Spinning Up Embedded Control Projects
By Jon Williams

For Nuts & Volts Magazine, May 2009

One of my dearest friends is a gentleman named Cliff Osmond. In addition to being my friend he is my acting coach and, as acting is about life, I learn a lot about life as I spend time with him. He's been around the block a few more times than me and what he has to say is valuable. Not long ago he said that – as an actor – we have to be comfortable with being uncomfortable, because that's when we're really alive. It's an acting lesson and a life lesson, and time to apply it to my embedded control projects. I've become happily comfortable with the SX the last couple years and at the urging of several friends and readers I'm going to delve more into the Propeller. Don't worry, from time-to-time I'll use the SX and BASIC Stamp, but getting out of my comfort zone, at least for a while, is going to be good for me. And fun!

(Re) Introducing the Propeller Chip

The Propeller is the latest in Parallax's line of microcontrollers though it differs from the BASIC Stamp and Javelin in that it is completely custom silicon, designed in house, entirely by Parallax staff. I often refer to the Propeller as a "multi-processor" instead of a microcontroller because it is comprised of eight 32-bit processors (which Parallax calls "cogs") that are controlled by a central "hub". Each cog has its own RAM so it can run independently, and it can communicate and share information with other cogs through shared RAM in the hub. The hub also handles gritty details like the system clock (which all cogs use and keeps them in sync) and providing access to shared resources like the system RAM and ROM tables.

As advanced as it is the Propeller maintains some of the "genetic" material born out of the original BASIC Stamp. For starters it is programmed through a simple serial port; no special tool is required. And like the BASIC Stamp the Propeller's program is stored in an off-board EEPROM (32K). This last point has stuck in the craw of some professional developers as code stored in an off-board EEPROM is easy to read and clone. This issue is being addressed in the next generation Propeller by encoding the EEPROM with an encryption key that is burned into the Propeller (OTP). This encoding will cause the contents of the EEPROM to look like gibberish to any Propeller chip that does not contain the proper key.

Those with BASIC Stamp experience know that running code from an external EEROM can take a real toll on overall speed. The Propeller borrows from another Parallax product, the Javelin Stamp, in that the EEPROM image is moved into the system (hub) RAM on power up. Using internal RAM gives us a tremendous speed boost and frees the EEPROM pins (28 and 29) for other tasks. I tend to use these pins as a general I2C buss as they're predefined that way for loading the Propeller code.

If you're brand new to the Propeller please see my articles in the April, May, and June 2006 issues of Nuts & Volts; digital reprints are available online.

Development Tools

Getting started with the Propeller means we'll need some sort of development platform and having been around a few years now there are a number of offerings from Parallax and several third parties – including anything we design ourselves. I have a few of the Parallax boards that I use when developing my own projects but that's not to say that products offered by others aren't equally excellent; I just don't have experience with them.

Many will start with the Propeller Demo Board shown in Figure 1. This is a nice little board for doing demos (hence the name) and small experiments but may hold you back when you start getting into bigger projects. If this is what you have, great – it has all the connectors you need for the fun stuff like audio, video, mouse, and keyboard, and has several LEDs which lets you do lots of code training without every having to wire anything.

On the other end of the scale – and my favorite piece of Propeller gear – is the Propeller Professional Development Board (PPDB) shown in Figure 2. This is descended from the tremendously-successful Professional Development Board (PDB) that many pros use to develop BASIC Stamp and SX projects. Now I know that some of you will cringe at the PPDB price tag (about $170) but I hope you'll believe me when I tell you it is worth every cent and so much more. As I just stated, I do a lot of development work for EFX-TEK using the PDB and the PPBD is the answer to my wishes for a similarly-equipped board for the Propeller. When I'm developing projects I don't want to go looking for anything except some hook-up wire (I keep a lot of it on my bench); with the PDB and PPDB I can focus code and connections instead of looking for resistors and LEDs.

What if you already have a PDB? You could, as I did prior to the PPDB, add a 3.3v regulator to the PDB breadboard and build a standard Propeller demo circuit (Figure 3) on the breadboard. If you're not used the Propeller but are familiar with I2C you may wonder why Parallax schematics have only one pull-up on the buss – just on the SDA pin. For reasons unknown to me, the Propeller drives the I2C SCL line high and low when transferring the EEPROM contents to hub RAM. After the hub is loaded both pins float and you can use other I2C devices on these pins – just make sure you don't write to EEPROM (I2C device type %1010) address %000 as you could corrupt your program. If you're going to use pins 28 and 29 for other I2C devices in your project be sure to add a pull-up to pin 28. IMPORTANT NOTE: If you decide to connect additional EEPROMs (device type %1010) to pins 28 and 29 you must ensure that they are addressed %001 and higher.

The PPDB is a great tool and yet at some point we'll want to make a project permanent. For my semi-permanent and permanent projects I decided to "liberate" a good idea from another microcontroller platform: the Arduino. The original Arduino has a base platform with the microcontroller, power, programming, and IO connections and what many have done is created various application "shields" that plug into the base board.

In Figure 4 you can see the first version of what I call my Propeller Platform. There is nothing magic at all about this board (see the schematic in Figure 5); I used the ExpressPCB mini-board size and built a standard Propeller circuit on it – with an additional socket for another EEPROM (address %001). The board goes together in just a few minutes using all through-hole parts. After building the board I made some refinements to the design files that you can download from Nuts & Volts: I changed the power switch to a right-angle style and found shorter caps for the power supply that allows a daughter-board to fully seat in the power and IO sockets.

The idea behind the Propeller Platform is to have a known good processor base that can move from project to project as desired – this should help keep the costs of future projects a little more manageable. One final note on the board: to keep the crystal low on the PCB I used two elements from a machine pin socket. You have to be careful removing the plastic but doing that will allow it to seat cleanly in the PCB.

Programming the Propeller

The BASIC Stamp started a revelation in small, high-level microcontrollers bringing the BASIC language that many of us learned as youngsters to the world of embedded control. One would think, then, that being a Parallax product the Propeller would be programmed in BASIC as well. After all, how many PBASIC knockoffs exist today? Too many to count – a tribute to how well

PBASIC performed and was accepted. In the end, though, when Parallax designed the Propeller they looked at other programming languages, especially those designed for coding efficiency, and from that study created as slightly-unique – yet familiar feeling – high-level language called "Spin."

Like PBASIC, Spin is an interpreted language but the difference stops right there. As mentioned earlier Spin byte codes are run from the system RAM instead of an EEPROM so there is a huge speed increase, the processor running Spin tends to run at 80MHz (more on that in a moment), and the Propeller is built on a 32-bit architecture which means it can do a lot of work with very few instructions.

An interesting difference between the Propeller and the other HLL (high-level language) modules on the market is that the Spin was developed in tandem with and specifically for "brains" underneath – most other embedded HLLs are created on top of existing microcontroller products and may in fact not always be as efficient as one would like.

As Spin and the Propeller Assembly language were created at the same time there are very close ties between them which allows Spin – even though it's a high level language – to be very efficient. And it's easy to use. I like Spin because it caters to my whacky desire for neat program listings; Spin actually uses indentation to define structures within the code. For those used to "messy" programming you may have to get used to this, while some coming from high-level scripting languages like Python will find feel right at home.

Connecting Spin & Assembly

An interesting thing about the Propeller is that after reset it defaults to Spin, the high-level language. Even if we want to write the entire application in Assembly we have to use a very simple Spin program to launch it so the connection between Spin and Assembly is important. If I was going to allow myself to fall into old habits I'd stick with nothing but Spin for a while but hey, let's push past our comfort level and learn some new tricks, shall we?

In March I had the opportunity to participate in a webinar with my friend and old Parallax colleague, Jeff Martin. Of particular interest to me was the connection between Spin and Assembly – in terms that I could understand and use in my own projects. While I don't normally believe in simple demo code I'm presenting an updated version of a program Jeff created for me – you can use this to test your own Propeller Platform. This doesn't do a heck of a lot, but clearly illustrates the connection between Spin and Assembly and cooperative work between cogs.

Let's cover the Spin section first.

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

VAR
  long  cmd
  long  paramL1

OBJ
  terminal : "simple_serial"
  myVal    : "numbers"

PUB Main
  cmd := 0
  cognew(@ASM, @cmd)
  terminal.Init(31, 30, 19200)
  myVal.Init

  terminal.tx(12)
  terminal.str(myVal.tostr(incval(100), myVal#DDEC))
  terminal.tx(13)
  terminal.str(myVal.tostr(decval(1000), myVal#DDEC))

  repeat

PUB incval(val) : result
  paramL1 := val
  cmd := 1
  repeat while cmd <> 0
  result := paramL1

PUB decval(val)
  paramL1 := val
  cmd := 2
  repeat while cmd <> 0
  result := paramL1
```

The CON section is what we'll tend to use as a standard: 5 MHz crystal with the PLL cranked up to 16x so we're running the cogs at 80 MHz.

In the VAR declarations we define two longs that will be used to communicate with the Assembly cog. The purpose of our demo is to send a command and value to the Assembly cog; the command will tell the Assembly cog what to do with the value.

In the OBJ section we reference a serial object to send information to a standard terminal (I tend to use HyperTerminal) and an object called *myVal* which allows us to create formatted numeric strings for the terminal.

At the top of Main we start by clearing *cmd* and then launching the Assembly cog with the **cognew** function. Since we're going to launch Assembly code we need provide a pointer to it using the @ symbol with the name of the label used (ASM). The next item in the function is the address of the command variable – as all cogs have access to the system RAM this allows the Spin and Assembly sections to work together. The address of *cmd* will be passed to the Assembly section inside the Propeller using the *par* register.

Okay, let's have a look at the Assembly section.

```
DAT

              org       0

ASM           mov       cmdAddr, par
              mov       valAddr, par
              add       valAddr, #4

WaitForCmd    rdlong    tmpL1, cmdAddr
              tjz       tmpL1, #WaitForCmd

CheckCmd      cmp       tmpL1, #1    wz
        if_z  jmp       #Increment
              cmp       tmpL1, #2    wz
        if_z  jmp       #Decrement

BadCmd        wrlong    zero, cmdAddr
              jmp       #WaitForCmd
```

```
Increment     rdlong    tmpL1, valAddr
              add       tmpL1, #1
              wrLong    tmpL1, valAddr
              wrLong    zero, cmdAddr
              jmp       #WaitForCmd

Decrement     rdlong    tmpL1, valAddr
              sub       tmpL1, #1
              wrLong    tmpL1, valAddr
              wrLong    zero, cmdAddr
              jmp       #WaitForCmd

' ENDASM


cmdAddr       long      0
valAddr       long      0

tmpL1         long      0
zero          long      0
```

If you look at the contents of any programmed microcontroller it would look like a bunch of random values that could be program code or data; it's the location of these values that allows the microcontroller to interpret them as what they actually are. This means, then, that we will write our Propeller Assembly code inside a DAT block of a Spin program. When we launch the new cog this code is copied from the Spin DAT section into the new cog's RAM to run.

The first thing our program does is retrieve the *par* register which is holding the address of the command that we're going to use later. This is copied to a local (inside the new cog) variable called *cmdAddr*. In actual fact we need to know two addresses: the address of the command value and the address of the variable to work on. As we can only pass one long to the Assembly section we placed our variables in order in the Spin section; in the Assembly section we can determine the address of the variable to work on by adding four (four bytes per long and hub memory is always addressed as bytes) to the value in the *par* register; this will be stored in *valAddr*. Using this strategy we an Assembly program can operate on as many system variables may be required.

Here's what the Assembly program is going to do: it will wait for a command from Spin, interpret the command and then do something with the variable located in *valAddr*. The Assembly section knows it has a command when the value in *cmdAddr* is non-zero. A two-line loop at the top will cause this Assembly program to wait using the **tjz** (test, jump on zero) instruction.

Once a command is detected it is tested for one or two – one for increment, two for decrement. Note that Propeller Assembly allows the programmer to determine the affect an instruction has on flags. We're telling the **cmp** (compare) instruction to set the *wz* bit when the result is zero (this signifies a match). After the compare we test the zero flag (**if_z**) and jump to the appropriate routine. The Increment section retrieves the value from *valAddr*, adds one to it, then puts the new value back. The code at Decrement is identical save for the fact that it subtracts one.

Now the important part: the command value in *cmdAddr* is cleared to zero. Let's go see why.

Back in the Spin code you'll see a function called **incval** that is going to be used to increment a value passed to it. The value passed is moved to *paramL1* which the Assembly section tracks in *valAddr*. Then we set *cmd*, which the Assembly tracts in *cmdAddr*, to one to indicate increment. An important reminder here is that the **cognew** function started the Assembly code in its own processor (cog) and it is running concurrent with and yet independent of the cog running the Spin program – it is the sharing of the locations of *cmd* and *paramL1* that lets them work together.

Since both programs are running at the same time we have to store the value first and then change the command. As soon as the command is changed to a non-zero value the Assembly program is going to take action. Remember how we had the Assembly section clear command to zero when finished? We can see now how this is used by the Spin program: after setting the command value a simple **repeat** loop holds the program until the command is cleared before returning the updated value to the caller.

When you run the program your terminal will print two numbers: 101 and 909 – these values are returned from the **incval** and **decval** functions in the Spin program, but the actual incrementing and decrementing took place in the Assembly program.

Okay, I know this program is barely a step up from blinking LEDs but I think it's really important to get a good grasp on the mechanics of connecting Spin and Assembly if we're going to take advantage of all the horsepower the Propeller has to offer. Do take a few minutes to load this program into your development system – whatever it might be – and run it. Better yet, run it and then modify it to do something more. Go beyond your comfort zone; this will make you a better programmer.

Intervalometer Follow-Up

Just a quick note for those that have or are considering building the intervalometer project from the March column: a friend of mine pointed out that one of my favorite places in Los Angeles, All Electronics, has a stick-on IR LED cable that is normally used for controlling VCRs – this makes the IR connection to the camera much more reliable. Look for part number #IR-21.

Another option – and the one I prefer – is an IR Extender from *www.smarthome.com*; the part number is #8170S. This costs more than the unit from All Electronics but is manufactured by them so you never have to worry about supplies, and it's quite a bit smaller so I find it easier to mount to my camera. Either way, using one of these stick-on emitters ensures we won't be missing any important shots with the intervalometer.

That's about it for now – until next time, Happy Spinning!

Bill of Materials

C1–C3	47	Mouser 140-L25V47-RC
C4–C5	0.1	Mouser 80-C315C104M5U
J1	2.1mm	Mouser 806-KLDX-0202-A
LED1	3mm red	Mouser 859-LTL-4221
LED2-LED2	3mm green	Mouser 859-LTL-4231
PGM	0.1 R/A Header	Mouser 517-500-01-36
R1	1K	Mouser 299-1K-RC
R2	300	Mouser 299-300-RC
R3	120	Mouser 299-120-RC
R4-R5	10K	Mouser 299-10K-RC
RST	N.O. pushbutton	Mouser 653-B3F-1002
SW1	SPDT	Mouser 506-SLS121RA04
U1	P8X32A-D40	Parallax P8X32A-D40
U2	24LC256	Mouser 579-24LC256-I/P
VR1	LD1085V50	Mouser 511-LD1085V50
VR2	LD1086V33	Mouser 511-LD1086V33
X1	5 MHz	Parallax 251-05000
S1	40-pin	Mouser 571-1-390262-5
S2-S3	8-pin	Mouser 571-1-390261-2
SV1-SV2	Board Mount Conn	Mouser 517-974-01-04-RK
SPx1-SPx2	Board Mount Conn	Mouser 517-974-01-16
SXR	Machine pin socket	Mouser 506-510-AG90DD1