SPINDLE

Operator's Handbook

Contents

Introduction . . . 5
Features . . . 5
License . . . 6
Building from source . . . 6
Acknowledgements . . . 6

I Loader and decruncher 7
1.1 Basic operation . . . 9
1.2 The load script . . . 10
1.3 Bank selection . . . 11
1.4 A humble trackmo . . . 12
1.5 Running Spin . . . 15
1.6 Labels and seeking . . . 16
1.7 Multi-side trackmos . . . 17
1.8 Directory art . . . 18
1.9 Advanced features . . . 18

II Effect linking framework 21
2.1 Overview . . . 23
2.2 Lifecycle . . . 25
2.3 The effect header . . . 26
2.4 Driver . . . 28
2.5 Early setup code . . . 29
2.6 Overlapping lifecycles . . . 30
2.7 Making a chain . . . 32
2.8 Music . . . 33
2.8.1 Installing a player . . . 33
2.8.2 Synchronisation . . . 34
2.9 Visualisation . . . 36
2.10 Labels, jumps, and loops . . . 39

2.11 Flip-disk parts 41
2.12 Streaming data 42
2.12.1 Streaming graphics 42
2.12.2 Streaming music 44
2.13 Effect debugging 45
2.14 Using the reserved areas 46
2.15 Methodology 46

Introduction

Congratulations on your discovery of Spindle, the integrated linking, loading, and decrunching solution for trackmos and other data-intensive applications for the Commodore 64 and Commodore 1541 platform.

The core of Spindle is a cutting-edge IRQ loader featuring extremely fast scattered loading and decrunching, a small RAM footprint, and state of the art serial transfer routines.

On top of this, Spindle provides an optional linking framework that automates much of the tedious work traditionally associated with trackmo development. By hiding the details of the storage model, Spindle allows the demo coder to focus on effects, transitions, and flow.

Features

- All-in-one scripted linker, cruncher, and D64 disk image writer.
- Fully documented.
- World class loading speed.
- Tiny memory footprint: One resident page, one buffer page, and five zero-page bytes. Buffer and zero-page areas are free to use between loader calls.
- Choice between a high-level demo linking framework and a low-level API with direct control over the loading process.
- Transparent and fast loading to RAM underneath the I/O registers.
- Works with other drives present on the IEC bus. Other drives can be 1541, 157x, and 1581. To ensure reliable operation, the serial transfer speed is slightly reduced when other drives are present.
- Support for multiple disk sides, with knock-codes to prevent compo accidents. Trackmos can be resumed from any side.
- Seekable load scripts, for all your looping and branching needs.
- Streaming: Refill graphics and music buffers within and across effects.
- Debugging tools: Effects can run standalone with simulated playroutine timing and memory boundary checks.
- Reads directory art in many formats: PETSCII, D64, Screen RAM, Charcoal.
- Fully open source, with prebuilt binaries for Linux and Windows.

- Extensively tested, including on THCM’s highly picky drive.
- Innovative hybrid scattered/linear decrunching algorithm.
- Advanced drivecode features: Headerless scanning, triple sector buffers, and GCR decoding on the fly.
- Invented Here Stepping™ keeps the data flowing during track changes.
- Reset detection.
- 40-track mode.

License

Go ahead and use this in your demos! You can include the spindle logo (`examples/pefchain/spindlelogo`) if you like, but you don’t have to. I would appreciate some credit, e.g. “Loader by lft”.

Please refer to the file `COPYING` for the formal stuff.

Building from source

Spindle is distributed with precompiled binaries for Linux (i386 and x86_64) and Windows.

To build the toolchain from source you need the `xa65` cross-assembler, a C compiler, and `make`. Under Linux, enter the `src` directory and run `make`. To cross-compile the Windows binaries on a Linux system, you need `mingw32`, and then you can run `make win`.

Acknowledgements

I would like to thank the following individuals for inspiration, suggestions, bug reports, and testing: **Bitbreaker**, **Krill**, **Ninja**, **Raistlin**, **Sparta**, and **THCM**.

Greetings to Spindle users all over the world, including Arise, Atlantis, Atwoods Studios, Bonzai, Chorus, Defame, Delysid, Desire, The Dreams, Extend, Genesis Project, Samar Productions, and Triad.

Part I

Loader and decruncher

Spindle consists of two layers: a high-level linking framework implemented on top of a low-level IRQ loader and decruncher.

The Spin tool offers direct access to the lower layer, giving the programmer full control over the loading and decrunching process, and the ability to specify exactly what to load and when. Spin could be seen as a slimmed-down alternative to the full linking framework described in Part II.

1.1 Basic operation

> The White Rabbit put on his spectacles. “Where shall I begin, please your Majesty?” he asked.
>
> “Begin at the beginning”, the King said, very gravely, “and go on till you come to the end: then stop.”
>
> —Lewis Carroll, *Alice’s Adventures in Wonderland*

At the heart of every Spin-based trackmo is the load script. Spin parses the script at compile-time, reads and crunches all files that it refers to, and bakes everything into a disk image in D64 format. On the disk image is a small program file—a *bootstrap*—that installs the loader and drivecode, loads the first set of files from the script, and jumps to an address of your choice.

Once the trackmo is up and running, it normally invokes the loader in a predetermined sequence. For each loader call, Spindle loads the data that was specified in the script at that point.

The loader occupies one page of C64 memory at all times, called the *resident page*. By default this is page 2 (i.e. 200-2ff, just after the stack). Because of the predefined load script, there is no need to pass any parameters to the loader at runtime. All you need to do is `jsr` to the beginning of the resident page (i.e. `jsr $200` when using the default location).

During loader calls, Spindle overwrites a separate buffer page (300-3ff by default) and five consecutive zero-page locations (f4-f8 by default). Like the resident page, these areas can be moved by passing commandline options to Spin. Between loader calls, you can use the buffer page and zero-page area as you please; only the resident page needs to be preserved.

Starting with version 3.0 of Spindle, it is also possible to break the predefined sequence by seeking to arbitrary points in the load script. This can be used to create looping or branching demos and is described in detail in a later section.

1.2 The load script

Let us have a look at a typical load script. It is taken from the complete Spin example included in the Spindle distribution (`examples/spin`).

```
demo.prg
Specular_Highlight.sid          1000    7e

pic1.kla                        6000    2       1f40
pic1.kla                        4000    1f42

pic2.kla                        6000    2       1f40
pic2.kla                        4000    1f42

pic3.kla                        6000    2       1f40
pic3.kla                        4000    1f42
```

Blank lines are used to divide the script into paragraphs, where each paragraph corresponds to a single loader call, also known as a *job*.

A job is made from one or more data chunks, listed in a four-column format. The first column specifies a filename, which may optionally be enclosed in quotes ("). This is the only mandatory column. The second column specifies a loading address. If the loading address is zero or omitted, the first two bytes of the file are used. Loading to shadow RAM (in the d000-dfff range) works transparently.

Earlier versions of Spin also allowed you to load directly into I/O registers by adding an exclamation mark after the load address. This rarely used feature has been removed for performance reasons.

The third column specifies a byte offset within the file. Spin will seek to this offset before reading from the file (also before reading the load address, where applicable). The fourth column specifies the number of bytes to read.

All numbers are assumed to be in hexadecimal notation by default. Decimal numbers are accepted with a + prefix.

In the example script there are four paragraphs, corresponding to four loader calls. The first call is implicit, because it will be made automatically by the bootstrap program. This loads some code from `demo.prg`, to the address specified in the file. Additionally, it loads a SID tune to address 1000, starting at byte offset 7e in the file (right past the PSID header). The remaining paragraphs will be used to load three different pictures in Koala format.

After making the first loader call, the bootstrap needs to know where to jump. You can specify this with the `-e` commandline option to Spin, or you can let it fall back on default behaviour, which is to take the load address from the first file in the first paragraph. In the example, `demo.prg` loads to address 800, so the bootstrap program will jump to the code at that address.

However, before we look into `demo.prg`, we need to consider some practicalities.

1.3 Bank selection

When you are working with Spindle, VIC bank selection must be done using dd02. Your code is not allowed to touch dd00. Use the following constants to select VIC bank:

dd02	VIC bank
3c	0000-3fff
3d	4000-7fff
3e	8000-bfff
3f	c000-ffff

During initialisation, Spindle disables CIA interrupts, sets the interrupt-disable flag (`sei`) and writes 35 into the bank selection register at address 1. You should leave it at 35 when making loader calls. You may change it as you please from within your interrupt handlers, as long as you restore it afterwards.

The loader will only ever write 35 or 34 to address 1, and it will only write 34 when it has been explicitly asked to load into shadow RAM (in the range d000–dfff). Thus, you only need to save and restore this register from your interrupt routines if they will execute during a loader call that involves shadow RAM.

To save a few cycles, you can use the following trick instead of preserving the value at address 1:

```
interrupt:
        dec     0       ; go from 2f to 2e
```

```
        ; ...interrupt code goes here...

        lsr     $d019   ; acknowledge the raster irq
        inc     0       ; back to 2f
        rti
```

By clearing the LSB of the data direction register, you are forcing the corresponding bank selection bit high. Because Spindle only ever writes 34 or 35 to address 1, what comes out of the port during the interrupt routine will always be 35.

1.4 A humble trackmo

We will now take a look at the source code for `demo.prg` (listed below). This is a very small slideshow demo. First, it initialises the SID tune that was also loaded as part of the first, implicit, loader call. Then it sets up a raster interrupt to call the playroutine.

After setting up the music player and selecting the proper graphics mode and VIC bank, the first explicit loader call is made. In accordance with the script above, this will load the first Koala picture. The bitmap is loaded (and decrunched) at 6000, and the video matrix, colour RAM and background colour are loaded at 4000. Next, the demo proceeds to copy the loaded colour data into colour RAM and store the desired background colour in d021.

The demo waits for space, loads the next picture, waits for space again, and loads the third and final picture. After this, the script is done, and calling 200 again would result in undefined behaviour.

Between loader calls—including after the final load operation—Spindle can detect when the C64 reboots. When that happens, the drivecode will automatically uninstall itself and reset the 1541. You can also trigger this behaviour manually by writing 0 to dd00.

```
; This is an example demo bundled with Spindle
; linusakesson.net/software/spindle/

        .word   entry
        * = $800
entry
        ; Call music init.

        lda     #0
```

```
        jsr     $1000

        ; Set up raster interrupt.

        lda     #$3b
        sta     $d011
        lda     #$ff
        sta     $d012
        lda     #$01
        sta     $d01a
        lsr     $d019

        ; Install IRQ handler to call playroutine.

        lda     #<irq
        sta     $fffe
        lda     #>irq
        sta     $ffff
        cli

        ; Switch banks so we can watch the loading process.

        lda     #$3d
        sta     $dd02
        lda     #$08
        sta     $d018
        lda     #$18
        sta     $d016
        lda     #$0
        sta     $d020

        ; Load the first picture.

        jsr     nextpicture

        ; Wait for space, then load the next picture, etc.

        jsr     wait4space
        jsr     nextpicture
        jsr     wait4space
        jsr     nextpicture

        ; All done.
        ; The drive will be reset when ATN is released (e.g. at system reset).

        jmp     *

nextpicture
```

```
        jsr     $200    ; Call the loader.

        ; Since Spindle 3.0, we cannot load directly into colour RAM, because
        ; that wouldn’t work with in-place decrunching. So we load the colours
        ; just after the vm data, and copy them into place.

        ldx     #0
copy
        lda     $4000+1000,x
        sta     $d800,x
        lda     $4100+1000,x
        sta     $d900,x
        lda     $4200+1000,x
        sta     $da00,x
        lda     $4300+1000,x
        sta     $db00,x
        inx
        bne     copy

        lda     $4000+2000
        sta     $d021
        rts

wait4space
        lda     #$ff
        sta     $dc02
        lsr
        sta     $dc00
        lda     #$10
        bit     $dc01
        beq     *-3
        bit     $dc01
        bne     *-3
        rts

        ; A simple IRQ to call the playroutine and acknowledge the interrupt.
irq
        pha
        txa
        pha
        tya
        pha
        jsr     $1003
        pla
        tay
        pla
        tax
        pla
```

```
        lsr     $d019
        rti
```

1.5 Running Spin

Armed with the above script, `demo.prg` and the three Koala pictures, we are ready to run the Spin commandline tool. This can of course be done manually or automated with a build system (which could be as simple as a batch file).

Since the example script makes use of relative filenames, we have to navigate to the directory that contains the files. Then it is a simple matter of:

```
spin script
```

This creates a disk image with the default output filename, `disk.d64`. While Spin uses its own D64-compatible storage format, it respects and coexists with the native Commodore disk structure, so demo disks can be adorned with noters and other auxiliary files using standard tools like `c1541`.

You can use the `-o` commandline option to change the output filename, and `-r`, `-b`, and `-z` to configure the resident page, buffer page, and zero-page area, respectively. Thus:

```
spin -o demo.d64 -r c -b d -z 2 script
```

would tell Spin to create a file called `demo.d64`, to place the resident loader at c00 and the buffer page at d00, and to use zero-page locations 02-06.

Normally, Spin will run rather quietly, only reporting the number of free blocks remaining on the disk. But you may find it illuminating to enable verbose output by specifying the `-v` option one or more times. For the example above, `-v` provides the following summary:

```
0800-0895 "demo.prg": 150 bytes crunched to 141, ratio 94.00%.
1000-35d6 "Specular_Highlight.sid": 9687 bytes crunched to 5405, ratio 55.80%.
6000-7f3f "pic1.kla": 8000 bytes crunched to 5480, ratio 68.50%.
4000-47d0 "pic1.kla": 2001 bytes crunched to 936, ratio 46.78%.
6000-7f3f "pic2.kla": 8000 bytes crunched to 5999, ratio 74.99%.
4000-47d0 "pic2.kla": 2001 bytes crunched to 1082, ratio 54.07%.
6000-7f3f "pic3.kla": 8000 bytes crunched to 6236, ratio 77.95%.
4000-47d0 "pic3.kla": 2001 bytes crunched to 972, ratio 48.58%.
At startup (with entry at $0800):
```

```
 * $0800-$0895 (a9 00 20 00 10 a9 3b 8d ...) from "demo.prg"
 * $1000-$35d6 (4c 10 14 a2 00 bd 00 19 ...) from "Specular_Highlight.sid"
Loader call #1:
 * $6000-$7f3f (55 55 55 55 55 55 55 55 ...) from "pic1.kla"
 * $4000-$47d0 (00 00 00 00 00 00 00 0b ...) from "pic1.kla"
Loader call #2:
 * $6000-$7f3f (fc 33 cc 33 cc 33 cf c3 ...) from "pic2.kla"
 * $4000-$47d0 (00 00 09 00 00 00 09 09 ...) from "pic2.kla"
Loader call #3:
 * $6000-$7f3f (ea aa ea aa aa aa aa aa ...) from "pic3.kla"
 * $4000-$47d0 (07 0a 0a 0a 0a a2 00 0a ...) from "pic3.kla"
demo.d64: 557 blocks free (538 for DOS).
```

The last line of output tells you how many sectors are left on the disk. Spindle needs to occupy whole tracks for performance reasons, so the last track usually contains some empty space that can’t be used by DOS. The first number (“blocks free”) tells you how much the demo can grow and still fit on the disk, while the second number (“for DOS”) tells you how much space is available for additional non-Spindle files.

1.6 Labels and seeking

Starting with version 3.0, Spin allows you to declare labels in the load script, and jump to these labels at runtime. A label is a hexadecimal number in the range 00-3f followed by a colon:

```
demo.prg
Specular_Highlight.sid          1000    7e

1:
pic1.kla                        6000    2       1f40
pic1.kla                        4000    1f42

2:
pic2.kla                        6000    2       1f40
pic2.kla                        4000    1f42

1f:
pic3.kla                        6000    2       1f40
pic3.kla                        4000    1f42
```

Labels can appear in any order in the script, and multiple labels may refer to the same job, but each label must be unique; they can be thought of as 6-bit filenames.

Seeking to a label is done by loading the desired label number into A and calling the routine `spin_seek` provided in `template/seek.s`. Doing so will only reposition the stream; a separate loader call is then required to actually load the corresponding group of files.

1.7 Multi-side trackmos

Spindle supports multi-side trackmos. Each disk side is compiled separately. This helps with keeping down build times (since the entire disk side is crunched when building), and allows you to specify title and directory art for each side independently. But there is still a need to somehow associate the disk sides with each other at compile time, so the flip-disk routines can wait for the correct disk to be inserted, rather than blindly load whatever data that turns up. To this end, Spindle uses a system of magic numbers: For each disk side except the first, you have to specify a unique 24-bit number known as a knock code (as in: Knock knock, what’s the password?). For each disk side except the last, you have to specify the correct knock code for the next disk side. You can generate knock codes using any random number generator. Please do that, rather than using mnemonic values that others might also pick.

For each disk side, you specify its own knock code using `--my-magic`, and the knock code of the next side using `--next-magic`. Here is an example, for a three-side demo:

```
spin -o side1.d64 --next-magic c7189d script1
spin -o side2.d64 --my-magic c7189d --next-magic 0f91e9 script2
spin -o side3.d64 --my-magic 0f91e9 script3
```

When you specify the `--next-magic` option, spin adds two invisible loader calls to the end of the script. The first of these extra loader calls will block until the next disk side has been inserted, and then return. This allows you to stop displaying “flip disk” on the screen as soon as the correct disk has been detected. The second extra loader call corresponds to the first, implicit, loader call of the next disk. So when this call returns, you’ll be in a position to jump to the newly loaded code, and continue the demo on the new side.

Note that each disk side will have its own entrypoint, by default the load address of the first file mentioned in the script. In this way, it is possible to restart the trackmo from any disk side. It also allows you to work on each disk side separately when developing and syncing the demo. But when you are transitioning from one

side to the next, the code on the current side explicitly has to jump into the code on the next side. This gives you the freedom to jump to a different address, and thus to treat the transition from side 1 to side 2 differently from the case where the user boots directly into side 2.

All disk sides must be compiled with the same version of Spindle and use the same resident page, buffer page, and zero-page area.

1.8 Directory art

Spin lets you load up to 48 lines of directory art from a file with `-a`. Directory art is always 16 characters wide, and you can't use colours or inverted PETSCII characters.

The file format is detected automatically, and several formats are supported:

- Plain PETSCII: A text file where uppercase ASCII letters correspond to uppercase PETSCII letters, while lowercase ASCII letters correspond to graphical PETSCII symbols.
- D64: Spindle copies the directory art from an existing D64 disk image.
- Screen RAM: A raw dump of screen memory, with up to 25 lines of art, prefixed by its load address (usually 400). The art should be positioned in the upper left corner of the screen. The screen codes are converted to PETSCII. Remember that inverted characters aren't supported.
- Charcoal: The file format used by the Charcoal PETSCII editor, with up to 25 lines of art. The art should be positioned in the upper left corner of the screen. Remember that inverted characters aren't supported.

Use the `-t` option to choose a different title (label) for the disk, and `-i` to specify a two-byte disk ID. The disk ID isn't used by Spindle, but it shows up in the directory listing.

1.9 Advanced features

Spin can create 40-track disk images with the `--40` commandline option.

Read errors can be simulated using the `-E` option. For instance, `spin -E 30 script` creates a disk image where the drivecode fails to read a sector 30% of the time.

This lets you identify loading bottlenecks and stress-test the timing of your trackmos.

If a loading job is about to begin just a few blocks ahead of a track change, Spindle inserts padding sectors to align the job boundary with the track change, to improve loading speed. The padding sectors aren’t wasted: They contain copies of other sectors from the same track, which reduces the average time spent scanning for sectors, further boosting the loading speed. But sometimes you really need every last block of disk space you can get, in which case you can disable the padding mechanism with the `--squeeze` option.

Spindle takes over the IEC bus completely. If other drives are present on the bus, they are silenced so they won’t interfere with the custom serial protocol used by the loader. This feature was inspired by earlier work by Monte Carlos and Krill. Spindle can silence 1541, 1541-II, 1570, 1571, and 1581 drives, and restore them as soon as a C64 reboot is detected.

Note that in the Vice emulator, drive silencing currently only works when the demo is loaded from the highest-numbered drive. This problem is not present on real hardware.

Normally, Spindle uses a very fast serial protocol (18 cycles per bit pair) when transferring data from the drive, which is close to the electrical limitations of the IEC bus. When Spindle detects that other drives are present on the bus, it will automatically switch to a slower serial protocol (19 cycles per bit pair) to ensure reliable data transfers.

C128DCR owners may want to connect an external floppy drive just to enforce the slow protocol, to cope with the high IEC bus capacitance on that machine.

Part II

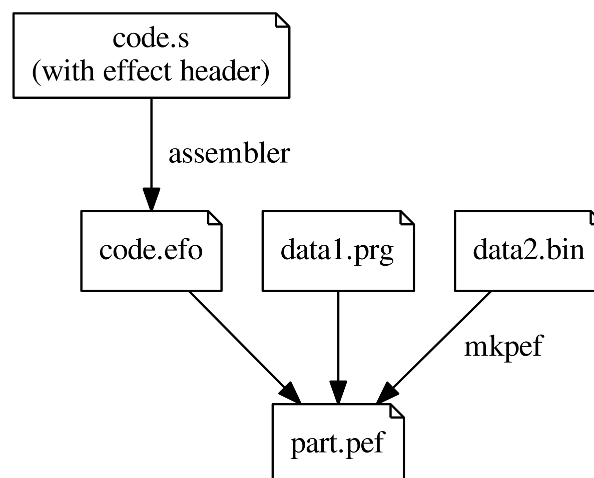
Effect linking framework

So far, we have seen that Spindle contains a highly efficient loader and decruncher that can be used on its own via the low-level Spin interface. To expose the full power of Spindle, we now turn to the high-level effect linking framework.

This part of the manual assumes some familiarity with the material covered in Part I.

2.1 Overview

The basic building block of a Spindle trackmo is the individual demo part, or *effect*. A compiled effect is stored in a single file with the extension `.pef` (packaged effect). The `mkpef` tool is used to bundle a blob of code, some metadata (called the *effect header*), and any number of data chunks with different target addresses, into such a file.

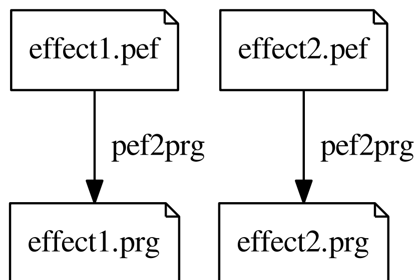


The `mkpef` tool is straightforward to use. Simply list the data files on the command-line, and optionally specify an output filename with `-o`. It is possible to supply load address, offset, and length using a comma-based syntax (run the tool without options to see brief usage instructions). For instance:

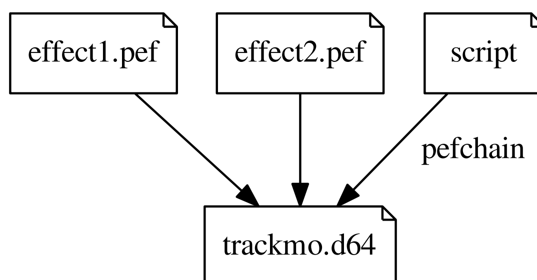
```
mkpef -o part.pef code.efo data1.prg data2.bin,4000
```

The first chunk of data in a packed effect is special, and usually contains all the code for the part. It’s called an *effect object* (filename extension `.efo`), but it’s just a regular binary file produced by your favourite assembler. It starts with a special *effect header* that contains, among other things, six pointers to routines that represent various stages of the effect lifecycle (described in detail below): `prepare`, `setup`, `interrupt`, `main`, `fadeout`, and `cleanup`.

During development and debugging, the packed effect can run as a standalone C64 program. The tool `pef2prg` converts a `.pef` file into a C64 executable file that runs the effect, optionally with a dummy music player that occupies a given number of rasterlines. The same tool can also produce a Vice monitor script to verify that the effect stays within the memory bounds declared in the effect header.



Another tool, *Pefchain*, links together several effects according to a script, and produces a complete trackmo in the form of a D64 image.



In contrast to Spin, which was reading a script of jobs (i.e. loader calls), Pefchain works with a script of effects:

```
part1.pef               space
part2.pef               space
```

In the effect script, you get to specify a transition condition for each part, typically `space` during early development (waiting for the user to press space). Later on, these conditions are changed into e.g. waiting for a memory address to contain a given value, for synchronising with music. The trigger condition is always `space` for standalone executables created by `pef2prg`.

An effect does not necessarily correspond to a loader job. Pefchain splits the chunks of each effect along page boundaries, recombines them, and arranges for the resulting snippets to be loaded at various loading slots throughout the trackmo.

This system provides creative freedom by allowing you to reorder effects very easily. Spindle automatically decides what to load and when, and computes an optimal loading schedule, while you are free to focus on aesthetics and flow.

2.2 Lifecycle

This is the lifecycle of a Spindle effect, regarded in isolation:

```
----- Preparations -----------------------------------------
1. Load any remaining sectors
2. Call prepare
----- Switchover -------------------------------------------
3. Disable interrupts
4. Store the address of the interrupt routine at fffe-ffff
5. Call setup
6. Enable interrupts
----- Running ----------------------------------------------
7. Repeatedly call main until some condition (e.g. space)
8. Repeatedly call main and fadeout until fadeout sets carry
----- Aftermath --------------------------------------------
9. Call cleanup
```

All of the routines are optional: Supply a null pointer to make Spindle ignore a particular vector. Most new effects start out with just `prepare`, `setup` and `interrupt`.

The `prepare` routine is responsible for the bulk of the initialisation. This is where

you generate speedcode and tables, make copies of graphics data across multiple banks and so on. You should not write to any VIC registers in `prepare`.

The job of the `setup` routine is to initialise the VIC registers (including colour RAM) just before the effect starts. This routine executes with interrupts disabled, and should be fast. Don't forget to initialise d011, d012, d015, d016, d018 and dd02. Your part may follow some other part that leaves unexpected values in these registers.

The `interrupt` routine obviously executes in interrupt context; this pointer is written directly to the vector at fffe. Of course you can modify fffe as part of your effect; Spindle only writes this vector at step 4 in order to minimise the amount of boilerplate code needed to get a new demo part up and running.

The `main` routine is intended for effects that fill a framebuffer as fast as possible but don't achieve full frame rate (sometimes known as "newskool effects"). Try to avoid using a `main` routine if you can, since Spindle cannot load while it is active.

The `fadeout` routine typically does two things: It triggers a fadeout operation, usually by setting a global flag that affects the behaviour of the running effect. It also monitors the fadeout in progress, returning with carry set if the fadeout has completed. Mnemonic: "Carry on" with the next effect. If the demo part is main-less, `fadeout` is simply called repeatedly in a tight loop. Otherwise, Spindle alternately calls `main` and `fadeout`.

In addition, `fadeout` plays a special role during non-linear operation (jumping around in the effect script), which will be described in a later section.

Finally, the `cleanup` routine can be used to tear down the demo part in a controlled fashion. It is called while interrupts are still enabled, but you can put a `sei` instruction inside `cleanup` if this is desirable. You could, for instance, install a non-maskable timer interrupt in `setup` and disable it in `cleanup`. Or you could use `cleanup` to wait for a particular rasterline before moving on to the next demo part.

2.3 The effect header

The `.efo` file, i.e. the file containing the assembled code for the effect, must start with a special header. All you have to do is place some constant declarations at the top of the source code, with information for the linking system and pointers to a few routines. Use `template/effect.s` as a starting point.

The code must start with the following fixed-size header (with no loading address before it):

```
.byt    "EF02"          ; fileformat magic
.word   prepare         ; prepare routine
.word   setup           ; setup routine
.word   interrupt       ; irq handler
.word   main            ; main routine
.word   fadeout         ; fadeout routine
.word   cleanup         ; cleanup routine
.word   callmusic       ; location of playroutine call
```

The fixed-size header is followed by a variable-size list of options. An option consists of a single tag byte followed by 0–2 parameter bytes (depending on the tag).

The valid tags are:

`.byt "P",FIRST,LAST`

Declares that this demo part uses the range of memory pages from FIRST up to and including LAST. There is no need to declare memory pages that are loaded, i.e. are in the loading range of one of the chunks that make up a packaged effect. But if you generate code or tables at runtime, you have to declare that memory. You can use this option multiple times.

`.byt "I",FIRST,LAST`

Declares that this demo part inherits the contents of the range of memory pages from FIRST up to and including LAST from the previous part. Normally, you don’t need to use this option. But it comes in handy when you start coding transitions between parts, such as when you load a bitmap as part of a preliminary fade-in part, and then wish to re-use that same bitmap in the actual part. Also, a BASIC fader will typically inherit the screen buffer with `.byt "I",$04,$07`. You can use this option multiple times.

`.byt "Z",FIRST,LAST`

Declares that this demo part uses zero-page locations from FIRST up to and including LAST. It is often convenient to keep all zero-page locations used by an effect close together, and declare a slightly larger range. You can use this option multiple times.

`.byt "S" (no parameter bytes)`

“Safe I/O”. Declares that any interrupt handlers used in this demo part are able to coexist with loading operations that access shadow RAM at d000-dfff. In practice, such interrupt handlers should back up the value at address 1, store 35 into 1, handle and acknowledge the interrupt,

and finally restore the previous value at address 1 (but see the earlier section on bank selection for a neat trick).

`.byt "U" (no parameter bytes)`

“Unsafe I/O”. Declares that this part may switch out the I/O registers in main context, i.e. during `prepare` or `main`. If this part follows a part that lacks the “S” tag, Spindle will make sure to insert a blank part between the two.

`.byt "A" (no parameter bytes)`

Avoid (i.e. minimise) loading during this demo part. This is handy for effects that use a lot of rastertime.

`.byt "X" (no parameter bytes)`

Don’t schedule any loading at all while this effect is active.

`.byt "M",LSB,MSB`

Declares that this demo part installs a music player with a given play-routine address. This option will be described in detail in the Music section.

`.byt "J" (no parameter bytes)`

Allows this part to jump to another position in the effect script. If this tag is present, then when the `fadeout` routine returns with carry set, it must leave a label number (in the range 00–3f) in the accumulator to trigger a jump, or a negative value to proceed normally with the script.

`.byt 0 (no parameter bytes)`

Marks the end of the tag list.

Following the final tag byte (null) is the loading address of the rest of the file, which normally contains the code for the demo part. The vectors in the fixed-size header typically point to routines inside this area.

2.4 Driver

Spindle automatically builds a small piece of code for each effect, called its *driver*, that is responsible for running the effect by calling the subroutines at the appropriate time. The driver is typically around 48 bytes in size, and will be located

immediately after the effect object. That means you have to be careful not to write beyond the end of your compiled code.

The reason for putting the driver immediately next to the effect object is to improve compression. By putting all machine code close together in RAM, the cruncher is more likely to find common sequences that can be exploited. Of course, if appending the driver would cause the effect object to spill over into some other page that is unavailable (e.g. because it's mentioned in a "P" tag), then the driver will be placed in some other free location instead.

Drivers are also responsible for making calls to the loader and for monitoring conditions (e.g. waiting for space). They typically end with a jump to the next effect driver.

In debug builds (generated by `pef2prg`), the driver is always located on page 2 (200–2ff).

2.5 Early setup code

You can rely on the Spindle linking framework to carry out certain preparations before launching the first effect. The preparations are only made once, just before the demo starts (or just before the standalone effect starts, in case of a debug build). Specifically, Spindle will make sure to:

- Disable CIA #1 interrupts.
- Enable raster interrupts by writing 1 to d01a.
- Enable keyboard input by writing ff to dc02.
- Block the Restore key by triggering a CIA #2 interrupt and not acknowledging it. You can easily undo this in your code by reading once from dd0d.
- Prepare CIA #1 timer B to assist with stable raster interrupts, as detailed below.

As a convenience for the demo coder, Spindle sets up timer B of CIA #1 to count down repeatedly with a 63-cycle period, synchronised with the raster position. The choice of timer is compatible with distributed jitter correction of NMIs. The phase of the countdown period supports delay-based jitter correction of raster interrupts using the following code snippet, which also appears in `template/effect.s`:

```
interrupt
        ; Jitter correction. Put earliest cycle in parenthesis.
```

```
; (10 with no sprites, 19 with all sprites, ...)
; Length of clockslide can be increased if more jitter
; is expected, e.g. due to NMIs.
dec     0               ; 10..18
sta     int_savea+1     ; 15..23
lda     #39-(10)        ; 19..27 <- (earliest cycle)
sec                     ; 21..29
sbc     $dc06           ; 23..31, A becomes 0..8
sta     *+4             ; 27..35
bpl     *+2             ; 31..39
lda     #$a9            ; 34
lda     #$a9            ; 36
lda     #$a9            ; 38
lda     $eaa5           ; 40

; at cycle 34+(10) = 44
```

The number in parenthesis (10 in the template code) indicates the earliest possible cycle on which the interrupt routine might start executing. If you enable sprite DMA, you will have to increase this number accordingly. If you can’t predict which sprites are enabled when the interrupt fires, you should increase the length of the clockslide accordingly.

Both Pefchain and `pef2prg` allow you to replace the early setup code with a custom routine, using the `-s` commandline option. The custom routine can be up to 128 bytes long, and must be able to run from any address (but it won’t cross a page boundary). The source code for the default setup routine is available in `template/earlysetup.s`.

2.6 Overlapping lifecycles

When several Spindle effects are linked together, their lifecycles overlap. Specifically, the call to `prepare` is made while the interrupt handler of the previous part is still active. This clearly won’t work if the memory ranges occupied by adjacent parts overlap, and that is one of the reasons for having to declare the memory usage of each part. If two adjacent parts would collide in memory, Pefchain inserts a blank part between them (and prints a warning about it). The blank part consists of a completely black screen with no badlines, along with an interrupt handler that merely calls the current music player.

Loading is performed while the parts are running. Spindle prefers to load during parts that lack a `main` routine, but if necessary, it can also schedule some loading

operations after `fadeout` returns with carry set, before the call to `cleanup`. In dire circumstances, Spindle may be forced to insert a blank part in order to do some loading (e.g. into shadow RAM), in which case it will also print a warning.

The following illustrates the switchover from one (main-less) demo part to another:

```
----- Preparations ------------------
1. Load any remaining sectors
   (and also load as much as possible
   in preparation for later parts)
2. Call prepare
----- Switchover --------------------
3. Disable interrupts
4. Install interrupt vector
5. Call setup
6. Enable interrupts
----- Running -----------------------   ----- Preparations ------------------
                                        1. Load any remaining sectors
                                           (and also load as much as possible
                                           in preparation for later parts)
                                        2. Call prepare
7. Wait for condition (e.g. space)
8. Call fadeout until it sets carry
----- Aftermath ---------------------   ----- Switchover --------------------
9. Call cleanup
                                        3. Disable interrupts
                                        4. Install interrupt vector
                                        5. Call setup
                                        6. Enable interrupts
                                        ----- Running -----------------------
                                        7. Call main until condition
                                        8. Call main and fadeout
                                           until fadeout sets carry
                                        ----- Aftermath ---------------------
                                        9. Call cleanup
```

When switching from a demo part with a main routine, the items in the simultaneous Running and Preparations phases are performed in a different order. In this case, Spindle starts with steps 7 and 8 of the first part, and then moves on to steps 1 and 2 of the second, and it tries to minimise rather than maximise the amount of loading.

2.7 Making a chain

The script file controls how all the parts fit together to make a trackmo. Here's an example:

```
# This line is a comment, and blank lines are ignored.

spindlelogo/spindlelogo.pef     -
music/music.pef                 -
ecmplasma/ecmplasma.pef         ed = 0b
lft/lft.pef                     space
-                               stay
```

This trackmo consists of five effects. The first four effects are supplied as `.pef` files (paths are relative to the current directory when Pefchain is invoked), and the fifth ("-") is the internal blank part, which is simply a black screen and an interrupt handler that calls the music player.

In the second column are transition conditions. These tell Spindle when it is time to advance from step 7 to step 8 in the effect lifecycle. There are several kinds of condition to choose from:

`space`

Wait for space to be pressed.

`-`

Drop through to the fadeout stage (after any scheduled loading has completed).

`address = value`

Wait until the given address contains the given value. In the example, it is assumed that the music playroutine stores the current song position in zero-page location ed.

`stay`

Never stop this effect. Mainly useful at the end of the script, to prevent a fadeout when experimenting with different effect orders.

`@...`

Music player interlock. Described in the section on Music.

With the special effect name "|", it is possible to extend effects over multiple script lines. Spindle will wait for each condition in turn, before transitioning to the next

script line, and the `fadeout` and `cleanup` routines won't get invoked until the very end of the extended effect. This can be used to wait for 16-bit values:

```
effect.pef              61 = 5
|                       60 = 40
```

2.8 Music

Music is a fundamental part of any trackmo. This is how it's handled by Spindle.

2.8.1 Installing a player

Any Spindle effect can *install* a music player. Such an effect would make a call from `setup` to the init routine of the tune, and also declare the address of the playroutine using the "M" tag. Please have a look at `examples/pefchain/music/install.s` for a minimal example.

The interrupt handlers of subsequent effects should be fitted with a dummy three-byte instruction, `bit !0` (that's `2c 00 00`), and the address of this instruction should be given in the last field of the fixed `.efo` header. At link time (not runtime), Spindle replaces the dummy instruction with a `jsr` to the currently installed playroutine. This makes it very easy to move parts around in a trackmo with multiple tunes, and to move tunes around in memory. The `bit` instruction remains if the part is scheduled to run even though no music player has been installed.

Only one music player may be active at a time, so installing a second player replaces the first one. To uninstall the current music player, use the "M" tag with a null parameter.

The data chunk containing the tune must of course remain in memory for the entire duration of the song. This is indicated by the `--music` option to `mkpef`, like this:

```
mkpef -o music.pef install.efo --music song.sid,,7c
```

You can list many files after the `--music` tag, and they will all remain in memory until the music player is uninstalled or replaced.

For clarity, the music-installing effect itself should call the playroutine directly, rather than having a modifiable `bit !0` instruction. Its "location of playroutine call"

header field should be null.

When an effect is launched as a standalone program by means of `pef2prg`, its playroutine call is diverted to a placeholder routine that changes the border colour and does nothing for a configurable number of rasterlines. You can adjust the time spent in this call (including `jsr` and `rts`) using the `-p` commandline option to `pef2prg`. The default is 25 rasterlines, and a value of 0 disables the feature.

An effect can have multiple playroutine calls scattered through the code. There’s only one header field, pointing to one of these call sites. But further call sites can be chained together using the operand bytes, like so:

```
        ... efo header ...

        .word   callmusic1      ; location of first playroutine call

        ...

callmusic1
        bit     callmusic2

        ...

callmusic2
        bit     callmusic3

        ...

callmusic3
        bit     !0
```

The call sites don’t have to be declared in any particular order, and they will all be replaced by a `jsr` to the currently installed playroutine.

2.8.2 Synchronisation

Spindle lets you synchronise effect transitions to music, from within the script. The simplest method is to wait for a memory location inside the playroutine to change, using the `address = value` syntax. Unfortunately this method is somewhat brittle: If there are read errors, loading might take longer than usual, and the playroutine will keep going and may eventually update the address a second time. Then, by the time the condition is actually checked, it will be too late, and the trackmo will hang forever on that effect.

A more robust (but also more complicated) approach is to use an *interlock*. This is a two-way handshake between the Pefchain driver and the music player, through a single memory location. It requires a bit of work on the music player side—often just a few instructions that are triggered from the music, e.g. by a special "sync here" tracker command.

The music initialisation routine clears the interlock location. Then, during playback, the procedure is as follows:

- The effect driver sets the location to 1.
- When the music player reaches a sync point, the location is supposed to contain 1. If not, the player must pause the music. This could be as simple as skipping the part where the current playback position is updated. As soon as the location contains 1, playback resumes.
- The music player clears the location to 0.
- The effect driver detects that the location is 0 and proceeds with the trackmo.

The address of the interlock is specified using the `-@` commandline option to Pefchain.

This is how the process might be implemented in a playroutine:

```
playroutine
        dec     timer
        bne     no_new_line

        lda     curr_tempo
        sta     timer

        ; handle music data on current line...

        ; ...

        cmp     #...            ; do we have a sync command?
        bne     no_sync

        lsr     INTERLOCK       ; clear and check
        bcc     no_new_line     ; loading error, sync delayed

no_sync
        inc     curr_line       ; advance to next line

no_new_line
        ; update glides, vibrato etc.
```

```
        ; ...

        rts
```

Pefchain ignores any characters following the “@” in the condition column. Thus, a music packer could be made to look at the Spindle effect script and automatically insert sync points into the music data based on these annotations.

The *Blackbird* music packer (called `birdcruncher`) supports this syntax: It can scan a Spindle script for sync points in the format `@SS:TT` where `SS` is the song position and `TT` is the track position, in hex. Given the following script:

```
part1.pef               @05:00
part2.pef               @0c:18
part3.pef               @10:00
part4.pef               stay
```

`birdcruncher` inserts three sync points into the music data (the first at song position 5, track position 0). Later in the build process, the music data is presumably baked into a `.pef` that installs the music player. Finally, Pefchain reads the same script (and the `.pef` file) and uses the interlock system to trigger the fadeout of the first three demo parts when those sync points are encountered.

2.9 Visualisation

Apart from producing a D64 image, Pefchain prints a chart detailing the memory usage of every effect. Here is an example:

```
0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   sectors
r|......................................***.....................  5 (loader)
r|..******..............................ccc................*.... 14 spindlelogo
r*..MMMMMM***...***.***.*******.*******.|||****.*******.***c***. 84 music
r...||||||..LUU....................c....***....................U  4 ecmplasma
r...||||||cc...ULLL.LLL.LLLLLLLULLL.LLL.LLLLLLLULLLLLLLULLL.LLL.    lft
rc..||||||......................................................    (blank)
demo.d64: 557 blocks free (538 for DOS).
```

By default, every column in the chart corresponds to four pages of RAM, but the `-w` option can be given once or twice to increase the granularity. Here is a legend for the characters:

`r`

This memory is reserved for Spindle. It consists of the resident page and the buffer page, configurable with Pefchain’s `-r` and `-b` options, respectively.

`.`

This memory is not used by the effect.

`L`

This memory was loaded from disk.

`c`

C is for code. This memory was loaded from disk as part of the `.efo` chunk, or it might contain driver code generated by Spindle.

`M`

This memory was loaded from disk and will remain in memory until the music player is uninstalled or replaced.

`U`

This memory is used by the effect (but not loaded).

`|`

This memory is inherited from the previous effect.

`*`

This memory is being loaded here in preparation for a later effect.

The number just to the left of the effect name indicates how many sectors are loaded from disk while the effect is running. This is roughly proportional to the number of stars (“*”) on the same row, but it also depends on how well the data could be compressed.

As a general rule, Spindle tries to load everthing as early as possible. This behaviour is often what you want, and if not, it is very easy to modify by adding false page-used declarations (“P” tags) to effects. Furthermore, the “X” tag can be used to minimise the loading that takes place during a particular effect. This has been done for `spindlelogo` in the example, because it is more interesting for the audience if the bulk of the loading occurs after the music has started.

In the example, the blocks loaded during “spindlelogo” correspond to the `MMMMMM` and `c` segments of music.pef. After “music” has been launched, Spindle loads

a large number of blocks comprising the `L` and `c` of “ecmplasma” and most of the `L` and `c` segments of “lft”. However, it cannot load into the memory range already occupied by the `ccc` of “spindlelogo”, because this memory is still in use: Since the video matrix and font of “spindlelogo” remain visible during “music”, the corresponding memory pages have been declared as inherited (with the “I” tag) in “music”. Once “ecmplasma” is up and running, Spindle loads and decrunches the remaining data into this area.

As you can see, the code and data of “ecmplasma” fits perfectly into gaps left by “lft”. This is no coincidence: The first time you run Pefchain, most parts will interfere with each other, and Spindle will be forced to insert blank parts between them. But if it is at all possible to alleviate the situation by moving things around in memory, a quick glance at the chart will often be enough to see how it should be done.

Whenever Spindle is forced to insert a blank part for some reason, it prints a warning. Where applicable, it will also suggest how to address the problem. For instance, if we switch the order of “ecmplasma” and “lft”, we get the following output:

```
Warning: Inserting blank filler because 'music' and 'lft' share pages a0-a8.
Suggestion: Move things around or insert a part that only touches pages 04-0f,
25-27,2e-3d,4b-4f,5c-5f,79-7b,8b-8f,9c-9f,b9-bb,d9-db,ed-ef,fb-ff and zero-page
locations 02,20-f3,f9-ff.
0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   sectors
r|......................................***.....................  5 (loader)
r|..******..............................ccc................*.... 14 spindlelogo
r*..MMMMMM***...***.***.*******.*******.|||****.*******.***c***. 85 music
rc..||||||..............................***.....................  4 (blank)
r...||||||cc...ULLL.LLL.LLLLLLLULLL.LLL.LLLLLLULLLLLLLLULLL.LLL.    lft
r...||||||..LUU....................c...........................U    ecmplasma
rc..||||||......................................................    (blank)
demo.d64: 556 blocks free (538 for DOS).
```

Depending on the demo, the brief black intermission might not be a problem. Another way of addressing the problem, as is visually clear from the chart, would be to relocate all of “spindlelogo” to 3400 (and adjust the inheritance declarations in “music”). But a third option is to follow the suggestion and add a filler part that doesn’t interfere with the memory of its neighbours. Spindle lists all memory pages and zero-page locations that are free. Be aware, however, that Spindle currently doesn’t track zero-page addresses used by the music player, so you have to take care of that yourself. For instance, my playroutine uses zero-page locations from e0 up, so I just make sure to stay below that for effect code.

2.10 Labels, jumps, and loops

Spindle is primarily designed and optimised for linear trackmos, where all data is loaded in a predictable sequence. But, starting with Spindle 3.0, non-linear loading is possible using *labels*.

Labels appear on a line of their own, as a number in the range 00–3f followed by a colon. They can appear in any order in the script, and multiple labels may refer to the same effect, but each label must be unique; they can be thought of as 6-bit filenames.

For instance, this is what an effect script might look like for a demo with a part selector:

```
intro.pef            space

0:
select.pef           -

1a:
effect1.pef          space
go-back.pef          -

1b:
effect2.pef          space
go-back.pef          -
```

In the above example, the effects called “select” and “go-back” are supposed to trigger *jumps* in the sequence. To do this, first of all they need to declare the “J” tag in the effect header. They also need to have a `fadeout` routine. As usual, `fadeout` returns with carry set when it’s time to proceed with the next effect. But when this happens for a “J” effect, Spindle also checks the value of the accumulator. A negative number means proceed with the next effect as usual, but a valid label (a number in the range 00–3f) will trigger a jump to that label.

Thus, in the example, “go-back” might have this as its `fadeout` routine:

```
fadeout
        lda     #0
        sec
        rts
```

And the “select” effect would have a `fadeout` routine that returns with carry clear until the user has selected a part. Then, it returns with carry set and `1a` or `1b` in

the accumulator.

Of course, it’s not strictly necessary to have a separate “go-back” effect: Each demo part could declare the “J” tag itself, and jump back to the selector part. But a “go-back” effect makes the design modular and simplifies rearranging the script.

Note that Pefchain will still optimise for the linear case, so for the above script it will actually load and prepare `effect1` while displaying the part selector, and `effect2` while displaying the first instance of the “go-back” effect. To prevent this preloading, you can insert blank effects between independent segments of the script:

```
intro.pef               space

0:
select.pef              -
-                       -

1a:
effect1.pef             space
go-back.pef             -
-                       -

1b:
effect2.pef             space
go-back.pef             -
```

Please be aware that when you jump to a label, the old interrupt routine remains active until it’s time to `setup` the new effect. You have to ensure that the new effect (and any other data that gets loaded in the same go) won’t interfere with the currently running effect or music player. When you’re not jumping, Pefchain ensures that adjacent effects don’t get in each other’s way, but as soon as you break the sequence you have to check this yourself. The visualisation chart is an invaluable tool for the job.

With the `--loop` (or `-L`) option, you can tell Pefchain to automatically jump to a specific label after reaching the end of the script. This is a special kind of jump that is predictable at link time, so Pefchain is able to check for memory collisions and insert a blank effect if necessary.

The ability to jump adds about 40 bytes to the effect driver.

The new (jumped-to) effect will be loaded, and its `prepare` routine called, before the `cleanup` routine of the old effect is called.

2.11 Flip-disk parts

Pefchain supports multi-side trackmos using the same system of knock codes as Spin:

```
pefchain -o side1.d64 --next-magic 86bc4f script1
pefchain -o side2.d64 --my-magic 86bc4f --next-magic c2c278 script2
pefchain -o side3.d64 --my-magic c2c278 script3
```

When you specify the `--next-magic` option, pefchain treats the last effect in the script specially. This will be referred to as the “flip-disk part”. It will not be allowed to have a `main` routine, and its transition condition will be the fact that the next disk side has been detected.

In other words, the flip-disk part will be initialised normally, by calling `prepare` and `setup` and installing the interrupt handler. It will remain in effect until the new disk side is detected. Then, the `fadeout` routine will be called in a loop until it returns with the carry flag set. You could thus use the first `fadeout` call to trigger some interrupt-driven animation that removes “flip disk” from the screen, and then keep carry clear until the animation completes in order to prevent a premature transition to the first effect on the next disk side.

When `fadeout` returns with carry set, Spindle will load the first part of the next disk side and `prepare` it while the flip-disk part remains active. When this is done, the `cleanup` routine of the flip-disk part is called. In this way, you can keep interesting things on the screen until the call to `cleanup` comes; in it, trigger some interrupt-driven animation to really fade out the effect, and busywait in the `cleanup` routine until this animation has completed.

> In earlier versions of Spindle, the `cleanup` routine of a flip-disk part was called before the first `prepare` routine on the new disk side. This inconsistency has been fixed in version 3.0.

Keep in mind that the flip-disk part mustn’t use the same memory as the first effect on the next disk side. Normally, Pefchain would assist you in detecting such collisions, but it cannot help you here since disk sides are compiled separately.

Finally, it may be useful to be able to detect whether the first part on a disk side was launched due to a transition from the previous disk side, or directly from the regular boot loader (e.g. starting in the middle of the trackmo). Before calling the `prepare` and `setup` routines *of the first effect on a disk side*, Spindle will set the carry flag if there was a transition from the previous disk side (we’ve “carried

over”), or if the effect was reached from a jump; otherwise—i.e. if the effect was started directly from the bootstrap—carry will be clear. For effects later on in the scripts, the state of the carry flag is undefined.

2.12 Streaming data

Pefchain provides two separate mechanisms for streaming data, one for graphics and one for music.

2.12.1 Streaming graphics

Normally, a packed effect describes a number of data chunks that should be present in memory when the effect starts. But with the `--stream` option to `mkpef`, it is possible to specify a series of chunks to be loaded in sequence, across subsequent entries in the effect script.

In the following example, data chunks are specified with an explicit loading address, to make it easier to see what’s going on. Normal `.prg` files are also supported, of course.

Suppose we have four packed effects (the backslash means that the command continues on the next line):

```
mkpef -o part1.pef effect1.efo \
      --stream data1.bin,4000 data2.bin,5000 data3.bin,4000 data4.bin,5000
mkpef -o part2.pef effect2.efo
mkpef -o part3.pef effect3.efo
mkpef -o part4.pef effect4.efo
```

And the following script:

```
part1.pef               space
part2.pef               space
part3.pef               space
part4.pef               space
```

Then the first data chunk (`data1.bin`) will be loaded in time for `part1`, `data2.bin` will be loaded in time for `part2.pef`, and so on. The result can look like this:

```
0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   sectors
r|......*.......**............................................  6 (loader)
```

```
r|......c***....LL..**..........................................  7 part1
r|.......c......**..LL..........................................  5 part2
r|........c.....LL..**..........................................  5 part3
r|.........c........LL..........................................    part4
```

You can also tell `mkpef` to be verbose with `-v` in order to get a visualisation chart for a single effect.

Note in particular that the first data chunk after `--stream` is not delayed. While this may seem counter-intuitive, it’s often what you want in practice.

The `--stream` feature is particularly useful when combined with extended effects. Consider this double buffered system: An interrupt routine consumes data from a front buffer at 4000, while the loader populates a back buffer at 5000. When the interrupt is done with the first buffer, it starts reading from 5000; at this point it’s time to start loading into the buffer at 4000, and so on. Suppose the interrupt routine stores the MSB of the front buffer address at zero-page location a1. Then we can do something like:

```
mkpef -o part1.pef effect1.efo \
      --stream data1.bin,4000 data2.bin,5000 data3.bin,4000 data4.bin,5000
```

and

```
part1.pef               a1 = 50
|                       a1 = 40
|                       a1 = 50
|                       -
```

To spell it out: The first chunk of data (`data1.bin`) is present at address 4000 when the effect starts to display. While it displays (or possibly earlier), the second chunk is loaded at 5000. Once the running effect starts to read the second buffer at 5000, the script proceeds to the second row (the first “|”), and the third data chunk is loaded at 4000. Once the running effect reads at 4000 again, the script moves to the third row, and the fourth data chunk is loaded at 5000. Finally, when the effect reads at 5000, the script moves to the last row, and immediately begins the fadeout process (due to the "-" condition).

Also have a look at `examples/streaming` which combines streaming graphics with a looping script.

2.12.2 Streaming music

Recall that an effect script line has two columns: The effect filename and the condition. But the condition can actually have a special suffix, starting with a colon and extending to the end of the line, carrying the filename of an additional data chunk. The filename has to end with `.prg` (case doesn’t matter here), and there is no need for whitespace before the colon.

The extra data chunk will be loaded in time for the effect listed on the same line, and will also be inherited to subsequent effects up to (and including!) the next line where extra data is added. The automatic inheriting also stops if the current music player is uninstalled or replaced.

So in the following example, the data from `extra1.prg` will be present in memory during “part1”, “part2”, and “part3”. The data from `extra2.prg` will be present during “part3”, “part4”, and “part5”, and so on.

```
part1.pef               space:extra1.prg
part2.pef               space
part3.pef               space:extra2.prg
part4.pef               space
part5.pef               space:extra3.prg
```

The resulting visualisation might look like this:

```
0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f   sectors
r|......*.......**..............................................  6 (loader)
r|......c****...LL..**..........................................  7 part1
r|.......c.....||...............................................    part2
r|........c....||..LL...........................................    part3
r|.........c...**..||...........................................  5 part4
r|..........c...LL..||..........................................    part5
```

The *Blackbird* music packer (called `birdcruncher`) supports this syntax. When you tell `birdcruncher` to use the distributed output format, it can look at a Pefchain script and create the necessary data chunks and sync points. A complete script might look like this:

```
install-music.pef       -          :musicdata1.prg
part1.pef               @05:00:1c00:musicdata2.prg
part2.pef               @0c:18
part3.pef               @10:00:1800:musicdata3.prg
part4.pef               @11:00
part5.pef               @11:0c:1c00:musicdata4.prg
```

The Blackbird packer only looks at characters following an “@” and expects them to follow a particular syntax. The initial chunk is only meant for Pefchain, so we can indulge in some whitespace to make the filenames line up. The corresponding `birdcruncher` commandline might look like this (all on a single line):

```
birdcruncher -t dist -o music-resident.prg
             -s 00:00:1800:musicdata1.prg
             -@ pefchain-script song.bb
```

A certain amount of manual fiddling is required to find the best loading spots and buffer sizes, but on the whole, Spindle and Blackbird form a very powerful toolchain for creating trackmos with streaming music.

2.13 Effect debugging

Pefchain trusts you to declare all memory ranges used by an effect with “P”, “I”, and “Z” tags. If you make a mistake in these declarations, the effect may still work as a standalone program. But in a script, the same effect could trigger obscure crashes by overwriting code or data that was preloaded for a later effect.

Luckily, the `pef2prg` tool has an option, `-m`, for creating a script of monitor commands for the Vice emulator. These commands check that you do not write to memory locations outside the declared ranges or loaded data chunks.

For instance, if your effect is allowed to touch pages 30–3b and zero-page locations 90–9f, you’d get the script shown below. Location 2ff is used internally by the `pef2prg` runtime to enable and disable checking.

```
watch store 0200 02fe if @ram:$2ff==$aa
watch store 0300 2fff if @ram:$2ff==$aa
watch store 3c00 cfff if @ram:$2ff==$aa
watch store e000 fff9 if @ram:$2ff==$aa
watch store 02 8f if @ram:$2ff==$aa
watch store a0 ff if @ram:$2ff==$aa
```

By passing the filename of the script to Vice with `-moncommands`, any out-of-bounds write will immediately freeze the program and launch the built-in monitor.

Note that this mechanism is not a 100% guarantee. For instance, writes to the I/O range (d000-dfff), the CPU vectors (fffc-ffff), and registers 0–1 are always allowed. But when it does work, it can literally save you hours of debugging, so be sure to use it whenever you can.

2.14 Using the reserved areas

Like Spin, Pefchain lets you specify a resident page, a buffer page, and a zero-page area using commandline options `-r`, `-b`, and `-z`, respectively.

The buffer page and zero-page area are available to effects; simply declare usage with “P” and/or “Z” tags. Naturally, Spindle cannot load anything while such effects are running, and may be forced to insert blank parts as a consequence.

Effects should never access the resident page.

2.15 Methodology

The following is merely a suggestion on how to work with Spindle. It is included partly as helpful advice, partly because it may shine a light on some of the design decisions underpinning the system.

First, create some demo parts. Start with the template, or add an `.efo` header to your existing code. In this early phase, you’ll probably only need the fields `prepare`, `setup`, `interrupt` and possibly `main` depending on the effect. Use `pef2prg` during development. Keep each demo part inside its own subdirectory of your main project directory for the demo, and give each part a short working name. The name of the `.pef` file should be based on this name, rather than something non-descriptive like `effect.pef`. As the part evolves from an experimental hack to an enjoyable demo effect, you should at some point declare what memory pages and zero-page locations it uses, before you forget all about it. Use the Vice monitor script feature to verify your declarations.

Once you have the parts, put them into a script in order of increasing awesomeness. All transition conditions should be “space” at this point. This stage corresponds to what filmmakers call “initial assembly”. Watch your demo a couple of times and try out different orders. Study the memory chart, and see if you can make some radical changes to improve the loading times, e.g. changing the order of parts, adding some fillers, and—if it isn’t too much work—moving large chunks of data around. But don’t start micro-optimising at this point, and don’t make any transitions yet.

Add music if you already know what SID tune you’re going to use. Otherwise, let the general flow of the effects inspire your choice of soundtrack (or the process of composing one). Change all transition conditions to the “@” or “address = value” kinds (or “-” where applicable), so the music drives the overall progress of the demo. Adjust things until you are happy, and then make a conscious decision that

you intend to stick with this part order and overall timing.

Now you have what filmmakers call a “rough cut”. Time to start working on the transitions. Begin with the big stuff, such as adding intermediate parts to e.g. make a background picture appear in anticipation of an upcoming effect. Since you know you won’t change the order anymore, you can start using “I” tags to inherit data across parts, and try to improve loading times in general. You can also work on eliminating the blank parts inserted by Spindle. At this stage you’ll probably add `fadeout` routines to several parts. As the script begins to gel, use the `-E` option to simulate disk errors in order to find out where the loading bottlenecks are.

By now you’ll probably have noticed that some of the switchovers are glitchy. Where applicable, add `cleanup` routines to e.g. turn off interrupts and wait for a particular rasterline before allowing the next part to run. Take care to insert extra calls to the playroutine where necessary (link them together using the operands of the `bit` instructions, as described in the Music section).

Inevitably, you’ll find yourself in an infinite loop where you watch the demo, notice some detail you wish to change, change it, then watch the demo again just to see if it still works, notice some other detail you wish to change, and so on. A pro tip is to write down the small things you notice, then fix them all in one batch before re-watching. If you are unsure about a fix, use `pef2prg` to watch that part in isolation. It’s also quite easy to make mini-scripts for checking a few effects and transitions at a time.

When you are satisfied with the demo (or the deadline is getting uncomfortably close, whichever happens first), don’t forget to add directory art using the `-a`, `-t`, and `-d` options. And, if you have the opportunity, always test your demo on a real drive.

May your Spindle productions amaze and inspire!