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 have been individual objects. Applications are typically organized into multiple objects, and each object does a particular job. Every application has a *top level object*, which is the object where the code execution starts. All the example objects in the preceding labs have been top level objects, but they have not made use of any other objects.

Top level objects typically declare and call methods in one or more other objects. Many of these objects are designed to simplify application development. For example, some are collections of useful methods that are published so that common coding tasks don't have to be done "from scratch." Other objects manage processes that get launched into other cogs. They take care of most of the tasks introduced in the Methods and Cogs lab, including declaring stack space, tracking which cog the process got launched into, and so on. 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 Library (included with the Propeller Tool software), the web Propeller Object Exchange, and some can also be found on the Propeller Chip forum. Each object typically has documentation that explains how to incorporate it into your application along with an example top level file that uses the object. In addition to using pre-written objects, you may find yourself wanting to modify one, or write a custom object for a particular group of tasks that doesn't already have an object.

This lab guides you through writing a variety of objects and incorporating them into your top level objects. 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 other 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.

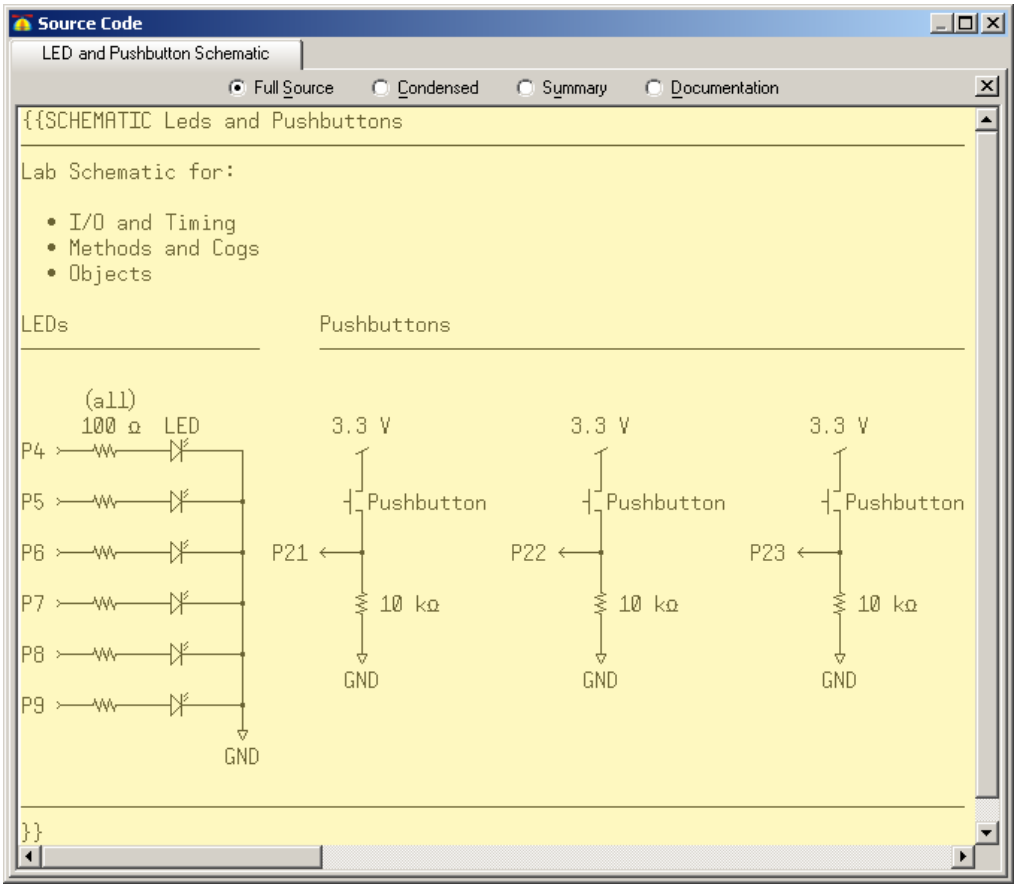
Prerequisites

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

Equipment, Parts, Schematic

Although the circuit is the same as the previous three 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 HyperTerminal, which will communicate with the Propeller chip via the Propeller Plug.

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.

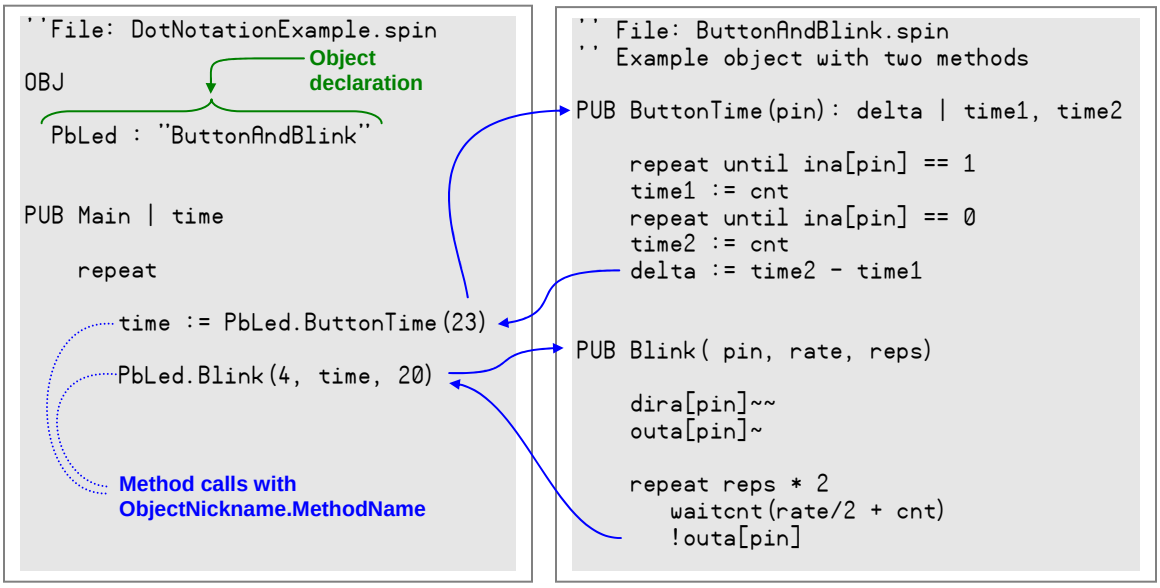
The Propeller Manual uses the term "symbolic reference" or "reference" instead of nickname.

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



- ✓ Load the DotNotationExample 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 DotNotationExample 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).

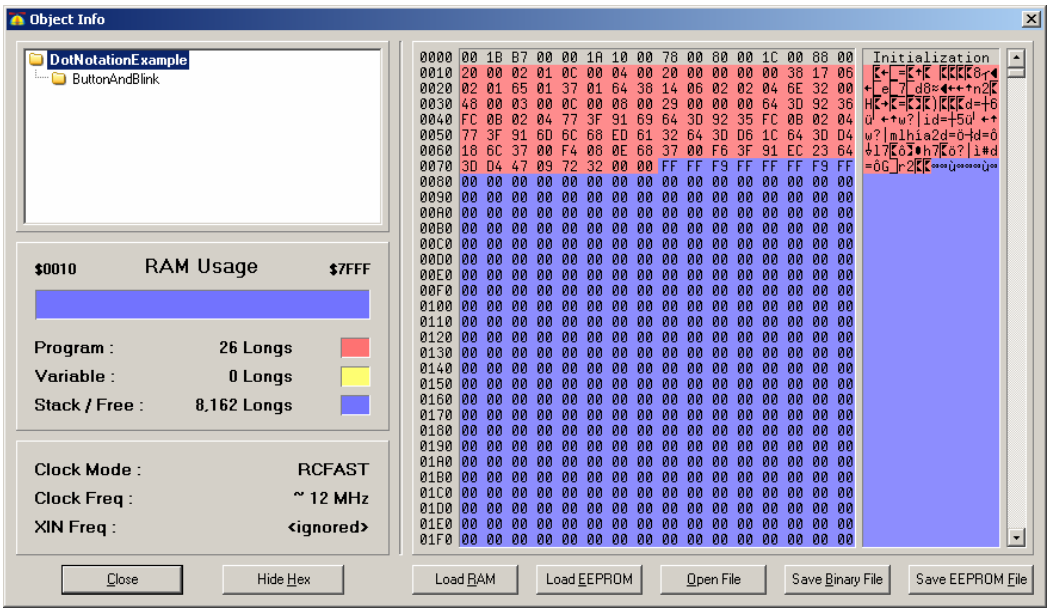
Object Organization

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).

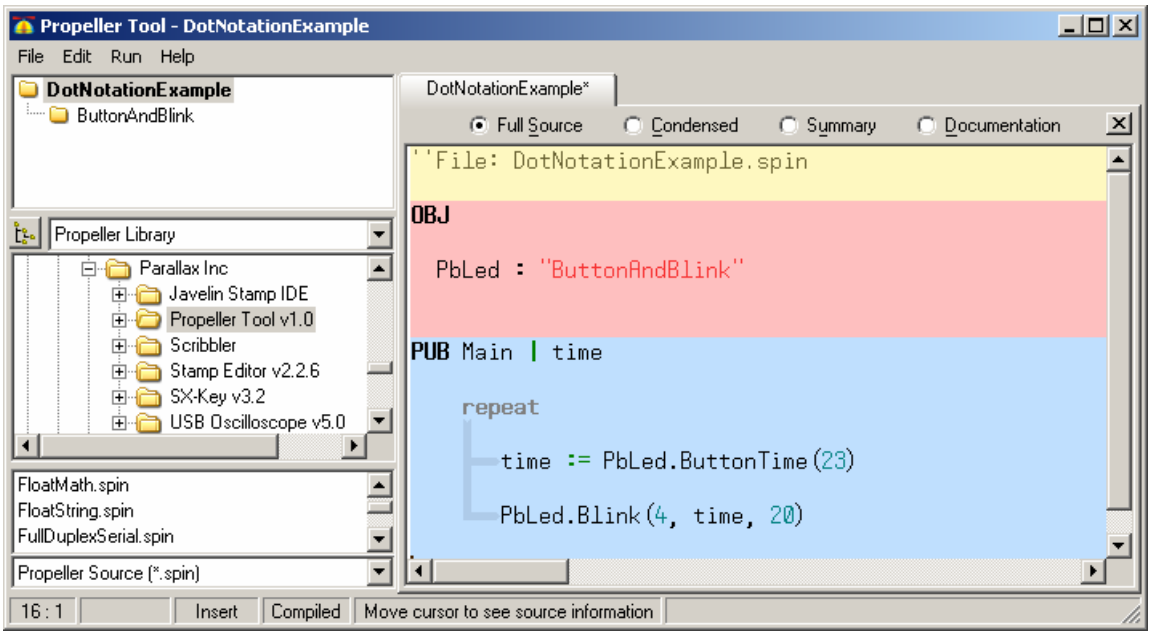
Notice that the object hierarchy is shown in the Object Info window's top-left corner. In this windowpane, you can single click each folder to see how much memory it occupies in the Propeller Chip's global RAM. You can also double-click each folder in the Object Info window to open the .spin file that 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. Left click the folder icon 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.

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. 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). That's because 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 call 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.

Again, the reason the LED will start blinking immediately is because the Blinker object automatically launched the LED blinking process into a new cog. This leaves Cog 0 free to repeatedly monitor the pushbutton for the next press/release.

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 into which cog it launched its `Blink` method.

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.

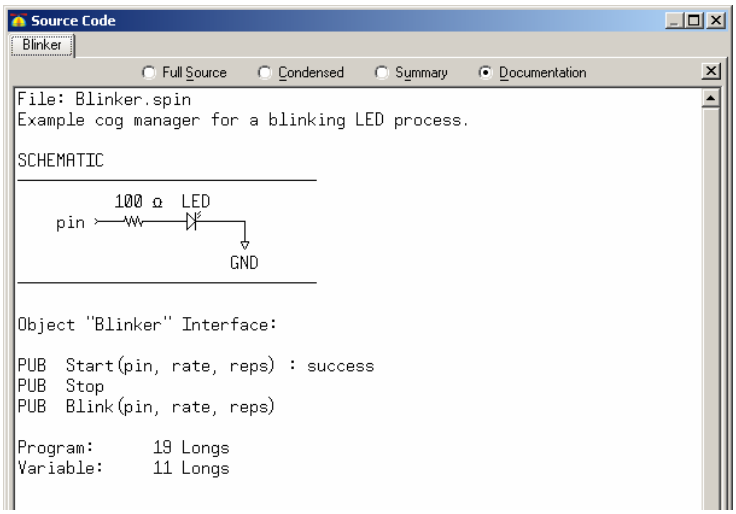
```
'' File: Button.spin
'' Beginnings of a useful object.

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.

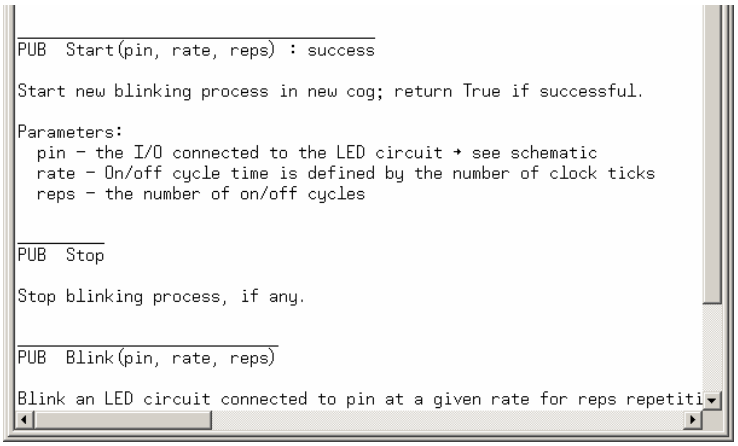
Figure 5: Documentation View



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

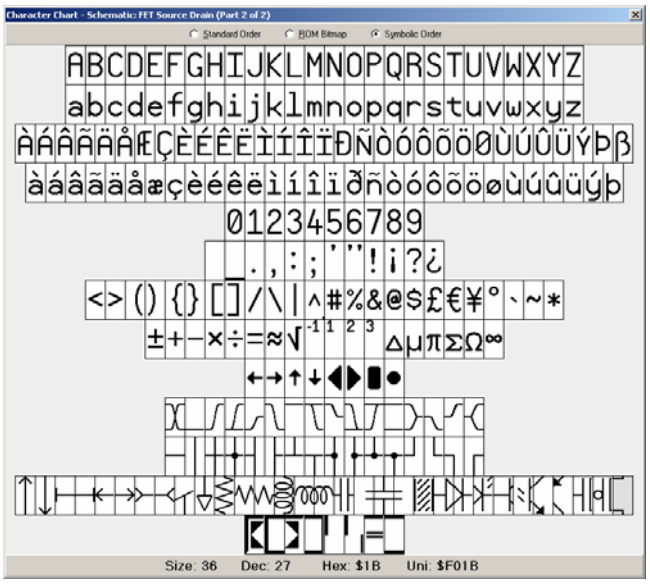


Drawing Schematics

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 Δ 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



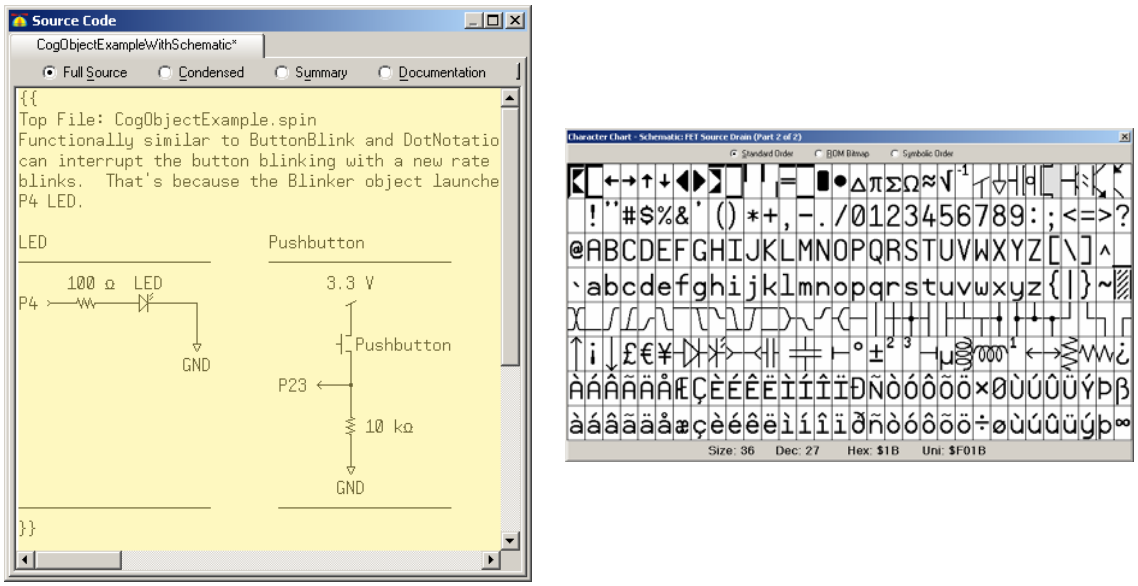
Top level files should also have schematics so that the circuit the code is written for can be built and tested. For example, the schematic shown in Figure 8 can be added to CogObjectExample. The pushbutton can be a little tricky. The character chart is shown in Figure 8, displayed in the standard order (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 simply call the `Blink` method.

- ✓ 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.

Some objects are written so that they have public methods that other objects can call, and private 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 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 simply declares 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 each button press. The first through sixth button presses launch new LED blinking processes in 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
```

Library Objects

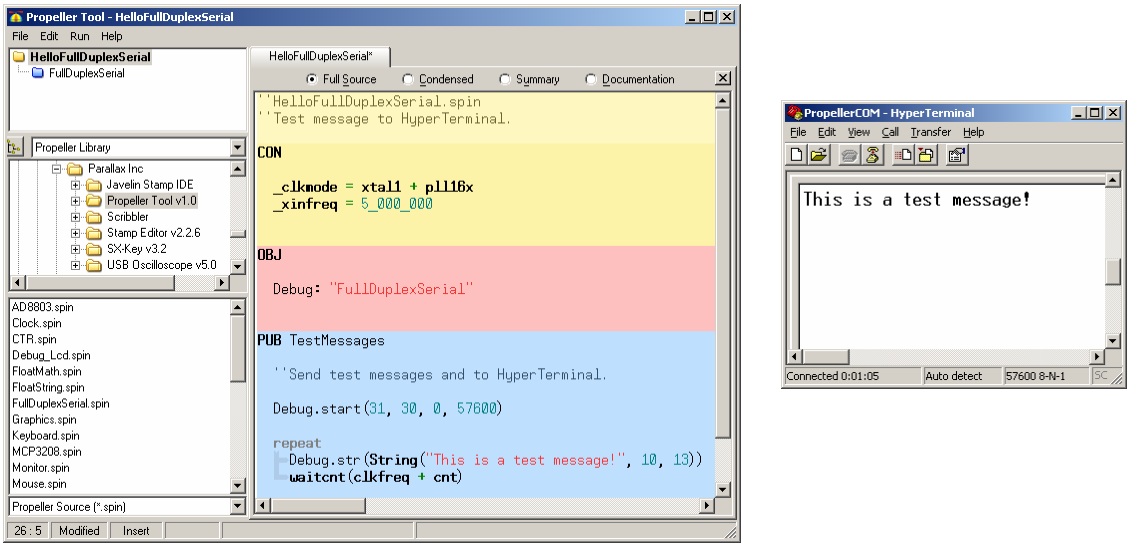
As mentioned earlier, code in an object can declare another object, so long as either:

- The two objects are in the same folder
- The object being declared is in the same folder as 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 and middle left windowpanes shown in Figure 9 and select *Propeller Library*. The Propeller Library's objects will appear in the lower left windowpane.

Figure 9: Using the FullDuplexSerial object to Display a Test Message in HyperTerminal



One of the objects in the Propeller Library that will be useful for this and other PE kit labs is the FullDuplexSerial object. True to its name, this object communicates with other devices using full duplex serial communication. *Full duplex* means that two channels of serial communication

(outbound and inbound) can occur simultaneously. One convenient and readily available device that communicates with full duplex serial is your PC. The HelloFullDuplexSerial object shown in Figure 9 uses the FullDuplexSerial object to send messages a program called HyperTerminal that's common to most Windows PCs.

Notice in Figure 9 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 v1.0.

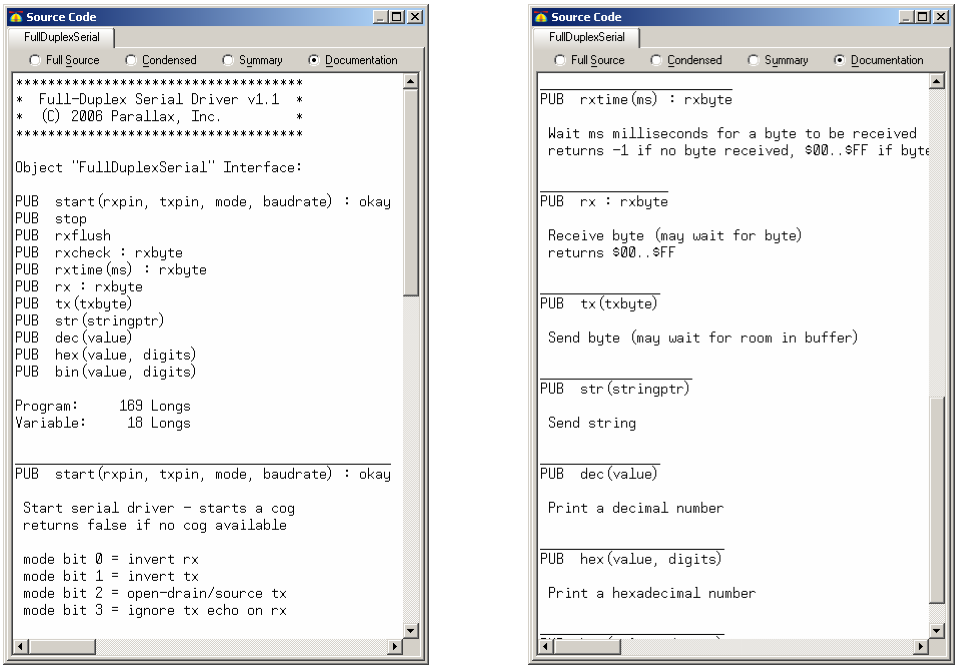
Serial to USB with the Propeller Plug: In the case of the PE Kit, the serial communication occurs between the Propeller chip and the Propeller Plug. The Propeller Plug converts the serial signals to USB signals, and sends them to the PC.

When using a library object, the first task is to examine its object interface to find out what methods can be used.

- ✓ 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 10.
- ✓ 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 10: FullDuplexSerial Object Documentation Views



The HelloFullDuplexSerial object below 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 HyperTerminal 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 PC using serial (EIA232) communication signaling.

```
''HelloFullDuplexSerial.spin
''Test message to HyperTerminal.

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000


OBJ

  Debug: "FullDuplexSerial"


PUB TestMessages

  ''Send test messages and to HyperTerminal.

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

  repeat
    Debug.str(string("This is a test message!", 10, 13))
    waitcnt(clkfreq + cnt)
```

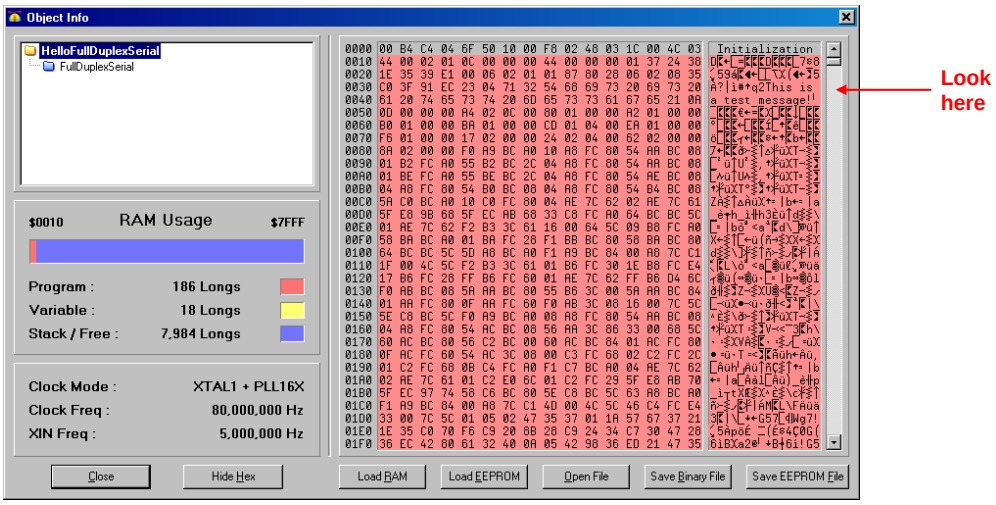
Let's take a closer look at Debug.str(String("This is a test message!", 10, 13)). Debug.str calls the FullDuplexSerial object's str method. The method declaration for the str method indicates that the parameter it expects to receive should be a string pointer. At compile, the **string** directive **string**("This is a test message!") stores the values that correspond to the characters in the text message in the Propeller chip's program memory, appended 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 transmit characters until it detects 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. The str method sends characters until it fetches the zero terminator.

Line Feed and Carriage Return: 10 is line feed, which puts the HyperTerminal's cursor on the next line, and 13 is carriage return, which moves it back to the leftmost position on the line.

You can see where the string gets stored in the program with the Object Info window.

- ✓ Open HelloFullDuplexSerial and view it with the Object Info window (F8).
- ✓ Look for the text in the rightmost column, 3rd and 4th lines. The hexadecimal ASCII codes occupy memory addresses 0038 through 0050 with the 0 terminator at address 51.

Figure 11: Finding a Text String in Memory

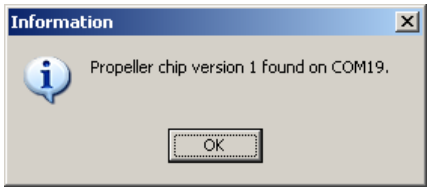


Setting up HyperTerminal

No additional physical connections need to be made for HyperTerminal, as we will be using the same port with which we are programming the Propeller chip, relying upon those circuits built in the Setup and Testing lab

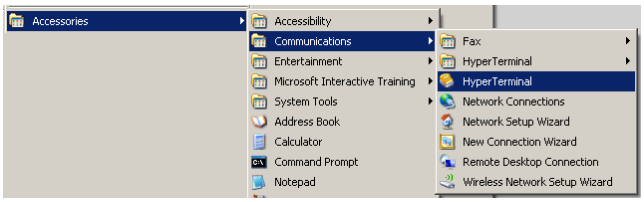
For the Propeller and HyperTerminal to communicate, they both have to be set to use the same serial communication settings. These settings determine the signaling and its timing. The HelloFullDuplexSerial configures the FullDuplexSerial object for a baud rate of 57600 bits per second. Other important details for configuring HyperTerminal are that FullDuplexSerial is designed to communicate with 8 data bits, 1 stop bit, no parity, and no flow control. Here's how to configure HyperTerminal to speak this particular serial communication dialect.

- ✓ Start in the Propeller Tool. Press F7, and make a note the COM port number shown in the Information window.

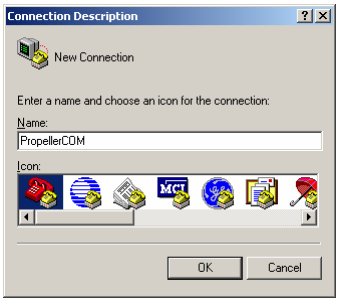


Most PCs with Windows 2000 or XP have HyperTerminal preinstalled with other Windows "Accessories". Follow these steps to set up the HyperTerminal to receive messages from the HelloFullDuplexSerial object.

- ✓ To open a new HyperTerminal Connection, click the Windows *Start* button, then select *All Programs* → *Accessories* → *Communications* → *HyperTerminal*. If you see both an icon and a folder, make sure to click the icon (click the application icon, not the folder).



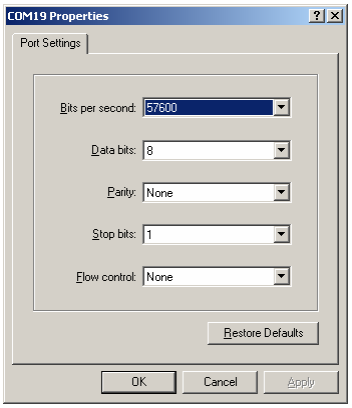
✓ Give the new HyperTerminal connection a name. These labs will use the name PropellerCOM.



✓ In the *Connect using field*, select the COM port you made a note of earlier with the Propeller Tool software's Information window.

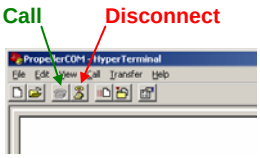
✓ In the COM properties window, select 57600 bits per second, 8 data bits, no parity, 1 stop bit, and no flow control.

✓ Then, click *OK* to continue. The HyperTerminal Window will appear.



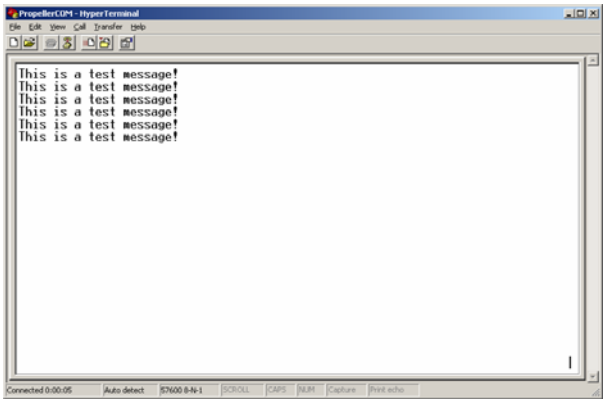
✓ Click *File* and select *Save as*...then save PropellerCom.ht with your Objects Lab .spin files.

✓ Click the *Disconnect* button to end the connection. This makes HyperTerminal release the COM port for the Propeller Tool to download programs to the Propeller chip.



✓ Use the Propeller Tool software to load HelloFullDuplexSerialCom into the Propeller chip.

✓ Click the HyperTerminalWindow's Call button. A new copy of "This is a test message!" should display every 1 second.



Keeping the COM Port Clear

Make sure to always disconnect the HyperTerminal software before loading an application into the Propeller chip.

- ✓ **Before loading a new Program into the Propeller chip, always click HyperTerminal's Disconnect button.**
- ✓ **After downloading a new program that sends messages to the Propeller chip, click HyperTerminal's Connect button to view the messages.**

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.

- ✓ Click HyperTerminal's disconnect button.
- ✓ Click File and select Properties.
- ✓ Click Configure
- ✓ Choose 115200 in the Bits per second dropdown menu, and then click OK.

- ✓ 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.
- ✓ Click HyperTerminal's Call 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.**

Displaying Values

Take another look at the FullDuplexSerial object in documentation mode. (See Figure 10 on page 14.) 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. 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("counter = "))
  Debug.dec(counter++)
  Debug.str(String(10, 13))
  waitcnt(clkfreq/5 + cnt)
```

- ✓ Remember to make sure that HyperTerminal's *Disconnect* button has been clicked.

- ✓ Use the Propeller Tool software to load the modified version of HelloFullDuplexSerial into the Propeller chip's EEPROM.
- ✓ Click HyperTerminal's Connect button, and verify that the updated value of counter is displayed five times every second. You can push the PE Platform's Reset button to start the count at 0 again.

Sending Values from HyperTerminal to the Propeller Chip

The FullDuplexSerial object does not have a corresponding `GetDec` method to complement `dec`. So, as written, you cannot type a decimal value and send it to the Propeller chip via HyperTermnial. 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 receive and interpret decimal, hexadecimal and binary character representations from HyperTerminal and convert them to their corresponding numeric values, which can in turn be stored in variables. Since it also has the same methods as FullDuplexSerial, calls like `Debug.start`, `Debug.str`, and `Debug.dec` still yield the same results.

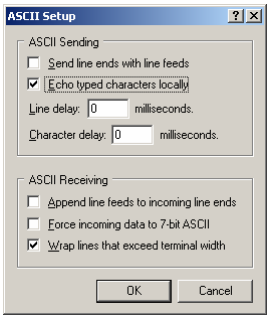
Remember that an object can be declared so long as it's either in the same folder with the object that's referencing it, or in the same folder as the Propeller Tool software. In this case, the FullDuplexSerialPlus object is in the same folder with this lab's example objects. So, it can be declared in 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.

Modifying HyperTerminal to Echo

HyperTerminal does not echo characters typed locally by default. That means, if you type a character, it won't show in the window. The characters still get sent to the Propeller chip, but it can be a little unnerving not having the feedback displayed by HyperTerminal. Here's how to configure HyperTerminal to echo characters typed locally.

- ✓ Open PropellerCOM.ht.
- ✓ Click the *Disconnect* button.
- ✓ Click *File* and select *Properties*.
- ✓ Click the PropellerCOM properties' *Settings* tab.
- ✓ Click the *ASCII Setup* button.
- ✓ Set a checkmark in the *Echo typed characters locally* checkbox.
- ✓ Click *File* and select *Save* so that this feature will be available next time you open PropellerCOM.ht.

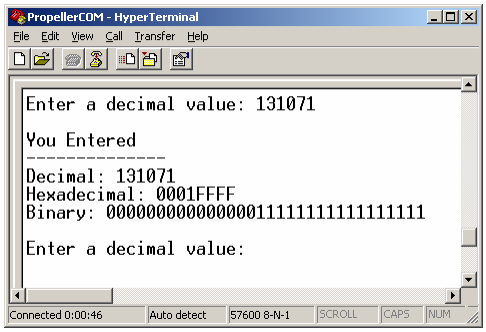


Testing HyperTerminal for Input Values

The EnterAndDisplayValues object below waits for you to enter a value into HyperTerminal. Then, it converts the characters that represent the value into a numeric equivalent and displays them in decimal, hexadecimal and binary format in HyperTerminal.

Figure 12 shows an example of testing the EnterAndDisplayValues object with HyperTerminal. The object makes the Propeller Chip send prompts that are displayed in HyperTerminal for entering a value. After pressing enter, the Propeller chip converts the characters to the 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 12: Testing for Input Values



- ✓ Make sure HyperTerminal is disconnected.
- ✓ Load EnterAndDisplayValues into EEPROM (F11).
- ✓ Connect HyperTerminal (click the Connect button).
- ✓ Restart the Propeller Chip's program by pressing and releasing the PE Platform's reset button (the one on the left side connected to RESn and GND).
- ✓ Follow the prompts in HyperTerminal. Start with 131071 and verify that it displays the values shown in Figure 12.

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 HyperTerminal.
'' 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 HyperTerminal number entry and display.

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

```
repeat

   Debug.Str(String("Enter a decimal value: "))
   value := Debug.getDec
   Debug.Str(String(10, 10, 13, "You Entered", 10, 13, "--------------"))
   Debug.Str(String(10, 13, "Decimal: "))
   Debug.Dec(value)
   Debug.Str(String(10, 13, "Hexadecimal: "))
   Debug.Hex(value, 8)
   Debug.Str(String(10, 13, "Binary: "))
   Debug.Bin(value, 32)
   repeat 2
      Debug.Str(String(10, 13))
```

Debug.dec vs. Debug.getDec

The FullDuplexSerialPlus object's `GetDec` method buffers characters it receives form HyperTerminal 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, that's all you'll need to display with the binary value.

- ✓ 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.

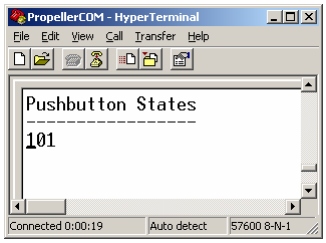
Each hexadecimal digit takes 4 bits. So, it will take 4 digits to display all possible values in a word variable (16-bits).

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

Terminal I/O Pin Input State Display

The HyperTerminal 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 13. A 1 in a particular slot indicates the pushbutton is pressed; a 0 indicates the pushbutton is not pressed. Figure 13 shows an example where the P21 and P23 pushbuttons are pressed.

Figure 13: HyperTerminal Pushbutton State Display

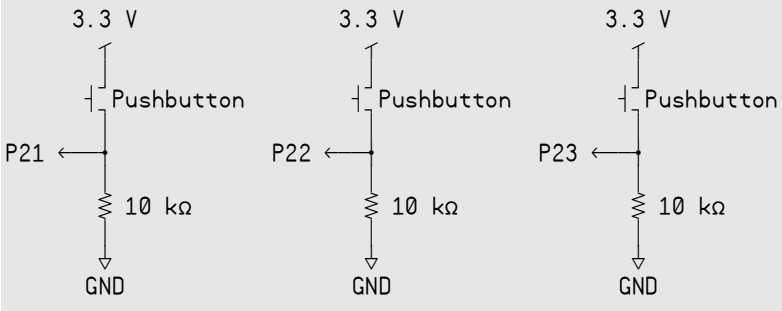


The DisplayPushbuttons object uses the command `Debug.Bin(ina[23..21], 3)` to display the pushbutton states. Recall from the I/O and Timing lab that `ina[23..21]` returns the value stored in bits 23 through 21 of the `INA` register. This result gets passed as a parameter to the FullDuplexSerialPlus object's `bin` method with the command `Debug.bin(ina[23..21], 3)`. Note that since there are only 3 bits displayed, the `bin` method's `bits` parameter is only 3, which in turn makes the method display only 3 binary digits. Notice also that the command that follows it is `Debug.tx(13)`. This sends the carriage return byte (13), but not the line feed byte (10). That's why the display refreshes on the same line.

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.

- ✓ Disconnect HyperTerminal.
- ✓ Load the DisplayPushbuttons object F11.
- ✓ You will have 3 seconds to connect HyperTerminal after the Propeller Tool is finished, or press the breadboard's Reset button if needed.
- ✓ Press the pushbuttons and verify that the display is correct.

```
{{
DisplayPushbuttons.spin
Display pushbutton states with HyperTerminal.

Pushbuttons
```

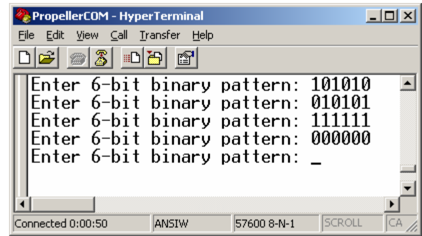



```
}}

CON

  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000

OBJ

  Debug: "FullDuplexSerialPlus"

PUB TerminalPushbuttonDisplay

  ''Read P23 through P21 pushbutton states and display with HyperTerminal.

  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*3 + cnt)
  Debug.str(String("Pushbutton States", 10, 13, "-----------------", 10, 13))

  repeat

     Debug.Bin(ina[23..21], 3)
     Debug.Tx(13)
     waitcnt(clkfreq/100 + cnt)
```

Terminal LED Output Control

Testing various actuators is also important during prototyping. The TerminalLedControl object demonstrates a convenient means for setting output states for testing various output circuits, shown in Figure 14. While this example uses LED indicator lights, it could just as easily be chip enable inputs, solenoids, relays, DC motors, or control circuits.

Figure 14: 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 characters representing a binary value that it receives. The value the `GetBin` method returns is assigned to `outa[9..4]`, which makes the corresponding LED pattern light up.

- ✓ Make sure HyperTerminal is disconnected.
- ✓ Load TerminalLedControl into the Propeller chip.
- ✓ You will have 3 seconds to connect HyperTerminal.
- ✓ Try the values shown in Figure 14, and verify that the corresponding LED patterns light up.

```
{{
TerminalLedControl.spin

Enter LED states into HyperTerminal.  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 HyperTerminal.
```

```
  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*3 + cnt)
  dira[4..9]~~

  repeat

     Debug.Str(String(10, 13, "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    10, 13, "This is test message number: ", 0
  MyOtherString   byte    "", "", 10, 13, "and this is another line of text.", 0
  CR              byte    10, 13, 0
```

Remember that the string directive returns the starting address of the 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.

- ✓ Make sure HyperTerminal is disconnected. Load the TestMessages object (F11), and then connect HyperTerminal.
- ✓ Verify that the three messages are displayed each second.

```
'' TestMessages.spin
'' Send text messages stored in the DAT block to HyperTerminal.

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*3 + cnt)

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

DAT

MyString        byte    10, 13, "This is test message number: ", 0
MyOtherString   byte    "", "", 10, 13, "and this is another line of text.", 0
CR              byte    10, 13, 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   10, 13, "The value is: ", 0
  ElNumTxt    byte   "", "", 10, 13, "and it's element #: ", 0
  ValueList   long   98, 5282, 299_792_458, 254, 0
  CR          byte   10, 13, 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 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 sizes 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 TwoWayCom | value, index

  Debug.start(31, 30, 0, 57600)
  waitcnt(clkfreq*3 + cnt)

  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(@CR)
        waitcnt(clkfreq + cnt)
```

- ✓ Test the modified object with the Propeller chip and HyperTerminal. 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 $\pm 3.403 \times 10^{38}$, or as small as $\pm 1.175 \times 10^{-38}$.

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. HyperTerminalFloatStringTest demonstrates some basic floating-point operations. First, `a := 1.5` and `b := pi` are using the Propeller Tool's 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`.

```
''HyperTerminalFloatStringTest.spin
''Solve a floating point math problem and display the result with HyperTerminal.

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)

  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.

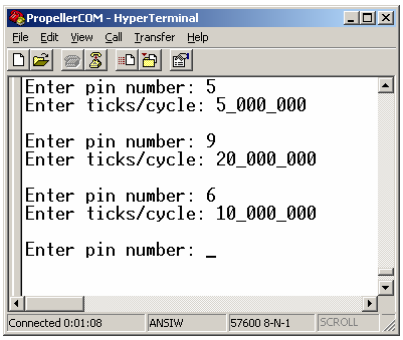
AddressBlinker is an example of an object that fetches values from its parent object's variables. Note that its Start method has parameters for two address values, pinAddress and rateAddress. The parent object has to pass the AddressBlinker object's Start method the address of a variable that stores the I/O pin number, and another that stores the rate. The Start method relays these pareameters 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.

```
'' 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 below uses the AddressBlinker object. 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.

- ✓ Disconnect PropellerCOM.
- ✓ Load AddressBlinkerControl into the EEPROM (F11). Connect PropellerCOM.
- ✓ Try the values shown in Figure 15, and keep an eye on the LEDs.

Figure 15: Entering Pin and Rate into HyperTerminal



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

```
'' AddressBlinkerControl.spin
'' Enter LED states into HyperTerminal and send to Propeller chip via HyperTerminal.

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)

  pin := 4
  rateDelay := 10_000_000

  AddrBlnk.start(@pin, @rateDelay)

  dira[4..9]~~

  repeat

     Debug.Str(String(10, 13, "Enter pin number: "))
     pin := Debug.getDec
     Debug.Str(String(10, 13, "Enter delay clock ticks:"))
     rateDelay := Debug.getDec
     Debug.Str(String(10, 13))
```

Displaying Addresses

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 HyperTerminal just before the repeat loop starts, and display the value of those variables each time they are entered.

Address offsets

Objects sometimes require that the parent object declare a sequence of variables, each storing a particular type of value that it will monitor from another cog. This approach is useful for preventing the parameter list length from getting out of hand. For example, if there are twenty variables that the object needs to monitor in the parent object, a single address at the start of the variables is all that needs to get passed to the object. Keep in mind these variables have to be declared in an order specified by the object's documentation.

AddressBlinkerWithOffsets demonstrates how this works. The only difference between this object and AddressBlinker is that it receives the address of the parent object's variable that stores the pin value. The address of this variable is the only one that gets passed, and the object requires that the variable storing the blink rateDelay variable be declared next, immediately to the right of the parent object's pin value variable declaration.

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

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


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.

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


PUB Stop
''Stop blinking process, if any.

  if Cog
    cogstop(Cog~ - 1)


PUB 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)
```

The AddressBlinkerControlWithOffsets object below does the same thing as the previous AddressBlinkerControl example object, but it uses the AddressBlinkerWithOffsets object, passing it a single `address` parameter (the address of its `pin` variable).

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. However, it's perfectly fine to declare long variables before and after these two. It's just that they have to be long, and they have to be declared in the specified order. Remember to keep an eye open for this feature in objects that launch processes into other cogs.

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

```
'' File: AddressBlinkerControlWithOffsets.spin
'' Another example cog manager that watches variables in its parent object.
'' This one only takes one address, but uses it as an anchor for watching
'' three varaibles in the parent object.

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 HyperTerminal.

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

  pin := 4
  rateDelay := 10_000_000

  AddrBlnk.start(@pin)

  dira[4..9]~~
```

```
repeat

   Debug.Str(String(10, 13, "Enter pin number: "))
   pin := Debug.getDec
   Debug.Str(String(10, 13, "Enter delay for 'rate':"))
   rateDelay := Debug.getDec
   Debug.Str(String(10, 13))
```

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 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 MyLedObjec.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`.

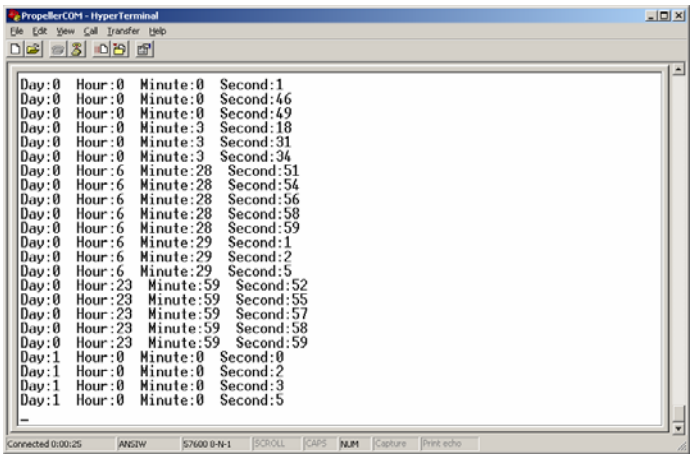
Projects

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 HyperTerminal 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 HyperTerminal display that gets updated each time the PE Platform's P23 pushbutton is pressed and released.



The HyperTerminal gets updated by the HtButtonLogger 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 HtButtonLogger 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 HtButtonLogger object. Make sure to use the second counting technique from the `GoodTimeCount` method from the I/O and Timing lab.

```
'' HtButtonLogger.spin
'' Log times the button connected to P23 was pressed/released in HyperTerminal.

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.
  Time.Start(0, 0, 0, 0)                 ' Start the TickTock object and initialize
                                         ' the day, hour, minute, and second.
  Debug.Str(@BtnPrompt)                  ' Display instructions in HyperTerminal
  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.Str(@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    10, 13, "Press/release P23 pushbutton periodically...", 0
CR          byte    10, 13, 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 multiline block of documentation text can be defined with double-braces like this `{{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) The FloatString object.
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 HyperTerminal, save a copy of PropellerCOM under a new name, such as TestPropellerStack.ht. Make sure HyperTerminal is disconnected, then click *File* and select *Properties*. Click the *Configure* button in the TestPropellerStack properties window, and change the value in the *Bits per second* field from 57600 to 19200. Click *OK* buttons until you get back to HyperTerminal.

   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` 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 StackLengthDemoModified.spin 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 HyperTerminal. 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

```
{{
┌──────────────────────────────────────────────────────────────────────────────┐
File: FullDuplexSerialPlus.spin

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.

└──────────────────────────────────────────────────────────────────────────────┘
}}


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[--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 a chunk of transmit 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 a chuck of transmit 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 a chunk of receive 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 according to
mode
                        test    rxtxmode,#%010  wc
        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 a chunk of receive 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
```