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

Methods and Cogs

PROPELLER EDUCATION KIT LAB SERIES

Introduction

Objects are organized into code building blocks called methods. Spin commands in methods can use other method's names to pass program control and optionally parameter values to those methods. When one method uses another method's name to pass it program control, it's called a method call. When the method that got called runs out of commands, it automatically returns program control and a result value to the line of code in the method that called it. Depending on how a method is written, it may also receive one or more parameter values when it gets called. Common uses for parameter values include configuration, defining the method's behavior, and input values for calculations.

Methods can also be launched into separate cogs so that their commands get processed in parallel with commands in other methods. The Spin language has commands for launching methods into cogs, identifying cogs, and stopping cogs. When Spin methods are launched into cogs, global variable arrays have to be declared to allocate memory for the methods to store return addresses, return values, parameters, and values used in calculations. This memory is commonly referred to as a stack or stack space.

This lab demonstrates techniques writing methods, calling methods, passing parameters to methods, and returning values from methods. This lab also demonstrates using method calls in commands that launch instances of methods into separate cogs, along with an overview of estimating how much stack space will be required for one or more spin methods that get executed by a given cog.

Prerequisite Labs

Setup and Testing
I/O and Timing Basics

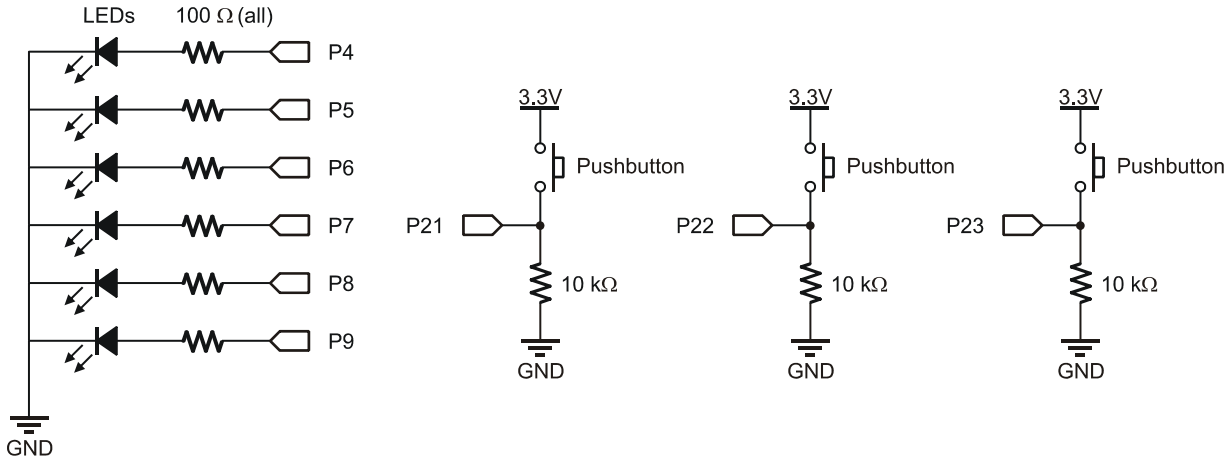
Parts List and Schematic

This lab will use six LED circuits and three pushbutton circuits (the same as I/O and Timing Basics)

(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



Defining a Method's Behavior with Local Variables

The AnotherBlinker object below uses three local variables, `pin`, `rate`, and `reps`, to define its **`repeat`** loop's LED on/off behavior. With the current variable settings, it makes P4 blink at 3 Hz for 9 on/off repetitions. Since the **`repeat`** loop only changes the LED state (instead of a complete on/off cycle), the object needs twice the number of state changes at half the specified delay between each state change. So, the `reps` variable has to be multiplied by 2 and `rate` has to be divided by 2. That's why the **`repeat`** loop repeats for `reps * 2` iterations instead of just `reps` iterations, and that's also why the **`waitcnt`** command uses `rate/2` instead of `rate` for the 3 Hz blink rate.

- ✓ Run the object, and verify that it makes the P4 LED blink at 3 Hz for 9 repetitions.
- ✓ Try a variety of `pin`, `rate` and `reps` settings and verify that they correctly define the **`repeat`** loop's behavior.

```
'' AnotherBlinker.spin

PUB Blink | pin, rate, reps

    pin := 4
    rate := clkfreq/3
    reps := 9

    dira[pin]~~
    outa[pin]~

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

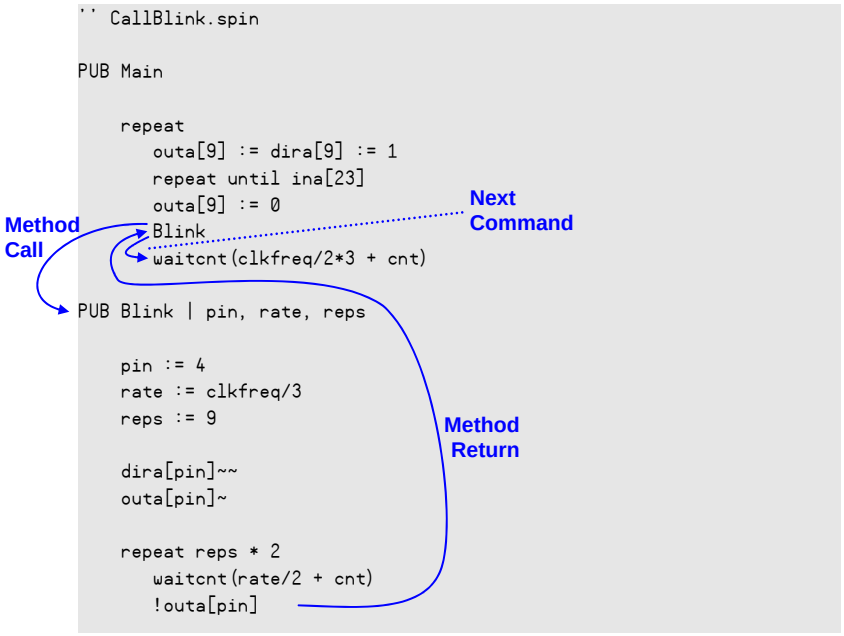
Calling a Method

The `Blink` method is used again in the next example object, CallBlink, along with another method named `Main`. Figure 2 shows how the `Blink` method is called from within the `Main` method. Program execution begins at `Main`, the first **`PUB`** block. When the program gets to the `Blink` line in the `Main` method, program control gets passed to the `Blink` method. That's a minimal version of a *method call*. When the `Blink` method is done blinking the LED 9 times, program control gets passed back to the `Blink` method call in the `Main` method. That's the *method return*, or just the "return."

Let's take a closer look at the CallBlink object's `main` method. It starts by turning on the P9 LED, to let the user know that the P23 pushbutton can be pressed. The **`repeat until ina`**`[23]` loop keeps repeating itself until the P23 button is pressed and the program moves on, turning off the P9 LED with **`outa`**`[9] := 0`. Then, it calls the `Blink` method, which blinks P4 at 3 Hz for 9 reps, and then returns. The next command is **`waitcnt`**`(`**`clkfreq`**`/2*3 + `**`cnt`**`)` which pauses for 3/2 s. Then, the outermost **`repeat`** loop in the `Main` method starts its next iteration. At that point, the P9 LED turns on again, indicating that the P23 pushbuttton can again trigger the P4, 3 Hz, 9 reps sequence.

- ✓ Load the CallBlink object into the Propeller chip.
- ✓ When the P9 LED turns on, press/release the P23 pushbutton.
- ✓ Wait for the P9 LED to turn on again after the P4 LED has blinked 9 times.
- ✓ Press/release the P23 pushbutton again to reinitiate the sequence.

Figure 2: Calling a Method



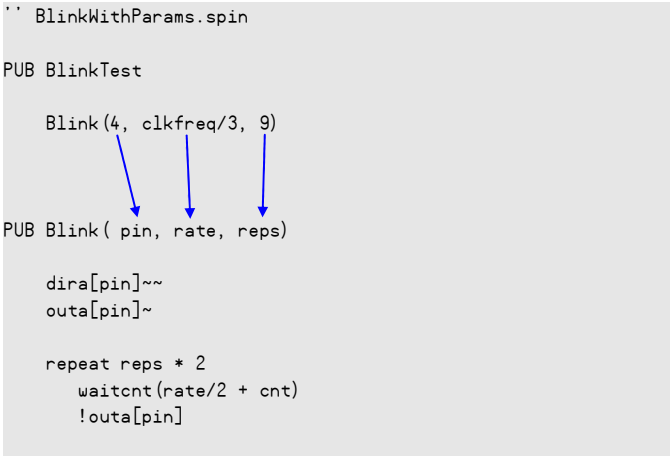
Parameter Passing

The `Blink` method we just used sets the values of its `pin`, `rate`, and `reps` local variables with individual *var* := *expression* instructions. To make methods more flexible and efficient to use, the value of their local variables can be defined in the method call instead of within the method.

Figure 3 below shows how this works in the BlinkWithParams object. The modified `Blink` method declaration now reads: `Blink(pin, rate, reps)`. The group of local variables between the parentheses is called the *parameter list*. Notice how the `Blink` method call in the `BlinkTest` method also has a parameter list. These parameter values get passed to the local variables in the `Blink` method declaration's parameter list. In this case, the `BlinkTest` passes `4` to `pin`, `clkfre/3` to `rate`, and `9` to `reps`. The result is the same as the AnotherBlinker object, but now code in one method can pass values to local variables in another method.

- ✓ Load BlinkWithParams into the Propeller chip and verify that the result is the same the previous AnotherBlinker object.
- ✓ Try adjusting the parameter values in the method call to adjust the `Blink` method's behavior.

Figure 3: Parameter Passing



```
'' BlinkWithParams.spin

PUB BlinkTest

    Blink(4, clkfreq/3, 9)


PUB Blink( pin, rate, reps)

    dira[pin]~~
    outa[pin]~

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

Methods can be re-used with different parameter values in each method call; here `Blink` is called three times with different parameters, and a 1 s pause in between.

```
PUB BlinkTest

    Blink(4, clkfreq/3, 9)
    waitcnt(clkfreq + cnt)
    Blink(5, clkfreq/7, 21)
    waitcnt(clkfreq + cnt)
    Blink(6, clkfreq/11, 39)
```

Here is another example that blinks a different LED each time the pushbutton is pressed and released. This is a variation of the `CallBlink` object's `Main` method, with a local variable named `led` and a **`repeat`** loop that sets the `led` variable to 4, 5, …, 8, 9, 4, 5, …8, 9, …. An updated `Blink` method call passes the value in the `led` variable to the `Blink` method's `pin` parameter. Since `led` changes with each iteration of the **`repeat`** `led`.... loop, the `pin` variable will receive a different value each time `Blink` is called. The result? Each time the pushbutton is pressed (after P9 lights up), a different LED will blink at 3 Hz for 9 reps.

```
PUB BlinkTest | led
    repeat
       repeat led from 4 to 9
          outa[9] := dira[9] := 1
          repeat until ina[23]
          outa[9] := 0
          Blink(led, clkfreq/3, 9)
          waitcnt(clkfreq/2*3 + cnt)
```

The `BlinkTest` method's local variable `led` could have been named `pin` because it's a local variable, so only code in the `BlinkTest` method uses it. Code in the `Blink` method also has a local variable `pin`, but again, only code in the `Blink` method will be aware of that `pin` variable's value.

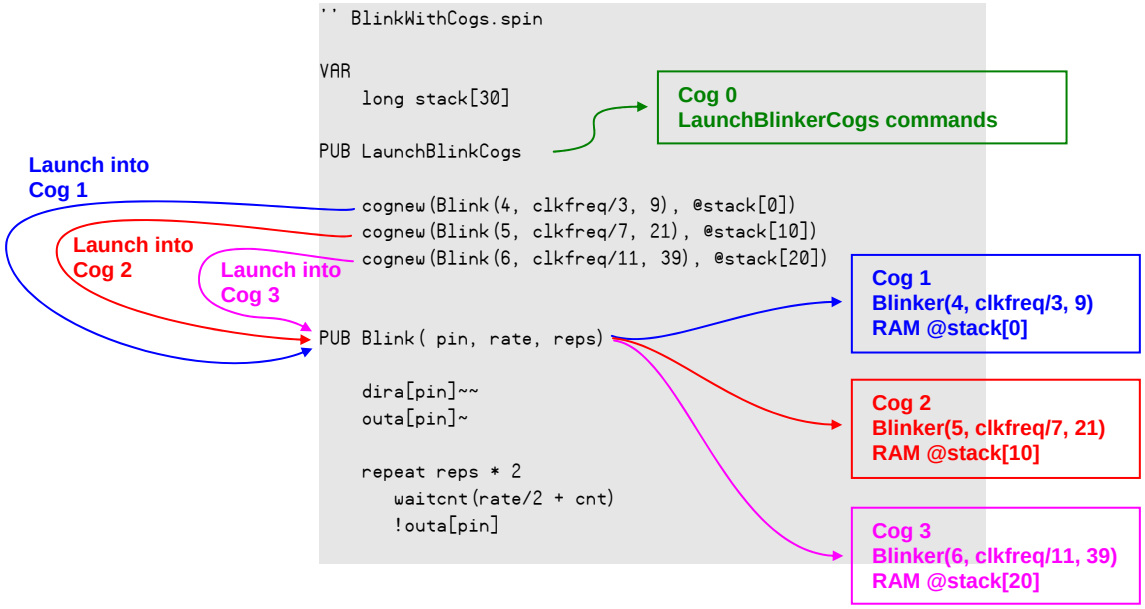
- ✓ Try the two modified versions of `BlinkTest` just discussed and make sure they make sense.
- ✓ Try changing the parameters so that the P4 LED blinks four times, P5 blinks 5 times, and so on.

Launching Methods into Cogs

All the methods in the objects up to this point have executed in just one of the Propeller chip's cogs, Cog 0. Each time the `Blink` method was called, it was called in sequence, so the LEDs blinked one at a time. The `Blink` method can also be launched into several different cogs, each with a different set of parameters to make the LEDs all blink at different rates simultaneously. The BlinkWithCogs object shown in Figure 4 demonstrates how to do this with three **`cognew`** commands.

The first method in a top level object automatically gets launched into Cog 0, so the Blinker object's `LaunchBlinkerCogs` method starts in Cog 0. It executes three **`cognew`** commands, and then runs out of instructions, so Cog 0 shuts down. Meanwhile, three other cogs have been started, each of which runs for about three seconds. After the last cog runs out of commands, the Propeller chip goes into low power mode.

Figure 4: Parameter Passing



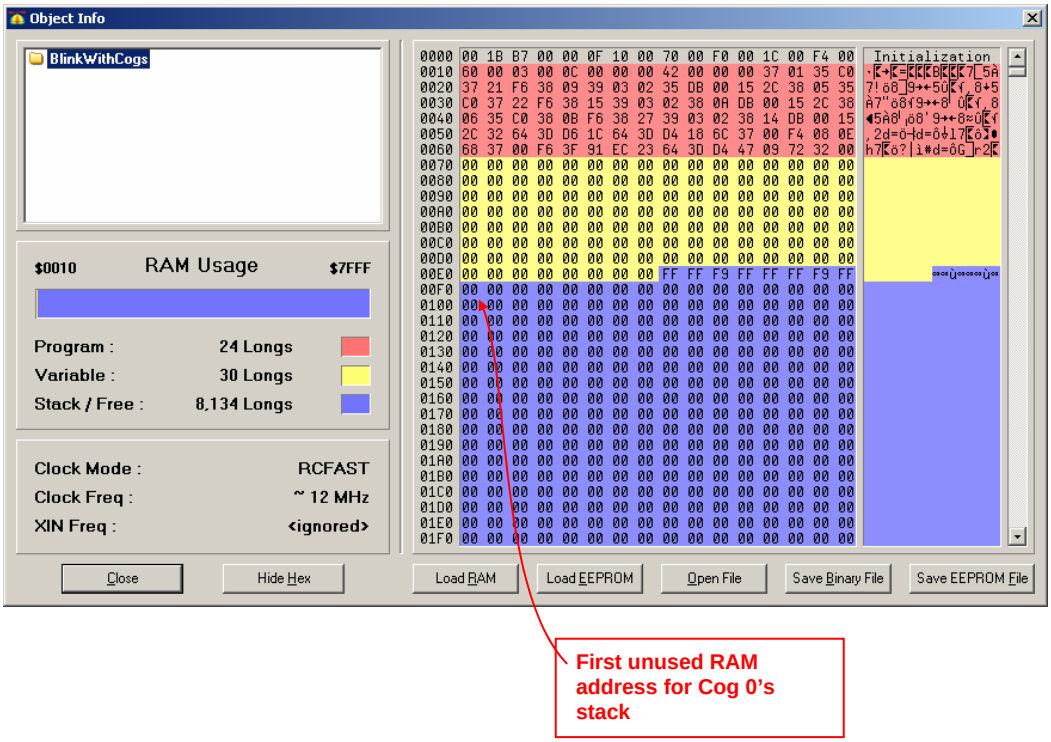
While Cog 0 accesses unused RAM that comes after the program codes to store method call return addresses, local variables and intermediate expression calculations, other cogs that execute Spin methods have to have variables set aside for them. Such variable space reserved in Global RAM for those temporary storage activities is called *stack space*, and the data stored there at any given moment is the *stack*. Notice that the `BlinkWithCogs` object in Figure 4 has a **`long`** `stack[30]` variable declaration. This declares an array of long variables named `stack` with 30 elements: `stack[0]`, `stack[1]`, `stack[2]`, …, `stack[28]`, `stack[29]`.

The command **`cognew`**`(Blink(4, clkfreq/3, 9), @stack[0])` calls the Blink method with the parameters 4, **`clkfreq`**/3, and 9 into the next available cog, which happens to be Cog 1. The **`@`**`stack[0]` argument passes the address of the `stack[0]` array element to Cog 1. So Cog 1 starts executing `Blink(4, clkfreq/3, 9)` using `stack[0]` and upward for its return address, local variables, and intermediate calculations. The command **`cognew`**`(Blink(5, clkfreq/7, 21), @stack[10])` launches `Blink(5, clkfreq/7, 20)` into Cog 2, with a pointer to `stack[10]`'s address in RAM so it uses from `stack[10]` and upwards. Then **`cognew`**`(Blink(6, clkfreq/11, 33), @stack[20])` does it again with different `Blink` method parameters and a different address in the `stack` array.

- ✓ Load the BlinkWithCogs object into the Propeller chip and verify that it makes the three LEDs blink at different rates at the same time (instead of in sequence).
- ✓ Examine the program and make notes of the new elements.

The unused RAM that Cog 0 uses for its stack can be viewed with the Object Info window shown in Figure 5. The gray color-coded bytes are initialization codes that launch the top level object into a cog, set the propeller's **clk** register, and various other initialization tasks. The red memory addresses store Spin program codes, the yellow indicates global variable space (the 30 long variable `stack` array), and what follows is blue unused RAM, some of which will be used by Cog 0 for its stack. The beginning RAM address of cog 0's stack is hexadecimal 00F0.

Figure 5: Object Info Window



Stopping Cogs

With **cognew** commands, the Propeller chip always looks for the next available cog and starts it automatically. In the BlinkWithCogs object, the pattern of cog assignments is predictable: the first **cognew** command launches `Blink(4, clkfreq/3, 9)` into Cog 1, `Blink(5, clkfreq/7, 21)` into Cog 2, and Blink(6, clkfreq/11, 39) into Cog 3.

Choose your Cog: Instead of using the next available cog, you can specify which cog you wish to launch by using the **coginit** command instead of **cognew**. For example, this command will launch Cog 6:

```
coginit(6, Blink(4, clkfreq/3, 9), @stack[0])
```

The **cogstop** command can be used to stop each of these cogs. Here is an example with each `reps` parameter set so that the object will keep flashing LEDs until one million repetitions have elapsed. After a 3 second delay, **cogstop** commands shut down each cog at one second intervals using the predicted cog ID so that none of the methods get close to one million reps.

```
PUB LaunchBlinkCogs

  cognew(Blink(4, clkfreq/3, 1_000_000), @stack[0])
  cognew(Blink(5, clkfreq/7, 1_000_000), @stack[10])
  cognew(Blink(6, clkfreq/11, 1_000_000), @stack[20])
  waitcnt(clkfreq * 3 + cnt)
  cogstop(1)
  waitcnt(clkfreq + cnt)
  cogstop(2)
  waitcnt(clkfreq + cnt)
  cogstop(3)
```

With some indexing tricks, the cogs can even be launched and shut down with **repeat** loops. Below is an example that uses an index local variable in a **repeat** loop to define the I/O pin, stack array element, and cog ID. It does exactly the same thing as the modified version of the LaunchBlinkCogs method above. Notice that the local variable index is declared with the pipe symbol. Then, **repeat** index **from** 0 **to** 2 increments index each time through the three **cognew** command executions. When index is 0, the Blink method call's pin parameter is 0 + 4, passing 4 to the Blink method's pin parameter. The second time through, index is 1, so pin becomes 5, and the third time through, it makes pin 6. For the **clkfreq** sequence of 3, 7, 11 with index values of 0, 1, and 2, (index * 4) + 3 fits the bill. For 0, 10, and 20 as the array element, index * 10 fits the bill. To stop cogs 1, 2, and 3, the second **repeat** loop sweeps index from 1 to 3. The first time through the loop, index is 1, so **cogstop**(index) becomes **cogstop**(1). The second time through, index is 2, so **cogstop**(2), and the third time through, index is 3 resulting in **cogstop**(3).

```
PUB LaunchBlinkCogs | index

    repeat index from 0 to 2
       cognew(Blink(index + 4, clkfreq/((index*4) + 3), 1_000_000), @stack[index * 10])

    waitcnt(clkfreq * 3 + cnt)

    repeat index from 1 to 3
       cogstop(index)
       waitcnt(clkfreq + cnt)
```

✓ Try the modified versions of the LaunchBlinkCogs methods.

Objects can be written so that they keep track of which cog is executing a certain method. One approach will be introduced in the Cog ID Indexing section on page 10. Other approaches will be introduced in the Objects lab.

How Much Stack Space for a Method Launched into a Cog?

Below is a list of the number of longs each method adds to the stack when it gets called.

- 2 – return address
- 1 – return result
- number of method parameters
- number of local variables
- workspace for intermediate expression calculations

Assume you have an object with three methods, A, B and C. When Method A calls Method B, the stack will grow, containing two sets of these longs, one for Method A, and one for Method B. If Method B calls Method C, there will be a third set. When Method C returns, the stack drops down to two sets.

The workspace is for storing values that exist during certain tasks and expression evaluations. For example, the Blink method's **repeat** reps * 2 uses the workspace in two different ways. First, the reps * 2 expression causes two elements to be pushed to the stack: the value stored by reps and 2. After the * calculation, 2 is popped from the stack, and the result of the calculation is stored in a single element. This element stays on the stack until the **repeat** loop is finished. Inside the **repeat** reps * 2 loop, two similar expansions and contractions of the stack occur with **waitcnt**(rate/2 + cnt), first with rate/2, and again when the result of rate/2 is added to **cnt**.

In this case of the Blink method, the most it uses for workspace and intermediate expression calculations is 3 longs: one long for holding the result of reps * 2 until the **repeat** loop is done, and two more for the various calculations with binary operators such as multiply (*) and divide (/). Knowing this, we can tally up the number of long variables a cog's stack will need to execute this method are listed below. So, the total number of stack space (long variables) a cog needs to execute the Blink method is 10.

- 2 – return address
- 1 – return parameter (even though it was not declared, every method has a built-in return parameter, which will be introduced in the next section)
- 3 – pin, freq, and reps parameters
- 1 – time local variable
- 3 – workspace for calculations.

- 10 – Total

As mentioned earlier, one cog needs enough stack space to for all the memory it might use, along with all the stack space of any method it calls. Some methods will have nested method calls, where Method A calls Method B, which in turn calls Method C. All those methods would need stack memory allocated if Method A is the one getting launched into the cog.

Err on the side of caution: The best way to set aside stack space for a cog that gets a Spin method launched into it is to err on the side of caution and declare way more memory that you think you'll need. Then, you can use an object in the Propeller Tool's object library (the folder the Propeller.exe file lives in) named Stack Length.spin to find out how many variables the method actually used. The Objects Lab will feature a project that uses the Stack Length object to verify the number of long variables required for a Spin method that gets launched into a cog.

Declaring a long variable array named stack in an object's **VAR** code block is a way of setting aside extra RAM for a cog that's going to run a Spin interpreter. The name of the array doesn't have to be stack; it just has to be a name the Spin language can use for variable name. The names blinkStack or methodStack would work fine too, so long as the name that is chosen is also the one whose address gets passed to the cog by the **cognew** command. Remember that the @ operator to the left of the variable name is what specifies the variable's ram address.

About _STACK: The Spin language also has a **_stack** constant, which can be used in **CON** blocks to reserve enough stack space for the top level method that gets launched into Cog 0. Unlike cogs that get launched with **cognew** that have coded addresses of array stack space, the Spin interpreter that gets launched into Cog 0 when the Propeller chip boots uses unallocated RAM that immediately follows the last program code. In large programs or programs that reserve a lot of memory, **_stack** can manually set aside enough memory for Cog 0's stack in unused RAM.

Return Parameter

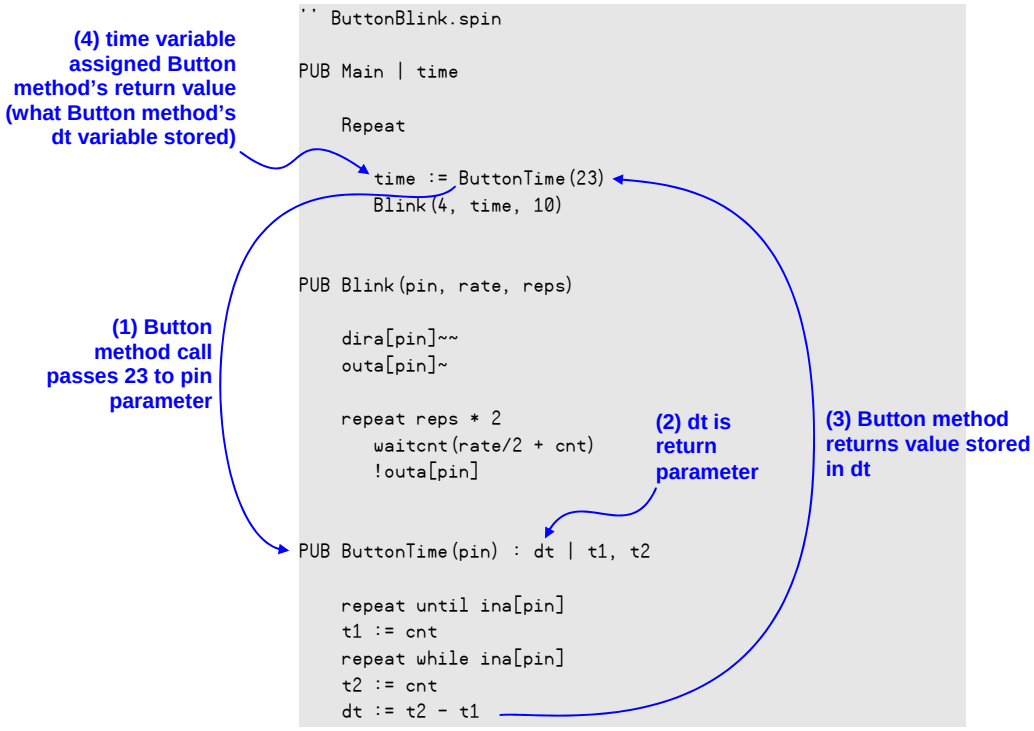
Method declarations can also specify a return parameter. A return parameter is the name of a variable whose value gets sent back to the method call. Figure 6 shows an important use of this feature where the value returned gets assigned to a variable, in this case with the command `time := ButtonTime(23)`. This command stores the result of the `ButtonTime(23)` method call in the `time` variable. Here's how it works:

1. `ButtonTime(23)` method calls the `ButtonTime` method, passing 23 to it's `pin` parameter.
2. `:dt` in the `ButtonTime` method declaration specifies the `dt` variable as its return parameter.
3. When the `ButtonTime` method returns, the method call becomes the value stored in the method's return parameter variable.
4. The `time := ButtonTime(23)` assignment operation copies the method's return parameter to the `time` variable so that the `Main` method can use it.

The ButtonBlink object's `ButtonTime` method measures and returns the approximate amount of time the button is pressed. The `Blink(4, time, 10)` command in the `Main` method passes this `time` measurement value to the `Blink` method's `rate` parameter. This in turn makes the P4 LED blink on/off at a rate that matches the time the LED was held down. Each time you press and release the P23 pushbutton, the ButtonBlink object uses the `ButtonTime` method to measure the time the pushbutton was pressed and held, and then flashes the LED 10 times, with each on/off cycle lasting as long as the pushbutton press.

- ✓ Load ButtonBlinkTime into the Propeller chip.
- ✓ Press and release the LED, and observe that the LED blinks ten times at a rate determined by how long you held the button down.
- ✓ After the LED finishes blinking, press and hold the pushbutton down for a different amount of time to set a different blink rate.
- ✓ Try various durations from a quick tap on the pushbutton to holding it down for a few seconds.

Figure 6: Parameter Passing



The Result Variable

Whenever a method is called, it sets aside a local variable named **result**. Whenever a method is done, it always returns the value stored by **result**, regardless of whether or not the optional return parameter is specified. When a return parameter is specified, it actually just provides an alias to the method's **result** variable. This alias name is useful, especially for making the code self documenting, but it is not required.

Below is a modified version of the ButtonTime method that demonstrates how the **result** variable can be used instead of a declared alias name for the return parameter. The :dt has been removed from the method declaration, and the last line now reads **result** := t2 – t1 instead of dt := t2 – t1. Keep in mind that dt was really just an alias to the **result** local variable and methods always return the value stored in **result**. So, from the method call's standpoint, this revised method still functions identically to the one in the ButtonBlink object.

```
PUB ButtonTime(pin) | t1, t2          ' Optional return parameter local variable name removed

    repeat until ina[pin]
    t1 := cnt
    repeat while ina[pin]
    t2 := cnt
    result := t2 - t1                 ' Value stored by result is automatically returned
```

✓ Substitute this modified version of the **ButtonTime** method into the ButtonBlink object and verify that it works the same.

Cog ID Indexing

As mentioned earlier, objects can't necessarily predict which cog a given method will get launched into. Just like methods return values, the **cognew** command returns the ID of the cog it launched a method into. Each time a method gets launched into a new cog, the cog ID the **cognew** command returns can be stored in a variable. This makes it possible to keep track of what each cog is doing.

The CogStartStopWithButton object demonstrates keeping track of cog IDs with an array variable in an application that launches a new cog each time the pushbutton is pressed and released. It uses the same `ButtonTime` method from the previous example object to measure the time the pushbutton was held down. Then, it launches the `Blink` method into a new cog with the `time` measurement determining the blink rate. The result, an application where each time you press and release the pushbutton, another LED starts blinking at a rate that matches the time you held down the pushbutton. After the sixth pushbutton press/release, the next six pushbuttons press/releases will shut down the cogs in reverse sequence. Since all the cog starting and stopping is nested into a repeat loop with no conditions, the 13th time you press/release the P23 pushbutton will have the same effect as the first press/release.

- ✓ Load CogStartStopWithButton into the Propeller chip, and use the P23 pushbutton to successively launch Blink cogs.
- ✓ Try a variety of button press times so that each LED is obviously blinking at a different rate.
- ✓ Make sure to press/release the P23 pushbutton at least twelve times to launch and then shut down cogs 1 through 7.

```
'' File: CogStartStopWithButton.spin

VAR

    long stack[60]

PUB ButtonBlinkTime | time, index, cog[6]

    repeat

       repeat index from 0 to 5
          time := ButtonTime(23)
          cog[index] := cognew(Blink(index + 4, time, 1_000_000), @stack[index * 10])

       repeat index from 5 to 0
          ButtonTime(23)
          cogstop(cog[index])


PUB Blink( pin, rate, reps)

    dira[pin]~~
    outa[pin]~

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


PUB ButtonTime(pin) : delta | time1, time2

    repeat until ina[pin] == 1
    time1 := cnt
    repeat until ina[pin] == 0
    time2 := cnt
    delta := time2 - time1
```

Inside ButtonBlinkTime

The CogStartStopWithButton object's ButtonBlinkTime method is declared with eight local variables, time, index, and an array named cog with six elements. The **repeat** command under the method declaration repeats the rest of the commands in the method since they are all indented further. Because this **repeat** command has no conditions, the rest of the commands in the method get repeated indefinitely.

```
PUB ButtonBlinkTime | time, index, cog[6]

    repeat
```

The first nested **repeat** loop increments the index variable from 0 to 5 each time through. The first command it repeats is time := ButtonTime(23), which gets a new button time measurement each time it's called. Next, cog[index] := **cognew**… launches Blink(index + 4, time, 1_000_000) into a new cog. The **cognew** command returns the cog ID, which gets stored in cog[index]. The first time through, index is 0, so the command becomes cog[0] := **cognew**(Blink(4, time, 1_000_000), @stack[0]). The second time through, it's cog[1] := **cognew**(Blink(5, time, 1_000_000), @stack[10]). The third time through, it's cog[2] := **cognew**(Blink(6, time, 1_000_000), @stack[20]), and so on. So, cog[0], cog[1], up through cog[5] each stores the cog ID for a different cog that a different version of Blink was launched into.

```
repeat index from 0 to 5
   time := ButtonTime(23)
   cog[index] := cognew(Blink(index + 4, time, 1_000_000), @stack[index * 10])
```

After the sixth button press/release, the code enters this **repeat** loop. Notice how the ButtonTime method gets called, but its return value doesn't get stored in the time variable. That's because this method is just being used to wait for the next pushbutton press/release so that it can shut down the next cog. Since nothing is done with its return parameter, it doesn't need to be stored by the time variable. This **repeat** loop goes from 5 to 0. So the first time through, **cogstop** will shut down the cog with the ID stored in cog[5]. The second time through, it will shut down the cog with the ID stored in cog[4], and so on, down to cog[0].

```
repeat index from 5 to 0
   ButtonTime(23)
   cogstop(cog[index])
```