The Object of the Machine
By Jon Williams

For Nuts & Volts Magazine, May 2006

So, is your head spinning after last month's introduction to the Propeller chip? Don't worry, it happens to all of us, and I promise that after a bit of time things will begin to click, a big smile will cross your face, and wonderful things you thought never possible will start happening. Last month we talked about the Spin programming language being object oriented, but didn't really take advantage of it. Let's change that, shall we? and unleash some of the power of the Propeller multi-controller.

While I consider myself a pretty fair programmer, I always qualify that statement with the assertion that I'm a pretty fair *high-level language* programmer. Of course, I can program a bit of assembly, but I really don't like to. What that means, then, is when I've wanted to incorporate assembly code written by another programmer (e.g., in an SX/B project), it's been a bit of work. I've got great news for Propeller users: using assembly language written by another programmer is no trouble at all, and we're going to see that this month.

But let's go through a bit of a review first. Remember that the Propeller chip has eight 32-bit cogs (processors) in it, and all can be running at the same time. Every cog that is running has direct access to the I/O pins, as well as to the main system counter (useful for generating delays). There is a system manager called the "hub" that controls access to the shared resources; specifically the main system RAM (32K).

A cog can run the Spin language interpreter, or a custom assembly language program. The fact is that the Spin interpreter is an assembly language program that is loaded from the system ROM when needed. So, for those of you concerned about each cog having only 2K of RAM, don't be; this is plenty for assembly programs (remember, this is a whole new assembly language and is very efficient). Any Spin code that we write actually resides in the main system RAM, so our Spin programs and their data space can be up to 32K. Of course, there is a performance difference between Spin and Assembly, by about a factor of 250x. That said, Chip has estimated that with a 5 MHz crystal and using the 16x PLL tap (system clock of 80 MHz), we can run about 80K Spin instructions per second. That's pretty fast.

So let's jump right in and demonstrate Propeller objects and the ability to use assembly language with our Spin projects. For our first project we're going to create a "debug" object that allows us to send information to a PC. Some of you may be surprised that this is not a built-in function – don't be. The Propeller is a different beast and you wouldn't want to be penalized by having code space consumed by unused functions. Let's say you'd rather send values to a TV; you can do that using the *TV_Terminal* object that Chip wrote and comes with the Propeller installation. In fact, I've borrowed the numeric conversion routines from *TV_Terminal* object for us in *PC_Debug*. Let's build that object.

The purpose of *PC_Debug* is, of course, to send information to a PC terminal program. What this means, then, is that we need a code to handle the serial transmission. While we could do that ourselves, why bother? as Chip has kindly written a high-performance UART object called *FullDuplex* that we can take advantage of. What we're going to do with *PC_Debug* is provide a convenient wrapper for *FullDuplex* that gives us access to most of the methods in *FullDuplex*, as well as adding any conveniences that we might like to have (like number-to-string conversion).

Notice that the ZIP file I've provided for downloading this month has a very specific name and naming convention; this is actually a Propeller archive file. We'll talk more about archives later, just know for the moment that an archive contains all the files we need for a given project. Expand the archive so that you can open the files with the Propeller Tool, and then have a look at *PC_Debug.spin*.

In order to use an object in our program we need to declare it; we do that in the **OBJ** block like this:

```
OBJ
  uart : "fullduplex"
```

We now have an object in our project called "uart" that – once started – gives us buffered serial communications using another cog (which means it can do things without affecting the program running in the our main program cog). What we've done, in essence, is added a serial coprocessor to our system. Pretty cool, huh? It gets better.

The Parallax philosophy is that support objects, i.e., those that are not intended to stand alone, will have a method called "*start*" that is used for instantiation. The *start* method will usually return True (-1) or False (0) based on the success of the code at *start*. Note that there is no hard-and-fast rule on this, it's just the current convention.

Since *PC_Debug* is also designed as a support object, it will also have a start method. Here it is:

```
PUB start(baud) : okay

  okay := uart.start(31, 30, baud)
```

This is a simple method, and yet a lot is happening. We start with the **PUB** declaration – we need this method to be public so that it can be accessed by higher-level objects. This method is expecting a parameter called *baud*. Note that no matter what size we need, a parameter is always passed as a Long. This method will return a value as well; the variable after the colon (*okay*) is what will be returned. Return values are also Longs but can be caste to smaller sizes (Word or Byte) if needed.

The code now is just one line: we're assigning the return value of the *uart.start* method to okay. As we can see, Spin uses the dot notation found in other object-oriented languages. We can also tell that the *uart.start* method expects three parameters: the receive pin, the transmit pin, and the baud rate). What we've done here is started the *uart* object using the Propeller's standard programming pins.

But what if we've got an extra port on our PC and would rather send information to it using a couple free I/O pins? No problem, we'll just create another method.

```
PUB startx(rx_pin, tx_pin, baud) : okay

  okay := uart.start(rx_pin, tx_pin, baud)
```

As you can see the *startx* (x for extra) method simply passes along the desired pins with the baud rate. Using this method we could actually open more than one terminal at the same time (using different ports on our PC, of course). Spin even lets us define an array of objects, so we could do this:

```
OBJ
  terminal[2] : "pc_debug"
```

Now we just need to assign the terminals to different Propeller I/O pins.

```
PUB main
```

```
  terminal[0].start(9600)
  terminal[1].startx(1, 0, 57600)
```

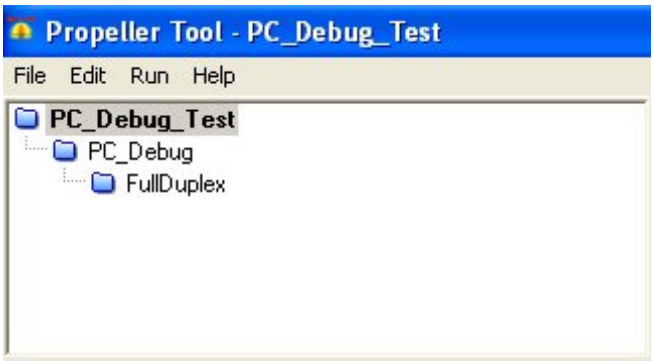
In this case *terminal.[0]* is using the default programming pins (A31 and A30) at 9600 baud, and *terminal.[1]* is using A1 (for RX) and A0 (for TX) at 57,600 baud. Keep in mind that the underneath the terminal objects is the *FullDuplex* UART object that requires its own cog, so the definition above would require two free cogs to operate.

Let's get back to our *PC_Debug* object. Again, this is a wrapper for *FullDuplex* that adds features convenient for sending data to a terminal. Since the *FullDuplex* object starts a new cog it also has a method for stopping that cog and making it available for other processes. By convention this method is called *stop* and we simply provide access to it.

```
PUB stop

  uart.stop
```

This may seem redundant but in fact it's not. You see, any program (top object) that uses *PC_Debug* will not direct access to methods in *FullDuplex* – we must explicitly provide wrappers for them. The good thing about this is that we can provide wrappers only as needed, and leave the other methods (even public) protected to a degree. Figure 1 shows object hierarchy of our project completed project; note that *PC_Debug_Test* does not have a direct connection to *Full_Duplex*.



As you look through the *PC_Debug* object you'll see that that are several other wrappers for objects in *FullDuplex*; they're self-evident and we don't need to describe them all in detail. Let's jump into the custom methods that are at the purpose of our project: converting values to strings so that we can send them to a terminal program.

Since we'll most frequently use decimal values, let's start there. The following method will print a signed decimal number.

```
PUB dec(value) | div, zpad

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

  div := 1_000_000_000
  zpad~
```

```
repeat 10
  if (value => div)
    out(value / div + "0")
    value //= div
    zpad~~
  elseif zpad or (div == 1)
    out("0")
  div /= 10
```

Okay, I know that this may look a little cryptic at first, but please trust me that once you get used to Spin you'll love the efficiency of the language. As I told you last month, Spin borrows from other languages, and those of you that have programmed in C will probably recognize some of the operators and constructs right away.

Let's start with the declaration because it includes something new. We can see that we're going to pass a value, and following that is a vertical bar and two symbols: *div* and *zpad*. The symbols are local variables that will be used by the method. Note that local variables are not persistent and will be destroyed when we exit from the method.

The beginning of the code is quite simple; we simply check to see if the value passed is negative and if it is we make it positive and print a "-" character with the *out* method. Next we initialize the divisor (*div*) and clear the *zpad* flag. There are a few cool things here: with 32 bits, we can deal with *really big* numbers (-2,147,483,648 to +2,147,483,647), and Spin lets us see this clearly by using and underscore character where a comma would normally be. Next is a new operator, the post clear (~) operator.

As you spend more time you'll find the Spin is very advanced, and the placement of an operator can change its meaning considerably. In our case the trailing tilde means that we're going to clear the variable to zero. So…

```
zpad~
```

is the same as

```
zpad := 0
```

but the former version is in fact more efficient internally. Now we get to the meat of the *dec* method. Since the largest value in the system can be up to 10 digits wide, we'll run the digit conversion loop 10 times. Again, note the efficiency with the simple **repeat 10** statement; this replaces for **x = 1 to 10** in BASIC (though there is an implementation of **repeat** that allows us to specify start and end values). You may be wondering about the control variable for the **repeat** loop; this comes from the interpreter's stack.

Now we check to see if *value* is equal to or greater than the divisor. If it is, we get the current column digit by dividing *value* by the divisor, and then convert it to ASCII by adding "0" (decimal 48). Now we remove the current column by taking the modulus of the divisor. Since we've started printing digits we will now set the *zpad* flag so that we print zeros in proceeding columns as needed. Note the post-set operator (two trailing tildes); this sets all the bits of the variable to 1 (making the value -1, which is generally used as True).

When the current value is less than the divisor, we check the *zpad* flag or for the current column being 1; if either of those conditions is true then we'll print a zero. The final step is to adjust the divisor between columns by dividing it by 10.

Okay, now let's look at binary and hex conversion. These routines are trim and elegant (*I didn't create them so I can say that*), yet also demonstrate some neat features in Spin. We'll start with binary as it is the simpler of the two.

```
PUB bin(value, digits)

  digits := 1 #> digits <# 32
  value <<= 32 - digits
  repeat digits
    out((value <-= 1) & %1 + "0")
```

This method differs slightly from *dec* in that we're required to specify a number of digits, but as you can see, there's really not much to the code. We start by qualifying the *digits* parameter with the **#>** (limit minimum) and **<#** (limit maximum) operators – this takes care of a bad value getting passed to the method. Then we shift the MSB of the printed output to bit 31 with the left shift operator. Note that as with many other operators, left-shift (**<<**) and variable assignment (**:=**) are combined into a single operator.

Now for the real work; a loop is used to print the number of digits passed. The code starts by rotating the bits left one position. Rotating differs from shifting in that no bits are lost, they simply wrap around to the other end of the value. So when we rotate left (**<-**) by one bit, what was in bit 31 ends up in bit 0. Now we AND this with %1, and then convert the digit to ASCII for printing. I don't know about you, but I think this routine is pretty darned nifty.

Okay… ready for hex conversion? It's similar, but we're dealing with nibbles so there's a little extra in the code.

```
PUB hex(value, digits)

  digits := 1 #> digits <# 8
  value <<= (8 - digits) << 2
  repeat digits
    out(lookupz((value <-= 4) & %1111 : "0".."9", "A".."F"))
```

Since a hexadecimal digit occupies four bits, we have to shift by four bits to align the most significant digit. After qualifying *digits* and subtracting that from eight, we shift the intermediate result by two – this is a more efficient way of multiplying by four. Our **repeat** loop works like it did in the *bin* method, except that we rotate value by four bits for each digit, AND with %1111, and finally use **lookupz** (zero-indexed **lookup**) for the correct digit character to print. A useful feature in **lookupz** is the ability pass an implicit list of values requiring only the starting and ending points, hence "0".."9" replaces "0123456789". Note, too, that we can create a compound list by separating multiple lists and items with commas.

I think it's about time to put our *PC_Debug* object to work, don't you? In the archive you'll find a simple program called *PC_Debug_Test.spin*. It's pretty short, and with what we've already been through we can focus on the body of the program; the **CON** and **OBJ** sections are very straightforward.

```
PUB main | idx

  debug.start(460_800)
  debug.str(string(FF, "Debug Test", CR, LF, LF))

  repeat
```

```
    debug.hex(idx, 2)
    debug.out(Space)
    if ((++idx // 16) == 0)
      debug.crlf
  until (idx == $100)

  debug.crlf
  debug.dec(-1)
  debug.crlf
  debug.ibin(-1, 32)
  debug.crlf
  debug.ihex(-1, 8)
  debug.stop
```

There's only one public method in the program, and I've called it *main* – this is a style choice and not required. Remember, the first public method is what runs when a Spin program is launched. The first thing we do is start the *debug* object, and have a look at that baud rate: 460,800 – that is not a typo, that is 460.8 kBaud. Remember I said Chip's *FullDuplex* object was "high performance"? Now you can see what I'm talking about. And this is with a 5 MHz crystal connected to the Propeller chip.

The first thing printed is a string that is composed of a form feed character (clears the screen in HyperTerminal), some text, a carriage return, and a couple line feeds. All this is assembled with the ***string*** method which creates the inline string and returns a pointer (the address in memory of) to it. The string pointer is what's used by the *debug.str* method for printing. This is fine for one-off strings, but if we're going to use the same text more than once it's better to embed it into a **DAT** block like this:

```
DAT

title   byte      "Nuts & Volts rocks!", 0
```

Note the zero terminator; this is important so don't leave it out. To print this string we can pass a pointer to it with the @ operator.

```
  debug.str(@title)
```

The main body of the program is a loop that prints hex values from $00 to $FF in a 16 by 16 array. After printing the digit and a space, the value of *idx* is incremented and then tested with modulus to see if 16 values have been printed on the current line. If so, and carriage-return and line feed are inserted. The value of *idx* is tested at the end of the loop for termination.

Of course, there are several ways to skin this cat – we could have constructed the start of the loop like this:

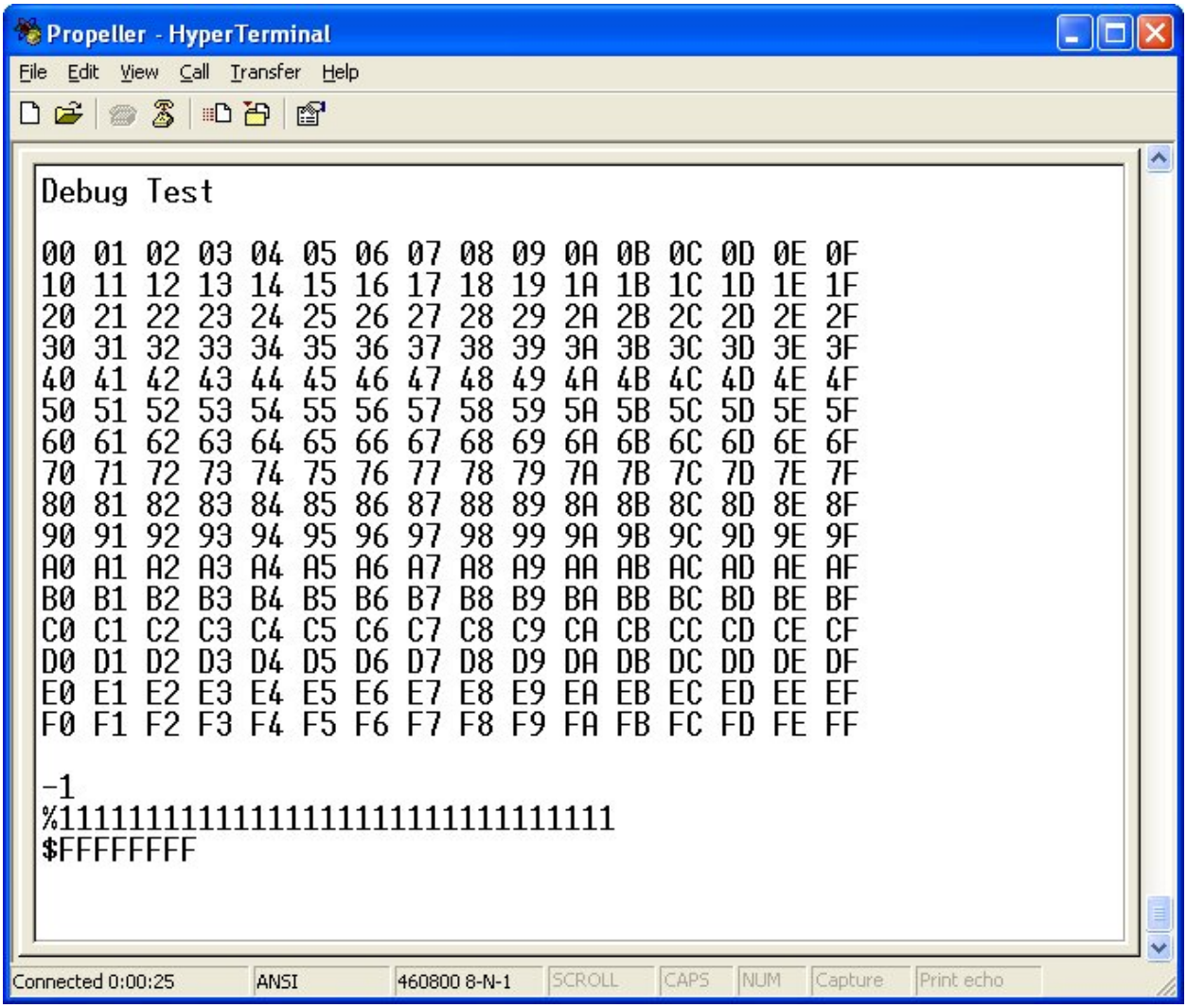
```
  repeat idx from $00 to $FF
```

Another option is to replace the **until** termination with:

```
  if (idx == $100)
    quit
```

My point is to show you that **repeat** – the only looping construct in Spin – is quite flexible and has a wide variety of options.

Okay, now that you've got a tool for sending values to a PC terminal program it is time to play; you have enough to experiment with the Spin programming language and get used to it before we start connecting external hardware.



Propeller Archives

You'll notice that the ZIP that contains the files for this month has a very specific name; this ZIP was created by the Archive selection of the Propeller Tool > File menu. This is a tremendously useful feature of the IDE; it lets us gather and archive all the files of a project, no matter where the files are located on the system. This makes sharing projects with others a breeze as you are ensured that they will get everything they need. There's also an Archive feature that includes the IDE! With this you can open an archive folder several years from now and know that you've got what you need to recreate that project.

Have fun with your Propeller, and until next time…. Happy Spinning! And yes, we'll be back to working with the BASIC Stamp and SX very soon.