Fundamentals: Propeller I/O and Timing Basics

PROPELLER EDUCATION KIT LAB SERIES

Overview

Most microcontroller applications involve reading inputs, making decisions, and controlling outputs. They also tend to be timing-sensitive, with the microcontroller determining when inputs are monitored and outputs are updated. The pushbutton circuits in this lab will provide simple outputs that the example applications can monitor with Propeller I/O pins set to input. Likewise, LED circuits will provide a simple and effective means of monitoring propeller I/O pin outputs and event timing.

While this lab's pushbutton and LED example applications might seem rather simple, they make it possible to clearly present a number of important coding techniques that will be used and reused in later labs. Here is a list of this lab's *example applications* and the *coding techniques* they introduce:

- ***Turn an LED on*** – *assigning I/O pin direction and output state*
- ***Turn groups of LEDs on*** – *group I/O assignments*
- ***Signal a pushbutton state with an LED*** – *monitoring an input, and setting an output accordingly*
- ***Signal a group of pushbutton states with LEDs*** – *parallel I/O, monitoring a group of inputs and writing to a group of outputs*
- ***Synchronized LED on/off signals*** – *event timing based on a register that counts clock ticks*
- ***Configure the Propeller chip's system clock*** – *choosing a clock source and configuring the Propeller chip's Phase-Locked Loop (PLL) frequency multiplier*
- ***Display on/off patterns*** – *Introduction to more Spin operators commonly used on I/O registers*
- ***Display binary counts*** – *introductions to several types of operators and conditional looping code block execution*
- ***Shift a light display*** – *conditional code block execution and shift operations*
- ***Shift a light display with pushbutton-controlled refresh rate*** – *global and local variables and more conditional code block execution*
- ***Timekeeping application with binary LED display of seconds*** – *Introduction to synchronized event timing that can function independently of other tasks in a given cog.*

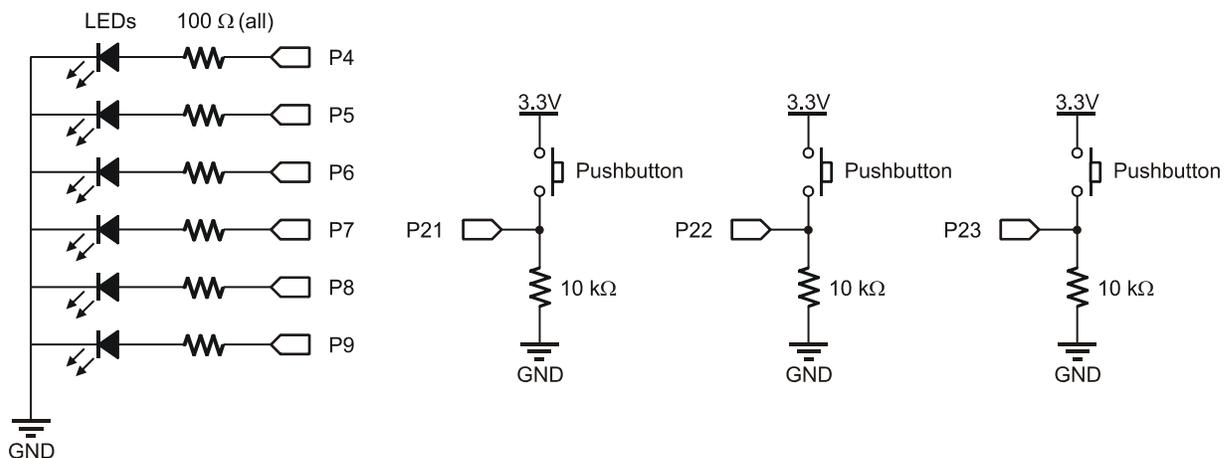## Prerequisites

PE Lab – Setup and Testing

Parts List and Schematic

This lab will use six LED circuits and three pushbutton circuits.

    (6) LEDs – assorted colors
    (6) Resistors – 100 Ω
    (3) Resistor – 10 kΩ
    (3) Pushbutton – normally open
    (misc) jumper wires

    ✓ Build the schematic shown in Figure 1.

Figure 1: LED Pushbutton Schematic

Propeller Nomenclature

The Propeller microcontroller's documentation makes frequent references to *cogs*, *Spin*, *objects*, *methods*, and *global and local variables*. Here are brief explanations of each term:

- **Cog** – a processor inside the Propeller chip. The Propeller chip has eight cogs, making it possible to perform lots of tasks in parallel. The Propeller is like a super-microcontroller with eight high speed 32-bit microcontrollers inside. Each internal microcontroller (cog) has access to the Propeller chip's I/O pins and 32 KB of global RAM. Each cog also has its own 2 KB of ram that can either run a Spin code interpreter or an assembly language program.

- **Spin and assembly languages** – The Spin language is the high-level programming language created by Parallax for the Propeller chip. Cogs executing Spin code do so by loading a Spin interpreter from the Propeller chip's ROM. This interpreter fetches and executes Spin command codes that get stored in the Propeller chip's Global Ram.

  Propeller cogs can also be programmed in low-level assembly language. Whereas high-level Spin tells a cog what to do, low-level assembly language tells a cog how to do it. Assembly language generates machine codes that reside in a cog's RAM and get executed directly by the cog. Assembly language programs make it possible to write code that optimizes a cog's performance; however, it requires a more in-depth understanding of the Propeller chip's architecture. The PE Kit Fundamentals labs focus on Spin programming. Assembly language programming will be introduced in Intermediate level PE kit labs.

- **Method** – a block of executable Spin commands that has a name, access rule, and can optionally receive and return parameter values and create local (temporary) variables.

- **Global and local variables** – *Global variables* are available to all the methods in a given object, and they reserve variable space as long as an application is running. *Local variables* are defined in a method, can only be used within that method, and only exist while that method executes commands. When it's done, the memory these local variables used becomes available to for other methods and their local variables. Local and global variables are defined with different syntax.

- **Object** – an application building block comprised of all the code in a given .spin file. Some Propeller applications use just one object but most use several. Objects have a variety of uses, depending partially on how they are written and partially on how they get configured and used by other objects. Some objects serve as *top level objects*, which provide the starting point where the first command in a given application gets executed. Other objects are written to provide a library of useful methods for top level or other objects to use.

  Objects can be written to use just one cog, or can include code that gets launched into one or more cogs. Some objects have methods that provide a means to exchange information with processes running in other cogs. One object can even make multiple copies of another object, and set each one to a different task. Objects can use other objects, which in turn can use still other objects. In more complex applications, a set of objects will form functional relationships that can be viewed as a file structure with the Propeller Tool's Object Info window.

The examples in this lab only involve single, top-level objects with just one method. Upcoming labs will introduce various building-block techniques for using multiple objects and methods in an application, as well as parallel multiprocessing applications using multiple cogs. Though the objects in this lab are simple, many of them will be modified later to serve as building blocks for other objects and/or future projects.

Lights on with Direction and Output Register Bits

The LedOnP4 object shown below has a method named `LedOn`, with commands that instruct a cog in the Propeller Chip to set its P4 I/O pin to output-high. This in turn causes the LED in the circuit connected to P4 to emit light.

✓ Load LedOnP4 into RAM by clicking Run → Compile Current → Load RAM (or press F10).

```
'' File: LedOnP4.spin

PUB LedOn                              ' Method declaration

   dira[4] := 1                        ' Set P4 to output
   outa[4] := 1                        ' Set P4 high

   repeat                              ' Endless loop prevents program from ending
```

How LedOnP4.spin Works

The first lines in the program is a documentation comment. Single-line documentation comments are denoted by two apostrophes (not a quotation mark) to the left of the documentation text.

✓ Click the Documentation radio button above the code in the Propeller Editor.

While commands like `dira :=`... and `repeat` don't show in documentation mode, notice that the text to the right of the double apostrophe documentation comments does appear. Notice also that the non-documentation comments in the code, preceded by single apostrophes, do not appear in Documentation mode.

✓ Try the other radio buttons and note what elements of the object they do/do not show.

> ⓘ **Block Comments:** There are also documentation block comments that can span multiple lines. They have to begin and end with double-braces like this: `{{ block of documentation comments }}`. Non-documentation comments can also span multiple lines, beginning and ending with single-braces like this: `{ block of non-documentation comments }`.

All Spin language commands that the Propeller chip executes have to be contained within a *method block*. Every method block has to be declared with at least an access rule and a name. Access rules and method names will be explored in depth in upcoming labs; for now, just keep in mind that **PUB** LedOn is a method block declaration with a public (**PUB**) access rule and the name LedOn.

> ⓘ **Bold or not bold?** In the discussion paragraphs, the Parallax font used in the Propeller Tool is also used for all text that is part of a program. The portions that are reserved words and necessary syntax will be in bold. The portions that are defined by the user, such as method, variable, and constant names and values, will not be in bold text. This mimics the Propeller Tool software's syntax highlighting. Code listings and snippets are not given the extra bolding. To see the full syntax-highlighted version, view it in the Propeller Tool.

The `dira` register is one of several special-purpose registers in Cog RAM; you can read and write to the `dira` register, which stores I/O pin directions for each I/O pin. A 1 in a given `dira` register bit sets that I/O pin to output; a 0 sets it to input. The symbol ":=" is the Assignment operator; the command `dira[4] := 1` assigns the value 1 to the `dira` register's Bit 4, which makes P4 an output. When an I/O pin is set to output, the value of its bit in the `outa` register either sets the I/O pin high (3.3 V) with a 1, or low (0 V) with a 0. The command `outa[4] := 1` sets I/O pin P4 high. Since the P4 LED circuit terminates at ground, the result is that the LED emits light.

> ⓘ **I/O Sharing among Cogs?** Each cog has its own I/O Output (`outa`) and I/O Direction (`dira`) registers. Since our applications use only one cog, we do not have to worry about two cogs trying to use the same I/O pin for different purposes at the same time. When multiple cogs are used in one application, each I/O pin 's direction and output state is the "wired--OR" of the entire cogs collective. How this works logically is described in the I/O Pin section in Chapter 1 of the Propeller Manual.

The `repeat` command is one of the Spin language's conditional commands. It can cause a block of commands to execute repeatedly based on various conditions. For `repeat` to affect a certain block of commands, they have to be below it and indented further by at least one space. The next command that is not indented further than `repeat` is not part of the block, and will be the next command executed after the `repeat` loop is done.

Since there's nothing below the `repeat` command in the LedOnP4 object, it just repeats itself over and over again. This command is necessary to prevent the Propeller chip from automatically going into low power mode after it runs out of commands to execute. If the `repeat` command weren't there,

the LED would turn on too briefly to see, and then the chip would go into low power mode. To our eyes it would appear that nothing happened.

Modifying LedOnP4

More than one assignment can be made on one line.

- ✓ Replace this:

```
dira[4] := 1
outa[4] := 1
```

    ...with this:

```
dira[4] := outa[4] := 1
```

Of course, you can also expand the LedOn method so that it turns on more than one LED.

- ✓ Modify the LedOn method as shown here to turn on both the P4 and P5 LEDs:

```
PUB LedOn

    dira[4] := outa[4] := 1
    dira[5] := outa[5] := 1

    repeat
```

If the repeat command was not the last command in the method, the LEDs would turn back off again so quickly that it could not be visually discerned as on for any amount of time. Only an oscilloscope or certain external circuits would be able to catch the brief "on" state.

- ✓ Try running the program with the repeat command commented with an apostrophe to its left.
- ✓ If you have an oscilloscope, set it to capture a single edge, and see if you can detect the signal.

I/O Pin Group Operations

The Spin language has provisions for assigning values to groups of bits in the dira and outa registers. Instead of using a single digit between the brackets next to the outa command, two values separated by two dots can be used to denote a contiguous group of bits. The binary number indicator % provides a convenient way of defining the bit patterns that get assigned to the group of bits in the outa or dira registers. For example, dira[4..9] := %111111 will set bits 4 through 9 in the dira register (to output.) Another example, outa[4..9] := %101010 sets P4, clears P5, sets P6, and so on. The result should be that the LED's connected to P4, P6, and P8 turn on while the others stay off.

- ✓ Load GroupIoSet into RAM (F10).
- ✓ Verify that the P4, P6, and P8 LEDs turn on.

```
'' File: GroupIoSet.spin

PUB LedsOn

    dira[4..9] := %111111
    outa[4..9] := %101010

    repeat
```

Modifying GrouploSet.spin

Notice that `outa[4..9] := %101010` causes the state of the `outa` register's bit 4 to be set (to 1), bit 5 cleared (to 0), and so on. If the pin group's start and end values are swapped, the same bit pattern will cause bit 9 to be set, bit 8 to be cleared, and so on…

- ✓ Replace

    ```
    outa[4..9] := %101010
    ```

    with this

    ```
    outa[9..4] := %101010
    ```

- ✓ Load the modified program into the Propeller chip's RAM and verify that the LEDs display a reversed bit pattern.

It doesn't matter what value is in an `outa` register bit if its `dira` register bit is zero. That's because the I/O pin functions as an input instead of an output when its `dira` register bit is cleared. An I/O pin set to input has no effect on circuits connected to the pin. Setting all the bits in `outa[4..9]` but not all the bits in `dira[4...9]` demonstrates how this works.

- ✓ Set all the `outa[4..9]` bits

    ```
    outa[4..9] := %111111
    ```

- ✓ Clear bits 6 and 7 in `dira[4..9]`

    ```
    dira[4..9] := %110011
    ```

- ✓ Load the modified program into the Propeller chip's RAM and verify that the `outa[6]` and `outa[7]` values have no effect on the P6 and P7 LEDs because their I/O pins are inputs instead of outputs.

Reading an Input, Controlling an Output

The `ina` register is a read-only register in Cog RAM whose bits store the voltage state of each I/O pin. When an I/O pin is set to output, its `ina` register bit will report the same value as the `outa` register bit since `ina` bits indicate high/low I/O pin voltages with 1 and 0. If the I/O pin is instead an input, its `ina` register bit updates based on the voltage applied to it. If a voltage above the I/O pin's 1.65 V logic threshold is applied, the `ina` register bit stores a 1; otherwise, it stores a 0. The `ina` register is updated with the voltage states of the I/O pins each time an `ina` command is issued to read this register.

The pushbutton connected to P21 will apply 3.3 V to P21 when pressed, or 0 V when not pressed. In the ButtonToLed object below, `dira[21]` is set to 0, making I/O pin P21 function as an input. So, it will store 1 if the P21 pushbutton is pressed, or 0 if it is not pressed. By repeatedly assigning the value stored in `ina[21]` to `outa[6]`, the `ButtonLed` method makes the P6 LED light whenever the P21 pushbutton is pressed. Notice also that the command `outa[6] := ina[21]` is indented below the `repeat` command, which causes this line to get executed over and over again indefinitely.

- ✓ Load ButtonToLed into RAM.
- ✓ Press and hold the pushbutton connected to P21 and verify that the LED connected to P6 lights while the pushbutton is held down.

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

PUB ButtonLed                           ' Pushbutton/Led Method

    dira[6]  := 1                       ' P6  → output
    dira[21] := 0                       ' P21 → input (this command is redundant)

    repeat                              ' Endless loop

        outa[6] := ina[21]             ' Copy P21 input to P6 ouput
```

Read Multiple Inputs, Control Multiple Outputs

A group of bits can be copied from the `ina` to `outa` registers with a command like `outa[6..4] :=`
`ina[21..23]`. The `dira[6] := 1` command will also have to be changed to `dira[6..4] := %111`
before the pushbuttons will make the LEDs light up.

✓ Save a copy of ButtonToLed, and modify it so that it makes the P23, P22, and P21
  pushbuttons light up the P4, P5 and P6 LEDs respectively. Hint: you need only one `outa`
  command.
✓ Try reversing the order of the pins in `outa[6..4]`. How does this affect the way the
  pushbutton inputs map to the LED outputs? What happens if you reverse the order of bits in
  `ina[21..23]`?

Timing Delays with the System Clock

Certain I/O operations are much easier to study with code that controls the timing of certain events,
such as when an LED lights or how long a pushbutton is pressed. The three basic Spin building
blocks for event timing are:

- `cnt` – a register in the Propeller chip that counts system clock ticks.
- `clkfreq` – a command that returns the Propeller chip's system clock frequency in Hz.
  Another useful way to think of it is as a value that stores the number of Propeller system
  clock ticks in one second.
- `waitcnt` – a command that waits for the `cnt` register to get to a certain value.

The `waitcnt` command waits for the `cnt` register to reach the value between its parentheses. To
control the amount of time `waitcnt` waits, it's best to add the number of clock ticks you want to wait
to `cnt`, the current number of clock ticks that have elapsed.

The example below adds `clkfreq`, the number of clock ticks in 1 second, to the current value of `cnt`.
The result of the calculation between the parentheses is the value the `cnt` register will reach 1 s later.
When the `cnt` register reaches that value, `waitcnt` lets the program move on to the next command.

```
        waitcnt(clkfreq + cnt)          ' wait for 1 s.
```

To calculate delays that last for fractions of a second, simply divide `clkfreq` by a value before adding
it to the `cnt` register. For example, here is a `waitcnt` command that delays for a third of a second, and
another that delays for 1 ms.

```
        waitcnt(clkfreq/3 + cnt)        ' wait for 1/3 s
        waitcnt(clkfreq/1000 + cnt)     ' wait for 1 ms
```

The LedOnOffP4 object uses the `waitcnt` command to set P4 on, wait for ¼ s, turn P4 off, and wait
for ¾ s. The LED will flash on/off at 1 Hz, and it will stay on for 25 % of the time.

```
'' File: LedOnOffP4.spin

PUB LedOnOff

    dira[4] := 1

    repeat

        outa[4] := 1
        waitcnt(clkfreq/4 + cnt)
        outa[4] := 0
        waitcnt(clkfreq/4*3 + cnt)
```
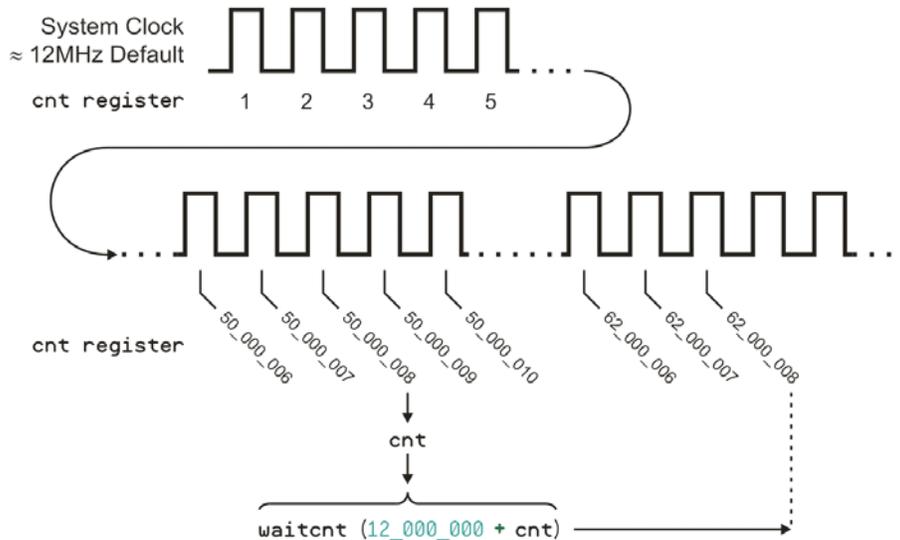
- ✓ Load LedOnOffP4 object into the Propeller chip's RAM and verify that the light flashes roughly every second, on ¼ of the time and off ¾ of the time.

!

Remember that indentation is important! Figure 2 shows a common mistake that can cause unexpected results. On the left, all four lines below the `repeat` command are indented further than `repeat`. This means they are nested in the `repeat` command, and all four commands will be repeated. On the right, the lines below `repeat` are not indented. They are at the same level as the `repeat` command. In that case, the program never gets to them because the `repeat` loop does nothing over and over again instead!

Notice the faint lines that connect the `r` in `repeat` to the commands below it. These lines indicate the commands in the block that `repeat` operates on.

- ✓ To enable this feature in the Propeller Tool software, click Edit and select Preferences. Under the *Appearance* tab, click the checkmark box next to *Show Block Group Indicators*.

Figure 2: Repeat Code Block

This `repeat` loop repeats four commands

The commands below repeat are not indented further, so they are not part of the `repeat` loop.

Inside waitcnt(clkfreq + cnt)

When *Run → Compile Current → Load…* is used to download an object, the Propeller Tool software examines it for certain constant declarations that configure the Propeller chip's system clock. If the object does not have any such clock configuration constants, the Propeller Tool software stores default values in the Propeller chip's CLK register which set it to use the internal RC oscillator to fast mode (approximately 12 MHz) for the system clock. With the default 12 MHz system clock, the instruction `waitcnt(clkfreq + cnt)` is equivalent to the instruction `waitcnt(12_000_000 + cnt)`.

Figure 3 shows how `waitcnt(12_000_000 + cnt)` waits for the `cnt` register to accumulate to 12 million more clock ticks than when the `waitcnt` command started. Keep in mind that the `cnt` register has been incrementing with every clock tick since the Propeller chip was either reset or booted. In this example, it has reached the 50,000,008[th] clock tick at the point when the `waitcnt` command is

executed. The `cnt` value `waitcnt` waits for is 12_000_000 + 50_000_008 = 62_000_008. So, the cog executing `waitcnt(12_000_000 + cnt)` is allowed to move on to the next command after the `cnt` register has counted the 62,000,008th clock tick.

Figure 3: waitcnt Command and the cnt Register

System Clock Configuration and Event Timing

Up to this point, our programs have been using the Propeller chip's default internal 12 MHz clock. Next, let's modify them to use the external 5 MHz oscillator in our PE Platform circuit. Both Spin and Propeller assembly have provisions for declaring constants that configure the system clock and making sure that all the objects know its current operating frequency. The `CON` block designator defines a section of code for declaring Propeller configuration settings, as well as global constant symbols for program use.

Declarations similar to ones in the `CON` block below can be added to a top level object to configure the Propeller chip's system clock. This particular set of declarations will make the Propeller chip's system clock run at top speed, 80 MHz.

```
CON
  _xinfreq = 5_000_000
  _clkmode = xtal1 + pll16x
```

The line `_xinfreq = 5_000_000` defines the expected frequency from the external oscillator, which in the PE Platform's case is 5 MHz. The line `_clkmode = xtal1 + pll16x` causes the Propeller Tool software's Spin compiler to set certain bits in the chip's `CLK` register when it downloads the program. The `xtal1` clock mode setting configures certain XO and XI pin circuit characteristics to work with external crystals in the 4 to 16 MHz range.

The frequency of the external crystal provides the input clock signal which the Propeller chip's Phase-Locked Loop (PLL) circuit multiplies for the system clock. `pll16x` is a predefined clock mode setting constant which makes the PLL circuit multiply the 5 MHz frequency by 16 to supply the system with an 80 MHz clock signal. The constant `pll8x` can be used with the same oscillator to run the Propeller chip's system clock at 40 MHz. `pll4x` will make the Propeller chip's system clock run at 20 Mhz, and so on. The full listing of valid `_clkmode` constant declarations can be found in the Propeller Manual's Spin Language Reference `_CLKMODE` section.

Crystal Precision

The Propeller chip's internal RC clock serves for non-timing-sensitive applications, such as controlling outputs based on inputs and blinking lights. For applications that are timing-sensitive like serial communication, tone generation, servo control, and timekeeping, the Propeller chip can be connected to crystal oscillators and other higher-precision external clock signals via its XI and XO pins.

The Propeller chip's internal oscillator in its default RCFAST mode is what the Propeller chip uses if the program does not specify the clock source or mode. This oscillator's nominal frequency is 12 MHz, but its actual frequency could fall anywhere in the 8 to 20 MHz range. That's an error of +66 to – 33%. Again, for applications that do not require precise timing, it suffices. On the other hand, an application like asynchronous serial communication can only tolerate a total of 5 % error, and that's the sum of both the transmitter's and receiver's timing errors. In practical designs, it would be best to shoot for an error of less than 1%. By using an external crystal for the Propeller chip's clock source, the clock frequency can be brought well within this tolerance, or even within timekeeping device tolerances.

The PE Platform has an ESC Inc. HC-49US quartz crystal connected to the Propeller chip's XI and XO pins that can be used in most timing-sensitive applications. The datasheet for this part rates its room temperature frequency tolerance at +/- 30 PPM, meaning +/- 30 clock ticks for every million. That's a percent error of only +/- 0.003%. Obviously, this is more than enough precision for asynchronous serial communication, and it's also great for servo control and tone generation. It's not necessarily ideal for watches or clocks though; this crystal's error could cause an alarm clock or watch to gain or lose up to 2.808 s per day. This might suffice for datalogging or clocks that periodically check in with an atomic clock for updates. Keep in mind that to make the Propeller chip function with digital wristwatch precision, all it takes is a more precise oscillator.

The HC-49US datasheet also has provisions for temperature (+/- 50 PPM) and aging (+/- 5 PPM per year). Even after 5 years, and at its rated -10 to + 70 ° C, the maximum error would be 105 PPM, which is still only +/- 0.0105% error. That's still great for asynchronous serial communication, tone generation, and servo control, but again, an alarm clock might gain or lose up to 9 s per day.

Since `clkfreq` stores the system clock frequency, object code can rely on it for correct timing, regardless of the system clock settings. The `clkfreq` command returns the number of ticks per second based on the Propeller chip's system clock settings. For example, this `CON` block uses `_xinfreq = 5_000_000` and `_clkmode = xtal1 + pll16x`., so `clkfreq` will return the value of 5_000_000 × 16, which equals 80_000_000.

ConstantBlinkRate.spin can be configured to a variety of system clock rates to demonstrate how `clkfreq` keeps the timing constant regardless of the clock frequency.

- ✓ Load ConstantBlinkRate.spin into the Propeller chip's RAM (F10). The system clock will be running at 80 MHz.
- ✓ Verify that the blink rate is 1 Hz.
- ✓ Modify the `_clkmode` constant declaration to read `_clkmode = xtal1 + pll8x` to make the system clock run at 40 MHz, and load the program into RAM (F10).

```
'' File: ConstantBlinkRate.spin

CON

  _xinfreq = 5_000_000
  _clkmode = xtal1 + pll16x

PUB LedOnOff

    dira[4] := 1

    repeat

        outa[4] := 1
        waitcnt(clkfreq/2 + cnt)
        outa[4] := 0
        waitcnt(clkfreq/2 + cnt)
```

The Propeller chip's system clock is now running at 40 MHz. Is the LED still blinking on/off at 1 Hz?

- ✓ Repeat for `pll4x`, `pll2x`, and `pll1x`. There should be no change in the blink rate at any of these system clock frequencies.

Timing with clkfreq vs. Timing with Constants

Let's say that a constant value is used in place of `clkfreq` to make the program work a certain way at one particular system clock frequency. What happens when the Propeller system clock frequency changes?

- ✓ Save a copy of the ConstantBlinkRate object as BlinkRatesWithConstants.spin.
- ✓ Make sure the PLL multiplier is set to `pll1x` so that the system clock runs at 5 MHz.
- ✓ For a 1 Hz on/off signal, replace both instances of `clkfrq/2` with `2_500_000`.
- ✓ Load the object into the Propeller chip's RAM and verify that the LED blinks at 1 Hz.
- ✓ Next, change the PLL multiplier to `pll2x`. Load the modified object into the Propeller chip's RAM. Does the light blink twice as fast? Try `pll4x`, `pll8x`, and `pll16x`.

When a constant value was used instead of `clkfreq`, a change in the system clock caused a change in event timing. This is why objects should use `clkfreq` when predictable delays are needed, especially for objects that are designed to be used by other objects. That way, the programmer can choose the best clock frequency for the application without having to worry about whether or not any of application's objects will behave differently.

More Output Register Operations

In the I/O Pin Group Operations section, binary values were assigned to groups of bits in the `dira` and `outa` registers. There are lots of shortcuts and tricks for manipulating groups of I/O pin values that you will see used in published code examples.

The Post-Set "~~" and Post-Clear "~" Operators

Below are two example objects that do the same thing. While the object on the left uses techniques covered earlier to set and clear all the bits in `dira[4..9]` and `outa[4..9]`, the one on the right does it differently, with the Post-Set "~~"and Post-Clear "~"operators. These operators come in handy when all the bits in a certain range have to be set or cleared.

```
''File: LedsOnOff.spin
''All LEDS on for 1/4 s and off
''for 3/4 s.

PUB BlinkLeds

    dira[4..9] := %111111

    repeat

        outa[4..9] := %111111
        waitcnt(clkfreq/4 + cnt)
        outa[4..9] := %000000
        waitcnt(clkfreq/4*3 + cnt)
```

```
''File: LedsOnOffAgain.spin
''All LEDS on for 1/4 s and off
''for 3/4 s with post set/clear.

PUB BlinkLeds

    dira[4..9]~~

    repeat

        outa[4..9]~~
        waitcnt(clkfreq/4 + cnt)
        outa[4..9]~
        waitcnt(clkfreq/4*3 + cnt)
```

- ✓ Load each program into the Propeller chip's RAM and verify that they function identically.
- ✓ Examine how the Post-Set operator replaces `:= %111111` and the Post-Clear operator replaces `:= %000000`.

✓ Try modifying both programs so that they only affect P4..P7. Notice that the Post-Set and Post-Clear operators require less maintenance since they automatically set or clear all the bits in the specified range.

The Bitwise Not "!" Operator

Here are two more example programs that do the same thing. This time, they both light alternate patterns of LEDs. The one on the left has familiar assignment operators in the `repeat` loop. The one on the right initializes the value of `outa[4..9]` before the repeat loop. Then in the `repeat` loop, it uses the Bitwise NOT "!" operator on `outa[4..9]`. If `outa[4..9]` stores `%100001`, the command `!outa[4..9]` inverts all the bits (1s become 0s, 0s become 1s). So, the result of `!outa[4..9]` will be `%011110`.

✓ Load each object into the Propeller chip's RAM and verify that they function identically.
✓ Try doubling the frequency of each object.

```
''File: LedsOnOff50Percent.spin
''Leds alternate on/off 50% of
''the time.

PUB BlinkLeds

    dira[4..9]~~

    repeat

        outa[4..9] := %100001
        waitcnt(clkfreq/4 + cnt)
        outa[4..9] := %011110
        waitcnt(clkfreq/4 + cnt)
```

```
''File: LedsOnOff50PercentAgain.spin
''Leds alternate on/off 50% of
''the time with the ! operator.

PUB BlinkLeds

    dira[4..9]~~
    outa[4..9] := %100001

    repeat

        !outa[4..9]
        waitcnt(clkfreq/4 + cnt)
```

Register Bit Patterns as Binary Values

A range of bits in a register can be regarded as digits in a binary number. For example, in the instruction `outa[9..4] := %000000`, recall that `%` is the binary number indicator; `%000000` is a 6-bit binary number with the value of zero. Operations can be performed on this value, and the result placed back in the register. The IncrementOuta object below adds 1 to `outa[9..4]` each time through a `repeat` loop. The result will be the following sequence of binary values, displayed on the LEDs:

Binary Value	Decimal Equivalent
%000000	0
%000001	1
%000010	2
%000011	3
%000100	4
%000101	5
etc…	
%111101	61
%111110	62
%111111	63

✓ Load IncrementOuta.spin it into RAM.

```
'' File: IncrementOuta.spin

PUB BlinkLeds

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

    repeat

        waitcnt(clkfreq/2 + cnt)
        outa[9..4] := outa[9..4] + 1
```

Incrementation starts by setting LED I/O pins to output with `dira[9..4]~~`. Next, `outa[9..4]~` clears all the bits in the `outa` register range 9..4 to %000000, binary zero. The first time through the `repeat` loop, 1 is added to it, the equivalent of `outa[9..4] := %000001`, which causes the P4 LED to light up. As the loop repeats indefinitely, the LED pattern cycles through every possible permutation.

The Increment "++" operator

The Increment "++"operator can be used instead of `+ 1` to increment a value. The command `outa[9..4]++` is equivalent to `outa[9..4] := outa[9..4] + 1`.

- ✓ Modify the `outa` command in the `repeat` loop to use only `outa[9..4]++` .
- ✓ Load the modified object into RAM. Do the LEDs behave the same way?

Conditional Repeat Commands

Syntax options for `repeat` make it possible to specify the number of times a block of commands is repeated. They can also be repeated `until` or `while` one or more conditions exist, or even to sweep a variable value `from` a start value to a `finish` value with an optional `step` delta.

- ✓ Read the syntax explanation in the `REPEAT` section of the Propeller Manual's Spin Language Reference, if you have it handy.

Let's modify IncrementOuta further to stop after the last value (%111111 = 63) has been displayed. To limit the loop to 63 cycles just add an optional Count expression to the `repeat` command, like this:

```
repeat 63
```

- ✓ Save IncrementOuta.spin as BinaryCount.spin.
- ✓ Add the Count value `63` after the `repeat` command.
- ✓ To keep the LEDs lit after the repeat block terminates, add a second `repeat` command below the block. Make sure it is not indented further than the first `repeat`.
        Load the BinaryCount object into the Propeller chip's RAM and verify that the LEDs light up according to the Binary Value sequence.

There are a lot of different ways to modify the `repeat` loop to count to a certain value and then stop. Here are a few `repeat` loop variations that count to decimal 20 (binary %010100); the second example uses the Is Equal "==" operator, the third uses Is Less Than "<"

```
repeat 20                     ' Repeat loop 20 times
repeat until outa[9..4] == 20  ' Repeat until outa[9..4] is equal to 20
repeat while outa[9..4] < 20   ' Repeat while outa[9..4] is less than 20
```

Operations in Conditions and Pre and Post Operator Positions

(11 more ways to count to 20)

The `outa[9..4]++` command can be removed from the code block in the `repeat` loop and incremented right inside the `repeat` command conditions. The IncrementUntilCondition object shows an example that counts to 20 with `outa[9..4]` incremented by the `++` operator right in the `repeat` loop's condition.

```
'' File: IncrementUntilCondition.spin

PUB BlinkLeds

    dira[4..9]~~

    repeat until outa[9..4]++ == 19

        waitcnt(clkfreq/2 + cnt)

    repeat
```

> ℹ️ **outa and dira initialize to zero** when the program starts, so there is no need to include `outa[9..4]~`.

- ✓ Load IncrementUntilCondition.spin into the Propeller and verify that it counts to 20.

Note that the loop repeats until 19, but the program actually counts up to 20. Another way to use `++` in the `repeat` loop's condition is to place it before `outa[9..4]`, like this:

```
repeat until ++outa[9..4] == 20
```

- ✓ Modify the IncrementUntilCondition object's `repeat` command, with its condition being `until ++outa[9..4] == 20`. Verify that it still stops counting at 20.

What's the difference?   If the `++` is placed to the left of `outa[9..4]`, it is typically called Pre-Increment and the operation is performed before the `++outa[9..4] ==`… condition is evaluated. (The operator `--` placed to the left called Pre-Decrement.) Likewise, if `++` or `--` is placed to the right of `outa[9..4]`, it is typically called Post-Increment or Post-Decrement, and the operation is performed after the condition is evaluated.

With `repeat until outa[9..4]++ == 19`, the loop delays when `outa[9..4]` stores 0, 1, 2…19. When `outa[9..4]` stores 19, the loop does not `repeat` the `waitcnt`. However, since the post-incrementing occurs after the condition is evaluated, another 1 gets added to `outa[9..4]` even though the loop doesn't get repeated again.

With `repeat until ++outa[9..4] == 20`, `outa[9..4]` is pre-incremented, so the first delay doesn't occur until after `outa[9..4]` gets bumped up to 1. The next delay occurs after 2, 3, and so on up through 19. The next repetition, `outa[9..4]` becomes 20, so `waitcnt` command inside the loop does not execute, but again, the last value that `outa[9..4]` holds is 20.

Instead of repeating `until` a condition is true, a loop can be repeated `while` a condition is true. Here are examples that count to 20 using the `while` condition, with Post- and Pre-Increment operators adding 1 to `outa[9..4]`:

```
repeat while outa[9..4]++ < 19      ' Repeat while outa[9..4] post-incremented is less
                                    ' than 19.
repeat while ++outa[9..4] < 20      ' Repeat while outa[9..4] pre-incremented is less
                                    ' than 20.
```

Notice that the post-incremented loop counts to 20, repeating while `outa`[9..4] is less than 19, but the pre-incremented version repeats while `outa`[9..4] is less than 20. Notice that with `repeat while`…, the Is Less Than "<" operator is used instead of the Is Equal "=="operator. These two approaches demonstrate the distinction between repeating *until* something is equal to a value as opposed to repeating *while* something is less than a value.

Of course, you could also use the Is Equal or Less "=<" operator, or even the Is Not Equal "<>" operator. Here are examples of those; in each case the LED display will stop at binary 20.

```
repeat while outa[9..4]++ =< 18     ' Repeat while outa[9..4] post-incremented is less
                                    ' than or equal to 18.
repeat while ++outa[9..4] =< 19     ' Repeat while outa[9..4] pre-incremented is less
                                    ' than 19.
repeat while ++outa[9..4] <> 20     ' Repeat while outa[9..4] pre-incremented is not
                                    ' equal to 20.
```

Is Greater ">" or even Is Equal or Greater "=>" also be used with `repeat until`…

```
repeat until outa[9..4]++ > 18      ' Repeat until outa[9..4] post-incremented is
                                    ' greater than 18.
repeat until ++outa[9..4] > 19      ' Repeat until outa[9..4] pre-incremented is
                                    ' greater than 19.
repeat until ++outa[9..4] => 20     ' Repeat until outa[9..4] pre-incremented is equal
                                    ' or greater than 20.
repeat until outa[9..4]++ => 19     ' Repeat until outa[9..4] post-incremented is equal
                                    ' or greater than 19.
```

✓ Examine each of the `repeat` commands and try each one in the IncrementUntilCondition object.

If there are any question marks in your brain about this, don't worry right now. The point of this section is to demonstrate that there is a variety of ways to make comparisons and to increment values. Upcoming labs will include better ways to display each loop repetition so that you can test each approach.

One More Repeat Variations with From…To…

(Or, Another 3 Ways to Count to 20)

Here is one more condition for `repeat`, repeating `outa`[9..4] *from* one value *to* another value. With each repetition of the loop, this form of `repeat` automatically adds 1 to the count each time through. Take a look at the code snippet below. The first time through the loop, `outa`[9..4] starts at 0. The second time through, 1 is automatically added, and the condition is checked to make sure `outa`[9..4] is greater than or equal to 0 or less than or equal to 19. 1 is added each time through the loop. After the repetition where `outa`[9..4] is equal to 19, it adds 1 to `outa`[9..4], making 20. Since 20 is not in the "from 0 to 19" range, the code in the loop does not execute.

```
repeat outa[9..4] from 0 to 19      ' Add 1 to outa[9..4] with each repetition
                                    ' start at 0 and count through 19.  Repeat's Code
                                    ' block when outa[9..4] gets to 20.
```

Here is a `repeat` command that serves a similar function using `and`. It tests for two conditions, both of which must be true in order for the loop to repeat. Here we need to increment `outa[9..4]` within the loop block:

```
repeat while (outa[9..4] => 0) and (outa[9..4] =< 19)
    outa[9..4]++
```

Another nice thing about the `repeat…from…to…` form is you can use an optional `step` argument. For example, if you want to repeat what's in a loop with `outa[9..4]` at all even values, and exit the loop leaving `outa[9..4]` at 20, here's a way to do it:

```
Repeat outa[9..4] from 0 to 18 step 2
```

✓ Try the various `repeat` command variations in this section in the IncrementUntilCondition object.

Some Operator Vocabulary

Unary operators have one *operand*. For example, the Negate operator "`-`" in the expression `-1` is a unary operator, and `1` is the operand. *Binary* operators have two operands; for example, the Subtract operator "`-`" in the expression `x - y` is a binary operator, and both `x` and `y` are operands.

Normal operators, such as Add "`+`", operate on their operands and provide a result for use by the rest of the expression without affecting the operand(s). Some operators we have used such as `:=`, `~~`, `~`, and `!` are *assignment operators*. Unary assignment operators, such as `~` , `~~`, and `++` write the result of the operation back to the operand whereas binary assignment operators, such as `:=`, assign the result to the operand to the immediate left. In both cases the result is available for use by the rest of the expression.

The *shift* operators Shift Left "`>>`"and Shift Right "`<<`" take the binary bit pattern of the value in the first operand and shift it to the right or the left by the number of bits specified by a second operand, and returns the value created by the new bit pattern. If an assignment form is used (`>>=` or `<<=`) the original value is overwritten with the result. The shift operators are part of a larger group, *Bitwise operators*, which perform various bit manipulations. The Bitwise NOT "`!`"operator we used earlier is an example.

Some normal and assignment operators have the additional characteristic of being a *comparison operator*. A comparison operator returns true (`-1`) if the values on both sides of the operator make the expression true, or false (`0`) if the values on both sides make the expression false. (These binary comparison operators are also called *Boolean* operators; there is also a unary Boolean operator, `NOT`.)

Conditional Blocks with if

As with many programming languages, Spin has an `if` command that allows a block of code to be executed conditionally, based on the outcome of a test. An `if` command can be used on its own, or as part of a more complex series of decisions when combined with `elseif`, `elseifnot` and `else`. Comparison operators are useful to test conditions in `if` statements:

```
if outa[9..4] == 0
    outa[9..4] := %100000

waitcnt(clkfreq/10 + cnt)
```

If the condition is true, the block of code (one line in this case) below it will be executed. Otherwise, the program will skip to the next command that's at the same level of indentation as the `if` statement (here it is `waitcnt`).

Shifting LED Display

The next example object, ShiftRightP9toP4, makes use of several types of operators to efficiently produce a shifting light pattern with our 6 LED circuits.

- ✓ Load ShiftRightP9toP4 into the Propeller chip's RAM.
- ✓ Orient your PE platform so that the light appears to be shifting from left to right over and over again.
- ✓ Verify that the pattern starts at P9 and ends at P4 before repeating.

```
'' File: ShiftRightP9toP4.spin
'' Demonstrates the right shift operator and if statement.

PUB ShiftLedsLeft

    dira[4..9] ~~

    repeat

        if outa[9..4] == 0
            outa[9..4] := %100000

        waitcnt(clkfreq/10 + cnt)
        outa[9..4] >>= 1
```

Each time through the `repeat` loop, the command `if [9..4] == 0` uses the `==` operator to compare `outa[9..4]` against the value 0. If the expression is true, the result of the comparison is -1. If it's false, the result is 0. Remember that by default `outa[9..4]` is initialized to zero, so the first time through the `repeat` loop `outa[9..4] == 0` evaluates to true. This makes the `if` statement execute the command `outa[9..4] := %100000`, which turns on the P9 LED.

After a 1/10 s delay, `>>=` (the shift right assignment operator) takes the bit pattern in `outa[9..4]` and shifts it right one bit with this instruction: `outa[9..4] >>= 1`. The rightmost bit that was in `outa[4]` is discarded, and the vacancy created in `outa[9]` gets filled with a 0. For example, if `outa[9..4]` stores %011000 before `outa[9..4] >>= 1`, it will store %001100 afterwards. If the command was `outa[4..9] >>= 3`, the resulting pattern would instead be %000011

Each time through the loop, the `outa[9..4] >>= 1` command shifts the pattern to the right, cycling through %100000, %010000, %001000,…, %000001, %000000. When `outa[9..4]` gets to %000000, the `if` command sees that `outa[9..4]`, stores a 0, so stores %100000 in `outa[9..4]`, and the shifting LED light repeats.

- ✓ Try changing the second operand in the shift right operation from 1 to 2, to make the pattern in `outa[4...9]` shift two bits at a time. You should now see every other LED blink from left to right.

Variable Example

The ButtonShiftSpeed object below is an expanded version of ShiftRightP9toP4 that allows you to use pushbuttons to control the speed at which the lit LED shifts right. If you hold the P21 pushbutton down the shift rate slows down; hold the P22 pushbutton down and the shift rate speeds up. The speed control is made possible by storing a value in a variable. The pattern that gets shifted from left to right is also stored in a variable, making a number of patterns possible that cannot be achieved by performing shift operations on the bits in `outa[9..4]`.

- ✓ Load ButtonShiftSpeed.spin into RAM.
- ✓ Try holding down the P22 pushbutton and observe the change in the LED behavior, then try holding down the P21 pushbutton.

```
'' File: ButtonShiftSpeed.spin
'' LED pattern is shifted left to right at variable speeds controlled by pushbuttons.

VAR

    Byte pattern, divide

PUB ShiftLedsLeft

    dira[4..9] ~~
    divide := 5

    repeat

        if pattern == 0
            pattern := %11000000

        if ina[22] == 1
            divide ++
            divide <#= 254
        elseif ina[21] == 1
            divide --
            divide #>= 1

        waitcnt(clkfreq/divide + cnt)
        outa[9..4] := pattern
        pattern >>= 1
```

ButtonShiftSpeed has a variable (`VAR`) block that declares two byte-size variables, `pattern` and `divide`. The `pattern` variable stores the bit pattern that gets manipulated and copied to `outa[9..4]`, and `divide` stores a value that gets divided into `clkfreq` for a variable-length delay.

`Byte` is one of three options for variable declarations, and it can store a value from `0` to `255`. Other options are `word` (`0` to `65536`) and `long` (`-2,147,483,648` to `2,147,483,647`). Variable arrays can be declared by specifying the number of array elements in brackets to the right of the variable name. For example, `byte myBytes[20]` would result in a 20-element array named `myBytes`. This would make available the variables `myBytes[0]`, `myBytes[1]`, `myBytes[2]`,…, `myBytes[18]`, and `myBytes[19]`.

The first `if` block in the `repeat` loop behaves similarly to the one in the ShiftRightP9toP4 object. Instead of `outa[9..4]`, the `if` statement examines the contents of the `pattern` variable, and if it's zero, the next line reassigns `pattern` the value `%11000000`.

The Limit Minimum "#>"and Limit Maximum "<#" Operators

Spin has Limit Minimum "`#>`" and Limit Maximum "`<#`"operators that can be used to keep the value of variables within a desired range as they are redefined by other expressions. In our example object,

the second `if` statement in the `repeat` loop is part of an `if...elseif...` statement that checks the pushbutton states. If the P22 pushbutton is pressed, `divide` gets incremented by 1 with `divide ++`, and then `divide` is limited to 254 with `<#=`, the assignment form of the Limit Maximum operator. So, if `divide ++` resulted in 255, the next line, `divide <#= 254` reduces its value back to 254. This prevents the value of `divide` from rolling over to 0, which is important because `divide` gets divided into `clkfreq` in a `waitcnt` command later in the `repeat` loop. If the P21 pushbutton is pressed instead of P22, the `divide` variable is decremented with `divide --`, which subtracts 1 from `divide`. The `#>=` assignment operator is used to make sure that `divide` never gets smaller than 1, again preventing it from getting to 0.

After the `if...elseif...` statement checks the pushbutton states and either increments or decrements the `divide` variable if one of the pushbuttons is pressed, it uses `waitcnt(clkfreq/divide + cnt)` to wait for a certain amount of time. Notice that as `divide` gets larger, the time `waitcnt` waits gets smaller. After the delay that's controlled by the `divide` variable, `pattern` gets stored in `outa` with `outa[9..4] := pattern`. Last of all, the `pattern` variable gets shifted right by 1 for the next time through the loop.

Comparison Operations vs. Conditions

Comparison operators return true (-1) or false (0); when used in `if` and `repeat` blocks the specified code executed if the condition is non-zero. This being the case, if `ina[22]` can be used instead of `if ina[22] == 1`. The code works the same, but with less processing since the comparison operation gets skipped.

When the button is pressed, the condition in `if ina[22] == 1` returns -1 since `ina[22]` stores a 1 making the comparison true. Using just if `ina[22]` will still cause the code block to execute when the button is pressed since `ina[22]` stores 1, which is still non-zero, causing the code block to execute. When the button is not pressed, `ina[22]` stores 0, and `ina[22] == 1` returns false (0). In either case, the `if` statement's condition is 0, so the code below either if `ina[22] == 0` or `if ina[22]` gets skipped.

✓ change `if ina[22] == 1 ...elseif ina[21] == 1` to `if ina[22] ...elseif ina[21]...`, and verify that the modified program still works.

Local Variables

While all the example objects in this lab have only used one method, objects frequently have more than one method, and applications typically are a collection of several objects. Methods in applications pass program control, and optionally parameters, back and forth between other methods in the same object as well as methods in other objects. These features will all be studied in upcoming labs. In preparation for working with multiple methods, let's look at how a method can create a local variable.

Variables declared in an object's `VAR` section are global to the object, meaning all methods in a given object can use them. Each method in an object can also declare local variables for its own use. These local variables only last as long as the method is being executed. If the method runs out of commands and passes program control back to whatever command called it, the local variable name and memory locations get thrown back in the heap for other local variables to use.

The two global variables in the ButtonShiftSpeed object can be replaced with local variables as follows:

✓ Remove the `VAR` block (including its byte variable declarations).

✓ Add the pipe | symbol to the right of the method block declaration followed by the two variable names separated by commas, then test the program verify it still functions properly.

```
PUB ShiftLedsLeft | pattern, divide
```

Aside from the fact that the `pattern` and `divide` variables are now local, meaning other methods in the object could not use them; since our object has just one method this is of no consequence here. There is one other difference. When we used the `VAR` block syntax, we had the option of defining our global variables as byte, word, or long in size. However, local variables are automatically defined as longs and there is no option for byte or word size local variables.

Timekeeping Applications

For clock and timekeeping applications, it's important to eliminate all possible errors, except for the accuracy of the crystal oscillator. Take a look at the two objects that perform timekeeping. Assuming you have a very accurate crystal, the program on the left has a serious problem! The problem is that each time the loop is repeated, the clock ticks elapsed during the execution of the commands in the loop are not accounted for, and this unknown delay accumulates along with `clkfreq + cnt`. So, the number of seconds the `seconds` variable will be off by will grow each day and will be significantly more than just the error introduced by the crystal's rated +/- PPM.

The program on the right solves this problem with two additional variables: `T` and `dT`. A time increment is set with `dT := clkfreq` which makes `dT` equal to one second with the precision of the crystal. A particular starting time is marked with `T := cnt`. Then, inside the loop, it recalculates the next `cnt` value that `waitcnt` will have to wait for with `T += dT`. This use of the add assignment operator `+=` allows us to create a precise offset from original marked value of `T`. With this system, each new target value for `waitcnt` is exactly 1 second's worth of clock ticks from the previous. It no longer matters how many tasks get performed between `waitcnt` command executions, the program on the right will never lose any clock ticks and maintain a constant 1 s time base that's as good as the signal that the Propeller chip is getting from the external crystal oscillator.

```
''File: TimekeepingBad.spin

CON

    _xinfreq = 5_000_000
    _clkmode = xtal1 + pll1x

VAR

    long seconds

PUB BadTimeCount

    dira[4]~~

    repeat
        waitcnt(clkfreq + cnt)
        seconds ++
        ! outa[4]
```

```
''File: TimekeepingGood.spin

CON

    _xinfreq = 5_000_000
    _clkmode = xtal1 + pll1x

VAR

    long seconds, dT, T

PUB GoodTimeCount

    dira[9..4]~~

    dT := clkfreq
    T  := cnt

    repeat
        T += dT
        waitcnt(T)
        seconds ++
        outa[9..4] := seconds
```

✓ Try running both objects. Without an oscilloscope, there should be no noticeable difference.

✓ Add a delay of 0.7 s to the end of each object (inside each repeat loop). The object on the left will now repeat every 1.7 s; the one on the right should still repeat every 1 s.

Instead of a delay, imagine how many other tasks the Propeller chip could accomplish in each second and still maintain an accurate time base!

Various multiples of a given time base can have different meanings and uses in different applications. For example, these objects have seconds as a time base, but we may be interested in minutes and hours. There are 60 seconds in a minute, 3,600 seconds in an hour and 86,400 seconds in a day. Let's say the application keeps a running count of seconds. A convenient way of determining whether another minute has elapsed is by testing to divide seconds by 60 and see if there is a remainder. The Modulus "//" operator returns the reminder of division problems. As the seconds pass, seconds // 60 is 0 when seconds is 0, 60, 120, 180, and so on. The rest of the time, the Modulus returns whatever is left over. For example, when seconds is 121, the result of seconds // 60 is 1. When seconds is 125, the result of seconds // 60 is 5, and so on.

This being the case, here's an expression that increments a minutes variable every time another 60 seconds goes by.

```
if seconds // 60 == 0
    minutes ++
```

Here's another example with hours:

```
if seconds // 3600 == 0
    hours ++
```

For every hour that passes, when minutes gets to 60, it should be reset to zero. Here is an example of a nested **if** statement that expands on the previous minutes calculation:

```
if seconds // 60 == 0
    minutes ++
    if minutes == 60
        minutes := 0
```

The TimeCounter object below uses synchronized timekeeping and a running total of seconds with the modulus operator to keep track of seconds, minutes, hours, and days based on the seconds count. The value of seconds is displayed in binary with the 6 LED circuits. Study this program carefully, because it contains keys to this lab's projects that increment a time setting based in different durations of holding down a button. It also has keys to another project in which LEDs are blinked at different rates without using multiple cogs. (When you use multiple cogs in later labs, it will be a lot easier!)

✓ Load TimeCounter.spin into EEPROM, and verify that it increments the LED count every 1 s.
✓ Modify the code so that the last command copies the value held by minutes into **outa**[9..4], and verify that the LED display increments every minute.

```
''File: TimeCounter.spin

CON

    _xinfreq = 5_000_000
    _clkmode = xtal1 + pll1x

VAR

    long seconds, minutes, hours, days, dT, T
```

```
PUB GoodTimeCount

    dira[9..4]~~

    dT := clkfreq
    T  := cnt

    repeat

       T += dT
       waitcnt(T)
       seconds++

       if seconds // 60 == 0
          minutes++
          if minutes == 60
             minutes := 0
       if seconds // 3600 == 0
          hours++
          if hours == 24
             hours := 0
       if seconds // 86400 == 0
          days++

       outa[9..4] := seconds
```

Eventually, the `seconds` variable will reach variable storage limitations. For example, when it gets to 2,147,483,647, the next value will be -2,147843,648, and after that, -2,147,843,647, -2,147,843,646, and so on down to -2, -1. So, how long will it take for the seconds timer to get to 2,147,483,647? The answer is 68 years. If this is still a concern for your application, consider resetting the second counter every year.

Questions
1) How many processors does the PE Kit's Propeller microcontroller have?
2) How much global RAM does the Propeller microcontroller have?
3) What's the Propeller chip's supply voltage? How does this relate to an I/O pin's high and low states?
4) Where does the Propeller chip store Spin code, and how is it executed?
5) How does executing Spin codes differ from executing assembly language codes?
6) What's the difference between a method and an object?
7) What's a top level object?
8) What do bits in the `dira` and `outa` registers determine?
9) Without optional arguments the `repeat` command repeats a block of code indefinitely. What types of optional arguments were used in this lab, and how did they limit the number of loop repetitions?
10) What Spin command used with `waitcnt` makes it possible to control timing without knowing the Propeller chip's system clock frequency in advance?
11) If commands are below a `repeat` command, how do you determine whether or not they will be repeated in the loop?
12) What was the most frequent means of calculating a target value for the `waitcnt` command, and what register does the `waitcnt` command compare this target value to?
13) What's the difference between `_xinfreq` and `_clkmode`.
14) What does the phase-locked loop circuit do to the crystal clock signal?
15) Why is it so important to use a fraction of `clkfreq` instead of a constant value for delays?
16) Which clock signal will be more accurate, the Propeller's internal RC clock or an external crystal?

17) What registers control I/O pin direction and output? If an I/O pin is set to input, what register's values will change as the application is running, and how are the values it returns determined by the Propeller?

18) What's the difference between `dira`/`outa`/`ina` syntax that refers to single bit in the register and syntax that denotes a group of bits?

19) What indicator provides a convenient means of assigning a group of bit values to a contiguous group of bits in a `dira`/`outa`/`ina` register?

20) How does an I/O pin respond if there is a 0 in its `dira` register bit and a 1 in its `outa` register bit?

21) If bits in either `dira` or `outa` are not initialized, what is their default value at startup?

22) What assignment operators were introduced in this lab?

23) What comparison operators were used in this lab?

24) What's the difference between the `:=` and `==` operators?

25) Are comparison operators necessary for if conditions?

26) What are the two different scopes a variable can have in an object?

27) What are the three different variable sizes that can be declared? What number range can each hold? Does the scope of a variable affect its size?

28) How does a method declare local variables? What character is required for declaring more than one local variable?

Exercises

1) Write a single line of code that sets P8 through P12 output-high.

2) Write commands to set P9 and P13 through 15 to outputs. P9 should be made output-high, and P13 through 15 should be low.

3) Write a single initialization command to set P0 through P2 to output and P3 through P8 to input.

4) Write a `repeat` block that toggles the states of P8 and P9 ever 1/100 s. Whenever P8 is on, P9 should be off, and visa versa.

5) Write a `repeat` loop that sets P0 through P7 to the opposite of the states sensed by P8 through P15. You may want to consult the Propeller Manual's list of assignment operators for the best option.

6) Write a `CON` block to make the Propeller chip's system clock run at 10 MHz.

7) Write code for a five second delay.

8) Write code that sets P5 through P11 high for 3 seconds, then sets P6, P8, and P10 low. Assume the correct `dira` bits have already been set.

9) Write a method named `LightsOn` with a `repeat` loop that turns on P4 the first second, P5 the second, P6 the third, and so on through P9. Assume that the I/O pin direction bits have not been set. Make sure the lights stay on after they have all bee turned on.

10) Write a method that turns an LED connected to P27 on for 5 s if a pushbutton connected to P0 has been pressed, even if the button is released before 5 s. Don't assume I/O directions have been set. Make sure to turn the P27 LED off after 5 s.

11) Write a second countdown method that displays on the P4 through P9 LEDs. It should count down from 59 to 0 in binary.

12) Write a second countdown method that displays on the P4 through P9 LEDs. It should count down from 59 to 0 in binary, over and over again, indefinitely.

13) Write a method named `PushTwoStart` that makes you press the buttons connected to P21 and P23 at the same time to start the application. For now, the application can do as little as turn an LED on and leave it on.

14) Write a method named `PushTwoCountdown` that makes you press the buttons connected to P21 and P23 at the same time to start the application. The application should count down from 59 to 0 using P9 through P4.

Projects

1) Connect red LEDs to P4 and P7, yellow LEDs to P5 and P8, and green LEDs to P6 and P9. Assume that one set of LEDs is pointing both directions on the north south street, and the other set is pointing both ways on the east west street. Write an non-actuated street controller object (one that follows a pattern without checking to find out which cars are at which intersections).

2) Repeat the previous project, but assume that the N/S street is busy, and defaults to green while the E/W street has sensors that turn the light.

3) Use a single cog to make LEDs blink at different rates (this is much easier with multiple cogs, as you will see in later labs). Make P4 blink at 1 Hz, P5 at 2 Hz, P6 at 3 Hz, P7 at 7 Hz, P8 at 12 Hz and P9 at 13 Hz.

4) Buttons for setting alarm clock times typically increment or decrement the time slowly until you have held the button down for a couple of seconds. Then, the time increments/decrements much more rapidly. Alarm clock buttons also let you increment/decrement the time by rapidly pressing and releasing the pushbutton. Write an application that lets you increase or decrease the binary count for minutes (from 0 to 59) with the P21 and P23 pushbuttons. As you hold the button, the first ten minutes increase/decrease every ½ s, then if you continue to hold down the button, the minutes increase/decrease 6 times as fast. Use the P9 through P4 LEDs to display the minutes in binary.

5) Extend project 1 by modifying the object from project 1 so that it is a countdown timer that gets set with the P21 and P23 buttons and started by the P22 button.

Question Solutions

1) Eight
2) 32 KB
3) The Propeller's supply voltage is 3.3 V. When an I/O pin is high, the Propeller chip internally connects the I/O pin to its 3.3 V supply, and when it's low, it's connected to GND or 0 V.
4) Spin code is stored in the Propeller chip's global RAM, and a cog running an interpreter program fetches and executes the codes.
5) Instead of executing Spin codes that get fetched from global RAM and executed, machine codes generated by assembly language get stored in a cog's 2 KB of RAM, and are executed directly by the cog.
6) There are a lot of ways to answer this. The most condensed and Propeller-centric answer would be that a method is a block of code with a minimum of a declared access rule and name; whereas, and object is a building block comprised of all the code in a .spin file. Every object also contains one or more methods.
7) It's the object that provides a starting point for a given application that gets loaded into the Propeller chip's RAM. Although it's not required, top level objects often organize and orchestrate the application's objects.
8) Each bit in `dira` sets the direction (output or input) of an I/O pin for a given cog. Each bit in `outa` sets the output state (on or off) for a given cog, provided the corresponding bit in the `dira` register is set to output.
9) There were four different types of conditions. The number of repetitions was placed to the right of the `repeat` command to specify how many times the loop gets repeated. The `while` condition specified to keep repeating a loop while a condition is true. The `until` condition was used to keep repeating code until a certain condition occurs. Finally, a variable was incremented each time through a `repeat` loop, from a certain value, to a certain value.
10) `clkfreq`
11) They need to be below and indented from the `repeat` command to be part of the loop. The next command following the `repeat` command that is at the same or less level of indentation

---

is not part of the `repeat` loop, nor is any command that follows it, regardless of its indentation.

12) The `waitcnt` command's target value was typically calculated by adding some fraction of `clkfreq` to the `cnt` register. Then, the `waitcnt` waits until the `cnt` register exceeds the `waitcnt` value.

13) `_xinfreq` stores the input oscillator's frequency; whereas, in this lab `_clkmode` was used to define the Propeller chip's crystal feedback and PLL multiplier settings. For more information, look these terms up in the Propeller Manual.

14) It multiplies the frequency by a value. Multiplier options are 1, 2, 4, 8, or 16.

15) The `clkfreq` constant adjusts with the Propeller chip's system clock; whereas, a constant value used for delays will result in delays that change with the system clock settings.

16) An external crystal.

17) The `dira` and `outa` registers control direction and output state respectively. If an I/O pin is set to input, the `ina` register's values will update at runtime when an ina command is issued, returning 1 or 0 for each bit depending voltage applied to the corresponding I/O pin. Voltages applied to an I/O pin above 1.65 V cause a 1 to be returned. Voltages below 1.65 V cause a 0 to be returned.

18) A single value in between the square brackets to the right of `dira`/`outa`/`ina` refers to a single bit in the register. Two values separated by two dots refer to a contiguous group of bits.

19) `%`, the binary number indicator.

20) The I/O pin is set to input, so it only monitors the voltage applied to the pin and stores a 1 in its `ina` bit if the voltage is above 1.65 V, or a 0 if it is below 1.65 V. As an input, the pin has no effect on external circuits.

21) Zero.

22) Assign-Equals `:=`, Post-Set `~~`, Post-Clear `~`, Bitwise NOT `!`, Limit Maximum `<#=`, Limit Minimum `#>=`, Pre- and Post-Increment `++`, Pre- and Post-Decrement `--`, Assign Shift Right `>>=`, and Assign Shift Left `<<=`.

23) Is Equal `==`, Is Not Equal `<>`, Is Less Than `<`, Is Greater Than `>`, Is Equal or Less `=<`, Is Equal or Greater `=>`.

24) `:=` is Assign-equals; whereas `==` is the comparison Is Equal. The result of `:=` assigns the value of the operand on the right to the operand on the left. The result of `==` simply compares two values, and returns `-1` if they are equal and `0` if they are not.

25) No, they are not necessary, though they can be useful. In this lab, the value returned by `ina` for a given bit was either 1 or 0, which worked fine for `if` blocks because the code would be executed if the condition is non-zero, or not executed if it's zero (-1 is non-zero).

26) Global and local. Global variables are declared in an object's `VAR` section. Local variables are only in use by a method as it executes.

27) The three sizes of variable are byte (0 to 255), word (0 to 65535) and long (`-2,147,483,648` to `2,147,483,647`). Local variables are automatically long-size, whereas global variables can be declared as byte, word, or long.

28) A pipe `|` character is used to declare local variables to the right of the method declaration. To the right of the pipe, more than one variable name may be declared, separated by commas.

Exercise Solutions

1) Solution:
```
outa[8..12] := dira[8..12] := %1111
```
2) Solution:
```
dira[9] := outa[9]:= 1
outa[13..15] := %000
dira[13..15] := %111
```

3) Solution:
```
     dira[0..8] :=%111000000.
```
4) Solution:
```
     outa[8]~~
     outa[9]~
     repeat
        !outa[8..9]
        waitcnt(clkfreq/100 + cnt)
```
5) Solution:
```
     repeat
        outa[0..7]!= ina[8..15]
```
6) Solution:
```
      CON
        _xinfreq = 5_000_000
        _clkmode = xtal1 + pll2x
```
7) Solution:
```
     waitcnt(clkfreq*5 + cnt)
```
8) Solution:
```
     outa[5..11]~~
     waitcnt(clkfreq*3 + cnt)
     outa[5..11] := %1010101
```
9) Solution:
```
     PUB LightsOn | counter
        dira[4..9] := %111111
        repeat counter from 4 to 9
           outa[counter] := 1
           waitcnt(clkfreq + cnt)
        repeat
```
10) Solution:
```
     PUB method
        dira[27] := 1
        repeat
           if ina[0]
              outa[27]~~
              waitcnt(clkfreq*5 + cnt)
           outa[27] ~
```
11) Solution:
```
     PUB SecondCountdown
        dira[9..4]~~
        repeat outa[9..4] from 59 to 0
           waitcnt(clkfreq + cnt)
```
12) Solution:
```
     PUB SecondCountdown
        dira[9..4]~~
        repeat
           repeat outa[9..4] from 59 to 0
              waitcnt(clkfreq + cnt)
```
13) Solution:
```
     PUB PushTwoStart
        dira[4]~~
        repeat until ina[23..21] == %101
        outa[4]~~
```
14) Solution:
```
     PUB PushTwoCountdown
        dira[9..4]~~
        repeat until ina[23..21] == %101
        outa[4]~~
        repeat outa[9..4] from 59 to 0
           waitcnt(clkfreq + cnt)
```

Project Solutions

1) Example solution:

```
''File: NonActuatedStreetlights.spin
''A high speed prototype of a N/S E/W streetlight controller.

PUB StreetLights

    dira[9..4]~~                              ' Set LED I/O pins to output

    repeat                                    ' Main loop

      outa[4..9] := %001100                   ' N/S green, E/W red
      waitcnt(clkfreq * 8 + cnt)              ' 8 s
      outa[4..9] := %010100                   ' N/S yellow, E/W red
      waitcnt(clkfreq * 3  + cnt)             ' 3 s
      outa[4..9] := %100001                   ' N/S red, E/W green
      waitcnt(clkfreq * 8 + cnt)              ' 8 s
      outa[4..9] := %100010                   ' N/S red, E/W yellow
      waitcnt(clkfreq * 3 + cnt)              ' 3 s
```

2) Example Solution:

```
''File: ActuatedStreetlightsEW.spin
''A high speed prototype of a N/S E/W streetlight controller.

PUB StreetLightsActuatedEW

    dira[9..4]~~                              ' Set LED I/O pins to output

    repeat                                    ' Main loop

      outa[4..9] := %001100                   ' N/S green, E/W red
      repeat until ina[21]                    ' Car on E/W street
      waitcnt(clkfreq * 3 + cnt)              ' 8 s
      outa[4..9] := %010100                   ' N/S yellow, E/W red
      waitcnt(clkfreq * 3  + cnt)             ' 3 s
      outa[4..9] := %100001                   ' N/S red, E/W green
      waitcnt(clkfreq * 8 + cnt)              ' 8 s
      outa[4..9] := %100010                   ' N/S red, E/W yellow
      waitcnt(clkfreq * 3 + cnt)              ' 3 s
```

3) Example solution:

```
''File: LedFrequenciesWithoutCogs.spin
''Experience the discomfort of developing processes that could otherwise run
''independently in separate cogs.  In this example, LEDs blink at 1, 2, 3, 5,
''7, and 11 Hz.

CON

    _xinfreq = 5_000_000                      ' 5 MHz external crystal
    _clkmode = xtal1 + pll16x                 ' 5 MHz crystal multiplied → 80 MHz

    T_LED_P4 = 2310                           ' Time increment constants
    T_LED_P5 = 1155
    T_LED_P6 = 770
    T_LED_P7 = 462
    T_LED_P8 = 330
    T_LED_P9 = 210

PUB Blinks | T, dT, count

    dira[9..4]~~                              ' Set LED I/O pins to output

      dT := clkfreq / 4620                    ' Set time increment
      T  := cnt                               ' Mark current time
```

```
    repeat                                      ' Main loop

       T += dT                                  ' Set next cnt target
       waitcnt(T)                               ' Wait for target

       if ++count == 2310                       ' Reset count every 2310
         count := 0

       ' Update each LED state at the correct count.
       if count // T_LED_P4 == 0
         !outa[4]
       if count // T_LED_P5 == 0
         !outa[5]
       if count // T_LED_P6 == 0
         !outa[6]
       if count // T_LED_P7 == 0
         !outa[7]
       if count // T_LED_P8 == 0
         !outa[8]
       if count // T_LED_P9 == 0
         !outa[9]
```

4) Example solution:

```
''File: MinuteSet.spin
''Emulates buttons that set alarm clock time.

PUB SetTimer | counter, divide

    dira[9..4]~~                                ' Set LED I/O pins to output

    repeat                                      ' Main loop

       'Delay for 1 ms.
       waitcnt(clkfreq/1000 + cnt)              ' Delay 1 ms

       {If a button is pressed...
       NOTE: Resetting the counter to -1 makes it possible to rapidly press
       and release the button and advance the minute display without the any
       apparent delay.}
       if ina[21] or ina[23]                    ' if a button is pressed
         counter++                              ' increment counter
       else                                     ' otherwise
         counter := -1                          ' set counter to -1

       'Reset minute overflows
       if outa[9..4] == 63                      ' If 0 rolls over to 63
         outa[9..4] := 59                       ' reset to 59
       elseif outa[9..4] == 60                  ' else if 59 increments to 60
         outa[9..4] := 0                        ' set to 0

       'Set counter ms time slice duration
       if counter > 2000                        ' If counter > 2000 (10 increments)
         divide := 50                           ' 50 ms between increments
       else                                     ' otherwise
         divide := 200                          ' 200 ms between increments

       'If one of the ms time slices has elapsed
       if counter // divide == 0                ' if a time slice has elapsed
         if ina[21]                             ' if P21 pushbutton is pressed
           outa[9..4]++                         ' increment outa[9..4]
         elseif ina[23]                         ' else if P23 pushbutton is pressed
           outa[9..4]--                         ' decrement outa[9..4]
```

5) Example solution:

```
''File: SecondCountdownTimer.spin
''Emulates buttons that set alarm clock time.

PUB SetTimerWiCountdown | counter, divide, T

    dira[9..4]~~                            ' Set LED I/O pins to output
    repeat                                  ' Main loop

        repeat until ina[22]                ' Break out if

            'Delay for 1 ms.
            waitcnt(clkfreq/1000 + cnt)     ' Delay 1 ms

            {If a button is pressed...
            NOTE: Resetting the counter to -1 makes it possible to rapidly press
            and release the button and advance the minute display without the any
            apparent delay.}
            if ina[21] or ina[23]           ' if a button is pressed
                counter++                   ' increment counter
            else                            ' otherwise
                counter := -1               ' set counter to -1

            'Reset minute overflows
            if outa[9..4] == 63             ' If 0 rolls over to 63
                outa[9..4] := 59            ' reset to 59
            elseif outa[9..4] == 60         ' else if 59 increments to 60
                outa[9..4] := 0             ' set to 0

            'Set counter ms time slice duration
            if counter > 2000               ' If counter > 2000 (10 increments)
                divide := 50                ' 50 ms between increments
            else                            ' otherwise
                divide := 200               ' 200 ms between increments

            'If one of the ms time slices has elapsed
            if counter // divide == 0       ' if a time slice has elapsed
                if ina[21]                  ' if P21 pushbutton is pressed
                    outa[9..4]++            ' increment outa[9..4]
                elseif ina[23]             ' else if P23 pushbutton is pressed
                    outa[9..4]--            ' decrement outa[9..4]

        T := cnt                            ' Mark the time
        repeat while outa[9..4]             ' Repeat while outa[9..4] is not 0
            T += clkfreq                    ' Calculate next second's clk value
            waitcnt(T)                      ' Wait for it...
            outa[9..4]--                    ' Decrement outa[9..4]
```

Tech Support Resources

Parallax Inc. offers several avenues through which to gain free technical support services:
- Email: support@parallax.com
- Fax: (916) 624-8006
- Telephone: Toll free in the U.S: (888) 99-STAMP; or (916) 624-8333, between the hours of 7:00 am and 5:00 pm Pacific time.
- Forums: http://forums.parallax.com/forums/. Here you will find an active forum dedicated to the Propeller chip, frequented by both Parallax customers and employees.