProbes.spin for an application example.

}}

VAR
  long cog, stack[20]                      ' Tip 1, global variables for cog and stack.
  long apin, thaddr, tladdr, pcntaddr      ' Tip 1, global variables for the process.
```

```
PUB start(pin, thighAddr, tlowaddr, pulsecntaddr) : okay

  '' Starts the object and launches PWM monitoring process into a new cog.
  '' All time measurements are in terms of system clock ticks.
  ''
  '' pin - I/O pin number
  '' tHighAddr - address of long that receives the current signal high time measurement.
  '' tLowAddr - address of long that receives the current signal low time measurement.
  '' pulseCntAddr - address of long that receives the current count of pulses that have
  ''                been measured.

  ' Copy method's local variables to object's global variables
  apin := pin                              ' Tip 2, copy parameters to global variables
  thaddr := tHighAddr                      ' that the process will use.
  tladdr := tLowAddr
  pcntaddr := pulseCntAddr

  ' Launch the new cog.
  okay := cog := cognew(PwmMonitor, @stack) + 1

PUB stop

  '' Stop the PWM monitoring process and free a cog.

  if cog
    cogstop(cog~ - 1)

PRI PwmMonitor

  ' Tip 3, set up counter modules and I/O pin configurations(from within the new cog!)

  ctra[30..26] := %01000                           ' POS detector
  ctra[5..0] := apin                               ' I/O pin
  frqa := 1

  ctrb[30..26] := %01100                           ' NEG detector
  ctrb[5..0] := apin                               ' I/O pin
  frqb := 1

  phsa~                                            ' Clear counts
  phsb~

  ' Set up I/O pin directions and states.
  dira[apin]~                                      ' Make apin an input

  ' PWM monitoring loop.

  repeat                                           ' Main loop for pulse
                                                   ' monitoring cog.
    waitpeq(|<apin, |<apin, 0)                     ' Wait for apin to go high.
    long[tladdr] := phsb                           ' Save tlow, then clear.
    phsb~
    waitpeq(0, |<apin,0)                           ' Wait for apin to go low.
    long[thaddr] := phsa                           ' Save thigh then clear.
    phsa~
    long[pcntaddr]++                               ' Increment pulse count.
```

Inside the MonitorPMW Object

The first thing MonitorPWM does is declare its global variables. Variables named `cog` and `stack` were introduced in the Objects lab. The `cog` variable is used to keep track of which cog the object's `start` method launched the process into. Later, if this object's `stop` method gets called, it knows which cog to shut down. Since two methods use this variable, it has to be global because methods cannot see each other's local variables. The `stack` variable provides stack space for the code that gets launched into the new cog for calculations, return pointers, etc.

```
VAR
  long cog, stack[20]                    ' Tip 1, global variables for cog and stack.
  long apin, thaddr, tladdr, pcntaddr    ' Tip 1, global variables for the process.
```

Global variables named `apin`, `thaddr`, `tladdr`, and `pcntaddr` are also declared. These variables get used by two different methods: `start` and `pwmMonitor`. The `start` method receives parameters from an object that calls it and copies them into these global variables so that the `pwmMonitor` method can use them. The `PwmMonitor` method uses the `apin` variable to configure I/O pins, and it uses the other three variables as address pointers for storing its measurements at the "mutually agreed upon locations in main RAM" mentioned earlier.

When another object calls this object's start method, it passes the I/O pin number that will be doing the signal monitoring along with addresses where the high and low pulse time measurements should be stored and an address to store the number of pulses that have been counted. Keep in mind that these parameters (`pin`, `thighAddr`, `tlowaddr`, `pulsecntaddr`) are local variables in the **`start`** method. To make these values available to other methods, the `start` method has to copy them to global variables. So, before launching the new cog, the `start` method copies `pin` to `apin`, `tHighAddr` to `thaddr`, `tLowAddr` to `tlAddr` and `pulseCntAddr` to `pcntaddr`. After that, the cognew command launches the `PwmMonitor` method into a new cog and passes the address of the `stack` array. The `stack` array was introduced in the Objects lab.

```
PUB start(pin, thighAddr, tlowaddr, pulsecntaddr) : okay
'...
' Copy method's local variables to object's global variables
  apin := pin                           ' Tip 2, copy parameters to global variables
  thaddr := tHighAddr                   ' that the process will use.
  tladdr := tLowAddr
  pcntaddr := pulseCntAddr

  ' Launch the new cog.
  okay := cog := cognew(PwmMonitor, @stack) + 1
```

Objects that launch new cogs that are designed to exchange information with other objects have `start` and `stop` methods by convention. Also by convention, if your object does not launch a new cog but it does need to be configured, use a method named `init` or `config` instead.

Look at the last line in the `start` method above. The **`cognew`** command returns -1 if there were no available cogs, or the number of the cog the `PwmMonitor` method got launched into, which could be 0 to 7. Next, one gets added to this value, and the result is stored in the object's `cog` variable and the `start` method's `okay` return parameter. So, the `start` method returns 0 (**`false`**) if the process failed to launch or nonzero if it succeeded. The object calling the `start` method can then use the value the `start` method returns in an `IF` block to decide what to do. Again, if the value returned is 0 (false) it means there were no cogs available; whereas, if the value is nonzero, the application knows the cog successfully launched.

The `stop` method can also determine if the process was successfully launched because the `cog` variable also stores the result **`cognew`** returned, plus one. If the `stop` method gets called, the first thing it does is use an `IF` statement to make sure there's really a cog that was started. For the third time, if the value of `cog` is zero, there's not currently a process under this object's control that needs to be stopped. On the other hand, if the value of `cog` is nonzero, **`cogstop`**`(cog~ - 1)` does 3 things:

(1) Subtract 1 from the value stored by `cog` to get the number of the cog that needs to be stopped. (Remember, a command in the `start` method added 1 to the `cog` variable).
(2) Stop the cog.
(3) Clear the value of `cog` so that the object knows it's not currently in charge of an active process (cog).

```
PUB stop

  '' Stop the PWM monitoring process and free a cog.

  if cog
    cogstop(cog~ - 1)
```

The `PwmMonitor` method gets launched into a new cog by a **`cognew`** command in the `start` method. So code in the `PwmMonitor` method is running in a separate processor from the code that called the **`start`** method. The first thing the `PwmMonitor` method does is configure the counter modules and I/O pins it is going to use. Remember, your code cannot do this from another cog; code executed by a given cog has to make its own counter configurations and I/O pin assignments. (See tip 3, discussed earlier.)

```
PRI PwmMonitor

  ' Tip 3, set up counter modules and I/O pin configurations(from within the new cog!)

  ctra[30..26] := %01000                             ' POS detector
  ctra[5..0] := apin                                 ' I/O pin
  frqa := 1

  ctrb[30..26] := %01100                             ' NEG detector
  ctrb[5..0] := apin                                 ' I/O pin
  frqb := 1

  phsa~                                              ' Clear counts
  phsb~

  ' Set up I/O pin directions and states.
  dira[apin]~                                        ' Make apin an input

  ' PWM monitoring loop.
```

The main loop in the `PwmMonitor` method waits for the signal to be high. Then it copies the contents of `phsb`, which accumulates the low time, to an address in main RAM. Remember that the address in main memory was passed to the `start` method's `thighaddr` parameter. The start method copied it to the global `thaddr` variable. Since `thaddr` is a global variable, it's accessible to this method too. Likewise with `tlowaddr` → `tladdr` and `pulsecntaddr` → `pcntaddr`. Before waiting to measure the signal's low time, the code clears the **`phsb`** register for the next measurement. After the signal goes low, it copies **`phsa`** to the memory set aside for measuring the high time. Before the next cycle gets measured, 1 gets added to the memory pointed to by the `pcntaddr` variable, which tacks the number of cycles.

```
  ' PWM monitoring loop.
  repeat                                             ' Main loop for pulse
                                                     ' monitoring cog.
    waitpeq(|<apin, |<apin, 0)                       ' Wait for apin to go high.
    long[tladdr] := phsb                             ' Save tlow, then clear.
    phsb~
    waitpeq(0, |<apin,0)                             ' Wait for apin to go low.
    long[thaddr] := phsa                             ' Save thigh then clear.
    phsa~
    long[pcntaddr]++                                 ' Increment pulse count.
```

PLL for High Frequency

Up to this point, we have used NCO mode for generating square waves in the audible (20 to 20 kHz) and IR detector (38 kHz) range. The NCO mode can be used to generate signals up to `clkfreq/2`. So, with the version of the Propeller chip used in these labs, the ceiling frequency for this mode is 40 MHz.

For signals faster than `clkfreq/2`, you can use the counter's PLL (phase-locked loop) mode. Instead of sending bit 31 of the PHS register straight to an I/O pin, PLL mode passes the signal through two additional subsystems before transmitting it. These subsystems are not only capable of sending frequencies from 500 kHz to 128 MHz, they also diminish the jitter inherent to NCO signals. The first subsystem (counter PLL) takes the frequency that bit 31 of the PHS register toggles at and multiplies it by 16 using a voltage-controlled oscillator (VCO) circuit. The Propeller Manual and CTR object call this the VCO frequency. The second subsystem (divider) divides the resulting frequency by a power of 2 ranging from 1 to 128.

The PLL is designed to accept PHS bit 31 frequencies from 4 to 8 MHz. The PLL subsystem multiplies this input frequency by 16, for a counter PLL frequency ranging from 64 to 128 MHz. The divider subsystem then divides this frequency by a power of two from 128 to 1, so the final output for PLL signals can range from 500 kHz to 128 MHz.

Configuring the Counter Module for PLL Mode

Figure 21 is the now-familiar excerpt from the Propeller Library's CTR object, this time with the PLL modes listed. "PLL internal" is used for synchronizing video signals. Although not discussed in this lab, you can see it applied in the Propeller Library's TV object. The CTRMODE values for routing the PLL signal to I/O pins are %00010 for single-ended, and %00011 for differential.

Figure 21: NCO Excerpts from the CTR Object's Counter Mode Table

CTRMODE	Description	Accumulate FRQ to PHS	APIN output*	BPIN output*
%00000	Counter disabled (off)	0 (never)	0 (none)	0 (none)
. . .				
%00001	PLL internal (video mode)	1 (always)	0	0
%00010	PLL single-ended	1	PLL	0
%00011	PLL differential	1	PLL	!PLL
. . .				
%11111	LOGIC always	1	0	0

```
* must set corresponding DIR bit to affect pin

  A¹ = APIN input delayed by 1 clock
  A² = APIN input delayed by 2 clocks
  B¹ = BPIN input delayed by 1 clock
```

The CTR Register's PLLDIV bit Field

With NCO mode, setting I/O pin frequencies was done directly through the FRQ register. The value in FRQ was added to PHS every clock tick, and that determined the toggle rate of PHS bit31, which directly controlled the I/O pin. While setting I/O pin frequencies with PLL mode still uses PHS bit 31, there are some extra steps.

In PLL mode, the toggle rate of PHS bit31 is still determined by the value of FRQ, but before the I/O pin transmits the signal, the PHS bit 31 signal gets multiplied by 16 and then divided down by a power of two of your choosing ($2^0 = 1$, $2^1 = 2$, $2^2 = 4$, … $2^6 = 64$, $2^7 = 128$). The power of 2 is selected by a value stored in the CTR register's PLLDIV bit field, (bits 25..23) in Figure 22.

Figure 22: CTRA/B Register Map from CTR.spin

bits	31	30..26	25..23	22..15	14..9	8..6	5..0
Name	—	CTRMODE	PLLDIV	———	BPIN	——	APIN

Calculating PLL Frequency Given FRQ and PLLDIV

Let's say you are examining a code example or object that's generating a certain PLL frequency. You can figure out what frequency it's generating using the values of `clkfreq`, the FRQ register, and the value in the CTR register's PLLDIV bit field. Just follow these three steps:

b. Calculate the PHS bit 31 frequency:

$$\text{PHS bit 31 frequency} = \frac{\text{clkfreq} \times \text{FRQ register}}{2^{32}}$$

c. Use the PHS bit 31 frequency to calculate the VCO frequency:

$$\text{VCO frequency} = 16 \times \text{PHS bit 31 frequency}$$

d. Divide the PLLDIV result, which is $2^{7-\text{PLLDIV}}$ into the VCO frequency:

$$\text{PLL frequency} = \frac{\text{VCO frequency}}{2^{7-\text{PLLDIV}}}$$

Example: Given a system clock frequency (`clkfreq`) of 80 MHz and the code below, calculate the PLL frequency transmitted on I/O Pin P15.

```
'Configure ctra module
ctra[30..26] := %00010
frqa := 322_122_547
ctra[25..23] := 2
ctra[5..0] := 15
dira[15]~~
```

1) Calculate the PHS bit 31 frequency:

$$\text{PHS bit 31 frequency} = \frac{80\_000\_000 \times 322\_122\_547}{2^{32}}$$

$$= 5\_999\_999$$

2) Use the PHS bit 31 frequency to calculate the VCO frequency:

```
VCO frequency = 16 × 5_999_999
              = 95_999_984
```

3) Divide the PLLDIV result ($2^{7-\text{PLLDIV}}$) into the VCO frequency:

$$\text{PLL frequency} = \frac{95\_999\_984}{2^{7-2}}$$

$$= 2\_999\_999 \text{ MHz}$$

$$\approx 3 \text{ MHz}$$

Calculating FRQ and PLLDIV Given a PLL Frequency

Figuring out the PLL frequency given some pre-written code is well and good, but what if you want to calculate FRQ register and PLLDIV bit fields values to generate a frequency with your own code? Here are four steps you can use to figure it out:

1) Use the table below to figure out which value to put in the CTR register's PLLDIV bit field based on the frequency you want to transmit.

MHz	PLLDIV	MHz	PLLDIV
0.5 to 1	0	8 to 16	4
1 to 2	1	16 to 32	5
2 to 4	2	32 to 64	6
4 to 8	3	64 to 128	7

2) Calculate the VCO frequency with the PLL frequency you want to transmit and the PLL divider, and round down to the next lowest integer.

$$\text{VCO frequency} = \text{PLL frequency} \times 2^{(7-\text{PLLDIV})}$$

3) Calculate the PHS bit 31 frequency you'll need for the VCO frequency. It's the VCO frequency divided by 16.

```
PHS bit 31 frequency = VCO frequency ÷ 16
```

4) Use the NCO frequency calculations to figure out the FRQ register value for the PHS bit 31 frequency.

$$\text{FRQ register} = \text{PHS bit 31 frequency} \times \frac{2^{32}}{\text{clkfreq}}$$

Example: `clkfreq` is running at 80 MHz, and you want to generate a 12 MHz signal with PLL. Figure out the FRQ register and PLLDIV bit fields.

1) Use the table to figure out which value to put in the CTR register's PLLDIV bit field:

Since 12 MHz falls in the 4 to 16 MHz range, PLLDIV is 4.7. Round down, and use 4.

2) Calculate the VCO frequency with the final PLL frequency and the PLL divider:

$$\begin{aligned}\text{VCO frequency} &= 12\text{ MHz} \times 2^{(7-4)} \\ &= 12\text{ MHz} \times 8 \\ &= 96\text{ MHz}\end{aligned}$$

3) Calculate the PHS bit 31 frequency you'll need for the VCO frequency. It's the VCO frequency divided by 16:

$$\begin{aligned}\text{PHS bit 31 frequency} &= 96\text{ MHz} \div 16 \\ &= 6\text{ MHz}\end{aligned}$$

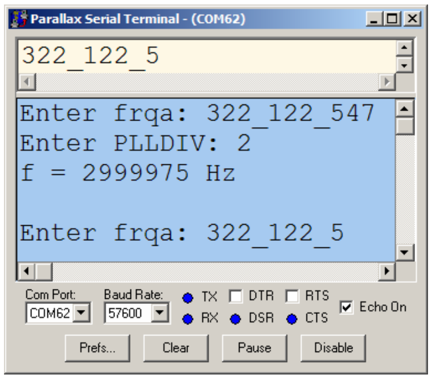
4) Use the NCO frequency calculations to figure out the FRQ register value for the PHS bit 31 frequency:

$$\begin{aligned}\text{FRQ register} &= 6\text{ MHz}\,\frac{2^{32}}{80\text{ MHz}} \\ &= 322\_122\_547\end{aligned}$$

Testing PLL Frequencies

The TestPllParameters object lets you control Counter A's PLL output frequency by hand-entering values for `frqa` and also for `ctra`'s PLLDIV bit field into Parallax Serial Terminal (Figure 23). The program transmits the frequency you entered for 1 s, counting the cycles with Counter B set to NEGEDGE detection mode.

Figure 23: Calculate Frequency Given FRQA and PLLDIV



```
322_122_5
```

```
Enter frqa: 322_122_547
Enter PLLDIV: 2
f = 2999975 Hz

Enter frqa: 322_122_5
```

Note that there is a slight difference between the measured frequency and the hand-calculated frequency discussed earlier. If the `delay := clkfreq + cnt` calculation in the object TestPllParameters.spin is placed immediately before `phsb~`, the frequency count will be slightly less than the actual frequency. If it were moved below `phsb~`, the measurement will be slightly larger than the actual frequency. An exact measurement can be obtained with the help of an assembly language object.

Although the PLL can generate frequencies up to 128 MHz, the Propeller chip's counters can only detect frequencies up to 40 MHz with counter modules. This concurs with the Nyquist sampling rate, which must be twice as fast as the highest frequency it could possibly measure. Also, if you consider that the edge detection mode adds FRQ to PHS when it detects a high signal during one clock tick and a low signal during the next, it needs at least two clock ticks to detect a signal's full cycle.

- ✓ Calculate FRQ register and PLLDIV bit field values for various frequencies in the 500 kHz to 40 MHz range.
- ✓ Use the Propeller Tool to load TestPllParameters.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Enter the FRQ and PLLDIV values into the Parallax Serial Terminal's transmit windowpane at the prompts and verify that the measured frequency is in the same neighborhood as your calculations.

```
{{
TestPllParameters.spin

Tests PLL frequencies up to 40 MHz.  PHS register and PLLDIV bit field values are
entered into Parallax Serial Terminal.  The Program uses these to synthesize square wave
with PLL mode using counter module A.  Counter module B counts the cycles in 1 s
and reports it.
}}

CON

  _clkmode = xtal1 + pll16x                ' System clock → 80 MHz
  _xinfreq = 5_000_000

  ' Constants for Parallax Serial Terminal.
  CLS = 16, CR = 13

OBJ

  SqrWave  : "SquareWave"
  debug    : "FullDuplexSerialPlus"

PUB TestFrequency | delay, cycles

  Debug.Start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)
  Debug.tx(CLS)

  ' Configure counter modules.
  ctra[30..26] := %00010                   'ctra module to PLL single-ended mode
  ctra[5..0] := 15

  ctrb[30..26] := %01110                   'ctrb module to NEGEDGE detector
  ctrb[5..0] := 15
  frqb:= 1

  repeat

    Debug.str(String("Enter frqa: "))      'frqa and PLLDIV are user intput
    frqa := Debug.GetDec

    Debug.str(String("Enter PLLDIV: "))
    ctra[25..23] := Debug.GetDec

    dira[15]~~                             'P15 → output
    delay := clkfreq + cnt                 'Precalculate delay ticks
    phsb~                                  'Wait 1 s.
    waitcnt(delay)
    cycles := phsb                         'Store cycles
    dira[15]~                              'P15 → input

    Debug.str(String("f = "))              'Display cycles as frequency
    debug.dec(cycles)
    debug.str(String(" Hz", CR, CR))
```

Metal Detection with PLL and Positive Detector Modes and an LC Circuit

Inductors are coils that when placed in a circuit have the capacity to store energy. They get used in many types of applications, one of which is metal detection. There are lots of different kinds of metal detection instruments aside from the ones you may have seen passed over the sands on just about any beach on any given weekend. Other examples include instruments that identify the type of metal, check for stress fractures in metal surfaces, and precisely measure the distance of a metal surface from an instrument.

Even though there aren't any inductors in the PE kit, there are lots of wires that can be shaped into metal loops. When current passes through a metal loop, it becomes a small inductor. This portion of the lab demonstrates how a cog can use two counters, one in PLL mode and the other in POS detector mode, to send high-frequency signals into an LC (inductor-capacitor) circuit's input, and infer the presence or absence of metal by examining the circuit's output signal.

Figure 24 shows a parts list and circuit for the PE Kit's metal detector. Because of the small part sizes and high frequencies involved, this circuit can be finicky. So, for best results, wire it exactly like the breadboard photo shown in Figure 25. The capacitor and resistors should all be sticking straight up off the board, and the two wires should be on the same plane as the board.

This circuit will also require some tuning. Figure 24 starts with R1 at 100 Ω, and R2 (100 Ω) and R3 (470 Ω) are in parallel. The notation these labs will use for parallel resistor combinations is R2 || R3. Your particular circuit may require a larger or smaller resistor in parallel with either R1 or R2, but for now, start with R1 = 100 Ω and R2 = 100 Ω || 470 Ω.

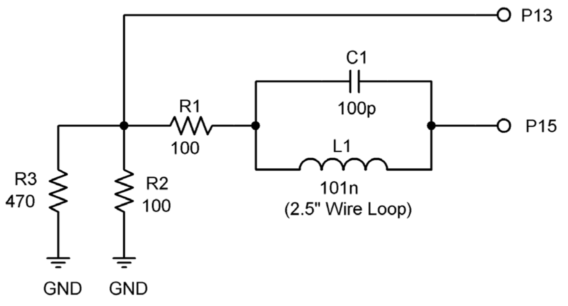
✓ Build the circuit in Figure 24 on your PE Platform exactly as shown in Figure 25.

Figure 24: Metal Detector Parts and Schematic

Parts List

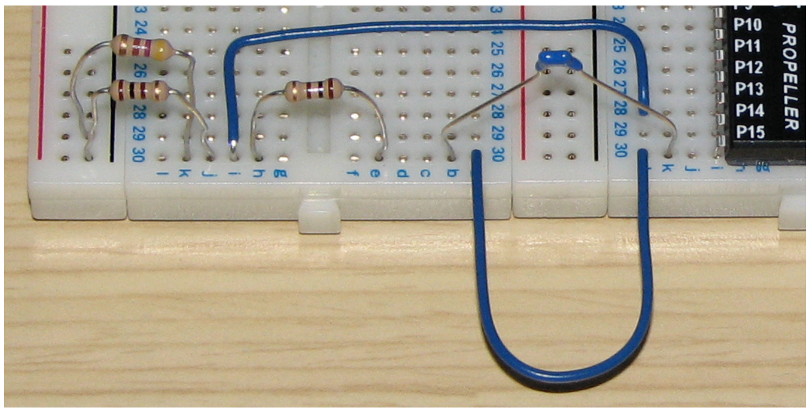
(1) Capacitor 10 pF
(2) Jumper Wires
(2) Resistors 100 Ω
(misc) resistors:
220, 470, 1000,
2000, 10k

Schematic



Make sure the wire loop you are using for an inductor is parallel to the surface of the board while the other parts are perpendicular.

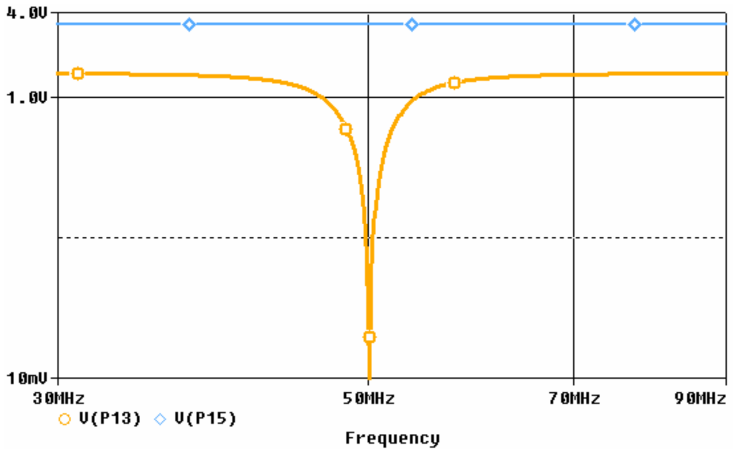
Figure 25: Metal Detector Wiring



Detecting Resonant Frequency

The LC circuit shown in Figure 24 is commonly called a *bandreject*, *bandstop*, or *notch* filter. The filter attenuates a certain frequency sine wave component from an input signal, ideally down to nothing at a certain frequency. The frequency that gets filtered is called the filter's *center frequency* as well as the LC circuit's *resonant frequency*. Figure 26 shows a simulated plot of how the filter responds to a range of input (P15) sine wave frequencies from 30 to 90 MHz. Notice that the filter's center frequency is 50 MHz. So, if the input were a sine wave, its amplitude would be attenuated almost to nothing; whereas at frequencies well outside the filter's center frequency, the output sine wave amplitude would instead be in the 1.6 V neighborhood.

Figure 26: Simulated P13 Output Vs. P15 Input for Sine Waves Frequencies



More about filters and simulation software:

If you swap R and C || L, you will have a bandpass filter. The frequency response is the upside-down version of what's shown in Figure 20. For more information on LC filters, look up terms *frequency selective circuits*, *filters*, *low-pass*, *high-pass*, *bandpass* and *bandreject* in an electronics textbook.

The simulations in this section were preformed with OrCAD Demo Software, which is available for free download from www.cadence.com.

Regardless of whether it's a bandreject or bandpass filter, the circuit's resonant frequency (f_R) can be calculated with this equation. L is the inductor's inductance, measured in henrys (H), and C is the capacitor's capacitance, measured in farads (F). Of course, the L and C in Figure 24 are minute fractions of henrys and farads, respectively.

$$f_R = \frac{1}{2\pi\sqrt{LC}}$$ **Eq. 5**

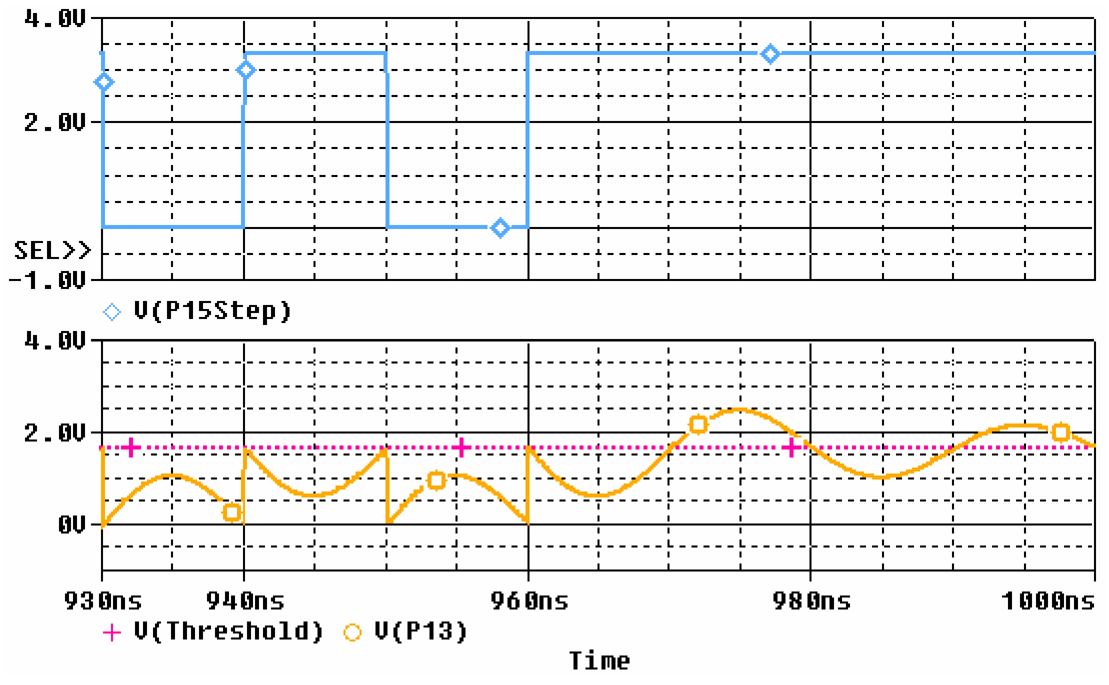
Rearranging terms makes it possible to calculate the inductance (L) based on frequency response tests.

$$L = \frac{1}{(2\pi f_R)^2 C}$$ **Eq. 6**

In this lab, the LC circuit's input will be a square wave from P15. Although the output is still related to the circuit's filtering characteristics, its behavior will make a lot more sense if examined from the *step response* standpoint. A circuit's step response is especially important to digital circuits, and the typical goal is to make the circuit's output quickly and accurately respond to the input and settle at its new value. The most desirable step response is called *critically damped* because it reaches the target value as quickly as possible without overshooting it. Some designs can get quicker responses with an *underdamped* circuit, but at a penalty of some oscillation above and below the new target voltage before the signal settles down. Other designs need an *overdamped* step response, which is slower to reach its target voltage, but ensures that no overshoot or ringing will occur.

The simulated step response shown in Figure 27 is a fairly drastic case of an underdamped step response. V(P15Step) in the upper plot is the LC circuit's input signal. V(P13) is the output signal, and V(Threshold) is a DC signal at the Propeller chip's 1.65 V threshold. The simulation is not really a typical step response because a 50 MHz square wave was applied for 960 ns before the so-called step (high signal) was applied. The result was that the inductor and capacitor both accumulated some stored energy, which makes V(P13)'s pseudo-step response to the right of the 960 ns mark more pronounced than it would otherwise be. The important thing to notice about V(P13) to the right of the 960 ns mark is that it's a sine wave that decays gradually. This sine wave occurs at the LC circuit's 50 MHz resonant frequency.

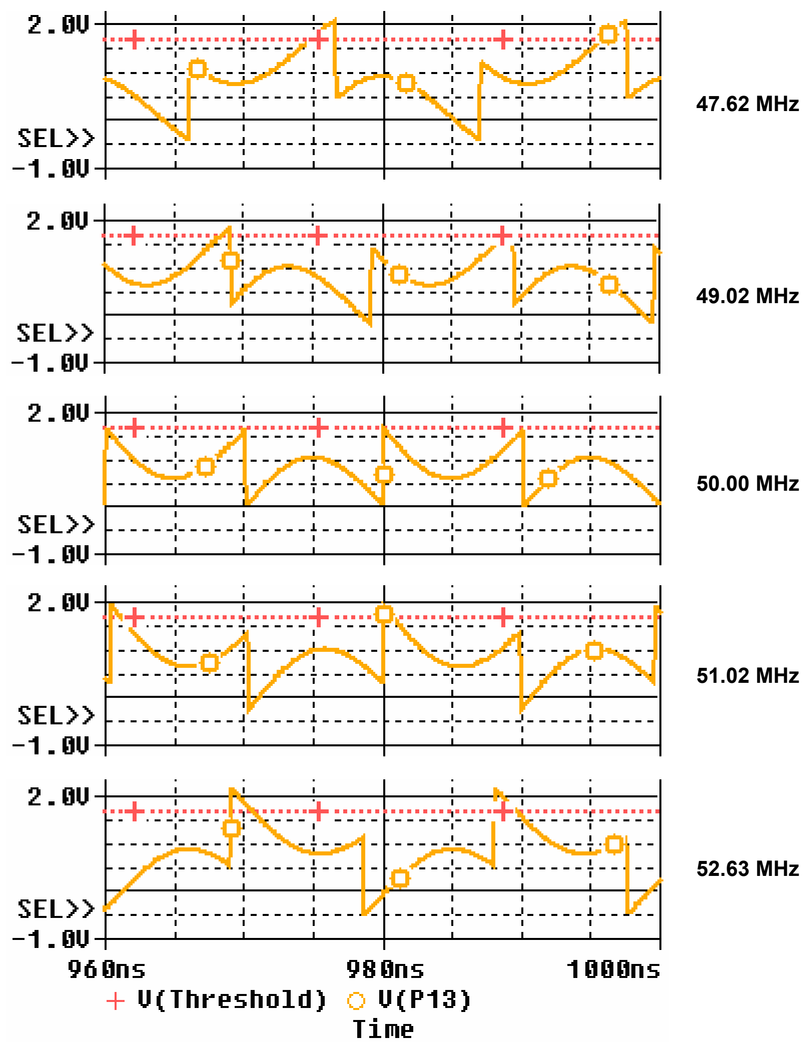
Figure 27: P13 Response to Resonant Frequency at P15



Also, take a look at the V(P13) trace between 930 and 960 ns. With each transition of the 50 MHz V(P15Step) signal, V(P13) starts a sine wave reaction that initially opposes the V(P15Step) input signal. Since the V(P13) signal only gets through half of its 50 MHz sine wave response before the V(P15Step) signal changes, the portions of those sine wave responses never make it above the Propeller chip's 1.65 V threshold.

Next, compare the V(P13) response to square wave frequencies slightly above and below the circuit's 50 MHz resonant frequency, shown in Figure 28. At 47.62 MHz, the sine wave completes slightly more than ½ of its cycle, part of which has climbed above the 1.65 V threshold voltage (designated by the line with the + characters). At 49.02 MHz, the sine wave is still repeating more than a full cycle, but not as much, so the signal spends less time above the threshold voltage. At 50 MHz, the input frequency matches the sinusoidal response, and since only half the sine wave repeats, the signal doesn't spend any time above the threshold voltage. At 51.02 and 52.63 MHz, the signal again spends some time above the 1.65 V I/O pin threshold, this time because the input signal changes before the sine wave has completed its cycle.

Figure 28: LC Circuit P13 Output Responses at Various Frequencies



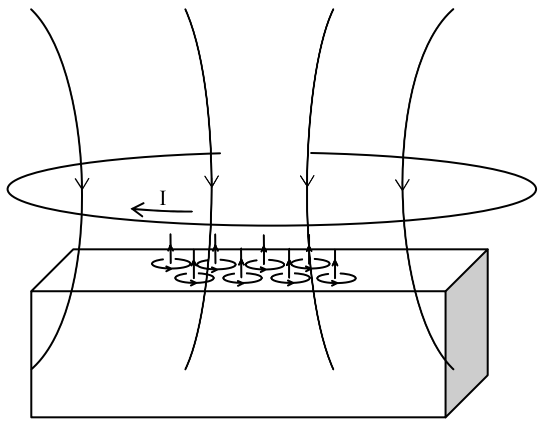
The most important thing Figure 28 indicates is that the output signal, which can be monitored by P13, will spend more time above the I/O pin's logic threshold when the P15 input signal is further away from the circuit's resonant frequency, either above or below. The Propeller can use a counter in PLL mode to generate square waves in the range of frequencies shown in Figure 28, and it can use another counter on POS detector mode to track how long the circuit's output signal spends above the P13 I/O pin's threshold voltage.

So, the Propeller chip can use two counter modules and a small amount of code to sweep the P15 PWM frequency through a range of values to find the resonant frequency of the Figure 24 circuit, but how does that make it possible to detect metal? The answer is that a nearby metal object electromagnetically interacts with the Figure 24 circuit's wire loop inductor in such a way that it changes its inductance, and also adds a small amount of resistance. When the circuit's inductance changes, its resonant frequency also changes, and the Propeller chip can detect that by sweeping P15 PLL frequencies and measuring P13 high times, which will reach a minimum at a different resonant frequency as a result of a nearby metal object.

How Eddy Currents in a nearby Metal Object Affect the Loop's Resonant Frequency

Figure 29 illustrates the electromagnetic interaction between a nearby metal object and the wire's loop inductance. The alternating currents through the loop cause alternating electromagnetic fields. These alternating magnetic fields cause groups of electrons in the conductive metal to travel in alternating circular paths. These magnetically induced circular paths are called *eddy currents*. The alternating eddy currents generate magnetic fields that oppose the fields generated by the wire loop.

Figure 29: Eddy Currents Causing Opposing Magnetic Fields



The eddy currents shown in Figure 29 provide a very small, high-frequency example of how power is transferred in AC lines. A coil connected to the power line is typically magnetically coupled with a coil of fewer turns. The alternating current in the primary induces an alternating magnetic field that induces AC current in the secondary winding. Figure 30 shows how the secondary winding and load affect the primary. The secondary winding's inductance and any resistive load can be seen in the primary, and can be accounted for as L2' and R'.

Figure 30: Eddy Currents Effects on the Loop's Inductance

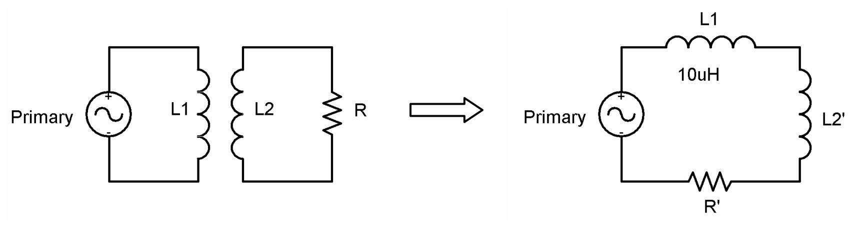


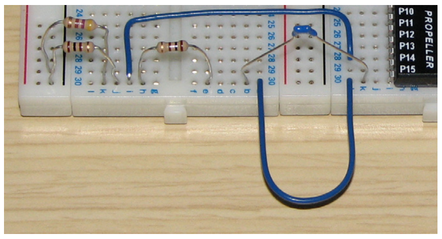
Figure 30 also represents how eddy currents, which have a certain inductance due to the fact that eddies (circular electron currents) are induced in the metal, affect the primary circuit's inductance and resistance. So, eddy currents in the nearby metal object affect the metal loop's inductance. Since the loop's inductance is measured by L in the resonance equation, it will change the LC circuit's resonant frequency. Also, since the Propeller chip can detect the circuit's resonant frequency by sweeping PLL square wave frequencies on one pin while measuring the number of ticks the circuit's output signal is above the threshold on another, the application can detect the presence or absence of nearby metals.

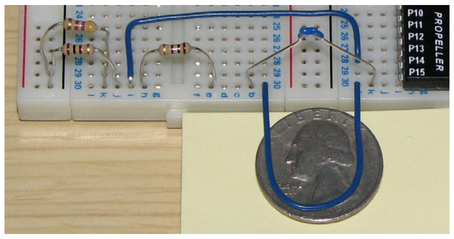
Testing for Resonant Frequency

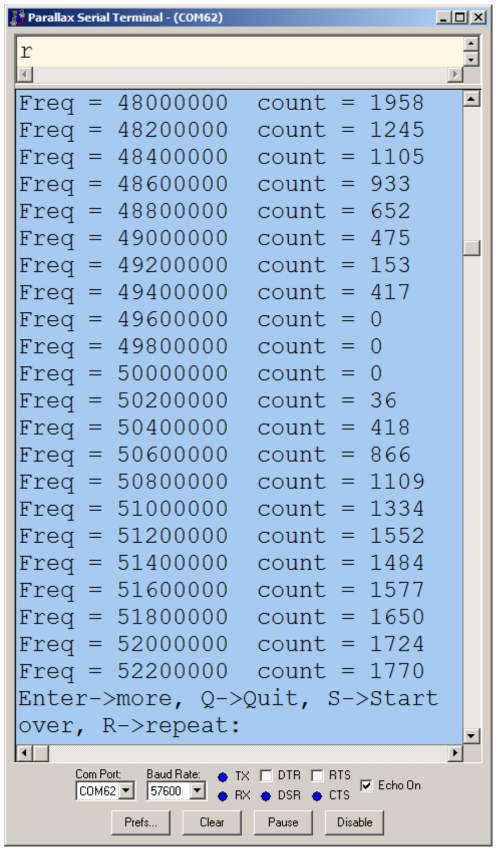
The Calibrate Metal Detector object below provides an interface for testing the LC circuit's frequency response with the Propeller chip. As mentioned earlier, the small values and relatively high frequencies used with this circuit make it a little finicky. For example, if the capacitor is more than 90° from the loop, the resonant frequency drops, if it is less than 90°, the resonant frequency increases. Also, the various parts will have slightly different characteristics, so it will take some tinkering to set up the circuit so that the resistor divider will cause the LC circuit's output signal to stay below the I/O pin threshold at resonant frequency and creep above it as the frequency sweep gets either further above or below it.

Figure 31 shows Calibrate Metal Detector.spin's output after the circuit has been calibrated. The high tick counts on the left actually resemble Figure 26's frequency response plot with a center frequency in the 49.8 to 50 MHz neighborhood. On the other hand, the tick counts on the right show that there is still a resonant frequency, but it's up at about 50.6 Mhz. Since the circuit inductive loop also experiences increased resistance, it may prevent the circuit from attenuating the signal so that the count measurements never quite make it to zero.

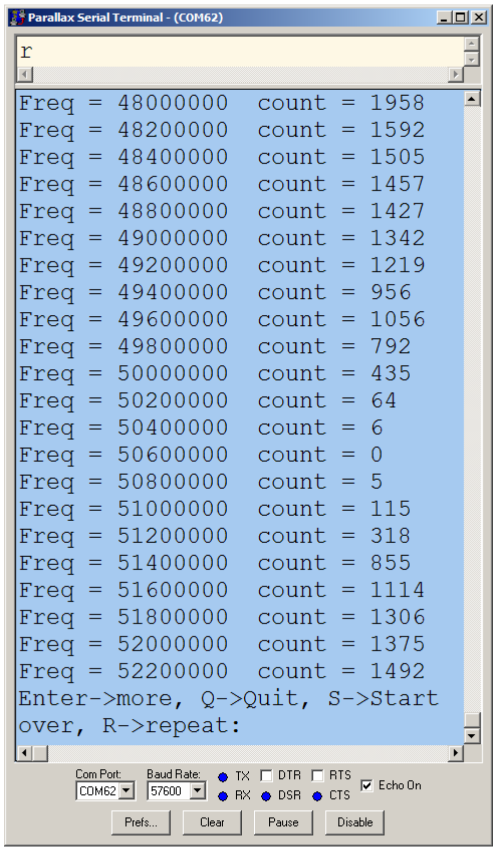
Figure 31: Calibrated Metal Detector Response – ***without metal (left) and with metal (right)***







```
r
Freq = 48000000   count = 1958
Freq = 48200000   count = 1245
Freq = 48400000   count = 1105
Freq = 48600000   count = 933
Freq = 48800000   count = 652
Freq = 49000000   count = 475
Freq = 49200000   count = 153
Freq = 49400000   count = 417
Freq = 49600000   count = 0
Freq = 49800000   count = 0
Freq = 50000000   count = 0
Freq = 50200000   count = 36
Freq = 50400000   count = 418
Freq = 50600000   count = 866
Freq = 50800000   count = 1109
Freq = 51000000   count = 1334
Freq = 51200000   count = 1552
Freq = 51400000   count = 1484
Freq = 51600000   count = 1577
Freq = 51800000   count = 1650
Freq = 52000000   count = 1724
Freq = 52200000   count = 1770
Enter->more, Q->Quit, S->Start
over, R->repeat:
```



```
r
Freq = 48000000   count = 1958
Freq = 48200000   count = 1592
Freq = 48400000   count = 1505
Freq = 48600000   count = 1457
Freq = 48800000   count = 1427
Freq = 49000000   count = 1342
Freq = 49200000   count = 1219
Freq = 49400000   count = 956
Freq = 49600000   count = 1056
Freq = 49800000   count = 792
Freq = 50000000   count = 435
Freq = 50200000   count = 64
Freq = 50400000   count = 6
Freq = 50600000   count = 0
Freq = 50800000   count = 5
Freq = 51000000   count = 115
Freq = 51200000   count = 318
Freq = 51400000   count = 855
Freq = 51600000   count = 1114
Freq = 51800000   count = 1306
Freq = 52000000   count = 1375
Freq = 52200000   count = 1492
Enter->more, Q->Quit, S->Start
over, R->repeat:
```

Here is how to manually calibrate your metal detector circuit. Automatic detection is left for the Projects section.

- ✓ Use the Propeller Tool to load CalibrateMetalDetector.spin into EEPROM (F11) and immediately click the Parallax Serial Terminal's *Enable* button. (Remember, you don't even have to wait for the program to finish loading.)
- ✓ When prompted, enter a starting frequency, try 48_000_000.
- ✓ When prompted, enter a frequency step, try 200_000.
- ✓ Compare your display to the left sweep shown in Figure 31, looking not so much for your values to match but that the overall profile is similar, which clearly indicates a resonant frequency centered where the count = 0.
- ✓ If your display is showing a clear resonant frequency, try placing a quarter coin directly under, but not quite touching, the metal loop, and press the R key on your keyboard to repeat the same frequency sweep.
- ✓ If your display changes significantly, like the right sweep shown in Figure 31, your metal detector apparently doesn't need any further calibration.
- ✓ If you are not seeing clear resonant frequencies, try refining the frequency start and frequency step values so that the sweep clearly indicates the presence and absence of metal.
- ✓ If there is no apparent filter response (either all zeros, or values that are larger without an apparent dip) try the suggestions below.

The circuit may instead need some tuning before it displays responses similar to those in Figure 31. If you instead see numbers that are either too high to show zeros or too low (all zeros), the voltage divider likely needs to be adjusted. It is designed to make the output just under the threshold voltage.

- ✓ If you see all zeros, the voltage divider needs to take less away from the signal. First, try successively larger resistors in place of R3. Try 1 kΩ, then 2 kΩ, then 10 kΩ.
- ✓ If the voltage divider is still taking too much away from the signal, disconnect R3 entirely, and instead add an R4 in parallel with R1. Start with a large resistor like 10 kΩ, and work downward again, 2 kΩ, 1 kΩ, and so on. Repeat the frequency sweep between each adjustment until you find a voltage divider that works for your circuit and Propeller chip's threshold voltage.
- ✓ If there is no apparent filter response, in other words, no cluster of low values like in Figure 31, you may need to search lower or higher frequencies after adjusting the voltage divider. This involves starting the sweep at lower values, like 46 MHz instead of 48, and using smaller increments, like 100_000 instead of 200_000, and selecting "M" or enter when prompted by Parallax Serial Terminal.
- ✓ Once you are getting good resonant frequencies, can you also discern the metal object's distance, say between 1 mm, 5 mm and 10 mm?

```
'' CalibrateMetalDetector.spin

CON

  _clkmode = xtal1 + pll16x            ' Set up 80 MHz internal clock
  _xinfreq = 5_000_000

  CLS = 16, CR = 13

OBJ
```

```
  Debug   : "FullDuplexSerialPlus"
  frq     : "SquareWave"


PUB Init | count, f, fstart, fstep, c

  'Start FullDuplexSerialPlus
  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*2 + cnt)
  Debug.tx(CLS)

  'Configure ctra module for 50 MHz square wave
  ctra[30..26] := %00010
  ctra[25..23] := %110
  ctra[8..0] := 15
  frq.Freq(0, 15, 50_000_000)
  dira[15]~~

  'Configure ctrb module for negative edge counting
  ctrb[30..26] := %01000
  ctrb[8..0] := 13
  frqb := 1

  c := "S"

  repeat' until c == "Q" or c == "q"

    case c
      "S", "s":
        Debug.Str(String("Starting Frequency: "))
        f := Debug.GetDec
        Debug.Str(String("Step size: "))
        fstep := Debug.GetDec
        Debug.tx(String(CR))

    case c
      "S", "s", 13, 10, "M", "m":
        repeat 22
          frq.Freq(0, 15, f)
          count := phsb
          waitcnt(clkfreq/10000 + cnt)
          count := phsb - count
          Debug.Str(String(CR, "Freq = "))
          Debug.Dec(f)
          Debug.Str(String("  count = "))
          Debug.Dec(count)
          waitcnt(clkfreq/20 + cnt)
          f += fstep

        Debug.str(String(CR,"Enter->more, Q->Quit, S->Start over, R->repeat: "))
        c := Debug.rx
        Debug.tx(CR)

      "R", "r":
        f -= (22 * fstep)
        c := "m"

      "Q", "q": quit

  Debug.str(String(10, 13, "Bye!"))
```

Questions

1) How many counter modules does each cog have, and what are they labeled?
2) What terms does this lab use to refer to a counter module's three registers without specifying which counter module is being used? In other words, what generic terms get used to refer to a counter module's three registers?
3) What are the three names used to refer to Counter A in Spin Code?
4) What are the three names used to refer to Counter B in Spin Code?
5) What register gets conditionally added to the PHS register with every clock tick?
6) What register can be used to set the condition(s) by which the PHS register gets updated?
7) How does the PHS register affect I/O pins with certain bits?
8) How does RC decay measurement indicate the state of an environmental variable?
9) Is a current limiting resistor necessary with an RC network connected to the Propeller chip?
10) It is possible to create an RC circuit that starts at 0 V and accumulates to 5 V during the measurement? What CTRMODE value would have to be used for measuring this kind of circuit?
11) How is a counter module's positive detector mode used to measure RC decay?
12) Where do the CTRMODE bits reside?
13) What do the CTRMODE bits select?
14) For RC decay measurements, which fields in the CTR register have to be set?
15) What value does the FRQ register have to store to make RC decay measurements?
16) What three steps are required to configure a counter module to take RC decay measurements?
17) Assuming a counter has been set up to take an RC decay measurement, what has to be done to start the measurement?
18) Why can RC decay measurements be taken concurrently?
19) How does a counter's interaction with an I/O pin differ between RC decay and D/A conversion applications?
20) How does the FRQ register control a duty mode D/A signal?
21) What component of the counter module actually controls the I/O pin?
22) What purpose does `scale = 16_777_216` serve in LedDutySweep.spin?
23) How are special-purpose registers 8 through 13 addressed?
24) What special-purpose register can be used to control the value of `ctrb`?
25) What special-purpose register can be used to set the value of FRQB?
26) What does `myVariable` hold after the command `myVariable := spr[13]` is executed?
27) What are two ways of assigning the value stored in `myVar` to `ctrb`?
28) How can you affect certain bits within `spr[8]` or `spr[9]`, and why is that useful?
29) What element of the counter special purpose registers controls and I/O pin in NCO Mode?
30) What's the condition for adding FRQ to PHS in NCO mode?
31) What ratio does the desired NCO frequency need to be multiplied by to determine the FRQ register value?
32) If a counter is set to NCO mode and a program copies a value to the counter's FRQ register, what ratio does the FRQ register need to be multiplied by to determine the frequency?
33) If an I/O pin is transmitting an NCO square wave, what are three ways of making it stop?
34) Can one cog send two square waves two unrelated frequencies?
35) What does a program have to do to change the NCO frequency a counter is transmitting?
36) Can a counter module be used to measure signal frequency?
37) Are POSEDGE and NEGEDGE incremented based on the edge of a signal?
38) There's a command that reads `repeat while phsb[31]` in BetterCountEdges.spin in the Faster Edge Detection section on page 34. Would it be possible to substitute a special purpose register in place of `phsb`?
39) What range of frequencies can a counter's PLL mode transmit?

40) What element from NCO mode does PLL use?
41) Unlike NCO mode, PLL mode does not use bit 31 of the PHS register to control the I/O pin. What happens to this signal?
42) What are the steps for calculating a PLL frequency given the values stored in the FRQ, PLLDIV, and CLKFRQ registers?
43) What are the steps for calculating FRQ and PLLDIV to synthesize a given PLL frequency?

Exercises

1) Modify TestRcDecay.spin so that it measures rise times instead of decay times.

```
...

~~' ctra[30..26] := %01000                  ' Set mode to "POS detector"~~
ctra[30..26] := %01100                     ' Set mode to "NEG detector"

...

   ~~' Charge RC circuit.~~
   ' Discharge RC circuit.

   ~~' dira[17] := outa[17] := 1                 ' Set pin to output-high~~
   dira[17] := 1                           ' Set pin to output-low
   outa[17] := 0

   ...
```

2) Initialize a single ended duty mode D/A conversion to 1 V on P7 using counter module B and the counter modules register names.
3) Initialize a single ended duty mode D/A conversion to 1 V on P7 using counter module B and special purpose registers. Be careful with using special purpose register array element that affects DIRA. In order to change just one bit in the entire DIRA register, you can take the existing value stored by the register and OR it with a mask with bit 7 set to 1.
4) Calculate the empty cells in Table 1 on page 18.
5) Assuming the Propeller chip's system clock is running at 20 MHz, write code to send a square wave approximation of the C7 note on P16 that uses Counter B.
6) Modify DoReMi.spin so that it plays all twelve notes from Table 1 on page 18.
7) Modify TwoTonesWithSquareWave.spin so that it correctly plays the notes with a 2 MHz crystal.
8) Modify IrDetector.spin so that it takes works on a scale of 0 to 128 instead of 0 to 256.
9) Modify CountEdgeTest.spin so that it counts positive instead of negative edges.
10) Modify 1Hz25PercentDutyCycle.spin so that it sends the center signal for a servo. This will cause a standard servo to hold a position in the center of its range of motion or a continuous rotation servo to stay still. The signal is a series of 1.5 ms pulses every 20 ms.

11) Modify 1Hz25PercentDutyCycle.spin so that it makes a servo's output sweeps from one extreme of its range of motion to the other in 1.5 seconds. For a 180 degree standard servo, the pulse durations should nominally sweep from 0.5 ms to 2.5 ms and back again. The pulses should still be delivered every 20 ms. In practice, it's good to make sure the servo doesn't attempt to turn beyond its mechanical stoppers. For Parallax standard servos, a safer range would be 0.7 to 2.2 ms.
12) Modify TestDualPwm so that it sweeps two servos between their opposite extremes of motion over a 1.5 second period.

Projects

1) Write a two channel a duty mode single ended DAC object that allows you to create and reclaim counter DAC channels (Counter A and Counter B). Each DAC channel should have its own resolution setting in terms of bits The DAC should support the test code and documentation shown below. If you are going with higher resolutions, remember to leave some room below the lowest and above the highest levels. See Tips for Setting Duty on page 12.

TEST CODE

```
''Test DAC 2 Channel.spin
''2 channel DAC.

OBJ

  dac : "DAC 2 Channel"

PUB TestDuty | level

  dac.Init(0, 4, 8, 0)               ' Ch0, P4, 8-bit DAC, starts at 0 V
  dac.Init(1, 5, 7, 64)              ' Ch1, P5, 7-bit DAC, starts at 1.65 V

  repeat
    repeat level from 0 to 256
      dac.Update(0, level)
      dac.Update(1, level + 64)      ' DAC output automatically truncated to 128
      waitcnt(clkfreq/100 + cnt)
```

OBJECT DOCUMENTATION

```
Object "DAC 2 Channel" Interface:

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

Program:      20 Longs
Variable:      2 Longs
```

```
PUB  Init(channel, ioPin, bits, level)
```

Initializes a DAC.

- channel - 0 or 1
- ioPin - Choose DAC I/O pin
- bits - Resolution (8 bits, 10 bits, etc.)
- level - Initial voltage level = 3.3 V * level ÷ 2^{bits}

```
PUB  Update(channel, level)
```

```
 Updates the level transmitted by an ADC channel to
                               bits
   level = 3.3 V * level ÷ 2
```

```
PUB  Remove(channel)
```

```
Reclaims the counter module and sets the associated I/O pin to input.
```

TIPS:

- Define a two long global variable lsb array to store the LSB for each DAC.
- The lsb variables are the adjustable versions of the scale constant in LedSweepWithSpr.spin.
- Define each lsb array element in the Init method using lsb[channel] := |< (32 - bits). For example if bits is 8, the encode operator sets bit 24 of the bits array element. What's the value? 16_777_216. That's the same as the scale constant that was declared for the 8-bit DAC in LedSweepWithSpr.spin.
- To set a voltage level, use spr[10 + channel] := level * lsb[channel], where level is the desired voltage level. For example, if bits is 8 (an 8-bit DAC), then a level of 128 would result in 1.65 V.

2) The solution for Exercise 12 (shown below) controls two servos using two counter modules. Each counter module in the repeat loop delivers a pulse in the 700 to 2200 µs range. Then the waitcnt command waits for the remaining 20 ms to elapse. The most time the servo pulses currently take is 2200 µs (2.2 ms). Since the repeat loop repeats every 20 ms, that leaves 17.8 ms for pulses to other servos. Modify the program so that it controls two more servos (for a total of four) during that 17.8 ms. Remember that the counters modules run independently, so you will have to insert delays to allow each pair of pulses to complete before moving on to the next pair.

```
{{
TestDualPWM.spin
Demonstrates using two counter modules to send a dual PWM signal.
The cycle time is the same for both signals, but the high times are independent of
each other.
}}

CON

  _clkmode = xtal1 + pll16x                     ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, tHb, t, us               ' <- Add us

  us := clkfreq/1_000_000                       ' <- Add

  ctra[30..26] := ctrb[30..26] := %00100        ' Counters A and B → NCO single-ended
  ctra[8..0] := 4                               ' Set pins for counters to control
  ctrb[8..0] := 6
  frqa := frqb := 1                             ' Add 1 to phs with each clock tick

  dira[4] := dira[6] := 1                       ' Set I/O pins to output

  tC := 20_000 * us                             ' <- Change Set up cycle time
  tHa := 700 * us                               ' <- Change Set up high times
  tHb := 2200 * us                              ' <- Change

  t := cnt                                      ' Mark current time.
  repeat tHa from (700 * us) to (2200 * us)  ' <- Change Repeat PWM signal
    phsa := -tHa                             ' Define and start the A pulse
    phsb := -tHb                             ' Define and start the B pulse
    t += tC                                  ' Calculate next cycle repeat
    waitcnt(t)                               ' Wait for next cycle
```

3) Develop an object that launches a cog and allows other objects to control its duty mode D/A conversion according to the object documentation below. Test this object with a top level object that uses a menu system to get D/A values from the user and pass them to control LED brightness.

```
''DualDac.spin

''Provides the two counter module channels from another cog for D/A conversion

How to Use this Object in Your Application
------------------------------------------
1) Declare variables the D/A channel(s).  Example:

   VAR
     ch[2]

2) Declare the DualDac object.  Example:

   OBJ
     dac : DualDac

3) Call the start method.  Example:

   PUB MethodInMyApp
     '...
     dac.start

4) Set D/A outputs.  Example:
     ch[0] := 3000
     ch[1] := 180

5) Configure the DAC Channel(s).  Example:
     'Channel 0, pin 4, 12-bit DAC, ch[0] stores the DAC value.
     dac.Config(0,4,12,@ch[0])
     'Since ch[0] was set to 3000 in Step 4, the DAC's P4 output will be
     ' 3.3V * (3000/4096)

     'Channel 1, pin 6, 8-bit DAC, ch[1] stores the DAC value.
     dac.Config(1,6,8,@ch[1])
     'Since ch[1] was set to 180 in Step 4, the DAC's P6 output will be
     ' 3.3V * (180/256)

6) Methods and features in this object als make it possible to:
       - remove a DAC channel
       - change a DAC channel's:
           o I/O pin
           o Resolution
           o Control variable address
           o Value stored by the control variable

See Also
--------
TestDualDac.spin for an application example.
```

```
Object "DualDac" Interface:

PUB  Start : okay
PUB  Stop
PUB  Config(channel, dacPin, resolution, dacAddress)
PUB  Remove(channel)
PUB  Update(channel, attribute, value)

Program:      73 Longs
Variable:     29 Longs

_________________
PUB  Start : okay

 Launches a new D/A cog.  Use Config method to set up a dac on a given pin.

_________
PUB  Stop

 Stops the DAC process and frees a cog.

____________________________________________________
PUB  Config(channel, dacPin, resolution, dacAddress)

 Configure a DAC.  Blocks program execution until other cog completes command.
   channel    - 0 = channel 0, 1 = channel 1
   dacPin     - I/O pin number that performs the D/A
   resolution - bits of D/A conversion (8 = 8 bits, 12 = 12 bits, etc.)
   dacAddress - address of the variable that holds the D/A conversion level,
                a value between 0 and (2^resolution) - 1.

____________________
PUB  Remove(channel)

 Remove a channel.  Sets channels I/O pin to input and clears the counter module.
 Blocks program execution until other cog completes command.

______________________________________
PUB  Update(channel, attribute, value)

 Update a DAC channel configuration.
 Blocks program execution until other cog completes command.
   channel    - 0 = channel 0, 1 = channel 1
   attribute  - the DAC attribute to update
     0 -> dacPin
     1 -> resolution
     2 -> dacAddr
     3 -> dacValue
   value      - the value of the attribute to be updated
```

Question Solutions

1) Each cog has two counter modules, A and B.
2) PHS, FRQ, and CTR.
3) PHSA, FRQA, and CTRA.
4) PHSB, FRQB, and CTRB.
5) The FRQ register.
6) The CTR (control) register.
7) In NCO mode, bit 31 of a given phs register is used to control one I/O pin in single ended mode, or two in differential mode. In PLL mode, the phase adder's carry flag (a.k.a PHS bit 32) controls the state of I/O pins

8) If a sensor's resistance or capacitance varies with an environmental variable, an RC decay measurement returns a time that's proportional to the sensor's value.
9) No, but it is with many other microcontrollers.
10) Yes, with NEG Detect mode, CTRMODE is %01100.
11) After the capacitor is fully charged, its voltage will take a certain amount of time to decay as its charge drains through the resistor. As a result, the voltage spends a certain amount of time above the I/O pin's 1.65 V logic threshold. For each clock tick that the voltage is above the I/O pin's logic threshold, it adds the value of FRQ to PHS. After the voltage has decayed below the threshold, FRQ no longer gets added to PHS, so PHS continues to store the number of ticks the signal was high.
12) In bits 30..26 of a given CTR register.
13) The mode of a given counter module's operation.
14) The CTRMODE and APIN fields.
15) The recommended value is 1, but so long as FRQ stores a non zero value that does not cause the PHS register to overflow during the measurement, it can be used to measure RC decay.
16) (1) set the CTR register's mode bit field. (2) Set the CTR register's PIN bit field. (3) Set the PHS register to a non zero value, preferably 1.
17) The capacitor in the RC circuit has to be charged. Then, the PHS register's initial value needs to be noted, or it can be cleared. Immediately after that, the I/O pin should be set to input.
18) Because each counter independently accumulates its PHS register based on the value at a given I/O pin. So, two counters can be accumulating their respective PHS registers while their respective RC circuits are decaying but still above the I/O pin's logic threshold. In the meantime, the cog can be executing other commands.
19) With RC decay measurements, the counter module is monitoring the voltage applied to an I/O pin. In D/A conversion, the counter module is controlling an I/O pin.
20) The ratio of FRQ to 2^{32} determines the duty.
21) The phase adder's carry bit, which you can think of as bit 32 of the PHS register.
22) It makes it possible for the code to select from 256 different levels instead of 2^{32} different leves.
23) `SPR[8]` through `SPR[13]`
24) `SPR[9]`
25) `SPR[10]`
26) The value in the `phsb` register.
27) (1) `ctrb := myVar`, and (2) `spr[9] := myVar`
28) `SPR[8]` is `ctra`, and `SPR[9]` is `ctrb`. Each of these registers has several bit fields that affect the counter's behavior. The left shift << operator can be used to shift a group of bits left to the correct position within one of these variables. A series of left shift operations can be combined with additions to determine each of a given CTR register's bit fields.
29) Bit 31 of the PHS register
30) The condition according to CTR.spin is Always, meaning that the FRQ register gets added to the PHS register with every clock cycle.
31) 2^{32}/`clkfreq`
32) `clkfreq`/2^{32}
33) (1) Set the I/O pin to input, (2) clear the FRQ register, or (3) clear bits 31..26 in the CNT register.
34) Yes, Counter A can send one signal, Counter B can send the other.
35) Update the value stored in the FRQ register.
36) Yes, either POSEDGE or NEGEDGE detectors can sample the number of transitions over a certain amount of time to store frequency. Positive and Negative detectors can also be used to track the cycle's high and low time, which can in turn be used to calculate the frequency of a signal.

37) No, they compare the I/O pin's current logic state to the previous clock tick's logic state. If, for example, the I/O pin's previous logic state was 0 and the current state is 1, POSEDGE mode would add FRQ to PHS because a positive edge transition occurred.
38) No. Although `spr[13]` refers to `phsb`, it is not bit addressable. The repeat while command is referring to a bit in the `phsb` register. Although it would be possible to determine the value of that bit using various operations, it would take a lot more time than simply checking `phsb[31]`
39) 500 kHz to 128 MHz
40) The counter module's PLL circuits needs to receive an input frequency from bit 31 of the PHS register. The value stored in the FRQ register determines the frequency of the PHS register's bit 31, just like it did in NCO mode.
41) The counter module's PLL circuit multiplies it by 16, then a divider reduces the frequency by a power of two that falls in the 1 to 128 range.
42) (a) Calculate the PHS bit 31 frequency. (b) Use the PHS bit 31 frequency to calculate the VCO frequency. (c) Divide the VCO frequency by $2^{7\text{-PLLDIV}}$.
43) (1) Figure out the PLLDIV, which is the power of two that the VCO frequency will have to be divided by to get the frequency the I/O pin will transmit. Page 48 has a useful table for this calculation. (2) Multiply the PLL frequency by $2^{(7\text{-PLLDIV})}$ to calculate the VCO frequency. (3) Given the VCO frequency, calculate 1/16 of that value, which is the PHS bit 31 (NCO) frequency that the PLL circuit will need. (4) Since the value stored in FRQ determines NCO frequency, use the NCO frequency to calculate the FRQ register value

Exercise Solutions

1) Solution:

```
...

~~' ctra[30..26] := %01000                         ' Set mode to "POS detector"~~
ctra[30..26] := %01100                           ' Set mode to "NEG detector"

...

   ~~' Charge RC circuit.~~
   ' Discharge RC circuit.

   ~~' dira[17] := outa[17] := 1                      ' Set pin to output-high~~
   dira[17] := 1                                 ' Set pin to output-low
   outa[17] := 0

   ...
```

2) Solution:

```
'The duty for this signal is 1/3.3.  Since duty = FRQ/2^32, we can solve 1/3.3 =
'FRQ/2^32 for FRQ.  FRQ = 1_301_505_241
ctrb[32..26] := %00110  ' Counter B to duty mode
ctrb[5..0] := 7
frqb := 1_301_505_241   ' Set duty for 3.3 V
dirb[7] := 1            ' Set P7 to output
```

3) Solution:

```
'The duty for this signal is 1/3.3.  Since duty = FRQ/2^32, we can solve 1/3.3 =
'FRQ/2^32 for FRQ.  FRQ = 1_301_505_241
spr[9] := (%00110<<26) + 7  ' Counter B to duty mode, transmit P7
spr[11] := 1_301_505_241    ' Set duty for 3.3 V
spr[6] |= |< 7              ' Set P7 to output
```

4) Using FRQ register = PHS bit 31 frequency × 2^{32} / (clkfreq = 80 MHz) rounded to the closest integer:
C6# → 59475, D6# → 66787, F6# → 79457, G6# → 89185, A6# → 100111
5) The frqa register will have to contain PHS bit 31 frequency × 2^{32} / (clkfreq = 20 MHz) = 224_734 (rounded to the closest integer).
```
  ctra[30..26] := %00100          ' Counter B to duty mode, transmit P16
  ctra[5..0] := 16
  frqb := 224_734                 ' 20 MHz C7
  dira[16]~~                      ' 20 MHz C7
```
6) Solution:
```
...

  '~~repeat index from 0 to 7~~
  repeat index from 0 to 12
...

DAT
'MODIFIED.........................................
'80 MHz frqa values for square wave musical note
' approximations with the counter module configured to NCO:
'          C6      C6#     D6      D6#     E6      F6      F6#
notes long 56_184, 59_475  63_066, 66787,  70_786, 74_995, 79457
'          G6      G6#     A6      A6#      B6       C7
      long 84_181, 89_185, 94_489, 100_111,  105_629, 112_528
```
7) Since the SquareWave object uses clkfreq to calculate its FRQ register values, the only change that needs to be made is _xinfreq = 2_000_000 instead of _xinfreq = 5_000_000.
8) Append scale = 16_777_216 with * 2, and then adjust the repeat loop from 0 to 255 to 0 to 127.
9) Change ctrb[30..26] := %01110 to ctrb[30..26] := %01010. To get full cycles, you can initialize the outa[27] high instead of low. This assumes a piezospeaker, which does not consume current when voltage is applied to it. Some speakers look like piezospeakers but have inductors built in, which draw a lot of current when DC voltage is applied.
10) Set tC to clkfreq/50 (that's 20 ms). For the 1.5 ms pulses, 1.5 × 10^{-3} × clkfreq is approximately equivalent to (1/667) × clkfreq, or clkfreq/667. So, tHa should be clkfreq/667. Another way to do it would be to add a con block with us = clkfreq/1_000_000. Then, tHa can be 1500 * us.
11) Add a con block with us = clkfreq/1_000_000. Initialize tC to 20_000 * us. Initialize tHa to 700 * us. Add a local variable named count to the TestPwm method. Change repeat to repeat tHa from (700 * us) to (2200 * us).
12) Solution:
```
{{
TestDualPWM(Exercise 12).spin
Demonstrates using two counter modules to send a dual PWM signal.
The cycle time is the same for both signals, but the high times are independent of
each other.
}}

CON

  _clkmode = xtal1 + pll16x                    ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, tHb, t, us              ' <- Add us

  us := clkfreq/1_000_000                      ' <- Add

  ctra[30..26] := ctrb[30..26] := %00100       ' Counters A and B → NCO single-ended
  ctra[8..0] := 4                              ' Set pins for counters to control
  ctrb[8..0] := 6
  frqa := frqb := 1                           ' Add 1 to phs with each clock tick

  dira[4] := dira[6] := 1                     ' Set I/O pins to output

  tC := 20_000 * us                           ' <- Change Set up cycle time
  tHa := 700 * us                             ' <- Change Set up high times
  tHb := 2200 * us                            ' <- Change

  t := cnt                                    ' Mark current time.

  repeat tHa from (700 * us) to (2200 * us)   ' <- Change Repeat PWM signal
    phsa := -tHa                              ' Define and start the A pulse
    phsb := -tHb                              ' Define and start the B pulse
    t += tC                                   ' Calculate next cycle repeat
    waitcnt(t)                                ' Wait for next cycle
```

Project Solutions

1) Solution:
Commented and uncommented versions of DAC 2 Channel.spin are shown below. Note in the uncommented version that it really doesn't take a lot of code to accomplish the project's specification.

```
''DAC 2 Channel.spin
''2 channel DAC object.  Each channel is configurable for both I/O pin and
''resolution (bits).

VAR
  ' Stores values that functions as an LSB scalars for the FRQ registers.
  long lsb[2]

PUB Init(channel, ioPin, bits, level)
{{
Initializes a DAC.
  • channel - 0 or 1
  • ioPin   - Choose DAC I/O pin
  • bits    - Resolution (8 bits, 10 bits, etc.)
                                                       bits
  • level   - Initial voltage level = 3.3 V * level ÷ 2
}}
  dira[ioPin]~                                 ' Set I/O pin to input
  spr[8 + channel] := (%00110 << 26) + ioPin   ' Configure CTR for duty mode and
                                               ' I/O pin
  lsb[channel] := |< (32 - bits)               ' Define LSB for FRQ register
  Update(channel, level)                       ' Set inital duty
  dira[ioPin] ~~                               ' Set I/O pin to output

PUB Update(channel, level)
'' Updates the level transmitted by an ADC channel to
''                              bits
''   level = 3.3 V * level ÷ 2
  spr[10 + channel] := level * lsb[channel]    ' Update DAC output

PUB Remove(channel)
''Reclaims the counter module and sets the associated I/O pin to input.
  dira[spr[8+channel] & %111111]~              ' Set I/O pin to input
  spr[8+channel]~                              ' Clear channel's CTR register
```

```
''DAC 2 Channel.spin (uncommented version)

VAR
```

```
  long lsb[2]


PUB Init(channel, ioPin, bits, level)

  dira[ioPin]~
  spr[8 + channel] := (%00110 << 26) + ioPin

  lsb[channel] := |< (32 - bits)
  Update(channel, level)
  dira[ioPin] ~~


PUB Update(channel, level)

  spr[10 + channel] := level * lsb[channel


PUB Remove(channel)

  dira[spr[8+channel] & %111111]~
  spr[8+channel]~
```

2) Solution:
Added lines are highlighted below. Let's assume the servos are connected to P5 and P7. In the `repeat` loop, the `ctra` and `ctrb` PIN fields will have to be set to 4 and 6 for the first pair of pulses, then changed to 5 and 7 for the second set of pulses. Also, a `waitcnt` has to be added after each pair of pulses so that the pulses have time to finish before moving on to the next pair of pulses.

At this point, the code still has about 15.6 ms left in the `repeat` loop, why not add a few more servos and make it a servo control object? See forums.parallax.com → Propeller Chip → Propeller Education Kit Labs → PE Kit Servo Control for an example.

```
{{
TestDualPWM (Project 2).spin
Demonstrates using two counter modules to send a dual PWM signal.
The cycle time is the same for both signals, but the high times are independent of
each other.

Modified to control four servos.

}}

CON

  _clkmode = xtal1 + pll16x                   ' System clock → 80 MHz
  _xinfreq = 5_000_000

PUB TestPwm | tc, tHa, tHb, t, us             ' <- Add us

  us := clkfreq/1_000_000                     ' <- Add

  ctra[30..26] := ctrb[30..26] := %00100      ' Counters A and B → NCO single-ended
  ctra[8..0] := 4                             ' Set pins for counters to control
  ctrb[8..0] := 6
  frqa := frqb := 1                           ' Add 1 to phs with each clock tick

  dira[4] := dira[6] := 1                     ' Set I/O pins to output
  dira[5] := dira[7] := 1

  tC := 20_000 * us                           ' <- Change Set up cycle time
  tHa := 700 * us                             ' <- Change Set up high times
  tHb := 2200 * us                             ' <- Change

  t := cnt                                     ' Mark current time.

  repeat tHa from (700 * us) to (2200 * us)   ' <- Change Repeat PWM signal

    ' First pair of pulses
    ctra[8..0] := 4                            ' Set pins for counters to control
    ctrb[8..0] := 6
    phsa := -tHa                               ' Define and start the A pulse
    phsb := -tHb                               ' Define and start the B pulse
    waitcnt(2200 * us + cnt)                   ' Wait for pulses to finish

    ' Second pair of pulses
    ctra[8..0] := 5                            ' Set pins for counters to control
    ctrb[8..0] := 7
    phsa := -tHa                               ' Define and start the A pulse
    phsb := -tHb                               ' Define and start the B pulse
    waitcnt(2200 * us + cnt)                   ' Wait for pulses to finish

    ' Wait for 20 ms cycle to complete before repeating loop
    t += tC                                    ' Calculate next cycle repeat
    waitcnt(t)                                 ' Wait for next cycle
```

3) DAC Object Solution: (It works, but keep in mind that it's not the only possible solution.)

```
{{
''DualDac.spin

''Provides the two counter module channels from another cog for D/A conversion

How to Use this Object in Your Application
------------------------------------------
1) Declare variables the D/A channel(s).  Example:

   VAR
     ch[2]

2) Declare the DualDac object.  Example:

   OBJ
     dac : DualDac

3) Call the start method.  Example:

   PUB MethodInMyApp
     '...
     dac.start

4) Set D/A outputs.  Example:
     ch[0] := 3000
     ch[1] := 180

5) Configure the DAC Channel(s).  Example:
     'Channel 0, pin 4, 12-bit DAC, ch[0] stores the DAC value.
     dac.Config(0,4,12,@ch[0])
     'Since ch[0] was set to 3000 in Step 4, the DAC's P4 output will be
     '3.3V * (3000/4096)

     'Channel 1, pin 6, 8-bit DAC, ch[1] stores the DAC value.
     dac.Config(1,6,8,@ch[1])
     'Since ch[1] was set to 180 in Step 4, the DAC's P6 output will be
     ' 3.3V * (180/256)

6) Methods and features in this object als make it possible to:
       - remove a DAC channel
       - change a DAC channel's:
           o I/O pin
           o Resolution
           o Control variable address
           o Value stored by the control variable

See Also
--------
TestDualDac.spin for an application example.

}}

VAR                                            ' Global variables
  long cog, stack[20]                          ' For object
  long cmd, ch, pin[2], dacAddr[2], bits[2]    ' For cog info exhcanges

PUB Start : okay

  '' Launches a new D/A cog.  Use Config method to set up a dac on a given pin.

  okay := cog := cognew(DacLoop, @stack) + 1

PUB Stop

  '' Stops the DAC process and frees a cog.

  if cog
    cogstop(cog~ - 1)

PUB Config(channel, dacPin, resolution, dacAddress)

  '' Configure a DAC.  Blocks program execution until other cog completes command.
  ''   channel    - 0 = channel 0, 1 = channel 1
  ''   dacPin     - I/O pin number that performs the D/A
  ''   resolution - bits of D/A conversion (8 = 8 bits, 12 = 12 bits, etc.)
  ''   dacAddress - address of the variable that holds the D/A conversion level,
  ''                a value between 0 and (2^resolution) - 1.

  ch               :=  channel             ' Copy paramaters to global variables.
  pin[channel]     :=  dacPin
  bits[channel]    :=  |<(32-resolution)
  dacAddr[channel] :=  dacAddress
  cmd              :=  4                   ' Set command for PRI DacLoop.
  repeat while cmd                         ' Block execution until cmd completed.

PUB Remove(channel)

  '' Remove a channel.  Sets channels I/O pin to input and clears the counter
module.
  '' Blocks program execution until other cog completes command.

  ch   :=  channel                         ' Copy parameter to global variable.
  cmd  :=  5                               ' Set command for PRI DacLoop.
  repeat while cmd                         ' Block execution until cmd completed.

PUB Update(channel, attribute, value)

  '' Update a DAC channel configuration.
  '' Blocks program execution until other cog completes command.
  ''   channel    - 0 = channel 0, 1 = channel 1
  ''   attribute  - the DAC attribute to update
  ''     0 -> dacPin
  ''     1 -> resolution
  ''     2 -> dacAddr
  ''     3 -> dacValue
  ''   value      - the value of the attribute to be updated

  ch  := channel                            ' Copy parameter to global variable.
  case attribute                            ' Use attribute param to decide what
to do.
    0 :                                     ' 0 = change DAC pin.
      cmd := attribute + (value << 16)      ' I/O pin in upper 16 bits, lower 16
cmd = 0.
    ' Options 1 through 3 do not require a command for PRI DacLoop -> PRI
DacConfig.
    ' They just require that certain global variables be updated.
    1 : bits[ch] := |<(32-value)            ' 1 = Change resolution.
    2 : dacAddr[channel] := value           ' 2 = Change control variable address.
    3 : long[dacAddr] := value              ' 3 = Change control variable value
  repeat while cmd                          ' Block execution until cmd completed.

PRI DacLoop | i                             ' Loop checks for cmd, then updates
                                            ' DAC output values.
  repeat                                    ' Main loop for launched cog.
    if cmd                                  ' If cmd <> 0
       DacConfig                            '    then call DatConfig
    repeat i from 0 to 1                    ' Update counter module FRQA & FRQB.
       spr[10+ch] := long[dacAddr][ch] * bits[ch]

PRI DacConfig | temp                        ' Update DAC configuration based on
cmd.

  temp := cmd >> 16                         ' If update attribute = 0, temp gets
pin.
  case cmd & $FF                            ' Mask cmd and evalueate case by case.
    4:                                      ' 4 -> Configure DAC.
      spr[8+ch] := (%00110 << 26) + pin[ch] ' Store mode and pin in CTR register.
      dira[pin[ch]]~~                       ' Pin direction -> outpup.
    5:                                      ' 5 -> Remove DAC.
      spr[8+ch]~                            ' Clear CTR register.
      dira[pin[ch]]~                        ' Make I/O pin input.
    0:                                      ' 0 -> update pin.
      dira[pin[ch]]~                        ' Make old pin input.
      pin[ch] := temp                       ' Get new pin from temp local
variable.
      spr[8+ch] := (%00110 << 26) + pin[ch] ' Update CTR with new pin.
      dira[pin[ch]]~~                       ' Update new I/O pin direction ->
output.
  cmd := 0                              ' Clear cmd to stop blocking in other cog.
```

Solution - Menu driven test object for DualDac.spin

```
''TestDualDAC.spin
''Menu driver user tests for DualDac.spin

CON

  _clkmode = xtal1 + pll16x                  ' System clock → 80 MHz
  _xinfreq = 5_000_000

  ' Parallax Serial Terminal constants
  CLS = 16, CR = 13, CLREOL = 11, CRSRXY = 2, BKSPC = 8, CLRDN = 12

OBJ

  debug : "FullDuplexSerialPlus"
  dac   : "DualDAC"

PUB TestPwm | channel, dacPin, resolution, ch[2], menu, choice
```

```
  debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq * 2 + cnt)
  debug.str(@_Menu)

  dac.start

  repeat

    debug.tx(">")
    case menu := debug.rx
      "C", "c":
         debug.str(@_Channel)
         channel := debug.getdec
         debug.str(@_Pin)
         dacPin := debug.getdec
         debug.str(@_Resolution)
         resolution := debug.getdec
         dac.Config(channel, dacPin, resolution, @ch[channel])
      "S", "s":
         debug.str(@_Channel)
         channel := debug.getdec
         debug.str(@_Value)
         ch[channel] := debug.getdec
      "U", "u":
         debug.str(@_Update)
         case choice := debug.rx
            "P", "p":
               debug.str(@_Channel)
               channel := debug.getdec
               debug.str(@_Pin)
               dacPin := debug.getdec
               dac.update(channel, 0, dacPin)
            "B", "b":
               debug.str(@_Channel)
               channel := debug.getdec
               debug.str(@_Resolution)
               resolution := debug.getdec
               dac.update(channel, 1, resolution)
      "R", "r":
         debug.str(@_Channel)
         channel := debug.getdec
         dac.Remove(channel)
    debug.str(String(CRSRXY, 1,4, BKSPC, CLRDN))

DAT
_Menu       byte CLS, "C = Configure DAC", CR, "S = Set DAC Output", CR
            byte "U = Update DAC Config", CR, "R = Remove DAC", CR, 0
_Channel    byte CR, "Channel (0/1) > ", 0
_Pin        byte "Pin > ", 0
_Resolution byte "Resolution (bits) > ", 0
_Value      byte "Value > ", 0
_Update     byte "Update Choices:", CR, "P = DAC Pin", CR,"B = Bits (resolution)"
            byte CR, 0
```

Appendix A – SquareWave Object

```
'' From Parallax Inc. Propeller Education Kit - Counters & Circuits Lab
'' SquareWave.spin

'' Can be used to make either or both of a given cog's counter modules transmit square
'' waves.

PUB Freq(Module, Pin, Frequency) | s, d, ctr

'' Determine CTR settings for synthesis of 0..128 MHz in 1 Hz steps
''
'' in:    Pin = pin to output frequency on
''        Freq = actual Hz to synthesize
''
'' out:   ctr and frq hold ctra/ctrb and frqa/frqb values
''
''   Uses NCO mode %00100 for 0..499_999 Hz
''   Uses PLL mode %00010 for 500_000..128_000_000 Hz
''

  Frequency := Frequency #> 0 <# 128_000_000     'limit frequency range

  if Frequency < 500_000                     'if 0 to 499_999 Hz,
    ctr := constant(%00100 << 26)            '..set NCO mode
    s := 1                                   '..shift = 1

  else                                       'if 500_000 to 128_000_000 Hz,
    ctr := constant(%00010 << 26)            '..set PLL mode
    d := >|((Frequency - 1) / 1_000_000)     'determine PLLDIV
    s := 4 - d                               'determine shift
    ctr |= d << 23                           'set PLLDIV

  spr[10 + module] := fraction(Frequency, CLKFREQ, s)    'Compute frqa/frqb value
  ctr |= Pin                                 'set PINA to complete ctra/ctrb value
  spr[8 + module] := ctr

  dira[pin]~~


PUB NcoFrqReg(frequency) : frqReg
{{
Returns frqReg = frequency × (2³² ÷ clkfreq) calculated with binary long
division.  This is faster than the floating point library, and takes less
code space.  This method is an adaptation of the CTR object's fraction
method.
}}
  frqReg := fraction(frequency, clkfreq, 1)


PRI fraction(a, b, shift) : f

  if shift > 0                               'if shift, pre-shift a or b left
    a <<= shift                              'to maintain significant bits while
  if shift < 0                               'insuring proper result
    b <<= -shift

  repeat 32                                  'perform long division of a/b
    f <<= 1
    if a => b
      a -= b
      f++
    a <<= 1
```