sphinxcompiler Documentation

mpark@sphinxcompiler.com

Introduction

Sphinx is a Propeller-based compiler for the Spin language (including Propeller assembly language). It consists of a suite of programs. Some of the programs constitute the compiler itself, the rest support the compiler by performing the functions of a minimal operating system.

Compiling and linking: a SOB story

The process of building a program with Sphinx is divided into two steps, compiling and linking. Compiling transforms a Spin source file into a Spin Object Binary (SOB) file. Linking converts a SOB and its sub-SOBs into an executable binary image.

Compiling

The Sphinx "compiler" proper actually consists of two programs, lex.bin and codegen.bin, but they work so closely together that they are best considered a unit. Lex.bin reads a .spn file, tokenizes it, and produces an intermediate .tok file. Codegen.bin reads the .tok file, parses it, and generates object code which it saves in a .sob file.

A SOB contains the Spin bytecodes for the object's PUB and PRI methods. In addition, it contains a list of the object's *imports* (sub-objects) and its *exports* (constants and PUB method signatures). A SOB is the compiled essence of an object. As far as Sphinx is concerned, a SOB contains the same information as its corresponding Spin file, just in convenient binary form.

The compiler only compiles a single .spn file at a time. If the .spn file contains a sub-object, the compiler reads the .sob file for that sub-object and retrieves its exported information. This gives the compiler enough information to compile the .spn file without having to compile additional .spn files.

This also means that programs have to be compiled from the bottom up. That is, sub-objects have to be compiled before any containing objects are compiled.

Consider a simple program that prints "Hello, world" on the screen. The top

Consider a simple program that prints "Hello, world" on the screen. The top object, *hello*, contains a sub-object, *tv_text*. *Tv_text* in turn contains sub-object *tv*. When you compile hello.spn, the compiler will need to read tv_text.sob, so you have to have compiled tv_text.spn beforehand. Similarly, before compiling tv_text.spn, you must compile tv.spn to produce tv.sob. So to compile *hello* from a standing start you must issue these commands:

```
c tv
c tv_text
cl hello
```

Of course, subsequently you will not have to compile all three objects every time, just the objects that change.

Linking

You invoke the linker (link.bin) with the name of the top-level object. The linker reads the top-level SOB and goes through its list of imports (in other words, its sub-objects). The linker recursively reads the sub-SOBs and their sub-SOBs.

Once all the SOBs are read, the linker determines how it will lay out all the objects' bytecode in memory and adjusts the various inter-object pointers so that they all refer to one another correctly. Then it writes an executable binary image (.bin file).

Timestamps

Consider the following situation:

1. sub.spn is compiled, producing sub.sob.
2. top.spn is compiled using sub.sob, producing top.sob.
3. sub.spn is modified and recompiled, producing a new sub.sob.

Now the top SOB is based on an old version of the sub-SOB. When the objects are linked, the resulting executable file may well fail because of that version mismatch.

In order to detect such version mismatches, Sphinx maintains a 32-bit "timestamp" in a file named timestmp.d8a. Every time you compile an object, Sphinx increments the timestamp and stores it in the .sob file. (Of course an actual timestamp could be used, but Sphinx takes this approach so as not to require a real-time clock.)

The linker compares timestamps and warns if any object being linked is out of date with respect to a sub-object.

Note that this timestamp mechanism cannot detect when a .sob file is out of date with respect to its .spn counterpart. That is, Sphinx cannot tell if a .spn file has been modified after being compiled. If you edit a source file, it is your file has been modified after being compiled. If you edit a source file, it is your responsibility to remember to compile it.

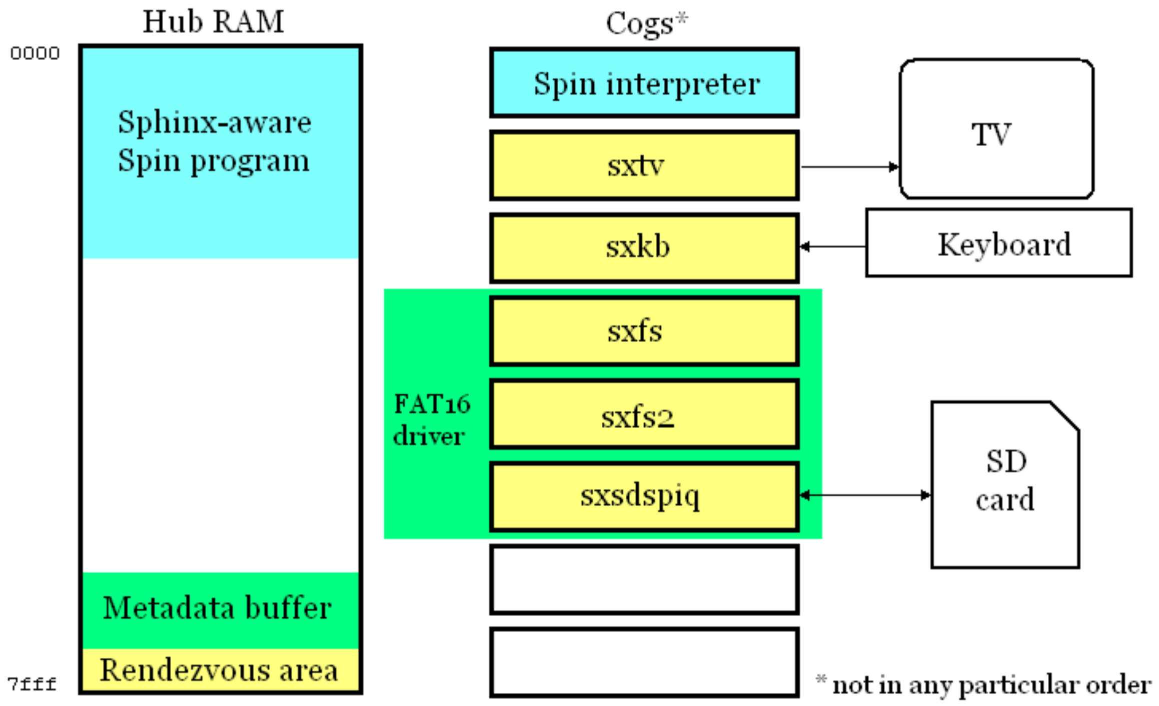
The Sphinx operating environment

Sphinx I/O is performed by device drivers that reside entirely in the Propeller's cogs, taking up almost no hub memory. For example, the video driver runs in a cog and maintains its screen buffer (13 lines of 40 characters) in its cog memory. It communicates with the rest of Sphinx through a single long memory location (known as a *rendezvous*, to use terminology borrowed from fsrw) at the top of hub RAM space.

Similarly, the keyboard driver occupies no hub memory except for a rendezvous location.

The SD card driver is somewhat more complicated. It takes up three cogs. One is a low-level SD SPI driver (fsrw's sdspiqasm, lightly modified). The other two cogs implement a very barebones FAT16 file system that provides basic read, write, and execute functionality. Programs can open multiple files but must allocate some hub memory for each file. Communication with the file system is through several rendezvous locations. The file system requires a 512-byte buffer for metadata.

There are simple interface objects that hide the details of using the drivers.



Sphinx memory and cog usage

Sphinx.bin installs the drivers when it first boots up. From that point on, the drivers remain resident and running even as different programs execute in hub memory (as long as they are Sphinx-aware programs). For example, if you run link.bin by typing "link" at the command prompt, what happens is this:

1. Sphinx.bin uses the Sphinx file system to load link.bin into hub memory and execute it while leaving all cogs running.
2. Link.bin runs, using the Sphinx drivers to perform TV, keyboard, and file I/O. Note that link.bin does not install any drivers.
3. When link.bin terminates, it uses the file system to load and execute sphinx.bin. Sphinx.bin determines that the drivers are already running and does not re-install them. It prints a prompt and awaits the next command.

Sphinx-aware programs do not have to carry device driver code inside themselves; they can rely on Sphinx drivers. Sphinx-aware programs can execute other programs, typically sphinx.bin, but not necessarily (for example, lex.bin executes codegen.bin so that the two parts of the compiler run seamlessly). Running sphinx.bin puts up a command prompt immediately after a program ends, without the long delay that a reset would cause. This, coupled with the fact that the display accumulates the output from each program (rather than clearing between each program), gives the convincing illusion of a traditional command-line shell.

Sphinx can also execute regular (that is, non-Sphinx-aware) programs by performing the equivalent of a reset before running the program. This shuts down the driver cogs. The program must perform its own I/O. When the program finishes, a reboot will presumably occur, either programmatically or manually. At that time, Sphinx will load in from EEPROM and install the Sphinx drivers.