PARALLAX
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

Objects

PROPELLER EDUCATION KIT LAB SERIES

Introduction

In the first three labs (*Setup and Testing*, *I/O and Timing*, and *Methods and Cogs*), all the application code examples were individual objects. However, applications are typically organized as collections of objects. Every application has a *top level object*, which is the object where the code execution starts. Top level objects can declare and call methods in one or more other objects. Those objects might in turn declare and call methods in other objects, and so on…

A lot of objects that get incorporated into an application are designed to simplify development. Some of these objects are collections of useful methods that have been published so that common coding tasks don't have to be done "from scratch." Other objects manage processes that get launched into cogs. They usually cover the tasks introduced in the Methods and Cogs lab, including declaring stack space and tracking which cog the process gets launched into. These objects that manage cogs also have methods for starting and stopping the processes.

Useful objects that can be incorporated into your application are available from a number of sources, including the Propeller Tool software's Propeller Library, the Propeller Object Exchange at obex.parallax.com, and the Propeller Chip forum at forums.parallax.com. Each object typically has documentation that explains how to incorporate it into your application along with one or more example top files that demonstrate how to declare the object and call its methods. In addition to using pre-written objects, you may find yourself wanting to modify an existing object to suit your application's needs, or even write a custom object. If you write an object that solves problems or performs tasks that are not yet available elsewhere, consider posting it to the Propeller Object Exchange.

This lab guides you through writing a variety of objects and incorporating them into your applications. Some of the objects are just collections of useful methods, while others manage processes that get launched into cogs. Some of the objects will be written from scratch, and others from the Propeller Library will be used as resources. The example applications will guide you through how to:

- Call methods in other objects
- Use objects that launch processes into cogs
- Write code that calls an object's methods based on its documentation
- Write object documentation and schematics
- Use objects from the Propeller Object library
- Access values and variables by their memory addresses
- Use objects to launch cogs that read and/or update the parent object's variables.

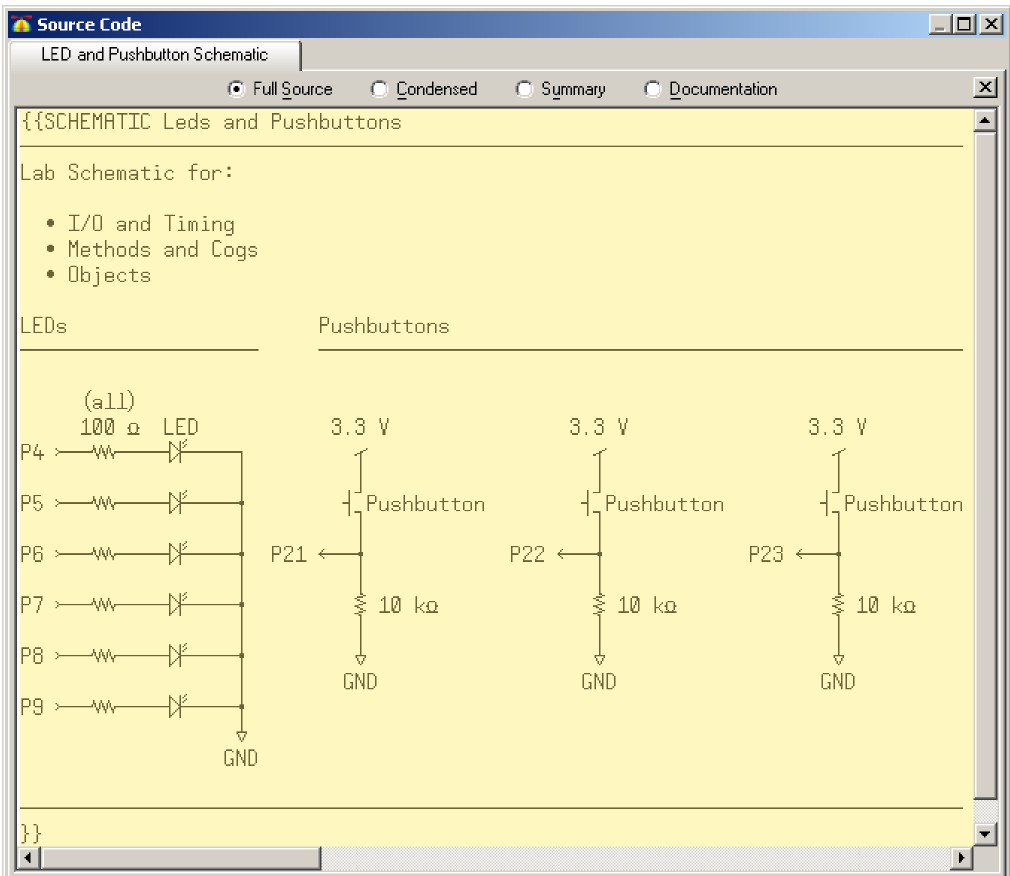
Prerequisites

Please complete the *Setup and Testing*, *I/O and Timing*, and *Methods and Cogs* labs before continuing here.

Equipment, Parts, Schematic

Although the circuit is the same one used in the previous two labs, there are a few twists. First, the schematic shown in Figure 1 was drawn using the Parallax font and the Propeller Tool software's character chart, which is an important component of documenting objects. Second, some of the coding examples allow you to monitor and control elements of the circuit from your PC with software bundled with this lab called Parallax Serial Terminal (PST.exe). The Propeller applications that communicate serially with Parallax Serial Terminal will do so with the help of an object named FullDuplexSerial.spin.

Figure 1: Schematic (drawn with the Propeller Tool software)



Method Call Review

The ButtonBlink object below is an example from the Methods and Cogs lab. Every time you press and release the pushbutton connected to P23, the object measures the approximate time the button is held down, and uses it to determine the full blink on/off period, and blinks the LED ten times. (Button debouncing is not required with the pushbuttons included in the PE kit.) The object accomplishes these tasks by calling other methods in the same object. Code in the `Main` method calls the `ButtonTime` method to get the time the button is held down. When `ButtonTime` returns a value, the `Blink` method gets called, with one of the parameters being the result of the `ButtonTime` measurement.

✓ Load ButtonBlink into the Propeller chip and test to make sure you can use the P23 pushbutton to set the P4 LED blink period.

```
'' ButtonBlink.spin

PUB Main | time

    Repeat

       time := ButtonTime(23)
       Blink(4, time, 10)


PUB Blink(pin, rate, reps)

    dira[pin]~~
    outa[pin]~

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

PUB ButtonTime(pin) : dt | t1, t2

    repeat until ina[pin]
    t1 := cnt
    repeat while ina[pin]
    t2 := cnt
    dt := t2 - t1
```

Calling Methods in Other Objects with Dot Notation

The ButtonBlink object's `ButtonTime` and `Blink` methods provide a simple example of code that might be useful in a number of different applications. These methods can be stored in a separate object file, and then any object that needs to blink an LED or measure a pushbutton press can access these methods by following two steps:

1) Declare the object in an `OBJ` code block, and give the object's filename a nickname.
2) Use *ObjectNickname.MethodName* to call the object's method.

> The Propeller Manual uses the term "symbolic reference" or "reference" instead of "nickname".

Figure 2 shows an example of how this works. The `ButtonTime` and `Blink` methods have been moved to an object named ButtonAndBlink. To get access to the ButtonAndBlink object's public methods, the DotNotationExample object has to start by declaring the ButtonAndBlink object and giving it a nickname. These object declarations are done in the DotNotationExample object's `OBJ` code block. The declaration `PbLed : "ButtonAndBlink"` gives the nickname `PbLed` to the ButtonAndBlink object.

The `PbLed` declaration makes it possible for the DotNotationExample object to call methods in the ButtonAndBlink object using the notation *ObjectNickname.MethodName*. So, DotNotationExample uses `time := PbLed.ButtonTime(23)` to call ButtonAndBlink's `ButtonTime` method, pass it the parameter 23, and assign the returned result to the `time` variable. DotNotationExample also uses the command `PbLed.Blink(4, time, 20)` to pass 4, the value stored in the `time` variable, and 20 to ButtonAndBlink's `Blink` method.

File Locations: An object has to either be in the same folder with the object that's declaring it, or in the same folder with the Propeller Tool.exe file. Objects stored with the Propeller Tool are commonly referred to as library objects.

Figure 2: Calling Methods in Another Object with Dot Notation

```
''File: DotNotationExample.spin

OBJ

  PbLed : "ButtonAndBlink"


PUB Main | time

    repeat

        time := PbLed.ButtonTime(23)

        PbLed.Blink(4, time, 20)
```

Object declaration

Method calls with ObjectNickname.MethodName

```
'' File: ButtonAndBlink.spin
'' Example object with two methods

PUB ButtonTime(pin): delta | time1, time2

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

PUB Blink( pin, rate, reps)

    dira[pin]~~
    outa[pin]~

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

- ✓ Load the DotNotationExample object into the Propeller chip. If you are hand entering this code, make sure to save both files in the same folder. Also, the ButtonAndBlink object's filename must be ButtonAndBlink.spin.
- ✓ Verify that the program does the same job as the previous example object (ButtonBlink).
- ✓ Follow the steps in Figure 2, and make sure it's clear how ButtonAndBlink gets a nickname in the `OBJ` section, and how that nickname is then used by DotNotationExample to call methods within the ButtonAndBlink object.
- ✓ Compare DotNotationExample.spin to the previous example object (ButtonBlink).

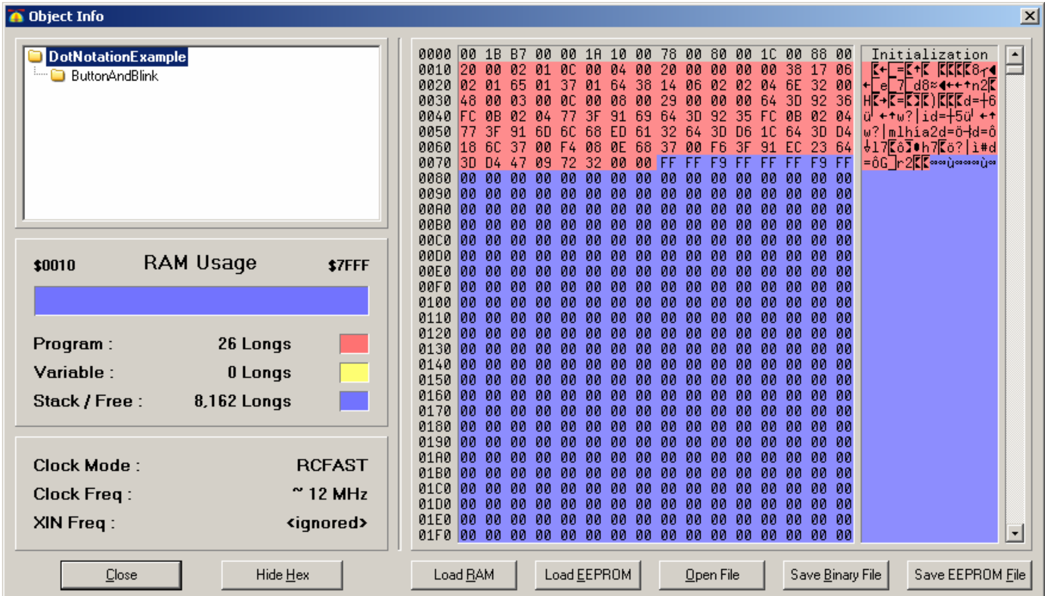
Object Organization

Objects can declare objects that can in turn declare other objects. It's important to be able to examine the interrelationships between parent objects, their children, grandchildren, and so on. There are a couple of ways to examine these object family trees. First, let's try viewing the relationships in the Object Info window with the Propeller Tool's Compile Current feature:

- ✓ Click the Propeller Tool's Run menu, and select Compile Current → View Info (F8).

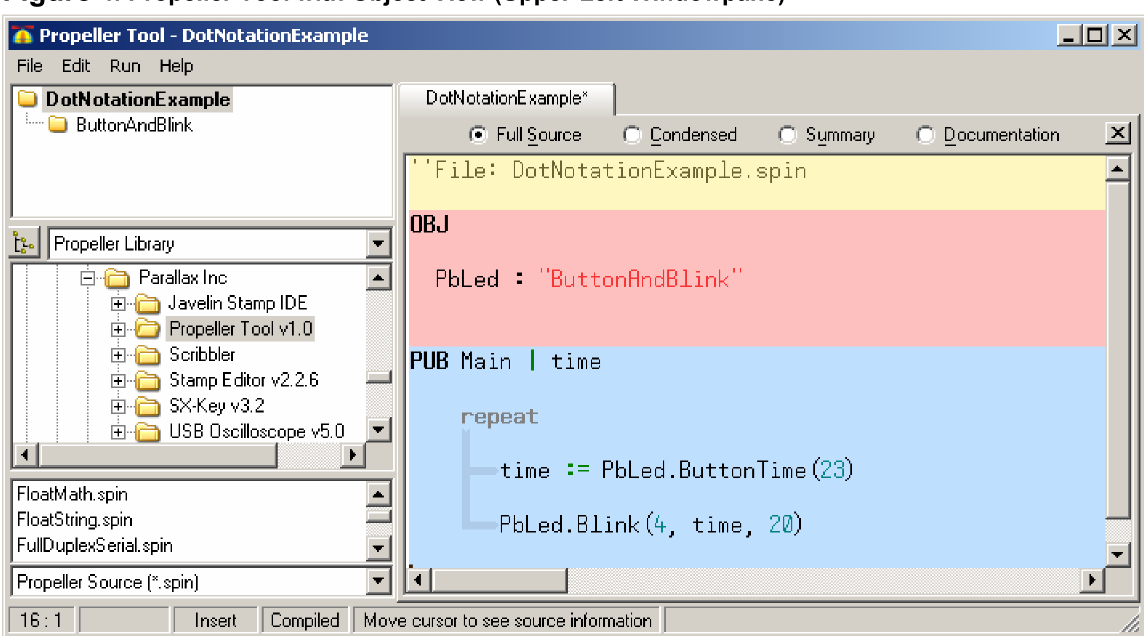
Notice that the object hierarchy is shown in the Object Info window's top-left corner. In this windowpane, you can single click each folder to see how much memory it occupies in the Propeller Chip's global RAM. You can also double-click each folder in the Object Info window to open the .spin file that contains the object code. Since DotNotationExample declared ButtonAndBlink, the ButtonAndBlink code becomes part of the DotNotationExample application, which is why it appears to have more code than ButtonAndBlink in the Object Info window even though it has much less actual typed code.

Figure 3: Object Info Window



After closing the Object Info window, the same Object View pane will be visible in the upper-left corner of the Propeller tool (see Figure 4). The objects in this pane can be opened with a single-click. The file folder icons can also be right-clicked to view a given object in documentation mode. They can then be left-clicked to return to Full Source view mode.

Figure 4: Propeller Tool with Object View (Upper-Left Windowpane)



Objects that Launch Processes into Cogs

In the Methods Lab, it took several steps to write a program that launches a method into a cog. First, additional variables had to be declared to give the cog stack space and track which cog is running which process before the `cognew` or `cogstart` commands could be used. Also, a variable that stored the cog's ID was needed to pick the right cog if the program needed to stop a given process.

Objects that launch processes into cogs can take care of all that for you. For example, here is a top file that declares two objects, named Button and Blinker. The Blinker object has a method named `Start` that takes care of launching its `Blink` method into a new cog and all the variable bookkeeping that accompanies it. So, all this top level object has to do is call the Blinker object's `Start` method.

```
{{
Top File: CogObjectExample.spin
Blinks an LED circuit for 20 repetitions.  The LED
blink period is determined by how long the P23 pushbutton
is pressed and held.
}}

OBJ

    Blinker : "Blinker"
    Button  : "Button"


PUB ButtonBlinkTime | time

    repeat

       time := Button.Time(23)
       Blinker.Start(4, time, 20)
```

Unlike the DotNotationExample object, you won't have to wait for 20 LED blinks before pressing the button again to change the blink rate (for the next 20 blinks). There are two reasons why. First, the Blinker object automatically launches the LED blinking process into a new cog. This leaves Cog 0 free to monitor the pushbutton for the next press/release while Cog 1 blinks the LED. Second, the Blinker object's `Start` method automatically stops any process it's currently running before launching the new process. So, as soon as the button measurement gets taken with `Button.Time(23)`, the `Blinker.Start` method stops any process (cog) that it might already be running before it launches the new process.

- ✓ If you are using the pre-written .spin files that accompany this PDF, they will already all be in the same folder. If you are hand entering code, make sure to hand enter and save all three objects in the same folder. The objects that will have to be saved are CogObjectExample (above), and Blinker , and Button (both below).
- ✓ Load CogObjectExample into the Propeller Chip.
- ✓ Try pressing and releasing the P23 pushbutton so that it makes the LED blink slowly.
- ✓ Before the 20th blink, press and release the P23 pushbutton rapidly. The LED should immediately start blinking at the faster rate.

Inside the Blinker Object

Objects that launch processes into cogs are typically written to take care of most cog record-keeping. Then, all a parent object has to do is declare the object, and then launch the process by calling the object's `Start` method, or halt it by calling the object's `Stop` method. For example, the Blinker example object below has the necessary variable array for the cog's stack operations while executing the `Blink` method. It also has another variable named `cog` for keeping track of which cog it launched its `Blink` method into.

The Blinker object also has the `Start` and `Stop` methods for launching the now familiar `Blink` method into a new cog and stopping it again. When the `Start` method launches the `Blink` method into a new cog, it copies the cog ID into the `cog` variable. The value it returns in the `success` variable is the cog ID + 1, which the parent object can treat as a Boolean value. So long as this value is non-zero, it means the process launched successfully. If the value is zero, it means the cog was not successfully launched. This typically happens when all eight of the Propeller chip's cogs are already in use.

The object's `Stop` method shuts the process down, using the `cog` variable, which the object uses to store the ID of the cog it launched the `Blink` method into.

```
{{
File: Blinker.spin
Example cog manager for a blinking LED process.

SCHEMATIC
───────────────────────────────
          100 Ω  LED
    pin ──────────────┐
                          │
                          
                         GND
───────────────────────────────
}}

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


PUB Start(pin, rate, reps) : success
{{Start new blinking process in new cog; return True if successful.

Parameters:
  pin - the I/O connected to the LED circuit → see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles
}}
  Stop
  success := (cog := cognew(Blink(pin, rate, reps), @stack) + 1)


PUB Stop
''Stop blinking process, if any.

  if Cog
    cogstop(Cog~ - 1)


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

Parameters:
```

```
  pin - the I/O connected to the LED circuit → see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles
}}

    dira[pin]~~
    outa[pin]~

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

The `Start` and `Stop` methods shown in this object are the recommended approach for objects that manage cogs. They were copied verbatim from the Propeller Manual's tutorial section, and then updated to fit the slightly different `Blink` method. The `Start` method's parameter list should have all the parameters the process will need to get launched into a cog. Note that these values are passed to the object's `Blink` method via a call in the `cognew` command.

> **Why does the Start method call the Stop method?** In the event that the object had already started a process, the Stop method call shuts that process down before launching a new process.

CogObjectExample also uses the Button object, which at this time has just one method, but it can be expanded into a collection of useful methods. Note that this version of the Button object doesn't launch any new processes into cogs, so it doesn't have a `Start` or `Stop` method.

Everything the Button object does is done in the same cog as the object that calls it. This object could be modified in several different ways. For example, other button-related methods could be added. The object could also be modified to work with a certain button or group of buttons. It could also have an `Init` or `Config` method added to set the object up to automatically monitor a certain button or group of buttons. The object could also be modified to monitor these buttons in a separate cog, but in that case, `Start` and `Stop` methods should be added.

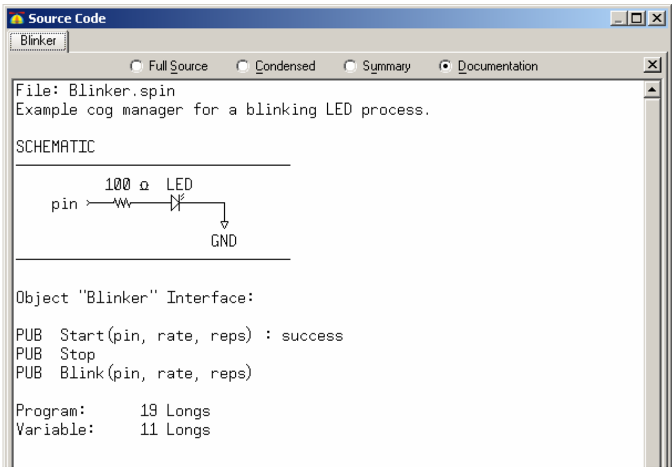
```
'' File: Button.spin
'' Beginnings of a useful object.

PUB Time(pin) : delta | time1, time2

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

Documentation Comments

Figure 5 shows the first part of the Blinker object displayed in documentation mode. To view the object in this mode, make sure it's the active tab (click the tab with the Blinker filename), then click the Documentation radio button just above the code. Remember from the I/O and Timing Lab that single line documentation comments are preceded by two apostrophes: `''comment`, and that documentation comments occupying more than one line are started and ended with double braces: `{{comments}}`. Take a look at the documentation comments in Full Source mode, and compare them to the comments in Documentation mode.

Documentation mode automatically adds some information above and beyond what's in the documentation comments. First, there's the Object Interface information which is a list of the object's public method declarations, including the method name, parameter list, and return variable name, if any. This gives the programmer an "at a glance" view of the object's methods. With this in mind, it's important to choose descriptive names for an object's method and its parameters. Documentation mode also lists how much memory the object's use would add to a program and how much it takes in the way of variables. These, of course, are also important "at a glance" features.

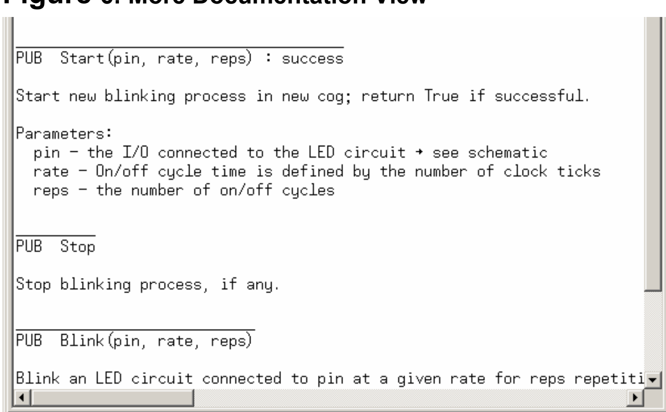
Figure 5: Documentation View



The Documentation view mode also inserts each method declaration (without local variables that are not used as parameters or return variable aliases). Notice how documentation comments below the method declaration also appear, and how they explain what the method does, what information its parameters should receive, and what it returns. Each public method's documentation should have enough information for a programmer to use it without switching back to Full Source view to reverse engineer the method and try to figure out what it does. This is another good reason to pick your method and parameter names carefully, because they will help make your documentation comments more concise. Below each public method declaration, explain what the method does with documentation comments. Then, explain each parameter, starting with its name and include any necessary information about the values the parameter has to receive. Do the same thing for the return parameter as well.

- ✓ Try adding a block documentation comment just below the CogObjectExample object's `ButtonBlinkTime` method, and verify that the documentation appears below the method declaration in Documentation view mode.

Figure 6: More Documentation View

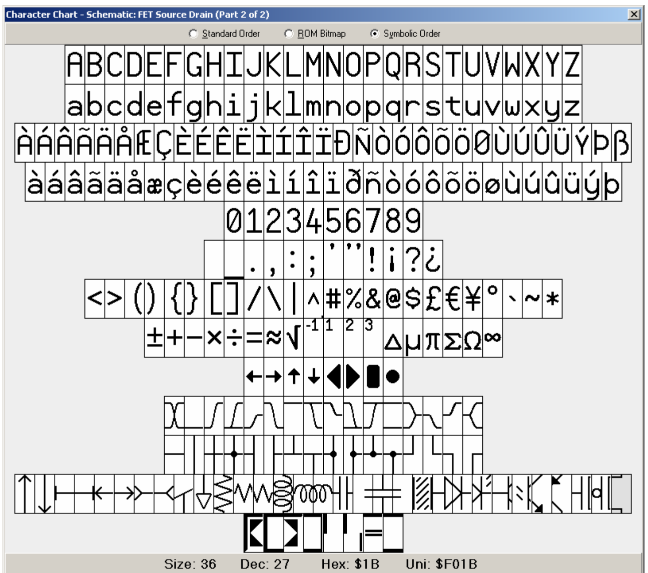


Drawing Schematics

The Parallax font has symbols built in for drawing schematics, and they should be used to document the circuits that objects are designed for. The Character Chart tool for inserting these characters into an object is shown in Figure 7. In addition to the symbols for drawing schematics, it has symbols for timing diagrams, math operators ± + − × ÷ = ≈ √ ⁻¹ ¹ ² ³,and Greek symbols for quantities and measurements Ω µ ∆ Σ π.

- ✓ Click *Help* and select *View Character Chart*.
- ✓ Click the character chart's symbolic *Order* button
- ✓ Place your cursor in a commented area of an object.
- ✓ Click various characters in the Character Chart, and verify that they appear in the object.

Figure 7: Propeller Tool Character Chart

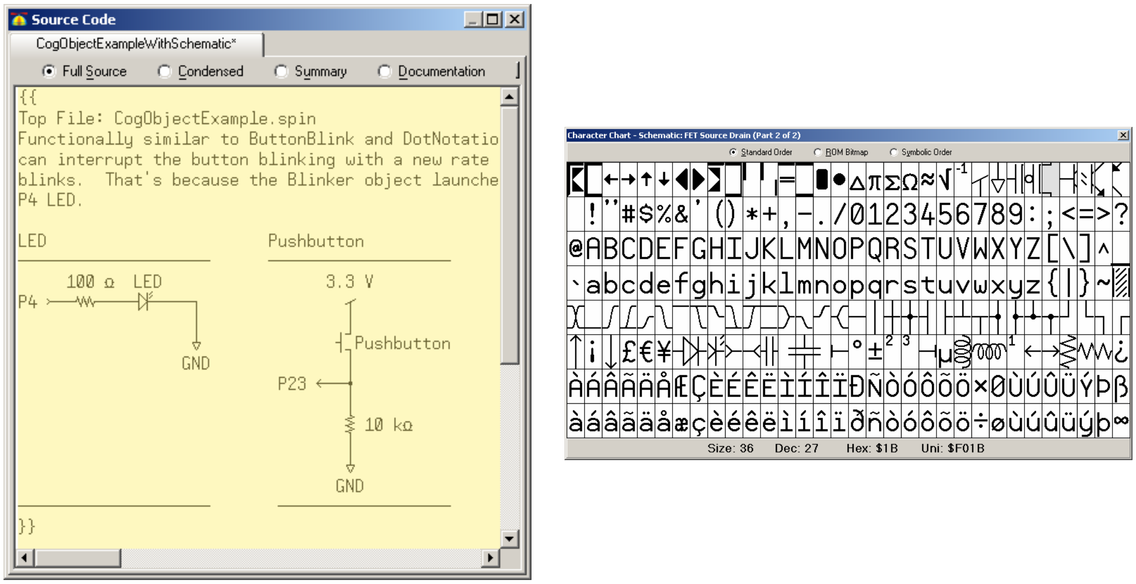


Files that involve circuits should also have schematics so that the circuit the code is written for can be built and tested. For example, the schematic shown in Figure 8 can be added to CogObjectExample. The pushbutton can be a little tricky. The character chart is shown in Figure 8, displayed in the standard order (click the Standard Order radio button). In this order, character 0 is the top left, character 1, the next one over from top-left, and so on, all the way down to character 255 on the bottom-right. Here is a list of characters you will need:

Pushbutton – 19, 23, 24, 27, 144, 145, 152, 186, 188
LED – 19, 24, 36, 144, 145, 158, 166, 168, 169, 189, 190

- ✓ Try adding the schematic shown in Figure 8 to your copy of CogObjectExample.

Figure 8: Drawing Schematics with the Character Chart



Public vs. Private methods

The Blinker object is currently written so that its parent object can call either its `Start` or `Blink` methods. For this particular object, it's useful because there are times when the programmer might not want to allow the 20 LED blinks to be interrupted. In that case, instead of calling the `Start` method, the parent object can call the `Blink` method directly.

✓ Modify a copy of CogObjectExample so that it calls the Blinker object's `Blink` method instead of its `Start` method.

The modified version will not let you interrupt the LED blinking to restart at a different rate. That's because all the code now gets executed in the same cog; whereas the unmodified version allows you to call the `Start` method at any time since the LED blinking happens in a separate cog. So while the cog is busy blinking the LED, it does not monitor the pushbutton.

Some objects are written so that they have public (`PUB`) methods that other objects can call, and private (`PRI`) methods, which can only be called from another method in the same object. Private methods tend to be ones that help the object do its job, but are not intended to be called by other objects. For example, sometimes an intricate task is separated into several methods. A public method might receive parameters and then call the private methods in a certain sequence. Especially if calling those methods in the wrong sequence could lead to undesirable results, those other methods should be private.

With the Blinker object's `Blink` method, there's no actual reason to make it private aside from examining what happens when a parent object tries to call another object's private method.

✓ Change the Blinker object's Blink method from `PUB` to `PRI`.

✓ Try to run the modified copy of CogObjectExample, and observe the error message. This demonstrates that the `Blink` method cannot now be accessed by another object since it's private.

- ✓ Run the unmodified copy (which only calls the public `Start` method, not the now private `Blink` method), and verify that it still works properly. This demonstrates how the now private `Blink` method can still be accessed from within the same (Blinker) object by its `Start` method.

Multiple Object Instances

Spin objects that launch and manage one or more cogs for a given process are typically written for just one copy of the process. If the application needs more than one copy of the process running concurrently, the application can simply declare more than one copy of the object. For example, the Propeller chip can control a television display with one cog, but each TV object only controls one television display. If the application needs to control more than one television, it declares more than one copy of the TV object.

> **Multiple object copies? No Problem!**
>
> There is no code space penalty for declaring multiple objects. The Propeller Tool's compiler optimizes so that only one instance of the code is executed by all the copies of the object. The only penalty for declaring more than one copy of the same object is that there will be more than one copy of the global variables the object declares, one set for each object. Since roughly the same number of extra variables would be required for a given application to do the same job without objects, it's not really a penalty.

The MultiCogObjectExample object below demonstrates how multiple copies of an object that manages a process can be launched with an object array. Like variables, objects can be declared as arrays. In this example, six copies of the Blinker object are declared in the `OBJ` block with `Blinker[6] : Blinker`. The six copies of Blinker can also be indexed the same way variable arrays are, with `Blinker[0]`, `Blinker[1]`, and so on, up through `Blinker[5]`. In MultiCogObjectExample, a **`repeat`** loop increments an `index` variable, so that `Blinker[index].Start`… calls each successive object's `Start` method.

The MultiCogObjectExample object is functionally equivalent to the Methods and Cogs lab's CogStartStopWithButton object. When the program is run, each successive press/release of the P23 pushbutton launches new cogs that blink successive LEDs (connected to P4 through P9) at rates determined by the duration of each button press. The first through sixth button presses launch new LED blinking processes into new cogs, and the seventh through twelfth presses successively stop each LED blinking cog in reverse order.

- ✓ Load the MultiCogObjectExample object into the Propeller chip.
- ✓ Press and hold the P23 pushbutton six successive times (each with a different duration) and verify that six cogs were launched.
- ✓ Press and release the P23 pushbutton six more times and verify that each LED blinking process halts in reverse order.

```
''Top File: MultiCogObjectExample.spin

OBJ

    Blinker[6] : "Blinker"
    Button     : "Button"


PUB ButtonBlinkTime | time, index

    repeat

       repeat index from 0 to 5
```

```
        time := Button.Time(23)
        Blinker[index].Start(index + 4, time, 1_000_000)

    repeat index from 5 to 0
        Button.Time(23)
        Blinker[index].Stop
```

Propeller Chip – PC Terminal Communication

Exchanging characters and values with the Propeller microcontroller using PC terminal software makes a number of applications really convenient. Some examples include computer monitored and controlled circuits, datalogging sensor measurements, and sending and receiving diagnostic information for system testing and debugging.

Terminal − Propeller Chip communication involves PC software and microcontroller code. For the PC software, we'll use the Parallax Serial Terminal, which is introduced next. For the microcontroller code, we'll make use of objects that take care of the electrical signaling and conversions between binary values and their character representations so that we can focus on writing applications.

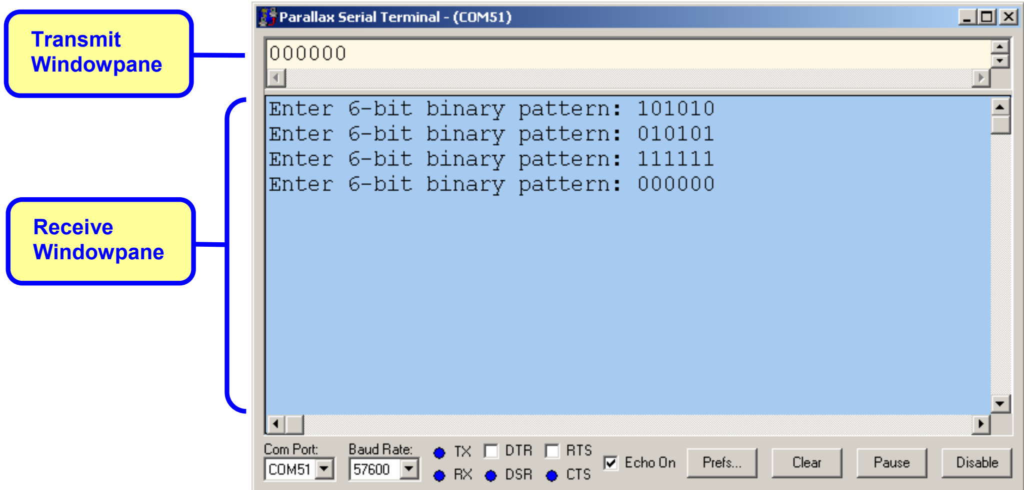
As you develop applications that make use of the serial communication objects, consider how those readily available objects simplify writing programs. It provides an example of how using objects from the Propeller Library, Propeller Object Exchange, and Propeller Chip forum make it possible to get a lot done with just a few lines of code.

Parallax Serial Terminal

The Parallax Serial Terminal software shown in Figure 9 is bundled with this lab. It's named PST.exe, and it's in the Parallax Serial Terminal subfolder. This software has a transmit windowpane that sends characters you type to the Propeller chip, and a receive windowpane that displays characters sent by the Propeller chip. It has dropdown menus for *Com Port* and *Baud Rate* selection and port activity indicators and checkbox controls for the various serial channels (*TX*, *RX*, etc). There's also an *Echo On* checkbox that is selected by default so that characters entered into the transmit windowpane also appear in the receive windowpane. On the Parallax Serial Terminal window's lower-right, it has control buttons that:

- Display and edit preferences (*Prefs*)
- (*Clear*) the terminal windows
- (*Pause*) the display of incoming data
- (*Disable/Enable*) the Parallax Serial Terminal's connection to the serial port

Figure 9: Parallax Serial Terminal



The *Disable/Enable* button in the Parallax Serial Terminal's lower-right corner is important. (See Figure 10.) When it displays *Disable*, it means the terminal is connected to the serial port. When you click the *Disable* button, the Parallax Serial Terminal releases the serial port so that the Propeller Tool can use it to load a program into the Propeller chip. While the Parallax Serial Terminal is disabled, the button displays *Enable*, flashing on/off. After the program has loaded, you can click the *Enable* button to resume terminal communication with the Propeller chip.

> **Automatic Disable/Enable Settings**
>
> In *Prefs -> Serial Port Selection*, the *Automatically disable…* and *Wait for busy…* checkboxes are selected by default. With these settings, you can just click the Propeller Tool software, load a program, and immediately click the *Enable* button to reconnect. There's no need to click *Disable* before switching to the Propeller Tool to load a program because the Parallax Serial Terminal will automatically disconnect from the serial port as soon as you have clicked another window. Likewise, you don't have to wait for the program to finish loading into the Propeller chip before clicking the *Enable* button. You can just click it as soon as you have started the program loading, and the Parallax Serial Terminal will detect that the serial port is still busy and wait until the Propeller Tool is done loading the program before it reconnects.

Figure 10: Connected vs. Disconnected (to/from the Com Port)

You can click the Parallax Serial Terminal's *Prefs* button to view the appearance and function preference tabs shown in Figure 11. The *Appearance* preferences allow you to define the terminal's colors, fonts, and other formatting. The *Function* preferences allow you to select special functions for non printable ASCII characters. Leave all of them checked for these labs since we'll be using them to clear the screen, display carriage returns, etc…

It's also best to leave both the boxes in the Serial Port Selection category checked. The *Automatically Disable…* feature makes the Parallax Serial Terminal automatically disable to free the serial port for program loading whenever you click the Propeller Tool software. The *Wait for busy port…* makes the Parallax Serial Terminal automatically wait up to 10 seconds if you click the *Enable* button before the Propeller tool is finished loading the program. (Not an issue with *Load RAM (F10)*, but *Load EEPROM (F11)* can take a few seconds.) If those features were unchecked, you would have to manually click *Disable* before loading a program and wait until the program is finished loading before clicking *Enable* to reconnect.

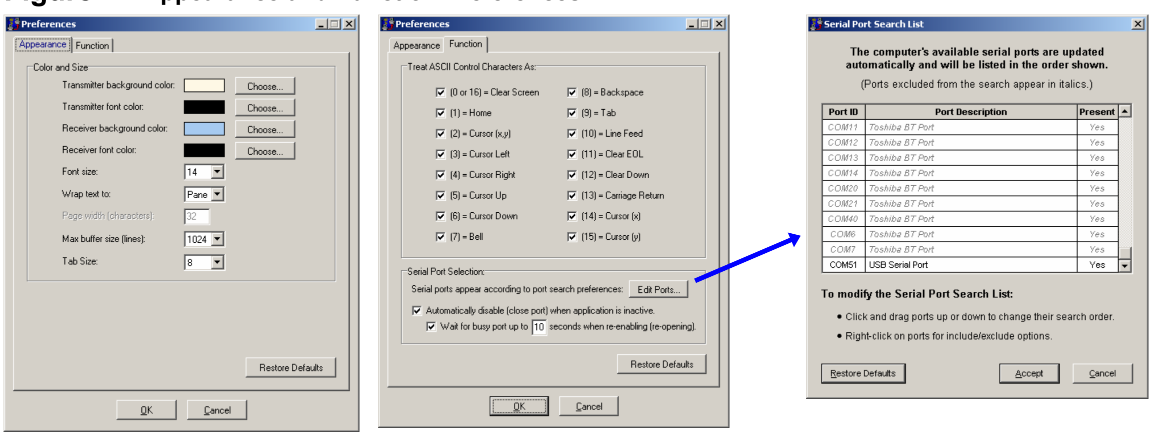
When to uncheck the *Automatically disable…* setting:

The *Automatically disable…* setting is very convenient for iteratively modifying code with the Propeller Tool software and observing the results in the Parallax Serial Terminal. The event that triggers the automatic *Disable* is the fact that you clicked another window.

Let's say you are instead switching back and forth between the Parallax Serial Terminal and some other software such as a spreadsheet for sensor measurement analysis. With the *Automatically disable…* setting, each time you click the other window, the Parallax Serial Terminal automatically disconnects from the serial port, and any messages sent by the Propeller chip will not be buffered or displayed.

To make the Parallax Serial Terminal maintain the serial port connection while you are working with other windows, uncheck the *Automatically disable…* setting. Then, the Parallax Serial Terminal will remain connected to the serial port and continue displaying updated messages, regardless of which window you are working in. Keep in mind that with this setting unchecked, you will have to manually click the *Disable* button before loading a program and then click the *Enable* button after the program is done loading.

Figure 11: Appearance and Function Preferences

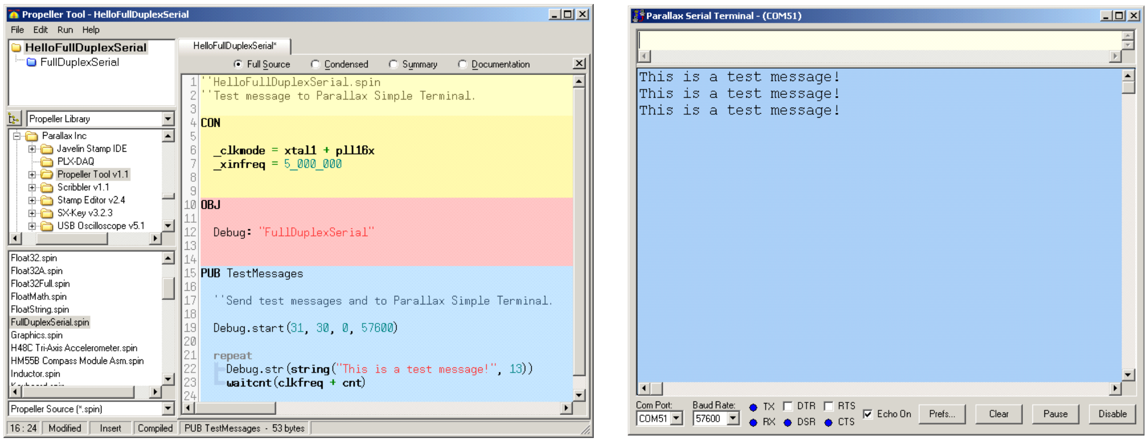


The Edit Ports button in Figure 11 opens the Serial Port Search List. You can drag entries in the list up and down to change the order they appear in the Parallax Serial Terminal's *Com Port* dropdown menu. You can also right-click an entry and to include or exclude it, or even create rules for which ports get included or excluded based on text in the Port Description column.

Parallax Serial Terminal Test Messages

Figure 12 shows the HelloFullDuplexSerial application on the left, and the repeated messages it sends to the Parallax Serial Terminal on the right. The HelloFullDuplexSerial program declares the FullDuplexSerial object and then uses its methods to send messages to the Parallax Serial Terminal. It first calls the FullDuplexSerial object's `start` method with `Debug.Start`, and then repeatedly calls the `str` (string) method with `Debug.str` in a `repeat` loop. Let's first give it a try, and then take a closer look at the FullDuplexSerial object and its features and methods.

Figure 12: Using the FullDuplexSerial object to Display Test Messages in Parallax Serial Terminal



The first time you open the Parallax Serial Terminal (PST.exe), you'll need to set the *Com Port* to the one the Propeller Tool software uses to load programs into the Propeller chip. You'll also need to set the *Baud Rate* to the one used by the Spin program. After that, just use the Propeller Tool software's Load EEPROM feature to load the program into the Propeller chip's EEPROM, and then click the Parallax Serial Terminal's *Enable* button to see the messages.

- ✓ Use Windows Explorer to open the Objects lab folder – PE-Lab-Objects-v1.1.
- ✓ Open HelloFullDuplexSerial.spin with the Propeller Tool software.
- ✓ Open the Parallax Serial Terminal subfolder, and double-click PST.exe to run it.
- ✓ Connect battery power to your PE Platform and verify that it is connected to the PC with the USB cable.
- ✓ In the Propeller Tool software, click *Run*, and select *Identify Hardware… (F7)*. Make a note of the COM port where the Propeller chip was found.
- ✓ Set the *Com Port* field in the bottom-left corner of the Parallax Serial Terminal to the Propeller's COM port you found in the previous step.
- ✓ Check the `baudrate` parameter in the `Debug.start` method call to find the baud rate. (It's currently 57600.)
- ✓ Set the baud rate field in the Parallax Serial Terminal to match. (Set it to 57600.)
- ✓ In the Propeller Tool software, use F11 to load HelloFullDuplexSerial.spin into the Propeller chip's EEPROM.
- ✓ In the Parallax Serial Terminal, click the *Enable* button to start displaying messages from the Propeller chip.

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

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug: "FullDuplexSerial"


PUB TestMessages

  ''Send test messages and to Parallax Serial Terminal.
```

```
Debug.start(31, 30, 0, 57600)

repeat
  Debug.str(string("This is a test message!", 13))
  waitcnt(clkfreq + cnt)
```

Changing Baud Rates

So long as the Baud rates are the same, you can select the baud rate that's best for your application. For example, you can change the baud rate from 57.6 to 115.2 kbps as follows:

- ✓ In the Propeller Tool, modify the HelloFullDuplexSerial object's `start` method call, so that it passes the value 115200 to the FullDuplexSerial object's `start` method's baudrate parameter, like this:

```
Debug.start(31, 30, 0, 115200)
```

- ✓ Load the modified version of HelloFullDuplexSerial into the Propeller chip.
- ✓ Choose 115200 in the Bits per second in the Parallax Serial Terminal's *Baud Rate* dropdown menu.
- ✓ Click Parallax Serial Terminal's *Enable* Button.
- ✓ Verify that the messages still display at the new baud rate.

- ✓ **Make sure to change the settings back to 57600 in both programs and test to make sure they still work before proceeding.**

FullDuplexSerial and Other Library Objects

The FullDuplexSerial object greatly simplifies exchanging data between the Propeller and peripheral devices that communicate with asynchronous serial protocols such as RS232. Just a few examples of serial devices that can be connected to the Propeller chip include the PC, other microcontrollers, phone modems, the Parallax Serial LCD, and the Pink Ethernet module.

> **Serial Communication:** For more information about asynchronous serial communication, see the *Serial Communication* and *RS-232* articles on Wikipedia.
>
> **Serial over USB:** For more information about how the FT232 chip built into the Propeller Plug and the PropsTick USB relays serial data to the PC over the USB connection, see the PropStick USB version of the Setup and Testing lab.

As mentioned earlier, code in an object can declare another object, so long as either:

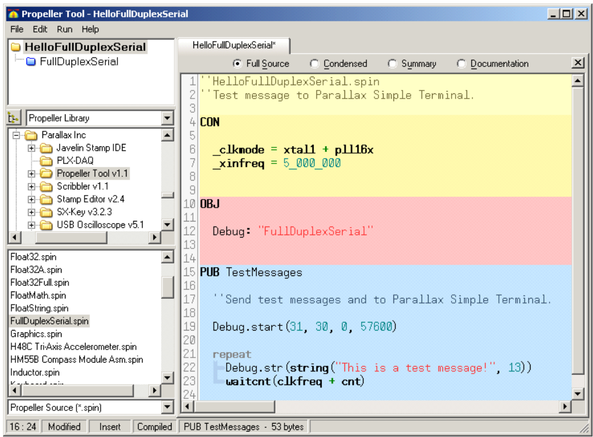
- The two objects are in the same folder
- The object being declared is in the same folder with the Propeller Tool software

The objects in the same folder with the Propeller Tool software are called Propeller Library objects. To view the contents of the Propeller Library:

- ✓ Click the dropdown menu between the upper-left and middle-left Explorer windowpanes shown in Figure 13 and select *Propeller Library*. The Propeller Library's objects will appear in the lower-left windowpane.

Notice in Figure 13 that the folder icon next to FullDuplexSerial in the Propeller Tool's upper left Object View windowpane is blue instead of yellow. This indicates that it's a file that resides in the Propeller Library. You can also see these files by using Windows Explorer to look in the Propeller Tool software's folder. Assuming a default install, the path would be: C:\Program Files\Parallax Inc\Propeller Tool v1.1.

Figure 13: Code That Declares a Library Object

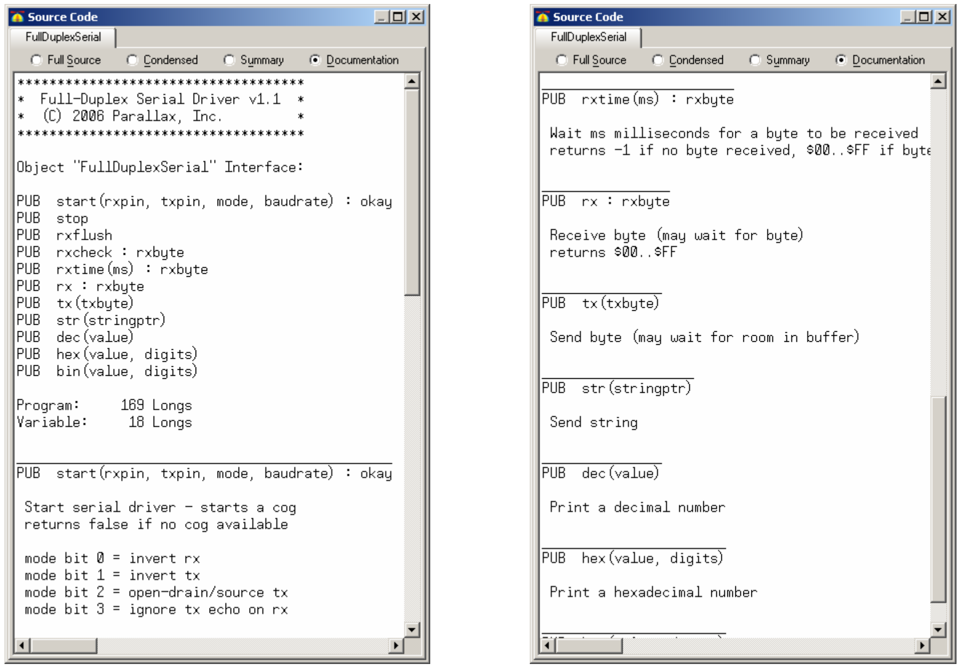


When using a library object, the first task is to examine its object interface to find out about its methods and what it can do.

- ✓ Double-click FullDuplexSerial in the Propeller Tool's lower left explorer pane, which should show the contents of the Propeller Library.
- ✓ When the Propeller Tool opens the FullDuplexSerial object, click the Documentation radio button so that the view resembles Figure 14.
- ✓ Check the list of methods in the Object "FullDuplexSerial" Interface section.
- ✓ Scroll down and find the documentation for the `start` and `str` methods, and examine them. They will be used in the next example object.

.

Figure 14: FullDuplexSerial Object Documentation Views



The HelloFullDuplexSerial object in Figure 13 declares the FullDuplexSerial object, giving it the nickname Debug. Then, it calls the FullDuplexSerial object's `start` method with the command `Debug.start(31, 30, 0, 57600)`. According to the documentation, this sets the parameter's `rxpin` to Propeller I/O pin 31, `txpin` to 30, `mode` to 0, and `baudrate` to 57600. After that, a **`repeat`** loop sends the same text message to the Parallax Serial Terminal once every second. The `Debug.str` method call is what transfers the `"This is a test message!"` string to the FullDuplexSerial object's buffer. After that, FullDuplexSerial takes care of sending each successive character in the string to the FT232 chip which forwards it to the PC via USB.

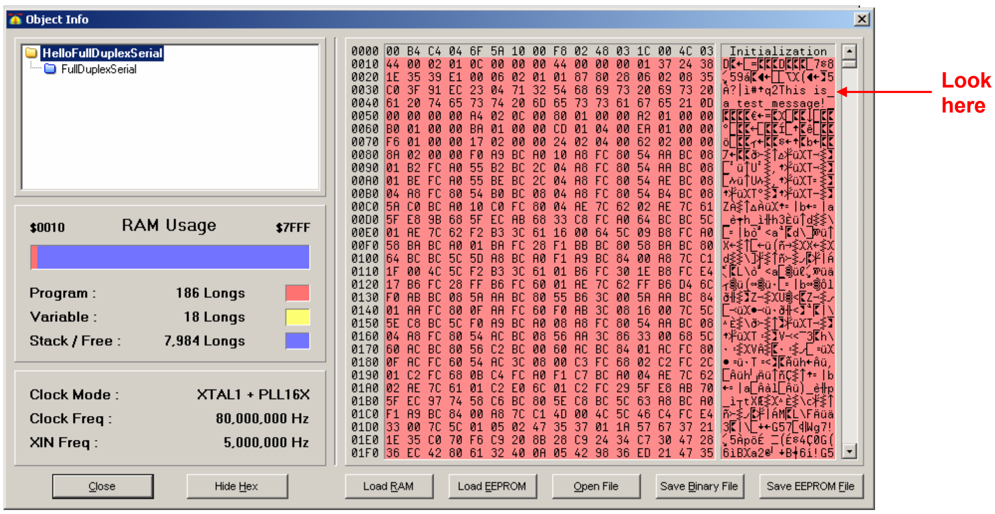
Let's take a closer look at `Debug.str(String("This is a test message!", 13))`. `Debug.str` calls the FullDuplexSerial object's `str` method. The method declaration for the `str` method indicates that the parameter it expects to receive should be a string pointer. At compile, the **`string`** directive **`string`**`("This is a test message!")` stores the values that correspond to the characters in the text message in the Propeller chip's program memory and appends them with a zero to make a zero-terminated string. Although the `str` method's documentation doesn't say so (It really should!), it expects a zero-terminated string so that it can fetch and transmit characters until it fetches a zero. At runtime, the **`string`** directive returns the starting address of the string. `Debug.str` passes this parameter to the FullDuplexSerial object's `str` method. Then, the `str` method sends characters until it fetches the zero terminator.

> ⓘ **What does the 13 do?** The 13 in `Debug.str(String("This is a test message!", 13))` is a control character that makes the Parallax Serial Terminal display a carriage return. That's why each "This is a text message!" appears on its own line, because the previous message was followed by a carriage return. See Figure 11 for the Parallax Serial Terminal's list of control characters.

You can see where the string gets stored in the program with the Propeller Tool Software's Object Info window.

- ✓ While viewing the HelloFullDuplexSerial object with the Propeller Tool, click *Run*, then point at *Compile Current*, and select *View info (F8)*. The Object Info window shown in Figure 15 should appear.
- ✓ Look for the text in the rightmost column's, 3rd and 4th lines. The hexadecimal ASCII codes occupy memory addresses 0038 through 004F with the 0 terminator at address 50.

Figure 15: Finding a Text String in Memory



Displaying Values

Take another look at the FullDuplexSerial object in documentation mode. (See Figure 14 on page 19.) Notice that it also has a `dec` method for displaying decimal numbers. This method takes a value and converts it to the characters that represent the value before transmitting them serially to the Parallax Serial Terminal. It's especially useful for displaying sensor readings and values stored by variables for figuring out program bugs.

- ✓ Modify the HelloFullDuplexSerial object's test messages declaration by adding a local variable declaration:

```
PUB TestMessages | counter
```

- ✓ Modify the the HelloFullDuplexSerial object's repeat loop as shown here:

```
repeat
  Debug.str(String(13, "counter = "))
  Debug.dec(counter++)
  waitcnt(clkfreq/5 + cnt)
```

- ✓ Use the Propeller Tool software to load the modified version of HelloFullDuplexSerial into the Propeller chip's EEPROM (F11).
- ✓ Click Parallax Serial Terminal's Enable button, and verify that the updated value of counter is displayed several times each second. You can press and release the PE Platform's Reset button to start the count at 0 again.

Sending Values from Parallax Serial Terminal to the Propeller Chip

The FullDuplexSerial object does not have a corresponding `GetDec` method to complement `dec`. So, as written, you cannot use FullDuplexSerial to receive a value from Parallax Serial Terminal. A modified version of FullDuplexSerial named FullDuplexSerialPlus is included with the .spin files that accompany this lab. The FullDuplexSerialPlus object has all the same methods as FullDuplexSerial, plus a few more, like `GetDec`, `GetBin`, and `GetHex`. The additional methods can be used to receive the character representations of decimal, hexadecimal and binary numbers from Parallax Serial Terminal, convert them to their corresponding numeric values, and store them in variables. Since FullDuplexSerialPlus also has the same methods as FullDuplexSerial, calls like `Debug.start`, `Debug.str`, and `Debug.dec` still yield the same results.

> **FullDuplexSerialPlus** is bundled with this lab, and a copy of the code is also in Appendix: FullDuplexSerialPlus.spin on page 44.

Remember that an object can be declared so long as it's either in the same folder with the object that's referencing it, or in the same folder as the Propeller Tool software. In this case, the FullDuplexSerialPlus object is in the same folder with this lab's example objects. So, it can be declared in a parent object's `OBJ` block almost same way FullDuplexSerial was. The only difference is that the parent object has to use the slightly different filename. So, instead of using a `Debug : FullDuplexSerial` declaration, use `Debug : FullDuplexSerialPlus`.

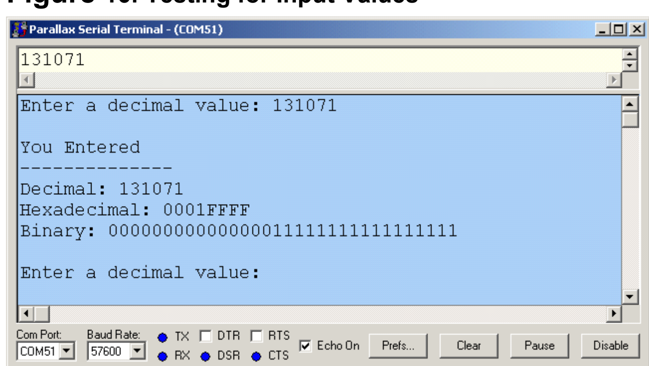
- ✓ Open both the FullDuplexSerial and FullDuplexSerialPlus objects in Documentation mode.
- ✓ Use the Object Interface section to see which methods have been added - there are 6, and the method names are capitalized.
- ✓ Check the documentation for the new methods. The documentation comments for the other methods were expanded too; look them over as well.

Test Application – EnterAndDisplayValues.spin

The EnterAndDisplayValues object below waits for you to enter a value into Parallax Serial Terminal's transmit windowpane. Then, it converts the characters that represent the value into a numeric equivalent and displays them in decimal, hexadecimal and binary format in Parallax Serial Terminal.

shows an example of testing the EnterAndDisplayValues object with Parallax Serial Terminal. The object makes the Propeller Chip send prompts that are displayed in Parallax Serial Terminal's receive windowpane. After typing a decimal value into the transmit windowpane and pressing enter, the Propeller chip converts the string of characters to its corresponding value, stores it in a variable, and then uses the FullDuplexSerialPlus object to send back the decimal, hexadecimal, and binary representations of the value.

Figure 16: Testing for Input Values



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

- ✓ The application gives you two seconds to connect Parallax Serial Terminal by clicking the *Enable* button. If no "Enter a decimal value:" prompt appears, you may not have clicked the *Enable* button in time. You can restart the application by pressing and releasing the PE Platform's reset button. You can also reset the Propeller chip from the terminal by checking and unchecking the DTR line.
- ✓ Follow the prompts in Parallax Serial Terminal. Start with 131071 and verify that it displays the values shown in .

The Propeller represents negative numbers with twos complement.

- ✓ Try entering these values: 4, 3, 2, 1, 0, -1, -2, -3, -4, -5, and discern the pattern of twos complement.

The Propeller chip's long variables store 32 bit signed integer values, ranging from -2,147,483,648 to 2,147,483,647.

- ✓ Try entering 2,147,483,645, 2,147,483,646, and 2,147,483,647 and examine the equivalent hexadecimal and binary values.
- ✓ Also try it with -2,147,483,646, -2,147,483,647, and -2,147,483,648.

```
'' File: EnterAndDisplayValues.spin
'' Messages to/from Propeller chip with Parallax Serial Terminal.
'' Prompts you to enter a value, and displays the value in decimal,
'' binary, and hexadecimal formats.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug: "FullDuplexSerialPlus"


PUB TwoWayCom | value

  ''Test Parallax Serial Terminal number entry and display.

  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*2 + cnt)
  Debug.tx(16)

  repeat

     Debug.Str(String("Enter a decimal value: "))
     value := Debug.getDec
     Debug.Str(String(13, "You Entered", 13, "--------------"))
     Debug.Str(String(13, "Decimal: "))
     Debug.Dec(value)
     Debug.Str(String(13, "Hexadecimal: "))
     Debug.Hex(value, 8)
     Debug.Str(String(13, "Binary: "))
     Debug.Bin(value, 32)
     repeat 2
        Debug.Str(String(13))
```

Debug.dec vs. Debug.getDec

The FullDuplexSerialPlus object's GetDec method buffers characters it receives form Parallax Serial Terminal until the enter key is pressed. Then, it converts the characters into their corresponding decimal value, and returns that value. The EnterAndDisplayValues object's command value := Debug.GetDec copies the result of the GetDec method call to the value variable. The command Debug.Dec(value) displays the value in decimal format. The command Debug.Hex(value, 8) displays the value in 8 character hexadecimal format, and the command Debug.Bin(value, 32) displays it in 32 character binary format.

Hex and Bin Character Counts

If you're sure you're only going to be displaying positive word or byte size variables, there's no reason to display all 32 bits of a binary value. Since word variables have 16 bits, and byte variables only have 8 bits, there's no reason to display 32 bits when examining those smaller variables.

- ✓ Make a copy of EnterAndDisplayValues and change the command Debug.Bin(value, 32) to Debug.Bin(value, 16).
- ✓ Remove the local variable | value from the TwoWayCom method declaration (remember that local variables are always 32 bits; whereas global variables can be declared long, word, or byte.)
- ✓ Add a VAR block to the object, declaring value as a word variable.
- ✓ Re-run the program, entering values that range from 0 to 65535.
- ✓ What happens if you enter 65536, 65537, and 65538? Try repeating this with the unmodified object, to see the missing bits.

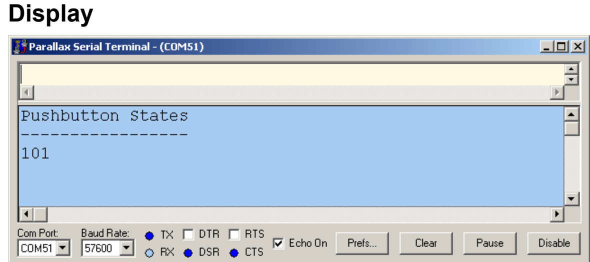
Each hexadecimal digit takes 4 bits. So, it will take 4 digits to display all possible values in a word variable (16-bits).

- ✓ Modify the copy of EnterAndDisplayValues so that it only displays 4 hexadecimal digits.

Terminal I/O Pin Input State Display

The Parallax Serial Terminal display provides a convenient means for testing sensors to make sure that both the program and wiring are correct. The DisplayPushbuttons object below displays the values stored in ina[23..21] in binary format as shown in Figure 17. A 1 in a particular slot indicates the pushbutton is pressed; a 0 indicates the pushbutton is not pressed. Figure 17 shows an example where the P23 and P21 pushbuttons are pressed.

Figure 17: Serial Terminal Pushbutton State Display



The DisplayPushbuttons object uses the command Debug.Bin(ina[23..21], 3) to display the pushbutton states. Recall from the I/O and Timing lab that ina[23..21] returns the value stored in bits 23 through 21 of the INA register. This result gets passed as a parameter to the FullDuplexSerialPlus object's bin method with the command Debug.bin(ina[23..21], 3). Note that since there are only 3 bits to display, the bin method's bits parameter is 3, which in turn makes the method display only 3 binary digits.

Since the FullDuplexSerialPlus object is running a serial driver in another cog, it is possible to transfer messages to it faster than the baud rate will allow it to send. The `waitcnt(clkfreq/100 + cnt)` command paces the updated values every 1/100 of a second to prevent buffer overflow.

- ✓ Use the Propeller Tool to load the DisplayPushbuttons object into EEPROM (F11), and immediately click the Parallax Serial Terminal's *Enable* button. Again, if you don't click it with 2 seconds after the download, just press the PE Platform's reset button to restart the program.
- ✓ Press and hold various combinations of the P23..P21 pushbuttons and verify that the display when they are pressed.

```
{{
DisplayPushbuttons.spin
Display pushbutton states with Parallax Serial Terminal.

Pushbuttons
──────────────────────────────────────────────────────────
     3.3 V                3.3 V                3.3 V
       ┬                    ┬                    ┬
       ┤ Pushbutton         ┤ Pushbutton         ┤ Pushbutton
       │                    │                    │
P21 ←──┫             P22 ←──┫             P23 ←──┫
       │                    │                    │
       ┋ 10 kΩ              ┋ 10 kΩ              ┋ 10 kΩ
       │                    │                    │
       ┴                    ┴                    ┴
      GND                  GND                  GND
──────────────────────────────────────────────────────────
}}

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

OBJ

  Debug: "FullDuplexSerialPlus"

PUB TerminalPushbuttonDisplay

  ''Read P23 through P21 pushbutton states and display with Parallax Serial Terminal.

  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*2 + cnt)
  Debug.tx(Debug#CLS)
  Debug.str(String("Pushbutton States", Debug#CR))
  Debug.str(String("-----------------", Debug#CR))

  repeat
    Debug.tx(Debug#CRSRX)
    Debug.tx(0)
    Debug.Bin(ina[23..21], 3)
    waitcnt(clkfreq/100 + cnt)
```

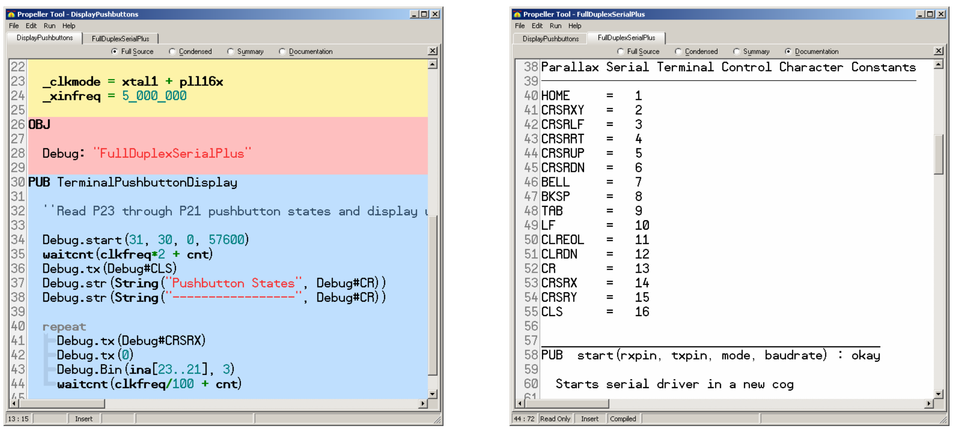
Accessing Constants in Objects with ObjectNickname#OBJECT_CONSTANT

You may have noticed that the expression `Debug#CR` replaced the number 13 for a carriage return. (See the left side of Figure 18.) That's because the constants for the Parallax Serial Terminal's control characters are declared in the FullDuplexSerialPlus object. You can see them in the

FullDuplexSerialPlus object documentation on the right side of Figure 18. Instead of using the numbers or declaring them a second time in the top level object, DisplayPushbuttons uses ObjectNickname#OBJECT_CONSTANT notation to specify control characters that get sent to Parallax Serial Terminal.

- ✓ Examine the FullDuplexSerialPlus object in both Full Source and Documentation mode.
- ✓ Make a note of how the constants are declared, and how they are documented with double-apostrophe ' ' comments.

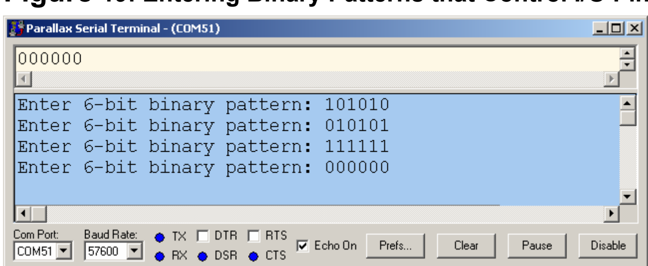
Figure 18: DisplayPushbuttons Full Source (left) and FullDuplexSerialPlus Documentation (right)



Terminal LED Output Control

Testing various actuators can also be important during prototyping. The TerminalLedControl object demonstrates a convenient means of setting output states for testing various output circuits. (See Figure 19.) While this example uses LED indicator lights, the I/O pin output signals could just as easily be sent to other chips' input pins, or inputs to circuits that control high current outputs such as solenoids, relays, DC motors, heaters, lamps, etc.

Figure 19: Entering Binary Patterns that Control I/O Pin Output States



The command `outa[9..4] := Debug.GetBin` calls the FullDuplexSerialPlus object's `GetBin` method. This method returns the value that corresponds to the binary characters (ones and zeros) you enter into the Parallax Serial Terminal's transmit windowpane. The value the `GetBin` method returns is assigned to `outa[9..4]`, which makes the corresponding LED pattern light.

- ✓ Use the Propeller Tool to Load TerminalLedControl into EEPROM (F11), and immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Try entering the values shown in Figure 19 into the transmit windowpane, and verify that the corresponding LED patterns light.

```
{{
TerminalLedControl.spin

Enter LED states into Parallax Serial Terminal.  Propeller chip receives the states and
lights the corresponding LEDs.

  LED SCHEMATIC
 ──────────────────────
      (all)
      100 Ω  LED
  P4 >──WW────|>──┐
                  │
  P5 >──WW────|>──┫
                  │
  P6 >──WW────|>──┫
                  │
  P7 >──WW────|>──┫
                  │
  P8 >──WW────|>──┫
                  │
  P9 >──WW────|>──┫
                  ▽
                 GND
 ──────────────────────
}}

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug : "FullDuplexSerialPlus"


PUB TerminalLedControl

  ''Set/clear I/O pin output states based binary patterns
  ''entered into Parallax Serial Terminal.

  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*2 + cnt)
  Debug.tx(Debug#CLS)
  dira[4..9]~~

  repeat

     Debug.Str(String("Enter 6-bit binary pattern: "))
     outa[4..9] := Debug.getBin
```

The DAT Block and Address Passing

One of the `DAT` block's uses is for storing sequences of values (including characters). Especially for longer messages and menu designs, keeping all the messages in a `DAT` block can be a lot more convenient than using `string("...")` in the code.

> **The `DAT` Block** can also be used to store assembly language code that gets launched into a cog. For an example, take a look at FullDuplexSerial in Full Source view mode. Assembly language techniques will be the subject of other labs.

Below is the DAT block from the next example object, TestMessages. Notice how each line has a label, a size, and a sequence of values (characters in this case).

```
DAT

  MyString        byte    "This is test message number: ", 0
  MyOtherString   byte    ", ", Debug#CR, "and this is another line of text.", 0
  BlankLine       byte    Debug#CR, Debug#CR, 0
```

Remember that the `string` directive returns the starting address of a string so that the FullDuplexSerial object's `str` method can start sending characters, and then stop when it encounters the zero termination character. With DAT blocks, the zero termination character has to be manually added. The name of a given DAT block directive makes it possible to pass the starting address of the sequence using the `@` operator. For example, `@MyString` returns the address of the first character in the `MyString` sequence. So, `Debug.str(@myString)` will start fetching and transmitting characters at the address of the first character in `MyString`, and will stop when it fetches the 0 that follows the "…`number: `" characters.

- ✓ Use the Propeller Tool to load the TestMessages object into EEPROM (F11), and then immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Verify that the three messages are displayed once every second.

```
'' TestMessages.spin
'' Send text messages stored in the DAT block to Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug: "FullDuplexSerialPlus"


PUB TestDatMessages | value, counter

  ''Send messates stored in the DAT block.

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

  repeat
     Debug.Str(@MyString)
     Debug.Dec(counter++)
     Debug.Str(@MyOtherString)
     Debug.Str(@BlankLine)
     waitcnt(clkfreq + cnt)

DAT

  MyString        byte    "This is test message number: ", 0
  MyOtherString   byte    ", ", Debug#CR, "and this is another line of text.", 0
  BlankLine       byte    Debug#CR, Debug#CR, 0
```

Expanding the DAT Section and Accessing its Elements

Here is a modified DAT section. The text messages have different content and different label names. In addition, there is a ValueList with long elements instead of byte elements.

```
DAT

  ValTxt      byte   Debug#CR, "The value is: ", 0
  ElNumTxt    byte   ", ", Debug#CR, "and it's element #: ", 0
  ValueList   long   98, 5282, 299_792_458, 254, 0
  BlankLine   byte   Debug#CR, 0
```

Individual elements in the list can be accessed with long, word, or byte. For example, long[@ValueList] would return the value 98, the first long. There's also an optional offset that can be added in a second bracket for accessing successive elements in the list. For example:

```
value := long[@ValueList][0]        ' copies 98 to the value variable
value := long[@ValueList][1]        ' copies 5282 to the value variable
value := long[@ValueList][2]        ' copies 299_792_458 to value
```

> **The long, word, and byte keywords have different uses in different types of blocks.**
>
> In VAR blocks, long, word and byte can be used to declare three different size variables. In DAT blocks, long, word, and byte can be used to declare the element size of lists. In PUB and PRI methods, long, word, and byte are used to retrieve values at certain addresses.

✓ Make a copy of the TestMessages object, and replace the DAT section with the one above. Replace the PUB section with the one shown below.

```
PUB TestDatMessages | value, index

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

  repeat
    repeat index from 0 to 4
       Debug.Str(@ValTxt)
       value := long[@valueList][index]
       Debug.Dec(value)
       Debug.Str(@ElNumTxt)
       Debug.Dec(index)
       Debug.Str(@BlankLine)
       waitcnt(clkfreq + cnt)
```

✓ Test the modified object with the Propeller chip and Parallax Serial Terminal. Note how an index variable is used in **long**[@ValueList][index] to return successive elements in the ValueList.

The Float and FloatString Objects

Floating-point is short for floating decimal point, and it refers to values that might contain a decimal point, preceded and/or followed by some number of digits. The IEEE754 single precision (32-bit) floating-point format is supported by the Propeller Tool software and by the Float and FloatString Propeller Library objects. This format uses a certain number of bits in a 32-bit variable for a number's significant digits, other bits to store the exponent, and a single bit to store the value's sign.

While calculations involving two single-precision floating-point values aren't as precise as those involving two 32-bit variables, it's great when you have fractional values to the right of the decimal point, including very large and small magnitude numbers. For example, while signed long variables can hold integers from -2,147,483,648 to 2,147,483,647, single-precision floating-point values can represent values as large as $\pm 3.403\times10^{38}$, or as small as $\pm 1.175\times10^{-38}$.

Another lab will delve further into floating-point mechanics and applications. For this lab, it's just important to know that the Propeller Library has objects that can be used to process floating-point values. TerminalFloatStringTest demonstrates some basic floating-point operations. First, `a := 1.5` and `b := pi` are using the Propeller Tool software's ability to recognize floating point values to pre-assign the floating-point version of 1.5 to the variable `a` and pi (3.141593) to `b`. Then, it uses the FloatMath object to add the floating-point values stored by the variables `a` and `b`. Finally, it uses the FloatString object to display the result, which gets stored in `c`.

- ✓ Use the Propeller Tool to load the FloatStringTest object into EEPROM (F11), and then immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Verify that the Parallax Serial Terminal's receive windowpane displays 1.5 + Pi = 4.641593.

```
''FloatStringTest.spin
''Solve a floating point math problem and display the result with Parallax Serial
''Terminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug   : "FullDuplexSerialPlus"
  fMath   : "FloatMath"
  fString : "FloatString"


PUB TwoWayCom | a, b, c

  '' Solve a floating point math problem and display the result.

  Debug.start(31, 30, 0, 57600)
  Waitcnt(clkfreq*2 + cnt)
  Debug.tx(Debug#CLS)

  a := 1.5
  b := pi

  c := fmath.FAdd(a, b)

  Debug.str(String("1.5 + Pi = "))

  debug.str(fstring.FloatToString(c))
```

Objects that Use Variable Addresses

Like elements in `DAT` blocks, variables also have addresses in RAM. Certain objects are designed to be started with variable address parameters. They often run in separate cogs, and either update their outputs based on a value stored in the parent object's variable(s) or update the parent object's variables based on measurements or incoming data, or both.

AddressBlinker is an example of an object that fetches values from its parent object's variables. Note that its `Start` method has parameters for two address values, `pinAddress` and `rateAddress`. The parent object has to pass the AddressBlinker object's `Start` method the address of a variable that stores the I/O pin number, and another that stores the rate. The `Start` method relays these parameters to the `Blink` method via the method call in the **`cognew`** command. So, when the `Blink` method gets launched into a new cog, it also receives copies of these addresses. Each time through the `Blink` method's **`repeat`** loop, it check's the values stored in its parent object's variables with `pin := long[rateAddress]` and `rate := long[rateAddress]`. Note that since the `pinAddress` and `rateAddress` already store addresses, the **`@`** operator is no longer needed.

- ✓ Examine the AddressBlinker object and pay careful attention to the variable interactions just discussed.

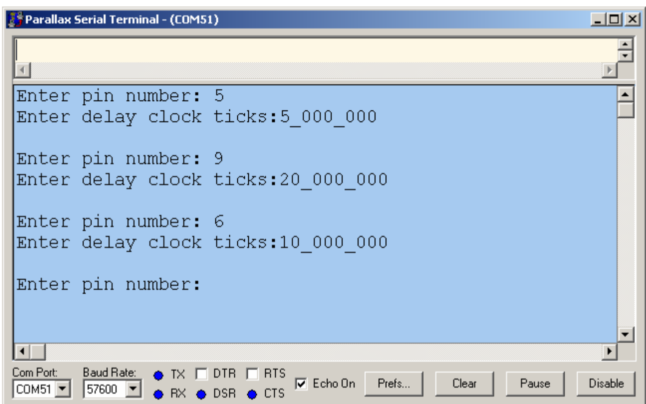
```
'' File: AddressBlinker.spin
'' Example cog manager that watches variables in its parent object


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


PUB Start(pinAddress, rateAddress) : success
''Start new blinking process in new cog; return True if successful.
''Parameters: pinAddress - long address of the variable that stores the I/O pin
''            rateAddress - long address of the variable that stores the rate

  Stop
  success := (cog := cognew(Blink(pinAddress, rateAddress), @stack) + 1)


PUB Stop
''Stop blinking process, if any.

  if Cog
    cogstop(Cog~ - 1)


PRI Blink(pinAddress, rateAddress) | pin, rate, pinOld, rateOld

    pin       :=   long[pinAddress]
    rate      :=   long[rateAddress]
    pinOld    :=   pin
    rateOld   :=   rate

    repeat
       pin := long[pinAddress]
       dira[pin]~~
       if pin <> pinOld
         dira[pinOld]~
       !outa[pin]
       pinOld := pin
       rate := long[rateAddress]
       waitcnt(rate/2 + cnt)
```

The AddressBlinkerControl object demonstrates one way of declaring variables, assigning their values, and passing their addresses to an object that will monitor them, the AddressBlinker object in this case. After it passes the addresses of its `pin` and `rateDelay` variables to AddressBlinker's `Start` method, the AddressBlinker object checks these variables between each LED state change. If the value of either `pin` or `rateDelay` has changed, AddressBlinker detects this and updates the LED's pin or blink rate accordingly.

- ✓ Use the Propeller Tool to load the AddressBlinkerControl object into EEPROM (F11), and then immediately click the Parallax Serial Terminal's *Enable* button.
- ✓ Enter the pin numbers and delay clock ticks shown in Figure 20 into the Parallax Serial Terminal's transmit windowpane, and verify that the application correctly selects the LED and determines its blink rate.

As soon as you press enter, the AddressBlinker object will update based on the new value stored in the AddressBlinkerControl object's `pin` or `rateDelay` variables.

Figure 20: Entering Pin and Rate into Serial Terminal




```
'' AddressBlinkerControl.spin
'' Enter LED states into Parallax Serial Terminal and send to Propeller chip via
'' Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug:    "FullDuplexSerialPlus"
  AddrBlnk: "AddressBlinker"


VAR

  long pin, rateDelay


PUB UpdateVariables

  '' Update variables that get watched by AddressBlinker object.

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

```
Debug.tx(Debug#CLS)

pin := 4
rateDelay := 10_000_000

AddrBlnk.start(@pin, @rateDelay)

dira[4..9]~~

repeat

   Debug.Str(String("Enter pin number: "))
   pin := Debug.getDec
   Debug.Str(String("Enter delay clock ticks:"))
   rateDelay := Debug.getDec
   Debug.Str(String(Debug#CR))
```

Displaying Addresses

In AddressBlinkerControl, the values of `pin` and `rateDelay` can be displayed with `Debug.Dec(pin)` and `Debug.Dec(rateDelay)`. The addresses of `pin` and `rateDelay` can be displayed with `Debug.Dec(@pin)` and `Debug.Dec(@rateDelay)`.

- ✓ Insert commands that display the addresses of the `pin` and `rateDelay` variables in Parallax Serial Terminal just before the repeat loop starts, and display the value of those variables each time they are entered. Note: The point of this exercise is to reinforce the distinction between a variable's contents and its address.

Passing Starting Addresses to Objects that Work with Variable Lists

Some objects monitor or update long lists of variables from another cog, in which case, they typically have documentation that explains the order and size of each variable that the parent object needs to declare. This kind of object's `Start` method typically just expects one value, the starting address of the list of variables in the parent object. The child object takes that one address and uses address offsets to access the rest of the variables in the parent object's list.

AddressBlinkerWithOffsets is an example of an object whose `start` method expects the starting address of a variable list. Unlike AddressBlinker, its `Start` method just receives the address of the parent object's long variable that stores the `pin` value. Its documentation requires the long variable storing the blink rate delay `to` be declared next, with no extra variables between.

Since the `baseAddress` parameter stores the address of the parent object's variable that stores the pin number, `long[baseAddress][0]` will access this value. Likewise, `long[baseAddress][1]` will access the variable that stores the blink rate. That's how this program fetches both variable values with just one address parameter.

- ✓ Examine the AddressBlinkerWithOffsets object. Note how its `start` method requires a baseAddress that it uses to find variables in its parent object that determine the pin and delay in the blink rate.
- ✓ Consider how this could be applied to longer lists of variables using address offsets.

```
'' File: AddressBlinkerWithOffsets.spin
'' Example cog manager that watches variables in its parent object
'' Parent object should declare a long that stores the LED I/O pin number
'' followed by a long that stores the number of click ticks between each
'' LED state change.  It should pass the address of the long that stores
'' the LED I/O pin number to the Start method.
```

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

PUB Start(baseAddress) : success
''Start new blinking process in new cog; return True if successful.
''
''baseAddress.......the address of the long variable that stores the LED pin number.
''baseAddress + 1...the address of the long variable that stores the blink rate delay.

  Stop
  success := (cog := cognew(Blink(baseAddress), @stack) + 1)


PUB Stop
''Stop blinking process, if any.

  if Cog
    cogstop(Cog~ - 1)

PRI Blink(baseAddress) | pin, rate, pinOld, rateOld

    pin       :=   long[baseAddress][0]
    rate      :=   long[baseAddress][1]
    pinOld    :=   pin
    rateOld   :=   rate

    repeat
       pin := long[baseAddress][0]
       dira[pin]~~
       if pin <> pinOld
         dira[pinOld]~
       !outa[pin]
       pinOld := pin
       rate := long[baseAddress][1]
       waitcnt(rate/2 + cnt)
```

Keep in mind that the point of this example is to demonstrate how a parent object can pass a base address to a child object whose documentation requires a list of variables of specified sizes that hold certain values and are declared in a certain order. The AddressBlinkerControlWithOffsets object works with the AddressBlinkerWithOffsets object in this way to perform the same application featured in the previous example, terminal controlled LED selection and blink rate. In keeping with the AddressBlinkerWithOffsets object's documentation, AddressBlinkerControlWithOffsets declares a long variable to store `pin`, and the next long variable it declares is `rateDelay`. Then, it passes the address of its `pin` variable to the AddressBlinkerControl object's `Start` method.

In this object, the variable declaration **`long`** `pin, rateDelay` is crucial. If the order of these two variables were swapped, the application wouldn't work right. Again, that's because the AddressBlinkerWithOffsets object expects to receive the address of a long variable that stores the `pin` value, and it expects the next consecutive long variable to store the `rateDelay` variable. Now, it's perfectly fine to declare long variables before and after these two. It's just that `pin` and `rateDelay` have to be long variables, and they have to be declared in the order specified by AddressBlinkerWithOffsets. The starting address of the variable list also has to get passed to the child object's start method, in this case with `AddrBlnk.start(@pin)`. Keep an eye open for this approach in objects that are designed to work with long lists of variables in their parent objects.

- ✓ Test AddressBlinkerControlWtihOffsets and verify that it is functionally identical to AddressBlinkerControl.

✓ Examine how AddressBlinkerControlWithOffsets is designed in accordance with the AddressBlinkerWithOffsets object's documentation.

```
'' File: AddressBlinkerControlWithOffsets.spin
''
'' Another example cog manager that relies on an object that watches variables in its
'' parent object.
''
'' This one's start method only passes one variable address, but uses it as an anchor
'' for two variables that are monitored by AddressBlinkerWithOffsets.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

VAR

  long pin, rateDelay

OBJ

  Debug:    "FullDuplexSerialPlus"
  AddrBlnk: "AddressBlinkerWithOffsets"


PUB TwoWayCom

  ''Send test messages and values to Parallax Serial Terminal.

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

  pin := 4
  rateDelay := 10_000_000

  AddrBlnk.start(@pin)

  dira[4..9]~~

  repeat

     Debug.Str(String("Enter pin number: "))
     pin := Debug.getDec
     Debug.Str(String("Enter delay for 'rate':"))
     rateDelay := Debug.getDec
     Debug.tx(Debug#CR)
```

Questions

1) What are the differences between calling a method in the same object and calling a method in another object?
2) Does calling a method in another object affect the way parameters and return values are passed?
3) What file location requirements have to be satisfied before one object can successfully declare another object?
4) Where can object hierarchy in your application be viewed?
5) How are documentation comments included in an object?
6) How do you view an object's documentation comments while filtering out code?

7) By convention, what method names do objects use for launching methods into new cogs and shutting down cogs?
8) What if an object manages one process in one new cog, but you want more than one instance of that process launched in multiple cogs?
9) What is the net effect of an object's `Start` method calling its `Stop` method?
10) How are custom characters for schematics, measurements, mathematical expressions and timing diagrams entered into object comments?
11) What's are the differences between a public and private method?
12) How do you declare multiple copies of the same object?
13) Where are Propeller Library objects stored?
14) How do you view Object Interface information
15) Where in RAM usage does the `String` directive cause character codes to be stored?
16) Why are zero-terminated strings important for the FullDuplexSerial object?
17) What should an object's documentation comments explain about a method?
18) How can character strings be stored, other than with the `String` declaration?
19) What are the three different uses of the `long`, `word` and `byte` keywords in the Spin language?
20) What method does the Float object use to add two floating-point numbers?
21) What object's methods can be used to display floating-point numbers as strings of characters?
22) Is the command `a := 1.5` processed by the `FloatMath` object?
23) How does a variable's address get passed to an object method's parameter?
24) How can passing an address to an object's method reduce the number of parameters required
25) Given a variable's address, how does an object's method access values stored in that variable and variables declared after it?
26) Given an address, can an object monitor a variable value?
27) Given an address, can an object update the variable in another object using that address.

Exercises

1) Given the file MyLedObject.spin, write a declaration for another object in the same folder so that it can use its methods. Use the nickname `led`.
2) Write a command that calls a method named `on` in an object nicknamed `led`. This method requires a `pin` parameter (use 4).
3) List the decimal values of the Parallax Font characters required to write this expression in a documentation comment $f = T^{-1}$.
4) Declare a private method named `calcArea` that accepts parameters `height` and `width`, and returns `area`.
5) Declare five copies of an object named FullDuplexSerial (which could be used for five simultaneous serial communication bidirectional serial connections). Use the nickname `uart`.
6) Call the third FullDuplexSerial object's `str` method, and send the string "Hello!!!". Assume the nickname `uart`.
7) Write a `DAT` block and include a string labeled `Hi` with the zero terminated string "Hello!!!".
8) Write a command that calculates the circumference (c) of a circle given the diameter (d). Assume the FloatMath object has been nicknamed `f`.
9) Given the variable `c`, which stores a floating-point value, pass this to a method in FloatString that returns the address of a stored string representation of the floating point value. Store this address in a variable named `address`. Assume the nickname `fst`.

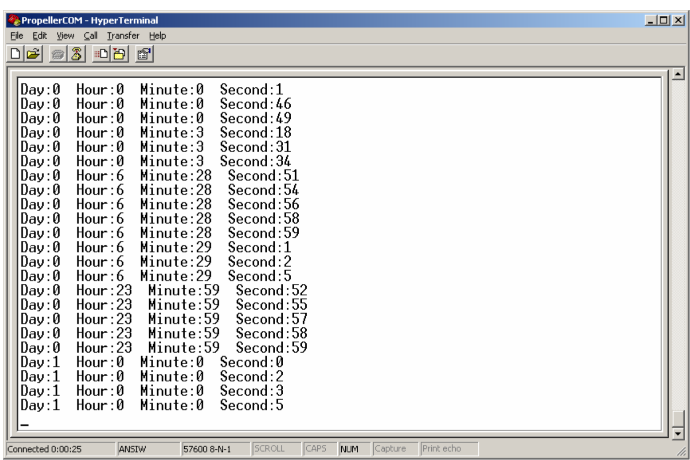
Projects

1) The TestBs2IoLiteObject uses method calls that are similar to the BASIC Stamp microcontroller's PBASIC programming language commands. This object needs a Bs2IoLite object with methods like `high`, `pause`, `low`, `in`, and `toggle`. Write an object that supports these method calls using the descriptions in the comments.

```
''Top File: TestBs2IoLiteObject.spin
''Turn P6 LED on for 1 s, then flash P5 LED at 5 Hz whenever the
''P21 pushbutton is held down.

OBJ

    stamp : "Bs2IoLite"


PUB ButtonBlinkTime | time, index

    stamp.high(6)             ' Set P6 to output-high
    stamp.pause(1000)         ' Delay 1 s
    stamp.low(6)              ' Set P6 to output-low
    stamp.low(5)              ' Set P5 to output-low
    repeat                    ' Repeat (like DO...LOOP in PBASIC)
      if stamp.in(21)         ' If P21 pushbutton pressed
        stamp.toggle(5)       ' Toggle P5 output state
      else
        stamp.low(5)
      stamp.pause(100)        ' Delay 0.1 s before repeat
```

2) Examine the Stack Length object in the Propeller Library, and the Stack Length Demo in the Propeller Library Demo folders. Make a copy of Stack Length Demo.spin, and modify it to test the stack space required for launching the Blinker object's `Blink` method (from the beginning of this lab) into a cog. Create a Parallax Serial Terminal connection based on StackLenthDemo's documentation to display the result. *NOTE: The instructions for using the Stack Length object are hidden in its THEORY OF OPERATION comments, which are visible in documentation view mode.*

3) Some applications will have a clock running in a cog for timekeeping. Below is a terminal display that gets updated each time the PE Platform's P23 pushbutton is pressed and released.



The Parallax Serial Terminal gets updated by the TerminalButtonLogger object below. There are two calls to the TickTock object. The first is call is `Time.Start(0, 0, 0, 0)`, which initializes the TickTock object's `day`, `hour`, `minute`, and `second` variables. The second method call is `Time.Get(@days, @hours, @minutes, @seconds)`. This method call passes the TickTock object the addresses of the TerminalButtonLogger object's `days`, `hours`, `minutes`, and `seconds` variables. The TickTock object updates these variables with the current time.

Your task in this project is to write the TickTock object that works with the TerminalButtonLogger object. Make sure to use the second counting technique from the `GoodTimeCount` method from the I/O and Timing lab.

```
'' TerminalButtonLogger.spin
'' Log times the button connected to P23 was pressed/released in
'' Parallax Serial Terminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug      : "FullDuplexSerialPlus"
  Button     : "Button"
  Time       : "TickTock"


VAR

  long days, hours, minutes, seconds


PUB TestDatMessages

  Debug.start(31, 30, 0, 57600)          ' Start FullDuplexSerialPlus object.
  waitcnt(clkfreq*3 + cnt)               ' Wait for three seconds.
  Debug.tx(Debug#CLS)


  Time.Start(0, 0, 0, 0)                 ' Start the TickTock object and initialize
                                         ' the day, hour, minute, and second.
  Debug.Str(@BtnPrompt)                  ' Display instructions in Parallax Serial
Terminal
  repeat

    if Button.Time(23)                   ' If button pressed.
       ' Pass variables to TickTock object for update.
       Time.Get(@days, @hours, @minutes, @seconds)
       DisplayTime                       ' Display the current time.


PUB DisplayTime

      Debug.tx(Debug#CR)
      Debug.Str(String("Day:"))
      Debug.Dec(days)
      Debug.Str(String("  Hour:"))
      Debug.Dec(hours)
      Debug.Str(String("  Minute:"))
      Debug.Dec(minutes)
      Debug.Str(String("  Second:"))
      Debug.Dec(seconds)


DAT

BtnPrompt   byte    Debug#CLS, "Press/release P23 pushbutton periodically...", 0
```

Question Solutions

1) A method call in the same object just uses the method's name. A call to a method in another object uses a nickname that was given to the object in `OBJ` block, then a dot, then the method's name. So the difference is instead of just using *MethodName*, it's *ObjectNickname.MethodName*.
2) No. Parameters are passed and returned the same way they would in a method in the same object.
3) The object that's getting declared has to either be in the same folder with the object that's declaring it, or in the same folder with the Propeller Tool software.
4) In the Object View pane, which can be viewed in the Object Info window (F8), and also in the upper-left corner of the Propeller Tool software's Explorer pane.
5) Two apostrophes can be placed to the left of a comment that should appear in the Propeller Tool software's documentation view. A block of documentation text can be defined with double-braces {{documentation comments}}.
6) By clicking the Documentation radio button above the code.
7) Method names `Start` and `Stop`.
8) Declare multiple copies of the object in the `OBJ` section, and call each of their `Start` methods.
9) If the process the object manages is already running in another cog, the call to the `Stop` method shuts it down before launching the process into a new cog.
10) By clicking on characters in the Propeller Tool Character Chart.
11) Public methods are declared with `PUB`, private with `PRI`. Public methods can be called by commands in other objects; private methods can only be called from within the same object.
12) Declare multiple copies of the same object by declaring an object array. For example, the command `nickname[3] : ObjectName` declares three copies of `ObjectName`, `nickname[0]`, `nickname[1]`, and `nickname[2]`. Note that it doesn't actually make extra copies of the object code. Each instance still uses the same copy of the Spin code that is loaded into the Propeller chip.
13) They are stored in the same folder with the Propeller Tool software .exe file.
14) To view the Object Interface information, click the *Documentation* radio button, and the Propeller Tool software automatically generates that information and displays it along with the documentation comments.
15) In the Program codes.
16) Given a start address in RAM, the FullDuplexSerial object's `Str` method fetches and transmits characters until it fetches a zero.
17) Documentation comments should explain what the method does, its parameters (if any) and its return value.
18) Character strings and other lists of values can be stored in an object's `DAT` section.
19) They are used to (1) declare variables in `VAR` blocks, (2) declare list element sizes in `DAT` blocks, and (3) return values stored at given addresses within `PUB` and `PRI` blocks.
20) The Float object uses `FAdd` to add two floating-point numbers.
21) What object's methods can be used to display floating-point numbers as strings of characters? FloatString.
22) No, the Propeller Tool packs 1.5 into floating-point format at compile time and stores it with the program byte codes. The command `a := 1.5` copies the value into a variable.
23) A variable's address get passed to an object method's parameter with the `@` operator. Instead of this format: *ObjectNickname.MethodName(variableName)*, use the following format: *ObjectNickname.MethodName(@variableName)*.
24) An object can declare a list of variables in a certain order, and then assign them each values that the object will use. Then, the address of the first variable in the list can be passed to the object's method.

25) The object will use either `long`, `word` or `byte` and the address. For example, if the address is passed to a parameter named `address`, the object can access the value stored by the variable with `long[address][0]` or just `long[address]`. To store the variable declared immediately to the right of the variable at `address`, `long[address][1]` can be used. For the second variable to the right, `long[address][2]` can be used, and so on.
26) Yes. This can be useful at times, because the parent object can simply update a variable value, and an object running another process will automatically update based on that value.
27) Yes. This comes in handy when a process is running in another cog, and the parent object needs one or more of its variables to be automatically updated by the other process.

Exercise Solutions

1) Solution:
```
led : "MyLedObject"
```
2) Solution:
```
led.On(4)
```
3) With the aid of the Propeller Tool software's Character Chart: 102, 32, 61, 32, 84, 22.
4) Solution:
```
PRI calcArea(height, width) : area
```
5) Solution:
```
Uart[5] : "FullDuplexSerial"
```
6) Solution:
```
uart[2].str(String("Hello!!!"))
```
7) Solution:
```
DAT
  Hi byte "Hello!!!", 0
```
8) Solution:
```
c := f.fmul(d, pi)
```
9) Solution:
```
address := fst(c)
```

Project Solutions

1) Example Object:

```
{{
Bs2IoLite.spin

This object features method calls similar to the PBASIC commands for the BASIC
Stamp
2 microcontroller, such as high, low, in0 through in15, toggle, and pause.

}}

PUB high(pin)
''Make pin output-high.

    outa[pin]~~
    dira[pin]~~
```

```
PUB low(pin)
''Make pin output-low

    outa[pin]~
    dira[pin]~~


PUB in(pin) : state
{{Return the state of pin.
If pin is an output, state reflects the
output signal.  If pin is an input, state will be 1 if the voltage
applied to pin is above 1.65 V, or 0 if it is below.}}

    state := ina[pin]


PUB toggle(pin)
''Change pin's output state (high to low or low to high).

    !outa[pin]

PUB pause(ms) | time
''Make the program pause for a certain number of ms.  This applies to
''the cog making the call.  Other cogs will not be affected.

    time := ms * (clkfreq/1000)
    waitcnt(time + cnt)
```

2) For modifying Parallax Serial Terminal, save a copy of PropellerCOM under a new name, such as TestPropellerStack.ht. Change the Parallax Serial Terminal's *Baud Rate* from 57600 to 19200.

   The modified Stack Length Demo object below has several changes. The code below the Code/Object Being Tested for Stack Usage heading was replaced with the Blinker object code. The Blinker object's stack was variable array was increased to 32 longs. Then, in the Temporary Code to Test Stack Usage section, the start method call was modified to work with the Blinker object.

   Run the modified Stack Length Demo object below to test the stack required by the Blink method for launching into another cog. After the Propeller Tool has completed its download, you will have 2 seconds to connect Parallax Serial Terminal. The result should be 9.

   Since the result is 9 instead of 10 predicted by the Methods lab, this project exposes an error in the Methods lab's section entitled: ""How Much Stack Space for a Method Launched into a Cog? " The `time` local variable was removed from the `Blink` method, but not from the discussion of how much stack space the `Blink` method requires.

```
{{
StackLengthDemoModified.spin

This is a modified version of Stack Length Demo object from the Propeller Library
Demos folder.  This modified version tests the Propeller Education Kit Objects
lab's Blinker object's Blink method for stack space requirements.  See Project #2
in the Objects lab for more information.
}}

{••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
  Temporary Code to Test Stack Usage
••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••}
```

```
CON
  _clkmode      = xtal1 + pll16x          'Use crystal * 16 for fast serial
  _xinfreq      = 5_000_000               'External 5 MHz crystal on XI & XO

OBJ
  Stk   :       "Stack Length" 'Include Stack Length Object

PUB TestStack
  Stk.Init(@Stack, 32)          'Initialize reserved Stack space (reserved below)
  start(4, clkfreq/10, 20)      'Exercise code/object under test
  waitcnt(clkfreq * 3 + cnt)    'Wait ample time for max stack usage
  Stk.GetLength(30, 19200)      'Transmit results serially out P30 at 19,200 baud


{••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••
Code/Object Being Tested for Stack Usage
••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••••}

{{
File: Blinker.spin
Example cog manager for a blinking LED process.

SCHEMATIC
───────────────────────────────
          100 Ω  LED
    pin >──/\/\/──|>|──┐
                       ↓
                      GND
───────────────────────────────
}}

VAR
  long  stack[32]                       'Cog stack space
  byte  cog                             'Cog ID


PUB Start(pin, rate, reps) : success
{{Start new blinking process in new cog; return True if successful.

Parameters:
  pin - the I/O connected to the LED circuit → see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles
}}
  Stop
  success := (cog := cognew(Blink(pin, rate, reps), @stack) + 1)


PUB Stop
''Stop blinking process, if any.

  if Cog
    cogstop(Cog~ - 1)


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

Parameters:
  pin - the I/O connected to the LED circuit → see schematic
  rate - On/off cycle time is defined by the number of clock ticks
  reps - the number of on/off cycles
}}

    dira[pin]~~
    outa[pin]~
```

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

3) This solution uses global variables for `days`, `hours`, `minutes`, and `seconds`, and the `GoodTimeCount` method updates all four values. It would also be possible to just track seconds, and use other methods to convert to days, hours, etc.

```
''File: TickTock.spin

VAR

  long stack[50]
  byte cog
  long days, hours, minutes, seconds


PUB Start(setDay, setHour, setMinutes, setSeconds) : success
{{
Track time in another cog.

  Parameters - starting values for:
    setDay     - day
    setHour    - hour
    setMinutes - minute
    setSeconds - second
}}

  days := setDay
  hours := setHour
  minutes := setMinutes
  seconds := setSeconds

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


PUB Stop
''Stop counting time.

  if Cog
    cogstop(Cog~ - 1)


PUB Get(dayAddr, hourAddr, minAddr, secAddr) | time
{{
Get the current time.  Values are loaded into variables at the
addresses provided to the method parameters.

  Parameters:
    dayAddr  -  day variable address
    hourAddr -  hour variable address
    minAddr  -  minute variable address
    secAddr  -  secondAddress
}}

  long[dayAddr]  := days
  long[hourAddr] := hours
  long[minAddr]  := minutes
  long[secAddr]  := seconds


PRI GoodTimeCount | dT, T
```

```
dT := clkfreq
T  := cnt

repeat

  T += dT
  waitcnt(T)
  seconds ++

  if seconds == 60
     seconds~
     minutes++
  if minutes == 60
     minutes~
     hours++
  if hours == 24
     hours~
     days++
```

Appendix: FullDuplexSerialPlus.spin

```
'' From Parallax Inc. Propeller Education Kit - Objects Lab v1.1
{{
────────────────────────────────────────────────────────────────────────────────
File: FullDuplexSerialPlus.spin
Version: 1.1
Copyright (c) 2008 Parallax, Inc.
See end of file for terms of use.

This is the FullDuplexSerial object v1.1 from the Propeller Tool's Library
folder with modified documentation and methods for converting text strings
into numeric values in several bases.

────────────────────────────────────────────────────────────────────────────────
}}

CON                                             ''
''Parallax Serial Terminal Control Character Constants
''────────────────────────────────────────────────────
  HOME     =   1                                ''HOME     =   1
  CRSRXY   =   2                                ''CRSRXY   =   2
  CRSRLF   =   3                                ''CRSRLF   =   3
  CRSRRT   =   4                                ''CRSRRT   =   4
  CRSRUP   =   5                                ''CRSRUP   =   5
  CRSRDN   =   6                                ''CRSRDN   =   6
  BELL     =   7                                ''BELL     =   7
  BKSP     =   8                                ''BKSP     =   8
  TAB      =   9                                ''TAB      =   9
  LF       =   10                               ''LF       =   10
  CLREOL   =   11                               ''CLREOL   =   11
  CLRDN    =   12                               ''CLRDN    =   12
  CR       =   13                               ''CR       =   13
  CRSRX    =   14                               ''CRSRX    =   14
  CRSRY    =   15                               ''CRSRY    =   15
  CLS      =   16                               ''CLS      =   16


VAR

  long  cog                     'cog flag/id

  long  rx_head                 '9 contiguous longs
  long  rx_tail
  long  tx_head
  long  tx_tail
  long  rx_pin
  long  tx_pin
  long  rxtx_mode
  long  bit_ticks
  long  buffer_ptr

  byte  rx_buffer[16]           'transmit and receive buffers
  byte  tx_buffer[16]


PUB start(rxpin, txpin, mode, baudrate) : okay
  {{
  Starts serial driver in a new cog

    rxpin - input receives signals from peripheral's TX pin
    txpin - output sends signals to peripheral's RX pin
    mode  - bits in this variable configure signaling
```

```
               bit 0 inverts rx
               bit 1 inverts tx
               bit 2 open drain/source tx
               bit 3 ignor tx echo on rx
    baudrate - bits per second

    okay - returns false if no cog is available.
  }}

  stop
  longfill(@rx_head, 0, 4)
  longmove(@rx_pin, @rxpin, 3)
  bit_ticks := clkfreq / baudrate
  buffer_ptr := @rx_buffer
  okay := cog := cognew(@entry, @rx_head) + 1


PUB stop

  '' Stops serial driver - frees a cog

  if cog
    cogstop(cog~ - 1)
  longfill(@rx_head, 0, 9)


PUB tx(txbyte)

  '' Sends byte (may wait for room in buffer)

  repeat until (tx_tail <> (tx_head + 1) & $F)
  tx_buffer[tx_head] := txbyte
  tx_head := (tx_head + 1) & $F

  if rxtx_mode & %1000
    rx

PUB rx : rxbyte

  '' Receives byte (may wait for byte)
  '' rxbyte returns $00..$FF

  repeat while (rxbyte := rxcheck) < 0

PUB rxflush

  '' Flush receive buffer

  repeat while rxcheck => 0

PUB rxcheck : rxbyte

  '' Check if byte received (never waits)
  '' rxbyte returns -1 if no byte received, $00..$FF if byte

  rxbyte--
  if rx_tail <> rx_head
    rxbyte := rx_buffer[rx_tail]
    rx_tail := (rx_tail + 1) & $F

PUB rxtime(ms) : rxbyte | t

  '' Wait ms milliseconds for a byte to be received
  '' returns -1 if no byte received, $00..$FF if byte

  t := cnt
```

```
  repeat until (rxbyte := rxcheck) => 0 or (cnt - t) / (clkfreq / 1000) > ms

PUB str(stringptr)

  '' Send zero terminated string that starts at the stringptr memory address

  repeat strsize(stringptr)
    tx(byte[stringptr++])

PUB getstr(stringptr) | index
    '' Gets zero terminated string and stores it, starting at the stringptr memory address
    index~
    repeat until ((byte[stringptr][index++] := rx) == 13)
    byte[stringptr][--index]~

PUB dec(value) | i

'' Prints a decimal number

  if value < 0
    -value
    tx("-")

  i := 1_000_000_000

  repeat 10
    if value => i
      tx(value / i + "0")
      value //= i
      result~~
    elseif result or i == 1
      tx("0")
    i /= 10


PUB GetDec : value | tempstr[11]

    '' Gets decimal character representation of a number from the terminal
    '' Returns the corresponding value

    GetStr(@tempstr)
    value := StrToDec(@tempstr)

PUB StrToDec(stringptr) : value | char, index, multiply

    '' Converts a zero terminated string representation of a decimal number to a value

    value := index := 0
    repeat until ((char := byte[stringptr][index++]) == 0)
       if char => "0" and char =< "9"
          value := value * 10 + (char - "0")
    if byte[stringptr] == "-"
       value := - value

PUB bin(value, digits)

  '' Sends the character representation of a binary number to the terminal.

  value <<= 32 - digits
  repeat digits
    tx((value <-= 1) & 1 + "0")

PUB GetBin : value | tempstr[11]

  '' Gets binary character representation of a number from the terminal
  '' Returns the corresponding value
```

```
  GetStr(@tempstr)
  value := StrToBin(@tempstr)

PUB StrToBin(stringptr) : value | char, index

  '' Converts a zero terminated string representaton of a binary number to a value

  value := index := 0
  repeat until ((char := byte[stringptr][index++]) == 0)
     if char => "0" and char =< "1"
        value := value * 2 + (char - "0")
  if byte[stringptr] == "-"
     value := - value

PUB hex(value, digits)

  '' Print a hexadecimal number

  value <<= (8 - digits) << 2
  repeat digits
    tx(lookupz((value <-= 4) & $F : "0".."9", "A".."F"))

PUB GetHex : value | tempstr[11]

    '' Gets hexadecimal character representation of a number from the terminal
    '' Returns the corresponding value

    GetStr(@tempstr)
    value := StrToHex(@tempstr)

PUB StrToHex(stringptr) : value | char, index

    '' Converts a zero terminated string representaton of a hexadecimal number to a value

    value := index := 0
    repeat until ((char := byte[stringptr][index++]) == 0)
       if (char => "0" and char =< "9")
          value := value * 16 + (char - "0")
       elseif (char => "A" and char =< "F")
          value := value * 16 + (10 + char - "A")
       elseif(char => "a" and char =< "f")
          value := value * 16 + (10 + char - "a")
    if byte[stringptr] == "-"
       value := - value

DAT

'***********************************
'* Assembly language serial driver *
'***********************************

                        org
'
'
' Entry
'
entry                   mov     t1,par                'get structure address
                        add     t1,#4 << 2            'skip past heads and tails

                        rdlong  t2,t1                 'get rx_pin
                        mov     rxmask,#1
                        shl     rxmask,t2

                        add     t1,#4                 'get tx_pin
                        rdlong  t2,t1
```

```
                        mov     txmask,#1
                        shl     txmask,t2

                        add     t1,#4                   'get rxtx_mode
                        rdlong  rxtxmode,t1

                        add     t1,#4                   'get bit_ticks
                        rdlong  bitticks,t1

                        add     t1,#4                   'get buffer_ptr
                        rdlong  rxbuff,t1
                        mov     txbuff,rxbuff
                        add     txbuff,#16

                        test    rxtxmode,#%100  wz      'init tx pin according to mode
                        test    rxtxmode,#%010  wc
        if_z_ne_c       or      outa,txmask
        if_z            or      dira,txmask

                        mov     txcode,#transmit        'initialize ping-pong multitasking
'
'
' Receive
'
receive                 jmpret  rxcode,txcode           'run chunk of tx code, then return

                        test    rxtxmode,#%001  wz      'wait for start bit on rx pin
                        test    rxmask,ina      wc
        if_z_eq_c       jmp     #receive

                        mov     rxbits,#9               'ready to receive byte
                        mov     rxcnt,bitticks
                        shr     rxcnt,#1
                        add     rxcnt,cnt

:bit                    add     rxcnt,bitticks          'ready next bit period

:wait                   jmpret  rxcode,txcode           'run chunk of tx code, then return

                        mov     t1,rxcnt                'check if bit receive period done
                        sub     t1,cnt
                        cmps    t1,#0           wc
        if_nc           jmp     #:wait

                        test    rxmask,ina      wc      'receive bit on rx pin
                        rcr     rxdata,#1
                        djnz    rxbits,#:bit

                        shr     rxdata,#32-9            'justify and trim received byte
                        and     rxdata,#$FF
                        test    rxtxmode,#%001  wz      'if rx inverted, invert byte
        if_nz           xor     rxdata,#$FF

                        rdlong  t2,par                  'save received byte and inc head
                        add     t2,rxbuff
                        wrbyte  rxdata,t2
                        sub     t2,rxbuff
                        add     t2,#1
                        and     t2,#$0F
                        wrlong  t2,par

                        jmp     #receive                'byte done, receive next byte
'
'
' Transmit
'
```

```
transmit                jmpret  txcode,rxcode           'run chunk of rx code, then return

                        mov     t1,par                  'check for head <> tail
                        add     t1,#2 << 2
                        rdlong  t2,t1
                        add     t1,#1 << 2
                        rdlong  t3,t1
                        cmp     t2,t3           wz
        if_z            jmp     #transmit

                        add     t3,txbuff               'get byte and inc tail
                        rdbyte  txdata,t3
                        sub     t3,txbuff
                        add     t3,#1
                        and     t3,#$0F
                        wrlong  t3,t1

                        or      txdata,#$100            'ready byte to transmit
                        shl     txdata,#2
                        or      txdata,#1
                        mov     txbits,#11
                        mov     txcnt,cnt

:bit                    test    rxtxmode,#%100  wz      'output bit on tx pin
                        test    rxtxmode,#%010  wc      'according to mode
        if_z_and_c      xor     txdata,#1
                        shr     txdata,#1       wc
        if_z            muxc    outa,txmask
        if_nz           muxnc   dira,txmask
                        add     txcnt,bitticks          'ready next cnt

:wait                   jmpret  txcode,rxcode           'run chunk of rx code, then return

                        mov     t1,txcnt                'check if bit transmit period done
                        sub     t1,cnt
                        cmps    t1,#0           wc
        if_nc           jmp     #:wait

                        djnz    txbits,#:bit            'another bit to transmit?

                        jmp     #transmit               'byte done, transmit next byte
'
'
' Uninitialized data
'
t1                      res     1
t2                      res     1
t3                      res     1

rxtxmode                res     1
bitticks                res     1

rxmask                  res     1
rxbuff                  res     1
rxdata                  res     1
rxbits                  res     1
rxcnt                   res     1
rxcode                  res     1

txmask                  res     1
txbuff                  res     1
txdata                  res     1
txbits                  res     1
txcnt                   res     1
txcode                  res     1
```

```
{{
┌──────────────────────────────────────────────────────────────────────────────────────┐
│                              TERMS OF USE: MIT License                               │
├──────────────────────────────────────────────────────────────────────────────────────┤
│Permission is hereby granted, free of charge, to any person obtaining a copy of this  │
│software and associated documentation files (the "Software"), to deal in the Software │
│without restriction, including without limitation the rights to use, copy, modify,   │
│merge, publish, distribute, sublicense, and/or sell copies of the Software, and to    │
│permit persons to whom the Software is furnished to do so, subject to the following   │
│conditions:                                                                           │
│                                                                                      │
│The above copyright notice and this permission notice shall be included in all copies │
│or substantial portions of the Software.                                              │
│                                                                                      │
│THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,   │
│INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A         │
│PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT    │
│HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION     │
│OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE        │
│SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.                                │
└──────────────────────────────────────────────────────────────────────────────────────┘
}}
```