Putting a Whole New Spin on Embedded Control
By Jon Williams

For Nuts & Volts Magazine, April 2006

When I first saw the ads in Nuts & Volts for the BASIC Stamp 1 (way back in 1993) I didn't believe it; I maintained an "It's just too good to be true." attitude, and actually put-off buying my first BS1 for about six months. Well, for some, that's about to happen again. After a long, and sometimes arduous development cycle, Parallax has produced its first piece of custom silicon: the Propeller chip. No, it's not a microcontroller, it's a heck of a lot more: it's a multi-controller.

The clever guy who created the BASIC Stamp, Chip (yes, that really is his name), has done it again: he's broken the perceived rules of what a small controller should be, and how it should operate. He's long known what he wanted in terms of a small controller, and silicon manufacturing technology and his knowledge of microcontroller design have come to the point where he could make that dream come true. The great thing is that you and I benefit from having a cool new device that will enable us to make our [project] dreams come true – and in most cases with just a bit of code and very simple hardware.

It all started back in 1998. Not satisfied with the state of microcontrollers, and frustrated with the effort required to manage complex tasks on a small micro, Chip set out to create his own. There were only a few rules: it had to be fast, it had to be relatively easy to program, and it had to be able to do multiple tasks without using interrupts – the bane of all but the hardiest of programmers.

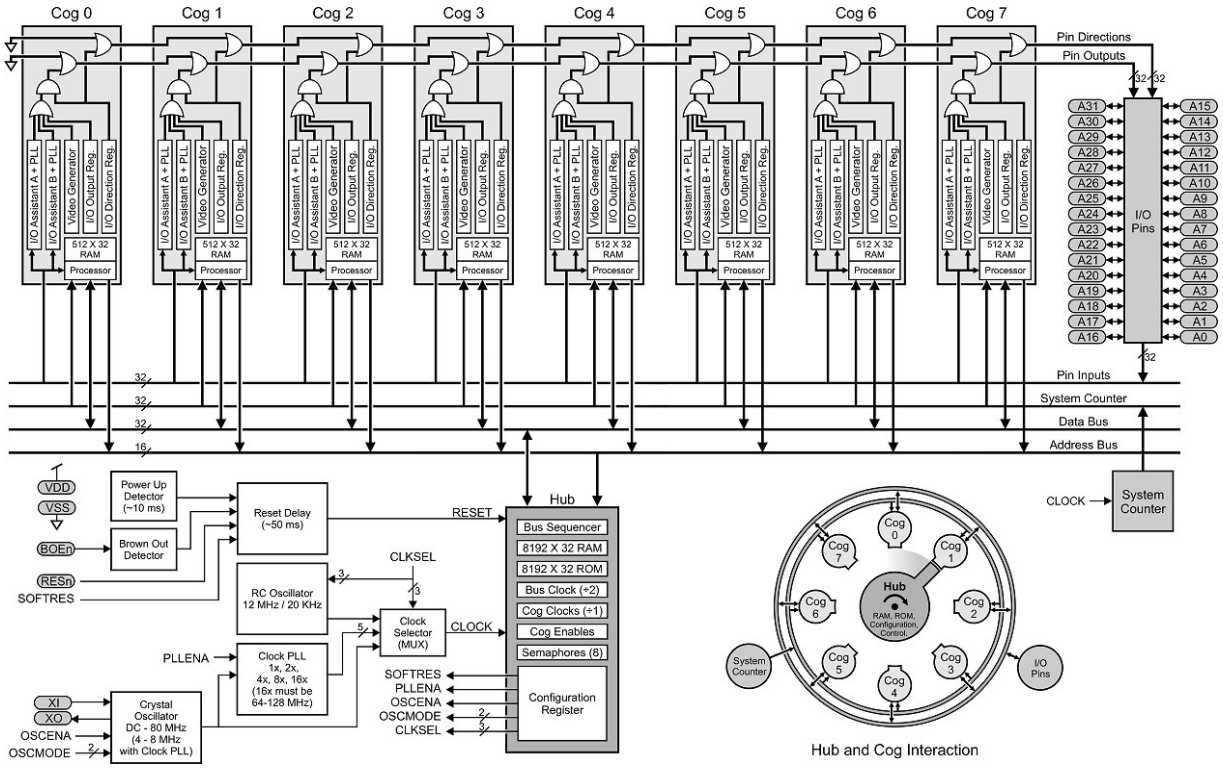
That last rule is the kicker. How does one handle multiple tasks, even those with critical timing requirements, on a small processor that doesn't have interrupts, and do it without making the programming chore a gigantic nightmare? Well, you do what we see happening more frequently in PCs these days: you use more than one processor.

First things first, and the first thing Chip had to do was develop the Propeller processor. After a lot of experimenting, Chip decided on a 32-bit engine. Mind you, this is not something you sit down and do with a handful of discrete components; to create a 32-bit processor from scratch requires special tools. Some of you may have noticed that a few years ago Parallax started offering development kits for high-end Altera FPGAs. Those actually started out as internal tools that Chip used to develop the processor core. The folks at Altera liked them, and they were working well for Parallax internal development, so it seemed like a good idea to make them available to the public.

The USB2SER adapter has similar origins. Another one of Chip's design rules is that the Propeller hardware be easy to program, and not require expensive tools; hence the development of the USB2SER. It took about three years of very devoted work, but in the end the Propeller core emerged and Chip was able to test it full-speed on an Altera FPGA. Once that task was complete, the next step was to bring eight cores together to share a 32K RAM, a 32K ROM, and cooperatively handle 32 I/O pins, and finally to transfer the design from and FPGA to custom silicon. Obviously, this was not a trivial task, but with Chip and his team it finally came together. I can tell you that it was a very exciting day around Parallax when the first Propeller chips arrived and they came to life with one of the video demo programs that are used as a showcase of the Propeller's incredible capability.

Propeller Overview

As I stated above, the Propeller chip consists of eight processor cores (called "cogs") that share access to a 32K ROM, a 32K RAM, 32 I/O pins, and other system resources. A system manager, called the "hub" takes care of keeping the cogs coordinated. Figure 1 shows a graphic depiction of the Propeller architecture.



Shared resources come in two types: 1) common, and 2) mutually exclusive. The I/O pins, for example, are common in that any cog can access any pin at any time. This is possible because the direction registers for each cog get are OR'd together before the final I/O stage. If any cog makes a pin an output, it will be an output. The same is true for the output state: if any cog that defines a pin as an output makes the pin high, that pin will go high. Note that inactive cogs or cogs that define a pin as an input have no effect on the output state of any pin.

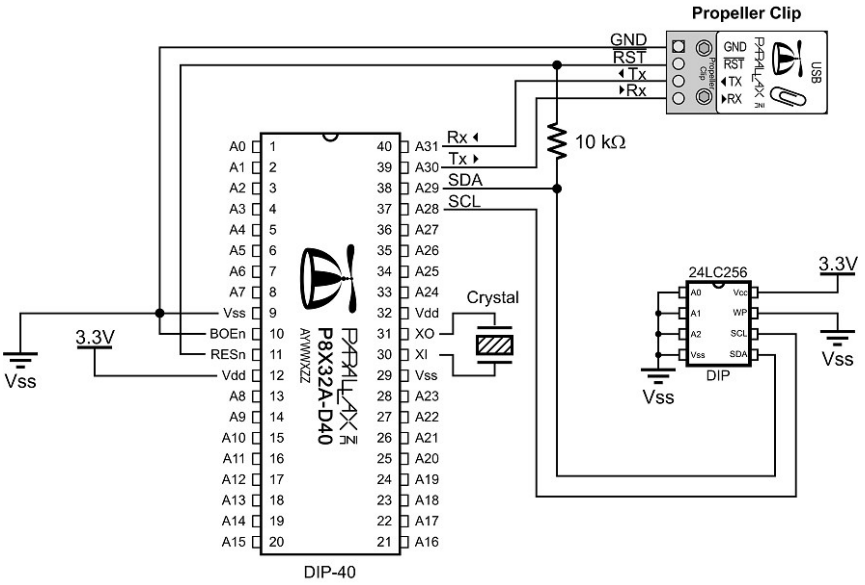
Another common resource is the system clock; this actually drives all cogs and allows them to be synchronized, even though they run independently of each other. The system clock also drives a global register called the System Counter. This 32-bit, read-only counter increments every clock cycle, and can be read by any cog via the **cnt** register. A command called **waitcnt** can be used by a cog to wait for a specific value in the **cnt** register; this is handy for creating custom delays.

The system RAM is a mutually-exclusive resource which means that the hub manages access to it, allowing any cog access, but only one at a time. This prevents one cog from clobbering a value being written by another. The neat thing about using a shared RAM is that it's an easy way for cogs running different processes to share information. One might write an application where a cog is devoted to monitoring a serial stream and parsing data from it, placing that data in a shared RAM location. Processes running on other cogs can read the shared RAM as required to use that data.

Getting Connected

The simplest practical Propeller system consists of a Propeller chip, a 24LC256 EEPROM, a 3.3 volt power supply, and a programming connection. If you build your own system you either use the USB2SER connector, or a derivative of it called the Propeller Clip. Both devices carry the same signals, although they have different layout and connection schemes, so be mindful of that in your designs. The Propeller Clip is designed to clip right to the edge of a PCB, requiring nothing more than a set of pads.

If you look at the simple system schematic in Figure 2 you'll see that four of the Propeller's 32 I/O pins serve a special purpose at boot up. Pins A30 and A31 provide the serial connection to the programming environment (called the Propeller Tool). If a programming host is detected the Propeller will converse with it to identify itself and possibly download a new program into the global RAM, and if commanded to do so will also transfer that program into the EEPROM.



If no host is detected the Propeller looks for the 24LC256 EEPROM (at address %000) connected to pins A28 (SCL) and A29 (SDA). If the EEPROM is detected the contents are transferred to the Propeller RAM and the program for cog #1 is started. Note that you can in fact do development work on a Propeller chip without an EEPROM as the Propeller Tool allows you to download and run a program right from RAM. This is very convenient for testing new code without overwriting what already exists in the system EEPROM. Just remember that a Propeller system without an EEPROM, and that is not connected to the programming tool will do nothing but put itself to sleep.

And for those of you wondering if you can hang other I2C devices from pins A28 and A29… of course you can. Just be sure you don't write anything to the Propeller's EEPROM or else you could corrupt it and prevent the Propeller from running properly after the next reset.

Let's stick with the idea of a typical system as on the Propeller Demo Board shown in Figure 3. When the system starts the user programs will be transferred from EEPROM to the Propeller RAM and the program for cog #1 is started. The Propeller can be programmed in two languages: 1) Spin, its high-level language, and 2) Propeller assembly. That said, even a pure assembly program gets started with a tiny bit of Spin code to make sure that things take off properly.

Objects in the Machine

Spin is an object-based language, which provides structure and code reusability. This is good for us mere mortals, as advanced programmers like Chip can write cool modules like the video driver and floating point library, and all we have to do is include them into our programs! How great is that?

As stated earlier, Propeller objects can be programmed with Spin and, optionally, Propeller Assembly code. Regardless of the programming mix, objects are stored as .spin files, and each .spin file is a discrete object. Like other object-oriented languages, one object can include, and indeed be built from other objects. When one object is created using others, it becomes known as the "top" object.

When a program is written it can be compiled and downloaded directly to RAM for testing or, optionally, it can be downloaded to RAM and then transferred to the Propeller EEPROM. Remember that this last step is required for stand-alone operation. The last program loaded into the Propeller EEPROM will be loaded and run after the next reset if no programming connection is present.

You might wonder what happens if the same object is used more than once in a Propeller project. Not to worry, the compiler optimizes the code space removing any redundant code with its object distiller engine. Once the code is downloaded and run the Propeller will load the Spin interpreter from its internal ROM to any cog that needs it (i.e., that is running Spin code). Not all cogs will require the interpreter. You can, for example, have one cog load an assembly program into another cog and start it. This is useful when you absolutely need the most performance out of a cog (the video module is an example where this is done). In a situation where one cog is used to launch an assembly program into another, the "launcher" cog is freed-up as soon as the other program is started.

Okay, we have to learn to walk before we can run (especially on eight legs!) so let's do the ubiquitous "Hello, World!" LED blinker program that we always start with when we get something new. Here's the complete listing:

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

  Led = 16

VAR
  long delayTime

PUB BlinkLED

  dira[Led] := 1

  repeat
    outa[Led] := !outa[Led]
    delayTime := cnt + 8_000_000
    waitcnt(delayTime)
```

The truth is, we could have made this simpler but this demo let's us show off some neat hardware features of the Propeller, and gives you an idea of how the code is structured. You'll notice right away that things are organized into blocks; in this program there is a CON (constants) block, a VAR (variables)

block, and a PUB (public) block. There are other block types, including OBJ (object), PRI (private), and DAT (data) blocks. One of the [many] nice features of the Propeller Tool is that each block gets its own color-coded background. And in those cases where you have to consecutive blocks of the same type, there are light and dark versions of each background color.

In the CON block we start by setting the system clock mode (to external crystal) and enable the PLL (phase locked loop) to multiply the clock frequency by eight; this is done by setting switches in the hub where the system clock is controlled. That's right, the Propeller PLL multiplies the external clock by 16x and provides us taps at 1x, 2x, 4x, 8x, and 16x. So what we've done is taken our external 5 MHz crystal and wound it up to 80 MHz! Note that 80 MHz is the maximum internal speed, so don't get the idea that you can drop a 20 MHz crystal onto the Propeller and wind it up to 320 MHz – this is not going to work. Remember that 16x is the maximum; if we wanted to back the internal clock down to 20 MHz all we have to do is change the PLL tap to **pll4x**.

The final step in the constants definition block is to set a pin to use for the LED; in this case we're selecting A16 (which has an LED right on the Propeller Demo Board).

For this simple program we're going to use just one variable, and we'll make it the native Long (32 bit) type, as this is the same size as the system counter that we're about to make use of. The Propeller can also use Word (16 bit) and Byte (8 bit) variables. Just be aware that the 32-bit processor handles 32-bit variables most efficiently.

The working part of the program is contained in the PUB section. Public blocks can be "seen" by other objects in a multi-object project – private blocks cannot. We will always have at least one public block in an object. The first step in our blinker program, then, is to make the LED control pin an output. This line:

```
  dira[Led] := 1
```

… is equivalent to the PBASIC line:

```
  DIRS.LOWBIT(Led) = 1
```

Some of you will recognize the assignment operator that is borrowed from Pascal. Spin is a really cool language, taking advantage of features found in other popular programming languages. Block definition by indenting – something found in Python – is one of those features.

The rest of the program forms an infinite loop using just the **repeat** keyword. Note that the three lines that follow **repeat** are indented to the same level; this identifies them as a code block. I know that a lot of you think I'm wacky for my "neatness counts" campaign when it comes to code formatting; well, here's why. If we don't indent properly the program won't run properly. The great thing is that by using indenting instead of block terminators we don't have to type as much. Chip loves efficiency and this is one of the many areas related to the Propeller where it shows up.

Okay, let's see how the program works. The first line in the indented block inverts the state of the output pin to toggle the LED. The second line reads the current value of the system counter (in a global register called **cnt**) and adds eight million (zoiks, that's a big number!) to it, saving the new value in the variable *delayTime*. The heart of the loop is with the **waitcnt** instruction that will hold this cog's program until the value of the system counter matches the value passed to it. So with a system clock of 80 MHz, waiting eight million cycles causes a 100 millisecond delay (8 divided by 80 is 0.1).

Yes, I know it seems a little silly to blink an LED with a multi-controller that can run at up to 80 MHz, but we have to start somewhere. Please believe me that we haven't even begun to scratch the surface of the Propeller, or even the Propeller Tool. One thing that I will point out is that the Parallax True Type® font includes schematic symbols – yes, we can even draw simple diagrams right in our listings! If you look carefully, you can see the LED schematic in the listing shown in Figure 4.

And for those of you worrying about the BASIC Stamp or SX microcontroller going away – the answer is, "No, of course not." Both are doing well, and now they have a big brother. Parallax recognizes that customers come in all shapes and sizes and desires for what they want in a product; the Propeller simply expands the line for those that are looking for a little (okay, a lot) more horsepower for their projects.

If your head isn't spinning (sorry, couldn't help myself) by now then you're made of tougher stuff than me! Next month we'll take it to the next step by using some of the prewritten objects provided by Parallax and create one of our own. It may take us a while to master the Propeller, but I can promise you that we will have a lot of fun along the way.

Until then – Happy Spinning!