Parallax™ Propeller Binary Format

Disclaimer: I am not affiliated with Parallax. This page is also unrelated to my employer and professional life.

When I discovered the Parallax Propeller 8-core microcontroller, it sounded perfect for my projects. However, like nearly everything out of Parallax ever, the tools are Windows-only (and apparently written in x86 assembler, so the obvious "laziness" excuse apparently doesn't apply).

Now, I use Windows for two things:

1. testing websites in IE, and
2. running tools from short-sighted, dense companies.

In the second case, I do it just long enough to generate good output cases for reverse engineering. In both cases, XP lives in a little VM session where it belongs, and where I can rewind to a snapshot when I make the mistake of connecting to the internet.

So, test cases in hand, I set to work. The data below is the result. (Chip Gracey, the patient and benevolent designer of the Propeller, also sent me some info, but I cleanroomed this because it's more fun that way. Also, Chip's sample code turns out not to work.)

The `.binary` file: overview

The `.binary` files generated by the Propeller Tool are simply memory images for the Propeller, a binary version of the `.hex` files us embedded devs are so accustomed to. The files are written to EEPROM directly after verification, with only a bit of data added (see below). The files can be loaded directly into RAM by the Propeller's bootloader, or from EEPROM; either way, the data appears starting from $0000.

A quick check with the Propeller Manual tells us that locations 0x0000 through 0x000F are the "Initialization Area." The manual notes that we might be interested in the longword at 0x0000 — which holds the initial clock frequency — and the byte at 0x0004, which configures the clock generator.

And that's it. The manual stops there. (Some manual.)

The Initialization Area

Here is the layout of the first 16 bytes of an image. Keep in mind that, because Parallax is filled with Intel fetishists, all multibyte quantities are little-endian.

Address	Length	Function
0x0000	4	Initial clock frequency, in Hz
0x0004	1	Initial CLK register contents (see manual p.28)
0x0005	1	Checksum (see below)
0x0006	2	Program base address. Must be 0x0010 (the word following the Initialization Area).
0x0008	2	Variable/stack space base address. Should generally point just beyond the end of the data in this file (and the Propeller Tool loader appears to enforce this). Thus, this is effectively the size of the image, in bytes. Before the image is written to EEPROM, the two longwords following this address are set to 0xFFF9FFFF. Bit twiddling suggests that the SPIN stack routines expect this to be there, perhaps for underflow detection.
0x000A	2	Stack base address. Should generally point eight bytes past the variable base address (above).
0x000C	2	Address of initial SPIN instruction to execute.
0x000E	2	Address of initial top-of-stack. Seems to always be four bytes *above* the stack base, leaving a single word of zeros.

Prop Tool appears to generate eight more bytes here, which I'm currently reproducing as voodoo. More as I figure it out.

The Checksum Algorithm

At $0005 lives a checksum byte. This byte is calculated so that all bytes in RAM sum to 0, modulo 256.

Notice I didn't say all bytes in your image. Nooooo, that'd be too easy.

I noted above that some loader process seems to write 0xFFF9FFFF, twice, at the start of variable/stack space. *This phantom data must be included when computing the checksum, even though it doesn't appear in your image.* It *does* appear in an "EEPROM file," which is a complete image of the 32kb RAM.

If you're lazy, like me, you can take advantage of the fact that the two longs happen to sum to 0x14, modulo 256. So, here's some pseudocode for computing the checksum.

```
bytes[5] = 0
sum = 0
for each b in bytes:
  sum = sum + b

bytes[5] = (0x14 - sum) & 0xFF
```

A generic loader sequence

After loading an image, Propeller jumps into some SPIN code. I have no interest in SPIN; it's a slow, crufty language, whose innards are entirely undocumented, and whose only compiler is embedded in a Windows-only IDE. However, if we want to code in a civilized language, we'll still have to use SPIN to bootstrap.

Here's my stab at a generic loader for Propeller machine code. It works for some basic tests, but I won't really have tested it until I integrate it into my assembler.

The loader uses a five-byte SPIN sequence at the *end* of your image. It configures the SPIN interpreter to execute this code, which loads a cog's worth of code starting at 0x10 (the base of the image) to replace the SPIN interpreter. From this point forward, you're in pure machine code. (Your machine code is free to overwrite the SPIN code once it bootstraps.)

Remember that the SPIN interpreter will write two 0xFFF9FFFF longs after the contents of your image. Once your machine code comes up, you can overwrite these longs, but you must make sure to leave space for them on load.

On to the loader:

```
0000  00 B4 C4 04  6F cc 10 00  bb bb ss ss  nn nn dd dd
0010  ...machine code and data goes here...
nnnn  35 37 03 35  2C [ pad with zero to nearest longword]
```

The variables:

- `cc` is the file's checksum, generated as described above.
- `bbbb` is the length of the image, in bytes.
- `ssss` is `bbbb`+8.
- `nnnn` is the address of the SPIN-code trailer, which is bold in the listing above..
- `dddd` is `ssss`+4.

The key here is the SPIN code fragment, which goes after your machine code and data. I'm working on reverse engineering the fragment, since the SPIN interpreter is undocumented. Here's my info so far:

1. `35` pushes a literal 0 longword onto the stack. (Thanks, Chip.)
2. `37 03` pushes a literal 0x10 longword onto the stack, the address of our machine code.
3. `35`, literal 0
4. `2C` invokes COGINIT, which takes three longword arguments from the stack.

(For more, see my SPIN bytecode page.)