UNRAVELING

THE ZX SPECTRUM NEXT

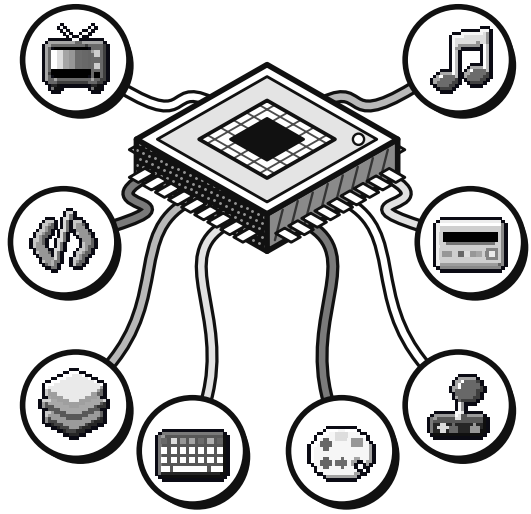
Volume 1: The Hardware, as a Programmer Sees It



István Novák

Unraveling the ZX Spectrum Next

Volume 1: The Hardware, as a Programmer Sees It



István Novák

Unraveling the ZX Spectrum Next, Volume 1: The Hardware, as a Programmer Sees It

Copyright © 2026 István Novák. All rights reserved.

No part of this book may be reproduced in any form without written permission from the author, except for brief quotations in reviews.

The facts in this book are for ZX Spectrum Next core 3.03.03; where the older core 3.02 differs, the text says so. ZX Spectrum is a trademark of its respective owner; this book is not endorsed by any company named in it.

Companion website: `https://kliveide.net/`

ISBN: *(to be assigned)*

To my late father and my mother,
thank you for raising me
into the family man and software engineer
I am proud to be.

Contents

Introduction **xiv**
The ZX Spectrum Next Challenge . . . xv
Why This Book, and How It Is Built . . . xv
A last word . . . xvii

I The Machine 1

1 The FPGA and Its Cores: How a Blank Chip Becomes a Next **2**
A chip that forgets . . . 2
What the blank chip is made of . . . 3
The flash and its slots . . . 5
The doorman: the anti-brick core . . . 5
Changing its mind . . . 6
From power-on to the prompt . . . 7
The firmware file . . . 8
Config mode . . . 9
Updating the core . . . 10
Repairing a broken slot 1 . . . 11
Extra cores . . . 13

2 The Z80N: A Familiar Heart and Twenty-Nine New Tricks **15**
A Z80 without a chip . . . 15
Registers, and one you cannot see . . . 16
Machine cycles and T-states . . . 17
The opcode fetch . . . 17
Reading, writing and talking to ports . . . 18
Four speeds . . . 20
Sharing the bus . . . 23
Interrupts, from the CPU's side . . . 23
The flags, documented and not . . . 26
Undocumented instructions . . . 28
Where the copy departs from the original . . . 28
Twenty-nine new instructions . . . 30

3 Memory, Ports and NextRegs: How the Z80N Reaches the Machine **36**
Three doors into one machine . . . 36
Eight slots of 8K . . . 37

From a CPU address to the SRAM . . . 39
Who answers an address . . . 43
The DivMMC's entry points . . . 44
The 128K's ports, rewritten . . . 46
The ROMs, and how to replace one . . . 47
Contention . . . 48
Ports: which bits count . . . 49
Switching ports off . . . 50
What you read when nobody answers . . . 50
Two ways into the NextRegs . . . 51
The Copper goes first . . . 52
A map of 256 numbers . . . 53
Resets . . . 54

4 Interrupts: Fourteen Voices, Two Wires **55**
Two wires into the CPU . . . 55
Fourteen voices in a fixed order . . . 56
Pulse mode: one pulse for everybody . . . 57
Hardware IM2 mode: a vector for every voice . . . 58
The daisy chain . . . 59
RETI releases, RETN does not . . . 61
Switches and status bits . . . 61
The sources up close . . . 63
How fast the CPU answers . . . 64
The DMA steps aside . . . 64
The fire alarm . . . 65
Who answers at $0066 . . . 66
The stackless NMI . . . 66
Resets . . . 67

5 DMA: One Bus, Two Masters **68**
A second master on the bus . . . 68
Two ports, two dialects . . . 70
Base bytes and their followers . . . 70
Port A and port B . . . 71
The mode and port B's address . . . 72
Enables and flags . . . 72
Commands . . . 73
One byte, six T-states . . . 74
The prescaler: spacing the bytes out . . . 75
Burst mode: the bus goes back between bytes . . . 76
Auto restart . . . 77
Reading the DMA back . . . 77
Stepping aside for interrupts . . . 78
Resets . . . 78

The zxnDDMA: the blitter . . . 78

6 CTC: Four Timers in a Ring 84
A Zilog chip in the FPGA . . . 84
One channel inside . . . 85
The control word . . . 86
Timer and counter . . . 87
The ring . . . 87
A stopwatch . . . 88
Interrupts . . . 90
Resets and core versions . . . 90

7 Input: One Matrix, Many Hands 92
The Spectrum's keyboard . . . 92
The clerk . . . 93
One matrix, many hands . . . 94
The sixteen extra keys . . . 94
Why CAPS SHIFT comes early . . . 95
The PS/2 keyboard . . . 96
Joysticks as keys . . . 96
Kempston ports and Mega Drive pads . . . 97
The interviewer . . . 98
The Kempston mouse . . . 99
The two buttons on the side . . . 100
Resets . . . 101

II The Picture 103

8 How a Television Draws a Picture 104
The electron beam . . . 104
The signal carries only "how bright, right now" . . . 105
Why you must follow the beam . . . 106
How colour is displayed . . . 106

9 The Big Picture: Five Producers, Four Layers, One Dot 110
From one chip to a pipeline . . . 110
The cast of layers . . . 111
Where each layer can draw . . . 113
Clip windows: a rectangle for each layer . . . 114
Scrolling without moving memory . . . 115
Two answers for every dot . . . 117
The video pipeline . . . 118
Two dots for every position of the beam . . . 118
Stage 0: gather . . . 119

Stage 1: look up . . . 119
Stage 2: decide . . . 120
One dot through the pipeline . . . 120
Three clocks . . . 121
512 colours . . . 121
What stays the same . . . 122

10 The Frame and the Beam: Two Counters, Four Timings 123
Two counters . . . 123
Counters that start early . . . 124
Four timings, two frame rates . . . 125
The 60 Hz frame . . . 125
The frame interrupt . . . 126
The line interrupt . . . 126
Line numbers for the wide area . . . 127
Dots, ticks and T-states . . . 129
From counters to cables . . . 129
The scan doubler and VGA . . . 130
HDMI . . . 130

11 Colours and Palettes: From an 8-Bit Index to a 9-Bit Colour 132
Nine bits of colour . . . 132
Eight palettes . . . 133
Writing and reading a palette . . . 134
The ULA: three maps of one palette . . . 135
The other layers: a dot and an offset . . . 135
Two rules for transparency . . . 136
`$E3`: the colour of nothing . . . 138
The fallback colour . . . 138
The colours at power-on . . . 138
What palettes make cheap . . . 138
From palette to screen . . . 140

12 The ULA Layer: The Spectrum's Screen, Extended 141
The same ULA, new memory . . . 141
Reading a line twice . . . 142
The Timex modes . . . 143
A second screen . . . 144
Hi-colour: a colour for every 8 × 1 dots . . . 144
Hi-res: 512 dots across . . . 145
ULANext: the attribute byte split anew . . . 145
ULA+: 64 colours . . . 146
The border in every mode . . . 147
Scrolling by the dot . . . 148
Half a dot . . . 149

Clipped and switched off . . . 149

13 LoRes and Radastan: Big Pixels in the ULA's Place 151
A pixel is four dots . . . 151
Where the bytes are . . . 152
From a byte to a colour . . . 153
Radastan: 16 colours in 6K . . . 153
In the ULA's place . . . 154
Scrolling by half a pixel . . . 154
What a screen costs . . . 156

14 Layer 2: A Byte for Every Dot 157
One byte, one dot . . . 157
Where Layer 2 lives . . . 158
Three resolutions . . . 159
Why the bytes run down the screen . . . 159
640 × 256: two dots in a byte . . . 160
The CPU's view through port `$123B` . . . 161
Through the MMU . . . 163
Scrolling, clipping and colours . . . 163
Fetching a byte for every dot . . . 165

15 The Tilemap: A Screen of 8 × 8 Tiles 166
The map and the tiles . . . 166
Bank 5 and bank 7: where the tables can live . . . 167
How big the two tables are . . . 168
Layouts that fit . . . 168
A trap: moving a setup from bank 5 to bank 7 . . . 169
The attribute byte . . . 170
A tile's dots . . . 171
Eight orientations . . . 171
40 or 80 columns . . . 172
Text mode . . . 172
Scrolling and clipping . . . 173
Fetching a character ahead . . . 174

16 Sprites: 128 Pictures and Two Line Buffers 175
Sprite space . . . 175
Patterns . . . 176
The attribute bytes . . . 177
Uploading through three ports . . . 178
Orientations and sizes . . . 178
Anchors and relative sprites . . . 180
Two lines, not a screen . . . 181
Which sprite is in front . . . 182

The time budget of a line . . . 182
The cut-off window . . . 182
Two examples on the machine . . . 183
What a program can count on . . . 184
Over the border, and clipping . . . 184
Collisions . . . 185

17 Putting It Together: Four Layers, One Colour 187
Four answers, one colour . . . 187
When a layer is transparent . . . 188
The ULA and the tilemap become one . . . 188
Six orders . . . 189
Two exceptions . . . 190
Blending: adding instead of choosing . . . 191
When nothing is opaque . . . 193

18 Raster Effects: Changing the Picture Mid-Frame 194
The beam as a clock . . . 194
When a write shows . . . 195
The line interrupt and its jitter . . . 195
The Copper's two instructions . . . 196
Where seven WAITs stop . . . 197
The Copper's memory . . . 198
Ending a list, and the four modes . . . 198
Why the Copper is exact . . . 199
A sky gradient from one palette entry . . . 199
A status bar that stands still . . . 200
Thirty-two sprites from eight . . . 202
One dot's way through the machine . . . 204

III The Sound 205

19 How a Computer Makes Sound 206
Sound is air pushed back and forth . . . 206
A sample is one number for one moment . . . 207
From a number to a voltage: the DAC . . . 208
From a staircase to a smooth wave: the filter . . . 208
Fast enough, or it sounds wrong . . . 209
Sound from a single bit . . . 210
Several sounds at once are added together . . . 211
Two speakers: stereo and space . . . 211
Where the numbers come from . . . 212

20 The Big Picture: Four Sources, One Mixer 213

From one bit to four kinds of source . . . 213
The cast . . . 214
The beeper, EAR and MIC . . . 215
Three AY chips . . . 215
Four 8-bit DACs . . . 216
The Raspberry Pi's audio . . . 216
Generators and level writers . . . 216
One mixer, two sides . . . 217
Where the sound goes . . . 218
The clocks behind the sound . . . 219
The switches . . . 219

21 The Beeper: EAR, MIC and the Tape 221
Two bits of sound . . . 221
EAR and MIC in the mixer . . . 222
A note from a loop . . . 222
The CPU's speed and contention . . . 223
Two voices from one bit . . . 224
What two voices cost . . . 226
Saving to tape . . . 227
Loading from tape . . . 228
What port `$FE` bit 6 reads . . . 229
The issue 2 keyboard . . . 229
The internal speaker . . . 230

22 One AY Chip: Tones, Noise and Envelopes 231
The parts of one chip . . . 231
Sixteen registers behind two ports . . . 232
A counter that makes a square wave . . . 233
Noise from a shift register . . . 234
Tone, noise, both or neither . . . 234
Sixteen volumes on a curve . . . 235
The envelope: a volume that moves . . . 236
The sixteen envelope shapes . . . 236
AY or YM . . . 238
Playing a chord . . . 239
Bass from the envelope . . . 240

23 TurboSound: Three AY Chips Behind Two Ports 242
Three chips, two ports . . . 242
One note, five writes . . . 243
TurboSound on, TurboSound off . . . 244
Where the channels go . . . 245
Panning a whole chip . . . 245
Nine channels in the mixer . . . 247

When the chips are reset . . . 247
What programs do with nine channels . . . 248

24 The 8-Bit DACs: Four Channels, Six Sound Cards 249
A number that holds . . . 249
Four channels, two sides . . . 250
Six sound cards on four channels . . . 250
The NextReg mirrors . . . 252
Who times the writes? . . . 252
A drum hit . . . 256
The DACs in the mixer . . . 257

25 Audio from the Raspberry Pi: Samples on Four Wires 258
Four wires and a clock from the Pi . . . 258
One register: NextReg `$A2` . . . 259
From a sample to a word . . . 260
When a word arrives . . . 260
Into the mixer . . . 261
Reading the Pi's samples . . . 261
Loading a tape from the Pi . . . 262
The way back: the Next's sound to the Pi . . . 264

26 The Mixer: One Sum, Three Outputs 265
Five inputs, one sum . . . 265
Silence is 1,536 . . . 266
Headroom . . . 267
The audio jack: one pin, 4,096 levels . . . 269
The internal speaker . . . 269
HDMI audio . . . 270

27 After the Sum: From the Mixer to Your Ears 272
The sum and the sound . . . 272
The jack: no sample rate . . . 273
HDMI: a number every 21 µs . . . 274
Which output for which source . . . 275
The whole of the sound . . . 275

IV Talking to the World 277

28 Storage: an SD Card on a Serial Bus 278
Five devices on one bus . . . 278
Choosing a device . . . 279
One byte, sixteen clocks . . . 280
Talking to an SD card . . . 281

Streaming with the DMA . . . 285
The flash and the Pi . . . 285
The DivMMC's part . . . 285
Tape and disks . . . 286
Resets . . . 287

29 UARTs: Two Serial Ports Behind One Door 288
Two UARTs, four ports . . . 288
A frame on the wire . . . 289
The baud rate . . . 290
The frame register . . . 291
Two buffers and the status . . . 291
Break . . . 294
Hardware flow control . . . 294
Interrupts . . . 295
The joystick port . . . 295
Resets . . . 295

30 Wi-Fi: A Telegraph Clerk on a Serial Line 297
A computer for a peripheral . . . 297
Five wires and a socket . . . 298
Waking the clerk . . . 299
Talking in AT . . . 300
Joining a network . . . 301
A conversation with a server . . . 302
Telling data from chatter . . . 303
The promise that stops at the module . . . 305
How fast it can go . . . 305
Firmware is the fine print . . . 306

31 Real-Time Clock: A Watchman on Two Wires 308
A clock outside the machine . . . 308
Two wires, and nobody pushes . . . 309
Two ports, one bit each . . . 310
The rules of the conversation . . . 310
Speaking the protocol yourself . . . 311
How fast is too fast . . . 314
The watchman's registers . . . 314
Reading the time . . . 315
Setting the clock . . . 317
Fifty-six bytes that remember . . . 318
Resets . . . 319

Summary: The Next in One Page 321

Introduction

Everyone has a personal story about their ZX Spectrum. Mine started before I even got my first Speccy. One day, my late father (I'm bearing his name, István Novák) brought back a TI-59 programmable calculator from his workspace. This wasn't just a calculator—it had keys with mysterious text that went far beyond simple arithmetic. As a kid with an affection for math, I learned programming from its user guide.

I live in Hungary. In 1981, when I was 13, Hungary was part of the communist bloc. My uncle, György Marczin, an electrical engineer working for a large state-owned company, had the privilege of traveling to Western Europe. He saw the spark in me and helped me get a ZX81 from West Germany. Getting it wasn't simple—my father had to buy Deutschmarks at the Hungarian black markets, since the Forint, the Hungarian currency, wasn't convertible.

The 16K expansion pack for the ZX81 was great, but it became too small. Two years later, my father bought me a ZX Spectrum 48K. I went through the same awakening as countless others who touched that rubber-keyed wonder.

In my first year at gymnasium, HT-1080Z school computers (TRS-80 clones) became available for Hungarian students. We had four of them in my school! With other awakened buddies, we formed a small group that spent time in the computer lab before and after teaching hours. A young electrical engineer, Tibor Veres, coached us. Soon we were participating in—and winning—programming contests across Hungary.

After the gymnasium, I became a software engineer at the Technical University of Budapest and earned a doctoral title in software technology.

At the beginning of the 2010s, I was a freelance technology coach, helping software development teams and evangelizing automated testing, and I needed something to demonstrate it. One January morning, I woke up with a dream: let me go back to my childhood and create a ZX Spectrum emulator. A high-fidelity emulator requires extensive automated tests—it would serve as the perfect example.

And so, my first emulator was born: SpectNetIDE (written in C#), integrated into Visual Studio. Years later, when I coached teams on Web technologies, I started implementing Klive IDE, a TypeScript implementation.

None of it would have happened without my parents: my late father, István Novák, and my mother, Judit Marczin. This book is dedicated to them.

The ZX Spectrum Next Challenge

I witnessed the ZX Spectrum revival with the ZX Spectrum Next. Though I missed the KS1 campaign, I have the KS2 and KS3 machines. I felt a personal challenge to emulate this powerful successor in my Klive IDE.

While working on the emulator, I learned a tremendous amount about this great machine from a programmer's point of view: how its Z80N CPU, memory system, graphics layers, audio, storage, and timing features fit together when you actually want to make something run. The Next pushes boundaries—it's not just nostalgia, it's modern retro hardware done right.

The official documentation on the developer site was not much help with that. It is dry and reference-like: it tells you what each register does, and it leaves you to work out how the pieces fit. For a long time I held a handful of puzzle pieces and had no picture on the box.

So I turned detective. I decoded what the documents said, and what they left unsaid. I played with the real hardware to see what it actually did. I reverse-engineered the public FPGA source code, and I read between the lines wherever everything else fell silent. Each answer opened a new question, and I enjoyed every one of them.

The reward was never only the answer. It was the way there, the research itself. And the more I found, the more I had to tell, which is how this book began.

Why This Book, and How It Is Built

I had been playing with the idea of writing a book for about a year. I had already written several web technology books and published them on Amazon under the Unraveling series, and this book seemed to fit into the same family. I do not like reciting reference information: the official documents already do that, and the Next deserves more. I wanted to take the topics of the ZX Spectrum Next to a new level, with a new approach, and to give away what I had learned along the way.

That is why the book is built as it is. It starts with the foundations of the machine, goes on to the picture and then to the sound, and ends with the way the Next communicates with the world. In every chapter I tried to show how and why things work, not only what they are.

Writing the book also helped me find the right features for Klive IDE. Whenever I touched a topic, I started to think about how Klive IDE could help you, the reader, understand and examine it. Many of the Klive IDE notes and the demos you will meet in the chapters were born that way.

AI and This Book

We live in an AI-powered era. You can like it or hate it: AI is here, and there seems to be no reason and no way to get rid of it.

I am a Hungarian, not a native English speaker, and I used AI to turn my Hungarian way of thinking into English and to find the best ways to visualize concepts.

Perhaps today an AI could write a book from a single prompt, something like "Write me a book about the ZX Spectrum Next with this title…". If I had taken that road, the book would not be me. I did use AI heavily, for research and for ghostwriting. But the concept, the structure, the approach and many subtle but delicate details came from me, and the work took more than 1,400 prompts.

Is this a book written by AI or by me? Let us leave that question to the philosophers. I stand behind the book as it is.

Which core, which board

The Next is an FPGA, so its hardware is a file, the core, and it changes with every update. This book describes **core 3.03.03**, the core of the 26.11a system release, on every board: Issue 2, Issue 4 and Issue 5. It runs at 28 MHz without wait states, and its DMA has become a blitter, the zxnDDMA. Where the older core 3.02, which the Next ran for years, behaves differently, the text says so. Where the Issue 5 board, the third Kickstarter's, differs from the earlier ones, the text says that too; for a program it is mostly a matter of 2 MB more memory, out of the MMU's reach. NextRegs `$01` and `$0E` tell a program which core it runs on, and NextReg `$0F` which board.

The demo program and the Try it boxes

Reading about a register is one thing; watching it change the screen is another. The book comes with a **demo program**, `book-demos.nex`, built from the `examples` folder (see "Downloads, and What Comes Next" below). A chapter that has a demo ends with a **Try it** box, a green frame with the demo's name in its title bar. It tells you where to find the demo in the menu and which file holds its code, shows the few lines that do the work, and says what to look for.

Running it. Open the `examples` folder as a project in **Klive IDE**, an emulator and development environment for the Next, and choose **Run**: it builds `book-demos.nex` and loads it into an emulated Next with NextZXOS. The IDE's documentation at kliveide.net shows how to stop a demo and look inside the machine. On a real Next, copy the file to the SD card and open it from the NextZXOS Browser.

Finding a demo. The first screen lists the four parts, with keys **1** to **4**: The Machine, The Picture, The Sound and Talking to the World. A part's screen lists its demos in chapter order, each with a key of its own, so "Part I, key 1" means press **1**, then **1**. SPACE goes back. Every demo ends with "Press any key for the menu" and leaves the machine as it found it.

Safe on real hardware. The demos never write to the SD card, the flash that holds the cores, the Wi-Fi module's settings or the clock, so running them on your own Next cannot damage what is stored there.

Downloads, and What Comes Next

You can download this book in PDF format, and the zip file with the examples project, from Klive IDE's home site:

```
https://kliveide.net/unraveling-book
```

The book is intentionally titled Volume 1. I plan to continue it, and I am curious to see how the story of the ZX Spectrum goes on.

Errors, Ambiguities and Questions

I am sure this book is not free from errors, ambiguities and subtleties that are easy to misunderstand. If you find any of them, or if you simply want to know more about something related to these topics that the book does not discuss in enough detail, feel free to contact me at **`dotneteer@hotmail.com`**. I promise to update the book accordingly.

A last word

I believe in the power of community. While working on Klive, I met great people of the retro-computer community, and I have been learning a lot from them. This book owes them more than I can say.

For me, writing an emulator, and working with retro hardware and its reincarnations like the ZX Spectrum Next, is a way to "save the souls" of those old computers. It is an emotional thing, and it is something that AI will never replace.

Let me share these learnings with the community.

— István Novák

Part I

The Machine

CHAPTER 1

The FPGA and Its Cores: How a Blank Chip Becomes a Next

Switch a Spectrum on, and its ULA is a ULA from the first microsecond. Switch a Next on, and for a short moment there is no ULA, no Z80N, no Copper and no sprites: just a large blank chip, waiting to be told what to be. A fraction of a second later it is a ZX Spectrum Next, or a different machine, from a file on the SD card.

Every core update goes through the mechanisms in this chapter. Know them, and an update stops being a ritual of held keys and crossed fingers.

- What a core is, and how a blank FPGA becomes a Next at every power-on
- How the anti-brick core, the boot ROM and the firmware hand over to each other
- How a single NextReg write makes the FPGA load another core
- How to update a core safely, and how to recover when an update goes wrong

A chip that forgets

An **FPGA**, a field-programmable gate array, is a chip full of small logic blocks, memories and wires whose connections are not fixed. A long string of bits, the **bitstream**, says how to connect them: which block computes what, which wire goes where. Load one bitstream and the chip is a Spectrum Next; load another and the same chip is a Commodore 64, or a test pattern generator.

The bitstream lives in the FPGA's configuration memory, and that memory is **volatile**: when the power goes, the chip forgets everything and is blank again. So every power-on starts with the FPGA reading a bitstream from somewhere and configuring itself from it. In the Next's world a bitstream that makes a whole machine is a **core**, and "the Next core" is the bitstream of the ZX Spectrum Next.

The two board generations use two FPGA families:

- **Issue 2 and Issue 3**: a Xilinx Spartan-6 (XC6SLX16) with a 16 MB flash chip. Issue 2 is the board of the first Kickstarter, KS1.
- **Issue 4**: a Xilinx Artix-7 (XC7A15T) with a 32 MB flash chip: the second Kickstarter's board, KS2.

- **Issue 5**: a larger and faster Artix-7 (XC7A35T, speed grade 2) with the same 32 MB flash chip, and 4 MB of SRAM instead of 2 MB: the third Kickstarter's board, KS3.

The Next keeps its settings in the **NextRegs**, a file of numbered registers that configure the machine. A program reaches them through ports `$243B` and `$253B`, or with the Z80N's `NEXTREG` instruction. The section "Two ways into the NextRegs" in Chapter 3 (page 51) covers them in detail.

The **Board ID register**, **NextReg `$0F`**, tells a program which board it runs on, in bits 3–0: 0 for Issue 2, 1 for Issue 3, 2 for Issue 4, 3 for Issue 5.

The same core behaves differently on the Issue 4 and Issue 5 boards in only three NextRegs: `$0F` itself; bit 7 of NextReg `$04`, which reaches the Issue 5 board's upper 2 MB (Chapter 3); and bit 3 of NextReg `$81`, which brings the +3's disk drive signals out to the expansion bus on the Issue 5 board only (Chapter 28). Everything else that people call "new on the KS3" comes with the core's version, and an Issue 4 board running the same core has it too.

What the blank chip is made of

Most of an FPGA is a sea of small, identical cells, each far too simple to be interesting on its own. The trick is that there are thousands of them, and that the wires between them can be joined almost any way. Here is the cast, from the smallest part up (Figure 1.1).

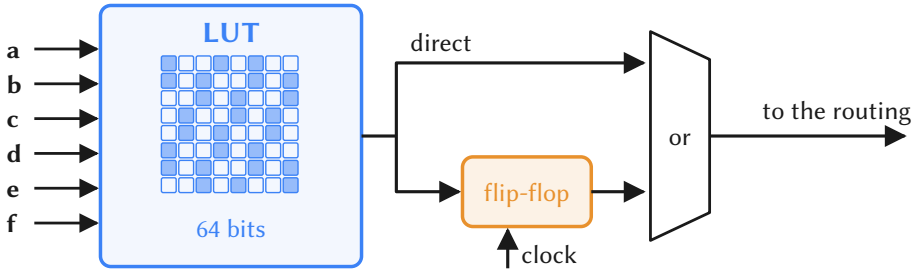


Figure 1.1: One logic cell: a look-up table and a flip-flop.

- **The look-up table**, LUT for short, is the cell that computes. In all of the Next's FPGAs it has six inputs and one output, and it holds **64 bits**: one answer for each of the 64 combinations of its inputs. It does not know what function it computes; it only looks the answer up.
- **The flip-flop** is the cell that remembers: one bit, taken on a clock edge and held until the next. LUTs compute, flip-flops remember, and every register, counter and state machine is LUTs feeding flip-flops feeding LUTs again.

- **The slice** groups **four LUTs and eight flip-flops**, with a **carry chain**: a fast, fixed wire from one slice to the next that makes adders and counters quick, because a carry does not have to wander through the general wiring.
- **Block RAM** is memory built into the chip as memory: blocks of 18 kilobits on the Spartan-6 (each usable as two of 9) and 36 kilobits on the Artix-7 (each usable as two of 18). A block has **two ports**, so two parts of a design can read and write it at the same time.
- **Distributed RAM** is memory made from LUTs: in some slices a LUT's 64 bits can be written while the design runs, turning it into a tiny 64 × 1 RAM. It suits small, fast memories such as register files and short buffers, but every LUT used this way is a LUT taken from the logic, so the table below gives only a ceiling.
- **DSP slices**, named after **digital signal processing**, the multiply-and-add work of filters, audio and video they were made for, are hard-wired multipliers with an adder and an accumulator: 18 × 18 bits on the Spartan-6, 25 × 18 with a pre-adder on the Artix-7. A multiplier made of LUTs would be many times larger and slower.
- **Clock managers** multiply and divide a clock. All the boards give the FPGA a **50 MHz** clock, and a clock manager makes the **28 MHz** master clock and its relatives from it.
- **I/O blocks** around the edge connect the logic to the chip's pins: the SRAM, the flash, the SD card, the keyboard, the video outputs and the expansion bus.
- **The routing**, a mesh of wires and programmable switches, joins all of these. Most of the bitstream sets those switches. If you ask me, the routing is the real FPGA; the cells are only what it connects.

The three chips, the Spartan-6 XC6SLX16 and the Artix-7 XC7A15T and XC7A35T, have these resources:

Resource	XC6SLX16	XC7A15T	XC7A35T
Slices	2,278	2,600	5,200
6-input LUTs	9,112	10,400	20,800
Flip-flops	18,224	20,800	41,600
Block RAM	576 Kb	900 Kb	1,800 Kb
Distributed RAM, at most	136 Kb	200 Kb	400 Kb
DSP slices	32	45	90
Clock management tiles	2	5	5

About ten thousand look-up tables on the first two boards: that is what a Z80N, a ULA, Layer 2, a tilemap, 128 sprites, a Copper, a DMA, three AY chips and all the rest have to fit in. The Issue 5 board's chip has twice as much of nearly everything, while the Next core is the same design on all of them.

Nobody draws those connections by hand. A design is written in a **hardware description language**, HDL for short: text that looks like a program but describes a circuit. **VHDL** and **Verilog** are the two most widely used (SystemVerilog extends Verilog). The Next is written almost entirely in VHDL, with a few parts in Verilog: the set-up of the clock managers, the PS/2 mouse and the HDMI encoder.

The flash and its slots

The bitstreams are kept in a **flash chip** beside the FPGA, a memory that keeps its contents without power, talked to over SPI. The flash is divided into equal **slots**, one core each, and the slot size depends on the board (Figure 1.2).

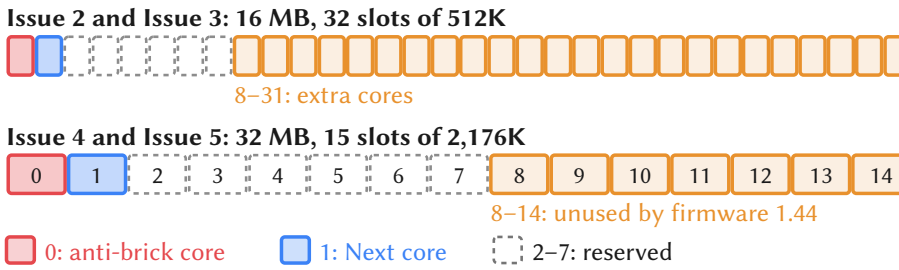


Figure 1.2: The flash chip's slots on the two board generations (the Issue 5 board has the Issue 4 layout)

- **Issue 2 and Issue 3**: 32 slots of 512K each, 16 MB in all.
- **Issue 4 and Issue 5**: 15 slots of 2,176K each, almost all of the 32 MB. The Artix-7's bitstream is bigger.
- **Slot 0** holds the **anti-brick core**, slot 1 the **Next core**, on every board.
- **Slots 2 to 7** are reserved, and on the Issue 2 board **slots 8 to 31** hold extra cores.

Slot 0 is the one the FPGA reads by itself at power-on: it is the start of the flash, where the chip looks for a bitstream when it wakes up. Every other slot is reached only when a running core asks for it, and that is the mechanism of this chapter.

I like this layout because it keeps the dangerous part small. Updating the Next core rewrites slot 1 and nothing else. If that goes wrong, slot 0 is still intact, and slot 0 is what the chip loads first.

The doorman: the anti-brick core

The **anti-brick core** is a Next core of its own, built with a few differences: the **Machine ID register**, **NextReg `$00`**, reads `$FA` instead of the Next's `$0A`; it has its own boot

ROM; and its function keys are almost all off, so F1 (hard reset), F4 (soft reset), the CPU speed key and the NMI buttons' keys do nothing. Only F2 and F3, the scan doubler and the 50/60 Hz switch, still work, so you can get a picture out of it.

I call it **the doorman**. At every power-on, its boot ROM checks the DRIVE and M1 buttons on the side of the machine. Unless both are held, it starts slot 1, and you never see it. If they are, it loads the updater from the SD card instead, as we will see in "Repairing a broken slot 1".

Its job explains why it is meant never to change. A Next with a broken slot 1 can be repaired as long as slot 0 works; a Next with a broken slot 0 cannot start at all without a programmer cable on its JTAG port.

Changing its mind

Here is the mechanism behind everything else: a running core can make the FPGA **reconfigure itself** from any slot of the flash. The FPGA has an internal configuration port through which logic inside the chip can give it commands. The Next's design keeps a short command list for it: synchronise, set the start address in the flash, and **reboot from there**. When the list runs, the FPGA throws its whole design away, the CPU, the video, every register, and loads the bitstream at that address, exactly as it does at power-on but from a different place.

Two things start that list:

- **The Core Boot register**, **NextReg `$10`**: bits 4 to 0 choose a slot, and writing bit 7 starts it. The slot number can be changed only in **config mode** (more about that below); outside it, a write of bit 7 restarts the slot that is already selected. A write of a slot number that does not exist is ignored, which is the recommended way to find the largest one; the Issue 4 and Issue 5 boards accept 0 to 14.
- **A hard reset**: the **F1 key** (on a PS/2 keyboard, or NMI + 1 on the Next's own keyboard), or bit 1 of the **Reset register**, **NextReg `$02`**. On the Next core this reloads **slot 1**, the Next core, from the flash. That is why the memory chapter could say that a hard reset is like switching off and on: in the FPGA's eyes it is a fresh start from the flash, only without the doorman.

When the anti-brick core waves you through, it does nothing more than write `$81` to NextReg `$10`: bit 7 to start a slot, and 1 for the slot. It must name the slot, because the doorman runs in config mode, where the write also sets the slot number; `$80` would choose slot 0 and start the doorman again.

What survives a reconfiguration? Not the FPGA's contents: those are replaced. But the **SRAM**, the 2 MB (4 MB on Issue 5) of memory chips beside the FPGA, keeps its power and its contents. That is how one core leaves a message for the next: it writes it into the SRAM and starts the other slot. The firmware uses that channel to pass settings to extra cores, and to ask itself to boot one. One caveat from the hardware's

own notes: on the Issue 2 board, data at an even multiple of 256K in the SRAM is not reliable across the change.

From power-on to the prompt

Now let's follow a normal start, from the power switch to NextZXOS's menu (Figure 1.3).

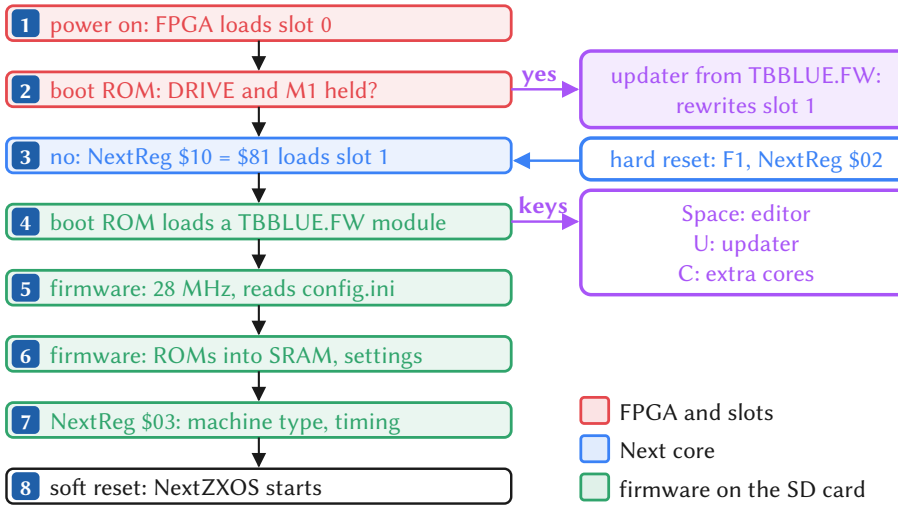


Figure 1.3: The start-up chain

1. **The FPGA loads slot 0**, the anti-brick core, by itself.
2. **The doorman checks the buttons** and, normally, writes `$81` to NextReg `$10`: the FPGA loads **slot 1**.
3. **The Next core starts in config mode** with its **boot ROM** in the bottom 16K. The CPU runs the boot ROM, which initialises the SD card, finds a FAT16 or FAT32 file system, and reads **`TBBLUE.FW`** from the root folder.
4. **The boot ROM loads one module** of the firmware into RAM at `$6000` and jumps to it. Which module depends on the keys held at that moment, as the next section shows; normally it is the **boot module**.
5. **The firmware takes over.** It switches to 28 MHz, switches the boot ROM off, reads `/machines/next/config.ini` (your settings) and the menu of machines, and shows the start-up screen with its prompts for a while.
6. **It builds the machine.** It loads the keyboard map, the DivMMC's ROM, the Multiface's ROM and the Spectrum ROMs of the chosen machine from `/machines/next/` into the SRAM, and writes your settings into the peripheral NextRegs.

7. **It leaves config mode** by writing the machine type and the display timing to NextReg `$03`.
8. **It soft-resets** with NextReg `$02`, and the CPU starts at `$0000` in the ROM it has just loaded. For the standard menu entry that is NextZXOS.

Who runs all this code? Not a second processor: the board has no microcontroller that starts the machine, and the only step no program takes is the first one, the FPGA loading slot 0 by itself. From then on every step is Z80N code, run by the **CPU inside the core that has just been loaded**. The boot ROM is part of the bitstream, so when the FPGA has configured itself and the reset ends, the CPU does what every Z80 does after a reset: it starts fetching at `$0000`, where the boot ROM answers. The anti-brick core's own Z80N runs the doorman's boot ROM; after the change to slot 1, the Next core's Z80N, the same CPU that will later run NextZXOS, runs the Next core's boot ROM and then the firmware. Both boot ROMs open with `DI`, `IM 1` and a jump, as plain Z80 programs do. They run at **3.5 MHz**, the speed every reset sets, and the firmware's boot module switches to 28 MHz as its first act.

Notice who does what. The FPGA design knows nothing about files, menus or `config.ini`; it provides only a boot ROM of 8K and the switches of config mode. Everything with a user interface is Z80N code on the SD card. That is why the firmware can be updated by copying a file, without touching the flash at all.

The firmware file

`TBBLUE.FW` is not one program but a small library of them. Its first 512-byte block is a **map**: for each module, the block it starts at and how many blocks it is long. The boot ROM reads the map and loads only the module it needs. The modules are the **boot** module, the **updater**, the settings **editor**, the **extra cores** module, a video **test card**, a module that **resets** `config.ini` to its defaults, and the start-up **screens**. A module that needs another one loads it the same way and jumps to it.

The keys you hold decide the module, and **when** you hold them matters. Keys are read at two moments:

1. **As the machine starts**: hold the key **before** you switch on, or before you press **F1** for a hard reset, and keep holding it until the screen changes. The boot ROM looks at the keyboard as soon as it begins, before the start-up screen appears, so a key pressed a moment later is too late for this chance.
2. **While the start-up screen is showing**, with its flashing "Press SPACEBAR for menu" and "Press C for extra cores" prompts: the boot module keeps checking the keys until it starts the machine, so you can also press them then.

A **soft reset** (F4, or the reset button) gives you neither chance: it restarts the chosen ROM without running the boot ROM or the firmware at all. And the doorman's buttons, **DRIVE and M1**, count only when held from **power-on**, because a hard reset starts from slot 1 and never meets the anti-brick core.

Here is what each key does:

- **Nothing**: the boot module, and the machine starts after the start-up screen.
- **Space** (at either moment): the settings editor, where you choose the machine from the menu and change `config.ini`.
- **U** (at either moment): the updater, if a `TBBLUE.TBU` is on the card. Let go of U when the updater's screen appears.
- **C** (at either moment): the extra cores module.
- **A, D, V or R** on the start-up screen: the video test card for all modes, digital (HDMI), VGA or RGB, which lets you choose a video mode your monitor can show.
- **N and X together** on the start-up screen: `config.ini` back to its defaults.

The firmware also checks the core it runs on. If the Next core is older than the oldest one it supports (3.01.10 for firmware 1.44), it refuses to start the machine and tells you to update the core, with U to enter the updater.



KLIVE IDE The firmware viewer

First set a ZX Spectrum Next model as the machine type. Then choose **Machine › SD Card › Show TBBLUE.FW**, or type `show-fw` at the command prompt, and Klive opens the card's `TBBLUE.FW` in a viewer. It shows the firmware's version, the module map, the keys that start each module, each code module as a Z80 disassembly, and the test card picture. The viewer only reads the file, so it is a safe way to see what a release contains.

Config mode

Config mode is the state the Next core starts in, and it is what makes the firmware's job possible. The **Machine Type register**, **NextReg `$03`**, says it: machine type 0 is config mode. While it lasts:

- **The boot ROM** answers reads of `$0000–$3FFF`, until the first write to NextReg `$03` switches it off.
- **The Config Mapping register**, **NextReg `$04`**, maps any 16K bank of the SRAM to `$0000–$3FFF`. The firmware uses it to write the ROM images into the banks where the machine will look for them: the Spectrum ROMs in the first 64K, then the DivMMC's ROM, the Multiface's ROM and RAM, and the alternate ROMs, 128K in all. On the Issue 5 board it keeps all 8 bits, so it reaches the whole 4 MB; on the other boards bit 7 is ignored.
- **The flash can be selected** on the SPI bus (`$7F` on port `$E7`), which the storage chapter's The flash and the Pi describes. The firmware reads a core's header and writes a core this way.
- **The slot number** of NextReg `$10` and the **video timing** of NextReg `$11` can be changed.
- **The soft reset key** F4 does nothing.

The firmware leaves config mode with one write to NextReg `$03`: the machine type (48K, 128K, +3 or Pentagon), the display timing, and bit 7 to allow the timing to change. From that moment the machine type is fixed, the boot ROM is gone, and the flash and the slot number are locked away.

There is one way back, and the register list does not mention it: writing `7` (`111`) to bits 2 to 0 of NextReg `$03` puts the machine back into config mode, from any program. The boot ROM does not return at once, but a soft reset in config mode switches it back on, and the CPU starts the boot ROM again. So the lock is not a wall; it is a door that opens only to a program that knows the exact knock. I find that a fair compromise: no stray write can do it, and a system program that must reach the flash or another slot still can. A hard reset, of course, always starts the whole chain over.

Updating the core

A core update replaces the bitstream in slot 1. It arrives as **`TBBLUE.TBU`**, a file you copy to the root of the SD card, and it is written by the firmware's **updater**. You start it by holding **U** from before you switch on or press F1 until its screen appears, or by pressing U on the start-up screen; then answer **Y** (or press DRIVE) to go ahead. Here is what the updater does:

1. **It reads the file's header**, the first 512 bytes, and checks its magic word.
2. **It picks the core for your board.** The current files carry several cores, one per board, each with its own record; the updater takes the records whose board ID matches NextReg `$0F` and ignores the rest. A file with no core for your board updates nothing.
3. **It shows** the board, the slot and the version it is about to write, and asks.
4. **It checks the image's checksum** by reading the whole image from the card, before it touches the flash. A damaged file stops here, and slot 1 is still the old core.
5. **It erases the slot** in blocks of 32K, then **writes** the image, 256 bytes at a time.
6. **It asks you to switch off and on.** On the Issue 2 board it also asks you to unplug the HDMI cable while the power is off.

The header has a fixed layout (Figure 1.4).

Older files, with the magic word `TBUFILE`, carry one core with a simpler checksum, and the updater checks their machine ID and their exact size instead. Both kinds end the same way: in slot 1.



Figure 1.4: The layout of a TBBLUE.TBU file



KLIVE IDE The update file viewer

First set a ZX Spectrum Next model as the machine type. Then choose **Machine › SD Card › Show TBBLUE.TBU**, or type `show-tbu`, and Klive opens the card's `TBBLUE.TBU`. It lists every core with its board, version and FPGA, checks each image's checksum as the updater does, says whether the image fits the board's slot, and lists the bitstream's configuration packets. Check a file there before you press U on a real Next.

Let me warn you about the one moment that matters. Between the erase and the end of the writing, slot 1 holds half a core. Switch off then, and the Next core is gone. That is not the end of the machine, because of the doorman, but it is the end of an ordinary start-up until you repair it, as the next section shows. So my advice is the boring one: a good power supply, no cats near the cable, and no hurry.

The current system release, **26.11a** (current at the writing of the book), puts **core 3.03.03** into slot 1 on all three boards and comes with **firmware 1.46**. One change matters for the update itself: from core 3.03.00 on, the Issue 2 board's core is split in two, one for HDMI and one for VGA. An Issue 2 machine must therefore update its core first and the rest of the system after it; done the other way round, the new firmware looks for a test card core that is not there yet, and the machine looks bricked although it is not. The Issue 5 board, which has two SD card readers, an internal one and an external one, updates both cards through a new updater program that the release installs first. My advice for any board: read the release's notes for your board before you press U.

Repairing a broken slot 1

If slot 1 does not start, because an update was interrupted or a core is faulty, the doorman is still there. Power on with **DRIVE and M1 held**, and the anti-brick core does not start slot 1: its own boot ROM loads the **updater** module from `TBBLUE.FW`, and the updater runs on the anti-brick core, at 3.5 MHz and with default settings. It

reads `TBBLUE.TBU` and writes slot 1 exactly as an ordinary update does. Nothing of the broken core is used, which is the whole point.

You will know you need this when the Next's start-up screen no longer appears after a power-on, usually after an update that was cut short. Here is the repair, step by step:

1. **Prepare the SD card on another computer.** It must have a FAT16 or FAT32 file system, and its root folder must hold `TBBLUE.FW` and a `TBBLUE.TBU` with a core for your board. Take both from the same release of the Next's system distribution, so that the firmware and the core belong together. The anti-brick core's boot ROM needs nothing else from the card.
2. **Find a picture.** On the anti-brick core the machine may show a picture on VGA only, so connect a VGA monitor if you have one. F2 (the scan doubler) and F3 (50/60 Hz) still work on the anti-brick core, and they are worth a press if the screen stays dark.
3. **Switch off completely**: unplug the power, and on the Issue 2 board the HDMI cable too.
4. **Hold DRIVE and M1, and switch on.** Keep both held until the updater's screen appears, titled **Anti-Brick** in red, then let go: the updater waits until both buttons are released before it goes on.
5. **Check what it shows**: the board, the slot (01, the Next core) and the version it is about to write. Then answer **Y**, or press **DRIVE**, to go ahead; **N** or **M1** cancels.
6. **Wait.** The updater checks the image's checksum, erases slot 1 and writes it, a little more slowly than usual at 3.5 MHz. Do not switch off while it works: this is the same window of risk as before.
7. **When it says "Updated!"**, switch off, unplug the HDMI cable on the Issue 2 board, and switch on normally, with no buttons held. The doorman waves you on to the repaired slot 1.

The anti-brick core's boot ROM and the updater stop with a message when something is wrong. These are the ones you may meet, and what to do about each:

- "**Error Mounting SD Card**": the card cannot be read, or it has no FAT16 or FAT32 file system. Try another card, formatted on a computer.
- "**TBBLUE.FW Not Found**", "**TBBLUE.TBU Not Found**": a file is missing from the card's root folder.
- "**Error Reading TBBLUE.FW**", "**Error Loading UPDATE Module**": the firmware file is damaged or does not belong to this release. Copy it again.
- "**Wrong Magic!**", "**Wrong image size**": the update file is not a `TBBLUE.TBU`, or it is incomplete.
- "**Wrong board id**": the update file is for another board. An older file with one core inside is for the Issue 2 board only; an Issue 4 or Issue 5 board needs a file that has a record for it.
- "**FAIL**" followed by two numbers: the checksum does not match, so the file is damaged. Nothing has been erased yet, so copy the file again and repeat.

None of these messages has touched the flash: every check comes before the erase. The only way to make things worse is to cut the power during the writing, and even then you can start again from step 3.

Extra cores

The flash has room for more than the Next. An **extra core** is a bitstream for another machine, built for the Next's board, kept in a folder of its own on the SD card: `/machines/<name>/core.bit`, often with a `core.cfg` that lists the files the core wants. Press **C** at start-up, and the extra cores module lists the slots and the folders:

- **It recognises a slot's core** by reading the bitstream's header from the flash, the design name, the date and the time the core was built, and matching it with the `core.bit` files on the card.
- **It writes a core** into a slot you choose, from slot 8 up, with the same erase and write as the updater. Slots 0 to 7 are never offered.
- **It starts a core** by writing your basic settings (timing, scan doubler, 50/60 Hz, scanlines, and a few more), the folder name and any files you chose into the SRAM, dropping to 3.5 MHz, and writing `$80` plus the slot number to NextReg `$10`. The FPGA becomes the other machine, and the other machine finds its settings in the SRAM, which survived the change.

A core can also be started without the menu. At every start the boot module looks for a 128-byte **core boot request** at SRAM address `$018000`: the magic word `COREBOOT`, the core's folder and a file name, protected by a checksum. When it finds a valid one, it starts that core at once. Since the SRAM survives a hard reset, a program can write the request and hard-reset, and the firmware does the rest.

Firmware 1.44's extra cores module runs only on the Issue 2 board; on the other boards it stops with a message saying so. The Issue 4 and Issue 5 boards have fewer, larger slots, and the hardware is ready for them: NextReg `$10` takes slots 0 to 14 there. Firmware 1.45, which comes with the Issue 5 board, knows that board by name, and the message that kept extra cores to the Issue 2 board is gone from it.

The Next core is not tied to the Next's board either. NextReg `$00` has values for the Next core running on other FPGA boards, such as `$DA` on a MiSTer, and `$08` for emulators. A program that reads it learns where it runs, and the firmware does exactly that.

▶ TRY IT Core Version

Part I, key 1. The demo reads the machine ID and the core's version and prints them. The code is in `examples/code/demos/1-machine/core-version.kz80.asm`, and three NextReg reads do the work:

▶ TRY IT Core Version *(continued)*

```
    ReadNextReg($00)
    ld (CvMachineId),a
    ReadNextReg($01)
    ld (CvVersion),a
    ReadNextReg($0E)
    ld (CvSubMinor),a
```

NextReg `$01` holds the core's version, the major number in bits 7–4 and the minor in bits 3–0, the same layout as a core's record in `TBBLUE.TBU`; NextReg `$0E` holds the sub-version. `ReadNextReg` is a macro: it loads the register's number into A and calls a routine that selects the register through port `$243B` and reads it through port `$253B`, the two ports Chapter 3 explains. In Klive the screen shows machine ID `$08`, the emulators' value, and the core version of the model you chose, such as 3.02.01. On a real Next it shows `$0A` and the version of the core you installed: 3.03.03, for example, with the 26.11a release, which NextReg `$01` reports as `$33` and NextReg `$0E` as `$03`.

In short. The FPGA forgets its bitstream at every power-off, so it loads the anti-brick core from flash slot 0, which normally starts the Next core in slot 1 by writing `$81` to NextReg `$10`. The boot ROM then loads `TBBLUE.FW` from the SD card, and the firmware builds the machine in config mode and leaves it with a write to NextReg `$03`. The SRAM survives a reconfiguration, so one core can leave a message for the next. Never cut the power while the updater erases and writes slot 1; to repair it, hold DRIVE and M1 at power-on.

CHAPTER 2

The Z80N: A Familiar Heart and Twenty-Nine New Tricks

Take a Z80 program written for the Spectrum in 1983, load it on a Next, and it runs. Every instruction takes the same number of T-states, every flag lands where it did, and the music player's carefully counted loops still keep time. Yet there is no Z80 chip anywhere on the Next's board. The CPU is a description of a Z80, written in a hardware language and built into the FPGA beside the ULA, the sprites and everything else. So how faithful is that copy, where does it quietly differ from the original, and what did its authors add while they were at it?

I think the CPU is the one part of the Next that every program meets in every cycle. Your raster effect lives or dies by T-states, your music player by the length of a loop, your copy routine by how many bytes a cycle can move. Knowing what the CPU does on the bus explains why an instruction takes as long as it does. And the Z80N's new instructions are a gift: a multiply, barrel shifts and screen arithmetic in eight T-states each, where a Z80 needs a subroutine.

- How an instruction breaks into machine cycles, and how long it takes at each of the four speeds
- How the CPU shares the bus with the DMA and responds to interrupts
- What the undocumented flags and instructions do, and where the Next's CPU differs from a Zilog chip
- What the twenty-nine Z80N instructions do, and how long each one takes

A Z80 without a chip

The Z80 appeared in 1976, made by Zilog in NMOS, the transistor process of its day; a CMOS version, which uses less power, followed in the 1980s. The two behave the same in every documented way and differ in a few undocumented ones, which will matter later. The ZX Spectrum used the NMOS chip.

The Next's CPU, the **Z80N**, is neither. It began life as an open-source description of a Z80 written for FPGAs, and the Next's designers corrected its timing, completed its undocumented behaviour and added the new instructions.

I like to call the Z80N **the understudy**. It learned the lead's part so well that the audience cannot tell who is on stage: the same instructions, the same T-states, the same documented flags. But an understudy also brings a few habits of its own, and the Z80N improvises a handful of lines that the original never spoke. You will meet them in the second half.

What being a description rather than a chip means in practice:

- **No analogue behaviour.** An NMOS chip has electrical quirks, such as how long a floating bus keeps its last value. The Z80N has none of its own.
- **The same T-states.** Every Z80 instruction takes exactly as many T-states as on a Zilog chip, including the undocumented ones and the conditional ones taken or not taken.
- **Four speeds.** The CPU Speed register, **NextReg `$07`**, runs it at 3.5, 7, 14 or 28 MHz.

Registers, and one you cannot see

The Z80N has the Z80's registers and not one more; none of its new instructions needed a new register.

- **The main set**: `A` and `F`, and the pairs `BC`, `DE` and `HL`.
- **The alternate set**: `A'`, `F'`, `BC'`, `DE'` and `HL'`, swapped in with `EX AF,AF'` and `EXX`.
- **The index registers** `IX` and `IY`, the stack pointer `SP` and the program counter `PC`.
- **`I`**, the high byte of the IM 2 table, and **`R`**, the refresh counter: its low seven bits count every opcode fetch, and bit 7 keeps whatever `LD R,A` last put there.
- **The interrupt state**: the two flip-flops **IFF1** and **IFF2**, and the interrupt mode, 0, 1 or 2.

The one you cannot see is a scratch register. **WZ**, also called **MEMPTR**, is a 16-bit register the CPU uses to build addresses: `LD A,(nn)`, `JP nn`, `CALL nn` and many others leave a value in it. No instruction reads it, but on a Zilog chip it leaks out through the flags of `BIT n,(HL)`. On the Next it does not, and we will come back to that.



KLIVE IDE The Z80 CPU panel

Pause the machine and open the **Z80 CPU** panel in the **Debug** activity. It shows every register of this section, `WZ` included: step over a jump or a load from memory and watch the value it leaves there. `IM`, `IF1` and `IF2` show the interrupt state, and `HLT` whether the CPU is halted.

Machine cycles and T-states

The Z80 does nothing in one go. Every instruction is a short series of **machine cycles**, M-cycles for short, and each M-cycle is one of a few kinds of conversation with the bus, a few T-states long. A **T-state** is one tick of the CPU's clock, 286 nanoseconds at 3.5 MHz. There are four kinds of M-cycle you need to know:

- **The opcode fetch, M1**: 4 T-states. It reads an opcode byte at `PC` and refreshes memory.
- **The memory read**: 3 T-states, for operands and data.
- **The memory write**: 3 T-states.
- **The I/O cycle**: 4 T-states, a read or a write on a port.

Some instructions add a few T-states of **internal work** to one of their cycles, while nothing happens on the bus. Take `LD A,(nn)`: an opcode fetch (4), two operand reads for the address (3 + 3), and a data read (3) make 13 T-states. `LD (nn),A` is the same, with a write in place of the last read, and also takes 13.

Prefixes are opcodes too. `CB`, `ED`, `DD` and `FD` each take an opcode fetch of their own, so an `ED` instruction is at least 8 T-states long and counts `R` up by two. The `DD CB` and `FD CB` instructions are the exception: after the two prefixes, the displacement and the final opcode are read as plain memory reads, so `R` counts only two.



KLIVE IDE Counting T-states

Put one breakpoint where a piece of code starts and one where it ends, and run to the second. The Z80 CPU panel's `TSP` value is the T-states the code took since the first stop: 13 for `LD A,(nn)`. Disable interrupts around the code, or a frame interrupt's handler is counted too.

The opcode fetch

The first M-cycle of every instruction does two jobs, one after the other. Here it is, signal by signal (Figure 2.1).

1. **T1**: the CPU puts `PC` on the address bus and pulls **`/M1`** low, so the world knows this is an opcode fetch. Half a T-state later it pulls **`/MREQ`** (memory request) and **`/RD`** (read) low.
2. **T2**: the memory has until the end of this T-state to put the byte on the data bus.
3. **Start of T3**: the CPU takes the byte, lets `/M1`, `/MREQ` and `/RD` go, and starts decoding.
4. **T3 and T4**: the **refresh**. The CPU puts `I` in the address bus's high half and `R` in its low half, pulls **`/RFSH`** low, gives a short `/MREQ` pulse, and adds 1 to `R`'s low seven bits.

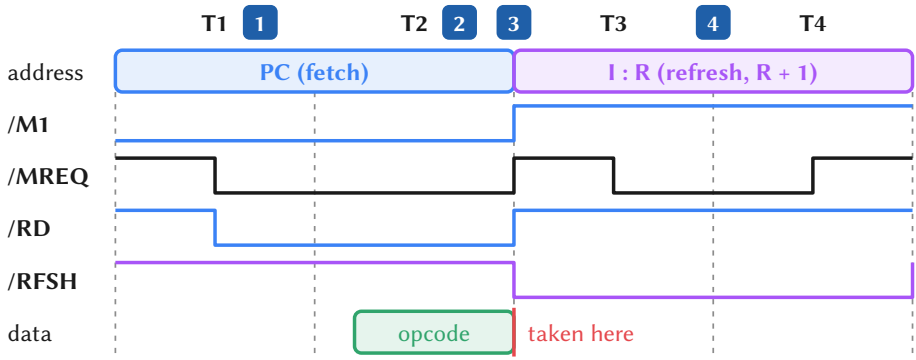


Figure 2.1: The opcode fetch cycle

Why refresh? The Spectrum's RAM was **dynamic**: each bit is a tiny capacitor that forgets in a few milliseconds unless something reads its row. The Z80 does that for free in the half of the M1 cycle where it is busy decoding anyway. I find it one of the cleverest ideas in the chip: a chore done in time nobody else wanted.

The Next's memory is **static RAM**, which never forgets, so it needs no refresh. The Z80N still runs the refresh half, because it is part of the cycle's timing, and `R` still counts. The Next keeps its memory out of the refresh cycle and passes `/RFSH` to the expansion bus, for a peripheral that might want it.

Reading, writing and talking to ports

Every M-cycle after the opcode fetch is one of four kinds: a memory read, a memory write, an I/O read or an I/O write. A **memory read** is the opcode fetch's first half with one T-state more (Figure 2.2).

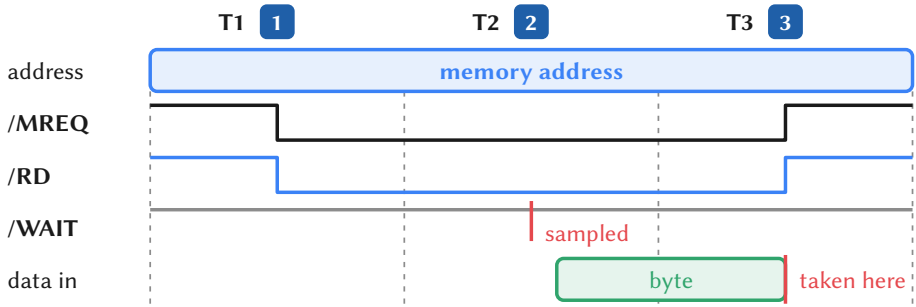


Figure 2.2: A memory read, to scale in T-states. `/MREQ` and `/RD` are low from the middle of T1 to the middle of T3, and the CPU takes the byte at that last edge, half a T-state later than in an opcode fetch.

1. **T1**: the CPU puts the address on the bus. Half a T-state later it pulls **/MREQ** and **/RD** low.
2. **Middle of T2**: the CPU looks at **/WAIT**. If a device holds it low, a wait state, TW, comes between T2 and T3, and the CPU looks again in the middle of each one.
3. **Middle of T3**: the CPU takes the byte and lets /MREQ and /RD go. With no refresh to do, the cycle ends here: three T-states, where the fetch took four.

A **memory write** has the same shape, but the CPU drives the data bus and the strobe is **/WR** (Figure 2.3).

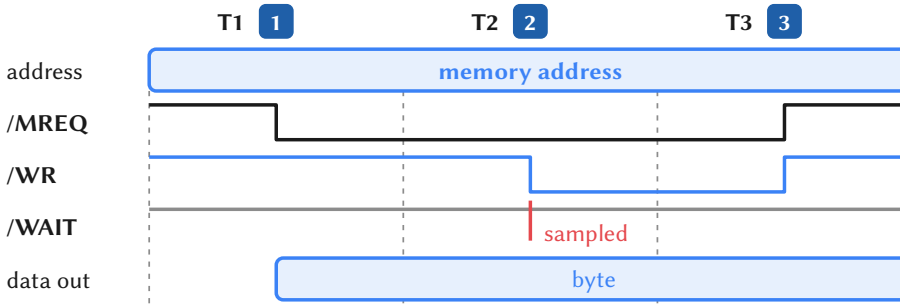


Figure 2.3: A memory write, to scale in T-states. The byte is on the data bus from the middle of T1, and /WR is low only from the middle of T2 to the middle of T3, while address and data are both stable.

1. **T1**: the address goes out. Half a T-state later /MREQ falls and the byte goes on the data bus.
2. **Middle of T2**: with the data settled, **/WR** falls; this is the moment the CPU looks at /WAIT.
3. **Middle of T3**: /WR and /MREQ rise, and the memory keeps the byte. The address and the data stay half a T-state longer, so the memory never sees them change under a live /WR.

An **I/O cycle** uses **/IORQ** instead of /MREQ, and it has one T-state more, an automatic wait state, because the peripherals of 1976 needed the time. IN and OUT share the cycle and differ in who drives the data bus (Figure 2.4).

1. **T1**: the port address goes out on all 16 lines. For an OUT, the byte follows half a T-state later.
2. **Start of T2**: **/IORQ** falls, with /RD for an IN or /WR for an OUT, half a T-state later than /MREQ would. The port has had a whole T-state to decode its address.
3. **TW**: the wait state every I/O cycle gets for free. The CPU looks at /WAIT in its middle, so a slow port can ask for more.
4. **Middle of T3**: an IN takes the byte, and all the strobes rise.

Two of the I/O cycle's details shape the Next:

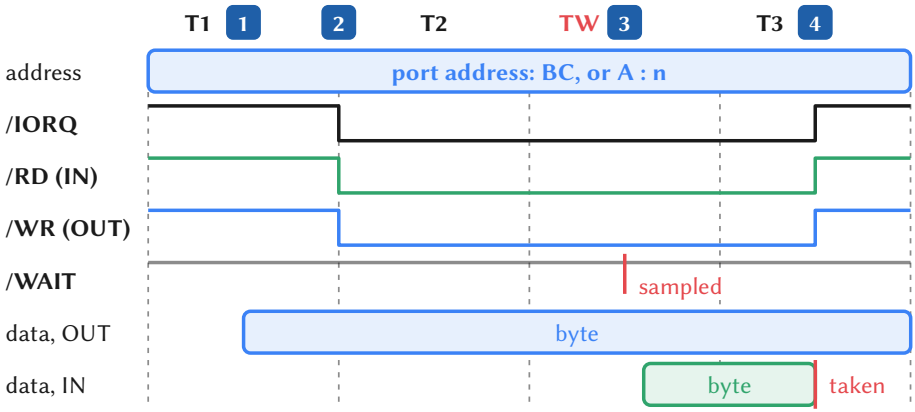


Figure 2.4: An I/O cycle, to scale in T-states, with the automatic wait state TW between T2 and T3. The strobes fall a whole T-state into the cycle, at the start of T2, and an `IN` takes its byte in the middle of T3.

- **The port address is 16 bits.** Zilog's manual talks about 256 ports, but the CPU always drives the whole address bus: `IN r,(C)` and `OUT (C),r` put `BC` on it, and `IN A,(n)` and `OUT (n),A` put `A` in the high half and `n` in the low half. The Next decodes most of its ports with all 16 bits, which the memory chapter's Ports: which bits count goes through.
- **A device can ask for more time.** Any cycle stretches while **`/WAIT`** is low, a T-state at a time. On the Next, four things hold it: contention at 3.5 MHz, the memory at 28 MHz on the older core 3.02, a palette read that must wait for the Copper, and a peripheral on the expansion bus.

Contention, the ULA holding the CPU back while both want the screen memory, exists only at 3.5 MHz, in the 48K, 128K and +3 timings; the memory chapter's Contention has the rules. The CPU's four speeds deserve a section of their own, and so does the wait state that the older core added at 28 MHz.



KLIVE IDE I/O breakpoints

An **I/O read** or **I/O write** breakpoint stops the machine when the CPU reads or writes a port: all 16 bits of it, or only the bits a port mask keeps. Set one in the Breakpoint dialog, or type `bp-set $FE -i` for reads of port `$FE`. The Z80 CPU panel's `IRV` and `IWV` show the last byte read from and written to a port.

Four speeds

The CPU runs at **3.5**, **7**, **14** or **28 MHz**, chosen by NextReg `$07`, and at every one of them an instruction takes exactly the T-states of the Zilog tables; a T-state only gets shorter. At **28 MHz** it lasts 36 nanoseconds, so `NOP` takes 4 T-states, `LD A,(nn)` 13 and

`POP BC` 10, at 28 MHz as at 3.5. The DMA keeps its six clocks a byte too (Chapter 5). That is the rule of **core 3.03**, the core of the 26.11a system release, on every board: Issue 2, Issue 4 and Issue 5. The older core 3.02 added a wait state to every memory read at 28 MHz, and the subsection after the demo tells that story for the machines that still run it.

Here are two instructions that look alike, at both ends of the speed range, and on both cores (Figure 2.5).

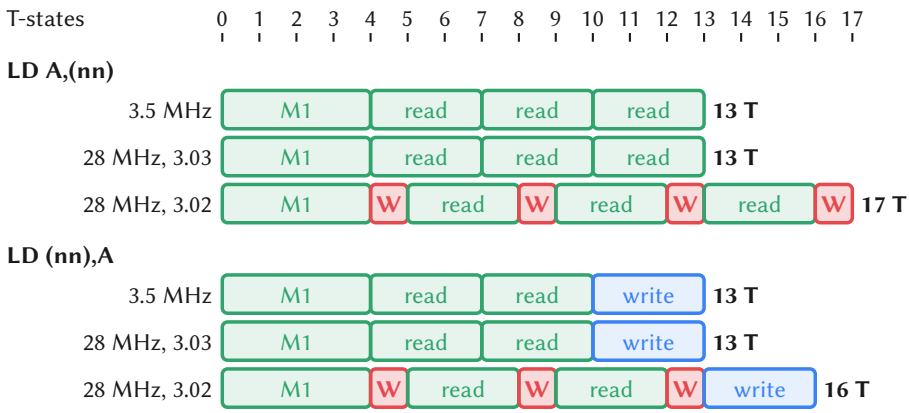


Figure 2.5: A load and a store at 3.5 MHz, and at 28 MHz on core 3.03 and on the older core 3.02 (W: wait state)

The top two bars of each instruction are the rule: the same T-states at 3.5 MHz and at 28 MHz, only eight times shorter. This is not a guess. A program that draws 32 small pictures five different ways, with the CPU and with the DMA, was timed on an Issue 5 board with the new core, and a machine with no wait states reproduces every one of its times to a tenth of a millisecond, while one with core 3.02's wait state misses each by 1 to 3 milliseconds (Figure 5.7). Those measurements cover the static RAM, where programs live. Whether reads from bank 5, which sits in the FPGA's own block RAM, are free of the wait as well has not been measured yet, so a loop whose exact length matters is better timed than counted if it runs from bank 5.

My advice is still to count the T-states of a timed loop at the speed it will run at, on the core it will run on, and not to scale a 3.5 MHz count by eight: contention, at 3.5 MHz only, stretches the slow count but not the fast one, and an older core stretches the fast one.

▶ TRY IT Four Speeds

Part I, key 2. The demo sets each speed in turn, counts how many times a short loop runs in one frame, and prints the four counts. The code is in `examples/code/demos/1-machine/four-speeds.kz80.asm`. `FsMeasure` counts loops from one frame interrupt to the next:

```
    nextreg $07,a           ; 0-3: 3.5, 7, 14, 28 MHz
    call FsMeasure          ; HL: loops in one frame
```

```
FsMeasure:
    halt                    ; start right after an interrupt
    ld hl,0
    ld a,(Frames)
    ld b,a
FsMeasureLoop:
    inc hl
    ld a,(Frames)
    cp b
    jr z,FsMeasureLoop
    ret
```

The counts double from 3.5 to 7, from 7 to 14 and from 14 to 28 MHz: one pass of the loop is 35 T-states (`INC HL` 6, `LD A,(nn)` 13, `CP B` 4, `JR Z` 12) at every speed, so the fourth count comes out close to eight times the first. On a machine still on core 3.02 it comes out only about 6.5 times; the next subsection tells why, so the demo also tells you which of the two cores you are running.

Core 3.02: one wait state at 28 MHz

The older core 3.02 adds **one wait state to every memory read** at 28 MHz, on every board: opcode fetches, operand bytes and data alike. Its memory takes a whole 28 MHz clock to answer, so at that speed the byte comes one T-state late, and the core holds **`/WAIT`** low for it. Writes and I/O cycles get no wait, and neither do reads from page `$0E`, the half of bank 7 in the FPGA's block RAM; bank 5's reads wait like the rest. The third bar of each instruction in Figure 2.5 shows the waits.

So add one T-state for every byte an instruction reads: `NOP` takes 5, `POP BC` 13 instead of 10, and `LD (IX+d),n` 23 instead of 19. The Four Speeds loop reads memory eight times a pass, so a pass takes 43 T-states instead of 35, and 8 × 35 ÷ 43 is 6.5. A DMA byte from memory to memory takes seven clocks instead of six. A delay loop counted for 3.02 comes out short on 3.03, so when a loop's length matters, read the core version from NextReg `$01` (`$33` for 3.03) and NextReg `$0E` (`$03` for 3.03.03), as the Core Version demo of Chapter 1 does.

Sharing the bus

The CPU is not the only master of the Next's bus. The **DMA** can move memory and ports on its own, and to do that it asks the CPU for the bus with **/BUSRQ**. The CPU lets go at the end of the **M-cycle** it is in, not at the end of the instruction, and says so with **/BUSAK**. While the DMA holds the bus, the CPU stands still in the middle of its instruction, does no refresh, and cannot take an interrupt. When the DMA lets go, the instruction carries on as if nothing had happened, only later.

That last point is why the DMA can make an interrupt pulse vanish, and why the Next has a way to make the DMA step aside. Both are in Chapter 5 and in the interrupts chapter's The DMA steps aside.

HALT is the other way the CPU stops, but it stops working, not fetching. A halted CPU runs opcode fetches of 4 T-states each and executes them as `NOP`s, so `R` keeps counting and the refresh keeps going, while `PC` already points past the `HALT`. When an interrupt comes, the CPU leaves the halt and the interrupt returns to the instruction after it.

Interrupts, from the CPU's side

A program can ask a device, again and again, whether anything has happened. That is **polling**, and it spends the CPU's time on asking. An **interrupt** turns the question around: the device signals when it needs attention, the CPU finishes its instruction, saves `PC` on the stack and jumps to a **handler**. The handler does its work and returns, and the program carries on, none the wiser, as long as the handler leaves the registers as it found them.

The Z80 has two interrupt inputs, for two kinds of interrupt.

Maskable interrupts

The everyday kind arrives on **/INT**. It is **maskable**: the program switches it off with **DI** and on with **EI**, which control a flip-flop in the CPU, **IFF1**. Accepting an interrupt clears IFF1 too, so a handler is not interrupted until it executes `EI` itself.

`/INT` is a **level**. A device holds it low until it is served, and the line has no memory: if it goes high before the CPU answers, the request is gone.

On the Next, `/INT` serves the **frame interrupt**, 50 or 60 times a second, which the Spectrum's ROM uses to read the keyboard and count time; the **line interrupt**, at a raster line the program chooses; the **CTC**'s four timers; and the two **UARTs**. The Next joins them onto the one wire in one of two ways:

- **Pulse mode**, the Spectrum's way and the default: any source starts the same short pulse, and a handler that serves several sources has to find out which one rang.

- **Hardware IM2 mode**: a source holds `/INT` until it is answered, so no request is lost, and hands the CPU a byte of its own, so in IM 2 each source gets its own handler. The source with the higher priority goes first.

The non-maskable interrupt

The other input, **`/NMI`**, is **non-maskable**: `DI` cannot stop it. On the Spectrum it became the **freeze button**: an add-on such as the Multiface pulls `/NMI` low, the CPU jumps to **`$0066`**, and the add-on pages its own ROM in at that address, so it can stop any game at any moment. The Next has two such buttons, **M1** for its built-in Multiface and **DRIVE** for the DivMMC's NMI handler, and a program, or even a Copper list, can raise the same NMIs from software.

`/NMI` is an **edge**: the CPU remembers that it fell until it can answer, and the NMI wins when both kinds are waiting. Since it can arrive in the middle of anything, the CPU keeps a second flip-flop, **IFF2**, to remember whether interrupts were on. `EI` and `DI` set and clear both; an NMI clears IFF1 and leaves the old value in IFF2, and the handler's **`RETN`** puts it back.

When the CPU looks

The CPU samples `/INT` once per instruction, at the start of the **last T-state**, and answers when the instruction ends. It never answers between a prefix (`CB`, `DD`, `ED`, `FD`) and the rest of its instruction, or right after `EI`. Here is the end of a handler, with a new request already waiting (Figure 2.6).

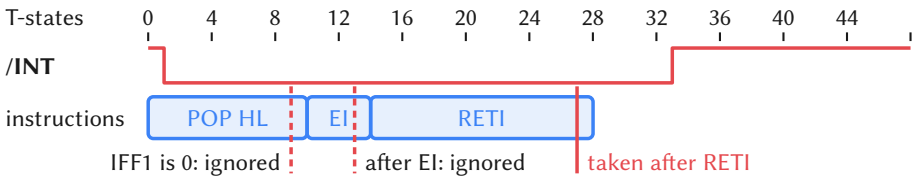


Figure 2.6: The instruction after EI

The gap after `EI` is what makes `EI` followed by `RETI` safe: the return always happens before the next interrupt, so return addresses can never pile up on the stack. A halted CPU keeps looking too: `HALT` executes NOP cycles of 4 T-states and samples `/INT` in each one.

The acknowledge cycle

When the CPU accepts an `/INT`, it clears IFF1 and IFF2 and runs a special M1 cycle, the **interrupt acknowledge**, to ask the device for a byte (Figure 2.7).

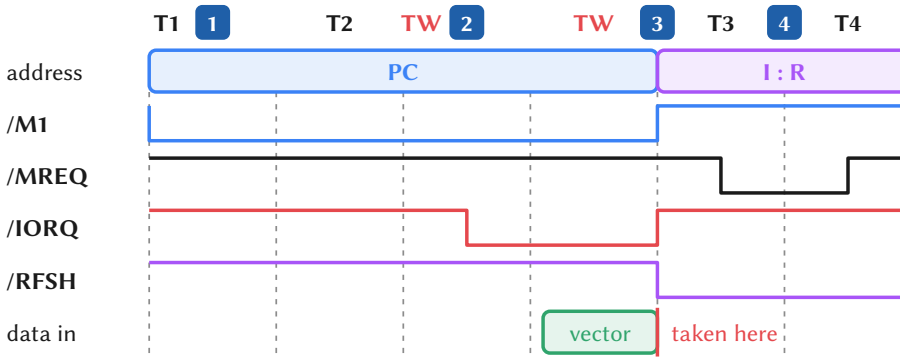


Figure 2.7: The interrupt acknowledge

1. **T1**: `PC` goes out and **`/M1`** falls, as for an opcode fetch, but `/MREQ` and `/RD` stay high: no memory is read.
2. **The two wait states** give a chain of devices time to agree which one answers. Half way through the first, **`/IORQ`** falls; `/M1` and `/IORQ` low together mark an acknowledge, and the winning device puts its byte on the data bus.
3. **Start of T3**: the CPU takes the byte, the **vector**, and lets `/M1` and `/IORQ` go.
4. **T3 and T4**: the refresh, as in every M1 cycle.



KLIVE IDE Interrupts in the Execution History

In a debug session the **Execution History** lists the instructions the CPU ran. An interrupt gets a row of its own that names its mode and its vector, and the handler's first instruction follows it. Switch on **Fold interrupts** to shrink each handler to one row.

The three interrupt modes

The **interrupt mode**, set by `IM 0`, `IM 1` or `IM 2`, decides what happens next (Figure 2.8).

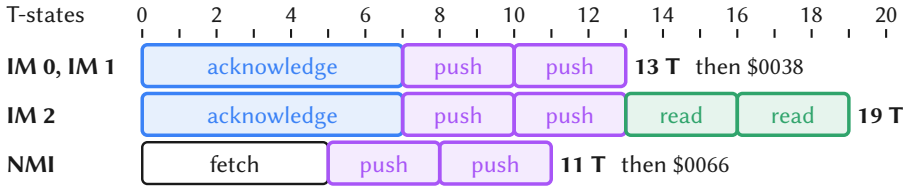


Figure 2.8: The responses to an interrupt

- **IM 0** executes the byte as an instruction, usually an `RST`. It is the 8080's way, and Spectrum software does not use it.
- **IM 1** ignores the byte and calls **`$0038`**: one handler for everything, **13 T-states** in all. The Spectrum's ROM runs in IM 1, and its routine there is the keyboard and clock handler.
- **IM 2** reads the handler's address from a table, low byte first, and calls it: **19 T-states**. **`I`** is the table address's high byte and the vector its low byte, so the table can sit anywhere in memory and tell 128 devices apart. Games use IM 2 to replace the ROM's handler, and the Next's hardware IM2 mode uses it to give every source its own.

At 28 MHz IM 1 takes 13 T-states and IM 2 takes 19, as at every speed. On the older core 3.02 only the reads get wait states: IM 1 still takes 13 T-states, and IM 2, which reads the table, 21.

The NMI's response

An NMI needs no device to name it. The CPU clears IFF1, copies the old IFF1 into IFF2, runs an opcode fetch at `PC` whose byte it ignores, and pushes the return address, then continues at **`$0066`**: 11 T-states. The handler can check IFF2: `LD A,I` and `LD A,R` copy it into the P/V flag.

`RETN` and **`RETI`** both pop the return address and copy IFF2 into IFF1. They differ only in what devices see: Zilog's peripherals watch the bus for `RETI`'s opcode to learn that their handler has finished, and the Next's interrupt controller, in hardware IM2 mode, keeps a source in service until then.

In pulse mode no device drives the data bus during the acknowledge, so the Next answers with **`$FF`**. In IM 0 the CPU executes it as `RST $38`, so IM 0 behaves exactly like IM 1; in IM 2 the CPU reads its handler's address from `I`:`$FF` and the byte after it.

Chapter 4 covers the Next's sources, priorities, vectors and NMI devices, and its section "How fast the CPU answers" (page 64) adds the wait for the running instruction to the costs above.

The flags, documented and not

The flag register `F` has eight bits, and Zilog documented six of them. The other two were marked "not used", and they are used all the time (Figure 2.9).

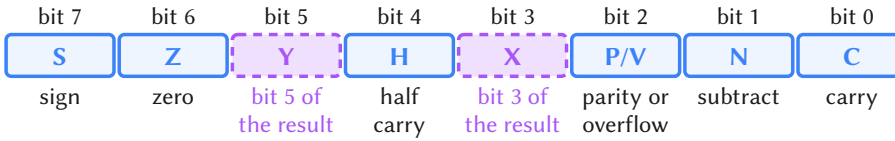


Figure 2.9: The flag register (dashed: undocumented by Zilog)

I will call bit 5 **Y** and bit 3 **X**. The general rule is simple: an instruction that changes the flags copies **bits 5 and 3 of its result** into Y and X. `ADD A,B` that gives `$28` sets Y (bit 5 of `$28` is 1) and X (bit 3 is 1). The exceptions are where it gets interesting:

- **`CP n`** takes Y and X from the operand, not from the subtraction it did, on both the Zilog chip and the Next. So `CP $28` sets both, whatever `A` holds.
- **`BIT n,r`** takes them from the register it tested.
- **`BIT n,(HL)`** on a Zilog chip takes them from the high byte of **WZ**, the hidden register. This is how WZ was discovered, and the only way a program can see it.
- **`SCF` and `CCF`** on a Zilog chip take them from `A`, mixed with the old flags in a way that depends on whether the instruction before changed the flags.

Undocumented does not mean invisible. No instruction tests Y or X directly: there is no jump, call or return on them, and they take no part in any calculation. But `F` is a register like any other the moment it leaves the CPU. `PUSH AF` writes it to the stack next to `A`, and any `POP` brings it back into a register a program can test:

```
; Was Y, bit 5 of F, set by the last instruction?
        push af          ; F goes to the stack, below A
        pop bc           ; and comes back in C
        bit 5,c          ; Z is 0 if Y was 1
```

So Y and X are part of every instruction's visible result. A checksum over the pushed flags changes with them, exerciser programs compare all of `F` after every instruction, and a program can tell one Z80 from another by how `SCF` and `CCF` set them. The Next matches a Zilog chip on every documented flag and on most of Y and X; the differences come two sections on.



KLIVE IDE The flag row

The top row of the Z80 CPU panel shows all eight bits of `F`, one column each: `S`, `Z`, `5`, `H`, `3`, `P`, `N` and `C`. Columns `5` and `3` are Y and X, so step through `SCF` or `CCF` and watch them change, without the `PUSH AF` trick above.

Undocumented instructions

The Z80's opcode tables have holes, and most of the holes do something. Programs found them, used them, and shipped with them, so the Next runs every one of them as a Zilog chip does:

- **The index registers' halves.** A `DD` or `FD` prefix before an instruction that uses `H` or `L` makes it use the high or low half of `IX` or `IY` instead: `LD A,IXH`, `INC IYL`, `ADD A,IXL`. Assemblers call them `IXH`, `IXL`, `IYH` and `IYL` (or `XH`, `XL`, `YH`, `YL`).
- **`SLL`**, the opcodes `CB 30` to `CB 37`: shift left and put a **1** into bit 0, where `SLA` puts a 0. Some assemblers call it `SL1` or `SLI`.
- **The indexed bit instructions with a copy.** `DD CB d xx` with a register in its low bits, such as `RLC (IX+d),B`, rotates the byte in memory and also copies the result into `B`. `RES` and `SET` do the same.
- **`IN (C)`**, also written `IN F,(C)`, opcode `ED 70`: reads the port, sets the flags as `IN r,(C)` would, and throws the byte away. A handy way to test bit 7 of a port in `S` without spending a register.
- **`OUT (C),0`**, opcode `ED 71`: writes a byte to port `BC`. On an NMOS Zilog chip that byte is 0, on a CMOS one it is `$FF`. On the Next it is **0**.
- **Duplicates.** `NEG` has eight opcodes (`ED 44`, `4C`, … `7C`), `RETN` seven (`ED 45`, `55`, … `7D`, next to `RETI` at `ED 4D`). `ED 4E`, `66` and `6E` set IM 0, `ED 76` IM 1, `ED 7E` IM 2. `ED 63` and `ED 6B` are long forms of `LD (nn),HL` and `LD HL,(nn)`, 4 T-states slower than the usual ones.
- **Everything else in the `ED` page** does nothing and takes 8 T-states: two opcode fetches.
- **Prefix chains.** A `DD` or `FD` before an instruction that uses no `HL` costs 4 T-states and changes nothing, and in a chain of them the last one wins.

The `ED` page is also where the Z80N put its new instructions, in holes no Zilog program ever used (Figure 2.10).

Where the copy departs from the original

This section compares the Zilog Z80 chip with its implementation in the ZX Spectrum Next's FPGA, the Z80N. The differences are the understudy's own lines. Every one of them is about the two undocumented flags, the hidden register or a byte the CPU puts on the bus; none changes a documented flag, a register or a T-state.

- **`BIT n,(HL)` and the documented `BIT n,(IX+d)`** clear Y and X on the Next. The undocumented forms of `BIT n,(IX+d)` copy them from the byte tested. A Zilog chip fills them from WZ's high byte in every form. The consequence is bigger than it looks: **WZ is invisible on the Next.** The register exists, the CPU uses it to build addresses, but no program can ever observe it.
- **`SCF` and `CCF`** copy Y and X from `A`, always. A Zilog chip does so only after an instruction that changed the flags; otherwise it mixes in the old Y and X.

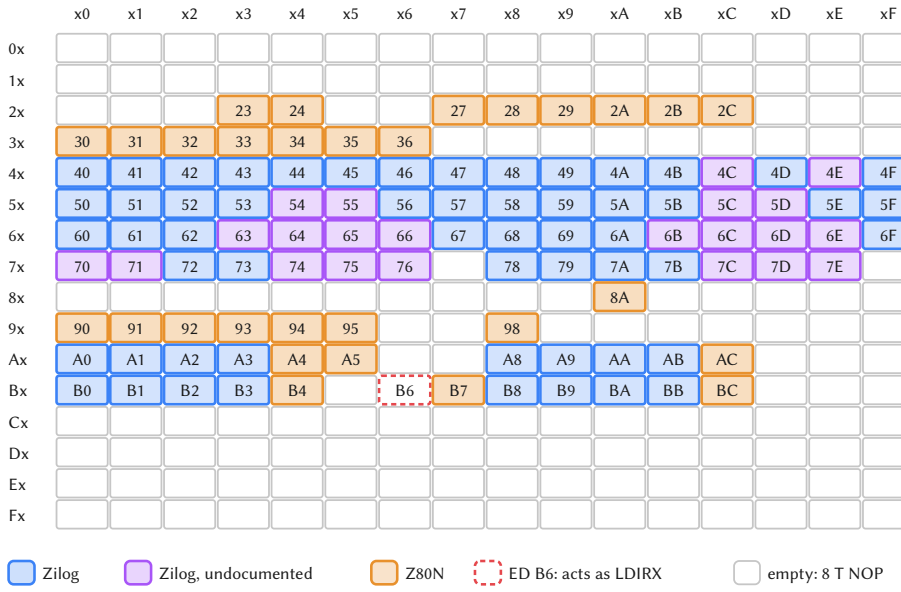


Figure 2.10: The ED page of the Next's CPU

- **`IN r,(C)`, `IN (C)`, `LD A,I` and `LD A,R`** keep Y and X as they were. A Zilog chip copies them from the byte read or loaded.
- **`CPI`, `CPD`, `CPIR` and `CPDR`** copy Y and X from the byte they compared. A Zilog chip takes them from bits 3 and 1 of `A` minus the byte, minus the half carry.
- **`INI`, `IND`, `OUTI`, `OUTD` and their repeating forms** set the flags of decrementing `B`: `N` set, `S` and `Z` from the new `B`, `H` when the decrement borrowed from bit 4, `P/V` when `B` was `$80`, and `C` when `B` was 0. A Zilog chip computes `N`, `H`, `C` and `P/V` from the byte moved and from `C` or `L`, in a formula people found by testing the chip.
- **A repeating block instruction** (`LDIR`, `CPIR`, `INIR`, `OTIR` and the others) sets the same flags on every repetition as on the last one. A Zilog chip, while it repeats, takes Y and X from the program counter.
- **`LD A,I` and `LD A,R`** copy IFF2 into `P/V`, correctly, even when an interrupt is taken right after. In that case an NMOS Zilog chip puts 0 in `P/V`, as if interrupts were off, a known fault that the CMOS chip fixed.

So which chip is the Next, NMOS or CMOS? Neither: it writes 0 with `OUT (C),0` like the NMOS chip, gets `LD A,I` right like the CMOS one, and has its own rules for Y and X. Test programs for the undocumented flags fail on a Next, though real games do not care, so never let your code depend on Y, X or WZ.

Twenty-nine new instructions

The Z80N's additions all live in the `ED` page, behind the prefix, so each costs at least 8 T-states. A Zilog chip runs every one of them as an 8-T-state `NOP`, so a program that uses them does nothing useful on a Spectrum, and does it quietly. They come in groups, and I will take them one group at a time. Unless a group says otherwise, an instruction leaves the flags alone.

Arithmetic

Instruction	Opcode	T-states	What it does
`MUL D,E`	`ED 30`	8	`DE` = `D` × `E`, unsigned
`ADD HL,A`	`ED 31`	8	`HL` = `HL` + `A`, unsigned; clears carry
`ADD DE,A`	`ED 32`	8	the same for `DE`
`ADD BC,A`	`ED 33`	8	the same for `BC`
`ADD HL,nn`	`ED 34 lo hi`	16	`HL` = `HL` + `nn`
`ADD DE,nn`	`ED 35 lo hi`	16	`DE` = `DE` + `nn`
`ADD BC,nn`	`ED 36 lo hi`	16	`BC` = `BC` + `nn`
`TEST n`	`ED 27 n`	11	the flags of `AND n`, and `A` stays

`MUL D,E` is the one I would miss most if it went away: an 8 × 8 multiply in 8 T-states, where a Z80 loop needs well over a hundred. Its result fits `DE` exactly, so it never overflows.

`ADD rr,A` adds `A` as an unsigned byte, so it moves a pointer forward by up to 255, and it wraps from `$FFFF` to 0. Let me warn you about its carry: it is **always cleared**, even when the sum wraps. Do not test the carry after it to detect a wrap; compare the result instead. The other flags stay. **`ADD rr,nn`** leaves every flag alone, carry included, and wraps the same way.

`TEST n` is to `AND n` what `CP n` is to `SUB n`: the flags without the result, so you can test bits of `A` without a copy.

Bits and shifts

Instruction	Opcode	T-states	What it does
`SWAPNIB`	`ED 23`	8	swaps the two halves of `A`
`MIRROR A`	`ED 24`	8	reverses the order of `A`'s bits
`BSLA DE,B`	`ED 28`	8	`DE` shifted left by `B`, zeros in
`BSRA DE,B`	`ED 29`	8	`DE` shifted right by `B`, copies of bit 15 in
`BSRL DE,B`	`ED 2A`	8	`DE` shifted right by `B`, zeros in
`BSRF DE,B`	`ED 2B`	8	`DE` shifted right by `B`, ones in
`BRLC DE,B`	`ED 2C`	8	`DE` rotated left by `B`

A **barrel shift** moves a value any number of places in one step, where `SLA` moves it one. The shifts use bits 4 to 0 of `B`, so a count from 0 to 31: 16 or more shifts every bit out, and `BSRA DE,B` then leaves `$0000` or `$FFFF` by the sign. `BRLC` uses bits 3 to 0, because rotating by 16 changes nothing. `MIRROR A` is a gift for sprites: flip a row of 1-bit graphics horizontally in one instruction.

The stack and a jump

Instruction	Opcode	T-states	What it does
`PUSH nn`	`ED 8A hi lo`	23	pushes the value `nn`
`JP (C)`	`ED 98`	13	jumps by a byte read from port `BC`

`PUSH nn` saves loading a register to push a constant, for a return address or a parameter. Notice its operand: unlike every other 16-bit operand of the Z80, it is stored **high byte first**. Your assembler takes care of that, but a disassembler that does not know the Z80N will show nonsense.

`JP (C)` reads a byte from port `BC` and jumps to it, times 64, inside the 16K block the next instruction is in. If `JP (C)` sits at `$8000` and the port answers `$10`, it jumps to `$8000` + `$10` × 64 = `$8400`. It was made for jump tables driven by hardware, such as a byte read from the expansion bus.

The screen

Instruction	Opcode	T-states	What it does
PIXELAD	ED 94	8	HL = the ULA screen address of the dot at E, D
PIXELDN	ED 93	8	HL = the address one dot row down
SETAE	ED 95	8	A = the bit of the dot in its byte

The Spectrum's screen layout, with its thirds and its interleaved rows, takes a Z80 a dozen instructions to turn a dot's position into an address. **PIXELAD** does it in one: E is the dot's column (x, 0 to 255), D its row (y, 0 to 191), and here is where their bits go (Figure 2.11).

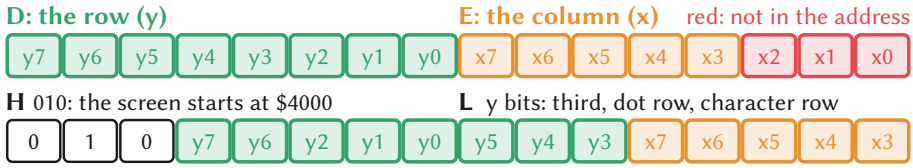


Figure 2.11: How PIXELAD builds the screen address

SETAE gives the dot's bit: $80 for column 0, $40 for column 1, down to $01 for column 7, from E's low three bits. **PIXELDN** moves HL one dot row down, across the character rows and the thirds, without you counting them. Together they draw a vertical line:

```
; Draws a vertical line of B dots down from column E, row D
VLine:  pixelad          ; HL = address of the dot (E, D)
        setae            ; A = the dot's bit in its byte
        ld c,a           ; keep it in C
Dot:    ld a,c
        or (hl)          ; set the dot
        ld (hl),a
        pixeldn          ; HL = same column, one row down
        djnz Dot
        ret
```

How it works:

- PIXELAD and SETAE turn the starting position into an address and a bit, once.
- Every dot is the classic read, set, write of a byte.
- PIXELDN replaces the twelve or so instructions a Z80 needs to step down a row, which is where the time went in the Spectrum's line-drawing routines.

A small warning: below row 191, `PIXELDN` goes on past the screen, into the attributes. Count your rows.

Block copies and an output

Instruction	Opcode	T-states	What it does
`LDIX`	`ED A4`	16	`LDI`, but skips the write if the byte equals `A`
`LDIRX`	`ED B4`	21 / 16	repeats `LDIX` while `BC` is not 0
`LDDX`	`ED AC`	16	as `LDIX`, with `HL` going down and `DE` up
`LDDRX`	`ED BC`	21 / 16	repeats `LDDX` while `BC` is not 0
`LDPIRX`	`ED B7`	21 / 16	repeats a copy from an 8-byte pattern
`LDWS`	`ED A5`	14	copies `(HL)` to `(DE)`, then `L` + 1 and `D` + 1
`OUTINB`	`ED 90`	16	sends `(HL)` to port `BC`, then `HL` + 1

The repeating forms take 21 T-states for a byte after which they repeat and 16 for the last, like `LDIR`.

The **`X`** in the names means **exclude**: the byte equal to `A` is not written. That is a transparent colour, in hardware. Set `A` to the colour your sprite uses for "nothing here", and `LDIRX` copies the sprite over a background without wiping it. `HL`, `DE` and `BC` move on for a skipped byte as for a written one.

Look closely at `LDDX`: `HL` goes **down** but `DE` goes **up**. It copies a row of bytes in reverse order, which mirrors a row of a 256-colour picture horizontally.

`LDPIRX` fills memory from an 8-byte pattern. It reads its byte from `HL` with the low three bits replaced by those of `E`, writes it to `DE` unless it equals `A`, and moves `DE` on, leaving `HL` alone. As `DE` walks forward, the pattern repeats every 8 bytes, lined up with the destination: a dithered fill, a striped background, a texture.

▶ TRY IT Pattern Fill

Part I, key 3. The demo paints six rows of colour on the ULA's attributes twice, then lays an 8-byte pattern over the second copy with one `LDPIRX`: two black cells, two holes, two bright white cells, two holes. The pattern repeats every 8 cells, and the colours show through the holes. The code is in `examples/code/demos/1-machine/pattern-fill.kz80.asm`:

```
    ld hl,PfPattern          ; the pattern; HL's low three bits do not
count
    ld de,PfAfter           ; the cells to paint
    ld bc,PfRows * 32
```

▶ TRY IT Pattern Fill *(continued)*

```
    ld a,PfHole             ; never written: the holes in the pattern
    ldpirx

    .align 8                ; the pattern's address ends in three 0 bits
PfPattern:
    .defb $00, $00, PfHole, PfHole, $78, $78, PfHole, PfHole
```

The `.align 8` matters. `LDPIRX` takes the pattern's byte from `HL` with its low three bits replaced, so a pattern that does not start on an 8-byte boundary is read from the wrong place. `DE` needs no alignment: start the fill at a cell whose address ends in 5, and the first byte written is the pattern's byte 5, the one the pattern puts there.

`LDWS` copies a byte and moves to the next column of the source and the next row of the destination, as `INC L` and `INC D` would; the flags are those of `INC D`. It was made for the Layer 2 screen, where `D` is a row and `E` a column. **`OUTINB`** is `OUTI` without touching `B`, so `B` keeps the port's high byte and `HL` streams a table to it.

The flags after the block copies follow `LDI`'s rules, more or less, and I would not rely on them: test `BC` yourself.

NextRegs

Instruction	Opcode	T-states	What it does
`NEXTREG r,n`	`ED 91 r n`	20	writes `n` to NextReg `r`
`NEXTREG r,A`	`ED 92 r`	17	writes `A` to NextReg `r`

These write a NextReg without the two ports. The usual way, through port `$243B` to select a register and port `$253B` to write it, takes two `OUT`s and a register to hold the port. More important to me than the speed: **`NEXTREG` leaves port `$243B`'s selection alone**. An interrupt handler that writes NextRegs with `NEXTREG` can never confuse a main program that selected a register and is about to write it.



KLIVE IDE NextRegs while the program runs

The **Next Registers** view, `show-nextregs`, shows the NextRegs as the program leaves them. A **NextReg write** breakpoint, `bp-set nr:$07` for the speed register, stops the machine when that register is written, whether through port `$253B`, by `NEXTREG` or by the DMA.

One instruction off the list, and the empty cells

`ED B6` is in no list of the Z80N's instructions, yet it behaves exactly like `LDIRX`: it is what is left of an instruction that was never finished. Do not use it, because a later core may change it.

Every other empty cell of the `ED` page is an 8-T-state `NOP` on the Next, but emulators have long borrowed these cells for private instructions. Keep those out of anything that runs on real hardware, where the CPU executes their operand bytes as code.

In short. Every Z80 instruction takes Zilog's T-states at all four speeds on core 3.03, the current core, on every board. The older core 3.02 adds one wait state to every memory read at 28 MHz, except from page `$0E`, so when a loop's length matters, read the core version from NextRegs `$01` and `$0E` and count at the real speed. The 29 Z80N instructions live in the `ED` page, where a Zilog chip runs them as `NOP`s; `MUL D,E` takes 8 T-states, and `NEXTREG` leaves port `$243B`'s selection alone. `ADD rr,A` always clears the carry, even when the sum wraps. Never depend on Y, X or WZ, because the Next departs from Zilog there.

CHAPTER 3

Memory, Ports and NextRegs: How the Z80N Reaches the Machine

A Z80 can name 65,536 bytes of memory, and the Next carries **2 MB** of RAM (the Issue 5 board 4 MB, but its upper half is a story of its own). On top of it sit some thirty devices behind I/O ports, and a file of 256 registers that decides how everything else behaves. The CPU still has sixteen address lines and not one more. So how does a 64K processor reach thirty-two times its own address space, and why does a game written for the 128K in 1986 never notice?

Every Next program of any size lives by paging, and every paging trick is a write to a NextReg or a port. Every surprise ("I wrote it there, why does it read back as the ROM?") is one of the rules you are about to meet. Know them, and paging becomes bookkeeping. Dull, reliable, wonderful bookkeeping.

It helps to know the Spectrum's own port `$FE`, which sets the border colour, drives the beeper and the tape, and reads the keyboard.

- How the CPU's 64K maps onto 2 MB through eight slots of 8K
- Who competes for the bottom 16K, and who wins
- How the 128K's paging ports, the ROMs and contention work on the Next
- How ports decode their addresses, and the two ways into the NextRegs

Three doors into one machine

A Z80 talks to the world in only two ways. A **memory cycle** (the MREQ line low) reads or writes a byte at a 16-bit address. An **I/O cycle** (the IORQ line low) reads or writes a byte at a 16-bit port address. The Next adds a third door of its own, the **NextRegs**, a file of registers that configure the machine. On the memory door stands the **MMU**, the **memory management unit**, which decides where in the Next's memory each address the CPU names really lands. Figure 3.1 shows the three doors.

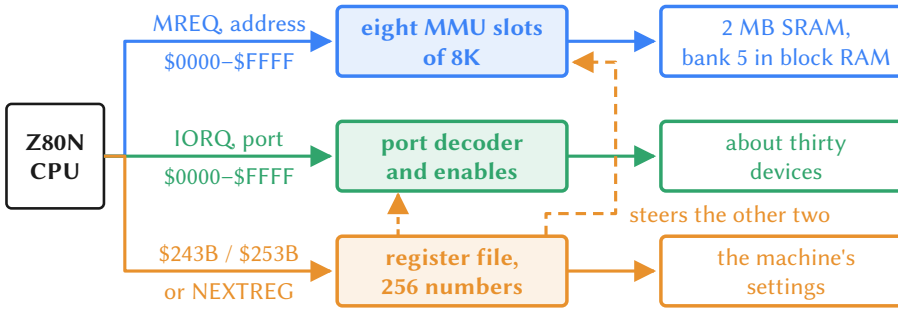


Figure 3.1: The CPU's three doors

Strictly speaking, the third door is a room behind the second one. You reach the NextRegs through two I/O ports, `$243B` and `$253B`, or through the Z80N's own `NEXTREG` instruction. But I give them a door of their own, because they are where the machine keeps its **settings**: which page sits in which slot, which ports are switched on, how fast the CPU runs, which layers the screen draws. The dashed arrows in Figure 3.1 are the point. Memory and I/O do the work; the NextRegs decide how the work is routed. I like to think of them as the machine's control room: nothing happens there, and yet everything is decided there.

Eight slots of 8K

The CPU's 64K is cut into **eight slots of 8K**. Bits 15–13 of an address pick the slot: `$0000–$1FFF` is slot 0, `$2000–$3FFF` slot 1, and so on up to `$E000–$FFFF`, slot 7. Each slot holds a **page**, an 8K piece of the Next's memory, and one register per slot says which: the **MMU slot 0 register**, **NextReg `$50`**, for slot 0, through the **MMU slot 7 register**, **NextReg `$57`**, for slot 7. Figure 3.2 shows the eight slots after a reset.

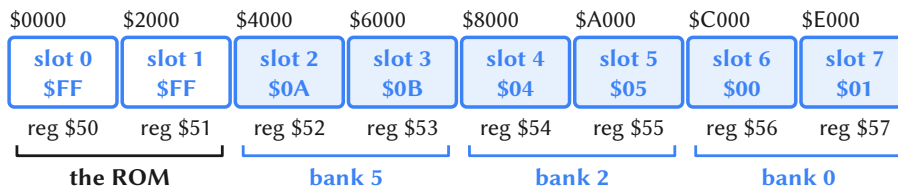


Figure 3.2: The eight slots after a reset

A page number is 8 bits wide. Pages `$00–$DF` are RAM, 224 pages of 8K, which makes 1,792K. The Spectrum world counts in **16K banks**, though, and the Next keeps that

habit: bank *n* is simply the two pages **2*n*** and **2*n* + 1**. Bank 5, the 48K screen, is pages `$0A` and `$0B`; bank 0 is pages `$00` and `$01`. Write `$0A` to NextReg `$52` and `$0B` to NextReg `$53`, and bank 5 sits at `$4000`, exactly where a 48K program expects its screen. I always do the doubling in my head the same way: bank 5, "ten and eleven".

After a reset the slots hold `$FF`, `$FF`, `$0A`, `$0B`, `$04`, `$05`, `$00`, `$01`: bank 5 at `$4000`, bank 2 at `$8000` and bank 0 at `$C000`, the layout of a 128K just switched on. And what is `$FF`? In slot 0 or 1 it means "**the ROM here**". In fact any page of `$E0` or more does in those two slots, because anything from `$E0` up points past the end of the RAM, and the machine falls back on the ROM. In slots 2–7 there is no ROM to fall back on: a write through such a page goes nowhere, and what a read returns is not defined. My advice: use `$FF` for the ROM and leave the other thirty-one values to the philosophers.

Moving a page costs one instruction. Here is a program that puts page `$20` at `$E000`, writes a byte, and puts bank 0 back:

```
nextreg $57,$20     ; slot 7 ($E000-$FFFF): page $20
ld a,$5A
ld ($E000),a        ; lands in page $20, offset 0
nextreg $57,$01     ; slot 7: page $01 (bank 0) again
```

Two things are worth seeing here. First, a page can sit in **several slots at once**: put page `$20` in slots 3 and 7, and a byte written at `$6000` reads back at `$E000`. Second, nothing stops you from paging out the code you are running.

Because the eight NextRegs read back what they hold, borrowing a slot politely is easy: read the slot's register, page in what you need, and write the old value back when you are done.

▶ TRY IT Bank Window

Part I, key 4. The demo borrows slot 7 and looks through it at four pages, one after another. The code is in `examples/code/demos/1-machine/bank-window.kz80.asm`, and its frame is the polite borrowing just described:

```
; Borrow slot 7 politely: remember which page it holds
ReadNextReg($57)
ld (BwOldPage),a
```

```
; Give slot 7 back
ld a,(BwOldPage)
nextreg $57,a
```

▶ TRY IT Bank Window *(continued)*

In between, `nextreg $57,a` pages in `$28`, `$29` and `$2A`, and the demo prints the text it finds at `$E000`. The NEX file loaded those pages: `bank-window-data.kz80.asm` puts a line of text at the start of bank 20, halfway into it, and at the start of bank 21. Last comes page `$0A`, the first half of bank 5, which slot 2 also holds. A bar written at `$E880` appears on the screen, because `$4880` is the same byte seen through the other slot. To watch it happen in Klive, set a breakpoint on a `nextreg $57,a` line and step over it: the **Next Memory Mapping** panel shows slot 7's page change.



KLIVE IDE Bank-relative breakpoints

`bp-set 05:+$0100` stops at offset `$0100` of 16K bank 5, in whichever slot the bank is paged at the time. Add `-w`, and it stops when that byte of the bank is written instead. The bank is a 16K bank, as in a NEX file and the `.bank` pragma, so on the Next it covers two 8K pages.

From a CPU address to the SRAM

The Next's main memory is an external **SRAM** of 2 MB, and its addresses are 21 bits wide. Turning a CPU address into one takes no arithmetic worth the name: the slot's page, plus `$20`, becomes the top 8 bits, and the CPU address's low 13 bits come along unchanged. Figure 3.3 follows one address through.

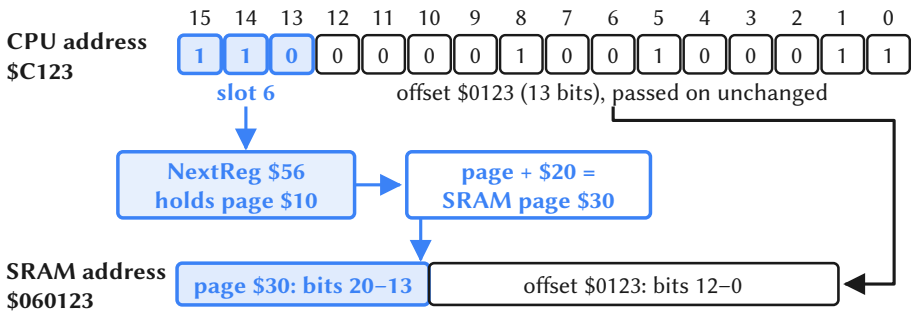


Figure 3.3: From a CPU address to an SRAM address

Let's take `$C123` step by step:

1. Bits 15–13 are `110`: slot 6.
2. The **MMU slot 6 register**, **NextReg $56**, holds page `$10`.
3. Page `$10` plus `$20` is SRAM page `$30`.

4. The low 13 bits are `$0123`, so the SRAM address is `$30` × 8K + `$0123`, which is `$060123`.

Why the `$20`? Because RAM page 0 does not start at the bottom of the SRAM. The first **256K**, 32 pages of 8K, belong to the machine itself, and no page number of the MMU can reach them (Figure 3.4).

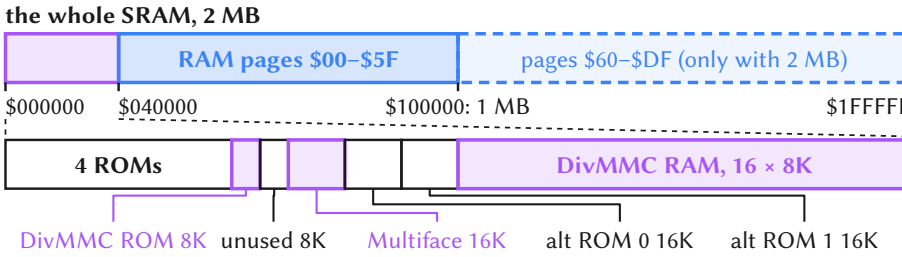


Figure 3.4: The SRAM, to scale

- **`$000000–$00FFFF`**: the Next's own four 16K ROMs, NextZXOS's ROM set, copied there from the SD card when the machine starts. They keep the +3's numbering, ROM 0 to ROM 3.
- **`$010000–$011FFF`**: the DivMMC's 8K ROM, and an 8K gap after it.
- **`$014000–$017FFF`**: the Multiface's 8K ROM and 8K RAM.
- **`$018000–$01FFFF`**: two **alternate ROMs** of 16K each, one for 128K mode and one for 48K, filled from the same ROM set.
- **`$020000–$03FFFF`**: the DivMMC's 128K of RAM, sixteen banks of 8K.
- **`$040000` up**: RAM page `$00`, then every page in order, up to page `$DF` at `$1FE000`.

I call this first 256K **the basement**: the machine keeps its own things down there, and your page numbers have no stairs leading to it. And look at what it holds. The ROMs are not chips on the board; they are ordinary RAM that the CPU is not allowed to write through the MMU. One memory chip plays a dozen parts, and the decoding logic decides who gets to see which.

There is one exception to "everything is SRAM", and it matters. **Bank 5**, pages `$0A` and `$0B`, and the first half of **bank 7**, page `$0E`, live in **block RAM** inside the FPGA, not in the SRAM. That block RAM has two ports, so the ULA and the tilemap can read the screen while the CPU writes it (the screen series' Big Picture tells why). The SRAM still has room for those three pages, and that has a curious consequence: the CPU's writes through the MMU go to the block RAM only, while **Layer 2 is always displayed from the SRAM**.



KLIVE IDE One page at a time

Turn off the Memory view's 64K view and pick a page in its **Selected bank** chooser: it shows any 8K page, `00` to `DF`, paged in or not, and you can edit it. A 16K bank *B* is the two pages `2B` and `2B+1`, so bank 5 is pages `0A` and `0B`.

The Issue 5 board's attic

The Issue 5 board, the third Kickstarter's, has twice the SRAM: **4 MB**, with 22-bit addresses. The upper 2 MB, `$200000–$3FFFFF`, sits above everything shown so far, and I like to call it **the attic**: plenty of room, but no stairs from the rooms your program lives in (Figure 3.5).

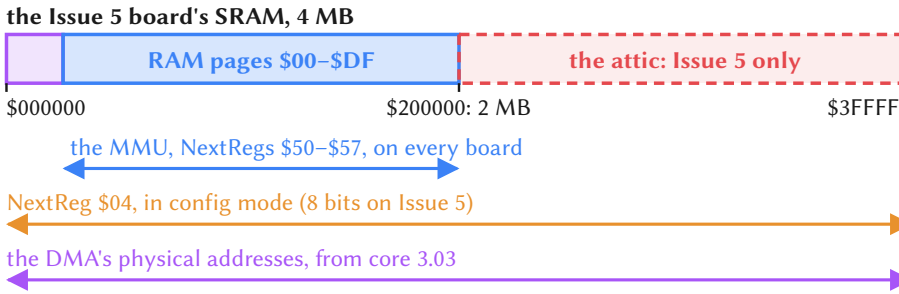


Figure 3.5: The Issue 5 board's SRAM, to scale, and the three ways into it

Your page numbers do not reach it, and the reason is the `$20` the MMU adds. The sum has nine bits. For pages `$00–$DF` the ninth bit is 0, and the other eight are the SRAM page. From page `$E0` up the sum reaches `$100`, and a ninth bit of 1 tells the machine that this is no SRAM address at all: slots 0 and 1 show the ROM instead, and slots 2 to 7 show nothing (Figure 3.6). On the Issue 5 board that ninth bit would have been address bit 21, the attic's, but the MMU spends it on "not RAM". So the MMU sees the same 2 MB on every board, and Layer 2, which finds its pages the same way, does too.

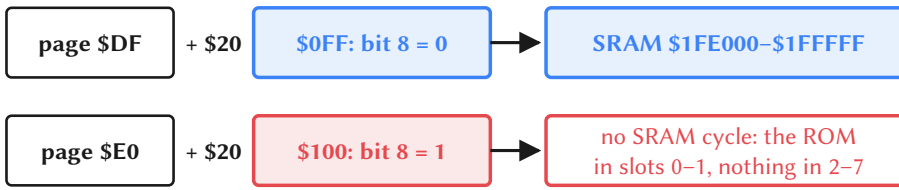


Figure 3.6: Why page `$E0` is the edge of the RAM, on every board

Two ways lead up there:

- **The Config Mapping register, NextReg `$04`**, maps a 16K bank of the whole SRAM at `$0000–$3FFF`, counting from the bottom of the basement: bank *b* is SRAM address *b* × `$4000`. On the Issue 5 board it keeps all 8 bits, so its banks `$80–$FF` are the attic; on the other boards bit 7 is ignored. It works only in **config mode**, the firmware's ground, which Chapter 1 describes.
- **The DMA's physical addresses**, a blitter feature of the current core, reach any byte of the 4 MB without the MMU, as "The zxnDDMA: the blitter" in Chapter 5 (page 78) explains. This is the way I recommend.

Config mode is not entirely out of a program's reach, though. Writing `111` to bits 2–0 of the **Machine Type register, NextReg `$03`**, switches it on from any program, and writing `101` or `110` switches it off again without changing the machine type. In between, NextReg `$04` answers in the bottom 16K, 8K half by 8K half, wherever the slot holds a page of `$E0` or more (the ROM's `$FF` will do) and the DivMMC, the Multiface and the Layer 2 mapping keep out of the way:

```
; run this from $4000 or above, on an Issue 5 board only
di
nextreg $50,$ff
nextreg $51,$ff
nextreg $03,%00000111   ; config mode on
nextreg $04,$80         ; $200000-$203FFF at $0000-$3FFF
; ... read or write the attic here ...
nextreg $03,%00000101   ; config mode off, machine type kept
; put MMU slots 0 and 1 back
ei
```

It is a narrow door. Interrupts must stay off, because the ROM's handler at `$0038` is gone while the attic is there, and the NMI is held off for as long as config mode lasts. A reset in config mode brings the boot ROM back, so keep the window short. And check the **Board ID register, NextReg `$0F`**, first: on an Issue 4 board bank `$80` is bank 0, and the first write lands in the ROMs. The attic is storage, not living space: nothing can run there or be shown from there, so a program copies to it and from it.

The DMA's physical addresses have the same trap. On an Issue 4 board they wrap round onto the basement, ROMs and all, so a program that wants the attic reads NextReg `$0F` first: 3 means the Issue 5 board.



KLIVE IDE The attic in the Memory view

On the KS3 model, the Memory view and the bank choosers list the upper 2 MB as **Upper RAM**, pages `U00` to `UFF`. You can watch what a program puts in the attic, or edit it, although no slot of the MMU can show it.

So treat the attic as a bonus, not as part of the house. Build your program to live in the 2 MB that every Next has. If NextReg `$0F` reports an Issue 5 board, use the attic for data you can do without, such as a cache, extra levels or an undo history, and move that data with the DMA. The program then runs on every board and makes good use of the larger one.

Who answers an address

The MMU is not alone. Four other pieces of hardware can claim an address before it, and I find it easier to remember them as characters than as signals:

- **The stagehand** is the **boot ROM**, which sets the scene while the machine starts.
- **The fire brigade** is the **Multiface**, a debugging cartridge built into the machine: press its button, and it takes over the ground floor.
- **The butler** is the **DivMMC**, the SD card interface that NextZXOS and esxDOS run on: there when you ring, gone when the job is done.
- **The painter** is the **Layer 2 mapping**, which puts up scaffolding over the bottom of the address space so the CPU can paint a Layer 2 picture through it.

Each one, when it is active, takes over part of the CPU's 64K, so the Next keeps a strict order of precedence (Figure 3.7).

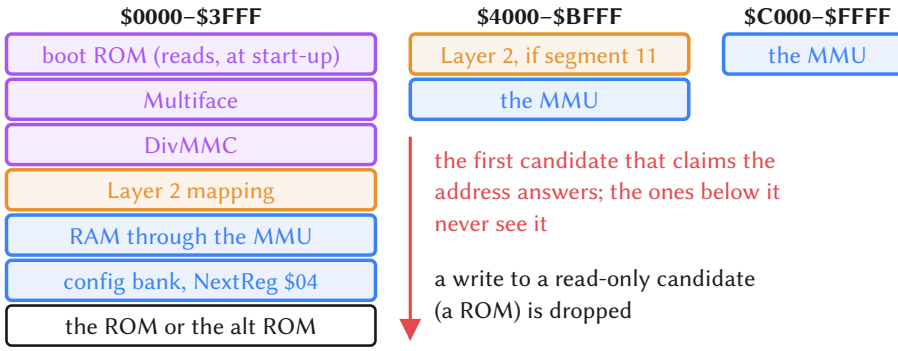


Figure 3.7: Who answers an address, highest first

Reading it from the top:

1. **The boot ROM** answers reads of `$0000–$3FFF` while the machine starts: 8K of ROM inside the FPGA, seen twice. It switches itself off the first time anyone writes the **Machine Type register, NextReg `$03`**. Exit the stagehand.
2. **The Multiface** takes the whole bottom 16K when its button is pressed: its read-only ROM at `$0000`, its RAM at `$2000`. When the fire brigade arrives, even the butler steps aside.

3. **The DivMMC** takes `$0000–$3FFF` when its ROM is paged in, by a program (port `$E3` bit 7) or automatically when the CPU fetches an instruction from one of its entry points. Its read-only ROM is at `$0000`, and one of its sixteen 8K RAM banks (port `$E3` bits 3–0) at `$2000`.
4. **The Layer 2 mapping**, switched on through port `$123B`, lays the picture over the bottom of the address space, with reads (bit 2) and writes (bit 0) switched on separately. Bits 7–6 choose the **segment**: `00`, `01` and `10` put the picture's first, second or third 16K at `$0000–$3FFF`, and `11` puts all 48K at `$0000–$BFFF`. The picture starts at the bank in NextReg `$12` (bank 8 after a reset), or in NextReg `$13` (bank 11) when bit 3 asks for the shadow picture.
5. **The MMU** answers with RAM when its page is below `$E0`.
6. **The config bank**, a 16K bank of the SRAM chosen by the **Config Mapping register, NextReg `$04`**, answers while the machine is in its start-up configuration mode. That is how the start-up code writes the ROM images into the first 64K of the SRAM: reads come from the boot ROM, writes go to the config bank underneath. The stagehand reads from one script and writes the next act on the floor beneath its feet. It answers only where the slot's MMU page is `$E0` or more, and on the Issue 5 board it can reach the attic (The Issue 5 board's attic above).
7. **The ROM**, or the alternate ROM, answers everything else.

Above `$C000` none of this exists. The top 16K is the MMU's alone, so I keep my stack and my interrupt routine there: whatever pages in at the bottom, they stay where they are.

One rule catches almost everyone out. A higher candidate wins **even over RAM the MMU pages into slots 0 and 1**. Put page `$20` in slot 0, and as soon as the butler pages himself in (NextZXOS rings for him all the time), your page is hidden until he leaves. And a write that reaches the ROM goes nowhere, quietly.

The DivMMC's entry points

The butler is the only character who comes in without being called by name. He watches the CPU's **instruction fetches**. When the CPU fetches an opcode from one of his **entry points**, the DivMMC **automaps**: its 8K ROM appears at `$0000`, one of its RAM banks at `$2000`, and its own code runs instead of the ROM's. A program calls `RST $08`, the CPU fetches the opcode at `$0008`, and the SD card's operating system is at its service.

On the Next, four registers choose the entry points:

- The **Divmmc Entry Points 0 register, NextReg `$B8`**, switches on the eight `RST` addresses, `$0000` to `$0038`, one bit each. After a reset it holds `$83`: `$0000`, `$0008` and `$0038`.

- The **Divmmc Entry Points Valid 0 register**, **NextReg $B9**, says, per `RST`, whether it maps **always** or **only when ROM 3 is present**. After a reset only `$0000` maps always.
- The **Divmmc Entry Points Timing 0 register**, **NextReg $BA**, says, per `RST`, whether the mapping is **instant** or **delayed** (all delayed after a reset).
- The **Divmmc Entry Points 1 register**, **NextReg $BB**, adds the rest: `$0066` (the NMI, after the DRIVE button), four addresses in the 48K ROM's tape routines (`$04C6`, `$0562`, `$04D7` and `$056A`), which let the DivMMC load a tape image from the SD card, and `$3D00`–`$3DFF`, where the Beta Disk interface's TR-DOS was entered.

With **instant** mapping, the opcode at the entry point already comes from the DivMMC. With **delayed** mapping, it still comes from the ROM and runs, and the DivMMC takes over from the next fetch. The tape addresses need that: the ROM's own instruction must run first. A good butler lets you finish your sentence before he takes the coat.

"**ROM 3 is present**" means that the opcode at the entry point is being read from the 48K BASIC ROM, not from RAM, another ROM or anything laid over the ROM. Anywhere else, `$04C6` is just some instruction in the middle of something else. All of these must hold at that fetch:

1. **The slot shows a ROM.** Its MMU register holds `$FF` (or any page from `$E0` up), the machine is not in its start-up config mode, and no Layer 2 read mapping covers the bottom 16K.
2. **That ROM is the machine type's 48K BASIC**: ROM 3 on a +3 or Next machine type, ROM 1 on a 128K, and the only ROM there is on a 48K.
3. **If the alternate ROM is on for reads** (the Alternate ROM register, NextReg `$8C`, with bit 7 set and bit 6 clear), it must be the 48K alternate ROM, not the 128K one.

Once mapped, the DivMMC stays until its code leaves, on a `RETN` or by fetching an instruction from `$1FF8`–`$1FFF` (with NextReg `$BB` bit 6 set). Bit 4 of the **Peripheral 5 Setting register**, **NextReg $0A**, is the master switch: clear it, and no entry point maps and any mapping in force is dropped. A program can also page the DivMMC in by hand with port `$E3` bit 7, until it clears the bit.



KLIVE IDE The DivMMC in the side bar

The **Next Memory Mapping** panel shows the DivMMC's control register and whether its memory is paged in, so you can see an entry point fire. The Memory view names the DivMMC's ROM `DM` and its sixteen 8K RAM banks `M0` to `MF`.

The 128K's ports, rewritten

A 128K program pages memory through port `$7FFD`: bits 2–0 choose the bank at `$C000`, bit 3 the screen, bit 4 the ROM, and bit 5 locks the port until the next reset. A +3 program adds port `$1FFD`, and the Next adds port `$DFFD` for bank numbers above 7. On the Next, **a paging port is a way of writing the MMU**. I call `$7FFD` **the interpreter**: everything it hears, it translates into MMU. The machine works out the 128K layout and loads it into the eight slots, and from then on only the MMU decides. Figure 3.8 shows what a write to port `$7FFD` does.

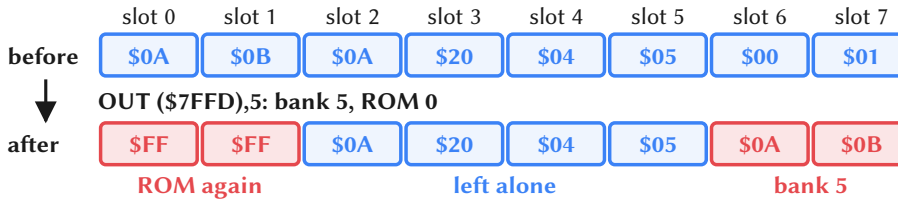


Figure 3.8: What a write to port $7FFD does

Here is what a write to `$7FFD`, `$DFFD` or `$1FFD` does, outside the +3's all-RAM layouts:

- **Slots 0 and 1** get `$FF`, the ROM, whatever you put there.
- **Slots 6 and 7** get the selected bank's two pages: bank 5 is `$0A` and `$0B`.
- **Slots 2–5 are left alone.** They are set to `$0A`, `$0B`, `$04`, `$05` only when a program leaves one of the +3's all-RAM layouts.

So a 128K program and a Next program can share a machine. My rule of thumb: if you mix the two, write the MMU after the port, never before, because the port overwrites slots 0, 1, 6 and 7.

The +3's **all-RAM layouts** (port `$1FFD` bit 0) load all eight slots at once, with RAM even where the ROM used to be, in one of four layouts chosen by `$1FFD` bits 2–1 (Figure 3.9).

bits 2–1	$0000	$4000	$8000	$C000
00	bank 0: $00, $01	bank 1: $02, $03	bank 2: $04, $05	bank 3: $06, $07
01	bank 4: $08, $09	bank 5: $0A, $0B	bank 6: $0C, $0D	bank 7: $0E, $0F
10	bank 4: $08, $09	bank 5: $0A, $0B	bank 6: $0C, $0D	bank 3: $06, $07
11	bank 4: $08, $09	bank 7: $0E, $0F	bank 6: $0C, $0D	bank 3: $06, $07

Figure 3.9: The all-RAM layouts, by port $1FFD bits 2–1 (filled: the two screens)

Three of the four layouts keep bank 3 in the top 16K, which makes it the spot I would choose for a stack and for the code that switches between layouts. Leave all-RAM mode, and slots 2–5 go back to banks 5 and 2.

Outside these layouts, `$1FFD` bit 2 and `$7FFD` bit 4 together choose one of the four 16K ROMs. Bit 5 of `$7FFD` blocks further writes to `$7FFD`, `$1FFD` and `$DFFD`, as on a 128K, but not to the MMU, and it can be undone without a reset: writing 1 to bit 7 of the **Peripheral 3 Setting register**, **NextReg `$08`**, clears the lock, and the bit reads back 1 while `$7FFD` is not locked.

`$7FFD` cannot be read, so the 128K's ROM kept a copy of every value it wrote there in a system variable, `BANKM` at `$5B5C`. On the Next, the **Spectrum 128K Memory Mapping register**, **NextReg `$8E`**, is **the interpreter's notebook**: it reads back the bank, the ROM and the all-RAM bits, and bit 6 of the **Display Control 1 register**, **NextReg `$69`**, reads and writes the screen bit. Save them, do your work, write them back.

A write to `$8E` sets the bank, the ROM and the all-RAM layout at once. With its bit 3 set, the bank in bits 6–4 (and bit 7) is stored and slots 6 and 7 follow it; with bit 3 clear, only the ROM changes, so you can swap the ROM without knowing which bank is at `$C000`.

The **Memory Mapping Mode register**, **NextReg `$8F`**, lets the Next page like the Pentagon, a family of Russian Spectrum clones. With `$8F` = 2 (**Pentagon 512K**), the bank comes from `$7FFD` bits 7–6 and 2–0, and `$DFFD` is ignored. With `$8F` = 3 (**Pentagon 1024K**), bit 5 of `$7FFD` stops being the lock and becomes a sixth bank bit, and bit 2 of port `$EFF7` can turn this mode back into standard paging. On a 1 MB Next, Pentagon 1024K reaches only 768K. Native Next code leaves `$8F` at 0.



KLIVE IDE The 128K's ports in the side bar

The **Next Memory Mapping** panel shows the last values written to ports `$7FFD`, `$1FFD`, `$DFFD` and `$EFF7`, the all-RAM layout, the current ROM and bank, and the page in each of the eight slots. Step over an `OUT` to `$7FFD` and watch slots 6 and 7 follow it.

The ROMs, and how to replace one

Slots 0 and 1 with `$FF` show **one 16K ROM**: slot 0 its first 8K, slot 1 its second. Which ROM it is depends on the machine type and the paging ports, just as on the +3: ROM 0 to ROM 3, chosen by `$1FFD` bit 2 and `$7FFD` bit 4. The ROMs themselves are not the +3's but the Next's own, from NextZXOS, and they keep the +3's numbering: **ROM 3 is the 48K BASIC**, with the 48K ROM's routines at their old addresses, the one a 48K program expects to find, and the one the butler's tape entry points look for. The ROMs are in RAM, but the decoder marks them **read-only**, so a write to `$0000` with the ROM paged in does nothing, as on every Spectrum.

The **Alternate ROM register**, **NextReg `$8C`**, can swap in one of the two alternate ROMs, the 128K one or the 48K one. Its bit 7 switches it on, and its bit 6 chooses which direction it works in:

- With bit 6 clear, **reads come from the alternate ROM**, and it is read-only like any ROM.
- With bit 6 set, **writes go to the alternate ROM** while reads still come from the normal one.

That second mode is the one I find delightful: it is how you load a ROM of your own. Set bit 6, copy 16K to `$0000`, clear bit 6, and the machine runs your ROM from then on. You are painting the new stage set behind the curtain while the old play is still running in front of it. Bits 5 and 4 lock the ROM selection to one ROM, whatever the paging ports say, and a soft reset copies bits 3–0 into bits 7–4, so the setting can survive a reset if you prepare it in advance.

Contention

The 48K and the 128K have **contended memory**: while the ULA reads the screen, the CPU must wait for some banks. The Next's ULA has its own memory and needs no such wait, but the Next reproduces the delay, cycle for cycle, so timing-critical code (a border effect, a tape loader) runs as it did. I call it **the nostalgia tax**.

Which pages are contended depends on the timing:

- **48K timing**: bank 5, pages `$0A` and `$0B`.
- **128K timing**: the odd banks, 1, 3, 5 and 7.
- **+3 timing**: banks 4–7.
- **Pentagon timing**: none.

Only pages `$00`–`$0F` can ever be contended. All of it happens only at **3.5 MHz**, and bit 6 of the **Peripheral 3 Setting register**, **NextReg `$08`**, switches it off there too. The ULA takes a copy of that bit only from dot 256 of every line, so a write early in a line changes nothing until the beam gets there, and reading NextReg `$08` back meanwhile returns the old bit.

At 28 MHz there is no delay at all on core 3.03, the current core. The older core 3.02 has a delay of a different kind there: the SRAM is not quite fast enough for it, so every read from the SRAM or from bank 5 adds one wait state. Writes and reads from page `$0E`, the block RAM half of bank 7, do not; page `$0F`, the other half, is SRAM and waits (Chapter 2).

I/O cycles pay the tax as well, in 48K and 128K timing: every even port, and port `$7FFD` in 128K timing. In +3 timing only memory is contended. For new code, run above 3.5 MHz or set bit 6 and forget the tax exists.

Ports: which bits count

A port address is 16 bits wide, but almost no Spectrum device ever looked at all of them. The 48K's ULA answered **every even address**, because it decoded only bit 0: `OUT ($FE),A` is the same port as `OUT ($7EFE),A`. I think of the ULA as the greediest tenant in the building: it holds the keys to half of all port addresses and uses one of them. The Next decodes each old port as loosely as the original did, and each new one strictly (Figure 3.10).

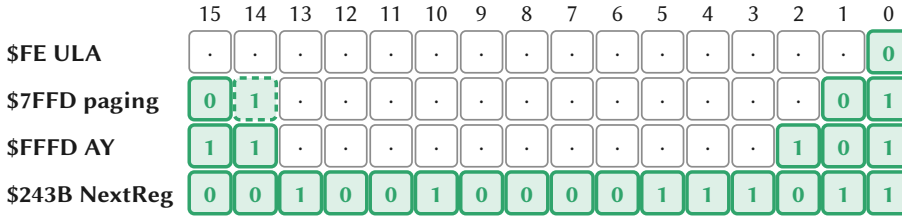


Figure 3.10: The address bits four ports decode (·: ignored; dashed: +3 timing only)

- **Port `$FE`**, the ULA: bit 0 = 0. Every even port is the ULA.
- **Port `$7FFD`**: bit 15 = 0, bits 1–0 = `01`, and bit 14 = 1 only in +3 timing. In 48K and 128K timing, `$3FFD` pages memory too, as it did on a real 128K.
- **Ports `$FFFD` and `$BFFD`**, the AY: bits 15–14 = `11` or `10`, bit 2 = 1, bits 1–0 = `01`.
- **The Next's own ports**, `$243B`, `$253B`, `$123B`, `$303B` and the rest ending in `$3B`: all sixteen bits.

The bits a port does not decode are not wasted. `IN A,($FE)` puts A on the top half of the address bus, and `IN r,(C)` puts all of BC there. The ULA reads bits 15–8 to select the **keyboard rows**: each 0 bit picks one half-row of five keys, and the five bits you read back are 0 for a key held down:

```
    ld a,$FB            ; %11111011: the Q-W-E-R-T half-row
    in a,($FE)          ; port $FBFE: the row in the top byte
    bit 0,a             ; bit 0 is Q: zero while it is held down
```

Clear several bits at once, and the rows are ANDed: a 0 comes back if the key is held in any of the rows you chose. The address is not only the name of whom you are asking; it is half of the question.

Every Next port is odd: `$3B`, `$E3`, `$57`, `$5B`, `$6B`. The greedy tenant owns every even address, so a new device there would collide with it, and with every 48K program's border writes.

When two devices answer the same write, both receive it. A write to a Soundrive DAC at `$F1` could also be a `$7FFD` write, so while that DAC is on, `$xxF1` and `$xxF9`

writes do not page memory. A drum sample that pages out your code is a bug I would not wish on anyone.

Switching ports off

A Next carries more devices than any one program wants, and some of their ports collide with hardware an old program expects, or with a peripheral on the expansion bus. So every internal port can be switched off, through the **Internal Port Decoding Enables registers**, **NextRegs $82–$85**, 28 bits in all. Bit 1 of NextReg $82 is port $7FFD, bit 7 of NextReg $83 is Layer 2's port $123B, bit 0 of NextReg $84 the AY, and so on. They all power on as 1. I picture a row of light switches by the front door, one per room.

A port that is switched off is gone: writes to it are dropped, and a read returns what an absent device would. Three ports have no switch: the ULA's $FE and the NextReg ports $243B and $253B. Switch off $253B, and you could never write NextReg $82 again to switch it back on.

The **Expansion Bus Decoding Enables registers**, **NextRegs $86–$89**, mirror $82–$85 for the time the bus is on: a port is answered inside only when both its bits are set, and otherwise the I/O cycle goes out to the bus. Bit 7 of NextReg $85 chooses whether a soft reset switches every internal port back on, so a program that disabled one can ask for it to stay disabled.

What you read when nobody answers

When the CPU executes an IN, every device compares the port address with its own. The Next uses **wired-OR logic**: every device that matches puts its byte out, every device that does not puts out 0, and all the bytes are ORed into one. Picture a meeting where those with nothing to say say exactly nothing, so you hear only the one who does. Figure 3.11 shows the read path of an IN.

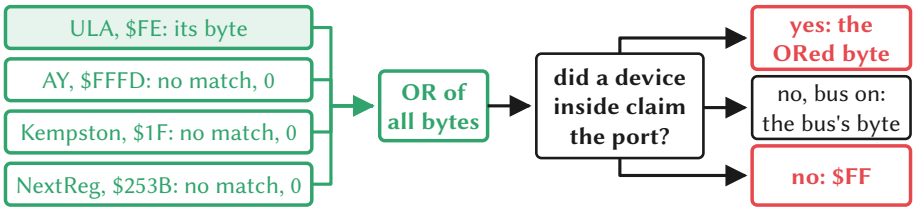


Figure 3.11: The read path of an IN

Normally exactly one device matches, and the OR is its byte. When two match, their bytes are ORed, the way two chips driving one data bus would combine on a real Spectrum. When none matches, the CPU reads **$FF**, the sound of silence on a Spectrum, or the expansion bus's byte when the bus is on and allowed to answer I/O.

There is one famous exception: **port $FF** in 48K and 128K timing returns the **floating bus**, the last byte the ULA fetched from the screen, because some 48K games synchronise with the beam by reading it. They are eavesdropping on the ULA, and the Next lets them. In +3 timing port $FF reads $FF, and the +3's floating bus appears instead at port $0FFD and the other addresses with bits 15–12 = 0000 and bits 1–0 = 01, only while $7FFD is not locked.

Some reads **do something**: reading port $303B returns the sprite collision flags and clears them, and reading a UART's data port takes a byte out of its receive buffer. Read such a port twice "just to check", and the second read sees something else. My advice: read once, and keep the value.

Two ways into the NextRegs

The NextRegs are numbered $00 to $FF, and there are two ways to write one. The first works on any Z80: write the register's number to **port $243B**, the **register select** port, and then the value to **port $253B**, the **data** port. The second is an instruction only the Z80N has, NEXTREG (Figure 3.12).

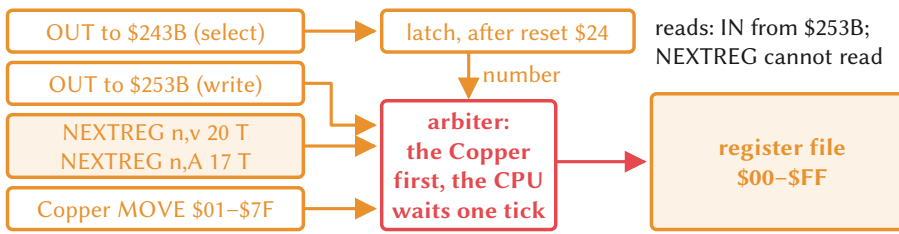


Figure 3.12: The ways into the register file

Compare the two on the same job, setting the CPU to 28 MHz through the **CPU Speed register**, **NextReg $07**:

```
; --- through the ports: 52 T-states
ld bc,$243B         ; 10 T
ld a,$07            ;  7 T
out (c),a           ; 12 T: select NextReg $07
inc b               ;  4 T: BC = $253B
ld a,$03            ;  7 T
```

```
    out (c),a           ; 12 T: write 3, 28 MHz

    ; --- with the Z80N instruction: 20 T-states
    nextreg $07,$03     ; NEXTREG n,v
```

`NEXTREG n,v` takes **20 T-states**, and `NEXTREG n,A`, with the value in A, **17**. Both go into the same write queue as a port `$253B` write. The important difference is another: `NEXTREG` carries its own register number, and **leaves the selection of port `$243B` alone**. An interrupt routine that uses the ports changes the selection under the feet of the code it interrupted. If it must use them, read port `$243B` first (it returns the selected number) and write it back before returning. My own preference is `NEXTREG` for every write, and the ports only where there is no choice.

Reading is such a place: there is no `NEXTREG` that reads. Select the number at port `$243B`, then `IN` from port `$253B`:

```
    ; --- which speed is the CPU really running at?
    ld bc,$243B
    ld a,$07
    out (c),a           ; select NextReg $07
    inc b
    in a,(c)            ; bits 5-4: actual, 1-0: requested
```

After a reset, `$243B` holds **`$24`**, the **Reserved register, NextReg `$24`**, which does nothing. An old program that writes `OUT ($3B),A` with A = `$25` writes to port `$253B` without ever meaning to. With `$24` selected, that stray write lands in **the decoy**, an empty room kept on purpose for visitors who came in by the wrong door.

The Copper goes first

The Copper, the Next's display co-processor (the screen series' raster effects chapter takes it apart), writes NextRegs too, with its MOVE instruction. Two rules govern it:

- **The Copper reaches only `$01`–`$7F`.** Its MOVE has seven bits for the register number, so the bottom half of the map is its whole world: video, palettes, the MMU, sound mirrors. A MOVE to register 0 is the Copper's NOP and writes nothing.
- **The Copper has priority.** When the Copper and the CPU write in the same 28 MHz tick, the Copper's write goes first and the CPU's waits one tick. Nothing is lost: the Copper writes at most every other tick, so the CPU always gets its turn.

So the Copper jumps the queue, but politely, and never twice in a row. I also like reading its reach the other way round: everything from `$80` up (the port enables, the expansion bus, the memory mapping registers, the interrupt controller) is safe from

a Copper list gone wrong. A runaway Copper can make your screen look dreadful, but it cannot switch off your ports.

A map of 256 numbers

Not all 256 numbers are used. The register list names 145 of them, in neighbourhoods (Figure 3.13).

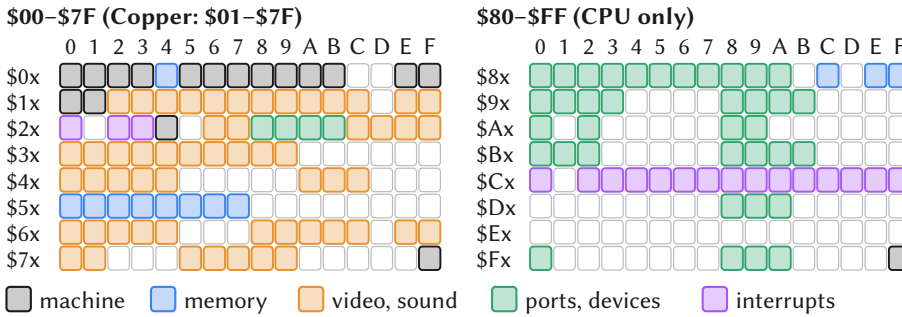


Figure 3.13: The 256 NextReg numbers (empty: unused)

- **`$00`–`$11`, the machine**: identity, resets, machine type and timing, CPU speed, peripheral settings.
- **`$12`–`$4C`, mostly video**: Layer 2, the layer order, scrolling, clip windows, sprites and palettes; plus the keymap, the line interrupt and the DAC mirrors.
- **`$50`–`$57`, the MMU.**
- **`$60`–`$7F`, video again**: the Copper, ULA, LoRes, tilemap and Layer 2 controls, and at `$7F` the **User Register 0**, which keeps a byte for you, across soft resets too.
- **`$80`–`$8F`, the bus and memory**: the expansion bus, port enables, alternate ROM, 128K mapping, Pentagon modes.
- **`$90`–`$BB`, peripherals**: the Pi's GPIO and audio, the ESP Wi-Fi GPIO, extended keys, DivMMC entry points.
- **`$C0`–`$CF`, interrupts.**
- **`$D8`–`$FF`, the rest**: I/O traps, the FPGA's analogue converter on some boards, and `$FF`, reserved.

Reading a NextReg does not always give back what you wrote. A number the register list does not use reads **0**, and so do write-only registers. Some registers read **state, not storage**: NextReg `$07` returns the speed you asked for in bits 1–0 and the speed the CPU is running at in bits 5–4. And some writes are **actions**: a write to the **Palette Value register**, **NextReg `$41`**, stores a colour and moves the palette index on by one, and a write to the **Reset register**, **NextReg `$02`**, resets the machine.



KLIVE IDE Every NextReg, named

View › Machine Views › Next Registers, or `show-nextregs`, lists each register's last written and current value. Select one to see its bit fields and their meanings.

Resets

The Next has two resets. A **hard reset** reloads the whole FPGA design, as if you had switched the machine off and on, and every register starts afresh: that is moving out of the house. A **soft reset** (the reset button, or bit 0 of the **Reset register**, **NextReg `$02`**) only tidies up the room: it puts the machine back into the state a program expects to find, but keeps the furniture. In detail, a soft reset:

- puts the MMU back to `$FF`, `$FF`, `$0A`, `$0B`, `$04`, `$05`, `$00`, `$01`, clears the paging ports and their lock, and selects the Reserved register, NextReg `$24`, at port `$243B`;
- resets most video, sound and interrupt registers to their power-on values;
- **keeps** the machine type and timing (the **Machine Type register**, **NextReg `$03`**), most of the peripheral settings, the User Register 0 at `$7F`, and the Pentagon mapping mode;
- switches the internal ports back on only if bit 7 of the **Internal Port Decoding Enables register**, **NextReg `$85`**, says so, and copies the prepared halves of the **Alternate ROM register**, **NextReg `$8C`**, and the **Expansion Bus Enable register**, **NextReg `$80`**, into their live halves.

That last rule is my favourite small piece of cleverness here: a program prepares what the machine should look like after the reset, and the reset makes it so, like laying out tomorrow's clothes before you go to bed. And here is a warning in the same spirit as TurboSound's: because a soft reset keeps the settings, a program must never assume that a register is at its power-on value just because a reset happened. Set what you depend on. It costs a few `NEXTREG`s, and I think that is the cheapest insurance on the machine.

In short. The CPU's 64K is eight 8K slots, each holding the page named by NextRegs `$50` to `$57`, and `$FF` in slots 0 and 1 means the ROM. The boot ROM, Multiface, DivMMC and Layer 2 mapping all beat the MMU, even over RAM paged into those two slots, and only the top 16K is safe from them, so keep stacks and handlers there. Port `$7FFD` overwrites slots 0, 1, 6 and 7, so write the MMU after it. A soft reset keeps many settings, so set every register you depend on. The Issue 5 board's upper 2 MB is out of the MMU's reach: only config-mode NextReg `$04` and, from core 3.03, the DMA's physical addresses get there, and only after NextReg `$0F` says 3.

CHAPTER 4

Interrupts: Fourteen Voices, Two Wires

A Next game is a busy place. The music player wants to run fifty times a second, a raster split wants its line, a sample needs its next byte from a timer, and somebody pressed the DRIVE button. Fourteen pieces of hardware would like the CPU's attention, often at the same moment, and the Z80 has exactly two wires to hear them with. So who decides who goes first, how does the CPU learn who called, and what happens to the ones that have to wait?

Polling works until it doesn't: the moment your code must keep a tune steady while it loads a level, you need the hardware to tap the CPU on the shoulder. The Next's answer is far richer than the Spectrum's single frame interrupt, and it has a few traps that look like bugs until you know the rule behind them.

This chapter builds on Chapter 3, for NextRegs, the DivMMC and the Multiface, and on Chapter 10, for where in the frame the frame and line interrupts happen.

- What the fourteen maskable sources are, and their order of priority
- How pulse mode and hardware IM2 mode differ
- How the daisy chain queues sources, and what `RETI` and `RETN` do to it
- How the NMI's sources, arbiter and stackless response work

Two wires into the CPU

The Z80 has two interrupt inputs. **`/INT`**, the maskable interrupt, is a level the CPU looks at once per instruction: if it is low when an instruction ends, and interrupts are enabled, the CPU answers instead of fetching the next one. I call it **the doorbell**: it rings, and the CPU answers when it has finished what it was doing, if it is willing to answer at all. **`/NMI`**, the non-maskable interrupt, works on an edge: the CPU remembers that it fell, and answers at the end of the instruction whatever `DI` said. That one is **the fire alarm**, and you will meet the fire brigade that answers it near the end. Figure 4.1 shows the two wires.

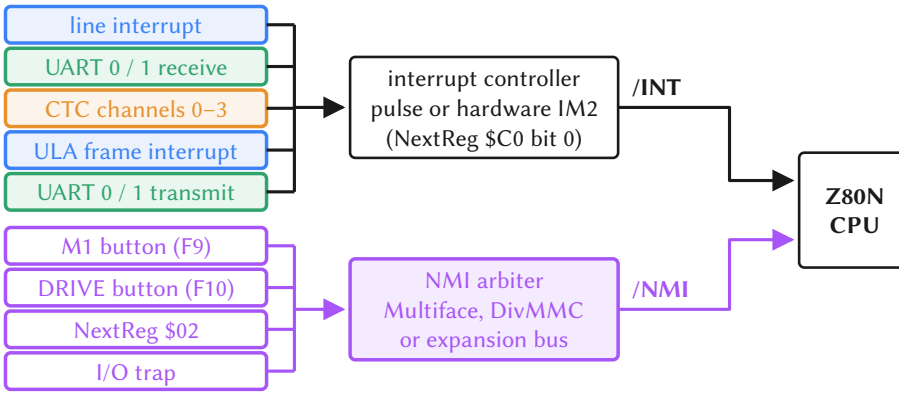


Figure 4.1: The two wires

A few rules of the CPU's own matter for everything that follows:

- The CPU decides at **the last T-state but one** of an instruction. `/INT` must be low at that moment.
- **Not after a prefix.** The CPU never stops between a `DD`, `FD`, `CB` or `ED` prefix and its opcode, so a long chain of prefixes postpones the answer.
- **`EI` takes effect one instruction late.** The instruction after `EI` always runs, which is what makes `EI` / `RETI` at the end of a handler safe; a run of `EI`s postpones the answer for as long as it lasts.
- **IM 1** calls `$0038`. **IM 2** reads a byte from the data bus, the **vector**, and calls the address stored at `I` × 256 + vector. **IM 0** executes the byte on the data bus as an instruction.

Fourteen voices in a fixed order

The Next has fourteen maskable sources, and they stand in a fixed order of priority, the same in every mode: the line interrupt first, then the two UARTs' receivers, then the CTC's channels, then the ULA's frame interrupt, and last the two UARTs' transmitters. Behind them all stands the expansion bus, for a peripheral plugged into the edge connector. Figure 4.2 lists the fourteen sources in priority order.

Two things in that list surprise people, and I like both of them. First, **the alarm clock beats the veteran**: the line interrupt, index 0, outranks the ULA's frame interrupt, index 11. That is the right way round. A raster split must land on its line or it is visibly wrong, while a once-a-frame job can start a few microseconds late and nobody notices. Second, the CTC has eight places but only **four channels**: channels 4 to 7 are reserved, never fire, and their enable and status bits read 0.

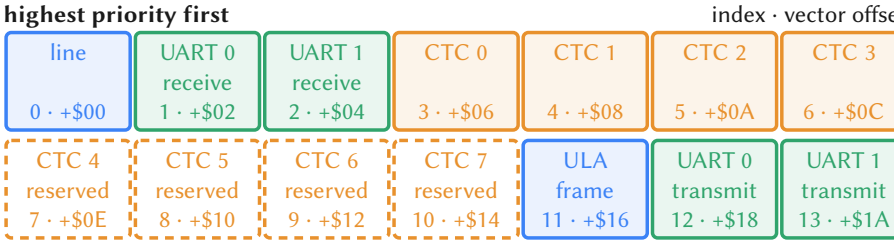


Figure 4.2: The fourteen sources in priority order (dashed: reserved)



KLIVE IDE Logpoints

A **logpoint** writes a line to the **Log** pane each time the CPU reaches it, and lets the program run on. Put one at the start of each handler, with a message that names the handler, and the log shows the order the voices are served in, without stopping the machine and spoiling the timing. For a handler at `$8000`, `bp-set $8000 -log "line interrupt, A={A}"` logs its name and the value of `A` on each visit.

Pulse mode: one pulse for everybody

After a reset the Next drives `/INT` in **pulse mode**, the way the Spectrum always did: when a source asks for attention, `/INT` goes low for a fixed time and then lets go, whatever the CPU did meanwhile. The time is counted in CPU clocks: **32 T-states** in 48K and +3 timing, **36** in 128K and Pentagon timing, at every CPU speed. So at 28 MHz the pulse is eight times shorter in real time than at 3.5 MHz. Figure 4.3 shows the pulse on a time line.

The pulse is a moment, not a promise, and here is what follows from that:

- **The pulse is shared.** Every enabled source in pulse mode starts the same pulse. A second request while the pulse is low starts nothing new, so two sources that fire close together give the CPU one interrupt.

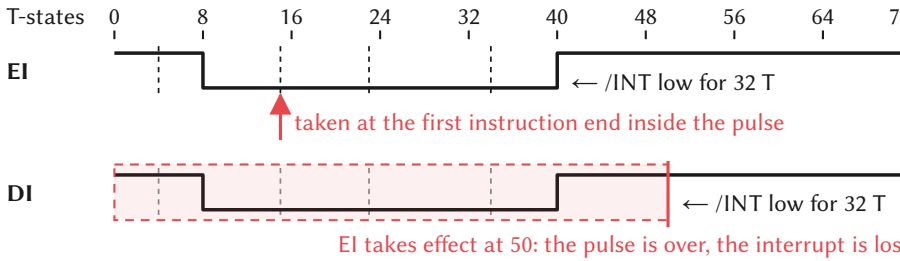


Figure 4.3: Pulse mode (dotted: instruction ends)

- **A pulse can be missed.** If interrupts stay disabled for longer than the pulse, or the DMA holds the bus across it, the CPU never sees it. Nothing holds the request; only its **status bit** remembers that it happened.
- **The CPU cannot tell who rang.** In IM 1 it calls `$0038` whatever the source. In IM 2 the data bus carries **`$FF`**, so the CPU reads its handler's address from `I × 256 + $FF`. And IM 0, which so much folklore warns about, is perfectly predictable on the Next: the FPGA drives `$FF` itself, and `$FF` is `RST $38`, so IM 0 behaves exactly like IM 1. (A device on an enabled expansion bus may put its own byte there instead.)

Since the vector says nothing, a pulse-mode handler that serves several sources reads the status bits to find out who rang. I find this the most practical trick of pulse mode:

```
Isr:    push af
        push bc
        ld bc,$243b           ; select INT Status 0
        ld a,$c8
        out (c),a
        inc b
        in a,(c)              ; bit 1: line, bit 0: ULA
        nextreg $c8,a         ; write the 1s back: clears them
        ; ... test bits 1 and 0, do the work ...
        pop bc
        pop af
        ei
        reti
```

If that handler can ever run late, it can still tell a missed pulse from no pulse at all: the status bit was set even though the doorbell rang while nobody was listening.

Hardware IM2 mode: a vector for every voice

Set bit 0 of the **Interrupt Control register**, NextReg **`$C0`**, and the Next drives `/INT` in **hardware IM2 mode**. Now a source holds `/INT` low until the CPU acknowledges it, so nothing is missed, and during the acknowledge the Next puts a vector on the data bus that says exactly who called (Figure 4.4).

Bits 7 to 5 come from NextReg `$C0`, bits 4 to 1 are the source's index from the priority list, and bit 0 is always 0, so every source gets a two-byte slot in a table of 28 bytes. Here is a setup with the table at `$BE00`, the vectors from `$A0`, and handlers for the ULA and CTC channel 0:

```
        di
        ld a,$be              ; vector table page: $BE00
        ld i,a
```

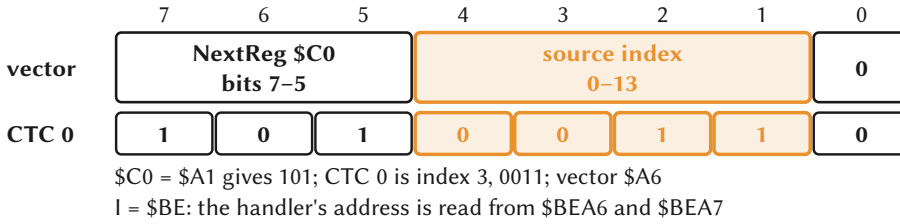


Figure 4.4: The hardware IM2 vector

```
        im 2
        ld hl,UlaIsr
        ld ($beb6),hl          ; ULA: $A0 + 11 x 2 = $B6
        ld hl,CtcIsr
        ld ($bea6),hl          ; CTC 0: $A0 + 3 x 2 = $A6
        nextreg $c0,$a1        ; vectors from $A0, hardware IM2
        ei
```

My advice: once a program has more than one source, use hardware IM2 mode. No 257-byte tables of identical bytes, no guessing who rang, and no interrupt lost because a long `LDIR` happened to run under `DI`.



KLIVE IDE Catching a setup change

A **NextReg write** breakpoint stops the machine when a register is written: `bp-set nr:$C0` for the mode and the vector, `nr:$C4` to `nr:$C6` for the enables. It catches the write whichever way it comes, through port `$253B` or by `NEXTREG`, so it finds the code that switched a source off.

The daisy chain

Behind hardware IM2 mode sits a **daisy chain**: the fourteen sources stand in a row in priority order, and each one passes a permission to the next. A source may interrupt only while the permission reaches it, and it withholds the permission from everybody behind it while it is busy. I call it **the queue**. Each source goes through four states:

1. **Idle.** It passes the permission on unchanged.
2. **Requesting.** It pulls `/INT` low if the permission reaches it and the CPU is in IM 2, and it withholds the permission, so nobody behind it can jump the queue.
3. **Acknowledged.** The CPU's acknowledge cycle reached it; it puts its vector on the bus.
4. **In service.** The CPU runs its handler. It withholds the permission until a `RETI` releases it.

Figure 4.5 shows the chain with CTC channel 0 in service.

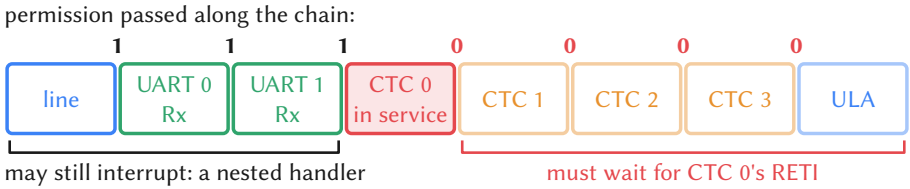


Figure 4.5: The chain with CTC channel 0 in service

In the examples below NextReg `$C0` holds `$A1`, so the vectors are `$A0` for the line interrupt, `$A6` for CTC channel 0, `$B6` for the ULA and `$B8` for UART 0's transmitter.

A raster split during the music, with the serial port waiting. Your game plays its music from the ULA's frame interrupt, splits the screen with the line interrupt, and sends bytes through UART 0 (Figure 4.6).

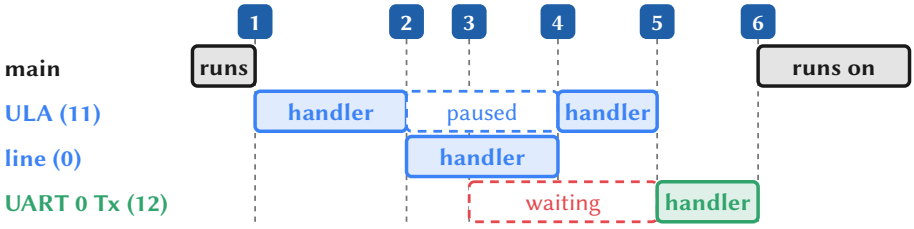


Figure 4.6: A nested handler and a waiting one (not to scale)

1. The ULA rings (vector `$B6`) and goes into service. Its handler, the music player, executes `EI` at once.
2. The alarm clock rings for the split. The line interrupt stands before the ULA, so the permission reaches it, and the music player's `EI` lets the CPU answer: vector `$A0`. The ULA's handler is paused, but the ULA stays in service.
3. UART 0's transmitter stands behind the ULA, so when its buffer empties it waits.
4. The line handler's `RETI` releases the line interrupt, the highest source in service. The transmitter still waits.
5. The music player's `RETI` releases the ULA, and the transmitter rings at once: vector `$B8`.
6. Its `RETI` releases it, and your main program carries on.

Two sources at once. The one further up the chain goes first. CTC channel 0 stands before the ULA, so a frame interrupt that fires with it starts later by the length of the CTC handler.

A handler that forgot `EI`. The CPU clears its interrupt enable when it acknowledges an interrupt, so a handler that leaves `EI` to its very end cannot be interrupted, not even by the alarm clock, and your raster split lands on a different line every frame. A handler that runs longer than a few dozen T-states executes `EI` early.

A timer faster than its handler. A source has one outstanding request, so a tick that arrives while its channel is still in service is folded into the one being served, and only the status bit records it. A sample player whose handler takes 250 T-states on a 200 T-state timer runs at half speed, and nothing crashes to tell you.

A handler that ends with `RET`. This one is my favourite bug. The chain never sees a `RETI`, so CTC channel 0 stays in service forever: CTC channels 1 to 3, the ULA and both transmitters never interrupt again, while the line interrupt and the UART receivers still work. In hardware IM2 mode a source's status bit reads 1 while it is in service, so NextReg `$C9` bit 0 stuck at 1 names the handler that forgot its `RETI`.

RETI releases, RETN does not

The chain learns that a handler has finished by watching the instructions the CPU fetches: when it sees **`RETI`**, the opcode pair `ED 4D`, the highest-priority source in service goes back to idle and the permission flows again. The rules are strict:

- **Only `RETI` releases.** `RETN` (`ED 45` and its aliases) releases nothing. A maskable handler that ends with `RETN` leaves its source in service, and that source and everything behind it are blocked for good.
- **Only in IM 2.** A `RETI` executed while the CPU is not in IM 2 releases nothing either. NextReg `$C0` bits 2–1 read back the mode the chain saw.

What if the CPU is in IM 1 while hardware IM2 mode is on? Then the chain cannot interrupt at all, with one exception: **the veteran still rings in the old way**. The ULA's frame interrupt alone falls back to a pulse, so a program that switches hardware IM2 mode on but stays in IM 1 still gets its fifty interrupts a second. The other sources move to "requesting" and wait there. The moment the CPU executes `IM 2`, they all ring at once, in priority order.



KLIVE IDE Back to the interrupted instruction

Pause in a handler and step back: from its first instruction, **Step Back** lands on the instruction the interrupt broke into, and **Reverse Step Out** gets there from anywhere in the handler. The **Call Stack** panel says how each routine was entered: by `CALL`, `RST`, an interrupt or an NMI.

Switches and status bits

Every source has an **enable**, and the enables sit in three registers, the **INT EN 0**, **INT EN 1** and **INT EN 2 registers**, NextRegs **`$C4`**, **`$C5`** and **`$C6`**:

- **NextReg `$C4`**: bit 0 the ULA, bit 1 the line interrupt, bit 7 the expansion bus. It reads `$81` after a reset: the ULA and the bus on, the line interrupt off.
- **NextReg `$C5`**: bits 3–0 the CTC channels. A channel's enable is the same flip-flop as bit 7 of its own control word, so writing either one overwrites the other.
- **NextReg `$C6`**: for each UART, receive (bit 0 for UART 0, bit 4 for UART 1), receive "near full only" (bits 1 and 5) and transmit (bits 2 and 6).

The ULA's switch is shared by three doors, all leading to one flip-flop: port `$FF` bit 6 (1 = off), bit 2 of the **Line Interrupt Control register**, **NextReg `$22`** (1 = off), and bit 0 of NextReg `$C4` (1 = on, the other way round). The line interrupt's enable is bit 1 of both NextReg `$22` and NextReg `$C4`. Whichever you wrote last wins, so pick one register and stick to it.

Every source also has a **status bit**, in the **INT Status 0, 1 and 2 registers**, **NextRegs `$C8`, `$C9` and `$CA`**, laid out like the enables. A status bit is set by the source's event, and **writing a 1 clears it**; writing a 0 leaves it alone.

- **The CTC and the UARTs set their status even when their interrupt is disabled.** That is "polled mode": leave the interrupt off and watch the bit. The ULA and the line interrupt are different: switched off, they raise nothing, not even a status bit.
- **In hardware IM2 mode a status bit reads 1 while its source is requesting or in service**, and writing a 1 cannot clear it until the `RETI`.
- **NextReg `$22` bit 7 reads the pulse.** It is 1 while the pulse-mode `/INT` is low, from any source, not only the ULA.

And a program can ring the doorbell itself. The **Generate Maskable Interrupt register**, **NextReg `$20`**, raises a request when you write a 1: bit 7 the line interrupt, bit 6 the ULA, bits 3–0 the CTC channels. These requests ignore the enables, set the status bit and behave like the real thing in either mode. I use it to test a handler without waiting for the hardware.



KLIVE IDE The status bits, one by one

Pause the machine and select NextRegs `$C0` to `$CF` one by one in the **Next Registers** view. The inspector gives each enable and status bit a line of its own, with its meaning, so you can see which interrupts are pending and which are switched off.

▶ TRY IT Fourteen Voices

Part I, key 5. The demo switches to hardware IM2 mode with the vectors from `$A0`, as in this chapter's setup, and gives three sources a handler and a counter each: the line interrupt at line 96 (`$A0`), CTC channel 0 at 437.5 times a second (`$A6`) and the ULA (`$B6`). The line handler turns the border red, and the ULA's handler turns it white

▶ TRY IT Fourteen Voices *(continued)*

again, so the screen splits at line 96. The code is in `examples/code/demos/1-machine/fourteen-voices.kz80.asm`. Each handler is as short as this one:

```
FvCtcIsr:
    push hl
    ld hl,(FvCtcCount)
    inc hl
    ld (FvCtcCount),hl
    pop hl
    ei
    reti
```

For an experiment with the daisy chain, change this handler's `RETI` to `RET` and run the demo again. CTC channel 0 stays in service: its counter stops at 1, the ULA's counter stops too, because the ULA stands behind it in the chain, and the border stays red. The line interrupt stands before both, so its counter goes on.

The sources up close

The ULA's frame interrupt comes once a frame at a fixed point of the beam: on line 0 in 48K timing and at 60 Hz, on line 1 in 128K and +3 timing at 50 Hz, and on the last line in Pentagon timing, which has no 60 Hz frame. Chapter 10 has the details.

The line interrupt, the alarm clock, rings for line n near the end of line $n - 1$'s display area, in the line numbers Chapter 10 explains. A line number one past the last line of the frame rings like line 0; anything higher never rings.

The CTC raises a channel's interrupt when the channel's count reaches zero. Its timer counts the 28 MHz master clock through a prescaler of 16 or 256, so its rate does not change with the CPU's speed. The vector a program writes into a Zilog CTC is thrown away on the Next; it comes from the chain. Chapter 6 opens the channels up.

The UARTs have two receive conditions and one transmit condition:

- **Receive, data available**: the receive buffer holds at least one byte.
- **Receive, near full**: the 512-byte buffer holds 384 bytes, three quarters. With the "near full only" bit set in NextReg `$C6`, only this one interrupts.
- **Transmit**: the transmit buffer has become empty. That is about one character's time before the line falls silent.

All requests are **edges**: a source asks once when its condition becomes true, not for as long as it stays true. For the UART that means a byte arriving into a buffer that is not empty raises nothing new, so drain the whole buffer in the handler.

How fast the CPU answers

Between a source's request and its handler's first instruction there are two delays: the instruction that is running must end, and then the acknowledge itself takes its time. Figure 4.7 shows both delays.

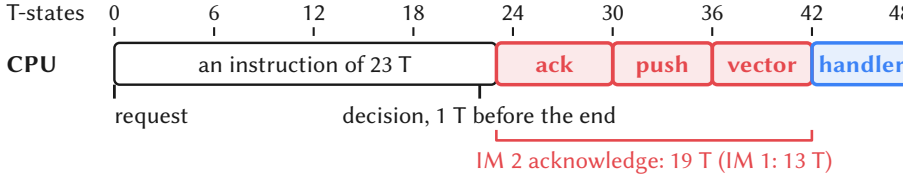


Figure 4.7: How fast the CPU answers

- **The running instruction ends.** The longest Z80N instruction takes 23 T-states, and the CPU decides one T-state before an instruction ends, so the wait is anything from 0 to 22 T: up to 44 dots of jitter at 3.5 MHz. Contention can make it longer still: in contended memory or I/O, each access can wait up to 6 more T-states, 7 in +3 timing.
- **The acknowledge.** In IM 1 it takes **13 T-states**: an acknowledge cycle of 7 and the two writes that push the return address. In IM 2 it takes **19**: the same, and two reads that fetch the handler's address.
- **`HALT` helps, but not completely.** A halted CPU answers only at the end of a 4 T-state NOP cycle, so `HALT` still leaves 0 to 3 T of jitter.



KLIVE IDE What interrupts cost

Debug › Start Profiling, a few frames, then **Debug › Stop Profiling**: the profiler shows each handler as a root of its own, such as `IM 2 → $FDFD (Music)`, so its time is not charged to the code it interrupted. The acknowledge cycles have a row of their own, **Interrupt acknowledge**.

The DMA steps aside

The DMA can hold the bus for a long transfer, and in pulse mode an interrupt that comes and goes during it is simply lost. In hardware IM2 mode the Next can do better: the **DMA INT EN 0, 1 and 2 registers**, **NextRegs `$CC`, `$CD` and `$CE`**, laid out like the enables, choose which sources may interrupt the DMA. While such a source is requesting or in service, the DMA stops asking for the bus between bytes, the CPU takes the interrupt and runs the handler, and the DMA carries on after the `RETI`. Bit 7 of NextReg `$CC` does the same for the NMI. This works only in hardware IM2 mode, and the DMA itself raises no interrupt when it finishes.

The fire alarm

On the Next the NMI belongs to two devices you met in Chapter 3: **the fire brigade**, the Multiface, and **the butler**, the DivMMC. Its sources are:

- **The M1 button** (F9 on a PS/2 keyboard) asks for a Multiface NMI, allowed by bit 3 of the **Peripheral 2 Setting register**, **NextReg `$06`**.
- **The DRIVE button** (F10) asks for a DivMMC NMI, allowed by bit 4 of NextReg `$06`. It works only while the DivMMC's port is switched on.
- **The Reset register**, **NextReg `$02`**: writing bit 3 asks for a Multiface NMI, bit 2 for a DivMMC NMI, under the same enables. The Copper can write it too, so a Copper list can raise an NMI at a chosen point of the frame. The bit stays set when you read the register back, so a handler can tell that a write, not a button, asked; writing the bit as 0 clears it.
- **An I/O trap**: with the **I/O Traps register**, **NextReg `$D8`**, bit 0 set, touching the +3's disk ports `$2FFD` and `$3FFD` asks for a Multiface NMI.
- **The expansion bus**, for a device on the edge connector.

Only one NMI can be under way at a time, and an **arbiter** picks the source (Figure 4.8).

1. **The Multiface first**, unless the DivMMC is mapped in for any reason or port `$E3` bit 7 holds the DivMMC in by hand.
2. **Then the DivMMC**, unless the Multiface is active.
3. **Then the expansion bus.**

A request that loses is gone; it does not wait. Once a source is taken, `/NMI` goes low at once. After the CPU has fetched the opcode at `$0066`, a new request changes nothing, not even the flags of NextReg `$02`, until the device that answered is done.

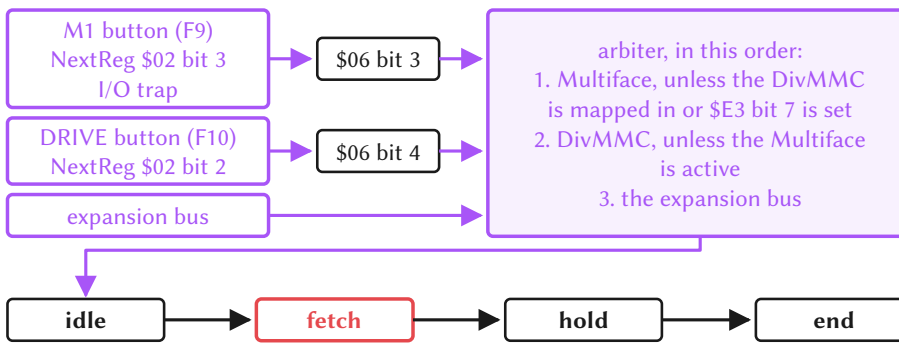


Figure 4.8: The NMI arbiter

Who answers at $0066

The NMI sends the CPU to `$0066`, and whoever the arbiter chose appears there to answer:

- **The fire brigade** pages itself in during that very fetch: the Multiface's ROM at `$0000`, its RAM at `$2000`. It leaves on a `RETN` or when the program touches its ports, which depend on the Multiface type chosen in the **Peripheral 5 Setting register**, **NextReg `$0A`**.
- **The butler** comes in through his `$0066` entry point, instant or delayed as bits 1 and 0 of the **Divmmc Entry Points 1 register**, **NextReg `$BB`**, say. He leaves on a `RETN` or through `$1FF8`–`$1FFF`.

For both, only the plain `RETN`, `ED 45`, counts as "the NMI is over"; its aliases and `RETI` do not.

The stackless NMI

An ordinary NMI pushes its return address onto the stack, wherever the stack happens to be. Set bit 3 of the Interrupt Control register, **NextReg `$C0`**, and the NMI becomes **stackless** (Figure 4.9).

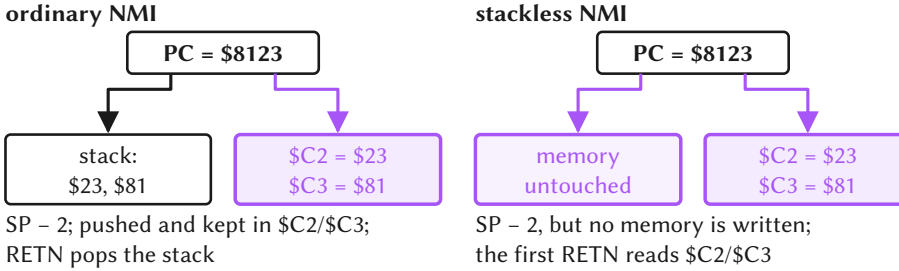


Figure 4.9: An ordinary NMI and a stackless one

- The CPU still runs both push cycles and `SP` still goes down by 2, but the memory request is held off, so **nothing is written**.
- The return address goes to the **NMI Return Address LSB and MSB registers**, **NextRegs `$C2` and `$C3`**. In fact they record it on **every** NMI, stackless or not, and a handler may read or rewrite them.
- **The first `RETN`** after the NMI takes its return address from NextRegs `$C2`/`$C3` instead of the stack, and `SP` goes up by 2 as usual. A `RETI` in the handler does not: it pops the real stack, like any return.



KLIVE IDE Watching an NMI

In the **Execution History** an NMI is a row of its own, `— NMI —`, before the first instruction at `$0066`, and the **Call Stack** panel marks the handler as entered by an NMI. The **Interrupts** panel in the **Machine info** activity shows the arbiter's state and the device it chose, the two enables, the I/O trap, the stackless mode and the return address. **Debug › Interrupt Timeline**, or `show-int-timeline`, marks each NMI and its `RETN` with the device that answered.

Resets

A reset puts the interrupt system back into the Spectrum's shape: NextReg `$C0` becomes 0 (pulse mode, the stackless NMI off), NextReg `$C4` becomes `$81` (the ULA and the expansion bus on, the line interrupt off), NextRegs `$C5`, `$C6` and `$CC`–`$CE` become 0, NextReg `$22` and `$23` become 0, and every status bit and every pending request is cleared. Leaving hardware IM2 mode does the same to the chain: every request and every handler in service is forgotten. So set the interrupt registers yourself at the start of your program.

In short. Fourteen maskable sources share `/INT` in a fixed priority, with the line interrupt above the ULA's frame interrupt. Pulse mode gives one short shared pulse that can be missed, so a handler serving several sources reads the status bits; bit 0 of NextReg `$C0` selects hardware IM2 mode, where a source holds `/INT` until acknowledged and supplies its own vector. Any handler longer than a few dozen T-states should execute `EI` early. End every handler with `RETI`, because `RET` or `RETN` leaves its source in service and blocks everything behind it for good.

CHAPTER 5

DMA: One Bus, Two Masters

Copy 16K with `LDIR` on a Next at 3.5 MHz and the CPU spends 344,059 T-states on it, almost five frames. Ask the DMA to do the same job and it is done in 98,304 T-states, less than a frame and a half. The DMA is not a faster CPU, and it does not run beside the CPU either. So where do the other three and a half frames go?

Sprite patterns, a Layer 2 screen to clear, a sample to play while the game runs, a sector to pull from the SD card: all of them are "move these bytes from here to there", and the DMA does nothing else. Once you know how it shares the bus, you can choose between a copy that stops the world for a moment and a trickle that the CPU hardly notices.

This chapter builds on Chapter 3, for the MMU and the port decoder, which the DMA uses as the CPU does, and on Chapter 4, for hardware IM2 mode, which lets an interrupt break into a transfer. The facts are for core 3.03, the current core, whose DMA is the **zxnDDMA**, the blitter; the older core 3.02 has the same DMA without the blitter commands of the last section, and the text says where else it differs.

- How the DMA borrows the bus from the CPU
- How its two ports differ, and how to program its registers
- How its timing works: six T-states a byte, the prescaler and burst mode
- How auto restart, reading back, interrupts and wait states work
- What the blitter commands add: physical addresses, rectangles, a see-through colour, fill and Layer 2 positions

A second master on the bus

The Z80 has a pair of pins for exactly this situation. **`/BUSRQ`**, bus request, is an input: a device pulls it low when it wants the bus. **`/BUSAK`**, bus acknowledge, is the CPU's answer: at the end of its current machine cycle the CPU lets go of its address, data and control lines and pulls `/BUSAK` low. The device then owns the bus, and the CPU waits, frozen in the middle of its instruction, until `/BUSRQ` goes high again.

The Next's DMA is such a device, and I call it **the courier**: you hand it the parcel and the address once, and it delivers every byte without asking you again. Inside the

FPGA its lines go through a switch, and while `/BUSAK` is low, the switch puts them where the CPU's lines would be (Figure 5.1).

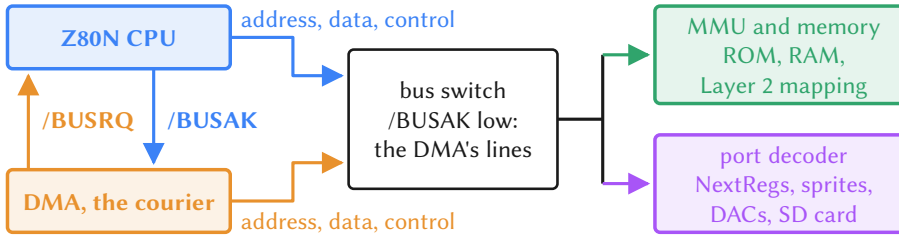


Figure 5.1: One bus, two masters

Everything behind the switch sees the DMA as if it were the CPU, and that is why it is so useful:

- **Memory goes through the MMU.** The DMA's addresses are 16-bit Z80 addresses, mapped by NextRegs `$50`–`$57`. To reach any bank, you page it in first. The ROM is read-only to the DMA too, and the Layer 2 write mapping of port `$123B` catches its writes. The one exception is the blitter's **physical mode** on core 3.03, which skips the MMU and names a byte of the static RAM directly (page 78).
- **I/O goes through the port decoder.** A DMA write to port `$5B` loads a sprite pattern byte, a write to `$DF` sets a DAC, a write to `$253B` writes the selected NextReg.
- **Two exceptions.** The DMA's own ports do not answer while it holds the bus, so it cannot reprogram itself. And the I/O traps of the **I/O Traps register**, **NextReg `$D8`**, do not fire for its accesses.

The CPU does not run while the courier works. The DMA saves time by moving a byte in 6 T-states where `LDIR` needs 21, and that answers the opening puzzle: the three and a half frames were never spent at all.

Two details of the handover matter. When an `OUT` starts the DMA, the CPU still fetches the next instruction's opcode before the DMA takes over. And handing the bus over takes a few clocks each way, so a transfer of a few bytes costs a few T-states more than its bytes alone.



KLIVE IDE The DMA in the Execution History

When the DMA holds the bus, the **Execution History** gets a row of its own with the T-states it held the bus for, the addresses it copied from and to, and the bytes left. The CPU's instructions before and after it show where the transfer started and where the CPU picked up again.

Two ports, two dialects

The DMA answers on two ports, **`$6B`** and **`$0B`**. Only the low byte of the port address is decoded, so `OTIR`, which changes `B` as it goes, sends a whole table to either port. Every read or write of `$6B` puts the DMA into **zxnDMA mode**, the Next's own, and every read or write of `$0B` into **Z80 DMA mode**, which copies the **Zilog Z80 DMA**, the Z8410, a companion chip for the Z80 and "the Zilog chip" in this chapter. The mode stays until you touch the other port, and a reset brings back zxnDMA mode. The blitter commands of the last section work in zxnDMA mode only.

The dialects differ only in where the byte counter starts. The transfer goes on as long as the counter is below the block length, checked after each byte and never before the first:

- **zxnDMA mode**: the counter starts at 0, so a block length of *N* moves exactly *N* bytes. Use this mode for every new program.
- **Z80 DMA mode**: the counter starts at `$FFFF`, so a block length of *N* moves *N* + 1 bytes, as on the Zilog chip.
- **A block length of 0 moves one byte** in both modes, never none. The largest block is `$FFFF` bytes in zxnDMA mode and 65,536 bytes in Z80 DMA mode.

Bit 5 of NextReg `$82` enables `$6B`, and bit 1 of NextReg `$85` enables `$0B`. Both are on after a reset.



KLIVE IDE Catching a DMA program

An **I/O write** breakpoint on the DMA's port stops at every byte a program sends it. `bp-set $6B -o -m $FF` compares only the port's low byte, so an `OTIR` stops there whatever is in `B`. Step on from each stop to watch the program arrive, one base byte and its followers at a time.

Base bytes and their followers

The DMA has no register select port. You write a stream of bytes, and the DMA works out what each one is. A **base byte** names a register group, WR0 to WR6, by a pattern of fixed bits (Figure 5.2), and some of its other bits announce **followers**, the parameter bytes that must come next, in a fixed order. After the followers, the next byte is a base byte again.

The shaded bits are fixed, and the dots are the group's settings: WR0 is any byte with bit 7 clear whose bits 1 and 0 are not both 0. A byte that matches no pattern, such as `$86`, is ignored. And a byte the DMA expects as a follower is taken as one whatever its value, so a sequence that announces three followers and sends two swallows your next base byte. Write each group in one place, with its followers next to it, and start every setup with a disable, `$83`.

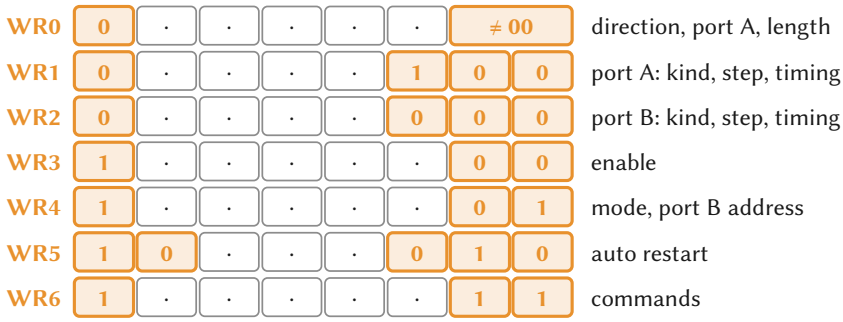


Figure 5.2: The base bytes, bit 7 on the left



KLIVE IDE Mark As DMA

In the Disassembly view, or in a bank popped out of a NEX file, select the bytes a program sends to `$6B` and choose **Mark As DMA**. Klive lists them as a DMA program, one `.dma` command per row, and each row assembles back to the same bytes: the decoding of this section, done for you.

Port A and port B

Like the Zilog chip, the DMA speaks not of a source and a destination, but of **port A** and **port B**. Each is an address with its own settings, and a direction bit says which way the bytes flow, so you can set up a buffer and a device once and run the transfer either way by changing one bit.

WR0 sets the direction, port A's starting address and the block length:

```
WR0:  0 L L A A D t t
  t t = not 00 (01 for a transfer)
  D   = direction: 1 = A to B, 0 = B to A
  A A = port A address high / low follows
  L L = block length high / low follows
followers, in this order:
  port A low, port A high, length low, length high
```

Only the followers whose bits are set are sent, but always in that order: `$6D`, with bits 6, 5 and 3 set, expects port A's low byte, then the length's low and high bytes. The Zilog chip's search does not exist here: every WR0 is a transfer.

WR1 describes port A and **WR2** port B, in the same layout:

```
WR1:  0 T M M I 1 0 0      WR2:  0 T M M I 0 0 0
      I   = 1: an I/O port, 0: memory
      M M = after each byte: 00 down, 01 up, 10 or 11 fixed
      T   = a timing byte follows
timing byte:  x x P x x x c c
```

```
      c c = cycle length: 00 4, 01 3, 10 2, 11 4 clocks
      P   = one more byte follows: WR2's prescaler
```

A copy steps both addresses up. A fill keeps a one-byte source **fixed** and steps the destination; a device such as a DAC keeps its port fixed. A move between overlapping buffers steps both addresses **down**, for the same reason `LDDR` exists. The **cycle length** is how many clocks the DMA spends on each read and each write of that port: three after a reset, until a **timing byte** sets it. WR1's timing byte can announce a prescaler byte too, but port A has no prescaler, so that byte is thrown away.



KLIVE IDE Breakpoints that see the DMA

A **NextReg write** breakpoint and a **sprite attribute** breakpoint fire when the DMA does the writing too: `bp-set sp:$0C` stops at any write to sprite 12's attributes, a DMA transfer to port `$57` included. A DMA that feeds the sprites or the NextRegs does not slip past them.

The mode and port B's address

WR4 chooses the operating mode and sets port B's starting address:

```
WR4:  1 m m i B B 0 1
  m m = 00 byte, 01 continuous, 10 burst
  B B = port B address high / low follows
  i   = interrupt control byte follows
followers, in this order:
  port B low, port B high
```

In **continuous mode** the DMA keeps the bus from the first byte to the last. In **burst mode** it gives the bus back while the prescaler makes it wait, and only then. **Byte mode** behaves exactly like continuous mode, and so does burst mode without a prescaler.

Leave bit 4 alone: the DMA raises no interrupts, and this bit has two traps. With port B's address bytes, the interrupt control byte you send after them is decoded as a new base byte. Without them, the DMA waits for an interrupt control byte it never takes, and stays deaf to every byte you write until the machine is reset. A base byte such as `$91` is enough to lock it.

Enables and flags

WR3 is mostly the Zilog chip's interrupt and search register, and almost none of it is here:

```
WR3:  1 E x H K x 0 0
  E = enable the DMA (like command $87)
```

```
K = a mask byte follows
H = a match byte follows (after the mask)
```

The mask and match bytes are taken and dropped, and the interrupt and stop-on-match bits do nothing.

WR5 is a single byte, `1 0 R C x 0 1 0`. Bit 5, **R**, is **auto restart**; bit 4, the Zilog chip's CE/WAIT setting, is stored but changes nothing. So the WR5 you write is `$82`, or `$A2` for auto restart.

Commands

WR6 is the command register: any base byte with bit 7 set and bits 1–0 = 11 is a command, and the DMA acts on it at once. These do something:

- **`$C3`, reset.** Stops any transfer, resets the status byte, sets both cycle lengths back to three clocks, clears the prescaler and auto restart, and clears every blitter setting. It keeps the start addresses, the block length, the direction, the address modes, the operating mode and the read mask.
- **`$C7` and `$CB`, reset port A's or port B's timing.** They set that port's cycle length back to three clocks and leave everything else alone; port B's prescaler stays.
- **`$CF`, load.** Copies the start addresses, and with them the blitter's extended address bytes, into the DMA's working source and destination, port A to the source for A to B, port B for B to A; sets the counter to its start; and clears the end-of-block flag.
- **`$D3`, continue.** Sets the counter to its start and clears the end-of-block flag, but keeps the addresses where they are, so the next block follows on from the last.
- **`$87`, enable.** Starts the transfer. **`$83`, disable**, stops it where it is; a later `$87` goes on from there.
- **`$BB`**, **`$A7`**, **`$BF` and `$8B`** belong to reading the DMA back, later in this chapter.
- **`$D7` to `$F7`** are the blitter's own commands, each with bytes of its own, in the last section.

The rest are the Zilog chip's, and the Next takes them but does nothing: the interrupt commands, `$AF` disable interrupts, `$AB` enable interrupts, `$A3` reset and disable interrupts and `$B7` enable after `RETI`, and `$B3`, force ready.

Here is the trap I would warn every newcomer about: **the addresses you write are not the addresses the DMA uses.** WR0 and WR4 fill the start registers; only `$CF` copies them into the working addresses. Enable without a load, and the DMA carries on from wherever its last transfer stopped, which after a finished block means one more byte from just past its end. So the last two bytes of every setup are `$CF`, `$87`, in that order.



KLIVE IDE Tools for the DMA

The **DMA** panel in the **Machine info** activity shows every register at a pause and warns about traps like an enable without a load. **Debug › DMA Port Log** lists each byte a program sent, as the DMA took it. `bp-set dma:end` stops when a block ends, and `step-dma` moves one byte at a time. **Debug › DMA Transfers** lists the transfers; select one to see its bytes in the Memory view.

One byte, six T-states

A transfer is a loop of two bus cycles: the DMA **reads** a byte from the source and **writes** it to the destination. Each cycle lasts the cycle length of its port, three clocks after a reset, so a byte takes **six T-states** (Figure 5.3).

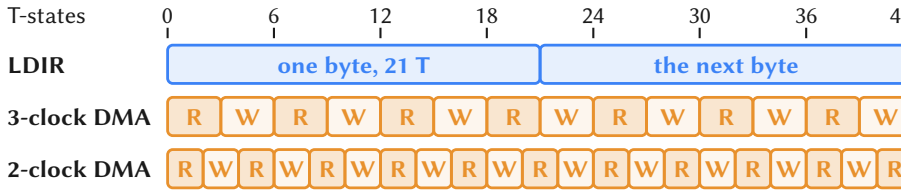


Figure 5.3: LDIR against the DMA, to scale (R: read, W: write)

For a 16K copy, `LDIR` takes 21 T-states for every byte but the last, which takes 16: 344,059 T-states. The DMA takes 6 × 16,384 = **98,304 T-states**, plus a few for the handover. With two-clock cycles, a timing byte of `$02` for both ports, a byte takes **four T-states** and the copy 65,536.

The clocks are the CPU's clocks, so at 28 MHz the same six clocks take an eighth of the time. Two things can stretch a byte, and on the older core a third, all of them wait states the CPU would get too:

- **Contention.** At 3.5 MHz, a DMA byte that touches contended memory in the display area waits, exactly as the CPU would. Contention disappears at 7 MHz and above, and when NextReg `$08` bit 6 switches it off.
- **The SD card.** Port `$EB` shifts each byte out one bit at a time, and it holds the DMA's next access until the last byte has finished, 16 CPU clocks. So a DMA from `$EB` to memory reads a sector at the speed of the card's serial link, without a single byte lost.
- **28 MHz memory, on the older core only.** The current core has no wait states at 28 MHz, so a memory to memory byte takes its six clocks there too. On the older core 3.02 a read from the main memory gets a wait state at 28 MHz, so the same byte takes seven.

If you ask me, the two-clock cycle is the one for memory to memory copies, and the three-clock cycle for anything that talks to a device. I have no evidence that a device minds; I like the margin.

▶ TRY IT DMA Versus LDIR

Part I, key 6. At 3.5 MHz, the demo copies the same 1K from the ROM three times a frame: by `LDIR`, by the DMA with three-clock cycles, and by the DMA with two-clock cycles. It changes the border's colour as each copy starts, so each copy draws a stripe as tall as the time it took: red for `LDIR`, green and yellow for the DMA. The code is in `examples/code/demos/1-machine/dma-versus-ldir.kz80.asm`, and the two-clock program is written with `.dma`, a Klive assembler pragma that turns each line into the DMA's register bytes. It is a convenience of Klive's Z80 assembler, not a Z80 instruction:

```
DvlDma2:
    .dma disable
    .dma wr0 a_to_b, transfer, DvlSource, DvlLength
    .dma wr1 memory, increment, 2t
    .dma wr2 memory, increment, 2t
    .dma wr4 continuous, DvlDest
    .dma wr5
    .dma load
    .dma enable
DvlDma2End:
```

The stripes come out about 95, 29 and 20 lines tall: 21, 6 and 4 T-states a byte, plus the handful of T-states that each `OTIR` spends sending its program. The three-clock program is the same but for `3t`. Each program sets its cycle length, because a DMA keeps the last one it was given; the demo's last act is a reset that puts three clocks back.

The prescaler: spacing the bytes out

For a sample, the rate *is* the sound: each byte must arrive at the DAC at exactly the right moment. The **prescaler**, WR2's third byte and a Next addition, does that.

It works with a timer that counts **28 MHz time**: it goes up by 8 for each CPU clock at 3.5 MHz, 4 at 7 MHz, 2 at 14 MHz and 1 at 28 MHz, and it starts again from zero at the start of every byte. After the write, the DMA waits until the timer has reached 32 times the prescaler, *P*. That gives the familiar rate: 28 MHz ÷ 32 ÷ *P*, **875 kHz ÷ *P***.

The real rate is a little lower. When the wait ends, the DMA needs two more clocks to get to the next read, and the first clock of a byte is spent before the timer starts counting. That makes a byte every **4 × *P* + 3 T-states** at 3.5 MHz, and every 32 × *P* + 3 clocks at 28 MHz:

- **Prescaler 100 at 3.5 MHz**: a byte every 403 T-states, 8,685 bytes a second, where the formula promises 8,750.
- **Prescaler 100 at 28 MHz**: a byte every 3,203 clocks, 8,742 bytes a second.

The rule holds whenever the wait is longer than the byte itself. At 3.5 MHz a prescaler of 1 is not, and changes nothing; 0 switches the prescaler off at every speed. The DMA also waits after the last byte, before it calls the block finished.

In continuous mode the DMA **keeps the bus while it waits**, and the CPU is frozen for the whole, much longer, block. With a prescaler, use burst mode.

Burst mode: the bus goes back between bytes

In burst mode the courier hands the bus back while it waits. On the first clock of its wait it lets go of `/BUSRQ`, and a couple of clocks later the CPU picks up its instruction where it was frozen. When the wait ends, the DMA pulls `/BUSRQ` again, the CPU stops at the end of its current machine cycle, and the next byte moves (Figure 5.4).

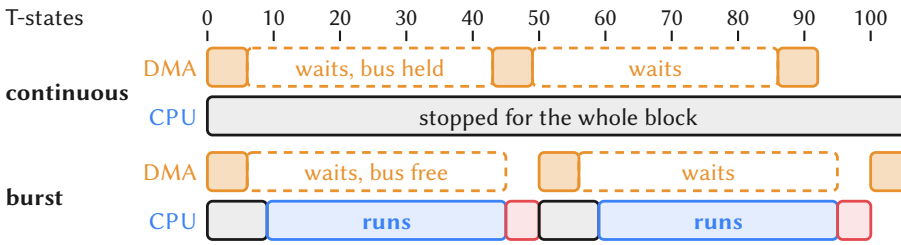


Figure 5.4: Prescaler 10 at 3.5 MHz, to scale (red: the DMA asks for the bus)

Burst mode costs a little: each byte comes a few T-states later than the prescaler alone would say, and the bytes jitter by a T-state or two. In return, at prescaler 100 the courier holds the bus for about a dozen T-states of every four hundred, and the CPU keeps well over 95 % of its time.

Let's play a sample. A **DAC** turns each byte written to it into a level on the sound output; Chapter 24 covers the Next's four of them. This program plays 2,048 bytes from `$C000` to the Specdrum DAC on port `$DF` at prescaler 55, about 15,500 bytes a second, while the CPU goes back to its work. That is the **sample rate**, not the pitch you hear: the pitch comes from the shape the bytes draw, and this rate carries tones up to half of it, about 7.7 kHz, enough for speech and drums. Here is the program:

```
        nextreg $08,$18         ; DACs and speaker on
        ld hl,Prog
        ld bc,$106b             ; B = 16 bytes, C = port
```

```
        otir
        ret                     ; the sample plays on
Prog:   .defb $83               ; disable
        .defb $7d,$00,$c0       ; WR0: A to B, port A $C000
        .defb $00,$08           ; ... length 2,048
        .defb $14               ; WR1: A is memory, upwards
        .defb $68,$21,55        ; WR2: B is I/O, fixed; P = 55
        .defb $cd,$df,$00       ; WR4: burst, port B $00DF
        .defb $82               ; WR5: no auto restart
        .defb $cf,$87           ; load, enable
```

`OTIR` sends the table to port `$6B`, which also selects zxnDMA mode, so the length means exactly 2,048 bytes.



KLIVE IDE What a transfer costs

Debug › Start Profiling, then **Stop Profiling**: the profiler puts the time the DMA held the bus in a row of its own, **DMA bus hold**, charged to no routine. Run the same transfer in continuous and in burst mode, and compare how much of a frame each one takes.

Auto restart

With WR5 bit 5 set, the DMA does not stop at the end of a block. It sets the end-of-block flag, reloads its working addresses and its counter, and moves the first byte of the next round, keeping the bus if it had it.

It has one dangerous corner. In continuous mode, with a prescaler or without, an auto-restarting DMA **never gives the bus back**: the CPU stops for good, and it cannot even write the `$83` that would stop the DMA. Only a reset, or an interrupt the DMA has been told to step aside for, gets the CPU running again, and that interrupt's handler can write the `$83`. Pair auto restart with burst mode and a prescaler, every time.

Reading the DMA back

Read the DMA's port, and you get the next entry of its **read sequence**: the status byte, the counter's low and high bytes, port A's low and high address bytes, and port B's, seven entries in this order. The **read mask**, the byte after a `$BB` command, chooses the entries, bit 0 for the status up to bit 6 for port B's high byte. After the last enabled entry the sequence starts again. `$A7` starts it again from its first enabled entry, and `$BF` makes the next read the status, whatever the mask.

The **status byte** has the form `00E1101T`:

- **E**, bit 5, is **0 once the end of a block has been reached**, and 1 before that. A load, a continue, a reset or `$8B` sets it back to 1.

- **T**, bit 0, is 1 when a byte has been moved in the transfer under way. It never reads 1 after a transfer has finished.
- So `$3A` means idle and nothing done yet, `$3B` a transfer under way, and `$1A` finished.

During a continuous transfer the CPU cannot read the DMA, so your main program sees only `$1A` at the end. In burst mode, polling works:

```
Wait:   ld a,$bf                ; read the status next
        out ($6b),a
        in a,($6b)
        bit 5,a                 ; end of block reached?
        jr nz,Wait
```

Stepping aside for interrupts

Chapter 4 explains how the **DMA INT EN 0, 1 and 2 registers, NextRegs `$CC`, `$CD` and `$CE`**, let chosen interrupt sources break into a transfer in hardware IM2 mode. Seen from the DMA:

- After each byte, if such a source is requesting or in service, the DMA lets go of the bus and waits until it is not. The transfer goes on after the `RETI`.
- Without these enables, or in pulse mode, the interrupt waits for the end of the transfer. A pulse-mode interrupt that comes and goes while the DMA holds the bus is lost, so a continuous transfer longer than a frame, such as a 16K copy at six T-states a byte, costs you a frame interrupt.
- A transfer started under `DI` can wait for ever: if one of the enabled sources is already requesting, the CPU never takes the interrupt, the request never goes away, and the DMA never gets its bus back. Start a transfer with these enables on only while interrupts are enabled.

Resets

A reset, hard or soft, puts the DMA into zxnDMA mode and continuous mode, idle, with three-clock cycles, no prescaler, no auto restart, the read mask `$7F` and the status `$3A`, and frees a DMA that WR4's bit 4 left deaf. It does not clear the start addresses, the block length, the direction or the address modes, so write every register a transfer needs. The command `$C3` also clears every blitter setting, as the next section explains.

The zxnDDMA: the blitter

Everything so far is the DMA moving a line of bytes. The Next's DMA can do more: its full name is the **zxnDimensionalDMA**, **zxnDDMA** for short, and the Next's release

notes simply call it the blitter. It is in the core of every board, Issue 2, Issue 4 and Issue 5. The blitter is a handful of extra commands, all shaped like WR6 commands, on top of everything this chapter has shown, and the DMA takes them **on port `$6B` only**. The older core 3.02 has none of them.

Here they are, each with the bytes that follow it:

- `$D7` and `$DB`, one byte each: **bits 23–16 of port A's and port B's address**, the extended byte.
- `$DF`, one byte: **the mode flags**, below.
- `$E3`, `$E7`, `$EB` and `$EF`, two bytes each, low byte first: the **row length**, the **number of rows**, **port A's stride** and **port B's stride**.
- `$F3`, one byte: the **transparent colour**. `$F7`, one byte: the **fill value**.

`$DF` switches the new behaviour on, bit by bit: bit 0 **physical**, bit 1 **2D**, bit 2 **transparent**, bit 3 **fill**, bit 4 **Layer 2 X,Y** and bit 5 **hidden screen**; bits 7–6 stay 0. Notice that **each `$DF` sets every flag at once**, so it is not a bit you turn on but a whole set you send.

Physical addresses

With bit 0 set, the DMA forgets the MMU. An address is 22 bits: bits 5–0 of the extended byte above the 16-bit address (Figure 5.5), and it names a byte of the static RAM directly, where RAM's 16K bank n starts at `$40000` + n × `$4000` and its 8K page n at `$40000` + n × `$2000`. Bits 7–6 of the extended byte are dropped. Counting up carries from the 16-bit address into the extended byte at `$FFFF`, and counting down borrows at `$0000`, so one transfer can run across a 64K boundary. No more paging memory in and out between the pieces of a copy: if you ask me, that alone is worth learning the blitter for.

Two things to keep in mind:

- **The static RAM only.** Bank 5 and page `$0E`, the first half of bank 7, are the FPGA's own block RAM when the MMU maps them, and the ULA reads its

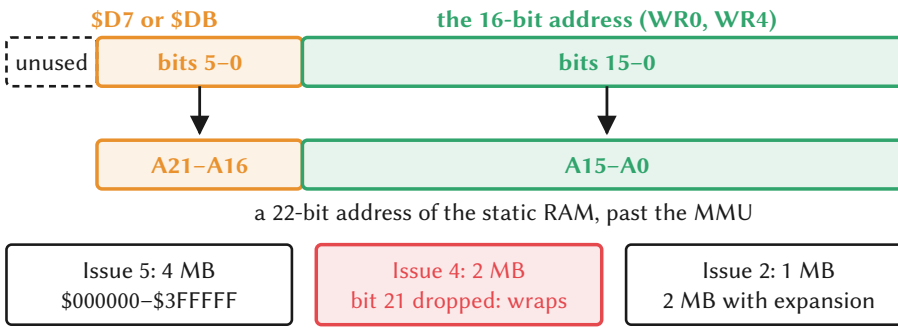


Figure 5.5: The zxnDDMA's physical address, and how much of it each board keeps

screen from there. A physical write to their addresses, `$054000` and `$05C000`, lands in the static RAM behind them, the twins of Chapter 3, which the ULA never shows. Copy to the ULA screen through the MMU, the old way. Page `$0F`, the second half of bank 7, is ordinary static RAM, so a physical write there does show through the MMU.

- **The board's size.** The Issue 5 board's 4 MB reach `$3FFFFF`. The Issue 4 board has 2 MB and drops address bit 21, so `$200000` and up wrap onto the bottom of the memory, which on a Next holds the ROMs and the DivMMC's memory. The Issue 2 board has 1 MB, or 2 MB with its expansion. Read the **Board ID register**, **NextReg `$0F`**, before you touch anything above 2 MB. The MMU never reaches past 2 MB on any board, so on the Issue 5 board physical mode is a program's way into its upper half, the attic of "The Issue 5 board's attic" (page 41).

Rectangles

With bit 1 set, the block length is not used. The DMA moves a rectangle instead, row length × number of rows bytes (Figure 5.6). Inside a row each port steps as WR1 and WR2 say; when a row ends, each port starts its next row at its previous row's start plus its **stride**, the distance from the start of one row to the start of the next, not the gap after it. In physical mode a stride is added to the full 22-bit start, so rows can cross 64K too. A row length or a number of rows of 0 still moves one byte. A 32 × 32 picture out of a sheet 256 bytes wide is row length 32, 32 rows and a source stride of 256.

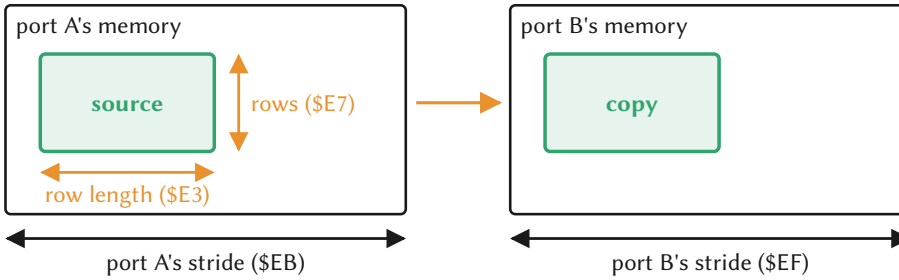


Figure 5.6: A rectangle copy: the row length and the number of rows give its size, and each port's stride its row spacing

Transparency and fill

With bit 2 set, a source byte equal to the `$F3` colour is not written, but both addresses move on, so the background around a character stays as it was: the sprites' see-through colour, now for anything you copy. It saves no time, though: the DMA still spends the write's cycle on a see-through byte and only keeps it from landing.

With bit 3 set, the DMA does no read at all, and every byte gets the `$F7` value; without its read cycle a byte takes half the time of a copy from a fixed address, which makes fill the quickest way to clear a screen.

Layer 2 positions

With bit 4 set, and bit 0 with it, three more bytes follow the flags: X's low byte, X's high byte and Y. The DMA works out port B's start address and its extended byte from them, using the Layer 2 bank in the **Layer 2 Active RAM Bank register**, **NextReg $12**, or with bit 5 the shadow bank in NextReg `$13`, and the screen mode in NextReg `$70` bits 5–4:

- **256 × 192**, stored row by row: the bank's start + Y × 256 + X. Only X's low byte counts.
- **320 × 256** and **640 × 256**, stored column by column: the bank's start + X × 256 + Y, with a 9-bit X (in the 640 mode, X is a byte column, 0–319).

Port B's stride becomes 256 and stays so until `$EF` or `$C3`: one row down in the 256 × 192 mode, one column to the right in the column modes. Send this `$DF` after WR4, or use a WR4 without address bytes, `$A1` or `$C1`: a WR4 that carries an address overwrites the one the DMA worked out. Draw with bit 5 and you paint the screen that is not shown, then swap the two, and the player never sees a half-drawn picture.

To draw many copies of one shape, there is no need to send the whole setup again. Between copies send WR0 with port A's address only (`$1D`, low byte, high byte), then either the `$DF` with its X and Y for a Layer 2 target or WR4 with port B's address only (`$AD`, low byte, high byte), then `$CF` and `$87`; add `$D7` and its byte when the source moves to another 64K.

Resets and lifetimes

The new settings stay until `$C3`, which clears them all: the flags, both extended bytes, the rectangle, the strides and both colours. `$CF` copies the extended bytes into the working addresses along with the 16-bit ones. Physical mode stays on until the next `$DF` or `$C3`, so a later ordinary transfer must turn it off; my rule is to start every zxnDDMA setup with `$C3`, always.

Let's copy 16 bytes from RAM page 40 to RAM page 41 without paging either of them in. Page 40 starts at `$40000` + 40 × `$2000` = `$090000`, page 41 at `$092000`:

```
        ld hl,Setup
        ld b,Size
        ld c,$6b            ; zxnDMA port
        otir
        ret
Setup:  .defb $c3           ; reset: zxnDDMA settings off
```

```
        .defb $d7,$09        ; port A bits 23-16
        .defb $db,$09        ; port B bits 23-16
        .defb $df,$01        ; flags: physical
        .defb $7d,$00,$00    ; WR0: A to B, port A $0000
        .defb $10,$00        ;   length 16
        .defb $14,$10        ; WR1, WR2: memory, up
        .defb $ad,$00,$20    ; WR4: continuous, B $2000
        .defb $82,$cf,$87    ; stop at end, load, enable
Size:   .equ $-Setup
```

How it works:

1. `$C3` clears every zxnDDMA setting left by an earlier transfer.
2. `$D7` and `$DB` give both ports the extended byte `$09`; with the 16-bit addresses `$0000` and `$2000` below them, they make `$090000` and `$092000`.
3. `$DF $01` turns physical mode on, so the MMU's slots play no part.
4. The rest is an ordinary zxnDMA setup: WR0, WR1, WR2 and WR4, then `$82`, `$CF` to load and `$87` to start.

The older core and the trap

Now the one trap, and I warn you about it because it bites quietly. On port `$0B`, and on the older core 3.02 and anything before it, the DMA does not know the blitter commands: it drops the command byte and then reads the bytes after it as ordinary base bytes. In the listing above, an old core would take `$09` as a WR0 that announces port A's low byte, and the setup would go astray from there. So a program that may meet an older core, or a DMA left in Z80 DMA mode, finds out first whether the blitter is there. A safe test sends the zxnDDMA bytes straight after `$C3`, keeps every one of their data bytes below `$80`, follows them with eight `$83`s, which finish any register write an old DMA was left in the middle of, and then copies one byte whose two possible destinations, the 16-bit one an old DMA uses and the physical one, are both harmless. Whichever of them changed tells you which DMA you have. Reading the core version from NextRegs `$01` and `$0E` works too, but the test also catches a DMA left in Z80 DMA mode.

How much faster

The release's own example draws 32 balls of 16 × 16 bytes on Layer 2, five ways, and times each at 28 MHz (Figure 5.7). The blitter draws a ball with one transfer, a rectangle with a see-through colour, where the plain DMA commands need one transfer per column and lose most of their speed to sending sixteen setups; so the blitter is more than three times as fast as anything else. The figure also shows what the older core 3.02 would take: the same program with its 28 MHz wait state is slower by 1 to 3 milliseconds a way, and it has no blitter at all.

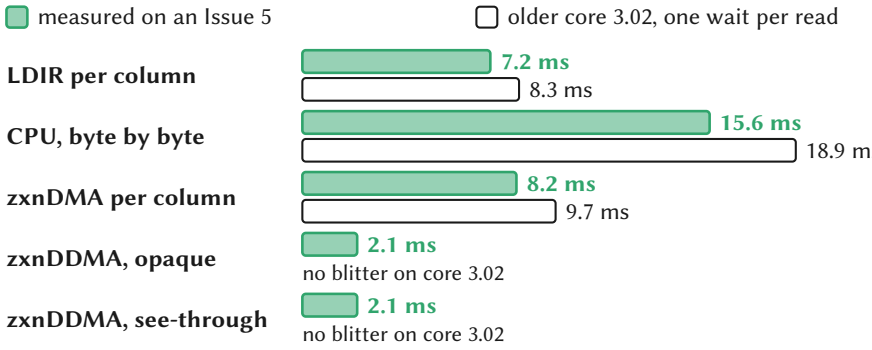


Figure 5.7: Thirty-two balls drawn five ways at 28 MHz: measured on an Issue 5 board with the blitter core, and the same program with the older core 3.02's wait state

In short. The DMA borrows the CPU's bus and moves a byte in six T-states, or four with two-clock cycles, where `LDIR` needs 21. Use port `$6B`, write each register group with its followers, and finish every setup with `$CF` and `$87`, because the addresses you write are not used until a load. Continuous mode keeps the bus and suits copies; burst mode with a prescaler gives it back between bytes and suits anything paced. An auto-restarting DMA in continuous mode never gives the bus back. The blitter's commands, on port `$6B` only, add 22-bit physical addresses past the MMU, rectangles with a stride per port, a see-through colour, fill and Layer 2 positions. Start every blitter setup with `$C3`; the older core 3.02 has no blitter and turns those bytes into register writes.

CHAPTER 6

CTC: Four Timers in a Ring

Switch a Next from 3.5 MHz to 28 MHz and every delay loop in your program runs eight times faster. The music player that counted T-states between two notes now plays like a chipmunk. Yet some things on the Next keep their pace whatever the CPU does. Who is keeping time for them?

The CPU is not a clock you can trust: its speed is a setting, contention stretches it, the DMA stops it. The **CTC**, the counter/timer circuit, counts the machine's master clock instead. Reach for it whenever something must happen at a rate: a sample clock, a game tick, a timeout, a stopwatch.

You should know Chapter 4 first: it explains how a CTC channel's interrupt reaches the CPU. Here we look at what makes the channel ring.

- What is inside one of the CTC's four channels
- How the control word makes a channel count time or events
- How the ring joins the channels into exact rates and a 16-bit stopwatch
- How interrupts and resets work

A Zilog chip in the FPGA

The Next's CTC is a copy of the Z80 CTC, the counter/timer chip Zilog made as a companion for the Z80, and programs written for it speak the same language here. I call it **the metronome**: it beats a steady rate, and it does not care how fast the musician plays.

- **Channel 0** is at port `$183B`, **channel 1** at `$193B`, **channel 2** at `$1A3B` and **channel 3** at `$1B3B`. Ports `$1C3B` to `$1F3B` have no hardware behind them: they read `$00` and ignore writes.
- **Writing** a channel's port sends it a control word or a time constant; **reading** it gives the channel's current count.
- Bit 3 of NextReg `$85` switches the ports off. They are on after a reset, and a channel that runs keeps running while its port is off.

A Zilog CTC has pins for an outside clock or trigger on each channel. The Next's channels have no outside pins: their trigger inputs are wired to each other, in a ring.

One channel inside

Each channel is a small machine of three parts (Figure 6.1).

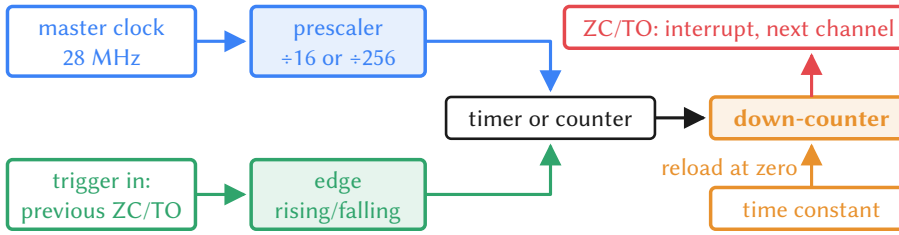


Figure 6.1: One channel

- The **prescaler** is an 8-bit counter on the 28 MHz master clock. It gives a pulse every **16** clocks or, when bit 5 of the control word asks for it, every **256**.
- The **down-counter** holds the channel's 8-bit count. In timer mode each prescaler pulse counts it down by one; in counter mode each edge on the trigger input does.
- The **time constant register** holds the value the count starts from. Where the count would reach zero, the channel sends one pulse on its output, **ZC/TO** ("zero count / time out"), and reloads the count from the time constant, with no help from your program. A time constant of 0 means **256**, in both modes: the first tick takes the count from 0 to 255.

So a timer channel's period is the prescaler times the time constant, in master clocks:

- **Prescaler 16**: a count every 16 clocks, 0.571 µs, exactly **2 T-states** at 3.5 MHz. Periods from 16 clocks to 4,096 clocks, 146 µs.
- **Prescaler 256**: a count every 256 clocks, 9.14 µs, **32 T-states** at 3.5 MHz. Periods from 256 clocks to 65,536 clocks, 2.34 ms.

Let's watch a channel with prescaler 16 and time constant 3 (Figure 6.2).

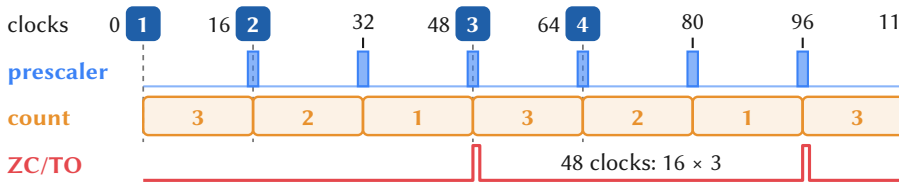


Figure 6.2: Prescaler 16, time constant 3, to scale

1. The channel starts with the count at 3 and the prescaler at 0.

2. Every 16 clocks the prescaler pulses, and the count goes to 2, then 1.
3. At clock 48 the count would become 0. The channel reloads 3 instead, and ZC/TO is high for one clock, 36 ns.
4. The prescaler never stopped, so the next pulse comes at clock 64 and the next zero count at clock 96.

Notice the first step, because the Zilog datasheet leaves it vague: while a channel is not running, its prescaler is held at 0 and its count at the time constant. So a timer you start at a known moment beats at known moments.

And this is the answer to the opening puzzle. The CPU's clock is made from the same master clock, so a channel that beats 17,500 times a second at 3.5 MHz beats 17,500 times a second at 28 MHz.

There is one exception. The **VGA modes** of NextReg `$11` run the master clock at up to 33 MHz (Chapter 10), and the metronome speeds up with it. So does the CPU: one count of prescaler 16 is still exactly 2 T-states at 3.5 MHz, and only the rate in seconds changes, by up to 18%.

The control word

A channel is programmed with a **control word**, a byte with bit 0 set, and often a **time constant** after it:

```
control word:  I M P E T C R 1
  I = 1: interrupt on every zero count
  M = mode: 0 timer, 1 counter
  P = prescaler: 0 divides by 16, 1 by 256
  E = trigger edge: 0 falling, 1 rising
  T = 1: a timer waits for a trigger to start
  C = 1: a time constant follows
  R = 1: software reset
```

A byte with bit 0 clear, when no time constant is expected, is an interrupt vector on the Zilog chip; the Next throws it away. The byte after a control word with C set is always the time constant, whatever its bit 0.

A channel is always in one of four states:

1. **Reset.** The channel holds still with its count at the time constant, which is 0. A control word without **C** is stored, but only one with C wakes the channel: the first control word must announce a time constant.
2. **Waiting for a time constant.** The next byte is the constant.
3. **Waiting for a trigger.** A timer with **T** set waits here for an edge on its trigger input; any other channel passes straight through.
4. **Running.** The channel counts, as long as nothing stops it.

The **software reset**, **R**, stops a channel from any state: with C clear it goes back to state 1, with C set to state 2. Without R, a control word does not stop a running

channel: the prescaler, the mode and the interrupt enable change at once, and the count goes on. With C set, the next byte becomes the new time constant, used from the next zero count, so a sample player can change its rate between two samples without a click.

Here is channel 0 set up as a timer of prescaler 16 and time constant 100, a zero count every 1,600 clocks, 17,500 times a second, with its interrupt on:

```
        ld bc,$183b             ; channel 0
        ld a,%10000101          ; interrupt, timer, /16,
        out (c),a               ; constant follows
        ld a,100
        out (c),a               ; starts on the next clock
```

If you inherit a channel in an unknown state, write `$03`, a software reset without C, twice. If the channel happened to expect a time constant, the first byte is taken as one; the second then leaves it in state 1.



KLIVE IDE Every byte the channels get

An **I/O write** breakpoint with a port mask watches all four channels at once: `bp-set $183B -o -m $FCFF` stops at a write to `$183B`, `$193B`, `$1A3B` or `$1B3B`. Step from stop to stop, and `IWV` in the Z80 CPU panel shows each control word and time constant as the program sends it.

Timer and counter

In **timer mode**, bit 6 clear, a channel counts the prescaler's pulses: it measures time. In **counter mode**, bit 6 set, it counts edges on its trigger input instead, which on the Next means the previous channel's zero counts.

A timer with bit 3, **T**, set does not start when it gets its time constant: it waits for the first edge on its trigger input, and then starts, with the prescaler at 0. So one channel can start another at an exact moment.

Bit 4, **E**, chooses the edge. The ZC/TO pulse is one master clock long, so the choice only moves the moment by 36 ns. But **writing a control word that changes E counts as an edge**: a counter counts one down, and a timer waiting for its trigger starts. Choose the edge when you set a channel up, and do not touch it afterwards.

The ring

Each channel counts the zero counts of the one before it, and channel 0 counts channel 3's (Figure 6.3).

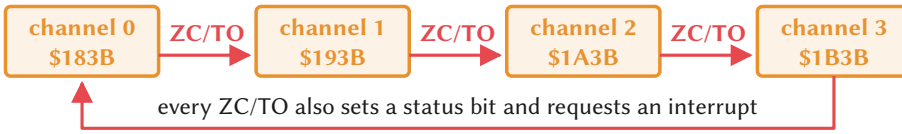


Figure 6.3: The four channels in a ring

A chain of channels is a chain of dividers, and it reaches rates no single channel can. Here is my favourite example, an exact 50 Hz tick that does not come from the display: the prescaler divides 28 MHz by 16, channel 0 by 140 and channel 1 by 250, and 16 × 140 × 250 = 560,000 clocks, 20 ms.

```
        ld bc,$183b             ; channel 0: timer, /16
        ld a,%00000101
        out (c),a
        ld a,140                ; 2,240 clocks: 12.5 kHz
        out (c),a
        inc b                   ; $193B: channel 1
        ld a,%11000101          ; interrupt, counter
        out (c),a
        ld a,250                ; 560,000 clocks: 50 Hz
        out (c),a
```

Only channel 1 interrupts, every 20 ms. The display's frame rate is close to 50 Hz but never exactly 50 Hz, so a game whose logic runs on this tick runs at the same speed on every display.

Four counters and no timer count nothing. Start every chain with a timer; a timer ignores its trigger input once it runs, so the ring is cut there.

A stopwatch

Reading a channel gives its count at that moment.

For longer code, chain two channels into a **16-bit counter**: channel 0 as a timer with prescaler 16 and time constant 0, the low byte, and channel 1 in counter mode with time constant 0, the high byte. That counts 65,536 times 16 clocks, 37.4 ms, with a resolution of 2 T-states.

The low byte can wrap between the two `IN` instructions. So read the high byte, the low byte, and the high byte again; if the two high bytes differ, read again. Read again too when the low byte is 0: channel 0 has just reached zero and channel 1 counts, and a reading there cannot tell which side of the count it is on:

```
; channel 0: timer, /16, constant 0; channel 1: counter, 0
; returns the 16-bit count in HL; changes A, BC
Read:   ld bc,$193b
        in h,(c)                ; high byte: channel 1
        dec b                   ; $183B: channel 0
        in l,(c)                ; low byte
        inc b
        in a,(c)                ; high byte again
        cp h
        jr nz,Read              ; it counted: read again
        inc l
        dec l
        jr z,Read               ; low byte 0: read again
        ret
```

Call `Read` before and after the code you measure, subtract the second count from the first, and double the difference for T-states at 3.5 MHz. Measure an empty block once and take that off, because the difference includes part of the reading itself.

▶ TRY IT Stopwatch

Part I, key 7. The demo chains channels 0 and 1 as described, reads them with this section's routine, and times three pieces of code: 100 `NOP`s and a 100-byte `LDIR` at 3.5 MHz, then the same `NOP`s at 28 MHz. The code is in `examples/code/demos/1-machine/stopwatch.kz80.asm`. It times the code at `IX` like this:

```
SwTime:
    di
    call SwRead
    push hl
    call SwCallIx
    call SwRead
    pop de
    ex de,hl                ; HL: the first count, DE: the second
    or a
    sbc hl,de               ; the channels count down
    add hl,hl               ; a count: 2 T-states at 3.5 MHz
    ei
    ret
```

It measures an empty block first and takes that off every result. Next to each time it prints what the instruction tables give: 400 T-states for the `NOP`s, 2,125 for the `LDIR` and the three `LD`s that set it up. The measured times land on them, give or take the 2 T-states of one count. The last row runs 100 `NOP`s at 28 MHz: 50 T-states of 3.5 MHz on core 3.03, an eighth of 400. On the older core 3.02 it shows a surprise, 62, because each one takes five clocks, its opcode fetch paying the 28 MHz wait state of Chapter 2.



KLIVE IDE Checking the stopwatch

Put a breakpoint before and after the code the stopwatch times, and run from one to the other at 3.5 MHz: the Z80 CPU panel's `TSP` value is the code's cost in T-states, which the stopwatch's doubled count should match. At a higher speed, `TSP` counts the CPU's own T-states, so divide by 2, 4 or 8 to compare.

Interrupts

Every zero count of a channel is an interrupt request (Chapter 4). From the CTC's side:

- **The enable is bit 7 of the control word**, and also bit *n* of NextReg `$C5` for channel *n*. They are the same flip-flop, so a control word without bit 7 switches the interrupt off.
- **The status is set on every zero count**, enabled or not, in bit *n* of NextReg `$C9`, so a program can leave the interrupt off and poll the bit.
- **In hardware IM2 mode** channels 0 to 3 are sources 3 to 6 of the chain, with vector offsets `$06` to `$0C`.

In the joystick's I/O mode, NextReg `$0B` with bit 7 set, the joystick port stops reading a joystick and becomes a small connector for your own hardware, with pin 7 as its output. Bits 5–4 choose what drives pin 7: 00 the value of bit 0, 01 channel 3 of the CTC, 10 and 11 a UART on the left or right port. With 01 and bit 0 set, pin 7 toggles on every zero count of channel 3, a square wave at half its rate that can clock a chip wired to the port; with bit 0 clear, the clock stops once it is high.

Resets and core versions

A reset, hard or soft, gives every channel control word 0, time constant 0, interrupt off and a count of 0.

In 3.02.01 the count holds at 0 for one clock before it reloads; from 3.02.04 on it reloads at once, as described here. And from 3.02.04 on, a control word with C while a timer waits for its trigger makes the channel wait for a new time constant; in 3.02.01 a trigger could start it with the old one.



KLIVE IDE Two cores side by side

Klive's **KS2, core 3.02.01** model keeps the older behaviour: the hold at 0 and the old time constant after a trigger. The 3.02.04 model and the two 3.03.03 models, built on it, behave as this chapter describes. Pick the model in **Machine › Select Machine…** and run the same program on both to see the difference.

In short. Each of the four channels divides the 28 MHz master clock by a prescaler of 16 or 256 and a time constant of 1 to 256, so its rate does not depend on the CPU's speed. The first control word must announce a time constant, and a new constant takes effect at the next zero count. The channels are wired in a ring, so a chain that starts with a timer reaches exact rates such as 50 Hz. Two chained channels, read with care, make a 16-bit stopwatch.

CHAPTER 7

Input: One Matrix, Many Hands

Plug a PC keyboard into a Next, press the up arrow, and a program written in 1983 sees CAPS SHIFT and 7, as if your fingers had found both keys on a rubber Spectrum. Plug in a joystick, set it to Sinclair mode, push it left, and the same old program sees the key 6. So who is doing the translating, and how does the program never notice?

This chapter builds on Chapter 3, for how ports are decoded and how a NextReg switches a group of them off. Bit 6 of port `$FE`, the tape input, shares its byte with the keyboard; Chapter 21 covers it.

- How port `$FE`'s half-rows and the Next's extra columns work
- How the scanner keeps a copy of the keyboard for the CPU
- How PS/2 keyboards and joysticks press keys in that scan
- How the extra keys, the joystick modes and the Kempston mouse work

The Spectrum's keyboard

The Spectrum's 40 keys sit in a **matrix**: eight rows of five keys, each key a switch at the crossing of one row wire and one column wire. To read it, the CPU executes an `IN` from port `$FE`, any port with address line A0 low. The upper half of the address chooses the rows: every one of A8 to A15 that is 0 selects its **half-row**, and bits 0 to 4 of the byte read are the five columns. A pressed key reads **0**; bits 5 and 7 always read 1, and bit 6 is the tape input.

With `$00` in the upper byte, `IN A,($FE)` reads all eight rows at once. That is how a program checks "is any key pressed?" in one instruction.

The Next keeps every bit of this. But its own keyboard has **seven** columns, not five (Figure 7.1).

	bit 0	bit 1	bit 2	bit 3	bit 4	col 5	col 6
A8	CAPS	Z	X	C	V	EXTEND	UP
A9	A	S	D	F	G	CAPS LK	GRAPH
A10	Q	W	E	R	T	TRUE VID	INV VID
A11	1	2	3	4	5	BREAK	EDIT
A12	0	9	8	7	6	;	"
A13	P	O	I	U	Y	,	.
A14	ENTER	L	K	J	H	DELETE	RIGHT
A15	SPACE	SYM	M	N	B	LEFT	DOWN

Figure 7.1: The Next's keyboard matrix (orange: the extra keys)

The two extra columns hold the sixteen keys a Spectrum+ or a 128 had as separate keys but the 48K made with shifts: the arrows, DELETE, EDIT, BREAK, the punctuation that needed SYMBOL SHIFT, and a few more. A Spectrum program reads only bits 0 to 4, so it cannot see columns 5 and 6.



KLIVE IDE Pressing the keys

On your keyboard, either Shift is CAPS SHIFT and the right Alt is SYMBOL SHIFT. The virtual keyboard, **View › Virtual Keyboard**, shows the Next's own: holding the mouse button down holds the key, the right button presses CAPS SHIFT with it, and a red legend presses SYMBOL SHIFT.

The clerk

On a 48K, the CPU's address lines drive the keyboard's rows directly. The Next's FPGA has its own scanner instead, which I call **the clerk**: it walks the membrane one row at a time, writes down what it finds, and hands the CPU its notes.

1. The clerk pulls one row wire low and reads the seven column wires, moving to the next row **every 18.3 µs**.
2. After the eighth row it takes one more step, in which it copies its notes into the **matrix copy**, all at once. A whole scan is **164.6 µs**: about 120 scans in a 50 Hz frame.
3. A read of port `$FE` never touches the membrane. The row lines of the `IN` select rows of the matrix copy, the selected rows are combined, and the result is on the bus at once.

So a key pressed now shows up in a read at most one scan later, and a program that reads the eight rows one after another can never catch half of an old scan and half of a new one in the same row.

One matrix, many hands

The clerk's column wires do not come only from the membrane. Two more sources press keys **in the clerk's scan**, at the very moment he reads a row:

- The **PS/2 keyboard**. Its keys are translated into membrane keys by a table, and while the clerk reads row 3, the PS/2 side pulls low the columns of every row-3 key that a PS/2 key is holding.
- The **joysticks** in the key modes (Sinclair, Cursor, user-defined). Pushing a joystick left in Sinclair mode pulls the column of key 6 while the clerk reads row 4.

A key is down when **any** of them presses it (Figure 7.2).

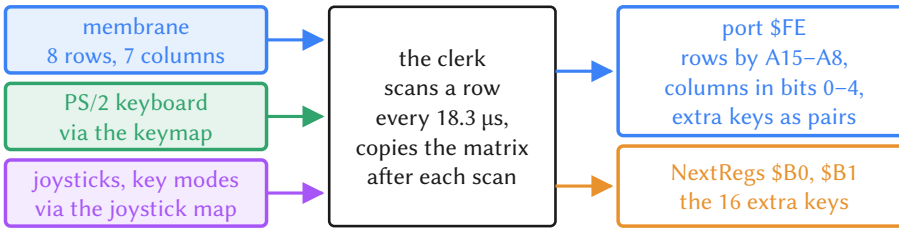


Figure 7.2: One matrix, many hands

Everything after the clerk works the same for every hand, so a program written before the Next existed works with all of them, without knowing any of them. If you ask me, this is how you add hardware to an old machine: by making the new hardware pretend to be the old one, as early as possible.

The sixteen extra keys

An extra key on its own would be invisible to a Spectrum program, so each one also **enters two matrix keys**, the pair a 48K owner would have pressed:

- **CAPS SHIFT** and a key: EXTEND = CAPS + SYMBOL SHIFT, EDIT = CAPS + 1, CAPS LOCK = CAPS + 2, TRUE VIDEO = CAPS + 3, INV VIDEO = CAPS + 4, LEFT = CAPS + 5, DOWN = CAPS + 6, UP = CAPS + 7, RIGHT = CAPS + 8, GRAPH = CAPS + 9, DELETE = CAPS + 0, BREAK = CAPS + SPACE.
- **SYMBOL SHIFT** and a key: `;` = SYM + O, `"` = SYM + P, `,` = SYM + N, `.` = SYM + M.

A new program can do better. The **Extended Keys 0 and 1 registers, NextRegs `$B0` and `$B1`**, report the sixteen keys themselves, one bit each, 1 for pressed:

```
$B0:  ;  "  ,  .  UP DOWN LEFT RIGHT        (bit 7 ... bit 0)
$B1:  DELETE EDIT BREAK INV TRUE GRAPH CAPSLOCK EXTEND
```

And bit 4 of the **ULA Control register**, **NextReg $68**, stops the extra keys from entering their pairs. Set it, and an arrow is just an arrow. The real CAPS SHIFT and 7 on the membrane are not affected:

```
; The arrows from NextReg $B0, without CAPS SHIFT
Arrows:
        ld bc,$243b
        ld a,$68                ; ULA Control
        out (c),a
        inc b
        in a,(c)
        or %00010000            ; bit 4: no matrix entries
        out (c),a
        dec b
        ld a,$b0                ; Extended Keys 0
        out (c),a
        inc b
        in a,(c)
        and %00001111           ; UP DOWN LEFT RIGHT
        ret
```

A soft reset clears bit 4 of `$68`, so set it again after a reset if you rely on it.

Why CAPS SHIFT comes early

A program that reads the rows one at a time may read row 0 (CAPS SHIFT) in one copy and row 4 (the 7) in the next. A read straddling them could see the 7 without CAPS SHIFT. So for every extra key, the shift enters the matrix **one scan before** the other key and leaves **two scans after** it (Figure 7.3).

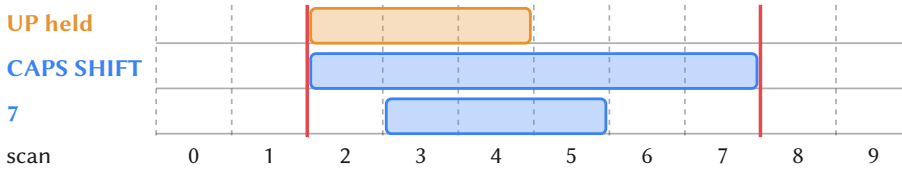


Figure 7.3: UP held in scans 2 to 4, as the CPU sees it

So whatever order a program reads the rows in, it cannot see the 7 without its CAPS SHIFT. The price is one scan of delay on the 7, 165 µs.

The PS/2 keyboard

A PS/2 keyboard sends a **scan code** when a key goes down and another when it comes up. The Next turns those codes into membrane keys with a **keymap**, a table of 512 entries: one for each scan code, plain and with the `$E0` prefix that marks a PC's extended keys. Each entry names a membrane row and column, and two flags: add CAPS SHIFT, add SYMBOL SHIFT. Backspace is DELETE, for example, and Esc is BREAK.

Three details make it work like a keyboard:

- **The shifts are counted, not switched.** Hold `/` (SYM + V) and `,` (SYM + N) together, release `/`, and SYMBOL SHIFT stays down for the `,`.
- **Repeats are ignored.** A membrane key does not repeat, so a code identical to the last one is thrown away.
- **Pause clears everything.** If a key ever seems stuck, the Pause/Break key resets the PS/2 side's state.

Some entries are **function keys**, the machine's own controls. In the US table F3 switches between 50 and 60 Hz, F4 is a soft reset, F8 changes the CPU speed, and F11 and F12 switch the expansion bus on and off; F9 and F10 stand in for the NMI and DivMMC buttons. NextReg `$06` can disable the speed, 50/60 Hz and expansion bus hotkeys.

The table can be rewritten for another national layout. NextReg `$28` bit 7 = 0 selects the keymap, bit 0 is the top bit of the address (the `$E0` flag) and **NextReg `$29`** the other eight; each write of **NextReg `$2B`** stores an entry and steps the address. Bit 2 of NextReg `$06` says which pins of the PS/2 connector carry the keyboard and which the mouse; it can only be changed in config mode, at boot.

Joysticks as keys

The Next has two joystick connectors, and the **Peripheral 1 Setting register**, **NextReg `$05`**, gives each a three-bit **mode**: joystick 1 in bits 7–6 with bit 3 on top, joystick 2 in bits 5–4 with bit 1 on top. The other bits are the 50/60 Hz and scandoubler settings, so always read `$05` and change only your bits. The modes are:

- `000` **Sinclair 2**: RIGHT 2, LEFT 1, DOWN 3, UP 4, fire 5.
- `011` **Sinclair 1**: RIGHT 7, LEFT 6, DOWN 8, UP 9, fire 0.
- `010` **Cursor**: RIGHT 8, LEFT 5, DOWN 6, UP 7, fire 0, without CAPS SHIFT.
- `111` **user-defined keys**: each of the twelve buttons presses a key you choose.
- `001` and `100` **Kempston 1** and **Kempston 2**, and `101` and `110` **MD 1** and **MD 2**: the port modes of the next section.

The first four are pure key presses in the clerk's scan, from a table I call the **joystick map**. A user-defined joystick is programmed through the keymap registers with NextReg `$28` bit 7 = 1, NextReg `$29` = 0 for the left connector or 16 for the right, and

twelve writes of NextReg `$2B`, one per button in the order R, L, D, U, B, C, A, START, Y, Z, X, MODE. Each byte is a membrane key, row in bits 5–3 and column in bits 2–0; column 7 means no key. Here is the left joystick as the classic Q, A, O, P and SPACE:

```
; Left joystick as Q A O P and SPACE (mode 111)
SetQaop:
        nextreg $28,$80         ; the key joystick's map
        nextreg $29,$00         ; left connector, button 0
        ld hl,Qaop
        ld b,12
QNext:  ld a,(hl)
        nextreg $2b,a           ; store, step to next button
        inc hl
        djnz QNext
        ld bc,$243b             ; read NextReg $05
        ld a,$05
        out (c),a
        inc b                   ; $253B
        in a,(c)
        or %11001000            ; joystick 1: mode 111
        nextreg $05,a
        ret
Qaop:   .defb $28,$29,$08,$10   ; R=P L=O D=A U=Q
        .defb $38,$07,$07,$07   ; B=SPACE; C A START: none
        .defb $07,$07,$07,$07   ; Y Z X MODE: none
```

P is row 5, column 0, so its entry is `%101000`, `$28`.

The map survives a soft reset. In the Kempston and MD modes, a button that the port does not report presses the user-defined key of that button.



KLIVE IDE The joystick modes

Each joystick's **Set machine mode** submenu, under **Machine › Input › Joystick**, picks Sinclair 1 or 2, Cursor, Kempston 1 or 2, MD 1 or 2, or the user-defined map. It writes NextReg `$05`, so NextZXOS may set the mode again when it starts.

Kempston ports and Mega Drive pads

The port modes do not press keys; a program reads the joystick as a byte, **1 for pressed**:

- **Kempston 1**, mode `001`, reads on port `$1F`; **Kempston 2**, mode `100`, on port `$37`. Bits 0–5 are RIGHT, LEFT, DOWN, UP, fire (B) and second fire (C); bits 6–7 read 0.

- **MD 1**, mode `101`, and **MD 2**, mode `110`, use the same ports and add **A in bit 6 and START in bit 7**. A six-button pad's X, Y, Z and MODE are in the **Extended MD Pad Buttons register, NextReg `$B2`**: the left pad in bits 3–0 (X, Z, Y, MODE), the right pad in bits 7–4.

Both connectors can be set to the same port, and their bits are then OR-ed. A port that no connector uses reads `$FF`, which a careful program can take as "no Kempston joystick here". Bits 6 and 7 of NextReg `$82` switch ports `$1F` and `$37` off. After the core is loaded, joystick 1 is Kempston 1 and joystick 2 is Sinclair 2.



KLIVE IDE Driving a joystick

Each joystick is driven by your keyboard, a gamepad, or nothing, and **Configure bindings...** in its submenu assigns a key to each pin. A key bound to a joystick no longer reaches the Next's keyboard: bind joystick 2 to W, A, S and D, and a program that reads those keys sees nothing.

The interviewer

Behind the ports sits a part of the FPGA I call **the interviewer**, because it questions every joystick in the same way. The board reads both connectors through **one set of input pins**, and a switch flips between them at every step. And a Mega Drive pad has more buttons than wires: it shows a different set of buttons depending on its **select** line, pin 7, which the Next drives.

So the interviewer runs a fixed script, sixteen steps of 4.57 µs (Figure 7.4).

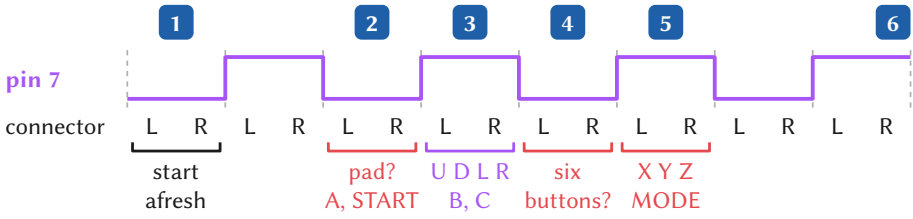


Figure 7.4: The joystick reader's script (red: only a pad answers)

1. The first low pulse clears the answers and starts afresh.
2. The second low pulse: if LEFT and RIGHT both read pressed, something no joystick can do, the connector holds a pad, and pins 6 and 9 are A and START.
3. The second high pulse: pins 1, 2, 3, 4, 6 and 9 are UP, DOWN, LEFT, RIGHT, B and C.
4. The third low pulse: if UP and DOWN both read pressed, the pad has six buttons.

5. The third high pulse, only for a six-button pad: X, Y, Z and MODE.
6. A rest of 496 steps, **2.27 ms**, so that a six-button pad knows the next pulse starts a new count.

An ordinary Atari-style joystick ignores pin 7 and never "answers" the pad questions. Every joystick is read 427 times a second.

The **Joystick I/O Mode register**, **NextReg `$0B`**, can take pin 7 for other uses. The interviewer then stops its script, presses no keys, and only reports the six raw pins on the Kempston ports.

The Kempston mouse

The mouse is a PS/2 mouse, and the Next presents it the way the Kempston mouse interface did: as counters. Every packet the mouse sends, up to 80 a second, **adds its movement** to three counters:

- **Port `$FBDF`**: X, an 8-bit count that goes up as the mouse moves right and wraps from 255 to 0.
- **Port `$FFDF`**: Y, which goes **up** as the mouse moves **up**, the opposite of screen rows.
- **Port `$FADF`**: the buttons and the wheel. Bit 0 is the right button, bit 1 the left, bit 2 the middle, and a pressed button reads **0**, unlike the joysticks; bits 7–4 are a 4-bit wheel count that wraps.

A counter says how far the mouse has moved since you last looked, and an 8-bit subtraction gives the right signed answer even across the wrap, as long as the mouse moves less than 128 counts between two reads:

```
; A = the mouse's X movement since the last call
MouseDx:
        ld bc,$fbdf             ; the X counter
        in a,(c)
        ld hl,LastX
        ld c,(hl)               ; the count read last time
        ld (hl),a               ; keep this one for next time
        sub c                   ; the difference, wrapped
        ret
LastX:  .defb 0
```

The **Peripheral 5 Setting register**, **NextReg `$0A`**, adjusts the mouse. Bit 3 swaps the left and right buttons. Bits 1–0 scale the movement: `00` doubles it, `01`, the default, keeps it, `10` halves it and `11` quarters it.



KLIVE IDE Capturing the mouse

Turn on **Machine › Input › Mouse › Capture the mouse**, then click the toolbar's mouse button or press Ctrl+M (Cmd+M on a Mac): your mouse now moves the Next's, and Esc lets it go. Klive's own **Sensitivity** setting is separate from the scaling a program sets in NextReg `$0A`.

TRY IT Input Monitor

Part I, key 8. The demo shows every input in this chapter, once a frame: the eight half-rows of port `$FE`, the sixteen extra keys of NextRegs `$B0` and `$B1`, Kempston joystick 1 on port `$1F`, and the mouse's two counters and its buttons. A key or button that is down is lit green. ENTER flips bit 4 of NextReg `$68`, and BREAK returns to the menu. The code is in `examples/code/demos/1-machine/input-monitor.kz80.asm`; this is its keyboard scan:

```
    ld bc,$FEFE
    ld hl,$5800 + 2 * 32 + 7    ; row 2, column 7: the first key cell
ImRow:
    in a,(c)
    cpl                     ; a pressed key reads 0: make it 1
    and $1F
    push bc
    push hl
    ld b,5
    ld d,3                  ; three columns a key
    call ImLight
    pop hl
    pop bc
    ld de,32
    add hl,de               ; the next row on the screen
    rlc b                   ; the next half-row
    jr c,ImRow
```

Press an arrow: it lights in NextReg `$B0`, and also lights CAPS SHIFT and a number key in the matrix, the pair it enters for old programs. Then press ENTER, which flips bit 4 of NextReg `$68`, and press the arrow again: now only `$B0` lights. In Klive, the PC's arrow keys, Backspace and Home, the comma, full stop, semicolon and quote, and the extra keys of the on-screen Next keyboard are the Next's extra keys.

The two buttons on the side

- **Reset**: a short press, under about 0.94 s, is a **soft reset**; a long one is a **hard reset**, which reloads the core.

- **NMI**: held on its own and released within about 0.94 s, it is the Multiface's NMI, when bit 3 of NextReg `$06` allows it. Held while you press a number key, it turns the number keys into **function keys**: 1 to 0 are F1 to F10.

Resets

A **soft reset** clears NextReg `$68` bit 4 and sets NextReg `$0B` back to `$01`, I/O mode off. It keeps the joystick modes in NextReg `$05`, the joystick map, the PS/2 keymap and the mouse's scaling. A **hard reset** loads the core again, which restores the joystick map's presets, the US keymap, and the joystick modes Kempston 1 and Sinclair 2.

In short. The membrane, a PS/2 keyboard and key-mode joysticks all press keys in one scan that renews a matrix copy every 164.6 µs, and port `$FE` reads that copy, so old programs work unchanged. A pressed key or mouse button reads 0, while a Kempston joystick reads 1 for pressed. NextRegs `$B0` and `$B1` report the sixteen extra keys directly, and bit 4 of NextReg `$68` stops them entering their shift pairs, but a soft reset clears it. The mouse ports are wrapping counters, so subtract successive readings, and remember that Y grows upwards.

Part II

The Picture

CHAPTER 8

How a Television Draws a Picture

Every home computer of the ZX Spectrum's generation was plugged into a television, and the television set the rules: how fast the picture had to be delivered, in what order, and what the signal had to look like. That explains a great deal about why these machines were built the way they were. You do not need to know anything about televisions or video hardware to follow this chapter.

- How an electron beam draws a picture, line by line
- How a video signal turns time into position, and what the sync pulses do
- Why a computer has to keep pace with the beam
- How colour rides on a signal made for black and white

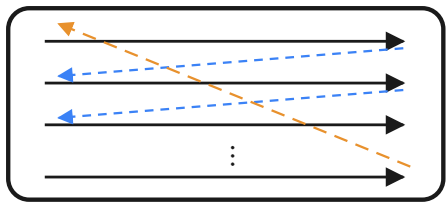
The electron beam

A cathode ray tube (CRT) television has an **electron gun** at the back of the glass. It shoots a thin beam of electrons at the inside of the screen, which is coated with **phosphor**: wherever the beam hits, the phosphor glows for a moment. A stronger beam makes a brighter dot; no beam leaves the spot dark.

The beam can light only one spot at a time. To make a whole picture, magnetic coils around the tube sweep it across the screen in a fixed pattern called a **raster** (Figure 8.1):

1. The beam travels from left to right, drawing one horizontal **scan line**.
2. At the right edge it snaps back to the left, a little lower, and draws the next line. This quick return is the **horizontal retrace**.
3. When it reaches the bottom, it snaps back to the top-left corner and starts again. This is the **vertical retrace**.

During both retraces the beam is switched off, so that it draws nothing on its way back. This off time is called **blanking**.



scan line
the beam draws

horizontal retrace (HSYNC)
back to the left, one line lower

vertical retrace (VSYNC)
back to the top: a new frame

Figure 8.1: The raster (dashed: the beam is off)

A European (PAL) television draws a line every **64 microseconds** and a full screen, a **frame**, 50 times a second. That is fast enough that your eye sees a steady picture instead of a moving dot.



KLIVE IDE Where the beam is

Pause the emulated machine, and a dashed line across the screen marks the row the beam is drawing, with a label that gives its raster line and its position in the line. When the pause falls in a retrace, the label says so: `HBLANK` between two lines, `VBLANK` between two frames. The **Show/Hide the beam position** button on the emulator’s toolbar, or `beam on` and `beam off`, switches the line on and off.

The signal carries only “how bright, right now”

Here is the part that matters most. The signal going into the television **does not contain pixels or coordinates**. It is one voltage that changes over time, and it means *“this is how bright the spot under the beam is, at this very instant”*.

The television knows only *when*. A dot appears 20 microseconds into a line because the voltage changed 20 microseconds after the line started. **In a video signal, time is position.**

So how does the television know when a line or a frame starts? The signal also carries two kinds of special pulses (Figure 8.2), each a short stretch of “blacker than black” voltage:

- **HSYNC** (horizontal sync): “start a new line now”. One arrives every 64 µs.
- **VSYNC** (vertical sync): “go back to the top now”. One arrives every frame. It lasts several lines, so the television can tell it from an HSYNC.

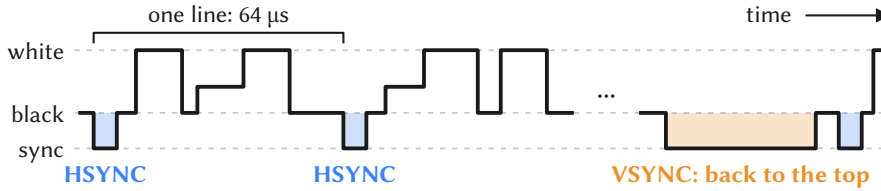


Figure 8.2: The video signal, simplified

The television locks onto these pulses. Its sweep circuits keep running at roughly the right speed even if a pulse is missing, which is why a picture rolls or tears rather than vanishing when the sync goes wrong.

Why you must follow the beam

There is no memory inside an old television. Whatever is on the signal wire **now** becomes the dot under the beam **now**, and a few milliseconds later it has faded away. So the computer has to deliver every dot **at exactly the right microsecond**, 50 times a second, forever. Programmers call this **racing the beam**.

> **PAL and NTSC.** "50 times a second" is the European **PAL** standard: a line every 64 µs and 50 pictures a second, with 625 lines split into two interlaced halves of 312½. North America and Japan use **NTSC**: about 60 pictures a second, with 525 lines in halves of 262½. Home computers ignore the interlacing and draw the same half-frame every time. The numbers in this chapter are PAL unless it says otherwise.



KLIVE IDE The paused screen

A paused screen is this frame down to the beam, and the last frame below it, hatched. Point at any spot, and Klive tells you how many clock ticks the beam needs to reach it: the time your code has left to change it. The **instant screen** redraws the whole picture from the memory as it is now.

How colour is displayed

Colour television had to be added on top of all this without breaking the millions of black-and-white sets already in people's homes. That one requirement shapes everything that follows.

Three phosphors, three beams

Mix red, green and blue light in the right proportions and the eye sees almost any colour: red and green make yellow, all three at full strength make white. A colour screen is covered with tiny dots of three phosphors that glow **red**, **green** and **blue**, in groups of three called **triads**, too small for your eye to see separately.

At the back of the tube there are **three electron guns**, sweeping the same raster together. Each beam must hit only the phosphor of its own colour, and that is the job of the **shadow mask**: a thin metal sheet just behind the screen, with one tiny hole for every triad. The guns sit side by side, so the beams arrive at each hole from slightly different angles and land on different spots (Figure 8.3).

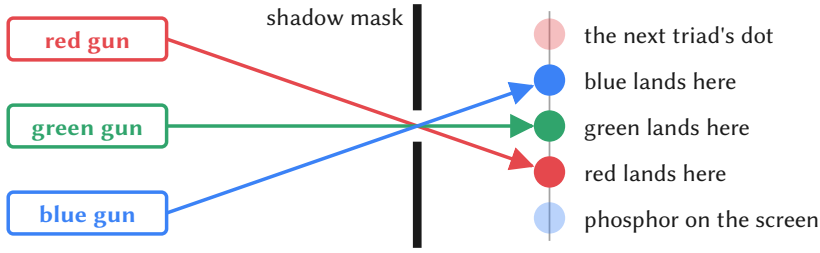


Figure 8.3: A shadow mask from the side, much enlarged

Inside the television, then, a colour picture is **three black-and-white pictures at once**, drawn by three beams that move in step. Everything above still holds for each of them.

Colour on a signal made for black and white

Three pictures need three signals, but the aerial socket has room for only one, and an old black-and-white set must still make sense of it. So the signal keeps carrying the brightness, now called the **luminance** (Y), a weighted mix of the three colours that matches how bright each looks to the eye:

$$Y = 0.299 \times \text{red} + 0.587 \times \text{green} + 0.114 \times \text{blue}$$

A black-and-white television displays Y and gets a correct grey picture.

The colour is added **on top**, as the **chrominance** (or **chroma**). Since Y already holds the brightness, two numbers are enough: how far the blue is from the brightness (**B** – **Y**) and how far the red is (**R** – **Y**). These **colour-difference signals** are called **U** and **V** in PAL, and the television recovers green by arithmetic.

The colour subcarrier: phase is hue, amplitude is saturation

U and V ride on a fast sine wave, the **colour subcarrier**: about **4.43 MHz** in PAL and **3.58 MHz** in NTSC. They are combined into one wave of that frequency, and its shape carries the colour:

- The **phase** of the wave (where its peaks fall in time) gives the **hue**.
- The **amplitude** of the wave (how big its swings are) gives the **saturation**: a large wave is a vivid colour, a small one a pale colour, no wave at all is grey.

This wave is added to the luminance, so where the picture is coloured, the signal wiggles fast around its average level.

The colour burst: the television's reference

Phase only means something against a reference. So every line carries a short sample of the subcarrier at a fixed phase: the **colour burst**, about ten cycles of the wave just after the HSYNC, before the picture of the line begins (Figure 8.4).

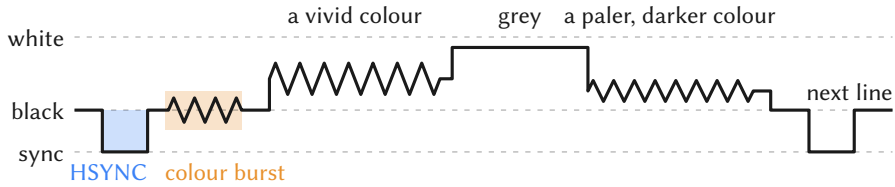


Figure 8.4: One line of a colour signal, simplified

On every line the television pulls its own oscillator into step with the burst, then compares the incoming wave against it. That gives U and V, and with Y, the voltages that drive the three guns. **In a colour signal, time is still position**, but on a far finer clock: a phase error of a few nanoseconds is enough to change a colour.

PAL and NTSC: what to do about phase errors

Anything that delays the subcarrier by a tiny amount shifts every colour around the colour circle, and faces turn green or purple. **NTSC** lives with it: its televisions have a **hue** knob, and the viewer turns it until faces look right. **PAL** (*Phase Alternating Line*) flips the sign of V on every other line, so an error shifts the hue one way on one line and the opposite way on the next. The television averages each line with the previous one, and the errors cancel, leaving a slight loss of saturation that the eye readily forgives.

Colour is blurrier than brightness

The eye sees fine detail in brightness much better than in colour, and the colour system takes advantage of it. The luminance has about 5 MHz of bandwidth in PAL, but U and V only about 1.3 MHz, so the colour can change only about a quarter as quickly along a line as the brightness can.

A designer who knows the colour is blurry has little reason to let it change from dot to dot, so many machines of the era give a whole block of dots a single colour. The ZX Spectrum's 8 × 8 colour cells are a well-known example.

How a computer produces it

A home computer's video chip usually works out red, green and blue, or luminance and colour-difference levels, for each dot, and hands them to a **PAL (or NTSC) encoder**. That chip mixes in the subcarrier, adds the burst, and produces the **composite video** signal, which a modulator turns into a signal for the aerial socket.

Every step costs some quality. An **RGB** connection skips the whole chain, carrying the red, green and blue voltages on separate wires, plus the sync, straight to the guns.

In short. A television draws one spot at a time, line by line, and its signal says only how bright that spot is right now, so time is position and the sync pulses keep the beam in step. With no memory in the set, a computer must deliver every dot at exactly the right moment. Colour rides on the same signal as a fast wave whose phase is the hue and whose amplitude is the saturation. Colour is blurrier than brightness, which is why machines like the Spectrum could give a whole block of dots one pair of colours.

CHAPTER 9

The Big Picture: Five Producers, Four Layers, One Dot

Load a modern game on a ZX Spectrum Next and look at one frame. A playfield built from tiles scrolls smoothly from right to left. Behind it a background in a few hundred colours scrolls more slowly. Dozens of characters move over both, and at the bottom sits a score panel in the unmistakable Spectrum font. How can one screen be all of that at once?

On the Spectrum, one chip produced every dot. On the Next, **five producers of dots** run side by side, each reading its own memory. They feed **four layers**: the ULA and LoRes share one. So for every position on the screen there are four candidate dots, and something has to pick one of them, about fourteen million times a second. No pressure. Nearly every surprise a Next program gives you on screen is that choice at work: a sprite that vanishes behind the background, a score panel that shows through a hole in Layer 2, a magenta that turns out to be no colour at all.

This chapter is the map; the screen chapters after it take each piece apart. They all assume you know the Spectrum's own screen, the ULA's bitmap and its attributes, and Chapter 8 explains the beam and the lines.

- What the Next kept from the Spectrum, and the layers it built beside it
- Where each layer may draw: the wide area, clip windows and scroll offsets
- Why every layer gives exactly two answers for every dot
- How a dot travels through the video pipeline to the screen

> The numbers in the screen chapters are for the Next's **128K/+3 display timing at 50 Hz**, with the master clock at exactly 28 MHz (VGA mode 0, or HDMI). The other timings change the size of the frame, not the way it is drawn; Chapter 10 lists them.

From one chip to a pipeline

The Spectrum's ULA reads the screen memory, turns bytes into dots and sends them to the television; nothing else takes part. The Next keeps that ULA, compatible to

the bit, so every 48K or 128K program that writes to `$4000` sees exactly the screen it expects.

What the Next adds is **more producers beside it**, each with its own memory and its own idea of what a picture is: a bitmap with a byte per dot, a grid of tiles, a set of sprites. They run at the same time, in step with the same beam (Figure 9.1). A **compositor** after them picks the winner for each dot. I call it **the referee**: it looks at the candidates, applies rules it never bends, and names a winner, without caring how any of them got there.

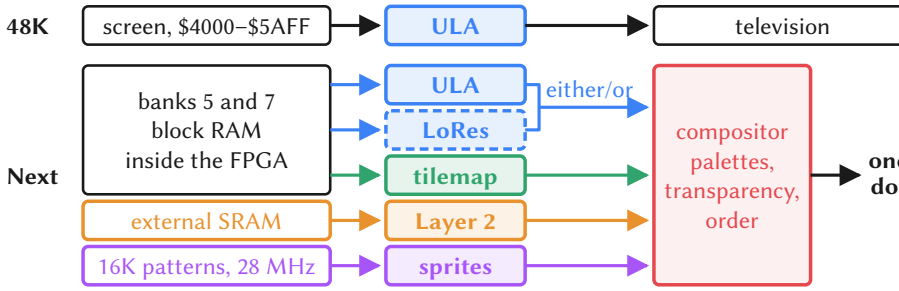


Figure 9.1: One path against five producers

The producers do not queue for one memory, as the CPU and the ULA do on the Spectrum. The FPGA keeps **bank 5** (and the first 8K of bank 7, the 128K's second screen) in its own block RAM, with two ports: one shared by the CPU and LoRes, the other by the ULA and the tilemap. **Layer 2** reads the external SRAM, the Next's main memory. The **sprites** keep their shapes in a 16K memory of their own, which the CPU reaches only through I/O ports. If you ask me, that is the real secret of the Next's screen: not faster memory for everyone, but no queue at all.

The cast of layers

The layers are a **stack** (Figure 9.2). At every dot, the top layer is shown if it has something there. Where it is **transparent**, the dot comes from the layer below it, and so on to the bottom of the stack. That is the referee's whole method: look down, take the first dot that is really there.

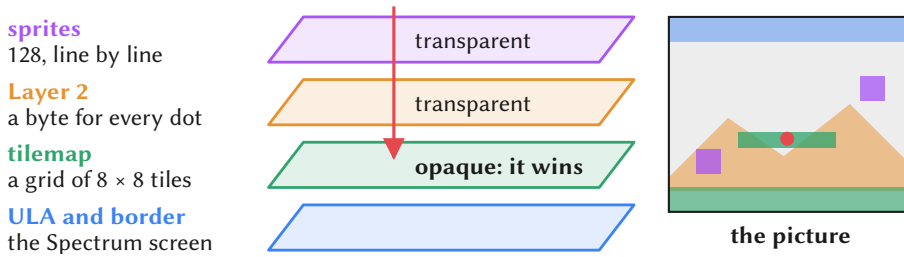


Figure 9.2: The layers pulled apart, in their default order

The order is **not fixed**. The Sprite and Layers System register, NextReg `$15`, picks one of six orders for the sprites, Layer 2 and the ULA (with the tilemap), and Chapter 17 shows them all. The figures here use the default: the sprites on top, then Layer 2, then the tilemap, then the ULA.

Each producer has a chapter of its own later. I give each a nickname as well, because I find a character far easier to remember than a register.

- **The ULA.** The Spectrum's own screen, `$4000` in bank 5 or the shadow screen in bank 7, with all of its rules. When a program asks, it does more: the Timex modes, **ULANext** and **ULA+**, hardware scrolling and a clip window. I call it **the veteran**: on stage since 1982, and it still plays every old part to the T-state.
- **LoRes.** A 128 × 96 screen with a byte, any of 256 colours, for every 2 × 2 dots. When it is on, it **replaces the ULA's picture** and takes the ULA's place in everything that follows; only the border stays the ULA's. That makes it **the understudy**.
- **Layer 2.** A plain bitmap with **one byte for every dot**, in 256 × 192, 320 × 256 or 640 × 256 (half a byte a dot). No attributes, no clash, at the price of 48K or 80K of memory. Layer 2 is **the painter**, the name Chapter 3 gives its mapping.
- **The tilemap.** A grid of 8 × 8 **tiles**, 40 or 80 across and 32 down, each a reference into a set of tile shapes and a byte saying how to draw it. A whole screen costs a few kilobytes. The tilemap is **the set builder**: it builds its scenery from a box of ready-made pieces.
- **The sprites.** **128 sprites** of 16 × 16 dots, each with its own position, shape, colours, mirroring, rotation and size. The hardware draws each line of them just before the beam needs it. The sprites are **the troupe**, each performer free to wander across the stage.

Two colours are not producers. **The border** is what the ULA layer shows outside its 256 × 192 area, in the colour of port `$FE`, which is why sprites and Layer 2 can be drawn over it. And when every layer says "nothing here", the dot takes **the fallback colour**, from the Fallback Colour register, NextReg `$4A`.

Here is the screen of a game in progress, split into what each layer contributes (Figure 9.3). The sky belongs to no layer: the ULA's screen is clipped to the panel at

the top, Layer 2 is transparent above its hills, and where all four are transparent the fallback colour shows.

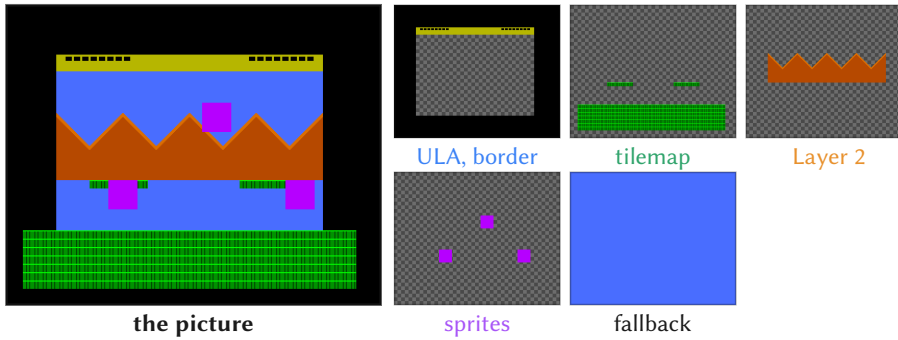


Figure 9.3: One frame, layer by layer (chequerboard: the layer is transparent)

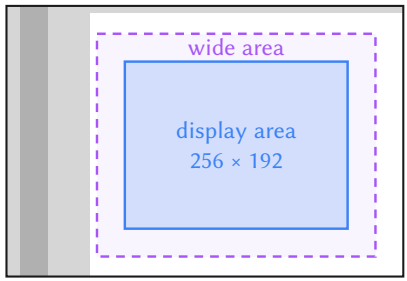


KLIVE IDE The Layers strip

Turn on **Settings › Emulator › Show the Layers strip**: a strip under the screen has a chip for each layer, top layer first, in the order NextReg $15 sets. Click a chip to hide that layer, or **S** to show it alone over a checkerboard. A layer the program has switched off has its name struck through.

Where each layer can draw

The Spectrum has one rectangle that matters, the 256 × 192 display area, and a border around it. The Next adds a second: the **wide area**, 320 × 256 dots, the display area with 32 dots added on every side (Figure 9.4).



one line: 456 dots, 228 T-states at 3.5 MHz

one frame: 311 lines

grey: blanking, the left 96 dots and the top 8 lines, with HSYNC and VSYNC inside

dashed: the wide area, 320 × 256, 32 dots or lines beyond the display area on every side

outside it: the border

Figure 9.4: The 128K frame at 50 Hz, to scale

Layer	Its canvas	Can it cover the border?
ULA	the whole visible frame: its 256 × 192 picture, and the border colour everywhere else	it *is* the border
LoRes	256 × 192 (its 128 × 96 dots doubled)	no
Layer 2	256 × 192, or 320 × 256 and 640 × 256 in the wide area	in the wide modes, up to the wide area's edge
Tilemap	320 × 256, always the wide area	up to the wide area's edge
Sprites	320 × 256 coordinate space	only when "sprites over border" is on (NextReg $15, bit 1); otherwise clipped to 256 × 192

Beyond the wide area, only the border colour is ever shown. An HDMI monitor shows a window of **360 × 288** dots: the wide area and a little border round it.

Clip windows: a rectangle for each layer

Each layer has a **clip window**, a rectangle on the screen. Outside it, the layer answers "transparent" for every dot, so the dot comes from the layers below. A clip window does not change the layer's memory or move its picture: it is a mask. That makes it the cheapest tool on the machine for a score panel, a framed playfield or a letterbox. One write of four bytes does it, where clearing the same area in memory would take thousands.

There are four windows, one for each layer (LoRes uses the ULA's):

Window	NextReg	Counts in	X unit	After reset
ULA and LoRes	$1A	display area	1 dot	0–255, 0–191
Layer 2, 256 × 192	$18	display area	1 dot	0–255, 0–191
Layer 2, wide modes	$18	wide area	2 dots	0–255, 0–191
Tilemap	$1B	wide area	2 dots	0–159, 0–255
Sprites	$19	display area	1 dot	0–255, 0–191
Sprites over the border	$19	wide area	2 dots	0–255, 0–191

Sprites go over the border when bit 1 of NextReg $15 is set. Their window then applies only if bit 5 is set too; if not, they cover the whole wide area.

Each window is four bytes, **X1**, **X2**, **Y1**, **Y2**, and both edges are inside it: X 0–255 is 256 dots wide. Your program writes all four to the same NextReg; a small index inside the machine says which coordinate the next write sets, steps on after every write, and

wraps from Y2 back to X1. The Clip Window Control register, NextReg `$1C`, resets the index of any of the four windows. My advice is to reset it before every window you write.

A coordinate is one byte, but the wide area is 320 dots across, so the windows that work there count X in **units of two dots** (Figure 9.5): X1 = 10 means dot 20, and X2 = 159 means dots up to 319. The Y coordinates stay in lines.

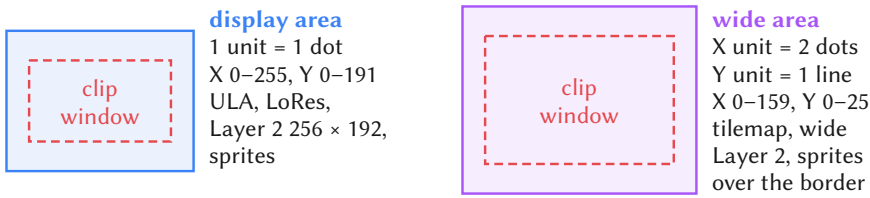


Figure 9.5: The two coordinate systems of the clip windows

One thing follows, and let me warn you, it surprises programmers: Layer 2's window starts as 0–255, 0–191 in every mode, which in the 320 × 256 mode covers the full width but only the top **192 of the 256 lines**. If your program uses the wide modes, set the window to 0–159, 0–255 first.

The ULA's window is special in one way: the border is never clipped. The veteran lets you fence in its picture, but never its border.



KLIVE IDE Seeing the windows

The Layers strip's dashed-square button draws all four clip windows on the screen, each in its layer's colour. A layer cut off by its window is then easy to tell from a layer that is transparent there.

Scrolling without moving memory

Moving a picture on the Spectrum meant moving bytes, and scrolling by one dot meant shifting every byte of the bitmap: about 117,000 T-states, more than one and a half frames of a 48K. To move a 48K Layer 2 picture by copying, a 3.5 MHz CPU would need about fourteen frames.

So the Next lets the hardware do it. All the producers except the sprites have **scroll offsets**: registers that say where in its picture the layer starts reading. With an X offset of 96, the top-left dot of the layer's area is the 97th dot of the first line, and everything shifts 96 dots to the left. One register write moves the whole picture. I think hardware scrolling is the single biggest gift the Next gives a games programmer.

And the picture **wraps round**: what is pushed off the left edge comes back in on the right, and what leaves the top comes back at the bottom (Figure 9.6). No byte of memory moves.

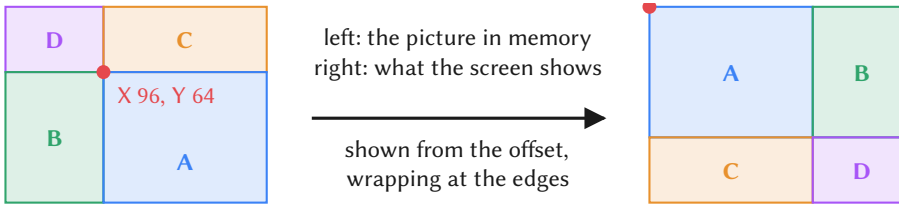


Figure 9.6: A scroll offset wraps the picture

Wrapping is what makes endless scrolling work. A game draws each new column of graphics into the column of memory that has just scrolled off the left edge, just before it comes back in on the right, so the picture never needs to be bigger than the screen.

Because each layer has its own offsets, the layers can scroll at **different speeds**, which is how a game makes a picture look deep. Figure 9.7 shows the scene from the cast three frames apart: the platforms move three times as fast as the hills, while the ULA's panel and the sprites stay put.

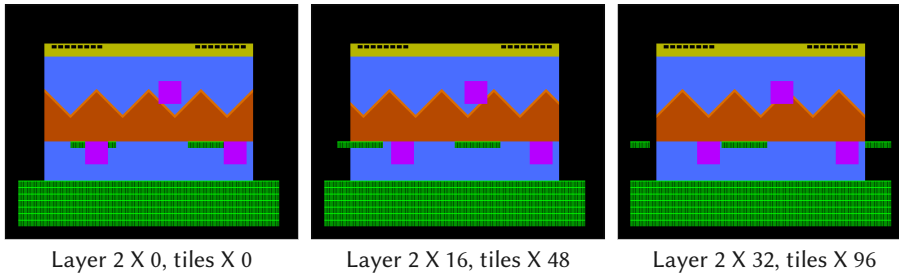


Figure 9.7: Parallax on the machine

The clip window belongs to the screen and the scroll offset to the layer's picture, so a layer can scroll inside a window that stays put. Each layer's chapter gives its offset registers; changing them mid-frame is the subject of Chapter 18.

Two answers for every dot

Here is the idea all the screen chapters are built on, and I think it is the most elegant one in the Next's design. For every position of the beam, every layer gives the compositor exactly **two answers**:

1. **Which colour?** As an 8-bit **index** into a palette, not as a colour.
2. **Am I transparent here?** Whether this layer has anything to show at this dot at all.

That is all the referee ever learns about a layer. Every rule the later chapters describe is a way of arriving at these two answers. A few layers add a **flag**: the border says "I am border", the tilemap says "I am behind the ULA", and Layer 2's palette can say "draw me in front of everything".

Look at the red lines in Figure 9.8: there are **two kinds of transparency**.

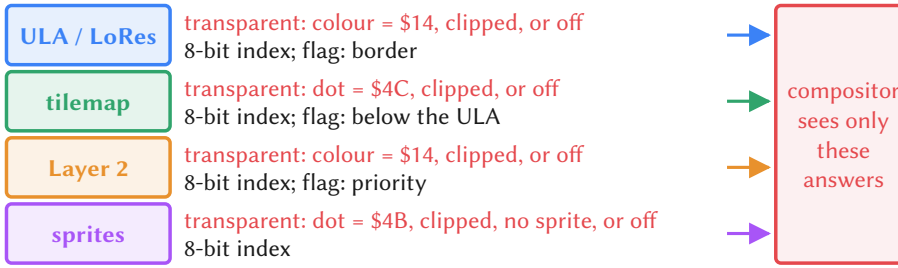


Figure 9.8: What each layer tells the compositor

- The ULA, LoRes and Layer 2 are transparent where their dot's **colour**, after the palette lookup, equals the **global transparency colour** in NextReg `$14`. "This colour means see-through" is the natural rule for a bitmap.
- The sprites and the tilemap are transparent where their **index**, before the lookup, equals the value in NextReg `$4B` (sprites) or NextReg `$4C` (tilemap). "This value in the pattern is the hole" is the natural rule for a shape. (The tilemap's text mode uses the colour rule instead.)

NextRegs `$14` and `$4B` both start at `$E3`, and so does the fallback colour. That is why `$E3`, a bright magenta, is the Next's way of saying "nobody home": in the default Layer 2 palette, index `$E3` *is* colour `$E3`, so a Layer 2 bitmap filled with `$E3` is invisible from the start.

A layer that is switched off, or clipped at this dot, answers "transparent" too. Three ways to say "not me", and the referee treats them all alike.



KLIVE IDE Marking transparency

The Layers strip's checkerboard button paints magenta wherever no layer is opaque, in place of the fallback colour. With a layer shown alone, it marks that layer's own transparent dots: its "not me" answers, made visible.

The video pipeline

The compositor is a short **pipeline**. A dot enters it when the beam's counters reach its position and leaves it, as a colour, a few clock ticks later, and a new dot comes out at every tick, like an assembly line that never stops (Figure 9.9). Stage 3 changes nothing: it only delays the colour by three more ticks, so that the logic of stage 2 can keep up with the clock.

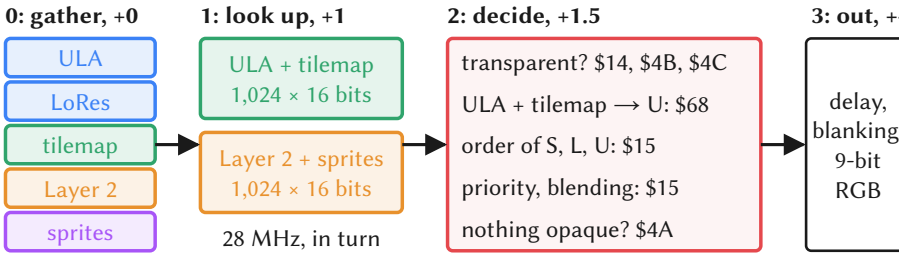


Figure 9.9: The video pipeline (+n: 7 MHz periods later)

Two dots for every position of the beam

So far, a dot has been one step of the beam's counters, one 7 MHz dot. The pipeline produces **two** colours for it, a **left half** and a **right half**, each one 14 MHz dot wide. A layer that draws 256 dots across offers the same answers for both halves. The layers that draw 512 or 640 dots across, the ULA's Timex hi-res mode, the tilemap's 80 columns and Layer 2's 640 × 256, offer a different answer for each.

The referee decides each half **on its own**, so the layers can mix inside one position of the beam: a hi-res tile dot that is opaque in the left half and transparent in the right shows the tile in one half and whatever lies below it in the other (Figure 9.10).

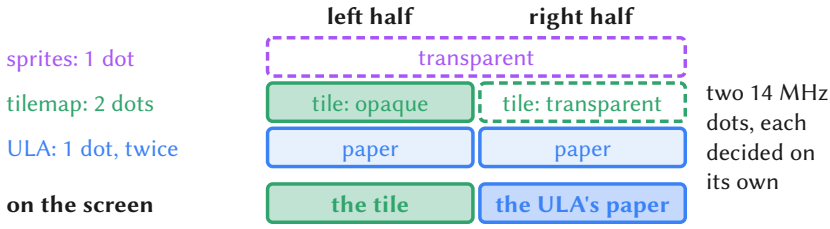


Figure 9.10: One position of the beam, two dots

Stage 0: gather

Each producer offers its two answers. In the same tick, the pipeline **latches the registers** that steer everything after it: the layer order, the transparency and fallback colours, which palettes are shown, which layers are on. That is how a write to one of these registers in the middle of a line takes effect: from the next dot the beam reaches, and never half-way through one. If you plan raster effects, I would keep this rule closer than any other. Chapter 18 lists the few settings taken less often.

Stage 1: look up

The indices go through the **palettes**. There are eight, a first and a second for each of the ULA, the tilemap, Layer 2 and the sprites, 256 entries each. But there are only **two memories**, each holding four of them: one for the ULA and the tilemap, one for Layer 2 and the sprites (Figure 9.11). An entry holds the 9-bit colour and a **priority bit**, which only Layer 2 uses: it can lift a Layer 2 dot above everything.

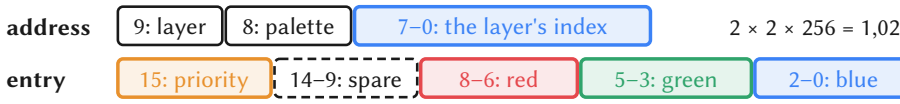


Figure 9.11: One palette memory

Sharing has a price in time, and the pipeline pays it with speed. Each memory is read at **every tick of the 28 MHz clock**, its two layers taking turns, so all four layers get their colours for every 14 MHz dot. That is why stage 1 has **two boxes** in Figure 9.9, not four: they are the two memories, working at the same time.

Stage 2: decide

Now the referee has four colours with their flags. It checks transparency, mixes the ULA and the tilemap into one layer, **U**, and applies one of the six orders of **S** (sprites), **L** (Layer 2) and **U**, from SLU, the default, to ULS. The first opaque one wins. A Layer 2 dot with the priority bit wins whatever the order, two settings of NextReg $15 add colours instead of ordering them, and if nothing is opaque, the fallback colour shows. Chapter 17 follows each rule.



KLIVE IDE The Layers document

View › Machine Views › Layers opens the priority stack that NextReg $15 selects, and a card for each layer with its picture alone and its clip window. A card also says why its layer might not be on the screen: switched off, an empty clip window, or every dot transparent.

One dot through the pipeline

Let's take the dot marked in the layers pulled apart, repeated here from page 112 (Figure 9.12), in the default settings, and follow it:

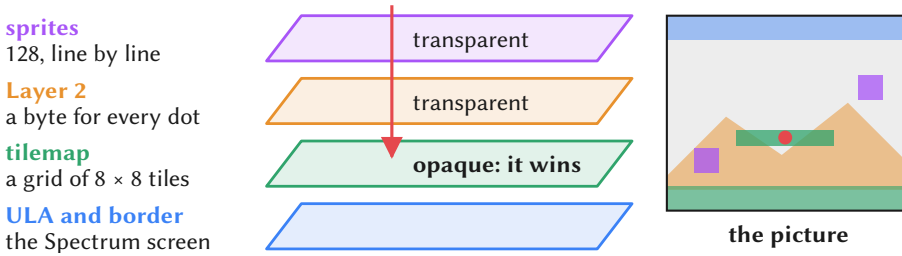


Figure 9.12: The dot to follow, in the layers pulled apart

1. **Stage 0.** The sprites have nothing here: transparent. Layer 2's bitmap holds $E3, below its band of hills. The tilemap offers a dot of a platform tile, index $13. The ULA offers its paper colour.
2. **Stage 1.** Layer 2's $E3 becomes the colour $E3. The tile's $13 becomes whatever the tilemap palette holds there, a green.
3. **Stage 2.** Layer 2's colour equals the value in NextReg $14: transparent. The tile is not behind the ULA, so U is the tile's green. The order is SLU: the sprites are transparent, Layer 2 is transparent, U is not. The dot is green.



KLIVE IDE The pixel probe

Pause the machine, turn on the Layers strip's crosshair and point at a dot. The probe lists each layer's answer there, the order and the fallback colour in force on that line, and the rule that chose the winner. Click the dot to open the inspector of the layer that drew it.

Three clocks

Three numbers make it all fit (Figure 9.13): the **28 MHz** master clock, which everything in the pipeline counts in; **4 ticks per dot**, because the beam's counters step at 7 MHz, and the memories hand out those ticks in turn; and **2 dots per step** for the fine layers, which run at 14 MHz.

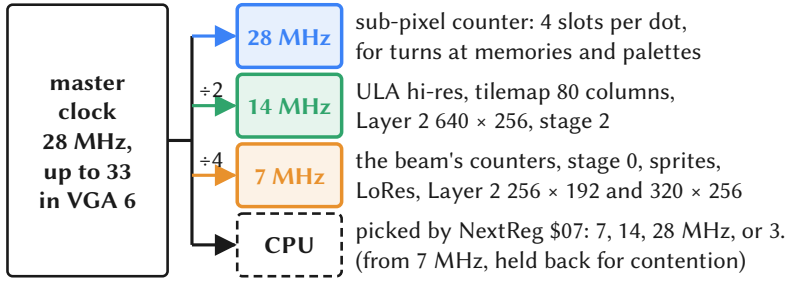


Figure 9.13: The clocks of the picture

The CPU's clock is **selected from the video clocks**, so its speed changes how much a program can do in a frame, never the picture.

512 colours

Every path through the pipeline ends in a **9-bit colour**, 3 bits each of red, green and blue, **RGB333**, because three bits a channel is what the Next's video connector puts out (Figure 9.14).

Bright magenta is the one Spectrum colour that has moved, by a step of green, so that it is not mistaken for `$E3`. Chapter 11 tells that story, and how the eight palettes are selected and written.

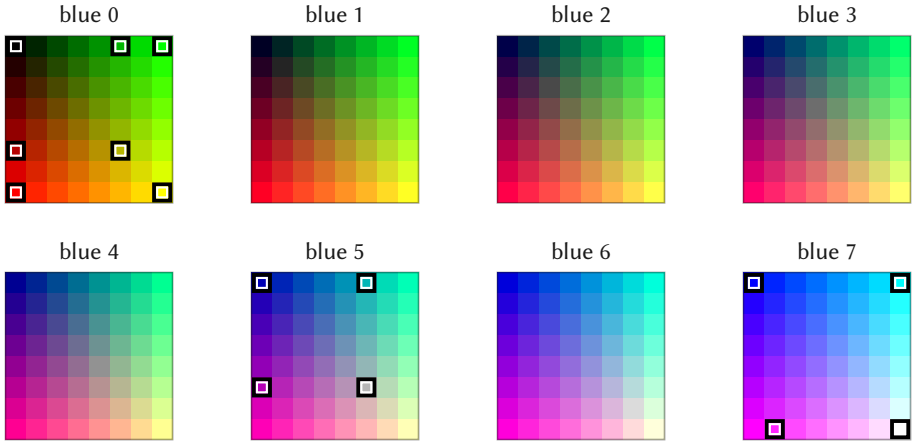


Figure 9.14: The 512 colours (ringed: the Spectrum's 15, as black and bright black are the same)

What stays the same

All this is added *beside* the Spectrum. Switch the Next to 48K or 128K timing and leave the new layers off, and the frame keeps the Spectrum's shape, the ULA reads the screen in the same order at the same moments, and the border is still port `$FE`. **Contention** stops the CPU at the same T-states, but only at **3.5 MHz**: a program that counts T-states against the beam works only at the speed it was written for. Chapter 12 covers what the Next changes about the ULA.

In short. Five producers feed four layers, and for every position of the beam each layer gives the compositor only two answers: an 8-bit colour index, and whether it is transparent. The compositor takes the first opaque dot in the order NextReg $15 sets, and shows the colour in NextReg `$4A` if there is none. Clip windows mask a layer on the screen, and scroll offsets change where it starts reading, without moving a byte. The pipeline decides two 14 MHz dots for every position of the beam.

CHAPTER 10

The Frame and the Beam: Two Counters, Four Timings

A 48K game that changes the border colour at an exact T-state works on the Next. So does a 128K demo on a frame of a different shape, and a Pentagon demo on a third. And the very same machine sends a sharp picture to a modern monitor over HDMI. One machine, four old frames and a new cable. How?

All of it rests on **two counters**. One counts the dots of a line, the other the lines of a frame, and every event of the picture is a moment at which they reach a certain value. Every effect that touches the screen at the right moment, a split border, a raster bar, a status panel that stays still, is timed against them.

This chapter builds on Chapter 9 and on the frame of the original 48K Spectrum. If the beam, blanking and sync are new to you, Chapter 8 explains them first.

- How two counters define every part of the frame
- What the four timings are, and what 60 Hz does to them
- How the frame interrupt and the line interrupt work
- How the picture leaves the machine as RGB, VGA or HDMI

> The figures use the **128K/+3 timing at 50 Hz** unless they say otherwise, and the frame rates are those of the master clock at exactly 28 MHz (VGA mode 0, or HDMI). That is how the Next starts: its default personality, ZX Spectrum Next (standard), is a +2A/+3/Next machine, and 50 Hz is the default display frequency.

Two counters

The **horizontal counter** counts the dots of a line at 7 MHz, from 0 at the start of the line's blanking to the last dot, 455 in 128K timing. Each time it starts again, the **vertical counter** counts one more line, from 0 to 310. I call the pair **the timekeepers**: they never do anything but count, and every other part of the screen asks them what time it is. Everything else is a **comparison** with these two numbers (Figure 10.1):

- **Blanking**: dots 0 to 95 of every line, and lines 0 to 7 of every frame.

- **Sync**: the horizontal sync pulse on dots 16 to 47, the vertical one on lines 1 to 4, inside the blanking.
- **The display area**: dots 136 to 391, lines 64 to 255, the Spectrum's 256 × 192.
- **The wide area**: dots 104 to 423, lines 32 to 287, the 320 × 256 of Layer 2's wide modes, the tilemap and the sprites.
- **The border**: everything else that is not blanked.

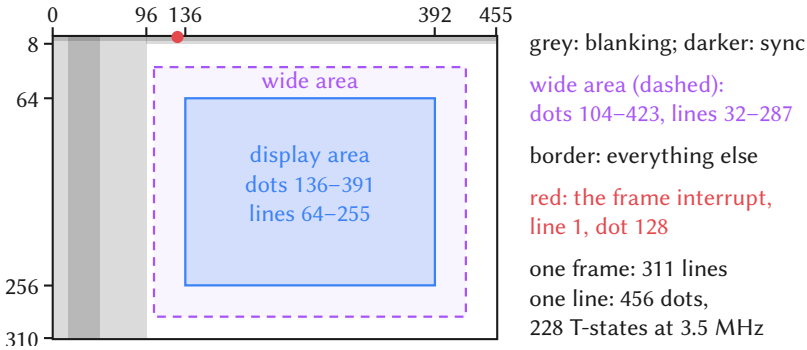


Figure 10.1: The 128K frame at 50 Hz, to scale (top: dots, left: lines)

Three numbers hold it together. A **7 MHz dot** is half a T-state of a 3.5 MHz CPU, as on the Spectrum. A line has **456 dots**, 228 T-states (448 and 224 on the 48K). A frame has **311 lines**, 70,908 T-states (312 and 69,888 on the 48K).



KLIVE IDE The two counters, paused

When the Next is paused, the beam line on Klive's screen has a label such as `line 158 · paper row 94 · hc 381 · HC 72,429`: the raster line, the position in the line in 7 MHz dots (`hc`), and the position in the whole frame, in the same dots (`HC`). These are this section's two counters, read at the stop.

Counters that start early

A dot has to be fetched from memory and looked up in a palette before it can be shown, so each producer works a little ahead of the beam. The ULA's counter starts 12 dots before the display area, the counter of the wide area 16 dots before it and 2 lines above it. The picture still appears exactly where the main counters say: the head start is used up by the work.

Your program can change the timing at any moment, but the counters take a new timing only **at the end of a frame**. A frame is never half one shape and half another.

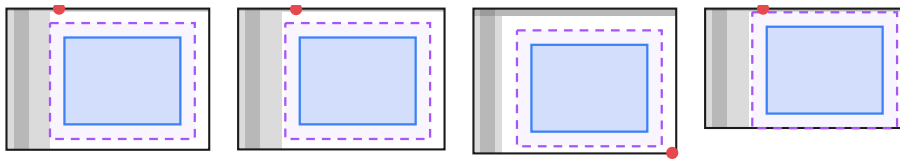
Four timings, two frame rates

The Next has four timings. Bits 6–4 of the Machine Type register, NextReg `$03`, choose one (with bit 7 set to allow the change):

Timing	Dots × lines	T-states a frame	Frame rate	Display from line
48K	448 × 312	69,888	50.08 Hz	64
128K/+2, +3	456 × 311	70,908	49.36 Hz	64
Pentagon	448 × 320	71,680	48.83 Hz	80

The shape of the frame is a **contract** with the programs: give a program that counts T-states a frame one line longer, and its stripes drift. So the Next has the frame of each machine, to the dot. This is the part of the Next I respect most: compatibility taken literally.

The 128K and the +3 share a frame, but the +3's frame interrupt comes one T-state earlier, and the two hold up the CPU in different memory banks and in a different pattern. The Pentagon, a Russian clone, has a taller frame, wider vertical blanking and no contention at all (Figure 10.2).



48K
448 × 312, 50.08 Hz
display from line 64

128K/+2, +3
456 × 311, 49.36 Hz
display from line 64

Pentagon
448 × 320, 48.83 Hz
display from line 80

128K at 60 Hz
456 × 264, 58.15 Hz
display from line 40

Figure 10.2: The frame shapes at the same scale, with 60 Hz (red: the frame interrupt)

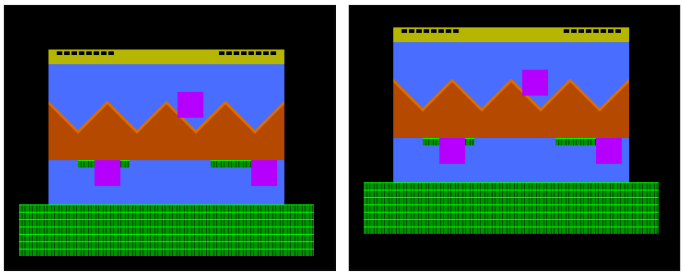
The Next's dots are exactly 7 MHz, where the original 128K's ran at about 7.09 MHz, so the same frame runs at **49.36 Hz** instead of 50.02 Hz. Your program cannot see the difference: every frame has the same number of T-states, and a program carries no stopwatch.

The 60 Hz frame

Bit 2 of the Peripheral 1 Setting register, NextReg `$05`, switches any timing but the Pentagon to **60 Hz**. The frame loses lines: 264 instead of 311 or 312, and the display area starts on line 40. So the wide area fills every visible line and there is no border

above or below it at all. The frame rate becomes 58.15 Hz (128K) or 59.19 Hz (48K). If your program uses the wide area, look at it at 60 Hz too.

Figure 10.3 shows the scene of the previous chapter at both rates. At 60 Hz the picture moves up, the top of the wide area falls outside the window, and below the last line of the frame there is nothing to draw.



left: 50 Hz

right: 60 Hz
the same program,
NextReg $05
bit 2 set

Figure 10.3: The same window, two frame rates

The frame interrupt

Once in every frame the ULA, the veteran of the cast, raises the **frame interrupt**, the Spectrum's 50 times a second. It comes well before the display area, so that a program that waits for it with `HALT` has time to update the screen before the beam gets there:

- **48K**: on line 0, 64 lines before the display area.
- **128K/+2** and **+3**: on line 1, 63 lines before it.
- **60 Hz**: on line 0, only 40 lines before it.
- **Pentagon**: on the very last line of the frame, so its 80 lines of head start begin at the bottom.

The T-states between the interrupt and any dot of the screen are those of the original machines. The interrupt line is held for 32 CPU cycles in 48K and +3 timing and 36 in 128K and Pentagon timing. Bit 2 of the Line Interrupt Control register, NextReg $22, switches the frame interrupt off.

The line interrupt

The Next adds a second one: the **line interrupt**, raised before a line your program chooses. Bit 1 of NextReg $22 switches it on, and the line number is nine bits, bit 0 of NextReg $22 and the eight bits of the Line Interrupt Value register, NextReg $23. I call it **the alarm clock**: you set it for a line, and it rings a little before that line arrives.

Now, here comes the curious part. The line numbers are counted from the display area, not from the top of the frame: **line 0 is the first line of the display area**, 191 the last, and the numbers go on through the bottom border to the end of the frame and then continue at the top of the next one. So in 128K timing at 50 Hz, the first line of the frame is 247 and the line just above the display area is 310 (Figure 10.4).

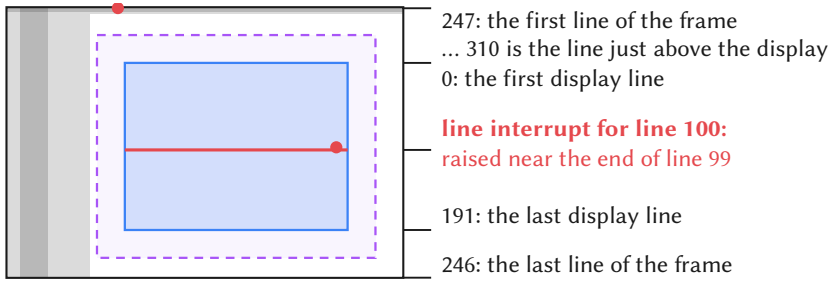


Figure 10.4: The line numbers of the line interrupt in 128K timing

The Active Video Line registers, NextRegs `$1E` and `$1F`, read the line the beam is on, in the same numbers: bit 8 in `$1E`, the low eight bits in `$1F`. They are two reads, and the beam can move on between them. So let me give you the safe way: read `$1E`, then `$1F`, then `$1E` again, and try again if the two `$1E` reads differ.

"Before the line" is meant literally. The interrupt for line n is raised on line $n - 1$, at dot 243 of its display area. From there to the start of line n's display area the beam still has the right border, the blanking and the left border to draw: a little over 100 T-states at 3.5 MHz, enough to set up a change that must be in place when line n begins.



KLIVE IDE Where the interrupt fires

Put a breakpoint at the line interrupt handler's first instruction. When it stops, the beam line on the paused screen shows where the beam really was: on the line before the one you asked for, a little past dot 243 of its display area.

Line numbers for the wide area

Line 0 suits the ULA and Layer 2 at 256 × 192. It does not suit the tilemap, the sprites and Layer 2's wide modes, which count their rows from the top of the **wide area**, 32 lines higher. A program that wants an interrupt before row r of the wide area has to ask for line $r - 32$, wrapped round the frame: a different number in every timing.

The Vertical Line Count Offset register, NextReg `$64`, removes the problem. It is an **offset** added to the line numbers. Set it to **32**, and line 0 becomes the first line of the

wide area, so the line numbers are exactly the rows of the 320 × 256 layers, in every timing (Figure 10.5). The offset applies to the line interrupt, the Active Video Line registers and the Copper. A new offset takes effect when the beam next reaches the first line of the display area, so it can take up to a frame.

	NextReg $64 = 0	= 32
first line of the frame	247	279
first line of the wide area	279	0
first line of the display area	0	32
last line of the display area	191	223
last line of the wide area	223	255
last line of the frame	246	278

Figure 10.5: The same frame, two numberings

The CPU reacts to a line interrupt only when its current instruction ends, a few T-states later or earlier from frame to frame. The Copper (Chapter 18) does the same job without that jitter, with the same line numbers.



KLIVE IDE Both numberings at once

The **Copper** panel in the **Debug** activity shows NextReg `$64` as **Line offset**, the raster line the Copper counts as its line 0, and the beam in the Copper's own numbering, `cvc`. Compare it with the raster line on the paused screen to see the offset at work.

▶ TRY IT Beam Clock

Part II, key 1. The demo sets the line interrupt for line 96 and prints the line's number. Q moves the line up and A moves it down, through all 311 lines of the 128K frame. One handler serves both interrupts and reads their status bits in NextReg `$C8`: the frame interrupt turns the border blue, the line interrupt red. The code is in `examples/code/demos/2-picture/beam-clock.kz80.asm`. It sets the line like this:

```
BcSetLine:
    ld hl,(BcLine)
    ld a,l
    nextreg $23,a           ; the line's low eight bits
    ld a,h                  ; bit 8 of the line...
    or %00000010            ; ...and bit 1: the line interrupt on
    nextreg $22,a
    ret
```

▶ TRY IT Beam Clock *(continued)*

Look at the right edge of the border: the red starts one line early there, because the interrupt for line n is raised near the end of line $n - 1$. Move the line past 191, and the change goes into the bottom border; past 246, into the top of the next frame.

Dots, ticks and T-states

The picture and the CPU run on one clock. The **28 MHz master clock** ticks four times for every dot, and the CPU's clock is made from it, at the speed set by the CPU Speed register, NextReg `$07`. So a T-state always lasts a whole number of ticks, and the CPU and the beam can never drift apart (Figure 10.6).

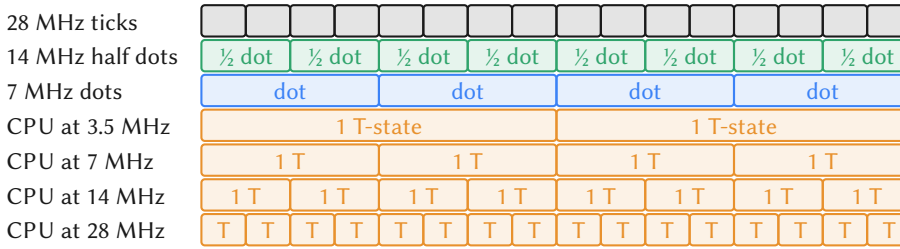


Figure 10.6: Sixteen ticks of 35.7 ns: four dots

What changes with the CPU speed is only how much your program can do in a line: 228 T-states in a 128K line at 3.5 MHz, 1,824 at 28 MHz. **Contention** exists only at **3.5 MHz**, in the pattern of the chosen timing, and bit 6 of the Peripheral 3 Setting register, NextReg `$08`, switches it off there too. Chapter 12 has the details.



KLIVE IDE How long until the beam gets there

Point at a dot on the paused screen, and Klive tells you how far away the beam is, in 7 MHz dots: `in 56,931 HC`. That is half as many T-states at 3.5 MHz and four times as many at 28 MHz: the time your code has left before the beam draws the dot.

From counters to cables

The video pipeline puts out a **9-bit colour** at 14 MHz, together with the blanking and the two sync signals. Its lines come at about 15.5 kHz, the line rate of a European television, and the Next has three ways out. The first is the simplest: with the **scan doubler off**, bit 0 of the Peripheral 1 Setting register, NextReg `$05`, clear, the picture

leaves exactly as the counters time it. A television or an old monitor with an RGB (SCART) input shows it the way it showed a Spectrum.

The scan doubler and VGA

A VGA monitor wants lines at about 31 kHz. The **scan doubler** gets there without changing the frame: it writes each line into a line buffer at 14 MHz and reads it out **twice** at 28 MHz, one line behind the original (Figure 10.7). I call it **the photocopier**.



Figure 10.7: Scan doubling (dashed: the copy that scanlines darken)

Bits 1–0 of the Peripheral 4 Setting register, NextReg `$09`, darken one of each pair of lines by 50%, 25% or 12.5%: **scanlines**.

Some monitors refuse the Next's 50 Hz frame. For them there are seven **VGA modes**, chosen in the Video Timing register, NextReg `$11`: modes 0 to 6 set the master clock from 28 MHz in steps up to 33 MHz. Everything speeds up together, the CPU, the beam, the frame rate, by up to 18%, until the monitor accepts it.

> **Faster, but not for the program.** In mode 3, at 30 MHz, the CPU's 28 MHz setting runs at 30 MHz, its 3.5 MHz at 3.75 MHz, and the frame rate rises to about 53.6 Hz. A program cannot tell: a frame has the same T-states, the beam reaches each line at the same T-state, and a CTC period is the same number of counts. Only time in seconds changes, so a game runs faster, and its music and samples play higher. NextReg `$11` is writable only in config mode, so the firmware sets it from your display settings; it is no way to overclock.

HDMI

HDMI carries fixed television standards, and the Next uses two, **720 × 576 at 50 Hz** and **720 × 480 at 60 Hz**. It takes a **window** of 360 × 288 dots around the wide area, 720 pixels of 14 MHz across, and sends every line twice (Figure 10.8). At 60 Hz the window is 360 × 240.

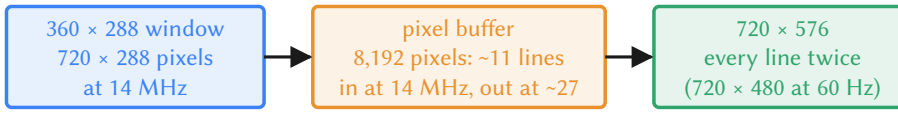


Figure 10.8: The HDMI path

The pixels pass through a buffer of about eleven lines. The trick that keeps it small: the HDMI clock is **made from the Next's own 28 MHz clock**, with a ratio chosen for each timing and frame rate, so that one HDMI frame lasts exactly as long as one Next frame. The two never drift apart, and the buffer never fills or runs dry. HDMI works in VGA modes 0 and 1 only.

HDMI also carries 8 bits per colour channel, not 3; Chapter 11 explains how the Next makes them.

In short. Two counters, one for dots and one for lines, time every event of the picture; the four timings change the frame's shape to the dot, and 60 Hz removes lines from the border. The frame interrupt comes before the display area, and the line interrupt counts its lines from the display area unless NextReg $64 moves the numbers. The CPU and the beam share one master clock, so they can never drift apart. RGB, VGA and HDMI all carry the same picture.

CHAPTER 11

Colours and Palettes: From an 8-Bit Index to a 9-Bit Colour

The Next can show 512 colours, yet not one of its layers can name a single one of them. Every layer speaks in 8-bit **indices**, and a palette turns each index into a colour only at the very last moment. Change one palette entry, and every dot that uses it changes colour at once. And one value, `$E3`, means "nothing here" almost everywhere. How can a colour mean "no colour"?

On an 8-bit CPU, the palette is the cheapest paintbrush you have: rewriting a picture costs thousands of bytes, and rewriting a palette entry costs one register write. The same mechanism decides what is transparent, so sooner or later every Next program meets it, usually at the moment a hole in its graphics stops being a hole.

Chapter 9 explains the layers and the pipeline the palettes sit in, including why the eight palettes live in two memories.

- How a colour fits in nine bits, and the 8-bit shortcut most programs use
- What the eight palettes are, which layer owns which, and how to write them
- Where each layer's colour index comes from
- Why the Next has two rules for transparency, and where `$E3` fits

Nine bits of colour

Every colour the Next shows is **RGB333**: three bits each of red, green and blue, levels 0 to 7, 512 colours in all. I think of a palette as **the wardrobe**: the layers never pick a colour themselves, they only say which costume number to wear.

Nine bits is one too many for an 8-bit machine. So most of the time your program writes colours in an **8-bit form**, `RRRGGGBB`, with only the upper two bits of blue. The hardware makes the missing lowest bit: it is **B1 OR B0** of the 8-bit value (Figure 11.1).

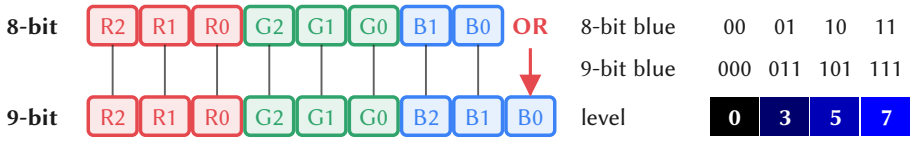


Figure 11.1: An 8-bit colour becomes a 9-bit one

Why OR, and not simply 0? With a 0, two blue bits would reach the levels 0, 2, 4 and 6, and never full blue. With the OR they reach **0, 3, 5 and 7**: no blue and full blue both, and two steps between them. One gate, and an 8-bit colour can be as bright as a 9-bit one.

Want all nine bits? Then write them in two goes, through the 9-bit Palette Value register, NextReg `$44`. The first write is the same `RRRGGGBB`, and the second carries the lowest blue bit in bit 0. Nothing is stored until the second write arrives. Bit 7 of the second write has another job, for Layer 2 only: the **priority bit** that lifts a Layer 2 dot above every other layer, whatever their order. My advice: use the 8-bit form until you need one of those two bits.



KLIVE IDE Palette files

Open a `.pal` or `.npl` file, and Klive edits its 256 colours on red, green and blue scales. Press `8` or `9` to work in the 8-bit or the 9-bit form, and Space to toggle an entry's priority bit, bit 7 of the second write.

Eight palettes

There are **eight palettes** of 256 entries: a first and a second one for each of the ULA (which LoRes shares), Layer 2, the sprites and the tilemap. Each layer shows one of its two at a time, and your program can edit any of the eight, shown or not (Figure 11.2).

	first	second	shown by	memory
ULA	000	100	$43 bit 1	1
Layer 2	001	101	$43 bit 2	2
sprites	010	110	$43 bit 3	2
tilemap	011	111	$6B bit 4	1

Figure 11.2: The eight palettes (codes: NextReg $43 bits 6–4, to edit)

Your program can prepare the second palette while the first is on screen, and then switch the whole layer to its new colours with one register write: a stage with two sets of lights, where the crew quietly rigs one while the audience sees the other.



KLIVE IDE The Next Palettes panel

The panel in the **Machine info** activity has a row for each layer's palettes. It marks the set the layer draws with now, chosen by NextReg `$43` (or `$6B` for the tilemap), and a dot after a layer's name says its palette was written since the last stop. For a closer look, **View › Machine Views › Palettes**, or `show-palettes`, shows a layer's two sets side by side and frames each entry written since the last stop, with the colour it replaced.

Writing and reading a palette

Bits 6–4 of the Palette Control register, NextReg `$43`, pick the palette. The Palette Index register, NextReg `$40`, sets the **index** of the first entry to write. Then each write to the 8-bit Palette Value register, NextReg `$41`, stores one 8-bit colour and moves the index on by one, unless bit 7 of NextReg `$43` stops it. Let's set four shades of red into Layer 2's first palette:

```
; Layer 2's first palette, entries 16-19: four reds
    nextreg $43,%00010000   ; Layer 2 palette 1, index moves
    nextreg $40,16          ; start at entry 16
    nextreg $41,%01000000   ; red 2
    nextreg $41,%10000000   ; red 4
    nextreg $41,%11000000   ; red 6
    nextreg $41,%11100000   ; red 7
```

With NextReg `$44` it works the same way, with two writes per entry; a write to NextReg `$40`, `$41` or `$43` in between restarts the pair, so your program can never leave half a colour behind. Reading NextReg `$41` gives the upper eight bits of the entry at the index, and reading NextReg `$44` its second byte. Reads never move the index on.



KLIVE IDE Catching a colour change

`bp-set pal:spr1:$03` stops the machine after the instruction that writes entry `$03` of the first sprite palette, by the CPU or the DMA, through any register or port. `pal:l2:$00-$0f` watches a range, `pal:any` every entry, and `pal:tm:live` a flip of the tilemap's live set. Add `-changed` to skip writes of the same colour, and `-c` to catch the Copper's writes too. The stop message names the old colour, the new one, and the address of the writing instruction.

The ULA: three maps of one palette

Every layer makes its 8-bit index in its own way, and the ULA's way is the oddest. A bitmap bit says "ink" or "paper", and the cell's attribute says which colours those are. The Next can read the same attribute byte three ways (Figure 11.3):

- **Standard**, as on the Spectrum: INK uses entries 0–7, BRIGHT INK 8–15, PAPER 16–23 and BRIGHT PAPER 24–31. The border uses the PAPER entries. Change these 32 entries, and old software keeps working, in new clothes.
- **ULANext** (bit 0 of NextReg `$43`): the attribute byte is split by a **mask** in the ULANext Attribute Byte Format register, NextReg `$42`. The bits the mask covers are the INK, an index from 0 upward; the rest are the PAPER and border, from 128 upward. A mask of `$07` gives 8 inks and 32 papers, and `$FF` gives 256 inks and takes paper and border from the fallback colour. FLASH and BRIGHT are gone.
- **ULA+** (bit 3 of the ULA Control register, NextReg `$68`, or ULA+'s own ports): 64 colours in entries 192–255. FLASH and BRIGHT choose one of four groups of 16, each with 8 inks and 8 papers. ULA+ software writes port `$FF3B` in its own bit order, green first; the Next reorders that into `RRRGGGBB`.

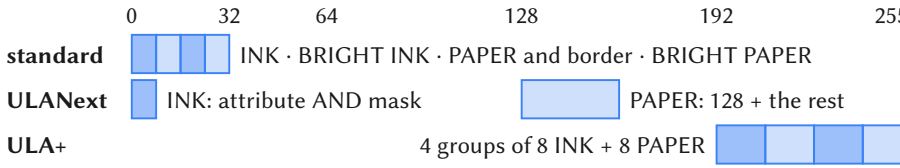


Figure 11.3: The ULA palette entries each mode uses (ULANext with mask $07)

For new code I would pick ULANext: it frees the most colours from one attribute byte. Chapter 12 has the details.

The other layers: a dot and an offset

The other layers have real pixels, here called **dots**, and each layer turns a dot into a palette **index**. Think of an index as two hex digits, and of the palette as **16 rows of 16**: the first digit is the row, the second the column. Entry `$59` is in row 5, column 9.

An **8-bit dot** is a whole byte, two hex digits, a complete index. Layer 2's dots at 256 × 192 and 320 × 256, and 8-bit sprite patterns, are 8-bit. A **4-bit dot** is half a byte, one hex digit: 4-bit sprite patterns, tiles, Layer 2's 640 × 256 mode and Radastan store their dots this way. They save half the memory, but a single digit can name only 16 entries.

The palette offset fills the gap. It is one more hex digit that each of these layers keeps beside its dots (Figure 11.4):

- **A 4-bit dot** has no first digit, so the offset **becomes the first digit**. Offset 5 and dot 9 make index `$59`: a 4-bit sprite with offset 5 uses row 5, and with offset 6 the same pattern uses row 6, in a different set of 16 colours.
- **An 8-bit dot** already has both digits, so the offset is **added to its first digit**. Layer 2 byte `$A8` with offset 1 shows entry `$B8`, one row down. If the first digit passes `F` it wraps round to `0`: `$F3` with offset 1 shows `$03`. With offset 0, nothing changes.

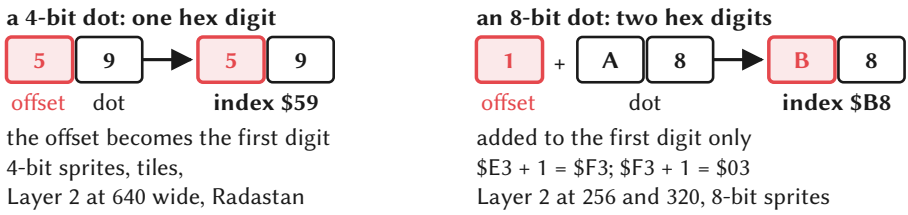


Figure 11.4: The two offset rules

Layer	Dots	Offset	Index
Layer 2 at 256 or 320	8-bit	NextReg `$70` bits 3–0	added to first digit
Layer 2 at 640	4-bit	NextReg `$70` bits 3–0	offset, then dot
Sprites, 8-bit	8-bit	each sprite's attributes	added to first digit
Sprites, 4-bit	4-bit	each sprite's attributes	offset, then dot
Tilemap	4-bit	each tile's attribute	offset, then dot
LoRes	8-bit	NextReg `$6A` bits 3–0	added to first digit
Radastan	4-bit	NextReg `$6A` bits 3–0	offset, then dot

Keep one consequence in mind: an offset moves **every** dot, and for an 8-bit layer that includes the byte your program uses for "nothing here".

Two rules for transparency

Chapter 9 says that every layer answers "am I transparent here?" for every dot. Here is how the answer is made, and if you read only one section, I would make it this one (Figure 11.5).

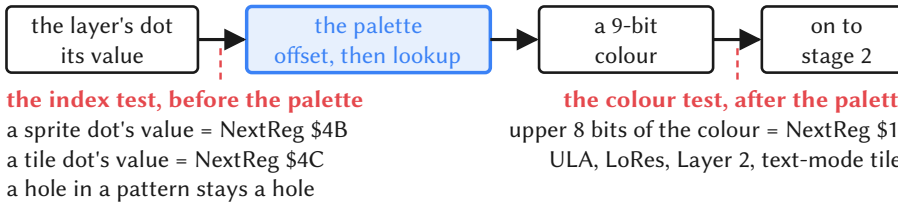


Figure 11.5: The two transparency tests

The **index rule** makes a hole in a shape. A **sprite** dot is transparent when its pattern value equals the value in the Sprite Transparency Index register, NextReg `$4B`, and a **tile** dot when its 4-bit value equals the value in the Tilemap Transparency Index register, NextReg `$4C` (for 4-bit sprites, the lower four bits of NextReg `$4B`). The test is made **before** the palette offset is applied, so the hole stays a hole whichever offset a sprite or tile is drawn with, and whatever the palette holds.

The **colour rule** makes a colour that means nothing. The **ULA**, **LoRes**, **Layer 2**, and tiles in text mode are transparent where their **colour**, after the palette, equals the **global transparency colour** in NextReg `$14`. It is compared with the upper eight bits of the 9-bit colour, so the lowest blue bit does not matter. Because the test is made on the colour, it has consequences, and every one of them will surprise you at least once:

- **Any index can be transparent**: every entry that holds the colour in NextReg `$14`.
- **A palette change can make dots appear or vanish.** Give an entry the colour in NextReg `$14` and its dots disappear; give the transparent entry another colour and the "empty" parts of the layer suddenly show.
- **The offset moves transparency too.** With an offset the "empty" byte `$E3` points at another entry, with another colour.
- **Changing NextReg $14 changes every layer that uses it.** That colour vanishes from the ULA, LoRes and Layer 2 alike, while every dot that was `$E3` becomes visible.



KLIVE IDE Transparency in the side bar

Expand the sprite or the tilemap palette in the **Next Palettes** panel, and it marks the entry its transparency index points at. The Palette document marks transparent entries for every layer: for the ULA and Layer 2, each entry whose colour matches NextReg `$14`. A `.npl` palette file can carry a transparency colour too: in the palette editor, `T` sets it and `C` clears it.

$E3: the colour of nothing

NextReg $14, NextReg $4B and the fallback colour all start at $E3, while NextReg $4C starts at $F. $E3 is 111 000 11, a bright magenta, a colour a picture rarely wants. In the default Layer 2 and sprite palettes, entry $E3 holds the colour $E3, so out of the box a Layer 2 byte $E3 and a sprite pattern value $E3 are both invisible, by two different rules that happen to agree.

Pure bright magenta would be the colour $E3 itself, and every bright magenta dot of the Spectrum screen would be a hole. So the default ULA palette gives bright magenta one step of green.

The fallback colour

When every layer answers "transparent" for a dot, the picture takes the **fallback colour**, from the Fallback Colour register, NextReg $4A.

The colours at power-on

The palette memories have no reset: it is the firmware that fills them at power-on. So a program that relies on a particular palette should write it.

- The **ULA palettes** get the Spectrum's colours (Figure 11.6). The same 16 colours repeat in every block of 16 entries, so PAPER has the same colours as INK.
- The **Layer 2, sprite and tilemap palettes** get the **identity**: entry *n* holds the 8-bit colour *n*.

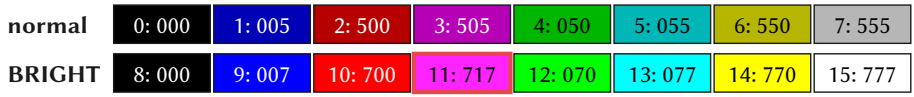


Figure 11.6: The default ULA palette: red, green, blue levels (ringed: bright magenta)

What palettes make cheap

A palette sits between the picture's memory and the screen, so changing it changes the screen without touching the memory:

- **Palette cycling.** Rotate the colours of a run of entries, and water flows while the bitmap stands still.
- **Fades.** Darken every entry a step at a time, at the cost of 256 writes per palette.
- **Recolouring.** One sprite pattern in eight colourings needs eight offsets, not eight patterns.

- **Changing it mid-frame.** A palette entry rewritten at every line gives the sky gradients of Next games. The **Copper** of Chapter 18 is the best tool for that: it costs the CPU nothing, and it is on time.

Figure 11.7 shows the scene of Chapter 9 four times, with the same Layer 2 bitmap. The second picture rewrites entry `$A8` (NextReg `$43` = `$10`, `$40` = `$A8`, `$41` = `$12`). The third sets palette offset 1 (NextReg `$70` = `$01`). The fourth gives entry `$A8` the colour `$E3`, and only the hills' edge, another entry, remains.

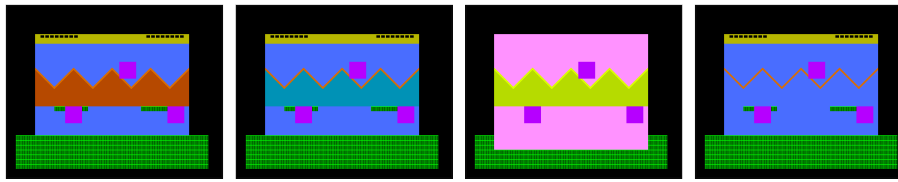


Figure 11.7: One bitmap, four pictures

Look at the third picture: it is the colour rule's trap in action. The offset turned the hills from `$A8` into `$B8`, as wanted, but it also turned every transparent `$E3` byte into `$F3`, an opaque pink, and Layer 2 now covers the layers below. A program that gives Layer 2 an offset has to give the entry its "empty" byte now lands on the transparency colour too.

▶ TRY IT Palette Cycle

Part II, key 2. The demo fills sixteen rows of the ULA screen with diagonal stripes in BRIGHT INK 1 to 7, which are entries 9 to 15 of the ULA palette. Then it never touches the screen again: every fourth frame it rotates seven blues through those seven entries, and the water flows. The code is in `examples/code/demos/2-picture/palette-cycle.kz80.asm`. One step is seven register writes:

```
    ; Rotate the table of seven colours by one...
    ld hl,PcColours
    ld a,(hl)
    ld de,PcColours
    inc hl
    ld bc,PcCount - 1
    ldir
    ld (de),a
    ; ...and write it to entries 9-15: seven register writes, no pixels
    nextreg $43,%00000000   ; ULA palette 1, the index moves on
    nextreg $40,PcFirst
```

▶ TRY IT Palette Cycle *(continued)*

```
    ld hl,PcColours
    ld b,PcCount
PcWrite:
    ld a,(hl)
    nextreg $41,a           ; RRRGGGBB
    inc hl
    djnz PcWrite
    ...
; Seven blues, dark to light and back, in RRRGGGBB
PcColours:
    .defb $01, $02, $03, $13, $7F, $13, $03
```

In Klive IDE, open the ULA palette with `show-palettes ula` and pause the machine: entries 9 to 15 are framed as written, each with the colour it replaced, while the screen's memory stays the same. Before it returns, the demo writes back the seven entries it found, all nine bits of each, through NextReg `$44`.

From palette to screen

The VGA and RGB outputs have three bits a channel, so the colour goes out as it is. The Issue 5 board's VGA output has a converter of eight bits a channel, but the core feeds it the same three bits, repeated as for HDMI below, so it shows the same 512 colours and not one more. HDMI carries eight bits a channel, and the Next makes eight from three by **repeating the bits**: `101` becomes `10110110`. It gives the same result as scaling by 255/7 and rounding.

In short. Layers name colours by 8-bit index, and eight palettes of 256 entries turn those indices into 9-bit colours, so one register write can recolour a whole picture. A 4-bit dot takes its first hex digit from a palette offset; an 8-bit dot has the offset added to it. Sprites and tiles test transparency on the dot before the palette, while the ULA, LoRes and Layer 2 test the colour after it, so an offset can make an empty `$E3` byte suddenly visible. A program should write any palette it relies on.

CHAPTER 12

The ULA Layer: The Spectrum's Screen, Extended

Load a 48K program on your Next, and it sees exactly the screen it expects: the same bytes at `$4000`, the same colours, the same border stripes, the same FLASH. Yet the very same ULA can put 512 dots on a line, give every 8 × 1 dots their own colours, show 64 colours at once, and scroll the whole screen smoothly by a single dot. So where does all that hide, and why does old software never trip over it?

Every extension sits behind a switch that old software never touches. Learn the switches, and you get the familiar screen with its worst limits taken away: colour clash, eight colours per cell, and scrolling by moving thousands of bytes.

This chapter builds on the Spectrum's own ULA, its bitmap, attributes and famous address order, on Chapter 9 for the layers and the pipeline, and on Chapter 11 for how an index becomes a colour.

- Why the Next imitates contention, and what the ULA's extra memory reads are for
- How the Timex modes give a second screen, hi-colour and hi-res
- How ULANext and ULA+ read the attribute byte in new ways
- How the hardware scroll and the clip window work

The same ULA, new memory

Leave every new switch alone, and the Next's ULA is the Spectrum's: the 256 × 192 bitmap at `$4000`, the 768 attributes at `$5800`, INK, PAPER, BRIGHT and FLASH, and the border from port `$FE`, read at the same moments of the frame.

What differs is the memory behind it. On a Spectrum, the ULA and the CPU take turns at one set of RAM chips, and the ULA always goes first, because the beam does not wait. The CPU is held up until it has finished. That is **contention**, and it follows a fixed pattern of delays across each line of the picture.

The Next has no such limit. **Bank 5** is kept in **block RAM inside the FPGA**, and so is the first half of **bank 7**, the 128K's shadow screen (Figure 12.1). Block RAM has **two**

ports: the CPU uses one, the ULA (and the tilemap) the other, and neither ever waits for the other. If you ask me, this is the change that makes every other one possible.

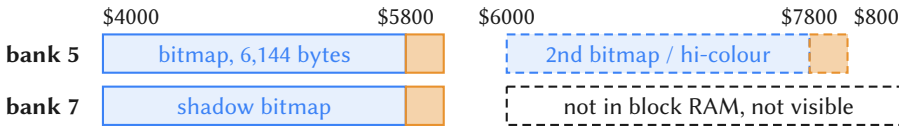


Figure 12.1: The memory the ULA reads, to scale (orange: attributes; dashed: Timex only)

Contention is an imitation. Old programs count T-states with the delays built in, so the ULA works out, from the beam's position, exactly when a Spectrum's CPU would have been held up, and holds the Next's CPU up then, only while it runs at 3.5 MHz. The veteran limps on cue, because the old programs expect the limp. And there is **no snow**: here the ULA has a port of its own.

> **KLIVE IDE Watching the limp**
>
> The **ULA & I/O** panel in the **Debug** activity counts the T-states the CPU lost to contention: `CON` since the machine started, `LCO` since the last pause. Run code that reads the screen memory from one pause to the next: `LCO` grows at 3.5 MHz and stays 0 at 7 MHz.

Bank 7 is only half there. The 8K in block RAM is plenty for the 128K shadow screen, but not for a second display file at `+$2000`. So while the shadow screen is shown, the Timex modes are switched off. If you want them, keep the picture in bank 5.

Reading a line twice

The Spectrum's ULA reads four bytes for every 16 dots. The Next's reads **eight**: for every 8 dots it fetches the bitmap byte and attribute of the cell under the beam **and of the cell after it**. The reason is the hardware scroll: with a scroll of 3 dots, the 8 dots the beam is about to draw are the last 5 dots of one cell and the first 3 of the next (Figure 12.2).

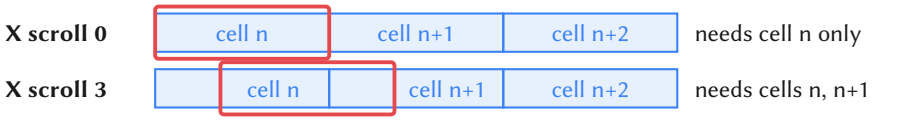


Figure 12.2: The 8 dots on screen (red): one cell, or two with a scroll

The ULA always fetches both, scroll or not (Figure 12.3).



Figure 12.3: Eight reads where the Spectrum made four (dashed: the neighbours)

The extra reads are invisible to your programs: the **floating bus** shows only the four reads a Spectrum would make, at the moments it would make them. With bit 2 of the Peripheral 3 Setting register, NextReg `$08`, a read of port `$FF` returns the Timex register of the next section instead.



KLIVE IDE The floating bus, paused

The ULA & I/O panel's `FLO` shows what a read of port `$FF` would return at the beam's position: a bitmap or attribute byte in the display area, `$FF` elsewhere. With bit 2 of NextReg `$08` set, it shows the Timex register instead, as the hardware does.

The Timex modes

The Timex Computer Corporation sold Spectrum-compatible machines in America whose ULA had more modes, switched by **port `$FF`**. I call them **the American cousins**: family of the Spectrum's screen, raised abroad.

Bits of port `$FF`	Meaning
7	not used
6	1 switches off the frame interrupt
5–3	ink colour of the hi-res mode (the paper is its opposite)
2–0	the screen mode: `000` standard, `001` second screen, `010` hi-colour, `110` hi-res

Bits 5–0 can also be written through the Display Control 1 register, **NextReg `$69`**, whose bit 7 switches Layer 2 on and bit 6 shows the shadow screen. The mode is taken again for every 8 dots, so your program, or the Copper, can change it between two cells of the same line.



KLIVE IDE The Timex port in the side bar

The **Next Memory Mapping** panel has a row for port `$FF`, **Port Timex**, with the last value written to it: bits 2–0 the screen mode, bits 5–3 the hi-res ink colour. Pause anywhere, and you see which mode the ULA is in.

A second screen

Mode `001` reads the same Spectrum screen from **`$6000`** instead of `$4000`, with its attributes at `$7800`. A program draws the next frame in one screen while the other is shown, and switches with one `OUT`: **double buffering**. Of the American cousins, I find this one the most useful day to day.

Hi-colour: a colour for every 8 × 1 dots

Mode `010` gives every **byte of the bitmap** a colour byte of its own: the colour of the bitmap byte at `$4000 + o` is the byte at **`$6000 + o`**. So the ULA's address for the colour is the bitmap address with one more bit set, where the Spectrum's attribute address drops the line's three low bits, which is what made the 8 × 8 cells in the first place (Figure 12.4).

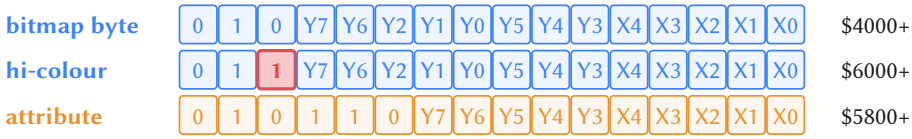


Figure 12.4: Three addresses the ULA builds, bit 15 on the left (Y: line, X: byte column)

The colour byte is an ordinary attribute in every other way. Figure 12.5 shows the same bitmap twice, a ball striped with a new colour on every line.

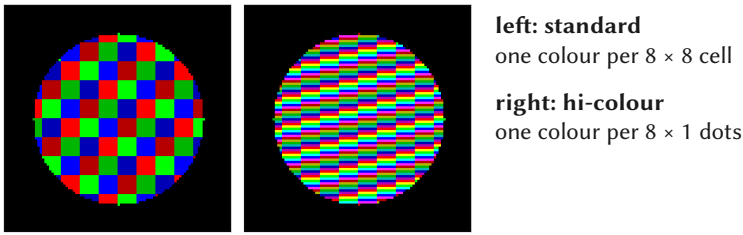


Figure 12.5: The same bitmap in the two modes, 96 × 96 dots

The price is memory: 12K for one screen instead of 6.75K, and 6,144 colour bytes to write where there were 768. The gain is that colour clash is gone from the vertical direction.

Hi-res: 512 dots across

Mode `110` is the other way round: one colour, twice the dots. A **column** here is a strip **8 dots wide**: on every line it is **one byte** of the bitmap, and with an 8 × 8 font it holds **one character**. A hi-res line is 64 bytes, twice what one display file holds for a line, so **both** files hold half each:

	Standard	**Hi-res**
Dots on a line	256	512
Width of a dot	one 7 MHz dot	one 14 MHz dot
Columns of 8 dots	**32**	**64**
Where column *c* is	byte *c* of the line at `$4000`	byte *c* ÷ 2 of the line, at `$4000` if *c* is even, at `$6000` if odd

The eight reads pay off a second time: the four "attribute" reads per 8 dots fetch from the second file's bitmap instead, so the ULA gets the bytes of both files with no extra reads at all, and the shift register sends each bit once instead of twice.

The whole screen has **one ink and one paper**, set by bits 5–3 of port `$FF`. The ink is that colour, BRIGHT; the paper is its opposite, also BRIGHT: `000` black on white, `001` blue on yellow, and so on to `111` white on black (Figure 12.6).

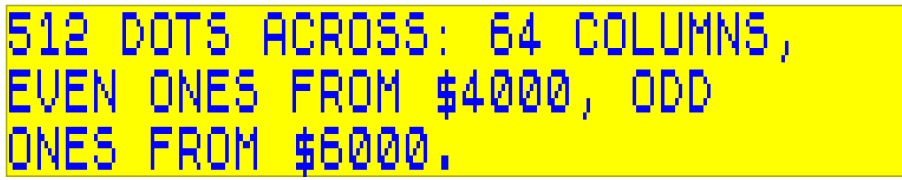


Figure 12.6: Hi-res text on the machine, the left half of the screen

It costs the same 12K as hi-colour.

ULANext: the attribute byte split anew

ULANext keeps the byte and the bitmap and changes only how the byte is read. Bit 0 of the Palette Control register, **NextReg $43**, switches it on, and the ULANext At-

tribute Byte Format register, **NextReg $42**, holds a **mask**: the bits the mask covers are the INK, the rest are the PAPER (Figure 12.7). For an ink dot, the index is the attribute **AND** the mask. For a paper dot, it is the bits above the mask, moved down, plus **128**.

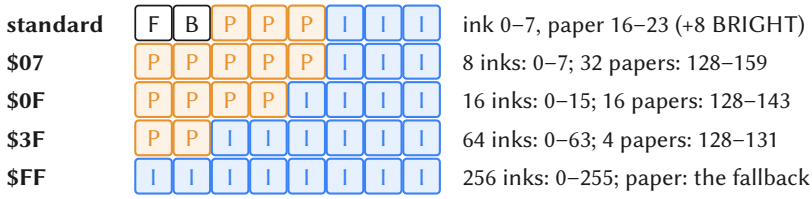


Figure 12.7: One attribute byte, four ULANext masks (I: ink, P: paper)

The mask must be a run of ones from bit 0, and each step doubles the inks and halves the papers:

Mask (NextReg $42)	Inks	Papers	Border entries
`$01` to `$7F` (`$07` after reset)	2 to 128	128 to 2	128–135
`$FF`	256	fallback colour	fallback colour
any other value	attribute AND mask	fallback colour	128–135

`$FF` is the **full-ink** mode: a cell can have any of 256 colours, and its paper dots show the Fallback Colour register, NextReg `$4A`. ULANext cannot change the bitmap, so a cell is still 8 × 8 dots with one ink and one paper, and colour clash remains. With hi-colour it does not: ULANext reads the hi-colour colour bytes the same way.

ULA+: 64 colours

ULA+ is a standard defined for Spectrum clones and add-ons, and the Next implements it so that their software runs. I think of it as **the guest**: welcome in the house, but it brought its own manners. Its FLASH and BRIGHT bits together pick one of **four groups** of 16 colours, each with 8 inks and 8 papers, in entries **192–255** of the ULA palette (Figure 12.8).

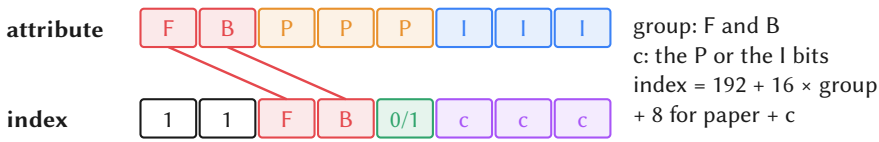


Figure 12.8: ULA+: from an attribute byte to an index (green: 0 for an ink dot, 1 for a paper dot or the border)

Software writes its colours through two ports. **Port `$BF3B`** selects what **port `$FF3B`** does: with bits 7–6 of the value `00`, bits 5–0 choose one of the 64 colours, and the next write to port `$FF3B` sets it; with `01`, bit 0 of the next write switches ULA+ on or off. A Next program can also switch ULA+ with bit 3 of the ULA Control register, NextReg `$68`, and write entries 192–255 through the ordinary palette registers.

The border uses entries **200–207**. If both ULA+ and ULANext are on, ULANext wins. And neither of them has FLASH: if your program needs it, stay with the standard map.

▶ TRY IT ULA Modes

Part II, key 3. Keys 1–5 show bank 5 in five ways: standard, hi-colour, hi-res, ULANext with the mask `$0F`, and ULA+. Keys 1 and 2 show the same bitmap. The first takes one colour for each 8 × 8 cell from `$5800`, the second one for each 8 × 1 dots from `$6000`. The code is in `examples/code/demos/2-picture/ula-modes.kz80.asm`; this loop writes a bitmap byte and its colour byte, 8K higher:

```
UmBandByte:
    ld (hl),$F0
    ld a,c
    and 7
    ld e,a                  ; the ink
    xor 7                   ; the paper
    add a,a
    add a,a
    add a,a
    or e
    or $40                  ; BRIGHT
    set 5,h                 ; the colour byte: 8K higher
    ld (hl),a
    res 5,h
    inc l
    inc c
    djnz UmBandByte
```

Key 3 prints 64 columns in blue on yellow, and reads no attribute at all. Keys 4 and 5 take their colours from the ULA's second palette, which the demo fills and puts back when it ends. Pause the machine in Klive IDE, and its side bar shows that palette and NextRegs `$42`, `$43` and `$68` as the mode left them.

The border in every mode

The border is still three bits of port `$FE`. What the Next adds is the choice of palette entry:

Mode	The border's palette entry
Standard (and the Timex second screen and hi-colour)	16–23: the PAPER entries, never BRIGHT
ULANext	128–135; with the mask `$FF`, the fallback colour
ULA+	200–207
Hi-res	the hi-res paper colour: port `$FE` is not used

A new border colour reaches the screen at the start of the next 8-dot cell, as on the Spectrum; with Pentagon timing, from the very next dot. And because the border is a ULA dot like any other, a border whose colour matches NextReg `$14` is transparent, as Chapter 17 explains.

Scrolling by the dot

The ULA has two offsets: the ULA X Scroll register, **NextReg $26**, 0–255 dots, and the ULA Y Scroll register, **NextReg $27**, 0–191 lines. The picture **wraps round**: across at 256, down at 192.

What happens to the **colours**? A program that scrolls the Spectrum's screen by one dot moves the bitmap and leaves the attributes. The ULA's scroll moves both. It loads both cells into a shift register, shifts them left by twice the scroll's low three bits, plus the half-dot bit, and switches from the first cell's attribute to the second's where the cell edge falls (Figure 12.9).

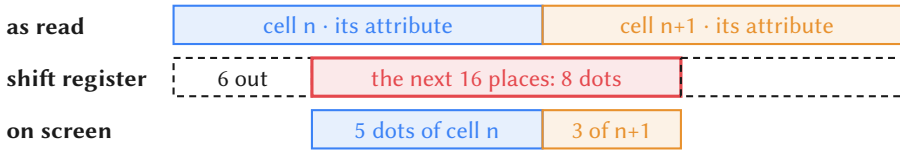
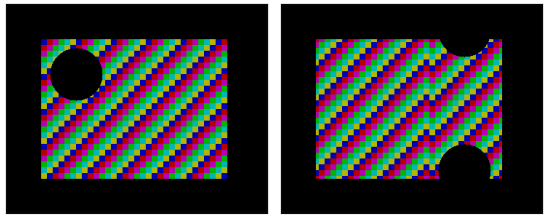


Figure 12.9: X scroll 3 inside 8 dots: 2 × 3 = 6 places out

So every dot keeps its own cell's colours, wherever the scroll puts it (Figure 12.10). The border does not scroll.



left: no scroll

right: NextReg $26 = 100, NextReg $27 = 60

Figure 12.10: A scroll on the machine

Half a dot

Bit 2 of the ULA Control register, **NextReg $68**, the **half-dot scroll**, moves the picture half a dot further left, one 14 MHz dot. In hi-res it is what makes the scroll exact: NextReg $26 moves a hi-res picture two dots at a time, and the half-dot bit adds the one between.

The ULA takes both offsets, the half-dot bit and the Timex mode once for every 8 dots, so a new value written in the middle of a line shows from the next cell on.

Clipped and switched off

The ULA's clip window, **NextReg $1A**, works in display-area coordinates, X 0–255 and Y 0–191. A clipped dot is not paper, it is **transparent**, and the **border is never clipped**.

Bit 7 of NextReg $68 switches the ULA **off**: it is then transparent everywhere, border included. The blend modes of Chapter 17 still read its colours.

All of the ULA's switches:

Register	Bits	What it does
port $FE	2–0	the border colour
port $FF, NextReg $69	5–3, 2–0	hi-res ink; the Timex mode
port $7FFD, NextReg $69 bit 6	3	the shadow screen in bank 7
NextReg $26, $27	7–0	the X and Y scroll
NextReg $68	7, 3, 2	ULA off; ULA+ on; half-dot scroll
NextReg $43	0	ULANext on
NextReg $42	7–0	the ULANext ink mask
ports $BF3B, $FF3B		ULA+ colours and on/off
NextReg $1A		the clip window

In short. The Next's ULA keeps the Spectrum's screen bit for bit, imitating contention at 3.5 MHz, and hides its extensions behind switches old software never touches. Port $FF selects the Timex modes: a second screen, a colour for every 8 × 1 dots, or 512 dots across. ULANext and ULA+ read the attribute byte in new ways and spend FLASH and BRIGHT on colour. The hardware scroll moves the bitmap and its colours together, one dot at a time.

CHAPTER 13

LoRes and Radastan: Big Pixels in the ULA's Place

A 128 × 96 screen sounds like a step backwards. The Spectrum's own screen has four times as many dots, so why would anyone want fewer? Because on the Spectrum every 8 × 8 cell of the screen can show only two colours, and you have to draw your picture around that. On a LoRes screen, **every pixel can be any of 256 colours**, the whole screen fits in **12K**, and it scrolls in both directions without your program copying a single byte. I would trade a few dots for that any day.

Think of a game or a demo that wants colour more than detail, on a CPU that has to redraw the screen 50 times a second. Fewer, fatter pixels, each in a colour of its own, suit it very well. The Next has two such modes. **LoRes** stores a byte per pixel. **Radastan**, a mode first made for the ZX-Uno, stores 4 bits per pixel: 16 colours in half the memory. Neither is a layer of its own. Switch one on, and it takes the ULA's place. That is why I call LoRes **the understudy**: it wears the veteran's costume, stands on the veteran's mark, and the rest of the machine never notices the swap.

This chapter builds on Chapter 9 for the layers, Chapter 11 for the palettes, and Chapter 12 for the ULA's clip window and border.

- What a LoRes pixel is, and where its bytes live in bank 5
- How a byte becomes a colour, and how Radastan fits 16 colours in 6K
- Why LoRes is the ULA in disguise
- How to scroll by half a pixel, and what a full redraw costs

A pixel is four dots

You switch LoRes on with bit 7 of the Sprite and Layers System register, **NextReg $15**. Its screen is **128 pixels across and 96 down**, and it fills the same **display area** as the Spectrum's screen, 256 × 192 dots. So each LoRes pixel covers **2 dots across and 2 lines down**, four dots of one colour. I like to think of it as a mosaic laid with big square stones, and close up the stones are plain to see (Figure 13.1).

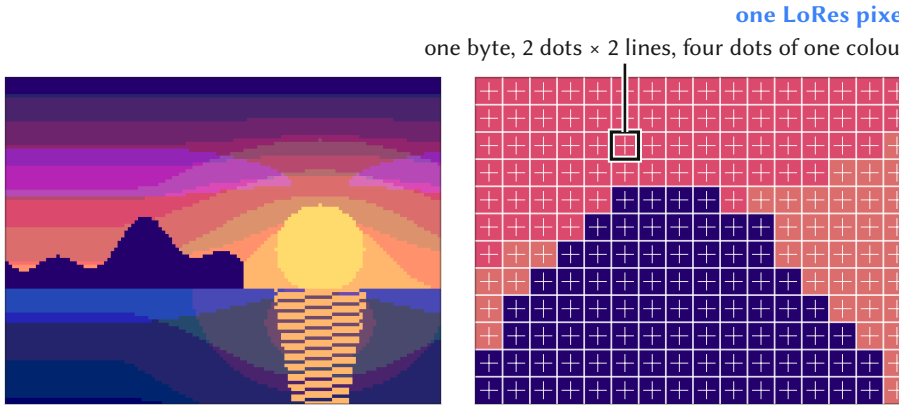


Figure 13.1: A LoRes picture and its four-dot blocks; dashed lines show the dots in each

Where the bytes are

LoRes keeps its picture in **bank 5**, the bank that holds the Spectrum's screen, in two halves of 6,144 bytes (Figure 13.2).

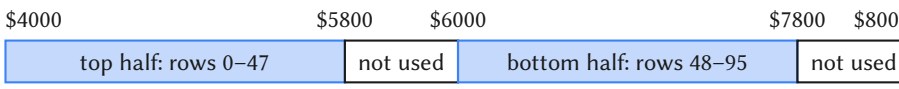


Figure 13.2: Where LoRes keeps its 12K

- The **top half**, rows 0–47, is at `$4000`–`$57FF`.
- The **bottom half**, rows 48–95, is at `$6000`–`$77FF`.
- Inside a half the rows are stored one after the other, **128 bytes a row**, left to right. Pixel (x, y) of the top half is at `$4000` + $128y + x$; of the bottom half, at `$6000` + $128(y - 48) + x$.

Did you spot what is missing? The scrambling. Unlike the ULA's bitmap, the next byte is the next pixel, and the next 128 bytes are the next row. If you have ever written a Spectrum plot routine, you will know why I find that a relief. The gap between the halves is where the ULA keeps its attributes, at `$5800`, and the two halves are exactly where the Timex modes keep their two bitmaps.

LoRes always reads bank 5, even while a 128K program shows the shadow screen in bank 7. It reads one byte for every dot through the CPU's port of the block RAM, and the CPU goes first. Each dot has time for both, so LoRes never delays the CPU. The ULA keeps running underneath, with only its picture replaced, so contention is the

same with LoRes on or off. My advice: if your program counts on contention timing, you can switch LoRes on without recounting a thing.

From a byte to a colour

A LoRes byte is a **palette index**. It goes through the **ULA's palette**, the first or the second one as bit 1 of the Palette Control register, **NextReg $43**, selects, and so any of the 256 entries can be used. The ULA's ways of reading an attribute, ULANext and ULA+, do not apply: a LoRes byte is never an attribute. No ink, no paper, no bright, no flash. I find that refreshingly simple.

The LoRes Control register, **NextReg $6A**, holds a **palette offset** in bits 3–0. It is added to the byte's high nibble before the lookup, as for an 8-bit Layer 2 dot: offset 1 turns byte $25 into index $35. With offset 0, the byte is the index.

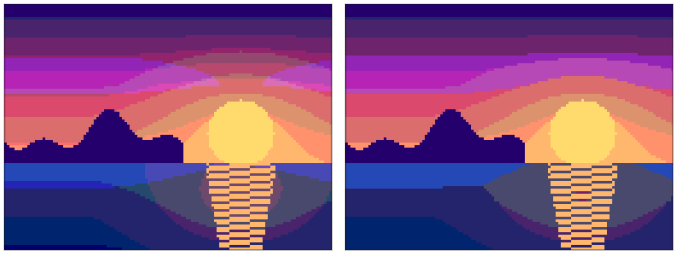
Radastan: 16 colours in 6K

Bit 5 of the LoRes Control register, **NextReg $6A**, switches LoRes to **Radastan mode**. The screen is still 128 × 96 pixels of 2 × 2 dots, but each pixel is **4 bits**, so a byte holds two of them and the screen takes half the memory. Radastan is the understudy on a budget: same stage, same mark, half the wardrobe.

- A row is **64 bytes**, with the left pixel of a pair in the byte's high nibble and the right one in its low nibble. The 96 rows follow one another with no gap: **6,144 bytes**, in one block.
- The pixel's 4 bits are the low half of the palette index. The high half is the palette offset, NextReg $6A bits 3–0, so the offset picks which **16 entries** of the ULA palette the picture uses.
- With **ULA+** switched on (the ULA Control register, NextReg $68, bit 3, or port $FF3B) and ULANext off, the high half is 11 followed by the offset's bits 1–0, so the picture takes one of the four 16-colour groups of the ULA+ palette, entries 192–255.

Here is the same sunset in both modes, so you can judge for yourself (Figure 13.3). Radastan keeps the 16 most common colours of the LoRes picture, and only the glow and the sea lose their finest steps. For half the memory, I call that a bargain.

At 6K a screen, bank 5 holds two, at $4000 and at $6000: **display file 0** and **display file 1**. The one shown is picked by bit 0 of the Timex control port, **port $FF**, the bit that picks the Timex modes' second display file, exclusive-or'ed with bit 4 of NextReg $6A. That makes double buffering simple: draw the next picture into the screen that is not shown, then flip one bit. The XOR bit lets your program leave port $FF the way other code expects and still choose either screen, a courtesy I like in a design. Plain LoRes does not look at port $FF: its two halves are one screen.



left: LoRes
12K, 256 colours

right: Radastan
6K, 16 colours

Figure 13.3: The same picture in both modes

In the ULA's place

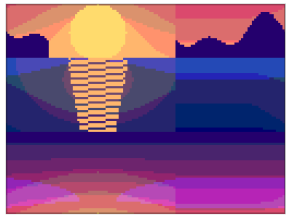
LoRes is not a sixth layer. When bit 7 of the Sprite and Layers System register, NextReg $15, is set, LoRes's dot **replaces the ULA's** on its way into the ULA layer, and from then on the machine cannot tell the two apart. This is the understudy's whole act: the ULA's costume, the ULA's mark on the stage, the ULA's cues.

- **The clip window.** LoRes has no clip window of its own; it uses the ULA's, the Clip Window ULA register, **NextReg $1A**, in display-area coordinates. Outside the window the ULA layer is transparent, and whatever is below shows through.
- **The border** is the ULA's: LoRes covers only the display area, and the border around it keeps the colour port $FE gives it.
- **Transparency**: a LoRes dot whose colour equals the Global Transparency Colour register, **NextReg $14**, is transparent, like a ULA dot (the section Two rules for transparency in Chapter 11).
- **Switching off**: bit 7 of the ULA Control register, **NextReg $68**, which switches the ULA off, switches LoRes off with it.
- **Layer order, stencil and blending** treat the LoRes picture as the ULA's (Chapter 17).

I think this is the neatest decision in the mode. Every rule you already know about the ULA layer holds for LoRes, so there is nothing new to learn about where it sits among the layers.

Scrolling by half a pixel

LoRes has its own scroll registers: the LoRes X Scroll register, **NextReg $32**, 0–255, and the LoRes Y Scroll register, **NextReg $33**, 0–191. They count in **dots and lines**, not in LoRes pixels, so one step is half a pixel, and a scroll moves smoothly. The picture **wraps round** at 256 dots across and 192 lines down: what leaves on one side comes back on the other, and your program never copies a byte to scroll. I am a big fan of this. Fat pixels usually mean jerky scrolling, and here they do not (Figure 13.4).



NextReg $32 = 101, NextReg $33 = 75

The sun has moved left and up.
The hills come in again from the right.
The top of the sky appears below the sea.

Figure 13.4: A scrolled LoRes picture

The two kinds of scroll keep to themselves: the ULA's own scroll registers do not move LoRes, and LoRes's do not move the ULA.

▶ TRY IT Big Pixels

Part II, key 4. Key 1 draws a 128 × 96 picture in LoRes. Every entry of the ULA's second palette holds its own index as a colour, so each byte is the colour of its pixel. Key 2 draws it again in Radastan, with 16 colours in entries 0–15. Key 3 starts or stops the scroll. The code is in `examples/code/demos/2-picture/big-pixels.kz80.asm`, and this is its frame loop:

```
BpFrame:
    halt
    ld a,(BpScroll)
    or a
    jr z,BpKeys
    ld a,(BpX)
    inc a                   ; one dot across: half a pixel; wraps at 256
    ld (BpX),a
    nextreg $32,a
    ld a,(BpY)
    inc a                   ; one line down: half a pixel; wraps at 192
    cp 192
    jr c,BpYOk
    xor a
BpYOk:
    ld (BpY),a
    nextreg $33,a
```

The picture moves half a pixel each frame, smoothly, and wraps round in both directions; the border stays put. The text is drawn into the picture, so it scrolls with it.

What a screen costs

A program that redraws the whole screen every frame has to copy every byte of it. At the Spectrum's 3.5 MHz the Z80's usual block copy, `LDIR`, takes 21 T-states a byte, and a 128K frame is 70,908 T-states (Figure 13.5).

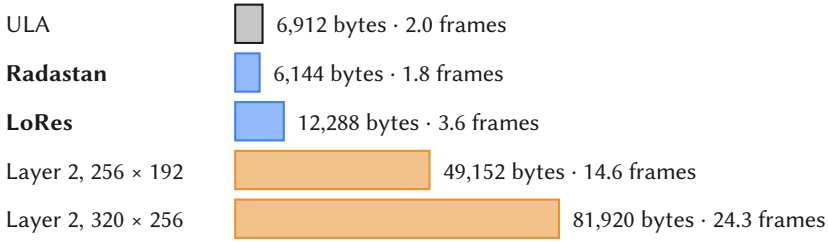


Figure 13.5: What a full screen costs at 3.5 MHz

LoRes gives every 2 × 2 block its own colour, out of 256, for **less than twice the ULA's memory** and a quarter of Layer 2's at 256 × 192. Radastan halves LoRes again: a 16-colour screen smaller than the ULA's own. At 28 MHz every copy is faster, but the proportions stay. That is why I keep coming back to these two modes when colour matters more than detail.

In short. LoRes replaces the ULA's picture with 128 × 96 pixels of 2 × 2 dots, a byte each through the ULA's palette, in two unscrambled 6K halves of bank 5 at `$4000` and `$6000`. Radastan, bit 5 of NextReg `$6A`, uses 4 bits a pixel, so bank 5 holds two 6K screens to flip between. LoRes is no layer of its own: it takes the ULA's clip window, border, transparency and place in the order, and NextReg `$68` bit 7 switches both off. Only its scroll is separate, NextRegs `$32` and `$33`, in half-pixel steps.

CHAPTER 14

Layer 2: A Byte for Every Dot

Layer 2 is the simplest idea on the Next: one byte, one dot, any colour. No attributes, no colour clash, no scrambled addresses. And yet, uhm, it has the most surprising memory layout of all the layers. In two of its three resolutions the bytes do not run across the screen, line after line, as in every other bitmap you have met. They run **down** it, column after column. Why on earth would anyone build it that way?

Layer 2 is where a Next program goes when it wants a picture without compromises: a backdrop in many colours, a photo, a screen the CPU can fill dot by dot without ever thinking about cells. That is why I call it **the painter**: no grid tells it where one colour must stop. But the freedom costs memory, lots of it, and knowing where those bytes are, and in which order, is what lets your code draw fast.

This chapter builds on Chapter 9 for the layers, the wide area, clip windows and scrolling, and on Chapter 11 for how a byte becomes a colour.

- What one byte per dot costs, and where those bytes live
- How the three resolutions lay out their bytes, and why two run down the screen
- How a program reaches the picture, through port $123B or the MMU
- How scrolling and clipping work, and why Layer 2 does not slow the CPU down

---

One byte, one dot

The ULA spends one bit on each dot and one attribute byte on every 8 × 8 cell, so 6,912 bytes paint the whole Spectrum screen, at the price of two colours per cell. **Layer 2** spends a whole byte on each dot. The byte is a palette index, so every dot can have any of the 256 colours of the Layer 2 palette, which are any 256 of the Next's 512.

The price is seven times the memory, and seven times the work for a CPU that redraws the picture. The ULA's memory is bank 5, 16K kept inside the FPGA, and a 48K or 80K picture does not fit there. So Layer 2 is a layer of its own, with its own way into the Next's main memory. I think that was the right call.

Where Layer 2 lives

Layer 2 lives in the Next's ordinary RAM, counted in **16K banks**. Bits 6–0 of the Layer 2 Active RAM bank register, **NextReg $12**, name the bank where the picture starts; it continues in the banks after it. A 256 × 192 picture takes **3 banks** (48K), the 320 × 256 and 640 × 256 pictures **5 banks** (80K).

The Layer 2 Shadow RAM bank register, **NextReg $13**, names a second picture: the **shadow**. It is never shown, and exists only for the CPU. Your program draws the next frame into the shadow while the active picture is on screen, then swaps the two numbers: **double buffering**, with not a single byte copied. The painter works on the canvas behind the easel while the audience admires the one in front. I find this one of the friendliest features of the layer (Figure 14.1).

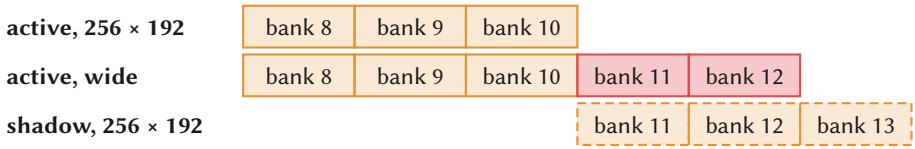


Figure 14.1: Layer 2's banks after a reset

Two details, and the first one is a trap. The defaults, banks 8 and 11, suit the 256 × 192 mode: in the wide modes the active picture runs into the shadow's first two banks, so if you want double buffering there, set NextReg $13 to 13 or beyond. My rule of thumb: the moment you pick a wide mode, move the shadow. And a picture placed so that it runs past the end of the memory the Next can address has **no dots** there: those parts of Layer 2 are transparent.



KLIVE IDE The Layer 2 Inspector

View › Machine Views › Layer 2 Inspector, or `show-layer2`, shows the banks NextReg $12 displays, the shadow banks of $13, or the banks the $123B window maps. Showing them in turn answers the question the screen cannot: where are my writes going?

Three resolutions

Bits 5–4 of the Layer 2 Control register, **NextReg $70**, choose the resolution:

Bits 5–4	Resolution	Bits a dot	Where	Memory	Byte order
00	256 × 192	8	the display area	48K, 3 banks	across: line after line
01	320 × 256	8	the wide area	80K, 5 banks	**down**: column after column
10	640 × 256	4	the wide area	80K, 5 banks	**down**: column after column

Look at the last column: that is the surprise from the opening.

Why the bytes run down the screen

Watch how the hardware turns a position into an address. In 256 × 192, the line y and the dot x are each a byte, and the address of the dot within the picture is the two bytes **side by side**: y as the high byte, x as the low byte. No arithmetic at all, and the dots run across the screen, because x is the part that counts up first. The top bits pick the 16K bank, counted from NextReg $12 (Figure 14.2).

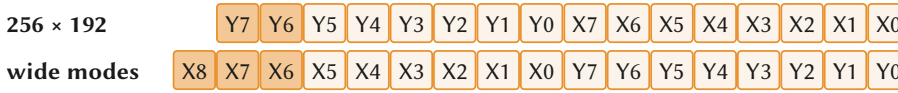
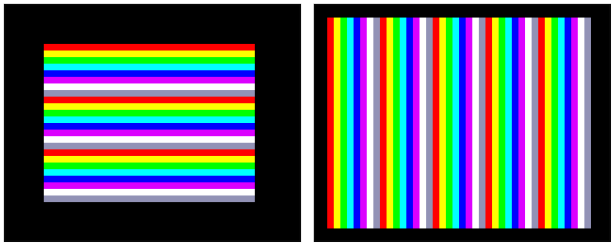


Figure 14.2: The address of a dot within the picture

Now try the same with 320 dots across. The dot no longer fits in a byte: it needs 9 bits, 0–319. So "line high, dot low" would take a multiplication, $y \times 320 + x$, and hardware hates multiplying in a hurry. So turn the picture on its side, and the problem disappears. I find this the most elegant dodge in the whole screen design. The wide modes have exactly **256 lines**, so the *line* is the coordinate that fills a byte, and the address is again two coordinates side by side: the column x as the high part, the line y as the low byte. The dots now run **down** the screen, and each column of the picture is one 256-byte page of memory.

So the rule is the same in every mode, and I find it easiest to remember it like this: whichever coordinate is 256 long goes in the low byte. Here it is on the machine: the same bytes in both modes, each of eight colours painted over 2,048 bytes in a row of memory, so that a colour is eight lines in one mode and eight columns in the other (Figure 14.3).



left: 256 × 192
a page is a line

right: 320 × 256
a page is a column

Figure 14.3: The same bytes in two resolutions

For your program, the column order has advantages of its own. Moving down a column is `INC L`, moving across is `INC H`, and an 8K memory page holds exactly 32 whole columns. My advice: when you design a routine for a wide mode, let it walk down the columns, not across the lines.



KLIVE IDE Bank boundaries

The inspector outlines each 16K bank with its number: horizontal bands at 256 × 192, vertical bands in the wide modes, which store their dots column by column. A program that switches to a wide mode without rearranging its bytes shows up as a picture turned on its side.

640 × 256: two dots in a byte

The third mode doubles the dots across, to **640**, each one a 14 MHz dot, half the width of a standard dot. The memory stays at 80K and the order stays column after column, with x still counting 0–319. What changes is the byte: it now holds **two dots**, four bits each, the high nibble on the left.

Why four bits? Because Layer 2 gets **one byte for every 7 MHz dot** from memory, in all three modes. At 640 across, the same byte has to serve two dots, so each gets half of it. I like how little this mode costs the hardware: nothing new to fetch, only a byte cut in two.

Four bits give each dot 16 colours. The **palette offset**, bits 3–0 of the Layer 2 Control register, NextReg `$70`, becomes the high four bits of every index, and so picks which 16 (Chapter 11). To my mind, this mode's real job is text: here is 80-column text, ink nibble `$F` and paper nibble `$0` (Figure 14.4).

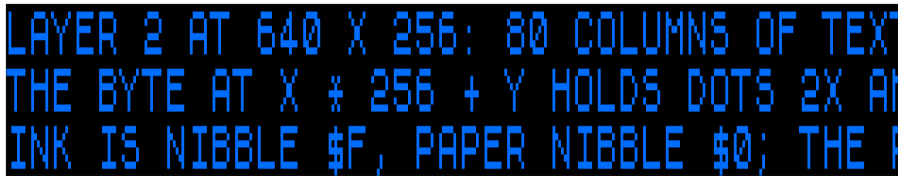


Figure 14.4: Text at 640 × 256: 40 of the 80 columns



KLIVE IDE Two dots in a byte

Click a dot in the inspector, and at 640 × 256 it shows the dot's nibble and which half of the byte it is in; in this mode the palette offset is the high nibble. The image is drawn twice as tall, so the dots keep their real shape.

The CPU's view through port $123B

Layer 2 reads its picture from banks the CPU does not see unless it asks to. There are two ways to ask: port `$123B` and the MMU.

The Layer 2 control port, **port `$123B`**, can lay the picture over the bottom of the CPU's memory: the painter's scaffolding of Chapter 3.

Bit	Meaning
7–6	which part: `00`, `01`, `10` the first, second or third 16K at `$0000`–`$3FFF`; `11` the first 48K at `$0000`–`$BFFF`
5	not used (0)
4	0 for this layout; 1 for the bank offset below
3	0 maps the active picture (NextReg `$12`), 1 the shadow (NextReg `$13`)
2	1: CPU reads come from Layer 2
1	1: Layer 2 is shown (also bit 7 of the Display Control 1 register, NextReg `$69`)
0	1: CPU writes go to Layer 2

A write with bit 4 set stores a **bank offset** in bits 2–0 instead, 0–7 banks, added to the part chosen in bits 7–6. The 48K window shows only three of a wide picture's five banks; with an offset, any of them can be reached.

If you ask me, the separate switches for reads and writes are what make the port useful. With **writes only**, the CPU still reads, and runs, whatever is normally at those addresses, the ROM at `$0000`–`$3FFF` for a start, while every write there lands in Layer 2.

One rule covers every use: mapped writes go to Layer 2, the CPU's own pushes included. With a 16K window the stack must be at `$4000` or above; with the 48K window, at `$C000` or above, and so must any variable your code updates. My advice is to make it a habit: whenever port `$123B` maps anything, keep your stack in the top 16K.

Plot a dot, one third at a time (a 16K window, writes only, 256 × 192). The three thirds of a 256 × 192 picture are 64 lines each, and bits 7–6 of the line number *are* the third's number. So the line's top two bits go straight into the port, and the rest of the line and the dot become the address, H the line within the third, L the dot:

```
; Plot x = E, y = D (0-191) in the colour at L2Ink
Plot:   ld a,d
        and $c0         ; y's bits 7-6: the third
        or $03          ; bit 1: shown; bit 0: writes mapped
        ld bc,$123b
        out (c),a
        ld a,d
        and $3f         ; the line within the third
        ld h,a
        ld l,e          ; the dot
        ld a,(L2Ink)
        ld (hl),a       ; to Layer 2; reads still see ROM
        ret
```

Reach all 80K of a wide picture (the 48K window with a bank offset, 320 × 256). In the wide modes the 48K window holds columns 0–191, with no gaps, and **H is the column and L the line**. For the columns from 192 on, a write with bit 4 set moves the window: with bank offset 2 it shows banks 2–4, columns 128–319, and column x is at H = x − 128. This draws the whole of column 300 in green:

```
        ld bc,$123b
        ld a,$c3        ; 48K window, writes mapped, shown
        out (c),a
        ld a,$12        ; bit 4 set: bank offset 2
        out (c),a
        ld h,300-128    ; column 300
        ld l,0          ; from line 0...
        ld b,0
Col:    ld (hl),$1c
        inc l           ; ...down all 256 lines
        djnz Col
        ld bc,$123b
        ld a,$10        ; the offset back to 0
        out (c),a
```

```
        ld a,$02        ; unmap
        out (c),a
```

Two more port values are worth knowing. `$C7` maps the whole 48K of a 256 × 192 picture for reads and writes, H the line and L the dot, so your program can read a dot, change it and write it back. Let me warn you about this one: everything the CPU fetches below `$C000` then comes from Layer 2, its own instructions included, so the code must be at `$C000` or above, with interrupts off. Forget that, and the CPU cheerfully runs your picture as a program. I have nothing against modern art, but not as machine code. And `$CB` maps the **shadow**, writes only, while the active picture stays on screen: that is how you draw the next frame for double buffering.



KLIVE IDE Where the window points

The inspector's globals strip says in words where port `$123B` sends the CPU's writes, and warns when the window maps a bank the display is reading: no double buffering, and a risk of tearing.

Through the MMU

The other way in is the Next's MMU, the eight registers from the MMU slot 0 register, **NextReg $50**, to the MMU slot 7 register, **NextReg $57**, which page any 8K page of RAM into any 8K slot of the CPU's 64K. A 16K bank b is 8K pages $2b$ and $2b + 1$, so the default picture, banks 8–12, is pages 16–25. In the wide modes your program can page in the 32 columns it is drawing and leave the rest of memory alone. Which way do I prefer? Port `$123B` for a quick job at the bottom of memory; the MMU when the painting takes a while and the rest of the program has to keep running around it.



KLIVE IDE A dot's four addresses

Click a dot, and the inspector gives its bank and offset, its 8K page, its physical address and its Z80 address. **Break on write** stops on a write to that byte wherever its bank is paged in, and, when `$123B` maps it, at the window's address too: a write through the window does not go through the MMU.

Scrolling, clipping and colours

Layer 2 scrolls by its own offsets: the Layer2 X Scroll LSB register, **NextReg $16**, with bit 0 of the Layer 2 X Scroll MSB register, **NextReg $71**, as a ninth bit, and the Layer2 Y Scroll register, **NextReg $17**. The picture wraps round at its own size:

Mode	X offset	X wrap	Y offset	Y wrap
256 × 192	0–255 dots (NextReg $71 is not used)	256	0–191	192
320 × 256	0–319 dots	320	0–255	256
640 × 256	0–319, in steps of two dots	640	0–255	256

Its clip window, the Clip Window Layer 2 register, **NextReg $18**, counts in the area the mode covers: in 256 × 192, the display area, one unit a dot; in the wide modes, the wide area, X in units of **two** 7 MHz dots (four dots in 640 × 256) so that 0–159 spans the 320 positions, and Y one unit a line. The clip window starts as 0–255, 0–191, so in a wide mode the bottom 64 lines stay clipped until your program writes 0–159, 0–255. Let me warn you, this one catches almost everyone.

Layer 2 is **transparent** where its colour, after the palette, equals the value in the Global Transparency Colour register, NextReg $14. And an entry of its two palettes can carry the **priority bit** that lifts its dots above every other layer (Chapter 17).



KLIVE IDE What the screen gets

As displayed applies the scroll and the clip window, so the inspector's image matches what reaches the mixer. A dot shows as transparent when its colour, not its index, equals NextReg $14, and **Highlight priority pixels** dims every dot without the priority bit.

▶ TRY IT Layer 2 Resolutions

Part II, key 5. Keys 1, 2 and 3 show the three resolutions, and Q, A, O and P scroll the picture with NextRegs $16, $71 and $17. The code is in `examples/code/demos/2-picture/layer-2-resolutions.kz80.asm`. One routine draws every mode, through MMU slot 6, 8K at a time from bank 8: a 256-byte run is a line at 256 × 192 and a column in the wide modes. This is how a mode is set, with its clip window:

```
    ld a,(L2Mode)
    swapnib                 ; the mode in bits 5-4
    cp $20
    jr nz,L2SetMode1
    or $0F                  ; 640 x 256: palette offset $F
L2SetMode1:
    nextreg $70,a
    ; The clip window: all but the bottom three text rows
    nextreg $1C,1           ; bit 0: Layer 2's clip index
    nextreg $18,0           ; X1
    ld a,(L2ClipX2)
    nextreg $18,a           ; X2: 255, or 159 in units of two dots
    nextreg $18,0           ; Y1
```

▶ **TRY IT** Layer 2 Resolutions *(continued)*

```
    ld a,(L2ClipY2)
    nextreg $18,a           ; Y2: 167, or 199 in the wide modes
```

The picture is a grid of 16-dot cells. At 320 × 256 it fills the border too, and at 640 × 256 its cells are half as wide. Scroll with O or P, and row 21 shows the X offset wrap round at 256, or at 320 in the wide modes.

Fetching a byte for every dot

Layer 2 reads the Next's main memory, an external static RAM shared with the CPU and the DMA. It needs **one byte for every 7 MHz dot** while it draws, a read about every 140 nanoseconds, and the CPU at 28 MHz wants the same memory too.

The memory is fast enough to be shared by time. It completes one access in every tick of the 28 MHz clock, **four accesses per dot**, and each tick goes to whoever asks, in a fixed order of priority: the **CPU** first and **Layer 2** second; the DMA, while it holds the bus, takes the CPU's place (Chapter 5). I picture it as a revolving door that turns four times for every dot, with the CPU always at the front of the queue. Which ticks the CPU takes depends on the instructions it runs (Figure 14.5).

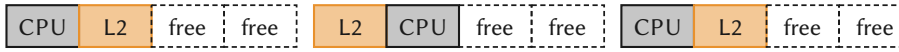


Figure 14.5: Three dots, twelve ticks of main memory (L2: Layer 2)

Layer 2 can afford second place because it is never in a hurry: it asks for each byte a dot ahead of the dot it draws, so a request that waits a tick still arrives in time. The CPU never waits for Layer 2 at all, at any speed; unlike the ULA's memory on a Spectrum, Layer 2's has **no contention**. A quiet triumph, I think.

In short. Layer 2 gives every dot a byte in main RAM, 48K at 256 × 192 and 80K in the wide modes, where bytes run down the screen in 256-byte columns and 640 × 256 packs two 4-bit dots into each. The CPU never waits for it, and swapping NextRegs $12 and $13 flips pictures, but wide modes need the shadow at bank 13 or beyond. While port $123B maps writes, every write to the mapped window lands in Layer 2, the stack's included. Switch to a wide mode without rewriting NextReg $18 and the bottom 64 lines stay clipped.

CHAPTER 15

The Tilemap: A Screen of 8 × 8 Tiles

Have you ever looked closely at a scrolling game? Almost always, the scenery repeats: the same brick, the same patch of grass, the same pipe. Game consoles were built on that observation. Store each little graphic once, and keep the screen as a list of which one goes where. So how small can a full-width picture get?

The Next's tilemap is that idea, and Chapter 9 calls it **the set builder**: it puts its scenery together from a box of ready-made pieces and never paints a dot of its own. A 320 × 256 picture takes 80K as a Layer 2 bitmap. The tilemap draws one from a **2.5K** map of 40 × 32 small numbers and 8K for 256 tiles: 10.5K in all. Scrolling is where the bargain pays: a new 8-dot column of scenery is 32 map entries, 64 bytes, where Layer 2 would need 2,048.

This chapter builds on Chapter 9 for the wide area and the clip windows, and on Chapter 11 for palette offsets.

- How the map and the tile definitions fit into bank 5 and half of bank 7
- How a map entry's attribute byte gives every tile eight orientations
- How text mode and 80 columns work, and how a tile goes above or below the ULA
- How scrolling and clipping work, and how tiles are fetched ahead of the beam

The map and the tiles

The tilemap covers the 320 × 256 **wide area** with a grid of **40 columns by 32 rows** of 8 × 8 cells (Figure 15.1). What it draws comes from **two separate areas of memory**:

1. The **map** says *which* tile goes where: one entry per cell, row after row, 40 entries to a row. It holds no dots at all.
2. The **tile definitions** say what each tile *looks like*: 32 bytes for each tile, tile 0 first. They say nothing about where a tile appears.

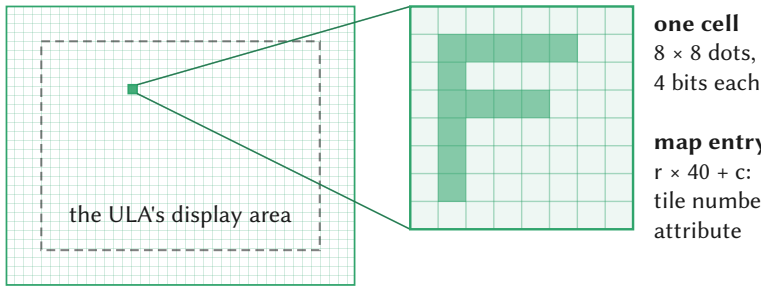


Figure 15.1: The tilemap's grid: 40 × 32 cells over the whole wide area.

The map refers to the tiles only by number, so any number of cells can name the same tile. Change one tile's 32 bytes, and every cell that shows it changes at once: a whole meadow of grass can sway in the wind from a single 32-byte write.



KLIVE IDE The Tilemap Inspector

View › Machine Views › Tilemap Inspector, or `show-tilemap`, shows the map and the tile definitions as the hardware decodes them, in two linked views. Select a cell in the **Map** to see its tile; select a tile in **Tiles** to outline every cell that uses it.

Bank 5 and bank 7: where the tables can live

The two areas cannot be just anywhere. The tilemap reads its bytes through the same memory port as the ULA, and that port reaches only the **block RAM inside the FPGA**: **24K**, all of **bank 5** and the first 8K of **bank 7** (Chapter 12 explains why the screen memory is kept there). To the CPU this block RAM simply *is* those banks, 8K pages 10, 11 and 14 of the MMU. Bank 7's second half, page 15, is ordinary main memory, out of the tilemap's reach.

Each area is placed on its own: the Tilemap Base Address register, **NextReg `$6E`**, says where the map starts, and the Tile Definitions Base Address register, **NextReg `$6F`**, where the tiles start. Both registers use the same layout:

Bit	Meaning
7	0: the area is in bank 5; 1: in bank 7 (its first 8K)
6	reserved: keep it 0
5–0	where in the bank it starts, in steps of 256 bytes: 0–63 in bank 5, 0–31 in bank 7

Bank 5 is normally at `$4000`, so a value of `$20` puts an area at `$6000`; `$80` puts it at the start of bank 7, which is `$C000` when bank 7 is paged in there. The two areas choose

their bank independently. One that runs past the end of its bank continues from the bank's start: past 16K in bank 5, past 8K in bank 7.

After a reset, `$6E` is `$2C` and `$6F` is `$0C`: the map at `$6C00` and the tiles at `$4C00`, inside the ULA's bitmap. So set both yourself before you switch the tilemap on with bit 7 of the Tilemap Control register, **NextReg `$6B`**.

How big the two tables are

The other bits of NextReg `$6B` decide the sizes, and the sections below take them one at a time:

Area	One item	How many	Total
The map	an entry: 2 bytes, or 1	a cell: 1,280, or 2,560 in 80 columns	1,280 to 5,120 bytes
The tile definitions	a tile: 32 bytes, or 8 in text mode	256 tiles, or 512	2,048 to 16,384 bytes

The sums are tight. A map of 2,560 bytes and 256 tiles of 8,192 come to 10,752 bytes, and with the ULA's 6,912 that is 17,664: more than bank 5's 16,384. This is where bank 7 earns its place: its 8K is exactly the size of 256 tiles.

Layouts that fit

Here are four common ways to share the 24K. Bank 5's addresses are those the CPU normally sees; bank 7's assume it is paged in at `$C000`. BASIC keeps its system variables from `$5B00`, so these layouts suit a program that does not return to it.

Layout	The map	The tile definitions	`$6E`, `$6F`
256 tiles beside the ULA	bank 5, `$5B00–$64FF`	bank 7, `$C000–$DFFF`	`$1B`, `$80`
All in bank 5	bank 5, `$5B00–$64FF`	bank 5, `$6500–$7FFF`: 216 tiles	`$1B`, `$25`
512 tiles	bank 7, `$C000–$C9FF`	all of bank 5	`$80`, `$00`
80 columns of text	bank 5, `$5B00–$6EFF`	bank 5, characters 32–127 at `$7000–$72FF`	`$1B`, `$2F`

The first uses page 14, which is also where a 128K program keeps its **shadow screen** (port `$7FFD` bit 3): the two cannot be used at once. The third fills bank 5 with tiles, so the ULA would show them as a scrambled picture; your program switches it off with bit 7 of the ULA Control register, NextReg `$68`. The text layout's base is `$6F00`, but a character's tile number is its code, so tiles 0–31 are never used and their 256 bytes

may hold anything. My favourite of the four is the first: it keeps the ULA's screen and leaves a full box of 256 pieces.



KLIVE IDE Overlaps, found for you

The inspector checks the map, the tiles the map uses and the ULA's bitmap and attributes against one another, and warns when two of them overlap, such as the map and the ULA's attributes. An overlap is the commonest tilemap bug, and this way you see it before the program writes into it.

A trap: moving a setup from bank 5 to bank 7

The 8K limit of bank 7 is easy to forget, because to the CPU bank 7 looks like any other 16K bank. Suppose your program has the map at `$4000` and the tiles at `$6000`, so `$6E` is `$00` and `$6F` is `$20`. It works. Now the program copies all of bank 5 into bank 7, byte for byte, and switches both areas with the same offsets: `$6E` = `$80` and `$6F` = `$A0`. What could possibly go wrong? Figure 15.2 has the answer.

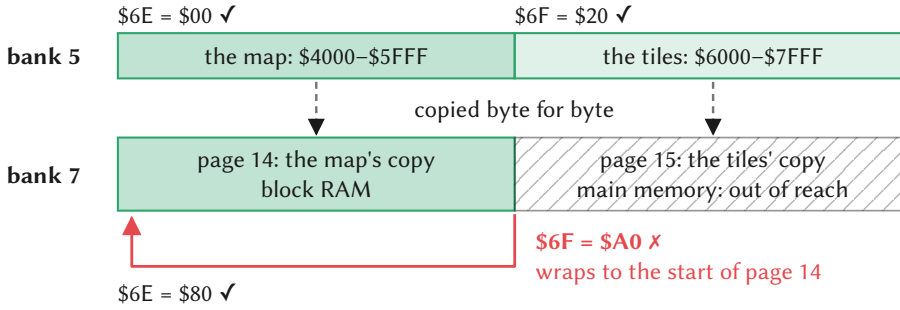


Figure 15.2: Copying bank 5 into bank 7 puts the tiles out of the tilemap's reach.

The map still works, because its copy is in page 14. The tiles' copy is in **page 15**, which the tilemap cannot see. And asking for offset 32, 8K into bank 7, does not reach page 15 anyway: only the low 13 bits of the address are used in bank 7, so 8K wraps round to the start of page 14. The tilemap reads its tile definitions **from the map's bytes**, and the screen shows the right arrangement of cells, each drawn as noise.

The fix is to keep everything the tilemap reads in bank 5 or page 14. Copy the 8K of tiles to the start of bank 7, set `$6F` to `$80`, and leave the map in bank 5. Both fit in bank 7 only if they share its 8K, as a 2,560-byte map and 176 tiles exactly do.



KLIVE IDE The trap, flagged

When the map or the tiles in bank 7 run past its 8K, the inspector warns that the table wraps: it continues at the start of bank 7, not in the next page, exactly as this section describes.

The attribute byte

A map entry is **two bytes**: the tile number and an **attribute** byte that says how to draw it (Figure 15.3).



Figure 15.3: A map entry: the tile number and its attribute byte.

The fields of the attribute byte are, from the top bit down:

- **Palette offset** (bits 7–4): the high four bits of the palette index for this tile's dots (see "A tile's dots" below).
- **X** (bit 3): mirror the tile left to right.
- **Y** (bit 2): mirror the tile top to bottom.
- **rot** (bit 1): rotate the tile 90 degrees clockwise. Together, X, Y and rot give eight orientations (see "Eight orientations" below).
- **ULA** (bit 0): decides which is in front, the tile or the ULA screen under it (explained next).

The tilemap and the ULA are mixed into one layer before the layer order is applied, and the attribute's bit 0 decides, tile by tile, which is in front: 0 puts the tile above the ULA, 1 puts the ULA above the tile. Bit 0 of NextReg `$6B` overrides every tile and puts the whole tilemap on top.

When every tile can be drawn the same way, the attribute is wasted space. Bit 5 of NextReg `$6B` removes it: each entry is then one byte, the map halves to 1,280 bytes, and every tile uses the Default Tilemap Attribute register, **NextReg `$6C`**.

Need more than 256 tiles? Bit 1 of NextReg `$6B` switches on **512-tile mode**: bit 0 of the attribute becomes the ninth bit of the tile number, and 512 tiles take the whole of bank 5. The bit no longer says "ULA over this tile", so every tile is below the ULA unless bit 0 of NextReg `$6B` puts them all on top.



KLIVE IDE One cell, field by field

Click a cell in the inspector: it lists the tile number, the attribute, the palette offset, the turns and the priority, with the raw bytes. In 512-tile mode the tile number says `bit 8 from attr`, and a warning reminds you that no cell can be over the ULA then.

A tile's dots

A tile is **8 × 8 dots of 4 bits**: 32 bytes, 4 for each row, two dots in a byte, the high nibble on the left. The dot at column x of row y of tile t is in byte $t \times 32 + y \times 4 + x \div 2$ of the tile definitions.

A dot is a number 0–15, and the attribute's **palette offset** becomes the high four bits of the index into the tilemap palette (Chapter 11). So one tile can be drawn in any of 16 sets of 16 colours: a red and a blue enemy are the same 32 bytes with a different offset. Bit 4 of NextReg `$6B` chooses which of the two tilemap palettes is used.

Transparency is decided on the 4-bit dot before the palette: a dot equal to bits 3–0 of the Tilemap Transparency Index register, **NextReg `$4C`**, `$F` after a reset, is transparent, whatever the offset.

Eight orientations

Bits 3–1 of the attribute turn a tile into any of **eight** versions of itself: the eight ways an 8 × 8 square can be laid down, four turns and their mirror images (Figure 15.4).

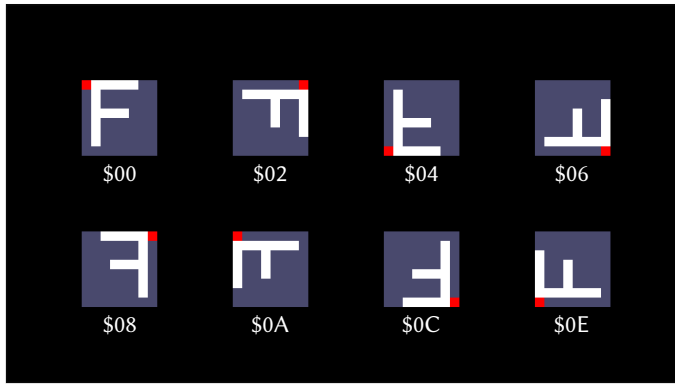


Figure 15.4: One tile in its eight orientations, on the machine.

The rule is: **rotate 90 degrees clockwise first, then mirror**. Rotation moves the tile's dot (u, v) to $(7 - v, u)$; then X mirror takes the column x to $7 - x$, and Y mirror the row y to $7 - y$. The two mirrors together are a half turn, so rotate plus both (`$0E`) is 270 degrees clockwise.

The hardware does all this without moving a dot. To draw column x, row y of the cell, it reads a different dot of the tile: it flips x or y for the mirrors, and for rotate swaps the two, with one more flip of x, before it forms the address. Three bits of the attribute change a few bits of the address, so the transformation costs nothing. I think this is the cleverest trick in the tilemap.

It pays because game graphics are full of symmetry: one tile per shape covers every direction of a pipe, a wall or a corner, and the tile set, the expensive part of the memory, shrinks by up to eight times.



KLIVE IDE A tile, turned

Open tile snapshot on a cell shows its tile **As shown**, rotated first and then mirrored, the way the hardware draws the cell, or **As stored**, as the bytes hold it. Switch between the two to watch the turns at work.

40 or 80 columns

Bit 6 of NextReg `$6B` doubles the columns to **80 × 32**. The cells keep their 8 dots, but they are 14 MHz dots, half as wide, so 80 cells fill the same 320-dot width, like Layer 2's 640 mode. The map doubles to 80 entries a row, 5,120 bytes with attributes. Four-bit tiles work here too, but eighty columns are for text, in my view.

Text mode

Bit 3 of NextReg `$6B` switches the tilemap to **text mode**. A tile becomes a character in the way the Spectrum's font is: **1 bit a dot, 8 bytes a tile**, one byte a row, bit 7 on the left. So 256 characters take 2K instead of 8K, and a copy of the Spectrum's font fits in 768 bytes.

A dot of one bit cannot pick a colour by itself, so the attribute's bits 7–1 become a **7-bit palette offset**, and the dot is bit 0 of the index. Every character picks its own **pair** of palette entries, even for paper and odd for ink, from 128 pairs. The price is the other attribute bits: text mode has no mirroring and no rotation, and bit 0 is colour too.

Transparency changes with it: a text-mode dot is transparent when its **colour**, after the palette, equals the Global Transparency Colour register, NextReg `$14`, the rule the ULA and Layer 2 use. Figure 15.5 shows it at 80 columns, its rows using attributes `$00`, `$02`, `$04` and `$06`. A whole screen of it is 5,120 bytes of map and 768 of font.

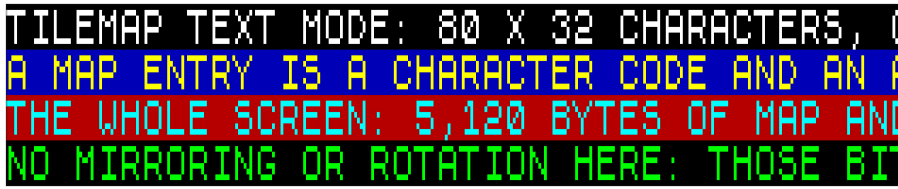


Figure 15.5: Text mode at 80 columns: the left half of the screen, four rows.

Scrolling and clipping

The tilemap has its own scroll: an X offset of 10 bits, bits 1–0 of **NextReg $2F** above **NextReg $30**, and a Y offset in **NextReg $31**. The picture wraps round at its own size: across at 320 dots, or 640 half-width dots in 80 columns, and down at 256 lines. The grid covers exactly the wide area, so a scroll never shows anything but the map. A game that scrolls endlessly writes the next column of tiles into the map column that has just left the screen.

The tilemap's clip window, the Clip Window Tilemap register, **NextReg $1B**, works in the wide area: X in units of two 7 MHz dots (four dots in 80 columns), so 0–159 covers the width, and Y one unit a line. It starts as 0–159, 0–255, the whole grid.



KLIVE IDE The window on the map

The inspector's **Whole map** view draws the map unscrolled, with a dashed outline for the part the screen shows. **As displayed** applies the scroll and the clip window, so the view matches the layer as it reaches the screen.

▶ TRY IT Tile Scroller

Part II, key 6. The demo builds one tile from the ROM's letter F, with a dot at its top-left corner, and fills the map with it in its eight orientations, eight columns and then two empty ones. The map is at `$5B00` and the tiles at `$6F00`, beside the ULA's screen (NextRegs `$6E` = `$1B`, `$6F` = `$2F`). The code is in `examples/code/demos/2-picture/tile-scroller.kz80.asm`; this part writes one map entry:

```
TsMapCell:
    ld a,b
    cp 8
    ld a,0
    jr nc,TsMapTile         ; columns 8 and 9 of a group: tile 0, empty
    inc a                   ; tile 1
TsMapTile:
    ld (hl),a
```

▶ TRY IT Tile Scroller *(continued)*

```
    inc hl
    ld a,b
    add a,a                 ; bits 3-1: X mirror, Y mirror, rotate
    or c
    ld (hl),a
    inc hl
```

The eight Fs of a group are attributes `$00` to `$0E` in a row, the eight of Figure 15.4. Press 8 for 80 columns: the map is filled again, 80 entries a row, and every F is half as wide. Q, A, O and P scroll the map by a dot a frame with NextRegs `$2F`, `$30` and `$31`, and it wraps round at 320 dots across, 640 in 80 columns.

Fetching a character ahead

The tilemap reads through the **same port of the block RAM as the ULA**. The ULA, which needs its bytes at exact moments, goes first; the tilemap takes every cycle the ULA does not use.

To be ready in time, it works **one character ahead**. While one cell's dots are drawn from one half of a small buffer, it fetches the next cell into the other half, the map entry and then the tile's row, and the halves swap at the cell edge. Its settings, the base addresses, the attribute mode, text mode, 512 tiles and the on-top bit, are taken again for every cell it fetches, so your program or the Copper can change them between two cells of the same line.

In short. The tilemap covers the wide area with 40 × 32 or 80 × 32 map entries, each naming a 32-byte tile of 4-bit dots, or an 8-byte tile in text mode, and only 4-bit tiles take eight orientations, rotated first, then mirrored. Map and tiles must both sit in 24K of block RAM: bank 5 and the first 8K of bank 7. Set NextRegs `$6E` and `$6F` yourself before switching on with bit 7 of `$6B`. Copying bank 5 into bank 7 is the trap: page 15 is out of reach and offsets wrap at 8K, so every cell draws noise.

CHAPTER 16

Sprites: 128 Pictures and Two Line Buffers

Picture a busy game screen: your ship, a dozen enemies, a hail of bullets. On the Next, every one of them can be a sprite, up to 128 at once, each moving on its own. You would expect the hardware to keep their picture somewhere, 320 × 256 dots of it. It does not. It holds **two lines**. So how do 128 sprites fit into two lines? I think the answer is one of the best bits of engineering in the whole machine.

Without sprites, as on the 48K Spectrum, the CPU must draw every moving thing into screen memory, then put back what was behind it before drawing it again. Hardware sprites take all of that off the CPU. Your program says "put this 16 × 16 picture at X 140, Y 90", and to move it changes two numbers. Chapter 9 calls the sprites **the troupe**: 128 performers, each up to 16 × 16 dots, enlarged up to eight times, turned and mirrored, and linked into bigger objects.

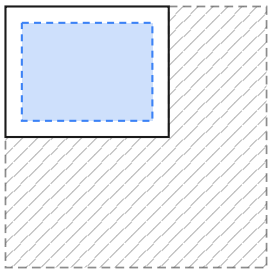
This chapter builds on Chapter 9 for the layers and the clip windows, and on Chapter 11 for the palette offset.

- How positions, patterns and attribute bytes define a sprite
- How orientations, sizes and anchors work
- How the two line buffers decide which sprite ends up in front
- How many sprites fit in a line's time budget, and what happens when they do not

---

Sprite space

A sprite's position is two **9-bit** numbers, X and Y, each 0–511. They place the sprite's top-left dot in a square of 512 × 512 positions, **sprite space**, of which only a corner is ever shown (Figure 16.1). A sprite at (32, 32) starts at the top-left corner of the ULA's picture, and one at (0, 0) at the top-left corner of the border.



sprite space: 512 × 512
X and Y are 9 bits each, 0–511

outlined: the wide area, 320 × 256
blue: the display area, from (32, 32)
hatched: never shown

Figure 16.1: Sprite space, to scale.

Why 512 positions for a 320-dot screen? So that a sprite can be **partly off** any edge. A 16-dot sprite moving left from X 0 goes to X 511, 510, …: positions wrap round at 512, so at X 500 its first 12 dots are out of sight past the right of sprite space, and its last 4 come back in at X 0–3. To your program it is X −12. The same happens at the top with Y.



KLIVE IDE The Sprite Inspector

View › Machine Views › Sprite Inspector, or `show-sprites`, shows what the Z80 cannot read back. Its map draws the whole 320 × 256 sprite space: the paper, the clip window, and every visible sprite as an outline. Click one to select it.

Patterns

A sprite's picture is a **pattern**, and the patterns live apart from the sprites, in **16K** of memory of their own inside the FPGA. Many sprites can show the same pattern: ten enemies of one kind cost one pattern, not ten. A pattern is **16 × 16 dots**, stored row after row, the left dot first, in one of two forms:

- **8-bit**: a byte a dot, **256 bytes** a pattern, so 16K holds **64**. The byte is the palette index, and the sprite's 4-bit palette offset is added to its upper four bits.
- **4-bit**: a nibble a dot, the upper nibble on the left: **128 bytes** a pattern, **128** in the same 16K. The nibble is the low four bits of the index, and the palette offset becomes the upper four.

The two forms share one memory, and your program mixes them freely: 8-bit pattern 5 and 4-bit patterns 10 and 11 are the same bytes. A sprite names its pattern with six bits, **N5–N0**, which pick the 256-byte slot. A 4-bit pattern needs one more bit, **N6**, which picks the half of the slot, so its number is 2 × slot + N6. N6 is its *lowest* bit, even though it is the seventh one added, and this is the bit everyone puts in the wrong place the first time.

A dot is **transparent** when its value, before the palette offset, equals the Sprite Transparency Index register, **NextReg `$4B`** (`$E3` after a reset): the whole byte for an 8-bit pattern, the low four bits for a 4-bit one. Bit 3 of the Palette Control register, NextReg `$43`, chooses between the two sprite palettes.



KLIVE IDE The pattern memory

The inspector's **Patterns** view draws the 16K as a sheet: 64 8-bit or 128 4-bit patterns, each the way the sprites that use it read it. A slot read both as 8-bit and as 4-bit gets an amber corner, which is almost always a bug.

The attribute bytes

Everything else about a sprite is in its **attributes**: four or five bytes for each of the 128 sprites, in a table of their own beside the pattern memory (Figure 16.2). X8 and Y8 are the ninth bits of the position.

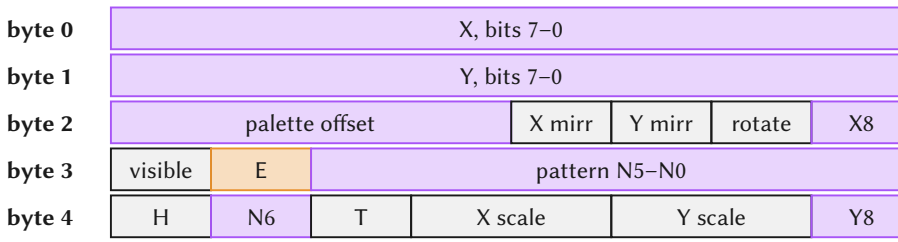


Figure 16.2: A sprite's five attribute bytes, bit 7 on the left.

Field	Meaning
E	1: byte 4 is used. Without it the sprite is 8-bit and unscaled, and Y is 0–255
Pattern	N5–N0 in byte 3, with N6 in byte 4 for a 4-bit pattern
H	1: a 4-bit pattern
T	for an anchor, how its relative sprites follow it

Look closely at byte 4 bits 7 and 6 set to `01`: an 8-bit pattern with the bit only a 4-bit pattern uses. It cannot mean anything sensible as a pattern, so the hardware gives it a job: it makes the sprite a **relative** sprite. I love this kind of thrift.



KLIVE IDE Every slot, decoded

The **Sprites** view has a row for each of the 128 slots with its attributes decoded, and **Raw** shows the five bytes. A 4-byte sprite's fifth byte is struck through, and a note warns when it would matter: writing four bytes where five were meant is a classic bug.

Uploading through three ports

The pattern memory and the attribute table are not in the CPU's memory map. Your program writes them through three ports, each with an index that steps on by itself:

Port	What a write does
`$303B`	bits 6–0 pick the sprite for port `$57`; bits 5–0 pick the pattern slot for port `$5B`, and bit 7 starts it 128 bytes in
`$57`	writes one attribute byte and steps on; after byte 4, or byte 3 when E is 0, to the next sprite
`$5B`	writes one pattern byte and steps on, through all 16K

So you upload the whole table in one stream, four or five bytes a sprite. A 4-byte sprite's byte 4 is not written, and nothing reads it while E is 0.

The attributes can also be written through NextRegs, which the Copper can do in the middle of a frame. **NextReg `$34`** picks the sprite, and **NextRegs `$35`–`$39`** write its bytes 0–4 directly, in any order. **NextRegs `$75`–`$79`** write the same bytes and then step to the next sprite. Normally `$34` and port `$303B` pick their sprites independently; bit 4 of NextReg `$09` ties them together.

Bit 0 of the Sprite and Layers System register, **NextReg `$15`**, switches the sprites on.

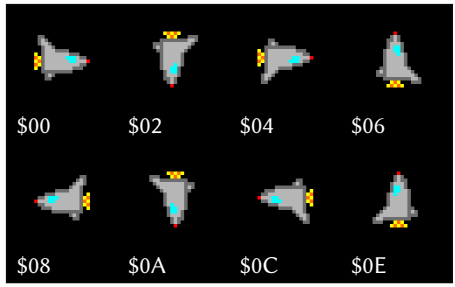


KLIVE IDE Where the next byte goes

The inspector's globals strip shows where the next byte written to port `$57`, to port `$5B` and to the NextReg mirrors will land, and an amber bar marks the next pattern byte in the sheet. When uploads go to the wrong slot, look here first.

Orientations and sizes

Bits 3–1 of byte 2 turn a pattern into any of the **eight orientations** of a tile, by the same rule (Chapter 15): **rotate 90 degrees clockwise first, then mirror**, with the tile's 7 replaced by 15. As with tiles, nothing is moved; the hardware only reads the pattern's dots in a different order. Figure 16.3 shows the spaceship pattern in all eight.

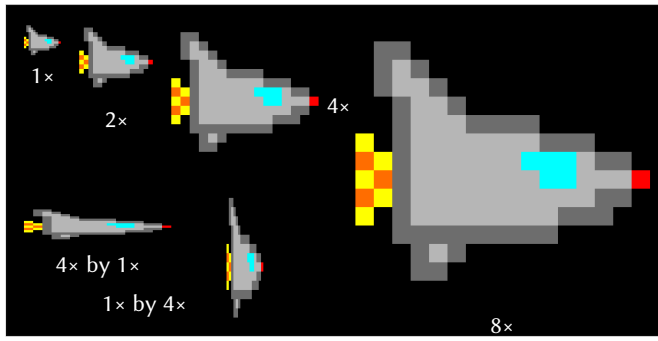


byte 2
$02: rotate
$04: Y mirror
$08: X mirror

stored pattern
$00: nose right

Figure 16.3: The spaceship in its eight orientations, on the machine.

Bits 4–3 of byte 4 scale a sprite **across** and bits 2–1 **down**, independently: `00` is 1×, `01` 2×, `10` 4× and `11` 8×. Each dot is repeated, so an 8× sprite is 128 × 128 dots from the same 16 × 16 pattern (Figure 16.4).



one pattern
16 × 16 dots,
each repeated
2, 4 or 8 times
across and down

Figure 16.4: One pattern at six sizes, on the machine.

The scale belongs to the screen, not to the pattern: a sprite 4× across and 1× down is wide whether or not it is rotated, because the hardware rotates first and stretches the result. Big is cheap in memory and, as you will see, dear in time.



KLIVE IDE Stored and shown

Select a sprite, and the inspector shows its pattern **as stored** and **as shown**: rotated first, then mirrored, then scaled, the way the hardware draws it.

Anchors and relative sprites

A boss or a mothership is several sprites moved together, and moving them one by one means rewriting every position each frame. **Relative sprites** let the hardware do it: a sprite can be placed relative to an **anchor**, the last ordinary sprite before it in the table, and moves when the anchor moves.

A sprite is relative when E is set and byte 4 bits 7–6 are `01`. Every other sprite, visible or not, is an anchor, and the relative sprites that follow it, up to the next anchor, belong to it:

Byte	In a relative sprite
0, 1	X and Y **offsets** from the anchor, signed: −128 to 127
2	palette offset, mirrors, rotate; bit 0: add the anchor's palette offset to its own
3	visible, E = 1, pattern N5–N0
4	`01` in bits 7–6; bit 5: N6; bits 4–1: its own scale (composite only); bit 0: add the anchor's pattern number to its own

The anchor's T bit chooses how its relative sprites follow it:

- **Composite** (T = 0): the relatives take only the anchor's **position**. Each keeps its own mirror, rotation and scale: a ship whose turret turns on its own.
- **Unified** (T = 1): the group is **one big sprite**. The anchor's rotation and mirroring are combined with each relative's own, and its scale replaces theirs, so the whole object turns and grows as one.

In both kinds a relative sprite is **visible only if its anchor is**, and it uses the anchor's 4-bit or 8-bit form. Figure 16.5 shows both kinds: an anchor (grey hull) with two relatives, at (+16, 0) (blue) and (0, +16) (green).

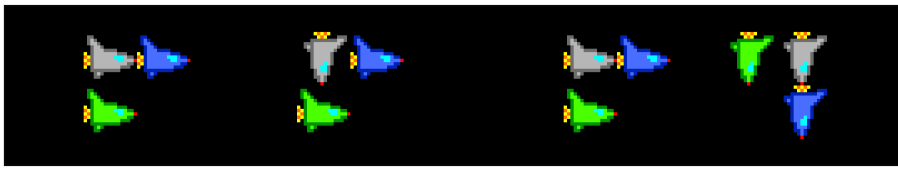


Figure 16.5: Rotating the anchor: one bit of one sprite, on the machine.

In a unified group the offset is transformed too. Rotation takes (x, y) to $(-y, x)$, X mirror negates x, Y mirror negates y, and the scale multiplies both by 2, 4 or 8. The two "add the anchor's" bits make objects cheap to animate and recolour: if an object's

frames are consecutive patterns, changing the anchor's pattern number moves every part to its next frame.



KLIVE IDE Anchors in the table

A relative sprite's row names its anchor, as ↳7 for "this sprite hangs from sprite 7" (the arrow points from the anchor to the sprite that follows it), and its type, composite or unified, and gives the **effective** position the sprite engine works out. When the anchor hides it, the note column says so, and the sprite-space map shows where the object really is.

Two lines, not a screen

Now for the key idea. The sprites have no picture memory. They have two **line buffers**, each 320 entries long, one for every dot of a line of the wide area. An entry is **9 bits**: the palette index of a sprite dot, and a bit that says "a sprite dot is here". I picture them as **two easels**: one stands in front of the audience, the other is backstage, being painted.

While the beam draws line *v*, one buffer is **shown**, read out at the 7 MHz dot rate and cleared as it is read. Meanwhile the **sprite engine** fills the other with line *v* + 1: it goes through the sprite table, sprite 0 first, and for every sprite with a row on that line writes the row's dots at the sprite's X, one dot per 28 MHz clock. When the line ends, the buffers swap (Figure 16.6).

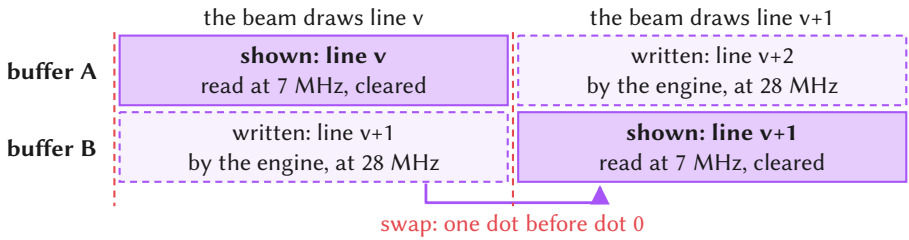


Figure 16.6: The two line buffers: the engine always works one line ahead of the beam.

The engine can write any of the 320 entries at any time, so a sprite drawn later can land **to the left** of one drawn earlier, and the line still comes out right, because nothing reads the buffer until the whole line is in it. I find this the cleverest point of the design: the hardware never sorts anything.

A line's sprites are drawn in the line before it is shown, from the attribute table **as it is at that moment**. So a program that rewrites a sprite below the beam can use it again further down the same frame (Chapter 18 puts this to work).

The sprite layer answers "opaque" where the buffer's ninth bit is set, and its index goes through the sprite palette to composition (Chapter 17). Bit 0 of NextReg $15 only gates this output: the engine keeps drawing while the sprites are switched off, and the collision flag keeps counting.

Which sprite is in front

When two sprites cover the same dot, both write it, and the engine always works from sprite 0 to sprite 127, so normally the **later write wins**: sprite 127 is in front. Bit 6 of NextReg $15 turns it round: a dot is written only where no sprite has written yet, so the **first write wins** and sprite 0 is in front. The drawing order and the time it takes are the same. I prefer this setting for games, because the sprite you care about most, at the top of the table, is then also the one in front.

The time budget of a line

The engine runs at 28 MHz, four clocks for every dot the beam moves. A 128K/+3 line is 456 dots, so the engine has **1,824 clocks** from one swap to the next (1,792 at 48K or Pentagon timing; the numbers below are for 128K/+3). The swap comes one dot before dot 0, so at clock c the beam is at dot $c \div 4 - 1$, rounded down. The beam is simply the engine's stopwatch here: it is showing the line before the one being drawn.

The engine goes through the table starting 3 clocks after the swap:

- A sprite **not on the line**, because it is hidden or has no row there, costs **1 clock**.
- A sprite **on the line** costs 1 clock and then **1 for every dot it walks**: 16, 32, 64 or 128. Transparent dots cost a clock too, and so do dots off the left of the line. Only a sprite that runs off the **right** of the line stops early, at dot 320.

So a 1× sprite costs **17 clocks**, and an 8× one 129. Dividing 1,824 by 17 suggests that about 107 fit on a line. In fact only **75** do when every sprite is on it, because of one more rule.

The cut-off window

Near the end of every line there is a short stretch in which the engine may not begin a sprite: the sprite's **cut-off window**. Its length is that sprite's own drawing time, so the wider the sprite, the earlier it opens, and all four end just before the beam reaches dot 320:

Sprite scale	The beam at	The engine's clocks	Its length
1×	dots 316–319	1,268–1,283	16 clocks
2×	dots 312–319	1,252–1,283	32 clocks
4×	dots 304–319	1,220–1,283	64 clocks
8×	dots 288–319	1,156–1,283	128 clocks

Where the sprite is on the screen makes no difference: what decides is how late in the line the engine gets to it. Each time the engine gets to a sprite that is on the line, it looks at the beam. If the beam is in that sprite's cut-off window, the engine **gives up** on the line: it draws neither this sprite nor any after it, and sets the "too many sprites" flag. Four things follow:

- What counts is the moment the engine gets to a sprite. One reached just before its window is drawn to its end.
- A sprite that is not on the line never makes the engine give up: its check does not look at the beam.
- Once the beam is past dot 319, the engine draws sprites again until the next swap.
- The swap is the one real deadline. It stops the engine even in the middle of a sprite, which can then appear cut short on its right, and that also sets the flag.

Two examples on the machine

Figure 16.7 puts two cases on one scale, the screen above the engine's work. Every sprite is a solid 16 × 16 square in one of 16 colours, and they are only 2 dots apart, so each covers all but the first 2 dots of the one before.

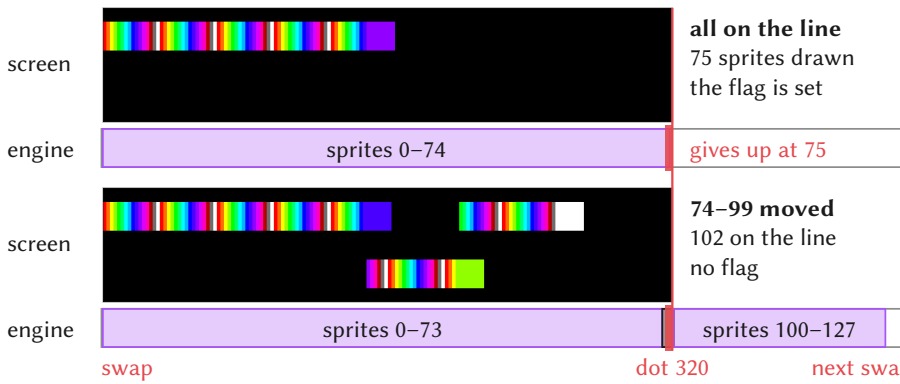


Figure 16.7: The same 128 sprites, all on the line (top), and with 74–99 moved away (bottom).

All on the line. All 128 are 1× and on the same 16 lines, sprite k at X = $2k$. Sprite k starts at clock $3 + 17k$.

1. Sprite 74 starts at clock 1,261, with the beam at dot 314: before the window, so it is drawn.
2. Sprite 75 comes up at clock 1,278, with the beam at dot 318: inside it, so the engine gives up.

That leaves **75** sprites. At 2×, 4× and 8× the wider windows arrive after **38**, **19** and **9**.

Sprites 74–99 moved 32 lines down.

1. Sprites 0–73 are drawn as before, and sprite 73 ends at clock 1,261.
2. Sprites 74–99 are not on this line, so the engine only checks them, 1 clock each. None of the checks looks at the beam, so these 26 clocks, up to 1,287, carry the engine across the window.
3. Sprite 100 comes up at clock 1,287, with the beam at dot 320. The window is behind it, so it is drawn. Sprite 127 ends at clock 1,763.

That is **102** sprites on one line, 27 more, and no flag. Moving sprites *away* from a line let more of them onto it. I would never have guessed it without counting.

What a program can count on

- The flag is bit 1 of port `$303B` when read. It is **set** the moment a line loses sprites, and it **stays set**. **Reading** the port returns it and **clears** it, together with collision flag bit 0. So it says that some line lost sprites since the last read, not which line or how many.
- Sprites low in the table are safe, because the engine always starts at sprite 0. Put the player and the bullets there.
- Every sprite costs at least one clock on every line, visible or not. A line whose work is done before clock 1,156, when the first window opens, can never lose a sprite.
- Wide sprites are expensive: one 8× sprite costs as much as seven and a half 1× ones.

Over the border, and clipping

By default the sprites are shown only over the Spectrum's display area. Bit 1 of NextReg `$15` lets them **over the border**. What is shown is decided as the buffer is read out: the engine draws every sprite across the whole line, and the Clip Window Sprites register, **NextReg $19**, says which dots may show (Chapter 9). Its meaning depends on two bits of NextReg `$15`, bit 1 (over the border) and bit 5 (clip):

Over	Clip	Where sprite dots show
0	either	inside the window, in display-area units: dots 32 + X1 to 32 + X2, lines 32 + Y1 to 32 + Y2, never below line 223
1	0	the whole wide area; the window is ignored
1	1	inside the window, X doubled: dots 2 × X1 to 2 × X2 + 1, lines Y1 to Y2

After a reset the window is X 0–255, Y 0–191, the display area exactly.

Collisions

Did your ship just fly into a rock? Bit 0 of port `$303B`, when read, says whether two sprites **collided**: whether an opaque sprite dot was written into the line buffer where another sprite had already written one. The flag is exact to the dot, not a test of the sprites' boxes: transparent dots are never written, and sprites side by side do not collide. A sprite the engine gave up on writes nothing, so it collides with nothing.

Because the check is made in the line buffer, it follows the buffer's rules, not the screen's: dots outside the clip window collide, overlaps hidden by "sprite 0 on top" collide, and sprites switched off with NextReg `$15` bit 0 still collide. And any two sprites count, including the parts of one object: build your ship from overlapping pieces, and the flag is set every frame, rock or no rock.

The flag says *that* two sprites touched, not *which* two or where. A game reads it once a frame as a cheap "something hit something", and only then compares the positions of the sprites it cares about.



KLIVE IDE Collisions without side effects

The inspector's globals strip shows **collision** and **too many** from port `$303B` without clearing them, so the program's own check still sees the flags. **Break on attribute write**, in a row's menu, stops whenever that sprite's attributes change.

▶ TRY IT Sprite Swarm

Part II, key 7. Sixty-four balls bounce round the screen, a rock sits in the middle, and four Fs make up an anchor and three relative sprites. Once a frame the demo uploads 69 sprites through port `$57` and reads port `$303B`; row 21 lights "Collision" when bit 0 is set. The code is in `examples/code/demos/2-picture/sprite-swarm.kz80.asm`, and this is the anchor's upload:

```
    ld bc,$303B
    xor a
```

▶ TRY IT Sprite Swarm *(continued)*

```
    out (c),a               ; sprite 0
    ld hl,(SsX)
    ld a,l
    out ($57),a             ; byte 0: X
    ld a,(SsY)
    out ($57),a             ; byte 1: Y
    ld a,(SsRotate)
    or h                    ; bit 0: X8
    or $F0                  ; palette offset $F
    out ($57),a             ; byte 2
    ld a,%11000001          ; visible, E: byte 4 follows, pattern 1
    out ($57),a             ; byte 3
    ld a,(SsUnified)
    out ($57),a             ; byte 4: T, an 8-bit pattern, 1x
```

Press S to hide the swarm: the flag goes out, and comes back only when you steer the Fs into the rock with Q, A, O and P. R sets the anchor's rotate bit, and only the anchor turns; U sets T, and all four turn as one.

In short. The 128 sprites have no picture memory, only two 320-entry line buffers filled one line ahead of the beam, sprite 0 first. A sprite costs one clock to check and one per dot of width, so at 128K/+3 timing only 75 1× sprites fit when all are on the line. If the engine reaches a sprite inside its cut-off window, that sprite and all later ones are lost, so put what matters low in the table. Reading port `$303B` returns and clears both the too-many-sprites and collision flags, and overlapping parts of one object collide every frame.

CHAPTER 17

Putting It Together: Four Layers, One Colour

Four candidates for every dot, and one winner. The rules that pick it fit on a page, yet they give the Next layers that can swap places on every line, stencils that cut one picture to the shape of another, and colours that add up across layers. They also decide what your player sees: whether your hero walks in front of the hills or behind them, and whether a sprite disappears behind the border.

Chapter 9 introduces the two answers every layer gives for every dot and the pipeline that carries them, and calls **stage 2**, the step that turns four answers into one colour, **the referee**. Chapter 11 explains how each layer's colour and transparency are made. Here we follow what stage 2 does with them.

- Why a layer can have nothing to show at a dot
- How the ULA and the tilemap combine into one layer
- How the six orders of the layers work, and their two exceptions
- How blending adds colours instead of choosing one

Four answers, one colour

For one dot, stage 2 gets four colours, one from each layer, and with each the layer's answer to "am I transparent here?" and a few flags. It makes one 9-bit colour in two steps. First the ULA and the tilemap are **mixed into one layer**, called **U**. Then the three remaining layers, **S** (sprites), **L** (Layer 2) and **U**, are put **in order**, and the first one that is not transparent wins (Figure 17.1).

All the settings stage 2 uses are taken once for each dot as it enters the pipeline, and they travel with it. So your program, or the Copper, can change them between two dots of the same line.

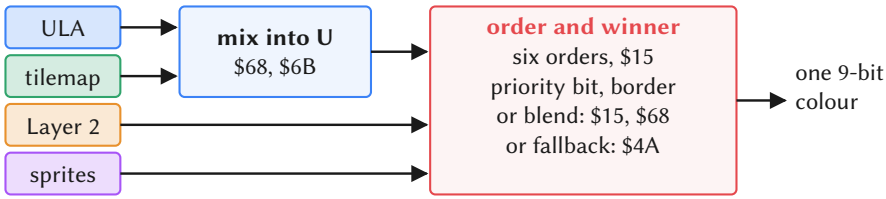


Figure 17.1: Stage 2: four layers in, one colour out.



KLIVE IDE When the picture looks wrong

Klive's guide *Why does my Next picture look wrong?* gives a routine: pause where the picture is wrong, probe the dot with the Layers strip, then open the inspector of the layer that drew it. It also has a table from symptom to tool. This chapter explains what the probe tells you.

When a layer is transparent

A layer is transparent at a dot for one of three reasons, and stage 2 does not care which: it is switched off, the dot is outside its clip window, or the dot fails its own test.

Layer	Switched off by	Clip	Transparent when
ULA, LoRes	NextReg `$68` bit 7	`$1A`	its colour equals NextReg `$14`
Tilemap	NextReg `$6B` bit 7 clear	`$1B`	its dot equals NextReg `$4C` (text mode: its colour equals `$14`)
Layer 2	port `$123B` bit 1 clear	`$18`	its colour equals NextReg `$14`
Sprites	NextReg `$15` bit 0 clear	`$19`	no sprite covers the dot, or its dot equals NextReg `$4B`

Two things are easy to miss. The ULA's colour test applies to its border too: a border whose colour equals NextReg `$14` is transparent. And switching the ULA off removes it **as a layer**, but not as a source for blending.

The ULA and the tilemap become one

The two are mixed by the rule for any two layers: the upper one shows where it is opaque, the lower one where the upper is transparent, and U is transparent only where both are. Which is upper is decided **tile by tile**, by bit 0 of the tile's attribute, unless bit 0 of NextReg `$6B` puts the whole tilemap on top. I like that the tile decides for itself: one map can put its platforms in front of the Spectrum picture and its sky behind it.

Stencil mode, bit 0 of the ULA Control register, NextReg `$68`, turns the mix into something else. With both layers switched on, U is opaque only where **both** are opaque, and its colour is their two colours combined bit by bit with **AND**. Each layer cuts the other to its own shape: a tilemap of solid colour shows only through the ULA's ink. The AND also changes the colours: a ULA yellow, `101 101 000`, and a tile cyan, `000 111 111`, give the green `000 101 000`. White lets the other layer's colour through unchanged, so a stencil can tint as well as cut.



KLIVE IDE One layer of two

At a dot, the pixel probe shows the tile's "below the ULA" bit and the ULA's border flag, so you can see which of the pair became U. The Layers document draws the ULA and the tilemap as one pair in its priority stack, with stencil mode when it is on.

Six orders

Bits 4–2 of the Sprite and Layers System register, **NextReg $15**, choose how S, L and U are stacked. Three layers have 3 × 2 × 1 = 6 orders; the two other values are the blend modes. Stage 2 takes the first layer, top first, that is not transparent. An order is written **top layer first**: SLU has the sprites on top, Layer 2 in the middle and the ULA at the bottom.

A Layer 2 backdrop with a Spectrum-style status bar wants the ULA in front of Layer 2, and sprites that walk behind scenery want to be below it. My rule of thumb: put on top the layer that must never be hidden. Figure 17.2 shows all six, with four points marked where two layers overlap: **A** a sprite on a hill, **B** the ULA's score panel on the edge of a hill, **C** a platform tile on a hill, **D** a sprite on the ground of tiles.

Bits	Order	A: S / L	B: U / L	C: U / L	D: S / U
000	SLU	sprite	Layer 2	Layer 2	sprite
001	LSU	Layer 2	Layer 2	Layer 2	sprite
010	SUL	sprite	ULA	tile	sprite
011	LUS	Layer 2	Layer 2	Layer 2	tile
100	USL	sprite	ULA	tile	tile
101	ULS	Layer 2	ULA	tile	tile

The machine starts with SLU. Read a column and you read one pair of layers: at A the sprite wins exactly where S is above L, at D exactly where S is above U. An order is nothing more than three such duels.

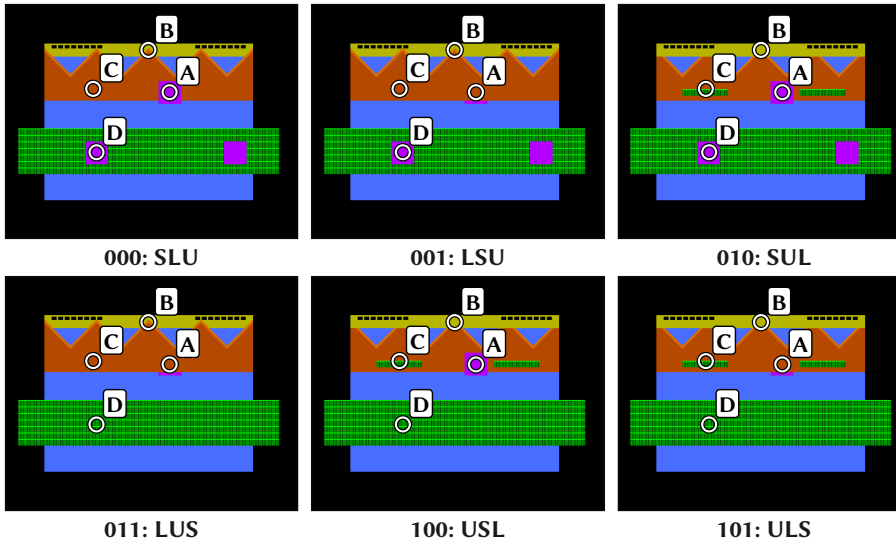


Figure 17.2: The same frame in the six orders of NextReg $15 bits 4–2.



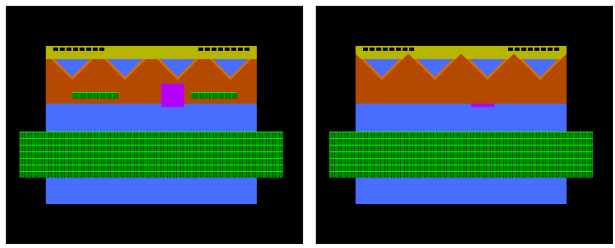
KLIVE IDE The order in force

The Layers strip starts with the order in force, `SLU`, `LSU` and so on, and follows NextReg $15 as it changes, the Copper's changes included. Under a Copper split the probe gives the order for the dot's own line, and `bp-set nr:$15` finds the code that changed it.

Two exceptions

The Layer 2 priority bit. A Layer 2 palette entry can carry a **priority bit** (bit 7 of the second write to NextReg $44). A Layer 2 dot whose entry has it, and which is not transparent, goes **above every other layer**, whatever the order says, and in the blend modes too. The bit belongs to the palette entry, so one colour of a picture can stand in front of the sprites, a window frame or a foreground column, while the rest stays behind them (Figure 17.3). Only the hills' body rises there; their edge, another entry, stays below.

Sprites over the border. In LUS, USL and ULS the ULA would hide every sprite behind its border, because the border is an opaque ULA dot. So where U's dot is the **border**, the tilemap is transparent and a sprite is opaque, the **sprite** wins. Sprites allowed over the border (bit 1 of NextReg $15) stay visible there in every order, which is exactly where a ship flies in from.



left: USL
as it is

right: USL
entry $A8 has
the priority bit

Figure 17.3: The priority bit on the machine.



KLIVE IDE Naming the exception

The probe says which rule chose a dot's colour: the top opaque layer in the order, the Layer 2 priority bit, or a sprite over the ULA's border. The two exceptions of this section are named at the very dot where they win.

Blending: adding instead of choosing

Values **110** and **111** of NextReg $15 bits 4–2 do not order Layer 2 and U at all. Here the referee stops choosing and **adds their colours**: each channel, levels 0 to 7, is added, and the sum, 0 to 14, is brought back into 0 to 7.

- **110: add and clamp.** A sum above 7 becomes 7. Adding brightens.
- **111: add, then subtract 5.** A sum up to 4 becomes 0, a sum of 12 or more becomes 7, and anything between loses 5. With a source at level 5, Layer 2 is unchanged; a lighter source lightens it and a darker one darkens it.

Take Layer 2 at levels 5, 2 and 0 with a source of 5, 5 and 0: the sums are 10, 7 and 0. Mode 110 gives 7, 7 and 0, a bright yellow, and mode 111 gives 5, 2 and 0, Layer 2 as it was.

What Layer 2 is added to, the **blend source**, is chosen by bits 6–5 of NextReg $68:

Bits 6–5	Blend source	Drawn separately
00	the ULA	the tilemap
10	the ULA and tilemap mix	nothing
11	the tilemap	the ULA
01	nothing (Layer 2 alone)	both

The blend modes then have a fixed order of their own, top first:

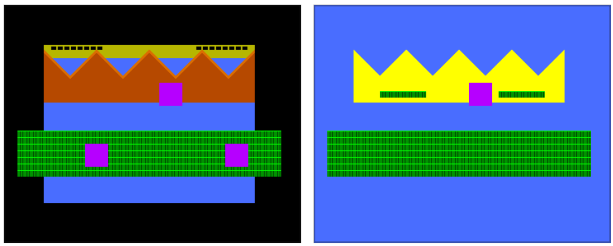
1. Layer 2 plus the source, if its palette entry has the priority bit
2. the upper of the layers drawn separately, by the tiles' below flag

3. the sprites
4. the lower of the layers drawn separately
5. Layer 2 plus the source, where Layer 2 is opaque

Two things follow, and both surprise programmers who expect "SLU with adding":

- **The source shows only through Layer 2.** Where Layer 2 is transparent, it is not shown at all. With the ULA as the source, the Spectrum screen and even its border disappear wherever Layer 2 has nothing.
- **The tilemap can rise above the sprites while Layer 2 stays below them**, a split the six orders cannot make.

Where the source itself is transparent, Layer 2 is shown as it is. Figure 17.4 shows mode 110 with the ULA as the source, its paper a yellow over the whole display area. The hills saturate to a bright yellow; elsewhere the yellow does not show and the fallback blue does, even over the border, and the tiles cover the sprites.



left: SLU
for comparison

right: 110
Layer 2 + ULA,
clamped

Figure 17.4: Blend mode 110 on the machine.

Blending is the Next's way of **lighting** a picture. Layer 2 holds the detail, and the source holds something coarse and cheap to change. Make the ULA the source, with an empty bitmap: in mode 111 a torch in a dark cave or the shadow of a cloud is a handful of attribute bytes, 768 for the whole screen, instead of a redrawn 48K picture. If every cell of the source shows one palette entry, one palette write per frame fades or flushes the whole picture. Change the source's colour between lines with the Copper, and the picture gets bands of sunset and deep water while Layer 2 stays untouched. Each channel has only eight levels, so steps are coarse, but whenever you are about to redraw a whole picture just to make it darker, I would reach for blending first.



KLIVE IDE Blending, explained

In the two blend modes, the Layers document's priority stack shows what is blended with what, and the probe names a blend as the rule that chose the dot's colour.

▶ TRY IT Layer Order

Part II, key 8. Three layers overlap: the ULA's red panel, a green block on Layer 2, and three yellow sprites. One sprite sits on the panel, one where panel and block overlap, one on the block. Black is the transparent colour (NextReg `$14` = 0), so the ULA's black paper and border show the dark blue fallback colour. Keys 1–6 write the six orders, keys 7 and 8 the two blend modes, with the ULA as the blend source. The code is in `examples/code/demos/2-picture/layer-order.kz80.asm`; this is how it writes a value:

```
    pop af
    push af
    cp 6
    ld a,$00                ; an order: no strips (transparent)
    jr c,LoStrips
    ld a,$B6                ; blending: grey strips, level 5
LoStrips:
    call LoDrawLayer2
    pop af
    add a,a
    add a,a                 ; bits 4-2
    or %00000001            ; bit 0: sprites visible
    nextreg $15,a
    ret
```

Compare each order with the table above: the overlap, with its sprite, shows all three duels at once. In the blend modes the panel vanishes except where it meets Layer 2: yellow under 110, a dark olive under 111. The text stays only because Layer 2 draws grey strips behind it, at level 5, which mode 111 leaves unchanged. In Klive IDE, hide or solo a layer, or ask a pixel which layer drew it.

When nothing is opaque

When every layer is transparent at a dot, in any mode, stage 2 shows the **fallback colour**, held in NextReg `$4A` (`$E3` after a reset). Programs usually set it to black, or, as the pictures here do, to a sky blue that then needs no layer at all.

In short. Stage 2 mixes the ULA and the tilemap into one layer, U, then shows the first opaque layer in the order NextReg `$15` bits 4--2 choose, or the fallback colour of NextReg `$4A` if none is. Two exceptions beat the order: a Layer 2 palette entry with the priority bit rises above everything, and an opaque sprite beats the border where the tilemap is transparent. Values `110` and `111` add Layer 2 to a blend source. That source shows only through Layer 2, so where Layer 2 is transparent, a ULA source and even its border disappear.

CHAPTER 18

Raster Effects: Changing the Picture Mid-Frame

Every screen chapter so far made one quiet assumption: the registers stay still while a frame is drawn. They do not have to. The machine draws the picture one dot at a time, and every dot uses the registers as they are at that moment. Change a palette entry, a scroll offset or the layer order at the right line, and one screen becomes several. I think this is where the Next's screen stops being a set of modes and becomes an instrument.

It gives a sky with more colours than the palette holds, a playfield that scrolls under a status bar that does not budge, and more sprites than the machine has. The CPU can do all of this, if somebody tells it when the beam reaches a line. But the Next also has a small processor whose only job is to do it on time: the **Copper**, which waits for a position of the beam and then writes a register. I think of it as **the stage manager**: it never paints a dot, but it calls every cue at exactly the same moment, frame after frame.

This chapter builds on the line numbers and the line interrupt of Chapter 10, and on the pipeline of Chapter 9. Its numbers are for 128K/+3 timing at 50 Hz: 311 lines of 456 dots, with the CPU at 3.5 MHz.

- Why the beam is a clock, and when a write takes effect
- How the line interrupt splits the screen, and why it shimmers
- How the Copper works: its two instructions, its memory and its four modes
- How to build a sky gradient, a split scroll and thirty-two ships from eight sprites

The beam as a clock

The picture is not stored anywhere. Every dot is made at the moment the beam reaches it, from the layers' memory and from the registers as they are **then**. So a register written in the middle of a frame changes the picture **from where the beam is**: the part drawn before the write keeps the old value, the rest gets the new one.

That makes the beam a clock, and every position on the screen a time: 456 dots a line, 311 lines a frame, the same every frame. Write a register at the same position

in every frame, and you get a steady picture with a band in it. Write it at a slightly different position each time, and the band's edge flickers. Everything that follows is about hitting the same position every time.

When a write shows

"From where the beam is" means slightly different things for different parts of the picture, because not every setting is read at every dot:

Settings	Taken	So a write shows
Palettes, layer order, transparency and fallback colours, the ULA off switch, Layer 2, LoRes, the ULA's clip window	at every dot	from the next dot
The ULA's scroll, Timex mode and border colour (Chapter 12)	at every 8-dot cell	from the next cell
The tilemap's settings (Chapter 15)	a character ahead	a cell or two later
The sprites' attributes (Chapter 16)	a line ahead	from a later line
The line offset, NextReg `$64` (Chapter 10)	at line 0	at the next frame

For most effects the first row is all that matters: a palette entry or a scroll value written at the start of a line shows on that whole line. Screen **memory** is different again: a byte written to the ULA's screen or to Layer 2 shows when the layer next reads it, which for a byte the beam has already passed is the next frame.

The line interrupt and its jitter

The Next can interrupt the CPU before a line of your program's choosing: the **line interrupt**, switched on by bit 1 of **NextReg $22**, with the line in bit 0 of `$22` and **NextReg $23**. It is raised on the line before, near the end of its display area, in the line numbers that NextReg `$64` sets (Chapter 10).

Your handler can then write the registers for the next band, but the CPU answers an interrupt only when the instruction it is running has finished. At 3.5 MHz an instruction takes up to 23 T-states, so the handler starts anywhere from 0 to 22 T-states late, and one T-state is two dots. A split made this way lands up to **44 dots** later in one frame than in another, more if contention stretches the instruction, and its edge shimmers (Figure 18.1).

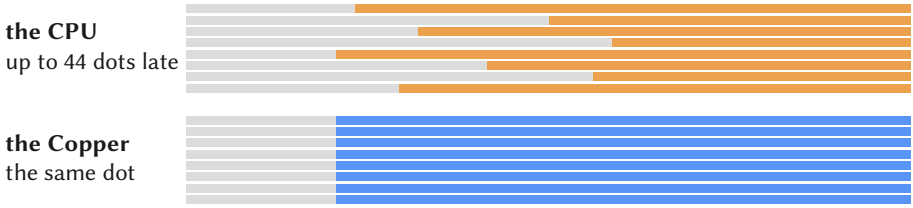


Figure 18.1: The same split made two ways, over eight frames.

Waiting in `HALT` removes most of the jitter: a halted CPU runs NOP cycles of 4 T-states and answers at the end of one, so only up to 6 dots are left. But then your program can do nothing else while it waits, and every split still costs an interrupt and a handler.

The Copper's two instructions

The **Copper** is a co-processor with exactly one job: wait for a position of the beam, then write a NextReg. It runs a list of instructions from its own memory, alongside the CPU, and costs the CPU nothing once it is started. It knows just two instructions (Figure 18.2).

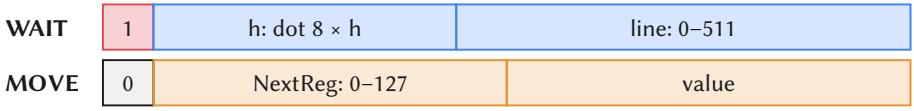


Figure 18.2: The Copper's two instructions, bit 15 on the left.

- **WAIT** (bit 15 = 1) waits until the beam is on **line** *n*, bits 8–0, at or past the horizontal position *h*, bits 14–9. The line is counted in the numbers that NextReg `$64` sets, the same as the line interrupt's: line 0 is the first line of the display area, or of the wide area with an offset of 32. The position *h* is matched at **dot 8 × *h*** of the display area, so a WAIT can stop at any 8-dot boundary.
- **MOVE** (bit 15 = 0) writes the **value** in bits 7–0 to the **NextReg** in bits 14–8, `$00`–`$7F`: the palettes, the layer settings, the scroll registers, the sprite attributes and the MMU are all there. A MOVE to NextReg `$00` writes nothing; it is the Copper's NOP.

Two instructions, and no jumps, no loops, no arithmetic.



KLIVE IDE The Copper List

View › Machine Views › Copper List, or `show-copper`, lists the Copper's memory one instruction a row, decoded: the line and position of a WAIT, and for a MOVE the register's name and the value, with a colour swatch for a palette write.

Where seven WAITs stop

Here are seven WAITs, with NextReg `$64` at 0:

	Instruction	Encoded	The Copper goes on when the beam reaches
1	WAIT line 0, *h* 0	`$8000`	dot 0 of line 0: the display area's top-left corner
2	WAIT line 96, *h* 16	`$A060`	dot 128 of line 96: the middle of the display area
3	WAIT line 191, *h* 31	`$BEBF`	dot 248 of line 191: the last cell of the last line
4	WAIT line 200, *h* 0	`$80C8`	dot 0 of line 200: the bottom border, 9 lines below the display area
5	WAIT line 300, *h* 34	`$C52C`	dot 272 of line 300: the top border, right of the display area
6	WAIT line 99, *h* 51	`$E663`	dot 408 of line 99: the left edge of line 100's border
7	WAIT line 278, *h* 53	`$EB16`	dot 424 of line 278: the top-left corner of the 320 × 256 wide area

Example 6 catches almost everyone out. The Copper's horizontal count starts **12 dots before** the display area, and its line number moves on there, at dot −12. So the border to the left of line 100, dots −48 to −13, is still part of line 99 for the Copper: dot 408 of line 99 and dot −48 of line 100 are the same place.

Example 7 follows from that. The wide area begins 32 dots left of the display area and 32 lines above it, so its corner is dot −32 of line 279. Dot −32 is left of dot −12, so for the Copper it is line 278, at dot −32 + 456 = 424, which is 8 × 53. Both numbers depend on the timing, and NextReg `$64` at 32 makes the wide area's first line the Copper's line 0:

Timing	Frame	The corner	WAIT, $64 = 0	WAIT, $64 = 32
128K/+3	456 × 311	dot −32, line 279	line 278, *h* 53: $EB16	line 310, *h* 53: $EB36
48K	448 × 312	dot −32, line 280	line 279, *h* 52: $E917	line 311, *h* 52: $E937
Pentagon	448 × 320	dot −32, line 288	line 287, *h* 52: $E91F	line 319, *h* 52: $E93F

On a 456-dot line, *h* = 55, dot 440, is the last position a WAIT can reach; on the 448-dot lines it is 54. And a WAIT compares the line only for **equality**. If the Copper reaches a WAIT for a line the beam has already passed, it waits for that line in the next frame.



KLIVE IDE The raster ruler

Beside the Copper List, a strip draws one frame in the Copper's own line numbers, line 0 at the first paper line, with a tick for every WAIT, the live beam and the last hit. It shows at a glance where in the frame a list acts.

The Copper's memory

The Copper has **2K** of its own memory, **1,024 instructions**, each with its high byte at the even address. Your program writes it through NextRegs:

- **NextReg $61** and bits 2–0 of **NextReg $62** set the **byte** address, 0–$7FF.
- **NextReg $63** takes an instruction in two writes, the high byte first, and writes **both bytes at once** on the second, so a list can be rewritten while it runs without the Copper ever reading half an instruction.
- **NextReg $60** writes one byte at a time.

Both move the address on after every write. Use $63 for whole instructions, and $60 only for the rare byte you mean to write alone.



KLIVE IDE The Copper panel

The **Copper** panel in the **Debug** activity shows the mode, the state, such as `Waiting for line 96, x 64`, the Copper's PC and where the CPU's next byte through $60 or $63 will go. After a soft reset the panel still shows the old list, marked as retained: the memory survives, as on the Next.

Ending a list, and the four modes

The Copper has **no end instruction**. It goes on to the next instruction in its memory, and after instruction 1,023 it goes on with instruction 0. What lies beyond the last instruction your program wrote is whatever the memory held before: an older list, or zeros, which are NOPs. So a list ends itself with a **WAIT that never comes true**:

one for line 511, which no frame reaches, or with *h* of 56 or more. `$FFFF` is the usual choice, and the Copper sits on it until something moves it on.

Bits 7–6 of the Copper Control register, **NextReg $62**, choose what the Copper does:

- **00, stopped.** The Copper keeps its place in the list. Use it while a new list is written.
- **01, run from the start.** The Copper starts at instruction 0 and never restarts by itself, so a list that ends with `$FFFF` runs **once**. Without WAITs, it writes registers two clocks a MOVE: 256 palette entries in 128 dots.
- **10, carry on.** The Copper runs on from the instruction where it stopped, so a program can stop it briefly to rewrite instructions further down and let it continue with the same frame.
- **11, every frame.** Like 01, but the list also starts again from instruction 0 at **line 0** of every frame. This is the mode for effects.

Writing the mode the Copper already has changes nothing. Changing to 01 or 11 always starts from instruction 0; changing to 10 does not.



KLIVE IDE Three checks on a list

The Copper List flags a list with no HALT, a WAIT that never matches (such as a list written for 50 Hz, which parks the Copper at 60 Hz), and a WAIT for an earlier line than the one before it, which in a looping mode waits for the next frame.

Why the Copper is exact

The Copper runs at **28 MHz**, four clocks for every dot, and checks its WAIT at **every clock**. A MOVE takes two clocks: half a dot. Its writes go through the same path as the CPU's NextReg writes, but **ahead** of them, so the CPU can never delay a Copper write. Nothing in it depends on what the CPU is doing, so a list does the same thing at the same dot in every frame, with no shimmer, no handler and no interrupt.

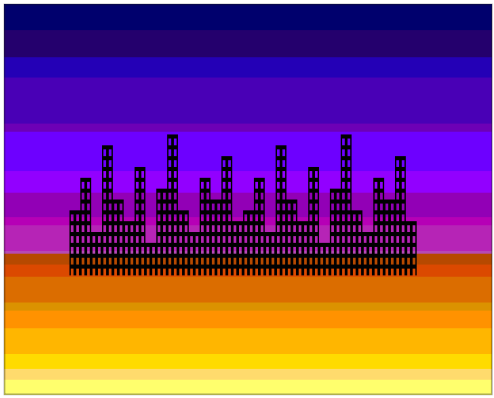


KLIVE IDE Stopping on an instruction

Click a row's gutter in the Copper List, or type `bp-set cu:$00B`, and the machine stops when the Copper completes that instruction; the list shows the beam at the hit. **Step Copper** runs on to the Copper's next instruction. Two MOVEs are only two 28 MHz ticks apart, so one Z80 instruction can see both.

A sky gradient from one palette entry

A palette entry is one colour, but nothing says it has to be the same colour on every line. Here the Copper rewrites ULA palette entry 16, which is both the border colour 0 and paper 0, before every one of the frame's 311 lines (Figure 18.3).



the whole frame
border and paper:
ULA palette entry 16,
a new colour
on every line

Figure 18.3: One palette entry, 311 colours.

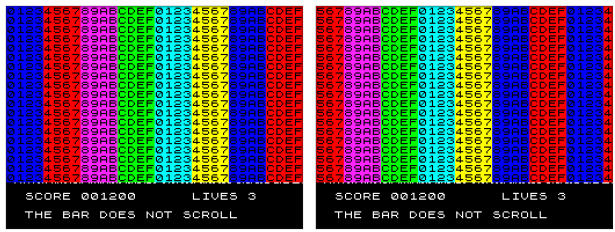
The list is 625 instructions: two to point the palette at entry 16, then a WAIT and a MOVE for every line. The WAIT does not stop at dot 0 of the line. The border to the left of the display area belongs, for the Copper, to the line before (see "Where seven WAITs stop"), so a colour written at dot 0 would reach the paper one line before it reaches the left border, and the two would be a line out of step. Each WAIT instead stops at **dot 408 of the line before**, which is dot −48 of the line it colours: the left edge of its border.

```
MOVE $43, $80          ; index stays put after a write
MOVE $40, 16           ; ULA palette entry 16
WAIT 0, 51             ; dot -48 of line 1
MOVE $41, colour(1)
WAIT 1, 51             ; dot -48 of line 2
MOVE $41, colour(2)
…                      ; to line 310, the last line above line 0
WAIT 310, 51           ; dot -48 of the next frame's line 0
MOVE $41, colour(0)
WAIT 511, 63           ; stop until the next frame's line 0
```

The ULA's picture, a skyline in ink colour 0, never changes; only the colour of its paper does. I think this is the prettiest effect for the least work on the whole machine.

A status bar that stands still

A status bar that stays still while the playfield scrolls is two values of one register in one frame: the ULA X Scroll register, **NextReg $26**, holds the playfield's scroll for lines 0–151 and 0 for the bar, lines 152–191 (Figure 18.4).



left: no scroll

right: $26 =
0 on lines 152–191,
40 elsewhere

Figure 18.4: A split scroll: the playfield moves, the status bar does not.

The list is five instructions, run in mode 11:

```
WAIT 151, 33          ; line 151, dot 264: right border
MOVE $26, 0           ; no scroll: the bar, lines 152-191
WAIT 191, 33          ; line 191, dot 264: after the bar
MOVE $26, 40          ; playfield scroll, for next line 0
WAIT 511, 63          ; stop until the next frame
```

1. At line 0 the scroll register already holds 40, written at the end of the previous frame, so the playfield is drawn scrolled 40 dots to the left.
2. At dot 264 of line 151, in the right border just after the playfield's last dot, the Copper writes 0, so the bar's first line, 152, starts with the new value.
3. At dot 264 of line 191, after the bar, it writes 40 again, ready for line 0 of the next frame.
4. The final WAIT never comes true; mode 11 starts the list again at line 0.

Why not WAIT for line 152 itself, at *h* 0? Because the ULA reads each cell's scroll a little **before** it shows the cell, so a write at dot 0 comes too late for the line's first cell, which would keep the old scroll. A split that must be clean from the very first dot is written in the border of the line before.

To scroll the playfield, the CPU changes one byte: the value of the second MOVE. That is instruction 3, at byte address 6, so its value byte is at address 7, written through NextRegs `$61`, `$62` and `$60` once a frame. The bar is never touched.



KLIVE IDE **A patched value**

Built with `.copper`, each row of the Copper List shows its source line. A value the program changes at run time, like this list's second `MOVE $26`, is marked * in amber, and the tooltip gives the word the assembler wrote.

▶ TRY IT Copper Sky

Part II, key 9. The demo puts two of this chapter's lists together: a sky gradient from ULA palette entry 16 behind a skyline in INK 0, and a status bar, lines 152 to 191, that

▶ TRY IT Copper Sky *(continued)*

stands still while the skyline scrolls above it. Once a frame the CPU writes one byte: the value of the second `MOVE $26`. The code is in `examples/code/demos/2-picture/copper-sky.kz80.asm`, and the list, 629 instructions, in `copper-sky-data.kz80.asm` beside it. The list is written with `.copper`, a Klive assembler pragma that turns each line into one Copper word, and `CsSkyLine` is a macro of two such lines that works out the line's colour. It is a convenience of Klive's Z80 assembler, not part of the hardware:

```
CsList:
    .copper move $43, $80   ; ULA palette 1, index kept
    .copper move $40, 16    ; entry 16: PAPER 0 and border 0
    .for cl = 1 .to 151
    CsSkyLine(cl, cl - 1)
    .next
    .copper wait 151, 33    ; line 151, dot 264: right border
    .copper move $26, 0     ; no scroll: the bar, lines 152-191
    .for cl = 152 .to 191
    CsSkyLine(cl, cl - 1)
    .next
    .copper wait 191, 33    ; line 191, dot 264: after the bar
CsScrollMove:
    .copper move $26, 0     ; playfield scroll: the CPU's byte
    .for cl = 192 .to 310
    CsSkyLine(cl, cl - 1)
    .next
    CsSkyLine(0, 310)       ; line 0 of the next frame, from the end of
this one
    .copper halt            ; WAIT 511, 63: stop until the next frame
CsListEnd:
```

In Klive IDE, the Copper view shows the live list next to these lines. The border changes colour on every line too, because border 0 is entry 16 as well.

Thirty-two sprites from eight

The sprite engine draws each line from the attribute table as it is when it reaches that line, a line ahead of the beam (Chapter 16). So a sprite can be shown in one band, then moved below the beam and shown again further down. Here the Copper moves sprites 0–7 three times a frame, through **NextReg $34**, which selects the sprite, and **NextRegs $35–$37**, its first three attribute bytes (Figure 18.5).

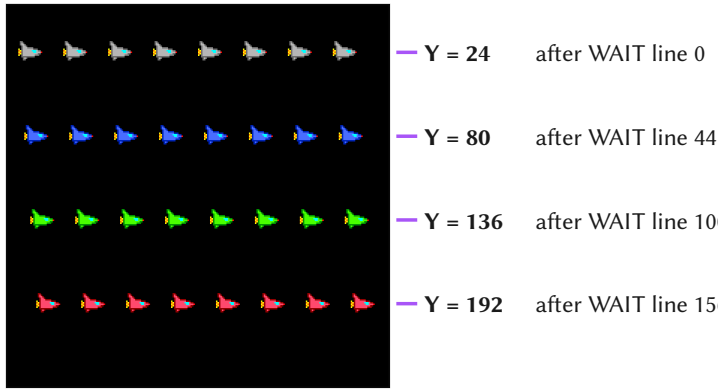


Figure 18.5: Thirty-two ships from eight sprites.

The list, run in mode 11 with NextReg $64 set to 32:

```
WAIT 0, 0              ; the top of the wide area
MOVE $34, 0            ; select sprite 0
MOVE $35, 8            ; X = 8
MOVE $36, 24           ; Y = 24: band 0
MOVE $37, $00          ; palette offset 0: the grey hull
MOVE $34, 1            ; sprite 1 ...
MOVE $35, 38
MOVE $36, 24
MOVE $37, $00
…                      ; the same for sprites 2-7
WAIT 44, 0             ; band 0 ended at line 39: free
MOVE $34, 0
MOVE $35, 12           ; X = 12: a little further right
MOVE $36, 80           ; Y = 80: band 1
MOVE $37, $10          ; palette offset 1: the blue hull
…                      ; sprites 1-7
WAIT 100, 0            ; band 2: Y = 136, palette offset 2
…
WAIT 156, 0            ; band 3: Y = 192, palette offset 3
…
WAIT 511, 63           ; stop until the next frame
```

NextReg $64 at 32 makes the Copper's line 0 the first line of the wide area, so a Copper line number is a sprite Y, and the WAITs can be written in the sprites' own coordinates. Eight sprites take 32 MOVEs, 64 clocks: 16 dots. Band 0 covers lines 24–39, so the engine has finished with its sprites once line 39 has been drawn, and band 1, from line 80, needs them from line 79: any WAIT from line 40 to 78 works.

The table has 128 sprites, so 32 ships do not need this; they only show the trick. You need it when a game wants more sprites on the screen than the table holds, or more on one line than the time budget allows.

One dot's way through the machine

Together, the ten chapters of this part follow one dot's way through the machine. Each producer reads its own memory and makes a dot for every position of the beam; the compositor picks one of the four layers' dots by the layer order and the transparency rules; the palettes turn indices into colours. And the registers that steer all of it can change between one dot and the next, which is what turns a set of fixed modes into a machine that can show more than any one of them.

In short. Every dot uses the registers as they are when the beam reaches it, so a mid-frame write shows from there on, though tilemap settings lag a cell or two and sprite attributes a line. A line-interrupt split can wander by up to 44 dots at 3.5 MHz; the Copper writes on the same dot every frame at no CPU cost. A WAIT tests the line only for equality, so a passed line means waiting for the next frame. Mode 11 restarts each frame, and every list must end with a WAIT that never matches, usually $FFFF.

Part III

The Sound

CHAPTER 19

How a Computer Makes Sound

A loudspeaker can do exactly one thing: push its cone out and pull it back in. That's it. Every sound you have ever heard from a computer, a beep, a chord, a voice, is nothing more than a list of positions for that cone. So how can a list of numbers sound like a voice?

You do not need any electronics to follow this chapter. Nothing here is about one machine: these are the basics that the later sound chapters build on.

- How a list of numbers stands for a sound, and how fast the numbers must come
- How a DAC and a filter turn numbers into a smooth wave
- Why too low a sample rate sounds wrong, and how a single bit can make sound
- How several sounds are added together, and how stereo works

Sound is air pushed back and forth

Sound is a change in air pressure that travels. A loudspeaker's cone moves forward and squeezes the air in front of it a little; it moves back and leaves the air a little thinner. These squeezed and thinned bands spread out from the speaker and reach your ear, where they push your eardrum in and out in the same rhythm.

Draw the pressure at one spot over time and you get a curve, the sound's **waveform**. Two things about it decide most of what you hear:

- **How far the cone moves** is the **amplitude**, which you hear as loudness. A tall waveform is a loud sound.
- **How often the cone moves back and forth** is the **frequency**, which you hear as pitch. It is counted in **hertz** (Hz), cycles a second. The A that orchestras tune to is 440 Hz; human hearing reaches from about 20 Hz to about 20,000 Hz.

A speaker moves its cone in proportion to the **voltage** on its wires: more voltage, further out. So making a sound means making a voltage that changes in the shape of the waveform. The whole job of a computer's sound hardware is to produce that voltage, and it does it in a chain of steps (Figure 19.1).

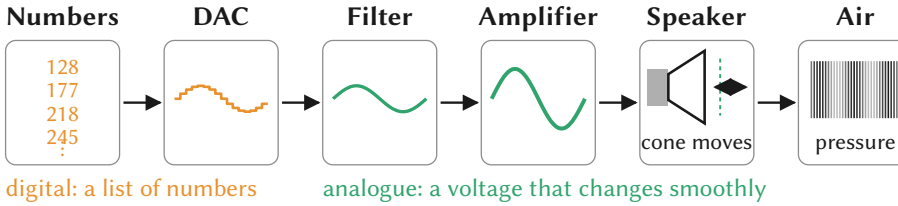


Figure 19.1: From numbers to sound.

A sample is one number for one moment

A computer cannot store a smooth curve. It can store only numbers, so it records the waveform's height at regular moments, one number each time. Each number is a **sample**, and the list of them is the sound in digital form.

Two settings decide how good the list is:

- The **sample rate** is how many samples are taken each second. More samples follow the curve more closely.
- The **resolution** is how many different values one sample can have. It is usually given in bits: an 8-bit sample has 256 possible values, a 16-bit sample 65,536.

Here is the same 1,000 Hz wave sampled at two rates (Figure 19.2).

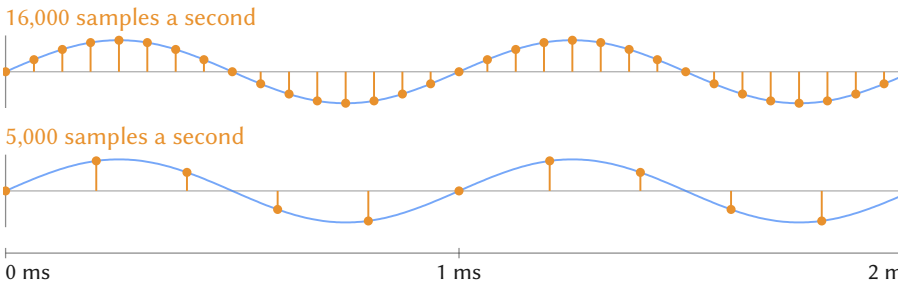


Figure 19.2: The same 1,000 Hz wave sampled at two rates.

Each orange dot is one sample. At 16,000 samples a second, 16 fall in each cycle of the wave and the dots trace it closely. At 5,000, only 5 fall in each cycle, and the outline is much coarser.

From a number to a voltage: the DAC

To play the list back, the computer needs a voltage for each number. The chip that makes it is a **digital-to-analogue converter**, a **DAC**. You give it a number, and its output becomes the voltage that belongs to that number.

Between two samples, a DAC does nothing at all. It **holds** its voltage until the next number arrives, then jumps to the new one. So its output is not a smooth curve but a **staircase**: flat steps, each one sample long, with a jump at every sample.

The steps are easiest to see when the resolution is low. In Figure 19.3, each sample has only 16 possible values (4 bits), and the wave is played back at 16,000 samples a second, the faster of the two rates above. The orange staircase is what the DAC delivers.

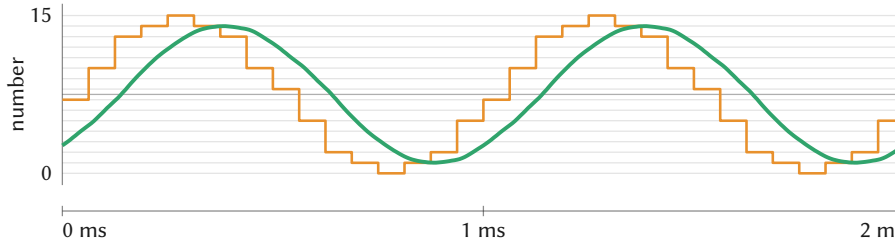


Figure 19.3: The DAC's staircase (orange), and the same output after a low-pass filter (green).

The staircase alone already sounds like the wave, but not quite: the sharp corners of the steps add a faint hiss or buzz of their own, much higher than the note. The green curve shows the cure.

From a staircase to a smooth wave: the filter

The note goes up and down once every millisecond, but the DAC jumps 16 times in that millisecond. The steps change far faster than anything in the wanted sound, and that difference in speed is what makes them easy to remove.

A **low-pass filter** lets slow changes through and holds back fast ones. In its simplest form it is just a resistor and a capacitor. The capacitor charges through the resistor, which takes time, so the voltage across it cannot follow a sudden jump. A slow change, like the 1,000 Hz note, comes through almost whole; the sudden jumps of the steps are rounded off.

That is the green curve in Figure 19.3: the staircase after a filter of two resistor-and-capacitor stages. The steps are gone, and what remains is a smooth wave. It comes

out a little smaller and about a tenth of a millisecond late, neither of which matters to the ear.

This is the step that makes the result analogue: after it, the voltage changes continuously, like the air pressure it will become.

Fast enough, or it sounds wrong

How many samples a second are enough? The rule is simple: **the sample rate must be more than twice the highest frequency in the sound**. A wave that goes up and down needs at least one sample on its way up and one on its way down in every cycle, or the samples cannot tell what it did.

Break the rule, and you do not merely get a rough copy of the high note. You get a **different**, **lower note** that was never in the original. This is called **aliasing**: the samples of the fast wave happen to lie exactly on a slow one, and the DAC and the filter faithfully play the slow one.

Take a telephone line, which carries 8,000 samples a second. Twice the highest frequency must stay below 8,000, so the line can carry sounds up to 4,000 Hz. Here is what happens to a 7,000 Hz tone sent through it (Figure 19.4).

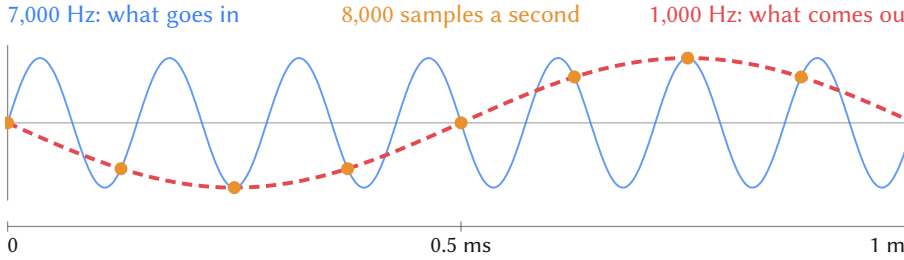


Figure 19.4: Aliasing: a 7,000 Hz tone sampled 8,000 times a second.

The fast blue wave goes through seven cycles in the millisecond, but the samples catch it only eight times. Each sample lands seven eighths of a cycle after the one before, the same height as an eighth of a cycle *before* it, so the samples walk slowly backwards through the wave and describe one cycle of 1,000 Hz (8,000 – 7,000). That is what comes out of the speaker.

No filter after the DAC can undo this: the samples are a correct record of a 1,000 Hz wave. So a sound must be free of frequencies above half the sample rate *before* it is sampled. A music CD's 44,100 samples a second were chosen with this in mind: half of it, 22,050 Hz, is just above the highest pitch anyone hears.

Sound from a single bit

The simplest sound hardware has no DAC at all. It has just one bit, which the computer switches on or off, and which drives the speaker fully out or fully in. The beeper of the ZX Spectrum works like this. On or off, nothing in between. How on earth can that make music?

A note from switching. Switch the bit on and off at a steady pace, and the cone goes out and in at that pace: a **square wave**. The pace is the pitch. To play the 440 Hz A, the program switches the bit 880 times a second, on and off once in each cycle of 2.27 milliseconds. It sounds harsher than a smooth wave, but it is clearly a note. The program has to time every switch itself, so while it plays, it usually does little else.

Any level from switching fast. Switch the bit much faster than any note, faster than the speaker and the ear can follow, and vary *how long it stays on* in each round. Then the filter from earlier, and the speaker's own sluggishness, average the pulses out: a bit that is on a quarter of the time gives a quarter of the full level, half of the time gives half. Figure 19.5 shows pulse-width modulation at work.

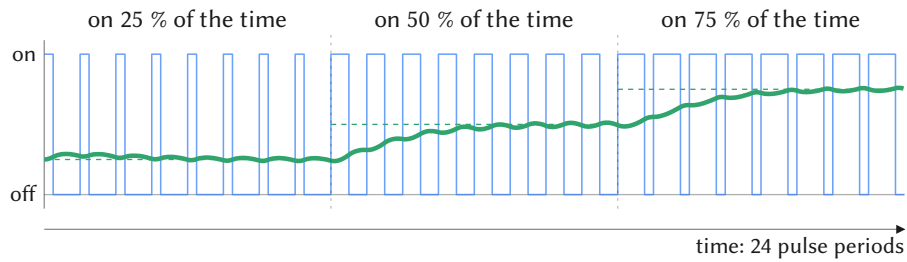


Figure 19.5: Pulse-width modulation: a one-bit signal (blue), filtered (green).

The green curve is the blue pulse train after a filter. Each time the on-time changes, it moves to the new average and settles at about a quarter, a half and three quarters of the full level, the dashed lines, with a small wobble far above hearing.

This is called **pulse-width modulation**: the width of each pulse carries the level. Change the widths in the shape of a waveform, and the averaged output follows that waveform, so one bit can play any sound a DAC could. I find that one of the nicest tricks in all of electronics.

A **delta-sigma converter** builds on the same idea: it switches its single output bit millions of times a second, each time checking whether the output so far has been above or below the wanted level, and a filter turns the stream of bits into a smooth voltage.

Several sounds at once are added together

Music has many voices at once, and a game plays its effects over its music. But the air in front of a speaker can be in only one state at a time, so several sounds can only reach it as one waveform. That waveform is simply their **sum**: at every moment, the levels of all the sounds are added together. In a computer, mixing is adding the numbers (or, in the electronics, adding the voltages). Figure 19.6 shows mixing as adding.

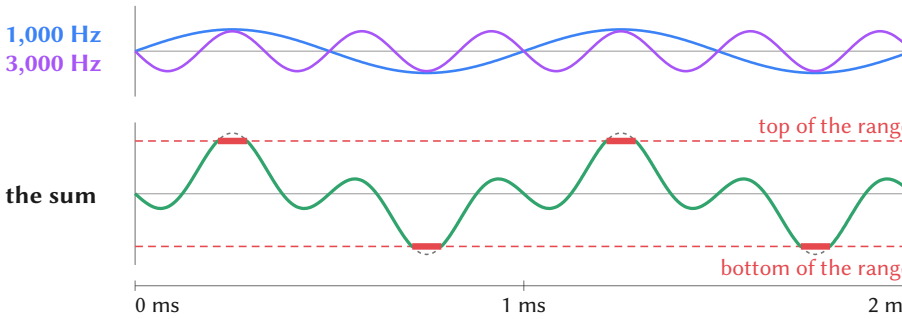


Figure 19.6: Mixing is adding.

Adding has one catch. A DAC, an amplifier or a speaker can deliver only a certain range of levels. When the sum goes past the top or the bottom of the range, the output stays at the limit and the tip of the wave is cut off flat: the green curve, where the dashed one is the true sum. This is **clipping**, marked in red in Figure 19.6, and it sounds like a harsh crackle. A designer avoids it by leaving **headroom**: making each source quiet enough that the loudest possible sum still fits.

There is a second, quieter consequence. Many sources never go below zero: 0 is "off", and the wave moves above it. Their sum then contains a constant extra level, an **offset**, which would only push the speaker's cone out and keep it there. A **capacitor** in the path removes it: it lets changes through but not a level that stays the same.

Two speakers: stereo and space

Your two ears hear every sound slightly differently. A sound from the left arrives at the left ear a moment earlier and a little louder than at the right one, and your brain uses those differences to tell where the sound comes from.

Stereo uses this. It keeps two **channels**, one for each speaker, each with its own DAC, filter and amplifier, and mixing happens twice, once for each side. A **mono** sound has one channel, or the same sound on both sides.

The simplest way to place a sound is **panning**: mix it into the two channels at different levels.

- Equal levels on both sides put the sound in the **centre**, between the speakers.
- More on one side moves it towards that speaker.
- A sound on one side only seems to come from that speaker alone.

Panning gives each voice its own place, the melody in the centre, an accompaniment to the left, a drum to the right. Other effects use the timing difference: **echo** and **reverberation** add quieter, delayed copies of the sound, the way the walls of a room would send it back, and tell the ear how large the space is.

Where the numbers come from

One question is left: who makes the numbers? They come from two kinds of place:

- **The CPU writes them.** The program works out each sample and hands it to a DAC (or switches a single bit), on time, every time. That can play anything, speech or recorded drums, but it costs the CPU work for every sample.
- **A sound chip makes them.** The program tells the chip "play this pitch at this loudness", and the chip's own counters produce the waveform from then on. That frees the CPU, but the sound is limited to what the chip can generate.

Most computers of the 1980s had one or both, and their sounds go through the same chain: a mixer, a filter, an amplifier and a speaker. The Next's sound chapters start from here, with Chapter 20.

In short. A sound is a list of samples, and a DAC holds each one as a step until a low-pass filter smooths the staircase into a wave. The sample rate must be more than twice the highest frequency, or aliasing plays a lower note that no filter after the DAC can remove. One bit can play any sound through pulse-width modulation, because the average on-time becomes the level. Mixing is adding, so each source must leave headroom against clipping, and stereo simply mixes twice, once for each side.

CHAPTER 20

The Big Picture: Four Sources, One Mixer

A 48K Spectrum had one bit of sound and a speaker the size of a coin. Play a modern game on a ZX Spectrum Next, and you hear chip music in nine voices, in stereo, with sampled drums and speech on top. And the same machine still loads a tape by listening to it. So where does each of those sounds come from, and how do they all end up in one pair of headphones?

This is the map of the Next's sound. The chapters that follow take each source apart; here you see how they fit together. Chapter 19 explains samples, the DAC, the filter, sound from a single bit and mixing, and I will not explain them again. The numbers in the sound chapters are for the Next's **28 MHz master clock**, with the CPU at **3.5 MHz** unless a chapter says otherwise, and levels are given in the **units of the Next's mixer**, whose sum runs from 0 to 8,191 per side.

- What the four kinds of sound source are, and how each one sounds
- Which sources make a sound by themselves, and which need the CPU at every step
- How the mixer adds them, and where the sum leaves the machine
- Why the CPU's speed changes the beeper's pitch but not the AY's

From one bit to four kinds of source

The ZX Spectrum 48K makes sound with one bit. Bit 4 of port `$FE`, called **EAR**, drives the speaker: the program sets it, clears it, sets it again, and the speaker's cone follows. Bit 3, **MIC**, drives the tape output and reaches the same speaker more weakly. The speaker is driven *in the program's own time*: while it plays, your program can do little else. The Spectrum 128K added a sound chip, the AY-3-8910, with three voices that play by themselves once they are set up.

The Next keeps both and adds more beside them, which makes **four kinds of source**: the beeper, with its two bits EAR and MIC (MIC also carries the tape input), the AY chips, the DACs and the Pi. The Spectrum's sound stays exactly as it was: a 48K program that toggles port `$FE` bit 4 hears the same square wave, and a 128K tune

plays on the first AY chip as it did on the 128K. Everything new is switched off or out of the way until a program asks for it.

The cast

Figure 20.1 shows the whole system. The sources are on the left, each in the colour it keeps throughout the sound chapters, with the ports and registers your program uses to drive it.

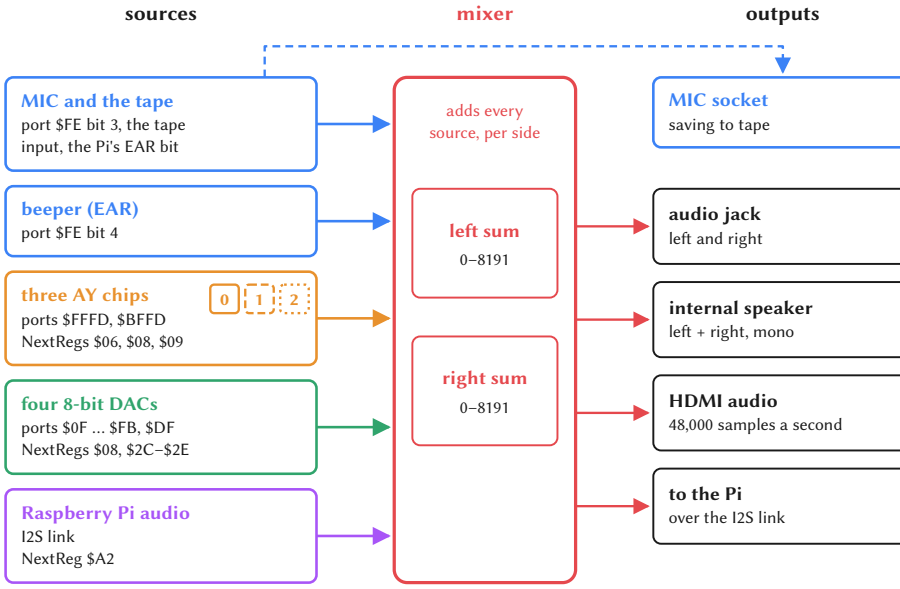


Figure 20.1: The Next's audio system.

Each capture in Figure 20.2 is a program playing a tone of about 1,000 Hz through one source alone, on one level scale. A DAC or the Pi at full scale is about twice as loud as the beeper, and one AY channel is half as loud as the beeper: the old one-bit beeper is no whisper.

The four tones are close to 1,000 Hz, but not equal. Each source gets its pitch from its own clock, divided by whole numbers the program chooses:

- **The beeper**: 3,500,000 ÷ (2 × 1,760) = 994 Hz. The CPU's clock, divided by the length of the program's loop in T-states, twice per cycle.
- **The AY channel**: 218,750 ÷ (2 × 109) = 1,003 Hz. The chip's counters, divided by the channel's period; the next period, 110, gives 994 Hz.

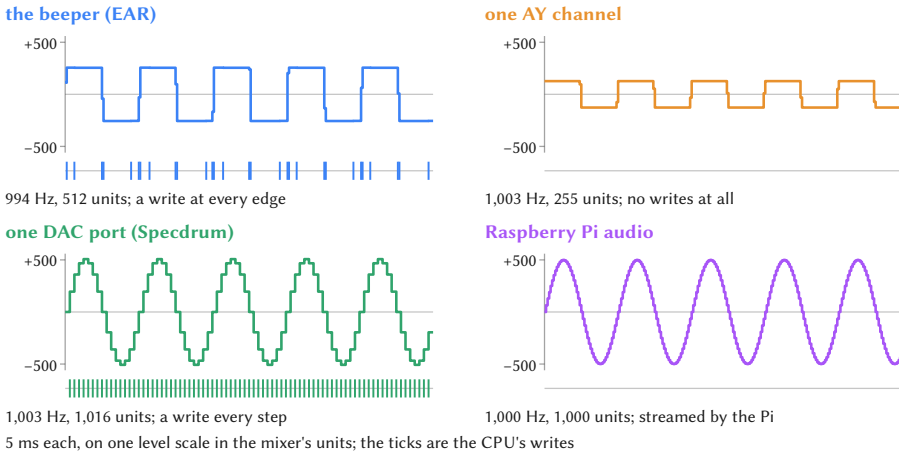


Figure 20.2: One capture per source, with the CPU's writes.

- **The DAC**: 3,500,000 ÷ (218 × 16) = 1,003 Hz. The CPU's clock again, divided by the 218 T-states between two writes and the 16 writes of one cycle.
- **The Pi**: exactly 1,000 Hz. The Pi sends samples it has already worked out, and the Next only plays them back.

The beeper, EAR and MIC

The beeper is the 48K's one bit, unchanged: **port $FE bit 4**. While the bit is 1, it adds 512 units to both sides of the mixer; while it is 0, nothing. Your program makes a note by switching it at the note's pitch.

Port $FE bit 3, MIC, adds 128 units while it is 1: a quarter of EAR, the ratio the 48K's resistors gave the two bits. Before it reaches the mixer, the MIC line is combined with the **tape input**, the signal arriving at the EAR socket from a tape recorder, and with an EAR bit the Pi's audio can produce. So you hear a tape load at MIC's level, as on the original machine. The same line goes out of the MIC socket, so a program saves to tape by switching bit 3. The beeper costs the CPU all its time while it plays, and its pitch depends on how fast the CPU runs (Chapter 21).

Three AY chips

The Next has three copies of the 128K's sound chip. Each has **three channels**, A, B and C, and each channel plays a square wave at a pitch set by a 12-bit period, mixed with noise if you want, at one of 16 fixed volumes or following an **envelope**, a shape that rises or falls by itself. A chip can behave as the original General Instrument **AY-3-8910** or as Yamaha's compatible **YM2149**, which has a finer envelope.

The 128K's own ports reach them: **port `$FFFD`** selects a register, **port `$BFFD`** writes it. A special value written to port `$FFFD`, one with bit 7 set, which no register number has, selects *which* chip the following writes go to, and that is all your program needs to play nine channels. This is **TurboSound Next**. One channel at full volume adds 255 units. In the stereo modes each side hears two of a chip's three channels, so the three chips give a side at most 1,530; with all three in mono, 2,295 (Chapters 22 and 23).

Four 8-bit DACs

A **DAC** channel holds an 8-bit number the CPU writes, and adds it to the mixer until the next write. The Next has four, **A** and **B** on the left side and **C** and **D** on the right. Each side adds its two channels and multiplies them by 4: up to 2,040 units.

The Spectrum and Pentagon world had half a dozen sound cards built on such DACs (Soundrive, Covox, Specdrum and others), each on its own ports. The Next answers on all of their ports, so their software plays unchanged, and adds NextRegs `$2C`–`$2E`, through which the Copper, or a program with no free port, can write them. The capture above writes **port `$DF`**, Specdrum's port, which writes channels A and D at once and so reaches both sides (Chapter 24).

The Raspberry Pi's audio

Inside the Next sits a Raspberry Pi Zero, connected by, among other things, an **I2S** link: a few wires that carry a stream of digital samples, left and right, in both directions. The Pi can play anything a small Linux computer can, an MP3, a speech synthesiser or another computer's sound chip in software, and send the result to the Next, which adds it to the mixer at 10 bits per side: up to 1,023 units.

The link also sends the Next's own sound to the Pi, and the Pi's stream can become the EAR bit of the tape input, so that the Next loads a program the Pi plays. The **PI I2S Audio Control register**, **NextReg `$A2`**, sets it all up (Chapter 25).



KLIVE IDE Klive plays the Pi's part

Klive does not emulate the Pi, but it supports its audio: the `pi-audio` command sends a WAV file or a test tone into the I2S link, and the Next's mixer adds it as the hardware would (Chapter 25).

Generators and level writers

The captures look alike, but they are made in two very different ways, and that decides what a sound costs your program:

- A **generator** makes a waveform by itself from a few settings. The AY chips are generators: the program writes a period, a volume and which channels are on, and the chip plays that tone until it is told otherwise, without any help from the CPU.
- A **level writer** holds whatever level it was given last. The beeper and the DACs are level writers: they make a waveform only if the CPU, the DMA or the Copper writes a new level at every step of it, on time.

The ticks under the captures show the difference. The DAC's sine costs 16 writes per cycle, 16,055 a second, each one on time; if a write comes late, the wave is distorted. Your program can afford that for a drum hit or a word of speech, or it can hand the job to the DMA. The AY's tone costs nothing while it plays, which is why games of the 128K era played their music on the AY and touched the chip only when the tune moved to the next note.

In return, a level writer can play *anything*, any recorded sound or computed waveform, while a generator plays only what its counters can make: square waves, noise and envelope shapes. My advice is AY music, with DAC drums over it.

One mixer, two sides

Every source ends in the **mixer**, which adds them. It has two sides, and every source gives each side its own number. For the beeper the two numbers are always the same; the AY chips, the DACs and the Pi can make them differ, which is what places a sound left or right. On each side, the mixer adds five numbers, many million times a second:

- **EAR**: 512 while port `$FE` bit 4 is 1, otherwise 0.
- **MIC**: 128 while the MIC line is 1, otherwise 0.
- **The AY chips**: 0 to 2,295, the three chips' channels on that side added together.
- **The DACs**: 0 to 2,040, that side's two channels added and multiplied by 4.
- **The Pi**: 0 to 1,023, the side's 10-bit sample.

The sum is a 13-bit number, 0 to 8,191 (Figure 20.3).

The sum always fits. Every input at its largest adds up to 5,998, well below 8,191. The audio jack is tighter: it takes only the sum's lowest 12 bits, up to 4,095. That is plenty for loud music with the beeper on top, but not for every source at full scale at once (Chapter 26).

Silence is not zero. Nothing is ever subtracted. A DAC at rest holds `$80`, the middle of its range, so that a sample can swing both ways, and the two on each side add 1,024 units even when no program has touched them. The Pi's rest level is the middle of its range too, 512. So a silent Next sits at 1,536, and a capacitor after the converters removes that constant level before the sound reaches a speaker.

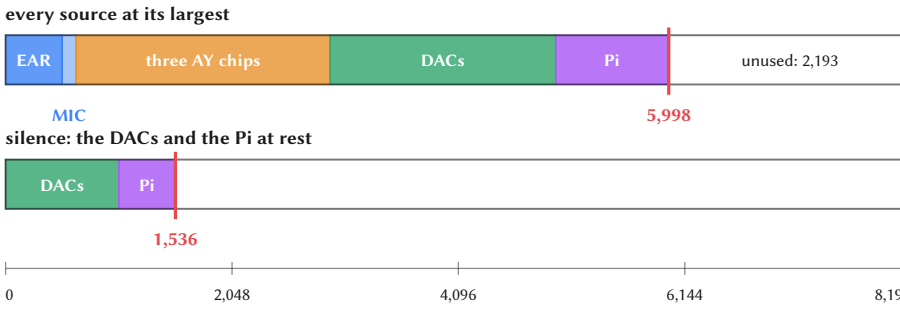


Figure 20.3: The mixer's range per side, to scale.

Where the sound goes

The two sums leave the machine four ways at once:

- **The audio jack** gets each side's lowest 12 bits through a **one-bit converter**: a single pin of the FPGA switched on and off at 28 MHz, more often "on" the larger the number, and a resistor and a capacitor on the board that smooth the pulses into a voltage. It is the delta-sigma converter of Chapter 19.
- **The internal speaker** gets one mono signal, the left and right sums added together, through one more converter of the same kind. Bit 4 of the **Peripheral 3 Setting register**, **NextReg `$08`**, switches it on, and it is on after power-on. With bit 6 of the **Peripheral 2 Setting register**, **NextReg `$06`**, also set, the speaker plays only the beeper, and EAR and MIC leave the mix: your program can click keys on the speaker without disturbing the music in the headphones.
- **HDMI audio** sends the sums as numbers, 48,000 samples a second, each a 16-bit word, beside the picture. Bit 2 of the **Peripheral 4 Setting register**, **NextReg `$09`**, silences it.
- **The Raspberry Pi** receives the sums over the I2S link, when the link is set up to send.

One line goes out separately: **the MIC line** drives the MIC socket, the tape recorder's input. And the jack and the speaker have no sample rate: their converters follow the sum as it changes, 28 million times a second. Only HDMI samples it.



KLIVE IDE Which output you hear

Machine › Sound › Output chooses which of the Next's outputs Klive plays: the audio jack (the default), the internal speaker or HDMI, each built as the real machine builds it. Play the same program through each, and you hear what each output does with the same sum.

The clocks behind the sound

Every part of the Next runs from one **master clock of 28 MHz**, and two branches of it time the sound.

The AY chips' clock is fixed. The master clock divided by 16 gives **1.75 MHz**, close to the 1.77 MHz of the 128K's AY, and each chip divides that by 8 again: its counters step 218,750 times a second. A tone's pitch is that rate divided by twice the channel's period, so the A of 440 Hz needs a period of 249 (439.3 Hz). Nothing a program does changes these clocks, so AY music plays at the same pitch at 3.5 MHz and at 28 MHz.

The beeper and the DACs follow the CPU. They change only when an instruction writes them, so their timing is the program's. A loop that flips port `$FE` bit 4 every 1,760 T-states makes 994 Hz at 3.5 MHz and 1,989 Hz at 7 MHz. The Next's CPU can run at 3.5, 7, 14 or 28 MHz (the **CPU Speed register, NextReg `$07`**), so a beeper tune written for the 48K plays an octave higher at 7 MHz. Contention stretches a loop that runs in screen memory, and its note comes out lower (Chapter 12). A DMA transfer timed by its own prescaler, or the Copper, does not depend on the CPU's speed.

The switches

Five registers decide which parts of the sound system are on and how they behave. An old 48K or 128K program never writes them; a Next program, or NextZXOS, sets them up:

- **The Peripheral 2 Setting register, NextReg `$06`**: bit 6 sends the beeper only to the internal speaker. Bits 1–0 are the AY chips' mode: `00` YM2149, `01` AY-3-8910, and `11` holds all three chips silent in reset. The register list also names a mode `10`, but the chips use only bit 0, so `10` behaves as YM2149.
- **The Peripheral 3 Setting register, NextReg `$08`**: bit 5 chooses how each chip's three channels are spread over the two sides (ABC or ACB), bit 4 switches the internal speaker on, bit 3 the DACs, bit 1 TurboSound (with it off, only the selected chip plays, as on a 128K), and bit 0 the issue 2 keyboard, which changes what port `$FE` reads back from the MIC bit.
- **The Peripheral 4 Setting register, NextReg `$09`**: bits 7–5 put AY chips 2, 1 and 0 in mono, bit 2 silences HDMI audio.
- **NextReg `$84`**, one byte of the four **Internal Port Decoding Enables registers, NextRegs `$82`–`$85`**: bit 0 enables the AY's ports, bits 1–7 the seven families of DAC ports, from Soundrive to Specdrum. Some of those ports collide with other devices, and these bits decide who answers.
- **The PI I2S Audio Control register, NextReg `$A2`**: bits 7–6 switch the Pi's audio on (left, right or both), bit 4 sets which way the data travels on its two pins, bits 3–2 mute each side, and bit 0 sends the Pi's stream to the tape input.

After power-on, the AY chips are in YM mode with TurboSound off, the internal speaker is on, the DACs are off and held at `$80`, every port family is enabled, HDMI

audio plays and the Pi's audio is off: a machine that sounds exactly like a 128K until something asks for more. Each source has a chapter of its own next, in any order you like, then the mixer, then what happens after the sum (Chapter 27).

In short. The Next adds the beeper, three AY chips, four 8-bit DACs and the Pi in one mixer whose sum runs from 0 to 8,191 per side, of which the audio jack takes only the lowest 12 bits. AY chips are generators that play alone from a fixed clock; the beeper and DACs are level writers, and their pitch belongs to whatever times their writes. A silent Next sits at 1,536, not zero. After power-on it sounds like a 128K: the DACs and TurboSound are off until a program asks.

CHAPTER 21

The Beeper: EAR, MIC and the Tape

Manic Miner plays a two-voice tune on a machine whose sound is a single bit. The same bit's neighbour tells the tape recorder what to save, and the same port reads back what the tape plays. So how does one bit make music, and how do music and data share one wire without tripping over each other?

The beeper is the oldest sound on the Next, and still the one every Spectrum program knows: a 48K game's music and effects, the click of a key in BASIC, the screech of a program loading, and every program ever saved to cassette. Your Next runs those programs unchanged, so it reproduces the beeper exactly: the levels, the timing, and the small details of what a program reads back.

This chapter builds on Chapter 20, and on Chapter 19 for square waves and sound from one bit. The CPU runs at **3.5 MHz** with contention **off** unless a section says otherwise, and the captures show the sound after the audio jack's capacitor, so they move around zero.

- How the two bits of port $FE make sound, and how loud each one is
- How a timing loop plays a note, and how CPU speed and contention change its pitch
- How short pulses let a single bit play two voices
- How saving and loading work through the MIC and EAR sockets

Two bits of sound

The beeper lives in **port $FE**, the ULA's port. Any write to an even port address (address bit 0 is 0) writes it, and three of its fields matter:

- **Bit 4, EAR**, is the beeper: the bit your program switches on and off to make a sound.
- **Bit 3, MIC**, is the tape output: the bit your program switches to save data to tape. You hear it too, more quietly.
- **Bits 2–0** are the border colour.

Bits 7–5 of a write are ignored. A reset clears the port: EAR and MIC start low, so a machine that has just been switched on is silent.

Why two sound bits? Because the 48K's ULA had a single pin for the tape sockets and the speaker: bits 4 and 3 drove it through resistors of different values, and the signal from the EAR socket arrived on the same pin. That is why MIC is quieter than EAR, and why what a program writes to bits 4 and 3 shows up when it reads the port back. The Next has no such pin, but it keeps the two bits, their loudness and the read-back rules, so that the programs that relied on them still work.

Reading port `$FE` returns the keyboard in bits 4–0 (a 0 for each key held in the rows the address selects), **bit 6, the tape input**, and 1 in bits 7 and 5. With no key held and nothing on bit 6, a read gives `$BF`.

EAR and MIC in the mixer

The mixer adds each bit as a fixed level, the same on the left and the right side:

- **EAR** adds **512** while port `$FE` bit 4 is 1, and 0 while it is 0.
- **MIC** adds **128** while the MIC line is 1, and 0 while it is 0. The MIC line is bit 3 combined with the tape input; with no tape playing it is just bit 3.

The two bits together give four levels, 0, 128, 512 and 640, a small part of the mixer's range. EAR is four times as loud as MIC, the ratio the 48K's two resistors gave them. But the beeper is no quiet source: its 512 is twice what one AY channel at full volume adds. Nor does it rest in the middle, as the DACs do. Its silence is 0, so the first time a program sets bit 4, the sum jumps by 512, and the speaker clicks, just as it did on the 48K (Chapter 26).



KLIVE IDE Watching EAR and MIC

Pause the machine, and the **ULA & I/O** panel shows `EAR` and `MIC`, the two sound bits of port `$FE`, as the program last wrote them: which of the four levels the beeper holds at that moment.

A note from a loop

The beeper holds whatever level it was given last. To play a note, your program switches bit 4 on and off at the note's pitch, at exactly the right moments. The usual way is a loop that counts the time between two writes in T-states:

```
        di              ; interrupts would stretch it
        ld c,$fe
        ld a,$10        ; bit 4 set: EAR on
Loop:   out (c),a       ; 12 T-states: write EAR
        xor $10         ;  7  next write's EAR: opposite
        ld b,133        ;  7
```

```
Dly:    djnz Dly        ; 13 × 132 + 8: wait
        jp Loop         ; 10
```

1. `out (c),a` writes the EAR bit; the speaker's cone moves.
2. `xor $10` flips bit 4 of A, so the next write is the opposite level.
3. `djnz` counts B down from 133. Each round that jumps back costs 13 T-states and the last one 8, so the wait is 13 × 132 + 8 = 1,724 T-states.
4. One pass of the loop is 12 + 7 + 7 + 1,724 + 10 = **1,760 T-states**: half a period of the note, since it takes two passes to go up and come back down.

The loop closes with `jp` rather than the shorter `jr`, because `jp` takes 10 T-states to `jr`'s 12, and in a timing loop every T-state is part of the note. At 3.5 MHz a pass takes 503 µs, so the note is 3,500,000 ÷ (2 × 1,760) = **994 Hz**.

The loop's length is the only thing that sets the pitch, and it changes in steps of 13 T-states, one round of `djnz`. B = 132 gives 1,747 T-states and 1,002 Hz, B = 134 gives 1,773 and 987 Hz. So a beeper tune is slightly out of tune wherever its notes fall between those steps.



KLIVE IDE Timing a half period

`bp-set $0000 -o -m $0001` sets an **I/O write** breakpoint on every even port, as the ULA decodes it, so it stops at every beeper write. Continue from one stop to the next, and the Z80 CPU panel's `TSP` shows the time between two writes: 1,760 T-states, half a period of the note.

The CPU's speed and contention

Because the pitch is a count of T-states, it is only as steady as the CPU's clock. Two things change how long the same loop takes:

- **The CPU's speed.** The Next's CPU runs at 3.5, 7, 14 or 28 MHz, set by the **CPU Speed register**, **NextReg $07**. A T-state at 7 MHz lasts half as long, so the same loop plays twice as high: 1,989 Hz, an octave up.
- **Contention.** At 3.5 MHz, the Next stops the CPU for a few T-states when it reads memory from `$4000` to `$7FFF` while the screen is being drawn, as the 48K's ULA did (Chapter 12). Bit 6 of the **Peripheral 3 Setting register**, **NextReg $08**, switches contention off, and at 7 MHz and above there is none.

Figure 21.1 shows the loop three ways.

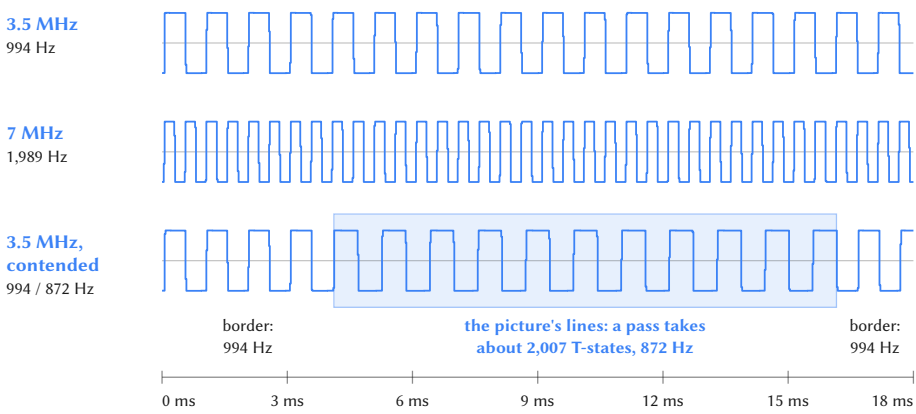


Figure 21.1: The same loop at 3.5 MHz, at 7 MHz, and in contended memory.

Contention does not lower the note; it **breaks it in two**. It happens only while the beam is on one of the picture's 192 lines, not in the border, so the loop runs at two speeds in every frame:

- In the border, about 8 ms of each 20 ms frame, a pass takes its 1,760 T-states, and the note is 994 Hz.
- On the picture's lines, about 12 ms of each frame, the CPU is held up again and again, and in this capture a pass takes about 2,007 T-states: 872 Hz.

So the ear hears a rough, warbling tone, not a lower note; another loop, or the same loop at another address, is stretched by a different amount. This is why beeper music on the Spectrum usually runs from `$8000` and up, with interrupts off, which would otherwise take the CPU away from the loop every 20 ms.



KLIVE IDE Checking speed and contention

Before you blame a warbling note on the loop, check the machine: the **Next Registers** view decodes `$07` and `$08` into their bit fields, showing the CPU's speed and whether bit 6 has switched contention off.

Two voices from one bit

When one voice wants the bit high and the other wants it low, there is no level in between to give. So beeper music makes each voice a train of **short pulses**, one at the start of each of its periods, with the bit low the rest of the time. The bit can carry both trains, and a train of pulses at a voice's pitch sounds at that pitch just as a square wave does.

This program gives each voice one fixed **slot** of every pass of its loop. Each voice keeps an 8-bit counter, its **phase**, and adds its **step** to it on every pass; when the addition carries out of 8 bits, the voice has finished a period and takes its slot with EAR on:

```
        ld hl,0         ; H, L: the two voices' phases
        ld de,$0405     ; D, E: their steps, 4 and 5
Loop:   ld a,h          ;  4
        add a,d         ;  4  voice 1's phase + its step
        ld h,a          ;  4
        sbc a,a         ;  4  $FF if that carried, else 0
        and $10         ;  7  ... so A is EAR on or off
        out ($fe),a     ; 11  voice 1's slot begins
        ld a,l          ;  4
        add a,e         ;  4  the same for voice 2
        ld l,a          ;  4
        sbc a,a         ;  4
        and $10         ;  7
        out ($fe),a     ; 11  voice 2's slot begins
        ld a,0          ;  7
        nop             ;  4  four nops: voice 2's slot
        nop             ;  4  is as long as voice 1's
        nop             ;  4
        nop             ;  4
        out ($fe),a     ; 11  both slots over: EAR off
        jp Loop         ; 10
```

1. Every instruction always takes the same time, whatever the phases hold: `sbc a,a` turns the carry into a mask instead of a jump. One pass is always **112 T-states**, so the loop runs 3,500,000 ÷ 112 = 31,250 times a second.
2. Voice 1's slot runs from its `out` to voice 2's `out`, 34 T-states (23 for the five instructions in between and 11 for the `out` itself). Voice 2's slot runs from its `out` to the last one, also 34. The remaining 44 T-states of the pass, from the last `out` round to the first, EAR is off.
3. Voice 1's phase gains 4 each pass, so it carries once every 256 ÷ 4 = 64 passes: its pitch is 31,250 × 4 ÷ 256 = **488.3 Hz**. Voice 2's gains 5, and carries every 51 or 52 passes: on average 31,250 × 5 ÷ 256 = **610.4 Hz**. The two notes are a major third apart, a ratio of exactly 5 to 4.
4. The slots of a pass follow each other and never overlap, so the bit is exactly the two pulse trains added together (Figure 21.2).

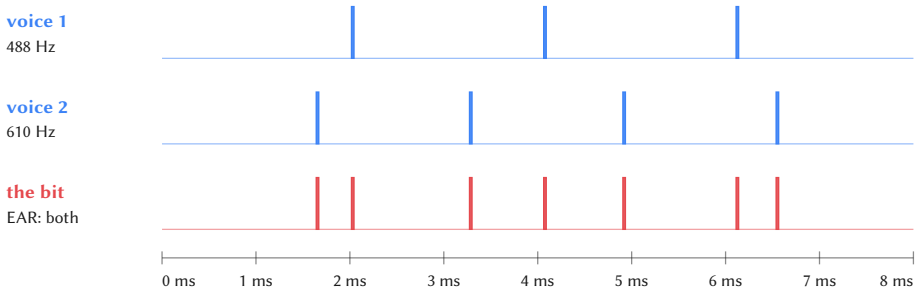


Figure 21.2: Two voices and the bit they share, over 8 ms.

The pulses are far too short to hear one by one; what your ear hears is how often they come. Each note comes with **overtones**, at whole multiples of its pitch, almost as strong as the note itself, which give the beeper's chords their buzzing, nasal colour.

What two voices cost

- **Loudness.** A voice is on for 34 of every 7,168 T-states (voice 1's period of 64 passes), less than half a percent of the time. Short pulses are quiet: the louder a voice should be, the longer its slot, so a program sets the **volume of a voice by the width of its pulses**. Two voices need two slots that never meet, so each can have at most half the pass.
- **Pitch.** A voice's pitch is the loop's rate times its step divided by 256. Finer pitches need a 16-bit phase, which costs more T-states per pass and lowers the rate.
- **The CPU.** While the tune plays, the CPU does nothing else: every T-state of the loop is counted. This is why beeper music on the 48K played on title screens and between levels, and why the 128K's AY chip, which plays by itself, was such a step forward.

▶ TRY IT Two Voices

Part III, key 1. The demo plays the two-voice loop at 3.5 MHz, with contention and interrupts off: each voice alone first, then six intervals, about a second each, with the row that plays highlighted. The steps are small whole numbers, so every interval is an exact ratio: 5 to 4 for the third, 3 to 2 for the fifth. The code is in `examples/code/demos/3-sound/two-voices.kz80.asm`. To move on to the next interval it must count passes, and the counter takes the place of one `nop`, so a pass still lasts 112 T-states:

▶ TRY IT Two Voices *(continued)*

```
TvLoop:
    ld a,h                  ;  4
    add a,d                 ;  4  voice 1's phase + its step
    ld h,a                  ;  4
    sbc a,a                 ;  4  $FF if that carried, else 0
    and $10                 ;  7  ... so A is EAR on or off
    out ($FE),a             ; 11  voice 1's slot begins
    ld a,l                  ;  4
    add a,e                 ;  4  the same for voice 2
    ld l,a                  ;  4
    sbc a,a                 ;  4
    and $10                 ;  7
    out ($FE),a             ; 11  voice 2's slot begins
    ld a,0                  ;  7
    dec c                   ;  4  the pass counter takes a NOP's place
    nop                     ;  4
    nop                     ;  4
    nop                     ;  4
    out ($FE),a             ; 11  both slots over: EAR off
    jp nz,TvLoop            ; 10  112 T-states a pass
    djnz TvLoop             ; the next 256 passes
```

`dec c` sets the zero flag that `jp nz` tests, and `ld a,0`, `nop` and `out` leave it alone. Once every 256 passes the loop falls through to `djnz`, 13 T-states in 28,672. To see a pass's length, put a breakpoint on `TvLoop` in Klive IDE and continue once: the Z80 CPU panel's TSP shows 112.

Saving to tape

A program saves to tape by switching **port `$FE` bit 3**, MIC, in a pattern of pulses that encodes its data. The **MIC socket** of the Next carries that pattern to a tape recorder, as the 48K's did.

The MIC line is not bit 3 alone, though. It is bit 3 combined with the tape input and with the EAR bit that the Raspberry Pi's audio link can produce (Chapter 25) by an **exclusive or**: the line is 1 when an odd number of the three are 1. The same line feeds both the MIC socket and the mixer's MIC input (Figure 21.3).

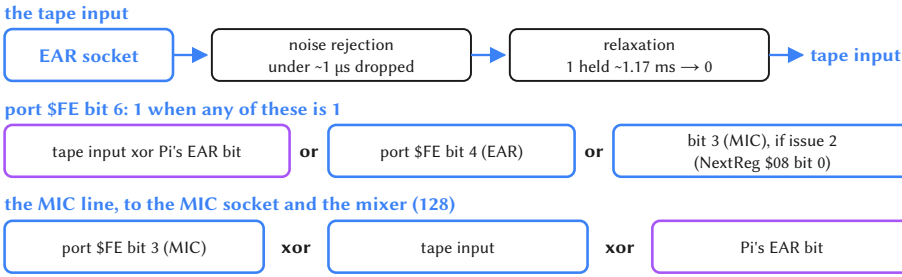


Figure 21.3: The beeper's lines: the tape input, port $FE bit 6 and the MIC line.

An exclusive or keeps each signal whole. With no tape playing, the MIC line is bit 3, exactly what your program writes. While a tape plays and the program holds bit 3 still, the line follows the tape: if bit 3 is 1, upside down, which sounds the same and carries the same edges. So:

- **A loading tape is heard**, at MIC's level of 128: the screech of a loading program comes out of the speaker, as it did on the 48K.
- **The MIC socket echoes the tape input**: a recorder plugged into it while another plays into EAR records what the Next hears.
- **A program saving data** switches bit 3 while no tape plays, and the socket carries its pulses unchanged.



KLIVE IDE Where a SAVE lands

Klive catches a `SAVE` through the 48K ROM's tape routine and writes it as a `.tzx` file: into the project's `SavedFiles` folder when a project is open, otherwise into `Klive/SavedFiles` in your home folder. Insert it with **Machine › Tape › Insert** to load it back.

Loading from tape

A tape recorder plays its signal into the **EAR socket**. What the Next reads from it is not the raw voltage but a clean level, 0 or 1, produced by an **input stage** in two steps, both counted on the master clock.



KLIVE IDE Inside a tape file

Open a `.tap` or `.tzx` file in the Explorer to see what it holds: a timeline of its blocks, each header decoded field by field, and every block ready to open as a memory dump, a disassembly or a BASIC listing.

A tape's signal is analogue and noisy: near an edge it may cross the threshold several times in a row. The first step, **noise rejection**, passes a new level on only when it

has held steady for **two ticks of the 1.75 MHz clock**: any change starts the count again. That takes between 0.6 and 1.14 µs, depending on where the ticks fall, so a pulse shorter than 0.6 µs never reaches the CPU and one longer than 1.14 µs always does. A real edge arrives about a microsecond late, which no loader notices.

The second step, **relaxation**, deals with a tape that stops on a 1. Some recordings, and some tape circuits, leave the level at 1 after the last pulse, and a loader waiting for the next edge would see the bit stuck. The 48K's input relaxed back to 0 after at most about 800 µs, and the ZX81 expects it to within about 1,300 µs. The Next copies them: a 1 that lasts 64 ticks of a 54,688 Hz clock, between 1.15 and 1.17 ms, turns into a 0, until the input goes low and high again. So a tone above about 430 Hz passes untouched, and a Spectrum tape, whose pilot pulses are 2,168 T-states, 619 µs, and whose data pulses are shorter still, is never bothered. The result of both steps is the **tape input**.



KLIVE IDE Loading a tape on the Next

Insert a `.tap` or `.tzx` file with **Machine › Tape › Insert**, and `LOAD ""` in the 48K BASIC loads it. Fast load copies the data straight into memory; with it off, the tape plays in real time through the emulated noise rejection and relaxation, and, as on the real machine, the load works only with the CPU at 3.5 MHz.

What port $FE bit 6 reads

Bit 6 is not the tape input alone. It is 1 when any of these is true at the moment of the read:

1. **The tape input**, combined by exclusive or with the Pi's EAR bit, is 1.
2. **Port $FE bit 4**, EAR, was last written as 1.
3. **Port $FE bit 3**, MIC, was last written as 1, and the **issue 2 keyboard** is on: bit 0 of the **Peripheral 3 Setting register**, **NextReg $08** (0 after power-on).

So a loader must keep EAR at 0 while it reads the tape, or bit 6 reads 1 whatever the tape plays; the 48K ROM's loader does, and keeps MIC at 1. If you write a loader of your own, clear EAR before the first read of the tape. Bit 6 follows the bits at once: the read straight after a write that clears bit 4 already returns 0.

The issue 2 keyboard

The 48K was built in several board revisions, called issues. On the early **issue 2** boards, writing a 1 to the MIC bit was enough to make bit 6 read 1; from **issue 3** on, only the EAR bit did. Software written on an issue 2 machine sometimes compared the whole byte read from port `$FE` against a fixed value, with bit 6 assumed to be 1, instead of testing the key bits alone. On a later machine that comparison fails.

The Next behaves as issue 3 after power-on, as most Spectrums did, and bit 0 of NextReg `$08` gives such a program the issue 2 machine it was written for. The setting has a price: the 48K ROM's loader holds MIC at 1, so bit 6 reads 1 throughout and loading from tape fails. Switch it on after the program has loaded.



KLIVE IDE Issue 2 at a glance

When a tape load fails for no visible reason, select `$08` in the **Next Registers** view: bit 0 reads "Implement issue 2 keyboard". If it is on, bit 6 reads 1 while MIC is held at 1, and the ROM's loader never sees the tape.

The internal speaker

The Next has a small speaker on its board, which plays the left and right sums added together. Bit 4 of NextReg `$08` switches it on (1, the value after power-on) or off. Bit 6 of the **Peripheral 2 Setting register**, **NextReg `$06`**, sends the beeper **only to the internal speaker**: with it set, and the speaker on, EAR and MIC leave the mix that reaches the audio jack and HDMI, and the speaker plays them alone, at their usual loudness and 4-to-1 ratio. It keeps key clicks and alerts in the machine while the music plays in your headphones.

In short. Port `$FE` bit 4 adds 512 to the mix and bit 3 adds 128, and a note is a loop whose length in T-states sets its pitch, so a faster CPU plays it higher. Contended memory breaks the note into two pitches every frame, so run beeper code from `$8000` up with interrupts off. Short pulses in separate slots let one bit carry two voices, using all the CPU. Port `$FE` bit 6 reads 1 while EAR is written as 1, so a loader must clear EAR first.

CHAPTER 22

One AY Chip: Tones, Noise and Envelopes

Sixteen registers. Write three numbers, and the chip plays a chord forever, with no further help from the CPU. The whole soundtrack of the ZX Spectrum 128K was made from three counters, a random-number generator and a staircase. So how can so little make music?

The AY chip is the sound of the 128K era and of most Spectrum music since: game soundtracks, demo music, the trackers musicians still write in today. It is a **generator**: once your program has set it up, it plays by itself. A beeper tune keeps the CPU busy for every edge of the wave; an AY tune costs a few register writes a frame. The Next has three of these chips. Here we take one apart, and Chapter 23 puts the three together.

This chapter builds on Chapter 20. The chip is in **YM2149 mode**, the state after power-on, unless a section says otherwise, and one channel at full volume adds **255** units to the mixer.

- How a program reaches one chip's sixteen registers through two ports
- How counters, noise and the mixer register make each channel's sound
- How the volume table and the sixteen envelope shapes work
- How the AY-3-8910 and the YM2149 differ, and how to make music with them

The parts of one chip

The chip was designed by General Instrument as the **AY-3-8910**, a programmable sound generator, or PSG; Yamaha made a compatible chip, the **YM2149**. The 128K used the first. The Next builds both into its FPGA, and a register chooses which one each of its three chips behaves as. Inside, one chip has a handful of parts, all driven by one clock:

- Three **tone generators**, one per channel, each a counter that makes a square wave at a pitch your program sets.
- One **noise generator**, a shift register that makes random bits, shared by the three channels.

- The **mixer register**, which decides for each channel whether it gets its tone, the noise, both, or neither.
- An **amplitude** per channel: a fixed volume, or the **envelope**, a volume that rises and falls by itself; one envelope generator is shared by the three channels.
- A **volume table**, the chip's DAC, which turns each channel's 4- or 5-bit volume into a level.
- Three **outputs**, channels **A**, **B** and **C**, each 0 to 255 units, which go to the mixer.

Every one of these parts is a small counter or a register, and none of them can make a curve: everything the chip plays is built from on/off signals and steps of volume.

Sixteen registers behind two ports

The chip has **16 registers**, R0 to R15, each 8 bits wide, though many use fewer (Figure 22.1).

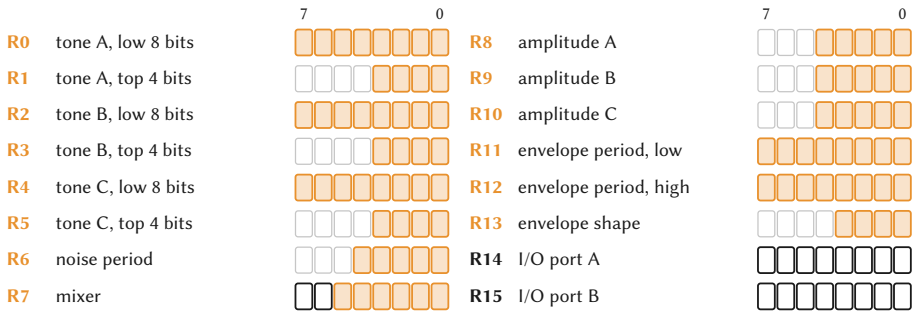


Figure 22.1: The sixteen registers, with the bits each one uses.

Your program reaches them through two ports, the 128K's own:

1. It writes the register's number to **port `$FFFD`**, the register select port. The value must have bits 7–5 clear: it is a register number, 0 to 31. Only 0 to 15 exist; a write to a register numbered 16 to 31 is ignored.
2. It writes the value to **port `$BFFD`**, the data port. The selected register stays selected, so the program can write it again without selecting it again.
3. It reads the selected register back from port `$FFFD`. (On the +3's timing, port `$BFFD` reads the same.)

A value written to port `$FFFD` with bit 7 set is not a register number; the Next uses those values to choose between its three chips (Chapter 23). Values with bits 7–5 anything else but `000` or the chip-select pattern select nothing.

After a reset, every register is 0 except **R7, which is `$FF`**: all tones and noise switched off, and every volume 0. My advice is never to rely on that: set every register your tune uses, R7 included, before the first note.



KLIVE IDE A log of register writes

Klive's PSG panel is not available on the Next, but two logpoints show every write in the Log pane while the tune plays on: the register selected, and the value written to it.

```
bp-set $FFFD -o -log "[AY] R{VAL}"
bp-set $BFFD -o -log "[AY] = {VAL:hex8}"
```

R14 and R15 are not sound registers at all: they are the chip's two 8-bit **I/O ports**, A and B. On the 128K, port A drove the serial connector. On the Next, nothing is connected to them. R7 bits 6 and 7 make each one an output (1) or an input (0); an input reads `$FF`, as if every pin were pulled up, and an output reads back the value last written.

A counter that makes a square wave

The chip runs from the Next's **1.75 MHz** PSG clock, the 28 MHz master clock divided by 16, and divides it by 8 again, so its counters step **218,750 times a second**. Call one of those steps a **tick**: every pitch, noise and envelope in this chapter is counted in ticks.

Each channel's **tone period** is a 12-bit number: the low 8 bits in R0 (channel A), R2 (B) or R4 (C), the top 4 bits in R1, R3 or R5. The tone generator is a counter that counts ticks:

1. At every tick, the counter goes up by one.
2. When it has counted to the period, it starts again from 0, and the channel's square wave **flips**: high becomes low, or low becomes high.
3. So each half of the square wave lasts `period` ticks, and a whole period lasts twice that.

The pitch is therefore

$$\text{frequency} = 218{,}750 \div (2 \times \text{period})$$

The A of 440 Hz needs a period of 218,750 ÷ 880 = 248.6, so 249, which plays 218,750 ÷ 498 = **439.3 Hz**: as close as a whole number allows, and too little out of tune to hear.



KLIVE IDE Periods from the assembler

Let Klive's assembler work out a period from a pitch, rounded to the nearest whole number: `.defw (218750 + 440) / 880` assembles as 249. A table of notes written this way shows each note's pitch in the source.

The 12 bits give periods from 1 to 4,095, and so pitches from 218,750 ÷ 8,190 = **26.7 Hz**, a low growl, up to 109,375 Hz, far above hearing. A period of 0 plays as 1. The steps between pitches are fine at the bottom of the range and coarse at the top: 249 and 250 are 1.8 Hz apart, but 10 and 11 are almost 1,000 Hz apart, so very high notes come out of tune. Write your tune where the periods are large.

The 128K's AY ran from a clock of about 1.77 MHz instead of 1.75 MHz. The same period plays about 1.3 % lower on the Next, a quarter of a semitone: a 128K tune keeps its tune, just a little lower.

Noise from a shift register

The noise generator makes random bits with a **17-bit shift register**:

1. At every **noise step**, all 17 bits move one place towards bit 0, and the bit that falls out of bit 0 is lost.
2. The new bit 16 is **bit 0 exclusive-or bit 2**, taken before the move. (If the register is all zeros, as it is after a reset, a 1 is fed in instead, so that it gets going.)
3. **Bit 0 is the noise signal**: on or off, like a tone.

The bits look random, though they are worked out exactly: the pattern repeats only after **114,681** steps, and each step changes the output with a chance of about one half.

The noise generator's counter runs at half the tone generators' rate, 109,375 steps a second, and the **noise period** in R6 bits 4–0 (1 to 31; 0 plays as 1) divides it once more. A small period gives a high, hissing noise; a large one a slower, rougher one, closer to a rumble. Even at period 31, 3,528 steps a second, the pattern takes 32.5 seconds to repeat, so no ear ever hears it come round. There is only one noise generator per chip, so its period is the same for every channel that uses it.

Tone, noise, both or neither

R7, the mixer register, decides what each channel plays. Bits 0–2 switch the tones of A, B and C, bits 3–5 their noise, and both are **active low**: a 0 switches the source on. Each channel's signal is

> (tone, or tone switched off) and (noise, or noise switched off)

which gives four cases (Figure 22.2):

- **Tone only** (tone bit 0, noise bit 1): the square wave.
- **Noise only** (tone bit 1, noise bit 0): the noise.
- **Tone and noise** (both 0): on only where the tone *and* the noise are on, so a noise that comes and goes at the tone's pitch: a pitched hiss, used for drums and engines.
- **Neither** (both 1): the signal is always on, so the channel plays its **volume as a steady level**.



Figure 22.2: Channel A with R7 set four ways, tone period 109 and noise period 8.

A channel with everything switched off is not silent, and that is the most counter-intuitive rule of the chip. It has two consequences:

- **A channel is silenced by its volume, not by R7.** Switching off a channel's tone and noise makes it a steady level, which you hear as a click when it appears and disappears. The silent way is volume 0. This catches almost everyone out once.
- **The steady level is a DAC.** A program that switches off a channel's tone and noise and then rewrites its volume many times a second plays a waveform through the volume table: that is how 128K programs played sampled drums and speech on the AY, before DAC sound cards.

Sixteen volumes on a curve

Each channel's **amplitude register**, R8 (A), R9 (B) or R10 (C), sets its loudness:

- **Bits 3–0**: a fixed volume, 0 to 15. 0 is silent.
- **Bit 4**: when it is 1, the channel ignores bits 3–0 and follows the envelope.

The volume is not the level. It goes through the chip's **volume table**, which is **logarithmic**: each step up multiplies the level by roughly the same factor, because your ear hears loudness as a ratio. On a straight scale, the top volumes would sound almost alike, and the bottom ones would jump.

The YM2149 has a table of 32 entries, and a fixed volume v uses entry $2 \times v + 1$, every other entry. The AY-3-8910 has a table of 16, one per volume (Figure 22.3).

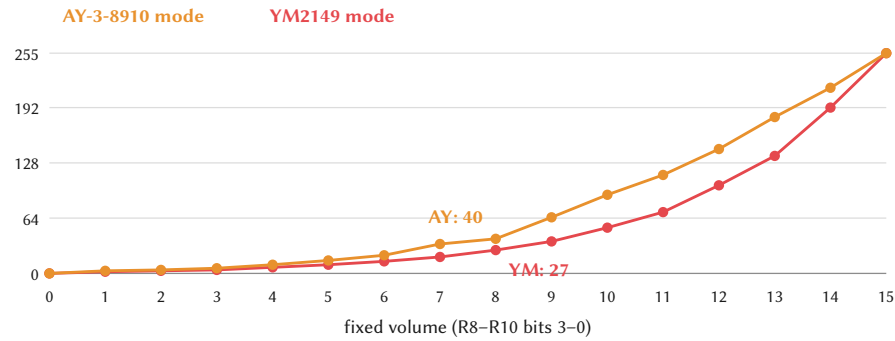


Figure 22.3: The level each fixed volume gives, in YM2149 and AY-3-8910 mode.

At volume 15, both give 255. Below it, the AY table is louder: volume 8 gives 40 in AY mode and 27 in YM mode, volume 1 gives 3 against 2. So a 128K tune that uses middle volumes for its accompaniment sounds more balanced in AY mode, the chip it was written for: if an old tune sounds thin, try AY mode first.

The envelope: a volume that moves

An **envelope** is a volume that moves by itself. A channel whose amplitude register has bit 4 set takes its volume from the chip's one **envelope generator**, which steps up or down at a rate your program sets, in one of sixteen **shapes**. It is my favourite part of the chip.

The **envelope period** is a 16-bit number, R11 (the low byte) and R12 (the high byte). Like a tone counter, the envelope's counter counts ticks, and every `period` ticks (0 plays as 1) the envelope takes one **step**. A ramp from silence to full volume, or back, is **32 steps**, so it lasts

$$32 \times \text{period} \div 218{,}750 \text{ seconds}$$

from 146 µs at period 1 to 9.6 seconds at 65,535: a drum hit needs a few milliseconds, a slowly fading note a second or more.

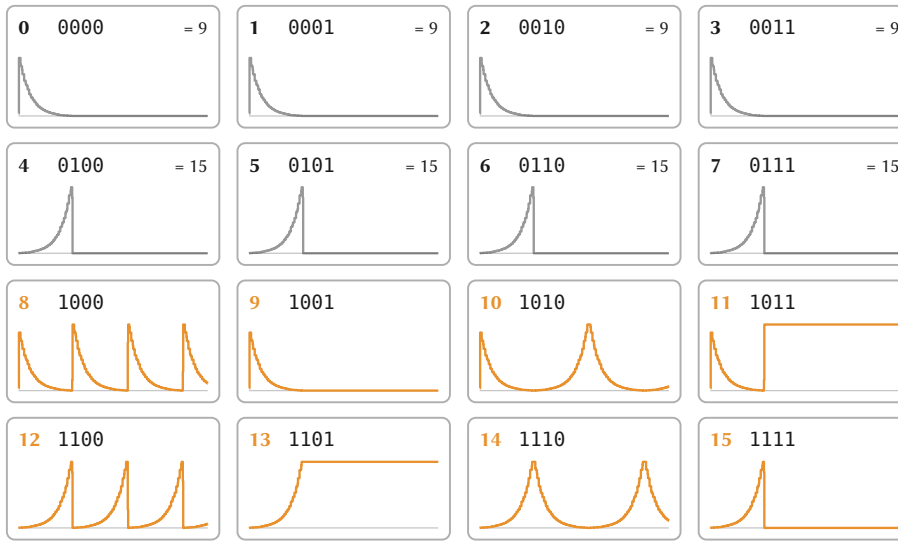
The sixteen envelope shapes

R13 bits 3–0 choose the shape, and each bit has a name:

- **ATT** (attack, bit 2): 1, the envelope starts at the bottom and climbs; 0, it starts at the top and falls.
- **CONT** (continue, bit 3): 0, after one ramp it drops to 0 and stays there, whatever the other two bits say.

- **HOLD** (bit 0), with CONT set: after one ramp it stops and holds its level.
- **ALT** (alternate, bit 1), with CONT set: the next ramp goes the other way. With HOLD, the envelope holds the level at the *opposite* end of where the ramp finished.

So **shape 13**, `1101` (CONT, ATT, HOLD), climbs and stays at full volume: a note that swells and then sustains. **Shape 14**, `1110` (CONT, ATT, ALT), climbs, turns, falls and turns again: a triangle, forever. Figure 22.4 shows all sixteen with the same period.



bits: CONT ATT ALT HOLD (R13 bits 3–0); 10 ms each, a ramp of 2.9 ms

Figure 22.4: The sixteen envelope shapes, captured from the machine.

With CONT clear, ALT and HOLD do nothing, so shapes 0–3 are the same as shape 9, and shapes 4–7 the same as shape 15. The ramps curve because every step goes through the logarithmic volume table, which your ear hears as an even climb. Sixteen shapes, eight sounds:

- Fall once and stay silent (0–3, 9), or climb once and fall silent (4–7, 15): a plucked or struck note.
- Fall, again and again (8), or climb, again and again (12): a saw tooth.
- Fall and climb (10), or climb and fall (14), forever: a triangle.
- Fall once and jump to full volume (11), or climb once and stay there (13).

Writing R13 restarts the envelope, even with the value it already holds: it goes back to the start of its shape at once. That is how your program strikes a new note with

the same shape, one write per note, and also why a player that rewrites R13 every frame "just to be sure" restarts the envelope every frame.

On the Next the restart also takes its first step at once, so the figure's first spike is a step short of the later ones: a falling shape starts at the second-highest level, 224 units instead of 255, and a rising one at the second-lowest. That is why shapes 0–3 and 9, which fall only once, never reach the top, while shape 8's later spikes do.



KLIVE IDE Catching envelope restarts

`bp-set $FFFD -o -log "[R13] restart" -if VAL == 13` logs a line each time a program selects R13. A line in every frame means the player rewrites R13 and restarts the envelope; drop `-log` and its message, and the breakpoint stops on the code that does it.

AY or YM

The **Peripheral 2 Setting register, NextReg `$06`**, bits 1–0, sets what all three of the Next's chips are:

- `00` (the power-on value): **YM2149**.
- `01`: **AY-3-8910**.
- `10`: the register list calls it "ZXN-8950", but the chips look only at bit 0, so it behaves exactly as YM2149.
- `11`: **holds all three chips in reset**: silent, registers cleared, writes ignored, until the mode is changed again.



KLIVE IDE Which chip, and who changed it

Select `$06` in the **Next Registers** view, and its **Audio chip mode** field names the mode the three chips are in. `bp-set nr:$06` stops on the write that changed it: the quickest way to explain chips that fell silent in mode 11.

What your program can see change between the two modes:

- **The volume table**: AY mode is louder below volume 15.
- **The envelope's resolution.** In YM mode, the envelope's 32 steps go through the 32-entry table, each to its own level, though the table's lowest entries come in equal pairs, so there are 29 different levels. In AY mode the envelope uses only its top four bits, so each of its 16 levels lasts two steps: the ramp takes exactly as long, in coarser steps.
- **The bits a register reads back.** In YM mode, every register reads back all 8 bits as written. In AY mode, the bits a register does not use read as 0: R1, R3, R5 and R13 keep their low 4 bits, R6, R8, R9 and R10 their low 5 bits; the others read whole.

- **Registers 16 to 31**, which do not exist: in YM mode they read `$FF`; in AY mode they read register number − 16, as if the top bit of the number were ignored. Writes to them are dropped in both modes.

So a program can find out which chip it is on by writing a value with the top bits set to R1, say, and reading it back: whole in YM mode, its low 4 bits in AY mode.

One more read is the Next's own: **port `$BFF5`** (an address that the data port `$BFFD` also answers) reads the selected chip's identity in bits 7–6 and the selected register's number in bits 4–0, for software that must know which chip and register it last selected.

Playing a chord

Three channels, three notes. C, E and G above middle C are periods 209, 166 and 140 (523.3, 658.9 and 781.3 Hz). Set once, they play until something changes them. A music player does not touch the chip while a note sounds. Once a frame, 50 times a second, it moves on in the tune: when a note ends, it writes the next note's period and volume, or restarts the envelope with R13.

The same once-a-frame player makes the classic effects, by changing a register a little every frame:

- **Vibrato**: the period moves up and down by one or two around the note, a wobble in pitch.
- **Tremolo**: the volume moves up and down, a wobble in loudness.
- **Arpeggio**: one channel cycles through the notes of a chord, a new one every frame or two, so a single voice sounds like a chord. Three channels can then play six or nine "notes", the chip-music sound.
- **Slides**: the period changes by a fixed amount every frame, for pitch bends and laser sounds.

▶ TRY IT Envelope Shapes

Part III, key 2. The demo plays this chord, C, E and G on channels A, B and C, for two seconds. Then channel A alone takes its volume from the envelope, with period 1,700, a ramp of a quarter of a second, and the sixteen shapes play in turn, two seconds each. Each shape's row shows its CONT, ATT, ALT and HOLD bits, and three ramps drawn from those bits; the row that plays is highlighted. The code is in `examples/code/demos/3-sound/envelope-shapes.kz80.asm`. A new shape is one write to R13:

```
    ; Channel A alone, its volume from the envelope
    ld hl,EsEnvelope
    call AyWriteList
    xor a
```

▶ TRY IT Envelope Shapes *(continued)*

```
EsShape:
    ld (EsCurrent),a
    ld e,a
     ld a,13                    ; R13: the shape; the write restarts the
envelope
    call AyWrite
    ld a,(EsCurrent)
    add a,EsFirstRow
    ld d,a
    call EsPlay
    jr nz,EsDone
    ld a,(EsCurrent)
    inc a
    cp 16
    jr nz,EsShape
```

`AyWrite`, in `examples/code/lib/sound.kz80.asm`, selects the register through port `$FFFD` and writes the value to port `$BFFD`. Listen to shapes 0 to 3 and 9: five numbers, one sound. Before it returns, the demo silences the chip with volume 0 on every channel.

Bass from the envelope

The envelope does not have to be slow. With a short period, a repeating shape is itself a waveform. Shape 8, a falling saw tooth, repeats every 32 steps, so with a period of 14 it plays 218,750 ÷ (32 × 14) = **488.3 Hz**. Shape 10, a triangle, takes 64 steps for one cycle, so a period of 7 gives the same pitch (Figure 22.5). Neither has a tone switched on: each is a steady channel whose volume follows the envelope.

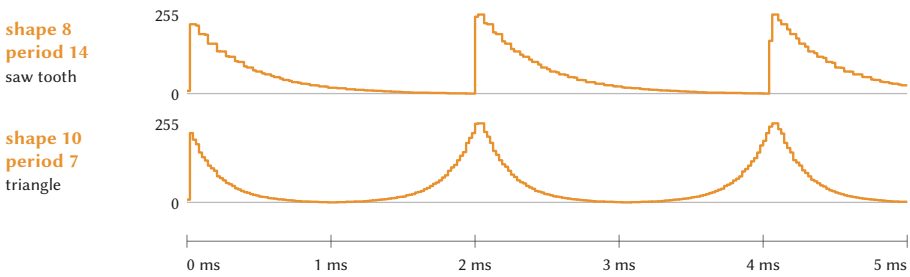


Figure 22.5: The envelope as a waveform: shapes 8 and 10, both at 488.3 Hz.

These waves sound much richer than the square tones, which made the “buzzer” bass a favourite of AY musicians. The price is that there is only one envelope per chip: a channel that plays the bass takes the envelope away from the other two. Musicians often switch on the channel’s tone as well, at a pitch related to the envelope’s: the mixer register then lets the envelope’s wave through only while the tone is high, which gives yet another colour.



KLIVE IDE Keeping a clip

Klive’s screen recording saves the machine’s sound in the video file, as MP4, WebM or MKV. Record the buzzer bass or the Envelope Shapes demo to listen to it again, or to share it.

In short. Each AY chip has 16 registers behind port `$FFFD` (select) and port `$BFFD` (data), and its counters step 218,750 times a second, so a tone plays at 218,750 ÷ (2 × period) Hz. A channel with tone and noise both off in R7 is not silent but a steady level, so silence it with volume 0. The three channels share one noise generator and one envelope, and any write to R13 restarts the envelope. AY mode is louder below volume 15 and has 16 envelope levels against the YM2149's 32.

CHAPTER 23

TurboSound: Three AY Chips Behind Two Ports

Two ports, three chips. In most of engineering, that sentence is the opening line of a bug report. On the Next, it is a feature. Load a 128K game that has never heard of the Next, and its tune plays on the first AY chip exactly as it did on the 128K. Load a Next tracker, and you hear nine channels at once, each chip parked at its own spot between your left and right ear. So how can three chips share two ports without the old software ever noticing that it has company?

Three chips give nine voices instead of three, enough for full chords, drums and sound effects that no longer steal a channel from the melody, and each chip can be placed left, right or in the middle.

This chapter builds on Chapter 22, which takes a single chip apart, and on Chapter 19 for stereo and panning. The chips are in YM2149 mode, and one channel at full volume adds 255 units to the mixer.

- How one small idea puts three chips behind the 128K's two ports
- How TurboSound is switched on and off, and why the chosen chip freezes
- How each chip's channels land between your ears in ABC, ACB or mono
- How much of the mixer nine channels take, and what programs do with them

Three chips, two ports

The Next has three AY chips, numbered **0**, **1 and 2**. All three listen to the 128K's two ports, **port `$FFFD`** (select a register) and **port `$BFFD`** (write it). But only one of them answers: the **selected chip**. Whatever you write, and whatever you read, goes to that chip.

Here is the trick. A register number is 0 to 31, so when a 128K program selects a register, **bits 7–5 are always 0**. TurboSound gives some of the other values a meaning of their own (Figure 23.1).

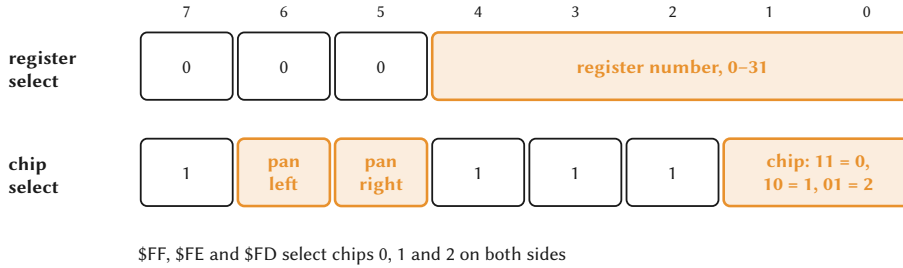


Figure 23.1: Port `$FFFD` written two ways, bit 7 on the left.

- **Bit 7** is 1: "this is not a register number".
- **Bits 4–2** are `111`: "I am choosing a chip".
- **Bits 1–0** pick the chip: `11` for chip 0, `10` for chip 1, `01` for chip 2. (`00` picks chip 0 as well.)
- **Bits 6–5** set that chip's **pan**: bit 6 lets the chip into the left side, bit 5 into the right.

With both pan bits set, that gives `$FF` for chip 0, `$FE` for chip 1 and `$FD` for chip 2. A value with bit 7 set but another pattern in bits 4–2, or with bit 7 clear but bits 6–5 not both 0, does nothing. An old program cannot select a chip by accident, because it never writes such a value, and a new program needs no new port. None of this works until TurboSound is switched on.

One note, five writes

To play a note on chip 1's channel A, you select the chip first, and then carry on as on a 128K:

1. Write `$FE` to port `$FFFD`: chip 1 is now the selected chip (and plays on both sides).
2. Write `0` to port `$FFFD`: you have selected register 0 of chip 1.
3. Write the period's low byte to port `$BFFD`: it lands in chip 1's R0.
4. Write `8` to port `$FFFD`, then the volume to port `$BFFD`: chip 1's R8, and the note sounds.

Each chip keeps its own selected register, so when you switch from one chip to another, select the register again rather than counting on the last one you used. The chip itself stays selected until your next chip select, which is why a music player usually visits each chip once per frame and writes all of its registers in one go.

Reading works the same way: port `$FFFD` gives you the selected register of the selected chip. And port `$BFF5` tells a program which chip and register it last selected (Chapter 22): its bits 7–6 read `11`, `10` or `01`, the same codes the select uses, and bits 4–0 hold the register.

TurboSound on, TurboSound off

TurboSound has a switch: bit 1 of the **Peripheral 3 Setting register**, **NextReg `$08`**. It is off after power-on, and then the Next behaves as a 128K:

- **Only the selected chip sounds.** The other two stay silent, whatever their registers hold.
- **The selection is frozen.** The Next ignores every chip select, so the selected chip cannot change. After power-on that is chip 0, which is why every 128K program plays on chip 0.

"Frozen" means the selection stays wherever it was. Switch TurboSound on, select chip 2, switch it off again, and chip 2 stays selected: all your writes go to it, and it is the only chip you hear. All three chips keep their registers meanwhile, and the other two come back with them when TurboSound is switched on again.



KLIVE IDE TurboSound's switch

Select `$08` in the **Next Registers** view, `show-nextregs $08`: bit 1 reads "Enable turbosound", and says that the selected AY is frozen while it is off. `bp-set nr:$08` stops on the write that switches it, so you can see whether a program ever does.

Figure 23.2 shows one program three ways. It selects each chip in turn and gives it one note on channel B: C on chip 0, E on chip 1, G on chip 2.

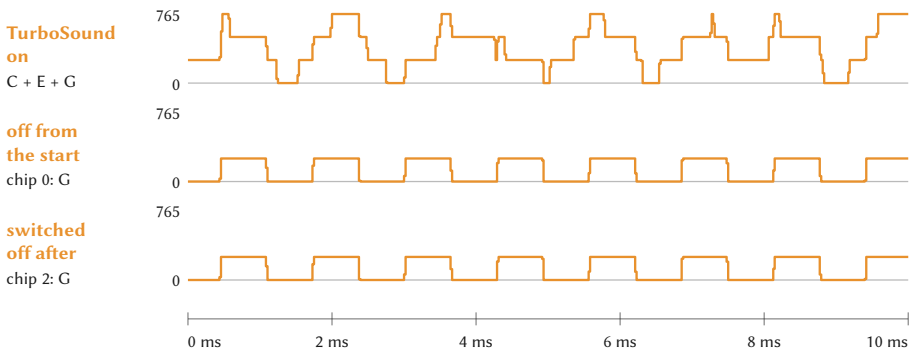


Figure 23.2: The same program with TurboSound on, off from the start, and switched off after the chord.

The middle capture is a TurboSound program on a Next where nobody switched TurboSound on: its selects are ignored, every write lands in chip 0, and the three

parts overwrite each other until only the last, G, is left. So a program that wants three chips must switch TurboSound on itself.

Where the channels go

Each chip has three channels, but you have only two ears. The Next gives you three ways to spread them, adding the channels up as it does:

- **ABC** (what you get after power-on): left = **A + B**, right = **C + B**. A sits on the left, C on the right, and B, which both sides hear equally, in the middle.
- **ACB**: left = **A + C**, right = **C + B**. A on the left, B on the right, and now **C is in the middle**.
- **Mono**: both sides = **A + B + C**.

Read the name from left to right, and the middle letter is the channel in the middle. That seat is where a musician puts the part that must not lean to either side, usually the bass or the melody.

Bit 5 of the **Peripheral 3 Setting register**, **NextReg `$08`**, chooses ACB (1) or ABC (0), for all three chips at once. Bits 7–5 of the **Peripheral 4 Setting register**, **NextReg `$09`**, put chips 2, 1 and 0 in mono, each on its own, and for a chip in mono, mono wins over ACB. With one chip at full volume:

- In ABC or ACB, each side adds two channels: at most 2 × 255 = **510**.
- In mono, each side adds all three: at most 3 × 255 = **765**, on both sides.



KLIVE IDE ABC, ACB and mono at a glance

The **Next Registers** view shows the layout before you blame the tune: `$08` bit 5 reads "AY stereo mode (0 = ABC, 1 = ACB)", and `$09` has a row per chip for bits 7–5, "Place AY 2 in mono mode" and so on.

Spectrum music was not all written for the same layout, and played with the other one, the middle part slides off to one side; mono suits tunes written for the 128K's single speaker.

Panning a whole chip

ABC, ACB and mono place the channels inside a chip. **Panning** then places the whole chip. Every chip has a 2-bit **pan**, set with bits 6–5 of the very value that selects the chip:

- `11`: both sides (what every chip has after a reset).
- `10`: left only.
- `01`: right only.
- `00`: muted.

The pan works on the chip's own mix, after ABC, ACB or mono: with pan `10`, the chip's left side goes to the mixer's left, and its right side is dropped. So with three chips each playing one note on channel B, chip 0's C panned left, chip 1's E to both sides and chip 2's G right, the left side plays C and E and the right side E and G.

Every chip select sets the pan again. Say you panned chip 1 to the left with `$DE` (`%1101 1110`: bit 6 set, bit 5 clear). Select it later with `$FE`, out of habit, and you have dragged it back to the middle. A pan written while TurboSound is off goes nowhere, ignored together with its select. My habit is to keep each chip's select value in one constant, pan bits included.



KLIVE IDE Watching chip selects

A logpoint lists every chip select in the Log pane, in binary, so a stray pan stands out: the chip is in bits 1–0, the pan in bits 6–5.

```
bp-set $FFFD -o -log "[TS] select {VAL:bits}" -if VAL >= $80
```

TRY IT Nine Channels

Part III, key 3. The demo switches TurboSound on and plays a C major chord on each chip, an octave apart: C3, E3 and G3 on chip 0, an octave higher on chip 1, and two octaves higher on chip 2. Chip 0 starts on the left, chip 1 on both sides and chip 2 on the right. Keys 1 to 3 move a chip's pan on, from both sides to left, right and muted, and key 4 switches between ABC and ACB; headphones make it plain. The code is in `examples/code/demos/3-sound/nine-channels.kz80.asm`. A new pan is a new chip select:

```
NcSelect:
    ld a,(NcChip)
    ld hl,NcPans
    add hl,a
    ld e,(hl)
    ld hl,NcPanBits
    ld a,e
    add hl,a
    ld e,(hl)               ; the pan's bits 6-5
    ld a,(NcChip)
    ld hl,NcChipBits
    add hl,a
    ld a,(hl)               ; %100111cc
    or e
    ld bc,$FFFD
    out (c),a
    ld (NcSelected),a
    ret
```

▶ TRY IT Nine Channels *(continued)*

The screen shows each chip's select value: `$DF` for chip 0 on the left, `$BD` for chip 2 on the right. Before it returns, the demo silences the chips, selects each one again with both pan bits set, and writes back the NextReg `$08` it found.

Nine channels in the mixer

The mixer adds the three chips' sides into one AY input per side:

- **Stereo** (ABC or ACB), every channel of every chip at full volume: 3 × 510 = **1,530** per side.
- **Mono** on all three chips: 3 × 765 = **2,295** per side, the most the AY input can ever carry.

The audio jack plays sums up to 4,095, and a silent Next already sits at 1,536 (Chapter 26), so 2,559 are left for everything else. Nine channels in stereo fit, with room for the beeper; three chips in mono fit on their own, but the beeper's 512 on top passes 4,095, the jack wraps around, and you hear a crackle.



KLIVE IDE Hearing the wrap

Klive builds the audio jack's output as the Next does, wrap past 4,095 included, so three chips in mono and the beeper crackle there too. Switch **Machine › Sound › Output** to HDMI, which plays the whole sum, and the crackle is gone.

When the chips are reset

A reset puts every chip's registers back to their reset values (all 0, R7 `$FF`), so all three fall silent; **chip 0 is selected**, and register 0 in it; and every **pan goes back to 11**, both sides. A soft reset does not touch the NextRegs, though: TurboSound's switch, ACB and the mono bits keep whatever they held, and only a hard reset clears them. So if your program relies on ABC, set NextReg `$08` yourself.



KLIVE IDE Soft and hard resets

Machine › ZX Spectrum Next › Hotkeys has the real machine's reset keys: **F4 - Soft Reset** keeps NextRegs `$08` and `$09`, and **F1 - Hard Reset** clears them. Run your program after F4 to check that it sets the layout it relies on itself.

Bits 1–0 of the **Peripheral 2 Setting register**, **NextReg `$06`**, set to `11`, also reset the chips, and keep them in reset for as long as the mode stays `11`: silent, cleared, chip 0 selected, every pan at both sides, and every write ignored. Choose YM or AY again, and they start afresh.

What programs do with nine channels

The most useful idea is to **keep music and effects apart**: play the music on chips 0 and 1 and the sound effects on chip 2, and an explosion never cuts a note short. Trackers for the Next write for all three chips at once, with chords, drums of their own and echoes repeated quietly on another chip.

In short. Three AY chips share ports `$FFFD` and `$BFFD`, and a `$FFFD` value with bit 7 set and bits 4 to 2 all 1 selects a chip and sets its pan. TurboSound is off after power-on and every chip select is then ignored, so a program wanting three chips must set NextReg `$08` bit 1 itself. Each chip keeps its own selected register, so select it again after switching chips. Every chip select also sets the pan, so `$FE` drags a left-panned chip 1 back to both sides.

CHAPTER 24

The 8-Bit DACs: Four Channels, Six Sound Cards

An AY chip can whistle, buzz and hiss, and with a little help it can even play a decent bass. But ask it to say "Game over" in a human voice, or to play a real snare drum, and it shrugs. It only knows square waves, noise and envelopes. A **DAC** knows nothing at all, and that is exactly its strength: it plays whatever number you give it. Give it the right numbers fast enough, and out comes a voice, a drum kit or a whole song from a file. The Next has four of them, and it answers on the ports of six different Spectrum and Pentagon sound cards, so the software written for those cards plays without a single change. How does one machine pretend to be six cards at once?

A drum hit on a DAC over AY music makes a chip tune sound like a band, and a spoken word or a sampled explosion is something no tone generator can fake. This chapter builds on Chapter 20, where a DAC is the purest **level writer**, which needs the CPU at every step, and on Chapter 19 for samples. Levels are a channel's own 8-bit value, `$00` to `$FF`, unless they are given in mixer units.

- What one DAC channel does, and how the four feed the mixer
- Which port of which sound card reaches which channel
- Who times the writes: a CPU loop, the CTC, the DMA or the Copper
- How to play a drum hit, and the DACs' share of the mixer

A number that holds

A DAC channel is an **8-bit register** with a converter behind it. Your program writes a value, `$00` to `$FF`, and the channel turns it into a level. Then it does nothing at all until the next write. No counter, no clock, no envelope: the level simply **holds**, so a DAC makes a waveform only if the program writes one step of it after another, on time.

`$80` is the **rest level**. A DAC's values run from 0 upwards, and a sound has to swing both ways, so silence is the middle of the range: a sample that goes up and down by the same amount swings about `$80`. Every channel starts there after a reset. A sample that ends at `$80` leaves the channel silent; one that ends anywhere else leaves

a small constant level, which you will not hear, since a capacitor after the converters removes it, but which takes some of the room the other sources could use. Treat `$80` as home.

Four channels, two sides

The Next has four channels, **A, B, C and D**. A and B belong to the left side, C and D to the right. Each side adds its two 8-bit channels into a 9-bit number, at most 255 + 255 = 510. The mixer multiplies it by 4, so the DACs can give a side up to **2,040 units**. Four channels are what it takes to copy every card: Soundrive drives four, two a side, Covox one a side, and the mono cards a single channel that you hear on both.

The DACs have a switch of their own: bit 3 of the **Peripheral 3 Setting register, NextReg `$08`**. It is 0 after power-on, so the DACs start **off**. While it is 0, every channel is held at `$80`, and every write to them, through a port or a NextReg, is ignored. Set the bit, and the channels start at `$80` and follow your writes. Clear it again, and they go back to `$80` at once: a quick way to silence all four.

Six sound cards on four channels

Every Spectrum and Pentagon sound card picked its own ports, and a program written for one card writes to that card's ports only. The Next answers on **all of them**. Under A to D is the **port**, only its low byte, that writes that channel (a dash: the card has none), and the **`$84` bit** is the bit of NextReg `$84` that enables the card (explained below the table). The NextRegs are not a card and need no bit:

Card	$84 bit	A	B	C	D
Soundrive mode 1	1	`$1F`	`$0F`	`$4F`	`$5F`
Soundrive mode 2	2	`$F1`	`$F3`	`$F9`	`$FB`
Profi Covox	3	`$3F`	–	–	`$5F`
Covox	4	–	`$0F`	`$4F`	–
Pentagon, ATM	5	`$FB`	–	–	`$FB`
GS Covox	6	–	`$B3`	`$B3`	–
Specdrum	7	`$DF`	–	–	`$DF`
NextRegs	no bit	`$2D`	`$2C`	`$2E`	`$2D`

A port is its low byte. The DACs look only at the low 8 bits of the port address, so `$00DF`, `$12DF` and `$7FDF` all reach Specdrum's channels. That keeps `out ($df),a` working whatever is in A, which is exactly what an `out (n),a` instruction puts in the high byte.



KLIVE IDE Breaking on port $DF

`bp-set $00DF -o -m $00FF` stops on every write to Specdrum's port and looks only at the low byte, as the DACs do: `$12DF` and `$7FDF` stop it too, whatever an `out (n),a` put in the high byte.

Mono ports write two channels. Pentagon's `$FB` and Specdrum's `$DF` write A and D, GS Covox's `$B3` writes B and C. One channel on each side, both set to the same value: the sample plays in the middle, at the same level as a stereo card playing it on both channels.

The enables. Each card's ports answer only while its bit is set in NextReg `$84`, one byte of the four **Internal Port Decoding Enables registers**, **NextRegs `$82`–`$85`** (bit 0 is the AY chips' ports). After power-on every bit is 1, so every card answers. A program that needs one of those ports for something else clears the bit, and the DAC lets go of it.



KLIVE IDE Which cards answer

Select `$84` in the **Next Registers** view, and each bit is named by its card's ports, "Port 0xdf dac mono specdrum" and so on: the live cards at a glance, and the bit to clear when a program needs one of their ports.

The ports two cards share. Some cards picked the same ports, and the Next sorts them out like this:

- `$0F` and `$4F` belong to both Soundrive mode 1 and Covox, and `$5F` to both Soundrive mode 1 and Profi Covox. They always reach the same channel, so two cards enabled at once simply agree.
- `$FB` is the one real clash. Soundrive mode 2 uses it for channel D alone; Pentagon uses it for A and D at once. **Soundrive mode 2 wins**: Pentagon's `$FB` answers only while bit 2 is clear, and then a write to `$FB` reaches both A and D.
- `$F1` and `$F9`, two of Soundrive mode 2's ports, end in the same two address bits as `$FD`, and the 128K's paging ports `$7FFD`, `$DFFD` and `$1FFD` look at only a few address bits. Without care, a sample written to `$7FF1` would also page your memory around. So while Soundrive mode 2 is on, a write to `$F1` or `$F9` reaches the DAC only.
- `$1F` is Soundrive mode 1's channel A for writes and the Kempston joystick for reads, and `$DF` is Specdrum's port for writes and, while Specdrum's bit is set and the mouse's ports are disabled, a second address of the joystick for reads. A write and a read never meet, so the DAC and the joystick share the ports in peace.

The NextReg mirrors

The last row of the table is the other way in. The **DAC B Mirror (left) register**, **NextReg `$2C`**, writes channel B, the **DAC A+D Mirror (mono) register**, **NextReg `$2D`**, writes A and D, and the **DAC C Mirror (right) register**, **NextReg `$2E`**, writes C. They are there for code that has no free port, and for the one device that can write NextRegs but no port at all: the **Copper**. Their writes need no NextReg `$84` bit, but they do need the DACs to be on. Reading the same three NextRegs gives the samples arriving from the Raspberry Pi instead (Chapter 25).



KLIVE IDE The DACs' switch

When a sample plays in silence, select `$08` in the **Next Registers** view: bit 3 reads "Enable 8-bit DACs (A,B,C,D)". While it is 0, every write to the DACs is ignored. `bp-set nr:$08` stops on the write that sets it, if any write ever does.

Who times the writes?

A DAC plays a sample at the rate your writes arrive: faster turns a voice into a chipmunk, and an uneven pace gives the sound a rough, gritty edge. So the real question is **who decides when** to write. Here are four answers, each playing the same 16-step sine.

A CPU loop. The simplest way of all: a loop that writes a step, waits, and goes round again. This one writes a step every 218 T-states:

```
        .org $8000
Start:  di
        nextreg $07,0     ; 3.5 MHz
        nextreg $08,$58   ; the DACs on (bit 3)
        ld hl,Table
Loop:   ld a,(hl)         ; 7
        out ($df),a       ; 11: Specdrum, channels A and D
        ld a,l            ; 4
        inc a             ; 4
        and $0f           ; 7: the next of 16 steps
        ld l,a            ; 4
        ld b,13           ; 7
Dly:    djnz Dly          ; 13 * 12 + 8
        jp Loop           ; 10: 218 T-states a step
        .org $9000
Table:  .defb 128,177,218,245,255,245,218,177
        .defb 128,79,38,11,1,11,38,79
```

At 3.5 MHz that is 3,500,000 ÷ 218 = 16,055 steps a second, and 16,055 ÷ 16 = **1,003 Hz**. The catch is that the loop's rate is the CPU's rate. Switch the CPU to 7 MHz with

the **CPU Speed register**, **NextReg $07**, and the same loop plays an octave higher; run it from contended memory at 3.5 MHz and it slows down while the screen is drawn. And while the loop plays, your program does nothing else.

The CTC (Chapter 6) counts the 28 MHz master clock, divided by a prescaler of 16 or 256 and by a **time constant** of 1 to 256. At every zero count it can interrupt the CPU, and the interrupt routine writes the next step:

```
        .org $8000
Start:  di
        nextreg $07,0
        nextreg $08,$58
        ld a,$be
        ld i,a
        im 2
        nextreg $22,$04   ; ULA and line interrupts off
        nextreg $c0,$a1   ; hardware IM 2, vectors from $A0
        nextreg $c9,$ff   ; clear the CTC status
        ld bc,$183b
        ld a,$85          ; control word: interrupt, timer,
                          ; prescaler 16, time constant next
        out (c),a
        ld a,109
        out (c),a
        ei
Park:   jr Park
Isr:    push af
        push hl
        ld hl,(Ptr)
        ld a,(hl)
        out ($df),a
        ld a,l
        inc a
        and $0f
        ld l,a
        ld (Ptr),hl
        pop hl
        pop af
        ei
        reti
Ptr:    .defw Table
        .org $9000
Table:  .defb 128,177,218,245,255,245,218,177
        .defb 128,79,38,11,1,11,38,79
        .org $bea6
        .defw Isr         ; CTC channel 0's vector
```

Channel 0's interrupt comes through vector `$A6`, so the word at `$BEA6`, the last line of the listing, holds the address of `Isr`. The rate is 28,000,000 ÷ 16 ÷ 109 = 16,055 interrupts a second, the same **1,003 Hz**, and it stays 1,003 Hz at every CPU speed, because the CTC never looks at the CPU's clock. The CPU still writes every sample, but only for the interrupt routine, about 150 T-states with the interrupt itself; between samples it is free to run your game.



KLIVE IDE Timing the interrupt routine

Put a breakpoint on `Isr` and one on its `reti`, and run from one to the other: the Z80 CPU panel's `TSP` shows what the routine costs. Add the interrupt's own T-states, and multiply by 16,055 to see how much of the CPU the sample takes.

The DMA (Chapter 5) copies bytes from memory to a port by itself, and its **prescaler** paces them: one byte every 32 × P master clocks, so about **875,000 ÷ P** bytes a second. Set it up once, point it at the sample and the DAC's port, and it plays the whole thing while the CPU does something else:

```
        .org $8000
Start:  di
        nextreg $07,0
        nextreg $08,$58
        ld hl,DmaProg
        ld b,16           ; the 16 bytes from DmaProg
Send:   ld a,(hl)
        out ($6b),a       ; the zxnDMA's port
        inc hl
        djnz Send
Park:   jr Park
DmaProg:
        .defb $83              ; disable
        .defb $7d,$00,$c0,16,0 ; WR0: A to B, $C000, 16 bytes
        .defb $14              ; WR1: A is memory, incrementing
        .defb $68,$21,55       ; WR2: B is I/O, fixed; P = 55
        .defb $cd,$df,$00      ; WR4: burst mode, B = $00DF
        .defb $a2              ; WR5: auto restart
        .defb $cf,$87          ; load, enable
        .org $c000
        .defb 128,177,218,245,255,245,218,177
        .defb 128,79,38,11,1,11,38,79
```

Burst mode makes the DMA give the bus back to the CPU while it waits for the next byte; in continuous mode it would keep the bus, and with it the CPU, for the whole block. **Auto restart** starts the block again at its end, so these 16 bytes play for ever. With P = 55, the prescaler alone would give 875,000 ÷ 55 = 15,909 bytes a second, 994 Hz. You get a little less, **968 Hz** at 3.5 MHz, because each byte also costs a few clocks

to take the bus back from the CPU, but the rate barely moves with the CPU's speed: 964 Hz at 7 MHz.

The Copper can write NextReg `$2D`, which writes channels A and D. So a Copper list that waits for each line and then moves the next sample into `$2D` plays a sample with no CPU at all, at one sample a line: 15,625 a second at 48K timing, 15,351 at 128K timing. But the samples sit in the list itself, which holds 1,024 instructions: fine for a short looping sound.



KLIVE IDE A sample in the Copper List

View › Machine Views › Copper List shows such a list with each `WAIT` on the raster ruler and each `MOVE` spelled out: the sample reads as a column of `$2D` writes, one a line, with the step's value beside each.

Figure 24.1 shows the CPU loop, the DMA and the CTC, each at two CPU speeds.

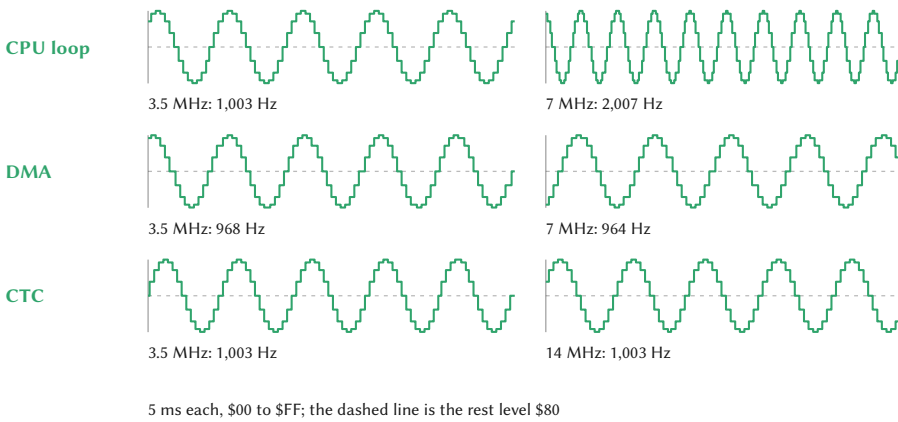


Figure 24.1: The same sine played three ways, each at two CPU speeds.

For a sample that plays alongside a game, use the **DMA** in burst mode: one setup, no CPU, and a rate that does not care how fast the CPU runs. For effects computed sample by sample, mixing two sounds, say, use the **CTC** and do the work in the interrupt routine. Keep the **CPU loop** for moments when nothing else runs, and the **Copper** for a short sound in a program whose Copper has a few instructions to spare.

A drum hit

Time for something that no generator can play. These are 2,048 bytes of noise that die away, a rough stand-in for a snare drum, played once by the DMA at prescaler 55, without auto restart (Figure 24.2).

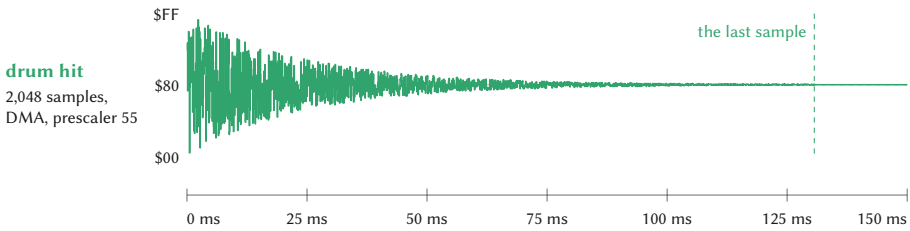


Figure 24.2: A drum hit: 2,048 samples played once by the DMA.

The block lasts 2,048 × 32 × 55 ÷ 28,000,000 = 128.7 ms by the prescaler alone, and a little longer with each byte's few extra clocks, about 131 ms. At the end the DMA stops, and the channel does what every DAC channel does: it **holds** the last byte. So a sample should end at `$80`, or near it. That is also why a drum over AY music costs so little: start the DMA, go back to the game, and the drum takes care of itself.

▶ TRY IT Drum Hit

Part III, key 4. The demo makes a drum like this one at `$C000`: 2,048 bytes of noise whose amplitude falls every 16 bytes, so that the last 700 or so are `$80`. Key 1 plays it through port `$DF` by the CTC, one interrupt and one byte 16,055 times a second; key 2 plays it with the DMA. While the drum plays, the CPU, at 3.5 MHz, goes round a short loop for six frames and counts, so each row shows how much of the CPU the drum leaves. In Klive, about 12,100 with nothing playing, 5,800 with the CTC and 11,600 with the DMA: at this speed the interrupt routine takes half the CPU, the DMA less than a twentieth. The code is in `examples/code/demos/3-sound/drum-hit.kz80.asm`, and its DMA program is this section's, written with `.dma`:

```
DhDma:
    .dma disable
    .dma wr0 a_to_b, transfer, DhSample, DhLength
    .dma wr1 memory, increment
    .dma wr2 io, fixed, 3t, 55  ; a byte every 32 x 55 clocks
    .dma wr4 burst, $00DF       ; give the bus back between bytes
    .dma wr5                    ; once, no auto restart
```

▶ TRY IT Drum Hit *(continued)*

```
    .dma load
    .dma enable
DhDmaEnd:
```

Open Klive IDE's Memory view on RAM page 0, which holds `$C000`, to see the bytes the demo made: wide swings at the start, settling down to `$80`.

The DACs in the mixer

At rest two channels at `$80` add 1,024 units a side, most of the 1,536 a silent Next sits at. A full-scale sample on both channels of a side swings from 0 to 2,040, four times the beeper's 512; through a mono port it swings about 1,020 (Chapter 26).

In short. A DAC channel holds the 8-bit value last written, so a sample plays at the rate its writes arrive and should end at the rest level, `$80`. The DACs are off after power-on, and every write is ignored until NextReg `$08` bit 3 is set. The Next answers on the ports of six sound cards, each enabled by a bit of NextReg `$84`, and `$DF` writes channels A and D at once. For a sample beside a game use the DMA in burst mode, since a CPU loop's pitch follows the CPU's speed.

CHAPTER 25

Audio from the Raspberry Pi: Samples on Four Wires

Open a Spectrum Next, and next to the FPGA you find a whole second computer: a Raspberry Pi Zero, with an ARM processor, hundreds of megabytes of memory and a Linux of its own. Compared with the Z80, it is a lorry parked next to a bicycle. A program on it can decode an MP3, speak from text, or pretend to be any sound chip you like. But a computer that clever is no use to the Next's sound unless its sound can get in. How do you pour the output of a Linux machine into a 1980s home computer, and even trick that home computer into loading a tape from it?

Music recorded in full, a long piece of speech, a whole tape played as sound for the Next to load: these are jobs for a machine with memory to spare and a processor to decode. The Pi does the heavy lifting, and the Next only has to listen. This chapter builds on Chapter 20, and on Chapter 21 for port `$FE` bit 6 and the MIC line. The Pi's samples are **signed 16-bit numbers**, −32,768 to 32,767, and the Next's are **10-bit words**, `$000` to `$3FF`.

- How the link carries words between the Pi and the Next
- How one register sets stereo, mono or a muted side
- How a sample from the Pi becomes the Next's 10-bit word
- Where the words go, including through a threshold into the tape input

---

Four wires and a clock from the Pi

The Pi and the Next talk sound over **I2S**, a link made for exactly one job: carrying a stream of stereo samples, one bit at a time, on a few wires:

- The **bit clock** ticks once for every bit sent.
- The **word select** says whose word the bits belong to: low for the left word, high for the right one.
- **Two data pins** carry the bits themselves, one in each direction.

On the Next, **the Pi is always the master**: it drives the bit clock and the word select, and the Next only listens to them. So the Pi, not the Next, decides the sample rate.

Whatever rate the Pi's software chooses, 44,100 or 48,000 a second or anything else, the Next takes the words as they come.



KLIVE IDE A Pi in Klive

The `pi-audio` command takes the Pi's place: `pi-audio "intro.wav"` plays a WAV file at any sample rate into the link, and `pi-audio -tone 440` an endless sine. The Next hears it as the hardware would, so a program must still switch the link on. The Pi itself is not emulated.

Each word starts one clock after the word select changes, its most significant bit first, and the Next **latches** it, takes the finished word and starts using it, at the next change: the left word at the word select's rise, in the middle of the period, and the right word at its fall, at the end. That half a period of difference matters later.

One register: NextReg $A2

Everything the Next does with the link is set in one register, the **PI I2S Audio Control register**, **NextReg `$A2`**:

- **Bits 7 and 6** enable the left and the right word. Both clear means the link is off: the Next ignores the Pi, and the receiver is held at rest. That is how it starts after a reset.
- **Bit 4** chooses which of the two data pins brings the Pi's samples in and which takes the Next's mix out. The register list calls the two settings "hats" (`0`) and "pi" (`1`).
- **Bits 3 and 2** mute the left and the right side.
- **Bit 0** sends the Pi's sound to EAR, the tape input, instead of to the mixer.
- **Bits 5 and 1** are reserved: the register list says to write bit 5 as 0 and bit 1 as 1.

Bits 7 and 6 do one more thing. With only one of them set, the side whose bit is clear does not go silent; it takes the other side's word. So `10` plays the Pi's left word on both sides, and `01` plays the right word on both. Figure 25.1 shows a stream with a 1,000 Hz sine on its left word and a 1,500 Hz sine on its right, in four settings.



KLIVE IDE Trying the settings

`pi-audio -tone 1000 -left` plays a tone on the left word alone. Write `$C2` to `$A2`, and you hear it on the left; `$82` plays it on both sides, and `$42`, the right word on both, falls silent. `show-nextregs $A2` shows `$A2`'s fields in the **Next Registers** view.

A muted side, a link that is off and a stream routed to EAR all give the mixer the same word: `$200`, the middle of the range, which is silence.

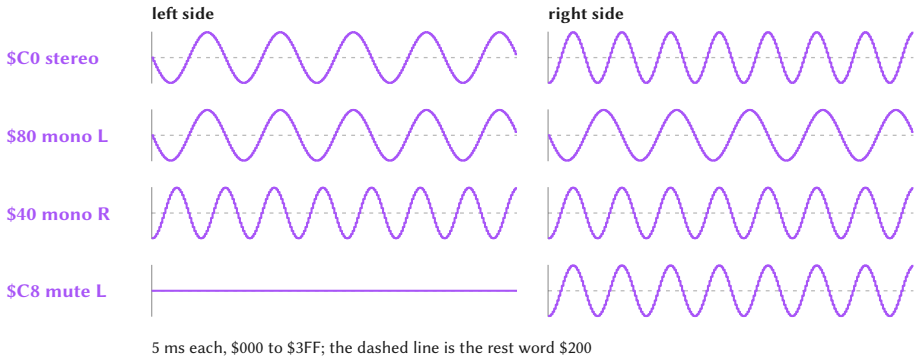


Figure 25.1: One stream in four settings of NextReg $A2, each side as the mixer gets it.

From a sample to a word

The Pi's software sends ordinary signed samples, typically 16 bits each, sometimes 24 or 32. The receiver keeps the **top 13 bits** of each word as it arrives, whatever the word's length, so a 16-, a 24- and a 32-bit stream all work: the extra bits fall off the bottom. The Next then uses the **top 10** of those and inverts the sign bit, which turns a signed number into one that counts from 0, the form every other source of the mixer uses: −32,768 becomes `$000`, 0 becomes `$200` and 32,767 becomes `$3FF`. For a 16-bit sample, the word is `((sample >> 6) & $3FF) xor $200`.

Ten bits match the rest of the Next: a DAC channel has 8 bits, and the mixer's whole range for a side is 13, so the Pi gets more steps than any one DAC.

When a word arrives

The left and the right word of one sample do not arrive together: the left word is latched half a sample period before the right. At a real sample rate the gap is tiny, about 10 µs at 48,000 samples a second, and no ear can hear it, but the tape input can see it. A link switched on starts at the rest word and takes the Pi's words from the next latch, and a link switched off goes back to rest at once. **If the Pi stops clocking**, nothing is latched any more, and the receiver holds its last two words: a stuck level with no sound coming in means the Pi has gone quiet, not that the Next has gone wrong.



KLIVE IDE When the Pi goes quiet

`pi-audio -off` stops the stream the way a Pi that stops clocking does: the receiver holds its last two words, and the level stays where the sound stopped.

Into the mixer

Each side's word goes into the mixer as it is: **0 to 1,023 units**, one unit a step, a quarter of a DAC step. So a Pi stream at full scale swings 1,023 units, about the same as one DAC channel at full scale, and twice the beeper's 512. At rest it adds 512 units, the other part of the 1,536 a silent Next sits at, next to the DACs' 1,024 (Chapter 26).

Reading the Pi's samples

A program can also read the words, through the same three NextRegs that write the DACs: **NextReg $2C** reads the left word and **NextReg $2E** the right one. But a NextReg has only eight bits, and a word has ten, so the read comes in two parts:

1. A read of NextReg `$2C` returns bits 9–2 of the left word, as the selection gives it: muted or off, that is the rest word. At the same moment it copies bits 1–0 into a latch.
2. A read of **NextReg $2D** returns those two latched bits in its bits 7 and 6.

NextReg `$2E` works the same way for the right word, through the same latch, which makes sure the low bits belong to the same word as the high bits you just read, however long you take. Most programs need only the top eight bits anyway: `$2C` alone gives a perfectly good 8-bit sample. With nothing connected, `$2C` and `$2E` read `$80` and `$2D` reads `$00`, the rest word in two parts.



KLIVE IDE A sample in the view

With a tone playing, pause the machine: the **Next Registers** view's current value of `$2C` is bits 9–2 of the left word, as a read would return it, and `$2E` the right. With the link off, both read `$80`.

▶ TRY IT Pi Audio

Part III, key 5. The demo switches the link on, both words to the mixer (NextReg `$A2` = `$C2`), and reads the left and the right word through NextRegs `$2C` and `$2E`. For each side it reads the register 500 times, at 3.5 MHz, and shows the lowest and the highest value it saw, and a bar with a cell for every 8 units between them: a loud, wide wave makes a long bar.

The demo only listens, so it is silent until something plays: with no Pi, both sides read `$80`, the rest word, and the bars stay empty. In Klive IDE, type `pi-audio -tone 440` while the demo runs, and the numbers and the bars come alive. The screen lists this command and `pi-audio -off`, which stops the stream. `pi-audio -tone 440 -left` puts the tone on the left word alone, and the right side stays at `$80`. On a real Next, start any sound on the Pi.

▶ TRY IT Pi Audio *(continued)*

The code is in `examples/code/demos/3-sound/pi-audio.kz80.asm`, and its measuring loop in the shared `examples/code/lib/nextreg.kz80.asm`. A NextReg stays selected after one write to port `$243B`, so the loop selects it once and reads port `$253B` again and again, keeping the lowest value in D and the highest in E:

```
; In: A the NextReg, HL the reads (500 here). Out: D lowest, E highest
ReadNextRegRange:
    ld bc,$243B
    out (c),a               ; select the NextReg once
    inc b                   ; BC: $253B
    ld de,$FF00             ; lowest so far $FF, highest $00
ReadNextRegRangeLoop:
    in a,(c)
    cp d
    jr nc,NotLower          ; not below the lowest so far
    ld d,a
NotLower:
    cp e
    jr c,NotHigher          ; below the highest so far
    ld e,a
NotHigher:
    dec hl
    ld a,h
    or l
    jr nz,ReadNextRegRangeLoop
    ret
```

The loop takes about 60 T-states a read, so 500 reads last about 8 ms: almost four cycles of 440 Hz, enough to meet a wave's peaks and troughs. A wave much lower than 125 Hz would show only part of its swing. When you leave, the demo switches the link off again, as it is after a reset.

Loading a tape from the Pi

Here is my favourite trick of the whole link. Set bit 0 of NextReg `$A2`, `$C3` with both words enabled and bit 1 written as the register list asks, and the Pi's words stop going to the mixer. Instead they drive a single bit, the **EAR bit**, which goes where a tape recorder's signal goes: into **port `$FE` bit 6**, the bit a tape loader reads. A program on the Pi plays a tape as sound, and the Next loads it, none the wiser. It even sounds like a tape loading, because the EAR bit also goes into the MIC line, which reaches the speaker.

How does a 10-bit word become one bit? Not with a plain threshold at the middle. A tape signal is never clean: at the moment a slow wave crosses a threshold, the noise

on it would flip the bit back and forth several times, and every flip looks like an edge to a loader. So the Next uses **hysteresis**, two thresholds instead of one:

- The bit **rises** when the top two bits of the left word and of the right word, ORed together, read `11`.
- It **falls** when they read `00`: both words' top two bits are `00`, so both are below `$100`, a quarter of the range.
- Anything in between, and the bit **holds** what it had.

You might expect the bit to rise only at three quarters of the range, at `$300`, where a word's own top bits read `11`. But the rule ORs the bits of *two* words, and the left word is latched half a period before the right. On a rising wave, at the moment it crosses the middle, the left word has just stepped to `$200` or a little above, top bits `10`, while the right word is still just below the middle, top bits `01`. And `10` OR `01` is `11`. So the bit rises right at the **upward zero crossing**, and falls only when the wave has dropped below a quarter (Figure 25.2).

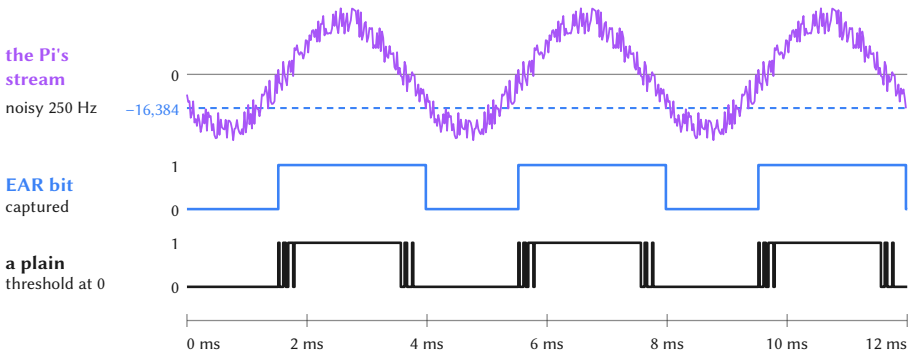


Figure 25.2: The EAR bit on a noisy 250 Hz wave from the Pi, against a plain threshold at 0.

The captured bit gives two clean edges a cycle, while a plain threshold at 0 chatters at every crossing. For a loader, that is the difference between loading and "R Tape loading error". For a real tape signal, which swings hard from one side to the other, it does not matter where exactly between the middle and a quarter the bit flips: the edges come at the same rate, and that is all a loader counts. A Z80 loop that reads port `$FE` bit 6 while the Pi streams a 1,000 Hz square wave counts 2,000 edges a second, as it would from a tape.



KLIVE IDE Loading a tape from a WAV

Play a WAV recording of a tape with `pi-audio`, write `$C3` to `$A2`, and type `LOAD ""` in 48K BASIC: the program loads through the hysteresis bit, with the loading stripes in the border and the screech in the speaker.

Port `$FE` bit 6 is the EAR bit XORed with the real tape input, ORed with the others the beeper chapter lists, so a quiet tape input leaves it as it is. The EAR bit is 0 whenever the link is off or the routing bit is clear.

The way back: the Next's sound to the Pi

On the other data pin, the Next sends the Pi its **whole mixed sound**, both sides, the same 13-bit sums the mixer adds, with the top bit inverted to make them signed again, at the top of each word. The Pi's software can record it, analyse it, or send it on. The Next needs no setting for this beyond enabling the link: whenever the Pi clocks a word in, the Next clocks its own word out.

In short. The Pi is always the I2S master, so it sets the sample rate, and the Next keeps the top ten bits of each sample with the sign bit inverted. A muted side, a link that is off and a stream routed to EAR through NextReg `$A2` all give the mixer the rest word `$200`. If the Pi stops clocking, the receiver holds its last two words. Setting bit 0 of `$A2` turns the words, with hysteresis, into the EAR bit at port `$FE` bit 6, so a tape can load from the Pi.

CHAPTER 26

The Mixer: One Sum, Three Outputs

Every sound source on the Next adds its number to one sum, 28 million times a second. Nothing is ever subtracted, so silence is not zero. And that one sum has to come out of a 3.5 mm jack, a small speaker inside the case and an HDMI cable, all at the same time. How does a single number become three different outputs, and how much room is there in it before the sound breaks?

Every program's sound passes through here, and whether it comes out clean depends on how its parts fit together. This chapter takes apart the mixer of Chapter 20, using the terms of Chapter 19. Levels are in the units of the Next's mixer, and each side is mixed on its own: the numbers are for one side.

- How five inputs add up in 13 bits without overflowing
- Why a silent Next adds up to 1,536
- How much headroom real music takes, and what the jack does with a sum it cannot hold
- How the sum leaves through the jack, the speaker and HDMI

Five inputs, one sum

The mixer has two sides, and on each it adds **five inputs**. The beeper gives two of them, one per bit of port `$FE`; each of the other three kinds of source gives one. The sum is taken again at every tick of the **28 MHz master clock**, so a change in any input reaches the sum within a 28-millionth of a second.

The sum is a **13-bit** number, 0 to 8,191. Each input is a binary number too, and the mixer lines it up at a fixed place in those 13 bits before adding (Figure 26.1).

- **EAR**, port `$FE` bit 4, is one bit placed at **bit 9 of the sum**: 512 while the port bit is 1, 0 while it is 0.
- **MIC** is one bit at **bit 7**: 128 or 0. It is port `$FE` bit 3, combined with the tape input and the Pi's EAR bit (Chapter 21). EAR and MIC count 4 to 1, as the 48K's resistors weighted them.
- **The AY chips** arrive as one 12-bit number per side, the channels of all three chips on that side already added together: **0 to 2,295**. One channel at full vol-

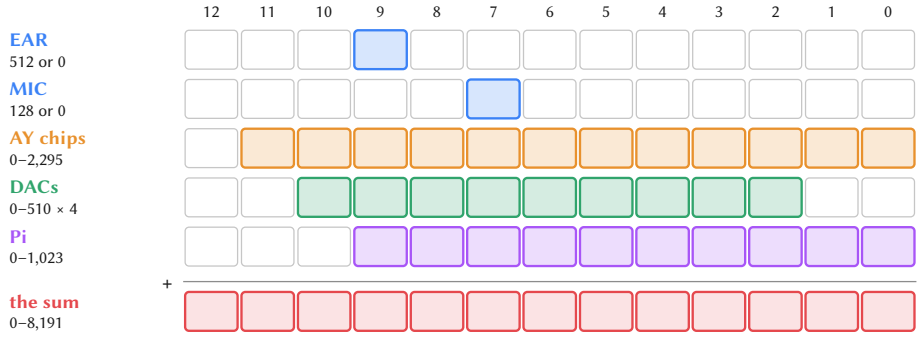


Figure 26.1: The five inputs as the bits of the sum they occupy.

ume is 255; in stereo each chip gives a side two channels, and with all three chips in mono, each gives both sides up to 765 (Chapter 23).

- **The DACs** arrive as one 9-bit number per side, the side's two 8-bit channels added (A + B on the left, C + D on the right), 0 to 510. Placed two bits up, it counts four times: **0 to 2,040**.
- **The Pi** arrives as a 10-bit sample per side, **0 to 1,023**, added as it is.

Every input at its largest gives 512 + 128 + 2,295 + 2,040 + 1,023 = **5,998**. The sum's 13 bits reach 8,191, so the sum itself can never overflow: the adder needs no check, and nothing is cut off.

The weights are no accident either. They keep EAR and MIC at the 48K's 4 to 1, and they make one DAC channel's full swing, 255 steps of 4, a little louder than all three channels of one AY chip together (1,020 against 765). That leaves room for drums and speech over AY music, which is what the Spectrum's DAC sound cards were for.

Silence is 1,536

Every input is a count upwards from zero, and the mixer never subtracts. A sound is a movement **up and down around some level**, and here that level is whatever the sources add while they are quiet. Two of them **rest in the middle** of their range:

- A **DAC** channel rests at `$80`, 128, halfway up its 0–255 range, so that a sample has room to swing both ways. The two channels of a side add 256, which counts four times: **1,024**.
- The **Pi** rests at 512, the middle of its 0–1,023, for the same reason.
- The **AY chips** and the **beeper** rest at 0: an AY channel at volume 0, and EAR and MIC low.

So a silent Next adds **1,536** on each side, and stays there however long nothing plays.

Why go to the trouble? Because of the moment a source is switched on. If the DACs rested at 0 and jumped to their middle when your program enabled them, the sum would leap by 1,024 in an instant, and the speaker would answer with a loud **click**. The Next avoids that by holding them at `$80` **even while they are switched off** with bit 3 of NextReg `$08`, so switching them on or off changes nothing you can hear. The Pi does the same: switched off or muted, it adds its resting 512. The beeper cannot: the first time a program sets port `$FE` bit 4, the sum rises by 512, which is exactly the click a 48K made too.

The constant 1,536 is never heard: after the converters, outside the FPGA, the jack's circuit lets only changes through, and what reaches your headphones is the sound moving around zero.



KLIVE IDE Where Klive listens

Klive's samples stand for the sound after the jack's capacitor, not the sum inside the FPGA: the DACs and the Pi count from their resting levels, and the beeper passes a filter that slowly removes its constant level. So silence is 0 in Klive, and a recording of it moves around zero, as your headphones hear it.

Headroom

How much of the range do real mixes use? Here are six of them, on the left side, as the hardware adds them:

1. **Silence**: the DACs at rest, 1,024, and the Pi at rest, 512: **1,536**.
2. **A 128K tune on one chip**, all three channels at full volume: on the left, channels A and B arrive, 2 × 255 = 510: **2,046**.
3. **TurboSound**, nine channels at full volume: three chips' 510 each, 1,530: **3,066**.
4. **The same, with the beeper on top**, EAR and MIC both high, 640: **3,706**.
5. **Three chips in mono** at full volume, each giving 765 to both sides, 2,295, with EAR on top: **4,343**.
6. **Every source at its largest**, the DACs and the Pi at full scale instead of resting: **5,998**.

The sum handles all six. **The audio jack does not.** Its converters take only the **lowest 12 bits** of the sum, 0 to 4,095, and leave bit 12 out. A sum below 4,096 arrives whole. A sum of 4,096 or more loses its top bit, which takes 4,096 off it: 4,343 arrives as 247. The sound does not stop at a ceiling, as clipping would; it **wraps around** to the bottom of the range, like an odometer rolling over from all nines to all zeros (Figure 26.2).

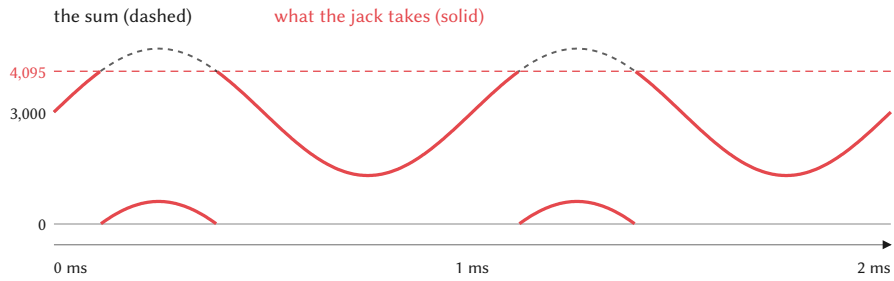


Figure 26.2: A tone too loud for the jack.

Each time the sum passes 4,095, the jack's value falls by 4,096, and the smooth wave becomes a series of jumps: a hard crackle, much harsher than a flattened top. In practice the jack has **2,559** units of room above silence, and it plays everything a 128K tune or TurboSound in stereo can produce, with the beeper on top. What goes past is the extreme: three chips in mono at full volume with the beeper, or a DAC or Pi sample near full scale on top of loud music. So keep your total below 4,096. HDMI, which carries all 13 bits, never wraps.



KLIVE IDE The jack's headroom in Klive

Klive multiplies the sum by 12.8, so the jack's 2,559 units above silence fill a 16-bit sample almost exactly, up to 32,755. Every mix the jack plays cleanly, Klive plays without loss, and the advice to stay below 4,096 holds there too.

▶ TRY IT Jack Wrap

Part III, key 6. The demo builds a sum the jack cannot hold. The three AY chips sit at full volume with their tones off, a constant 3 × 510 = 1,530 on a side. A 16-step sine of about 1 kHz swings all four DAC channels from 0 to 255, 2,040 units. The Pi, switched off, rests at 512. Key 2 leaves EAR low: the sum swings from 2,042 to 4,082, below 4,096, and the jack plays a clean tone.

Key 1 sets EAR, port `$FE` bit 4. That bit is a constant 512 and makes no sound by itself, but it lifts the sum to 4,594, and the top 499 units of every wave wrap round to the bottom: a harsh crackle from a bit you cannot hear. In Klive IDE, switch **Machine › Sound › Output** to HDMI while key 1 is on. HDMI carries all 13 bits, so the crackle is gone, and both keys sound the same. The code is in `examples/code/demos/3-sound/jack-wrap-code.kz80.asm`. This is its loop, one period of the sine; Specdrum's port `$DF` writes channels A and D, GS Covox's port `$B3` writes B and C, and `d` holds EAR's level:

▶ **TRY IT** Jack Wrap *(continued)*

```
JwStep:
    ld a,(hl)               ; the sine's next step, 0-255
    out ($DF),a             ; channels A and D
    out ($B3),a             ; channels B and C
    ld a,d
    out ($FE),a             ; EAR: $10 for key 1, 0 for key 2
    inc hl
    ld c,9
JwDelay:
    dec c                   ; about 200 T-states a step
    jr nz,JwDelay
    djnz JwStep
```

The audio jack: one pin, 4,096 levels

The jack's sound leaves the FPGA as **one pin per side** that can only be on or off, driven by a **one-bit converter** of the delta-sigma kind (Chapter 19). It works on the 28 MHz clock, one decision per tick:

1. The converter keeps a running total. At every tick it adds the input, the 12-bit value from the sum.
2. If the total is in the top half of its own range, the pin is **on** for this tick, and 4,096 is taken off the total. Otherwise the pin is **off**.
3. Over many ticks, the pin is on for exactly the share of ticks that the input is of 4,096. An input of 1,024 switches it on one tick in four; 1,536, three ticks in eight.

The pin's pulses are far too fast to hear. A **resistor and a capacitor** on the board after the pin, a low-pass filter, average them into a smooth voltage, and the circuit that removes the constant level leaves the sound alone for your headphones or amplifier.

So the jack has **no sample rate**: its converter follows the sum as it changes, at 28 MHz, so nothing is ever sampled and nothing is aliased on the way out. And its resolution is the whole 12 bits: a change of a single unit in the sum moves the share of on-ticks by one in 4,096.

The internal speaker

The Next has a small speaker of its own on the board, driven by one more one-bit converter of the same kind. It has no left and right, so it gets a **mono mix**: the left and right 12-bit values the jack takes, added together. Some boards bias the result to suit the transistor that drives the speaker. Two registers control it:

- The **Peripheral 3 Setting register**, **NextReg `$08`**, bit 4 switches the speaker **on** (1, the power-on value) or off. Off, the converter keeps running but its pin is held low.
- The **Peripheral 2 Setting register**, **NextReg `$06`**, bit 6 makes the speaker play **the beeper only**. With it set and the speaker on, EAR and MIC leave the mix of the jack and HDMI altogether, and the speaker gets a word of its own with only them in it: EAR at bit 10 and MIC at bit 8, 1,024 and 256, the same 4-to-1 ratio.

That mode keeps a program's key clicks and alerts in the machine while its music goes to the headphones. But the beeper's sound is moved, not copied: the headphones no longer hear it.



KLIVE IDE Hearing the beep move

Set `$06` bit 6, and switch **Machine › Sound › Output** between the jack and the speaker: on the speaker the beeper plays alone, as loud as it was on the jack, and on the jack it is gone. That is "moved, not copied", made audible.

HDMI audio

HDMI carries the picture as numbers, and its sound as numbers too: **48,000 samples a second** for each side, travelling between the lines of the picture. The Next builds each sample as a **16-bit word**: a 0, the whole 13-bit sum, and two 0 bits at the bottom. The word is therefore the sum times 4, and it never wraps: 5,998 arrives as 23,992. Bit 2 of the **Peripheral 4 Setting register**, **NextReg `$09`**, silences HDMI audio, for a television whose own speakers should stay quiet while the sound goes through the jack.



KLIVE IDE The output switches

The **Next Registers** view names the three output switches by field: `$08` bit 4, "Enable internal speaker"; `$06` bit 6, "Divert BEEP only to internal speaker"; and `$09` bit 2, "1 to silence hdmi audio".

Unlike the jack, HDMI **samples** the sum: 48,000 times a second it takes the sum's value at that moment. A change that falls between two samples is seen only at the next one, up to 21 µs late. That is enough for anything a human hears, but on the sharp edges of a high beeper tone it adds a faint roughness, a little aliasing, that the jack does not have.



KLIVE IDE Jack against HDMI

For the jack and the speaker, Klive averages each source over a sample's window; for HDMI it takes the sum's value at each sample's moment, as the hardware does. Play a high beeper tone through each output, and HDMI's faint roughness is there to hear.

In short. Each side adds five inputs into a 13-bit sum that can reach only 5,998 and so never overflows, and a silent Next sits at 1,536 because the DACs and the Pi rest in the middle of their ranges. The audio jack takes only the lowest 12 bits, so a sum of 4,096 or more wraps around to the bottom of the range and crackles. HDMI carries all 13 bits at 48,000 samples a second and never wraps, while the jack has no sample rate. Keep your total below 4,096, which leaves 2,559 units above silence.

CHAPTER 27

After the Sum: From the Mixer to Your Ears

The Next's mixer ends in a number: 13 bits per side, worked out again 28 million times a second inside the FPGA. Nobody has ever heard a number. Between that sum and your ears there is still a converter, a filter, a capacitor, a cable and, for HDMI, a whole second way of carrying sound. Two different sounds can leave the machine from one sum. So what happens on the way out, and why can a high beeper tone sound clean in your headphones and rough on your television?

This last sound chapter follows the sum out of the machine. It builds on Chapter 26, which builds the sum and its three outputs, and on Chapter 19 for sample rates and aliasing. Levels are in the units of the Next's mixer, for one side.

- What the jack's capacitor does to the sum
- Why the jack has no sample rate, and HDMI has one
- How point sampling folds a high beeper tone into notes nobody played
- Which output plays each source best

The sum and the sound

Inside the FPGA, the sum is a count from 0 upwards. A silent Next sits at 1,536, the DACs and the Pi resting in the middle of their ranges, and every sound moves up and down around whatever constant level its sources add. That is fine for a number, but not for a speaker: a constant level would only push the cone out and keep it there.

So the audio jack's circuit ends in a **capacitor**, which lets changes through but not a level that stays the same. After it, silence is 0, whatever the sum's resting level, and a sound moves around zero. Switch the beeper on from silence, and the sum jumps from 1,536 to a square between 1,536 and 2,048. After the capacitor, the first step comes through whole, 512 above zero, and the square then slides down until it is centred on zero, from +256 to −256 (Figure 27.1).

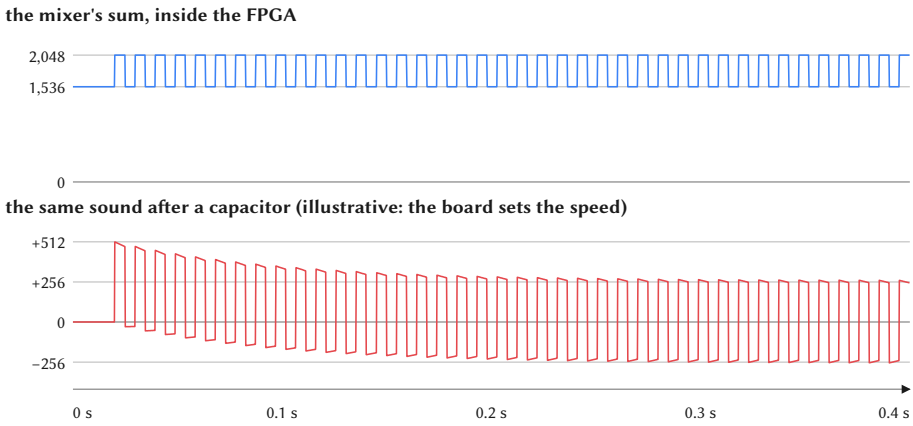


Figure 27.1: The beeper switched on from silence: the sum, and the same sound after a capacitor.

How fast it slides is set by the resistor and capacitor on the board, which are not part of the machine's logic, so the figure's speed is only an illustration. Either way it is far below any pitch: every note passes untouched, and only the constant level fades. The step at the start is the **click** you hear when a program first sets port `$FE` bit 4, as on the 48K. The DACs and the Pi make no such click, because they rest in the middle even while switched off. And an AY channel held at one fixed volume, a steady level, fades to silence after the capacitor in the same way.



KLIVE IDE Klive's capacitor

Klive plays the capacitor's part for the beeper with a high-pass filter at 1.4 Hz: the click at switch-on comes through whole, and the constant level drifts back to zero over a few tenths of a second. The speed is Klive's choice, since the real board's resistor and capacitor are not part of the machine's design.

The jack: no sample rate

The jack's sound leaves the FPGA as one pin per side, switched on and off at 28 MHz by a one-bit converter, and smoothed into a voltage by a resistor and a capacitor on the board. Nothing on that path takes samples: the converter follows the sum as it changes, at every tick of the master clock. So the jack has **no sample rate**, and nothing is aliased on the way out. A beeper edge comes out where the program put it, to the nearest 28-millionth of a second, and the harmonics of a square wave fade away above hearing in the filter, as they would from any analogue circuit. The internal speaker works the same way, with one more converter of the same kind and a mono mix.

The jack's limit is not time but size: it takes only the sum's lowest 12 bits, so a sum of 4,096 or more wraps round to the bottom of the range (Chapter 26).

HDMI: a number every 21 µs

HDMI carries sound as numbers, **48,000 samples a second** for each side, travelling between the lines of the picture. Each is a 16-bit word, the whole 13-bit sum times 4, so HDMI never wraps. It also never removes the constant level: a silent Next sends 1,536 × 4 = 6,144 in every word, and it is the television or the amplifier whose own circuits take it away.



KLIVE IDE What Klive leaves out

Two differences, neither of which you can hear. Klive filters only the beeper, so an AY channel held at one volume stays in its samples as a steady offset instead of fading. And Klive's HDMI output is centred on zero, where the real words carry the silent 6,144.

HDMI does take samples, and it takes them by **point sampling**: at each of its 48,000 moments it takes the sum's value as it is, without looking at what happened since the last one. A beeper edge that falls between two moments is seen only at the next, up to 21 µs late. For a single edge, nobody hears that. For a high beeper tone, it matters.

A square wave is not one tone: it has harmonics at three, five, seven times its pitch, and the higher ones lie far above the 24,000 Hz that 48,000 samples a second can hold. Point sampling does not drop them; it **folds them back** into the range you hear, as aliases. Take a beeper loop of 187 T-states a half period at 3.5 MHz: 3,500,000 ÷ 374 = **9,358 Hz**. Its fifth harmonic, 46,791 Hz, folds back to 48,000 − 46,791 = **1,209 Hz**, a low growl the program never played. The third, 28,075 Hz, folds to 19,925 Hz, at the very top of hearing, and the others land in between (Figure 27.2).

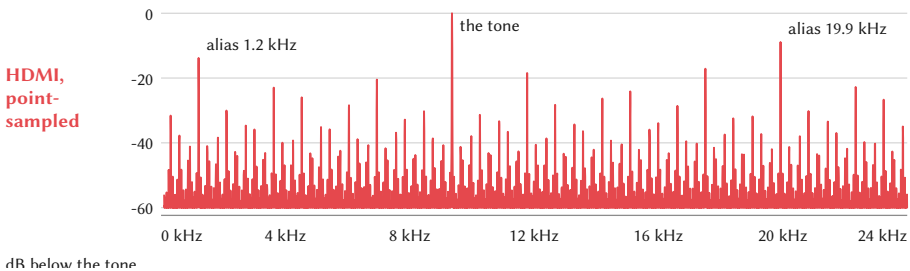


Figure 27.2: A 9,358 Hz beeper tone point-sampled as HDMI takes it.



KLIVE IDE Hearing the growl

Play the 9,358 Hz tone, 187 T-states a half period at 3.5 MHz, with **Machine › Sound › Output** set to HDMI, and the 1,209 Hz alias is there. Switch to the jack and it nearly vanishes: there Klive averages each sample's window, which leaves about half a per cent of it.

The jack plays the same tone with none of these: the tone, and its harmonics fading away above hearing. That is why a high beeper tune can sound clean in headphones on the jack and rough through a television.

Which output for which source

- **The beeper**: the jack or the speaker for the cleanest sound. HDMI is fine for low notes, but adds aliases to high ones, and moves each edge by up to 21 µs.
- **The AY chips**: anywhere. Their square waves fold back on HDMI too, but tunes rarely sit high enough for it to matter.
- **The DACs and the Pi**: anywhere. Their levels change only when a new value arrives, at sample rates of their own, so HDMI's moments lose little.
- **A loud mix**: HDMI, or keep the total below 4,096 for the jack, which wraps above it.

The whole of the sound

That is the Next's sound from end to end. The beeper's one bit and the tape's MIC line, three AY chips behind two ports, four DACs answering on six cards' ports, and a Raspberry Pi's stream all add their numbers into one 13-bit sum per side, 28 million times a second, from a silent level of 1,536. The sum leaves through a one-bit converter and a capacitor to the jack and the speaker, with no sample rate and a 12-bit limit, and through HDMI as 48,000 words a second that never wrap but fold high harmonics back.

In short. The mixer's sum is a count from 0 upwards with a silent level of 1,536, and the capacitor after the audio jack removes that constant level, so a sound moves around zero and switching the beeper on gives a click. The jack and the speaker have no sample rate, so nothing is aliased on the way out. HDMI point-samples the sum 48,000 times a second and keeps the constant level, so a high beeper tone's harmonics fold back as aliases, such as 1,209 Hz from a 9,358 Hz tone.

Part IV

Talking to the World

CHAPTER 28

Storage: an SD Card on a Serial Bus

A Spectrum loaded its games from a cassette, at 1,000 to 2,000 bits a second, and you could make a cup of tea while it did. A Next keeps gigabytes on an SD card and loads a whole level before you have reached for the kettle. Yet the CPU still talks to that card through a single 8-bit port, one byte at a time, as if it were a humble peripheral. How does a byte get from a flash chip inside the card into your program, and why is it so fast?

NextZXOS and esxDOS hide storage behind a file API, but streaming a sample from the card, loading a level while the music plays, or a loader that works at 3.5 MHz and fails at 28 all depend on what sits under it. This chapter builds on Chapter 3 for the DivMMC, on Chapter 5 for the DMA, and on Chapter 21 for the cassette, which the Next keeps too.

- How SPI works, and how five devices share it
- How ports `$E7` and `$EB` choose a device and move a byte
- How to talk to an SD card, with a sector read written out by hand
- How the DMA, the flash chip, the Pi and the DivMMC fit in

Five devices on one bus

An SD card can talk in two ways: its own fast four-bit bus, and **SPI**, the Serial Peripheral Interface, a simple serial bus that almost every microcontroller speaks. The Next uses SPI. An SPI bus has a **master**, which runs the show, and **slaves**, which answer. Four wires are enough:

- **SCK**, the clock, driven by the master. One bit moves in each direction per clock pulse.
- **MOSI**, master out, slave in: the master's data, one bit at a time, the most significant bit first.
- **MISO**, master in, slave out: the slave's data, at the same time, on the same clock.
- **A select line** per slave, active low. Only the selected slave listens and drives MISO; the others stay quiet.

The Next's FPGA is always the master. Five slaves share its bus:

- **Two SD cards**, card 0 and card 1. The FPGA has a select line for each.
- **Two SPI devices on the Raspberry Pi's connector**, once the **PI Peripheral Enable register**, **NextReg $A0**, bit 0, turns those GPIO pins into an SPI bus.
- **The FPGA's flash chip**, which holds the cores the machine starts from.

SPI is **full duplex**: every byte the master sends brings a byte back, whether anyone wanted it or not. To read, the master sends a byte that means nothing, `$FF`; to write, it ignores what comes back.

Choosing a device

Port **$E7**, the SPI chip select, chooses the slave. You write it, and its low bits decide:

- **Low bits 10**, such as `$FE`, select **SD card 0**, whatever the other six bits are.
- **Low bits 01**, such as `$FD`, select **SD card 1**, likewise.
- **$FB** selects the Pi's SPI device 0, and **$F7** its device 1. These need the exact value.
- **$7F** selects the flash, but only in config mode, when the machine type in NextReg `$03` is 0, or after a power-on until the first soft reset. At any other time it deselects everything.
- **Anything else** deselects every slave.

The SD cards are forgiving because SD card software from other machines writes junk into the upper bits. The selection cannot be read back, so keep your own copy if you need it. After a reset nothing is selected, and bit 3 of NextReg `$83` switches both ports, `$E7` and `$EB`, off: they then do nothing, and a read gives `$FF`.

The two cards can swap places. Bit 5 of the **Peripheral 5 Setting register**, **NextReg $0A**, makes `$FE` select card 1 and `$FD` card 0 from the next write of `$E7` on, so the system can start from either card without changing its code. It can be written only in config mode, and only by a write that also sets bit 2, which guards it and always reads 0; a hard reset clears it. The swap is a feature of the core (from 3.02.03, with the lock from 3.02.04), so it works on every board. The Issue 5 board is the first with two card readers on board, an internal one and an external one, wired to the two select lines.



KLIVE IDE Two cards in Klive

The **ZX Spectrum Next (KS3, core 3.03.03)** model has a second reader: **Machine › SD Card › Select Image for SD Card 2...** puts a `.cim`, `.img` or `.hdf` image in it. That is the card `$E7` selects with `$FD`, and Klive's models swap the two with NextReg `$0A` bit 5 as the core does.

One byte, sixteen clocks

Port **$EB**, SPI data, moves the bytes. The SPI master runs on the **CPU's clock**, and SCK is half of it, so one byte takes **16 CPU clocks**, which is 16 T-states at every CPU speed: eight pulses of SCK, each two clocks long. The bus works in what SPI calls mode 0: SCK rests low, the master changes MOSI while SCK is low, and the slave's MISO is sampled as SCK rises. In Figure 28.1 the host writes $51, the first byte of the read command we meet below, and the card, which has nothing to say while a command arrives, answers with $FF.

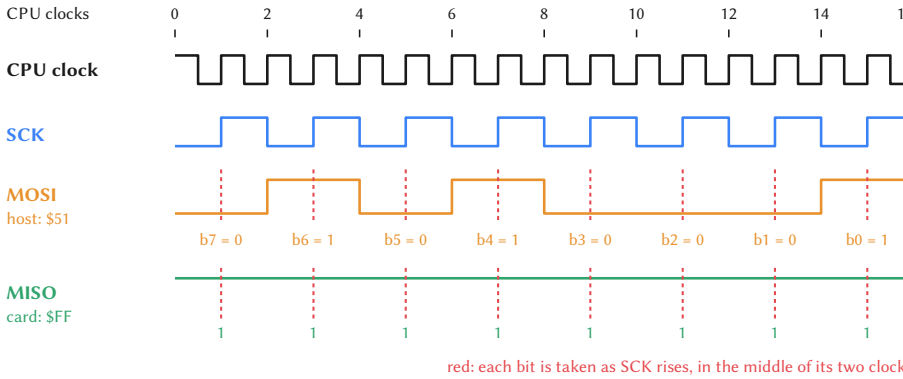


Figure 28.1: One byte on the SPI bus: $51 out while $FF comes back, in CPU clocks.

Read the MOSI row at the red marks, left to right, and you get `0101 0001`, $51, with the most significant bit first. Read the MISO row the same way and you get eight 1s. When the sixteenth clock ends, the $51 has left and the $FF has arrived, and the next read of $EB will return that $FF. Three rules follow:

- **A write to $EB sends the byte** you wrote. The byte that comes back is kept.
- **A read of $EB returns the byte that came back during the previous exchange**, and starts a new exchange that sends $FF. So a read is always one exchange behind: you get the answer to what was sent before. This sounds awkward, but it is what makes `INIR` work.
- **A new exchange starts only when the last one is finished**, or in its very last clock. An access to $EB sooner than that is simply lost: a write sends nothing, and a read returns the old byte and starts nothing. The CPU gets no wait state to warn it.

The last rule is the one that bites, because the SPI master counts the CPU's own clocks. Two `OUT (C),r` in a row come 12 T-states apart, and 12 is less than 16: the second byte is lost, at 3.5 MHz just as at 28. Two `IN r,(C)` in a row lose the second exchange

the same way. `OTIR` and `INIR`, at 21 T-states a byte, are fine, and so is a polling loop. My rule of thumb: never two accesses to `$EB` with nothing between them; send a command from a table with `OTIR`, and put a `NOP` between two lone `IN`s.



KLIVE IDE Where Klive is kinder

Klive's card answers command by command, but Klive does not model the 16-clock exchange: a byte sent too soon still arrives, and a read returns the card's next byte at once. Code that breaks the rule can work in Klive and fail on a Next, so test any code that touches `$EB` directly on the real machine, or keep to `OTIR`, `INIR` and the `NOP`.

The speed follows the CPU. SCK is 1.75 MHz at 3.5 MHz and 14 MHz at 28 MHz, so the bus can move about 219K to 1.75 MB a second; in practice the program around it decides. At 28 MHz the bus is some ten thousand times faster than a tape.

Talking to an SD card

The card is not part of the FPGA; it is a small computer of its own, with a controller that speaks the SD card specification. In SPI mode the conversation goes like this:

- **A command is six bytes**: `$40` plus the command's number, a 32-bit argument, most significant byte first, and a CRC byte. The specification names each command after its number: **CMD*n*** is the command whose first byte is `$40` + *n*, so CMD17 starts with `$51`. The CRC is checked only for the first two commands a card gets when it starts, CMD0 and CMD8 below; after that a card in SPI mode ignores it.
- **The card answers after a pause.** Between zero and eight `$FF` bytes come first, then a one-byte response, **R1**. `$00` means "done, no error"; bit 0, `$01`, means "still in the idle state"; any other bit is an error.
- **`$FF` is silence.** Between commands the master keeps sending `$FF`, and the card answers `$FF`. A command always starts with the bits `01`, so `$FF` can never be mistaken for one.

Some commands are **application commands**, named **ACMD*n***. They are sent in two parts: first CMD55, whose only job is to say "the next command is an application command", then the command itself. Starting a card takes a handful of commands, which the driver sends once:

1. **CMD0, go to the idle state**, resets the card. The card first gets at least 74 clocks with nothing selected; then CMD0 with its select line low makes it leave its own bus and speak SPI from then on, and it answers `$01`, idle.
2. **CMD8, send the interface condition**, tells the card the voltage the host supplies and a check pattern. A card that understands it echoes both back, and that tells the driver it is a modern card, version 2.0 of the specification or later; an older card answers that the command is illegal.

3. **CMD55** then **ACMD41**, **send the operating condition**, starts the card's own start-up, and its argument says that the host can handle a high-capacity card. The card answers `$01` while it is still busy with its start-up, so the driver repeats the pair until the answer is `$00`: the card is ready.
4. **CMD58**, **read the OCR**, reads the card's 32-bit **operating conditions register**. One of its bits says whether the card is **SDHC** or larger, a high-capacity card. Such a card is addressed in **sectors** of 512 bytes, not in bytes, so sector 5 is argument 5, where an older card would need byte address 2,560.

After that, reading and writing are one command each:

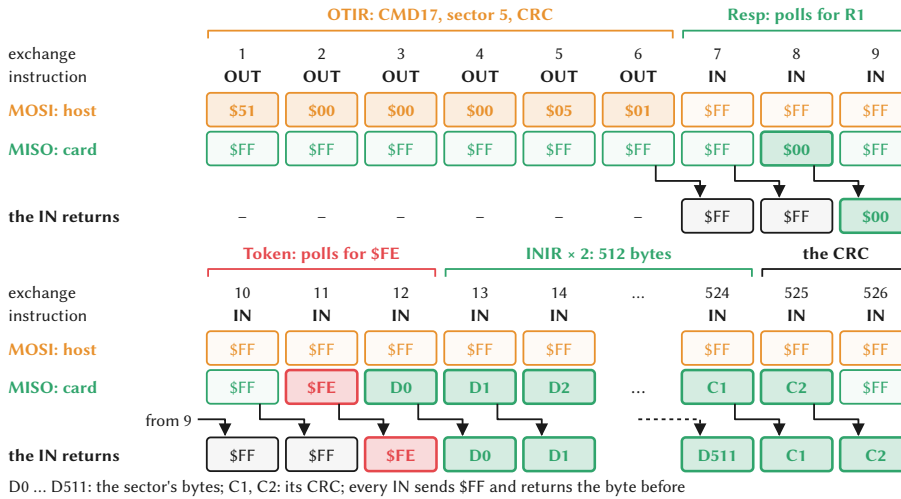
- **CMD17**, **read a single block**, **reads one sector.** After R1 the card sends `$FF` while it fetches the data, then the **data token** `$FE`, the 512 bytes and a 16-bit CRC.
- **CMD24**, **write a block**, **writes one sector.** After R1 the host sends `$FE`, the 512 bytes and a CRC; the card answers a **data response** byte, `xxx00101` when it accepted the block, and then `$00` while it is busy writing.
- **CMD18**, **read multiple blocks**, **reads sector after sector**, each with its own `$FE` and CRC, until the host sends **CMD12**, **stop transmission**. That is how a file of many sectors streams without a command per sector.

Let's read a sector. This routine assumes a card that is started, selected, and SDHC:

```
; read sector DE (high word 0) of an SDHC card to (HL)
ReadSec: ld a,d
         ld (Cmd17+3),a         ; sector bits 15-8
         ld a,e
         ld (Cmd17+4),a         ; sector bits 7-0
         push hl
         ld hl,Cmd17
         ld bc,$06eb            ; 6 bytes to the SPI data port
         otir
         pop hl
Resp:    in a,(c)               ; $FF until R1 comes
         cp $ff
         jr z,Resp
         or a
         ret nz                 ; R1 not 0: an error
Token:   in a,(c)               ; $FF until the token
         cp $fe
         jr nz,Token
         ld b,0
         inir                   ; bytes 0-255
         inir                   ; bytes 256-511
         in a,(c)               ; CRC, high byte
         nop                    ; keep 16 T-states apart
```

```
        in a,(c)                ; CRC, low byte
        ret
Cmd17:  .defb $51,0,0,0,0,$01   ; CMD17, sector, CRC
```

The sector number goes into bytes 3 and 4 of the command at `Cmd17`, most significant byte first; the table lives in RAM, so the routine can patch it. The CRC byte, `$01`, is ignored by a card in SPI mode after CMD8. A real driver also gives up after a while in both polling loops, and checks the CRC if it cares. Figure 28.2 shows the whole read of sector 5, exchange by exchange, from a card that answers R1 after one `$FF` and sends its token after two more.



D0 ... D511: the sector's bytes; C1, C2: its CRC; every IN sends $FF and returns the byte before

Figure 28.2: The listing's sector read, one column per exchange.

- **Exchanges 1–6** are `OTIR` sending the command: `$51`, the sector number `$00 $00 $00 $05`, and the CRC byte, 21 T-states apart, so none is lost. The card sends `$FF` all the while, and no instruction reads it.
- **Exchanges 7–9** are `Resp`. Each `IN` sends `$FF`. The card sends R1, `$00`, in exchange 8, but the `IN` of exchange 8 returns what came in during exchange 7, `$FF`. Only exchange 9's `IN` returns the `$00`.
- **Exchanges 10–12** are `Token`. The card sends `$FE` in exchange 11, exchange 12's `IN` returns it, and that same exchange has already brought in the first data byte, D0, so the first `IN` of `INIR` returns byte 0.
- **From exchange 13** `INIR` reads one data byte per exchange, each one an exchange late: D0 in 13, D511 in 524. The two last `IN`s, 525 and 526, with a `NOP` between them, return the CRC.



KLIVE IDE The conversation in the Log pane

Two logpoints log every byte the program sends and gets, in order; switch them on just around a sector read, or the Log pane fills fast.

```
bp-set $00EB -o -m $00FF -log "[SD] > {VAL:hex8}"
bp-set $00EB -i -m $00FF -log "[SD] < {VAL:hex8}"
```

The conversation is one exchange longer than the bytes it carries: the last `IN` sends a `$FF` that brings back a byte nobody reads. That is the price of reading behind, and the reason the data comes out in the right order.

TRY IT SD Sector

Part IV, key 1. The demo starts SD card 0 with the four steps above, reads sector 0 with CMD17 at 3.5 MHz, and shows it 128 bytes a page, keys 1–4. It opens on page 4, where an MBR keeps its partition table, in cyan, and its signature `$55 $AA`, in yellow. Rows 2 and 3 show what the card answered: R1 for CMD0, CMD8 and ACMD41, the OCR, and how many times ACMD41 went out. The card code is in `examples/code/lib/sd.kz80.asm`, the screen in `examples/code/demos/4-world/sd-sector.kz80.asm`. This is step 3, a frame between tries:

```
SdInitTry:
    ld a,(SdTries)
    inc a
    ld (SdTries),a
    ld hl,SdCmd55           ; the next command is an application command
    call SdCommand
    jr c,SdFail
    ld hl,SdAcmd41          ; start up: $01 while busy, $00 when ready
    call SdCommand
    ld (SdR1Acmd41),a
    jr c,SdFail
    or a
    jr z,SdReady
    cp $01
    jr nz,SdFail
    halt                    ; give the card a frame
    ld a,(SdTries)
    cp 250                  ; about five seconds
    jr c,SdInitTry
    jr SdFail
```

Every command goes through `SdCommand`, which checks its first byte against six read-only commands and sends nothing else, so the demo cannot write to the card. Klive's card is ready at the first ACMD41, where a real card usually needs a few tries. Run it from Klive IDE, which boots from the SD card image; `klive run` starts without one.

Streaming with the DMA

The DMA can do the `INIR` part for you, and the SD card's port is the one I/O port the DMA treats with special care: the SPI master holds the DMA with **wait states** until the byte under way has finished. So a DMA transfer from port `$EB`, fixed, to memory, incrementing, never reads too early and never loses a byte, at any CPU speed. Each of its reads returns the byte before and starts the next exchange, just as the CPU's do, so the DMA starts after the data token, as `INIR` did. To load while a game runs, the CPU sends the command and waits for the token, then hands the 512 bytes to the DMA in burst mode, or in continuous mode for the fastest load (Chapter 5).

The flash and the Pi

The flash chip holds the FPGA's cores. The board loads the FPGA from it when the power comes on, before any Z80 code runs. The Next's programs reach it only to update a core, which is why `$E7` selects it only in config mode or straight after a power-on: an ordinary program cannot select by accident the chip the machine starts from, the one place where a stray `OUT` could turn a computer into a brick.

The Raspberry Pi's connector carries an SPI bus too, once NextReg `$A0` bit 0 gives it the GPIO pins, with two selects, `$FB` and `$F7`. The Next is the master there as well, so a Pi, or any SPI device on that connector, is a slave the program talks to exactly as it talks to the card.

The DivMMC's part

The SPI master moves bytes; it knows nothing of files. Files are the business of the DivMMC, the butler of Chapter 3, and its tools are mostly memory. Its 8K ROM holds the start of the SD card's operating system, and it has **128K of RAM**, sixteen banks of 8K, for the rest of its code, its buffers and its tables. Port **`$E3`**, the DivMMC control port, sets it up:

- **Bit 7, CONMEM**, maps the DivMMC in by hand: its ROM at `$0000`, read-only, and a RAM bank at `$2000`.
- **Bits 3–0** choose that bank, 0 to 15.
- **Bit 6, MAPRAM**, replaces the ROM with **RAM bank 3**: the operating system runs from RAM, and bank 3 becomes read-only, at `$0000` and also at `$2000` if bits 3–0 select it there. MAPRAM can only be set: writing 0 leaves it on, as on the original DivMMC. On the Next, a reset or a 1 written to NextReg `$09` bit 3 clears it.
- **Reading `$E3`** gives bits 7, 6 and 3–0, with bits 5 and 4 reading 0.

The DivMMC's other way in is the **automap** at its entry points, and the one that matters for files is `$0008`, the address `RST $08` calls. With the entry-point settings after a reset, NextRegs `$B8` to `$BA`, `$0008` maps the DivMMC **only when the 48K**

BASIC ROM is at `$0000`: ROM 3 on a +3 or Next machine type, ROM 1 on a 128K one. In the 128K editor's ROM, the +3DOS ROM or RAM, `RST $08` simply runs whatever is at `$0008` there. The mapping is also **delayed**: the 48K ROM's own opcode at `$0008` is fetched first, and the DivMMC's code takes over from the next fetch.



KLIVE IDE Is the DivMMC in

The **Next Memory Mapping** panel's `DivMMC:` row shows port `$E3`, and its `DivMMC In:` row whether the DivMMC's memory is paged in. Step into an `RST $08` with the 48K BASIC ROM at `$0000`, and keep stepping to watch the automap take over.

So "calling the operating system" means a program with the 48K BASIC ROM paged in executes `RST $08` followed by a byte that names the function. The code that answers is in the DivMMC's memory: **esxDOS**, or the part of NextZXOS loaded there that offers the same calls. It drives `$E7` and `$EB`, and it leaves on a `RETN`. The whole stack is short: your program calls `RST $08`, the DivMMC's code turns the call into SD commands, and the SPI master turns the commands into bits.



KLIVE IDE Files on the card

`ncp to "build/data.bin" "/data/"` copies a file onto the stopped Next's card, and the **Disk Explorer** opens the whole `.cim` image with its folders and long file names: the files the DivMMC's code will find there.

Tape and disks

The Next keeps the Spectrum's **tape**: the EAR input and MIC output of port `$FE` (Chapter 21), so a real cassette still loads. And the DivMMC can stand in for the cassette: four of its entry points sit inside the 48K ROM's tape routines, so a `LOAD ""` can be served from a tape image on the SD card instead.

The +3's **floppy disk** is a different case. The Next has no disk controller at all. What it has is an **I/O trap**: with bit 0 of the **I/O Traps register**, **NextReg `$D8`**, set, an access to the +3's disk ports `$2FFD` and `$3FFD` raises a Multiface NMI, and the handler can do what the disk controller would have done, from a disk image on the card (Chapter 4).

The Issue 5 board goes one step further for a real drive: with bit 3 of the **Expansion Bus Control register**, **NextReg `$81`**, set, it brings the +3's disk signals out on the expansion bus: the motor line, which bit 3 of port `$1FFD` drives, and the drive's read and write lines. On the other boards the bit is stored and reads back, but no signal leaves the board.

Resets

A reset, hard or soft, deselects every SPI device, so `$E7` behaves as if `$FF` had been written. The SPI master itself is not reset: a byte under way finishes, and the next one waits for a program. A reset clears port `$E3`, MAPRAM included, so the DivMMC is mapped out with RAM bank 0 selected. An SD card, being outside the FPGA, is not reset at all: it keeps whatever state it was in, and its driver starts it again with CMD0, as on every power-on.

In short. Port `$E7` selects one of five SPI devices and port `$EB` moves the bytes, each taking 16 T-states at every CPU speed. An access to `$EB` before the previous exchange ends is lost without warning, so two `OUT (C),r` in a row fail, while `OTIR` and `INIR` at 21 T-states are safe. A read returns the byte from the previous exchange and starts a new one that sends `$FF`. The DMA gets wait states and never loses a byte, and after a reset `RST $08` reaches the DivMMC only with the 48K BASIC ROM paged in.

CHAPTER 29

UARTs: Two Serial Ports Behind One Door

A Next can fetch a web page, chat over the Internet and talk to a Raspberry Pi, yet it has no network chip of the kind a PC has. What it has is older than the Spectrum itself: a serial port, a wire on which bits follow one another at an agreed pace. Behind that wire sits a Wi-Fi module that does the networking, and behind another one a Pi that can do almost anything. How does a Z80 keep up with a line that delivers a byte every 87 microseconds, and what happens when it doesn't?

A file from the Internet, a game played across two machines, a MIDI keyboard or a debugger on a PC all arrive as bytes on a serial line, and the UART's buffers, flags and timing decide whether your program gets every byte. This chapter builds on Chapter 4 for the UARTs' interrupts, and on Chapter 10 for the master clock and the VGA modes the baud rate is counted from.

- How the two UARTs share four ports, and what a frame of bits is
- How the prescaler sets the baud rate, and the frame register the frame's shape
- How the buffers and the status register handle waiting bytes and errors
- How break, hardware flow control and interrupts work

Two UARTs, four ports

A **UART**, a universal asynchronous receiver-transmitter, turns bytes into a stream of bits on one wire and a stream of bits on another wire back into bytes. The Next has two of them, identical in every way:

- **UART 0** is wired to the **Wi-Fi module**, an ESP chip on the board, which does the networking and talks to the Next through text commands on this line.
- **UART 1** is wired to the **Raspberry Pi's connector**, on GPIO 14 and 15, once bit 5 of the **PI Peripheral Enable register**, **NextReg `$A0`**, gives those pins to the UART.

The two share four ports, and a switch that says which UART they reach.

- **Port $153B, UART select.** Bit 6 chooses the UART: 0 for UART 0, 1 for UART 1. Every other port talks to the chosen one. Bit 4 and bits 2–0 are part of the baud rate, as the next sections explain.
- **Port $163B, UART frame**: the shape of the frame, and three switches.
- **Port $133B, UART TX**: a write sends a byte; a read gives the **status**.
- **Port $143B, UART RX**: a read takes a received byte; a write sets part of the baud rate.

Bit 4 of NextReg $83 switches all four ports off: they then read $FF and ignore writes.



KLIVE IDE An empty line

Klive runs both UARTs frame by frame on the master clock, but nothing sits at the other end: no Wi-Fi module and no Pi. An AT command goes out and no answer comes back. Sending, the status bits and the timing behave as on the machine; receiving needs a real Next.

A frame on the wire

A serial line has no clock wire. The two ends agree on a speed, the **baud rate**, in bits a second, and the receiver times the bits itself. A line at rest is high, at 1. A byte travels as a **frame** (Figure 29.1).



Figure 29.1: The letter A, $41, in an 8N1 frame, to scale in bit periods.

1. The **start bit**, a 0, ends the rest and tells the receiver that a frame begins.
2. The **data bits**, 5 to 8 of them, **least significant first**.
3. A **parity bit**, if the two ends agreed on one: it makes the number of 1s even or odd, so that one flipped bit is noticed.
4. One or two **stop bits**, 1s, before the next start bit can come.

The receiver's part is the clever one. It sees the falling edge of the start bit, waits half a bit, checks that the line is still low, and then takes each bit one bit period later, in its middle, as far as possible from the edges where the level changes. The Next's receiver does better still: every time the line changes level, it moves its sampling moment back to the middle of the bit, so a sender whose clock is a little fast or slow is

followed through the whole frame. It also ignores pulses shorter than 4 master clocks, 143 ns, which are noise, not bits.

With 8 data bits, no parity and one stop bit, written **8N1**, a frame is 10 bits, so 115,200 baud carries 11,520 bytes a second, one every 86.8 µs. That is the answer to the opening puzzle: at 3.5 MHz the CPU has about 300 T-states per byte, which is plenty for a loop, and nothing like enough if the loop has to wait for the screen or a disk.



KLIVE IDE Timing a frame

Put a breakpoint before a write to `$133B` and one after a loop that polls status bit 4 until the line is quiet. At 3.5 MHz the Z80 CPU panel's `TSP` shows a little over 300 T-states: 10 bits of 243 master clocks each, 86.8 µs, plus the loop's last poll.

The baud rate

Each UART times its bits with a **prescaler**: a bit lasts exactly the prescaler's number of master clocks, so the baud rate is 28,000,000 ÷ prescaler. The prescaler has 17 bits, written in three pieces:

```
port $143B, bit 7 = 0:  bits 6-0 -> prescaler bits 6-0
port $143B, bit 7 = 1:  bits 6-0 -> prescaler bits 13-7
port $153B, bit 4 = 1:  bits 2-0 -> prescaler bits 16-14
```

The `$143B` writes go to the UART that is selected. The `$153B` write is a little different: its bits 2–0 go to the UART that **bit 6 of the same byte** selects, so one write can choose a UART and set its top bits together. A read of `$153B` gives bit 6, which UART is selected, and that UART's bits 16–14; the lower 14 bits cannot be read back.

The prescaler is a whole number, so the rate is rarely exact. Here are some common ones at 28 MHz:

- **9,600 baud**: prescaler 2,917, 9,599 baud.
- **31,250 baud**, MIDI's rate: prescaler 896, exact.
- **115,200 baud**: prescaler **243**, 115,226 baud, 0.02% fast. This is the value after power-on.
- **230,400 baud**: prescaler 122, 229,508 baud, 0.4% slow.
- **921,600 baud**: prescaler 30, 933,333 baud, 1.3% fast.
- **2,000,000 baud**: prescaler 14, exact.

The largest prescaler, 131,071, gives about 214 baud. A UART at the other end is happy with an error of a couple of per cent. The prescaler counts the **master clock**, not the CPU's, so the baud rate does not change with the CPU speed; but it does change with the VGA mode, which runs the master clock faster, so a program that cares reads NextReg `$11` and computes its prescaler from that mode's clock. And both ends of a

frame take the prescaler when the frame starts, so a new one applies from the next frame, never to the middle of one.

Here is UART 0 set to 230,400 baud:

```
        ld bc,$153b
        ld a,%00010000          ; UART 0, bits 16-14 follow:
        out (c),a               ; 000
        ld b,$14                ; $143B
        ld a,122                ; bit 7 = 0: bits 6-0
        out (c),a
        ld a,%10000000          ; bit 7 = 1: bits 13-7 = 0
        out (c),a
```

The frame register

Port `$163B` holds the frame's shape for the selected UART, and three switches. It reads back all eight bits:

```
$163B:  R B F N N P O S
  R   = 1: reset the UART and hold it while the bit is set
  B   = 1: send a break once the transmitter is idle
  F   = 1: hardware flow control
  N N = data bits: 11 8, 10 7, 01 6, 00 5
  P   = 1: a parity bit
  O   = parity: 0 even, 1 odd
  S   = stop bits: 0 one, 1 two
```

The value after power-on is `$18`: 8 data bits, no parity, one stop bit, everything else off.

Bit 7 deserves a warning, because it does more than its name suggests. While it is 1, both buffers are empty and **both state machines are held in reset**: a byte being sent stops in the middle, nothing is received and nothing can be sent. It is not a pulse. Write the register with bit 7 set to clear the UART, then write it again with bit 7 clear, or the UART stays dead. Write `$98`, then `$18`.

Two buffers and the status

Between the CPU and each wire sits a buffer, a **FIFO**, first in, first out:

- **The transmit buffer holds 64 bytes.** A write to `$133B` puts a byte in it, and the transmitter takes them out one by one, as fast as the baud rate allows. A byte written while the buffer is full is lost.
- **The receive buffer holds 512 bytes**, and one more waits behind it while it is full. A read of `$143B` takes the oldest; a read of an empty buffer gives 0, which could also be a byte, so check the status first.

The **status**, read from `$133B`, says how both buffers and the line are doing:

- **Bit 0**: the receive buffer has at least one byte.
- **Bit 1**: the transmit buffer is full. Wait before writing.
- **Bit 2**: the receive buffer **overflowed**: a byte arrived while the 512 bytes and the one waiting behind them were all there, and it was lost.
- **Bit 3**: the receive buffer is **near full**, at least 384 bytes, three quarters.
- **Bit 4**: the transmit buffer is empty **and** the transmitter has finished its last frame: the line is quiet. This is the bit to wait for before changing the baud rate or the frame.
- **Bit 5**: the byte at the head of the receive buffer arrived after an error, an overflow or a broken frame. Bytes are stored with this mark, so you can tell exactly where data went missing.
- **Bit 6**: a **framing error** (a stop bit that was 0) or a **parity error** happened.
- **Bit 7**: the line is in **break**, as the next section explains.

Bits 6 and 2 are cleared by reading the status: each read reports what happened since the last one. And here is the rule that surprises people: **a frame with an error is thrown away**. A byte whose stop bit or parity is wrong never reaches the buffer. Bit 6 tells you it existed, and bit 5 marks the next byte that made it through.

Two loops do most of the work. This one sends a text, waiting whenever the transmit buffer is full:

```
; send the zero-terminated text at HL on UART 0
Send:   ld bc,$153b
        xor a                   ; UART 0, prescaler untouched
        out (c),a
        ld b,$13                ; $133B: TX and status
Next:   ld a,(hl)
        or a
        ret z                   ; end of the text
Full:   in a,(c)                ; status
        bit 1,a                 ; TX buffer full?
        jr nz,Full
        ld a,(hl)
        out (c),a               ; into the TX buffer
        inc hl
        jr Next
```



KLIVE IDE What the program sent

With nobody on the line in Klive, a logpoint shows what goes out: `bp-set $133B -o -log "[UART] {VAL:hex8}"` logs every byte written to the transmit buffer, so a malformed command shows up before it ever reaches a real ESP.

And this one empties the receive buffer into memory at DE, as long as it has bytes:

```
; copy what the selected UART has received to (DE)
Drain:  ld bc,$133b
Loop:   in a,(c)                ; status
        rra                     ; bit 0 into carry
        ret nc                  ; the buffer is empty
        inc b                   ; $143B: RX
        in a,(c)                ; the oldest byte
        dec b
        ld (de),a
        inc de
        jr Loop
```

When receiving, drain the whole buffer every time you look, not one byte. At 115,200 baud the 512 bytes fill in 44 ms, a little over two frames, so a program that reads the UART once a frame has time to spare, and one that stops reading while it loads a level or waits for a key does not.

▶ TRY IT Serial Echo

Part IV, key 2. The demo sends `Hello, Next!` through UART 1, whose pins stay away from the Pi while NextReg `$A0` is untouched, so nobody receives it. It echoes the UART's own view: the select and frame registers read back, the frame of `H` bit by bit, the status before sending, just after and once the line is quiet, and how long the 12 frames take at 9,599, 31,250 and 115,226 baud, next to 120 × prescaler ÷ 28 µs. The code is in `examples/code/demos/4-world/serial-echo.kz80.asm`. At 3.5 MHz, with interrupts off, it counts polls of status bit 4:

```
    ld hl,SeMessage
    ld d,12
SeSend:
    ld a,(hl)
    out (c),a               ; into the transmit buffer
    inc hl
    dec d
    jp nz,SeSend            ; 39 T-states a byte
    in a,(c)
    ld (SeStatus1),a        ; just after the last byte
    ld hl,9                   ; 484 T-states since the first byte: about
nine polls
SePoll:
    in a,(c)                ; 12  the status
    bit 4,a                 ;  8  the line is quiet?
    jp nz,SeQuiet           ; 10
    inc hl                  ;  6
```

▶ TRY IT Serial Echo *(continued)*

```
    bit 3,h                 ;  8  2,048 polls: give up
    jr z,SePoll             ; 12  56 T-states, 16 us
SeQuiet:
```

A poll is 56 T-states, 16 µs, so the measured times come out a poll or so above the prescaler's: in Klive, 12,512 µs against 12,501 at 9,599 baud, and 1,056 against 1,041 at 115,226. Bit 4 drops to 0 with the first byte and comes back only after the last stop bit.

Break

A **break** is the line held low for longer than a frame: no start bit, data and stop bit can look like that, so both ends know it is a signal, not data. It is how some devices are reset or woken.

- **Sending a break**: set bit 6 of the frame register. The transmitter finishes the frame it is sending, then holds the line low and sends nothing more until the bit is cleared.
- **Receiving a break**: the frame fails its stop bit, a framing error that sets status bit 6, and when the line stays low for eight more bit periods, status bit 7 says **break** for as long as it stays low.

Hardware flow control

The receive buffer can fill, so a fast sender needs a way to be told to wait. **Hardware flow control**, bit 5 of the frame register, adds two signals beside the data lines, both active low:

- **CTS**, clear to send, is an input. The UART starts a frame only while the other end holds it low; while it is high, the transmitter takes the next byte from its buffer and waits with it in hand.
- **RTR**, ready to receive, is an output. The UART drops it when its receive buffer holds **510 bytes**, two short of full, so that a sender who stops at once loses nothing.

With flow control, a program can stop reading for as long as it likes, and no byte is lost: the sender is paused instead. For the Pi, the two signals are GPIO 16 and 17. The Wi-Fi module takes part in flow control only on issue 4 boards, so on an earlier board a program must read fast enough.

Interrupts

Each UART can interrupt the CPU (Chapter 4). From the UART's side:

- **Receive**: when the buffer gets its first byte, or, with the "near full only" bit, when it reaches 384. The enables are bits 0 and 1 of NextReg `$C6` for UART 0, bits 4 and 5 for UART 1.
- **Transmit**: when the transmit buffer becomes empty, which is a frame before the line falls silent, because the last byte is still being sent. The enables are bit 2 for UART 0 and bit 6 for UART 1.
- The receivers stand high in the daisy chain, right after the line interrupt, and the transmitters last, after the ULA: a byte arriving is urgent, a buffer to refill is not.
- **The requests are edges**: a byte arriving into a buffer that is not empty raises nothing new. That is the second reason to drain the whole buffer in the handler.



KLIVE IDE UART interrupts at a glance

`show-nextregs $C6` opens the **Next Registers** view on `$C6`, and its inspector names the six enables one by one, from "UART0 Rx available" to "UART1 Tx empty": the quickest check of why a UART handler never runs.

The joystick port

Either UART can leave its own pins and use a joystick port instead, through the **Joystick I/O Mode register**, **NextReg `$0B`**: bit 7 turns the I/O mode on, bits 5–4 = 10 choose the left port and 11 the right one, and bit 0 chooses the UART, 0 for UART 0 and 1 for UART 1. The UART then sends on pin 7 and receives on pin 9, and with flow control it takes CTS from pin 6; there is no pin for RTR. Its own lines rest high meanwhile. A cable from a joystick port to another machine is then a serial link with no expansion at all.

Resets

A **soft reset** selects UART 0, empties both buffers in both UARTs, clears their errors and stops whatever was being sent. It keeps each UART's prescaler and frame register. Only loading the core, which a hard reset through NextReg `$02` does, sets them back to 115,200 baud and `$18`. So a program that changed the baud rate leaves it changed for the next program after a soft reset, so set the prescaler and the frame yourself, even when you want the defaults.

In short. Two identical UARTs share four ports, and port $153B bit 6 chooses UART 0 for the Wi-Fi module or UART 1 for the Pi's connector. The baud rate is the master clock divided by a 17-bit prescaler, so it changes with the VGA mode but not with the CPU speed. The receive buffer holds 512 bytes and fills in 44 ms at 115,200 baud, so drain all of it each time you look; a frame with an error never reaches it. Bit 7 of port $163B holds the UART in reset until cleared: write $98, then $18.

CHAPTER 30

Wi-Fi: A Telegraph Clerk on a Serial Line

WPA2 encryption, radio packets, TCP with its timers and retransmissions, DNS lookups: a modern network connection is a lot of software, and a Z80 running at 3.5 MHz has 64K to put it in. So how does a Next fetch a web page? It doesn't. It hires somebody else to do it, and talks to him in plain text. Anything on the Next that touches the Internet goes through this one narrow door, which has rules, a speed limit, and one nasty surprise about what "reliable" means.

The previous chapter explains the serial port itself: ports, baud rate, buffers and status bits. I lean on it throughout and will not repeat it.

- Why the Wi-Fi module is a computer of its own, and how it is wired to the Next
- How AT commands join a network and hold a conversation with a server
- How to tell a server's data from the module's own chatter
- Where TCP's guarantee ends, and how fast the link can go

> The Next's master clock is **28 MHz** and the serial line runs at **115,200 baud, 8 data bits, no parity, one stop bit**. The module is not part of the Next's own design: what it answers depends on the firmware loaded into it, and the exchanges shown here are those of Espressif's standard AT firmware.

A computer for a peripheral

The Wi-Fi hardware is an **ESP-01**, a small board that pushes onto a header behind the VGA port, with no soldering. On it are two things that matter: an **ESP8266** chip from Espressif and a flash memory holding its firmware. The chip contains the radio, for 802.11 b/g/n on the 2.4 GHz band, with WPA and WPA2.

It also contains a processor, and here the story turns funny. The ESP8266's core is a 32-bit RISC running at 80 MHz, with 160 KB of RAM on the chip: the humblest peripheral in the machine has more raw processing power than the computer it is plugged into.

I think of the module as **the telegraph clerk**: you hand him a message, and he passes the answer back through the hatch without you ever touching the wires. The Next contains no network logic at all: no MAC, no packet buffers, no checksum engine. All of it runs as firmware inside the module, and the FPGA provides only a serial port and a couple of control lines.

Five wires and a socket

Here is everything that connects the Next to its module (Figure 30.1).

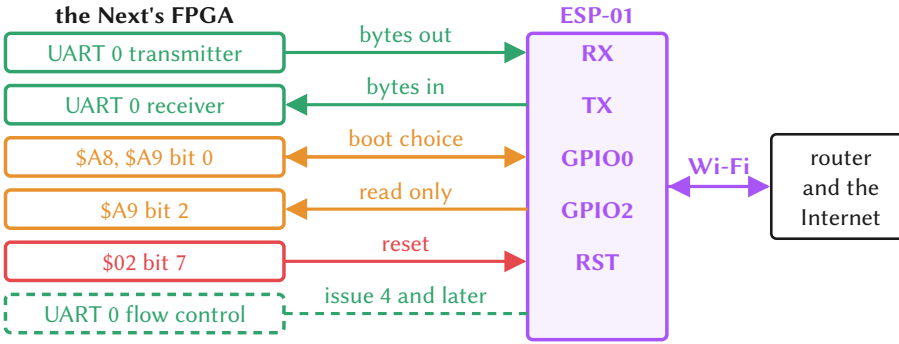


Figure 30.1: The wires between the FPGA and the module (dashed: issue 4 boards and later)

- **The serial pair.** The transmitter of **UART 0** drives the module's RX pin, and the module's TX pin drives the receiver of UART 0. Every byte of the conversation goes over these two wires.
- **GPIO0.** A pin the module looks at when it starts, and the one wire the Next can both drive and read. The **ESP Wifi GPIO Output Enable register**, **NextReg `$A8`**, bit 0 lets the Next drive it, and the **ESP Wifi GPIO register**, **NextReg `$A9`**, bit 0 is the level to drive. Reading `$A9` gives the pin's real level.
- **GPIO2.** Another start-up pin, which the Next only reads, in bit 2 of NextReg `$A9`. The FPGA never drives it, and bit 2 of NextReg `$A8` is fixed at 0.
- **Reset.** Bit 7 of the **Reset register**, **NextReg `$02`** holds the module in reset for as long as it is set.

A soft reset releases GPIO0 by clearing NextReg `$A8` and sets the level in `$A9` back to 1. The FPGA also holds weak pull-ups on both GPIO pins and on the receive line, so anything left undriven sits high, which is the level the module wants to see for a normal start.

Two details are worth remembering. **The reset line is shared with the expansion bus**, so bit 7 of NextReg `$02` also resets whatever is plugged in there; I would not touch it in a program that might run with a peripheral attached. And if the **Joystick I/O**

Mode register, **NextReg `$0B`**, moves UART 0 to a joystick port (the previous chapter has the details), the module's receive pin rests high and it hears nothing more until UART 0 comes back.

The dashed row in Figure 30.1 is hardware flow control, the two signals that let a receiver say "wait". Issue 2 boards lack them, and on later boards the eight pins of a standard ESP-01 include neither, so write your code as if nothing will ever pause the module. We will soon see why that matters.

Waking the clerk

When the ESP8266 comes out of reset, its boot ROM looks at three pins to decide what to do. Two are the ones the Next can see, and the third is held low on the module:

- **GPIO0 high and GPIO2 high**: start the firmware in flash. This is the normal case, and the pull-ups make it the default.
- **GPIO0 low and GPIO2 high**: wait for a new firmware on the serial line. This is **download mode**, and it is how a firmware update tool running on the Next rewrites the module's flash without taking him out of the machine.

Here is the whole recipe for putting the module into download mode, using the Z80N's `nextreg` instruction:

```
        nextreg $a9,%00000000  ; GPIO0 will be low
        nextreg $a8,%00000001  ; the Next drives it
        nextreg $02,%10000000  ; hold the module in reset
        ; ...wait a moment...
        nextreg $02,%00000000  ; release: GPIO0 is sampled now
        ; ...talk to the boot ROM over UART 0...
        nextreg $a8,%00000000  ; let GPIO0 float high again
```

For an ordinary restart, leave the first two lines out. Write exactly `$80` and then `$00` to NextReg `$02`: bits 0 and 1 of that register cause a soft and a hard reset of the Next itself, and you do not want one of those by accident. Bit 7 alone only reaches the module and the expansion bus.



KLIVE IDE Catching a write to $02

`bp-set nr:$02` stops before Klive carries out a write to the Reset register, with the program counter still on the instruction that wrote it, so a stray bit 0 or 1 shows up before it resets the Next. The **Next Registers** view names bit 7 and the GPIO bits of `$A8` and `$A9`.

A hard reset of the Next restarts the module too. A soft reset does so only when the expansion bus is enabled; otherwise the clerk stays at his desk, connection intact,

while the Next reboots around him, so a program may find him in the middle of a conversation that an earlier program started.

Now a small trap. In its first moments the boot ROM prints a diagnostic message on the serial line, and on the usual module it does so at **74,880 baud**, a rate that falls out of its 26 MHz crystal. The Next is listening at 115,200, so what lands in the receive buffer is a burst of nonsense. Some of those frames fail their stop bit and are thrown away; the rest arrive as random bytes. After any module reset, wait, then clear UART 0 with the `$98`, `$18` pair on the frame register, and only then start talking.

Talking in AT

The module speaks the language of Hayes's 1981 modems: text commands beginning with **AT**, for "attention". Lines of text go in, and lines of text come out.

A command is ASCII text ending in carriage return and line feed, written to UART 0 with a loop like the `Send` routine in the UART chapter. The simplest possible exchange checks that anybody is home:

```
AT
OK
```



KLIVE IDE **No clerk at the desk**

Klive emulates UART 0 but no module behind it, so nobody answers `AT` there. Test the sending side in Klive, and the conversation on a real Next.

If you see `OK`, the module is awake and both ends agree on the baud rate. If you see garbage or nothing, they probably don't. The Next powers up at 115,200, which is the speed most modules use, but some leave the factory at 57,600 or 19,200. The Next's `.UART` terminal has a `-f` option that hunts for the right rate. Once found, the command `AT+UART_DEF=115200,8,1,0,0` (plain `AT+UART=` on older firmware) stores the speed in the module's flash, so you only do this once.

By default the module also **echoes** everything you send, which a human at a terminal likes and a program does not. `ATE0` switches echo off, and it is the first command I would send.

▶ TRY IT AT Probe

Part IV, key 3. The demo sets UART 0 to 115,200 baud and 8N1, empties it with `$98`, `$18`, then sends `AT` and `AT+GMR`, each with CR and LF, and shows whatever arrives in the next half second: on a Next, the echo, `OK`, and the firmware's version lines. It sends no other command, so it changes no setting. The code is in `examples/code/demos/4-world/at-probe.kz80.asm`. It drains the receive buffer for 25 frames, counted with the shell's frame counter:

▶ TRY IT AT Probe *(continued)*

```
AtListenLoop:
    ld bc,$133B
    in a,(c)                ; status
    rra                     ; bit 0 into carry: a byte is waiting
    jr nc,AtListenTime
    inc b                   ; $143B: RX
    in a,(c)                ; the oldest byte
    ld (hl),a
    push hl
    ld bc,AtBufferEnd
    or a
    sbc hl,bc
    pop hl
    jr nc,AtListenLoop      ; the buffer is full: keep reading, keep the
last byte
    inc hl
    jr AtListenLoop
```

In Klive both commands end with "(no answer in half a second)": the bytes go out, and nobody is at the desk.

Joining a network

Getting on to Wi-Fi takes three commands, and you normally type them only once:

```
AT+CWMODE=1
OK
AT+CWJAP="MyNetwork","MyPassword"
WIFI CONNECTED
WIFI GOT IP

OK
AT+CIFSR
+CIFSR:STAIP,"192.168.1.42"
+CIFSR:STAMAC,"5c:cf:7f:12:34:56"

OK
```

`AT+CWMODE=1` makes the module a **station**, a client of an access point, rather than an access point of its own. `AT+CWJAP` joins the network and is slow, because this is where the radio, the password handshake and the address request all happen. `AT+CIFSR` shows the address the router gave it.

The module **remembers**: the network name and password stay in its flash, and after a restart it rejoins unasked, printing `WIFI CONNECTED` and `WIFI GOT IP`. So the Next's `WIFI.BAS` set-up program runs once. The darker side is that your password lives on that little board, so wipe it with `AT+CWQAP` before you sell the module on.

A conversation with a server

Let's fetch a page from a small web server on the local network, at address 192.168.1.20 and port 8080. It is the kind of thing every network program on the Next does, and three parties take part: your program, the module and the server.

Here is the serial traffic, in order. Lines beginning with `>>` are what your program sends and the others are the module's answers:

```
>> AT+CIPSTART="TCP","192.168.1.20",8080
CONNECT

OK
>> AT+CIPSEND=38
OK
>
>> GET / HTTP/1.0<CR><LF>Host: 192.168.1.20<CR><LF><CR><LF>
SEND OK

+IPD,31:HTTP/1.0 200 OK<CR><LF><CR><LF>Hello, Next!
CLOSED
```

Walk through it with me:

1. `AT+CIPSTART` asks the module to open a connection, and he handles the whole TCP handshake himself. `TCP` could have been `UDP` or `SSL`, and a host name works as well as an address, because the module does the DNS lookup too.
2. `AT+CIPSEND=38` announces exactly how many bytes are coming. The module answers `OK` and then a `>` prompt. Wait for the prompt: bytes sent before it are not part of the message.
3. Now the 38 bytes go out, with no line ending added; the count is the only thing that says where the message stops. Send too many and the module answers `busy` to the excess; send too few and it keeps waiting, taking your next command as part of the data.
4. `SEND OK` says the bytes have left the module. It does not say the server has read them.
5. The reply arrives unasked, after `+IPD,31:`, "here come 31 bytes from the network". When the server closes the connection, `CLOSED` follows the last byte.

Network data can arrive at any moment, even while you are sending. One `AT+CIPSEND` carries at most 2,048 bytes, so longer messages go out in several rounds.

Telling data from chatter

The module uses one serial line for two kinds of thing: its own remarks, such as `OK` or `CLOSED`, and raw data from the network, which can contain absolutely anything. It keeps them apart with a tiny piece of framing (Figure 30.2).

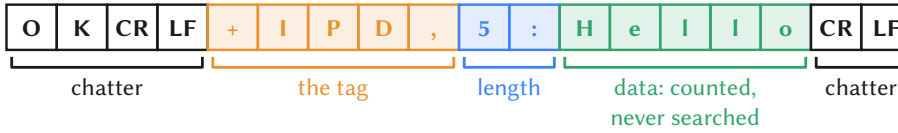


Figure 30.2: The serial line, one cell per byte

The rule that makes it work is that **the data is counted, never searched**. A web page that happens to contain the text `+IPD,` or `CLOSED` is fine, because once the length is known, the next that many bytes are data regardless of what they say.

So a reader needs just three steps: look for the tag, read a number, copy that many bytes. Here it is in Z80N:

```
; read one +IPD packet from UART 0 to (DE); HL = its length
GetPacket:
        ld hl,Tag
Match:  call GetByte
        cp (hl)
        jr z,Hit
        ld hl,Tag               ; no: start over, but this byte
        cp (hl)                 ; may be a new "+"
        jr nz,Match
Hit:    inc hl
        ld a,(hl)
        or a
        jr nz,Match             ; more of the tag to find
        ld hl,0                 ; now the length, in decimal
Digit:  call GetByte
        cp ':'
        jr z,Body
        sub '0'
        add hl,hl               ; HL * 2
        ld b,h
        ld c,l
        add hl,hl               ; * 4
        add hl,hl               ; * 8
        add hl,bc               ; * 10
```

```
        add hl,a                ; Z80N: plus the digit
        jr Digit
Body:   push hl                 ; keep the length
Copy:   ld a,h
        or l
        jr z,Done
        call GetByte
        ld (de),a
        inc de
        dec hl
        jr Copy
Done:   pop hl
        ret

GetByte:
        ld bc,$133b
Wait:   in a,(c)                ; status
        rra
        jr nc,Wait              ; nothing yet
        inc b                   ; $143B: RX
        in a,(c)                ; the oldest byte
        ret

Tag:    .defb "+IPD,",0
```

How it works:

- **The matching loop** walks `HL` along the tag. On a mismatch it goes back to the start of the tag, but it first checks whether the byte that failed is itself a `+`. Without that check, the text `++IPD,5:` would slip past.
- **The digits** build the length with the usual multiply-by-ten, in which `A` survives because `add hl` leaves it alone. `add hl,a` is a Z80N instruction; on a plain Z80 you would load `BC` and add that.



KLIVE IDE Testing the reader without a module

Klive's unit tests run code on a machine of their own and report each test as passed or failed. Link `GetPacket` against a `GetByte` of the test's own that reads a table such as `"OK",13,10,"++IPD,5:Hello"`, and check that it returns the five bytes: the `++IPD` case, tested with no module and no network.

This is the honest minimum, and I would not ship it as it stands: `GetByte` waits forever, so a connection that dies in mid-packet hangs the program. A real reader counts down a timeout in `Wait` and gives up.

The promise that stops at the module

TCP is the reliable protocol: every byte is numbered and acknowledged, and a lost packet is sent again. So surely your program receives every byte? It doesn't, because of where TCP ends (Figure 30.3).

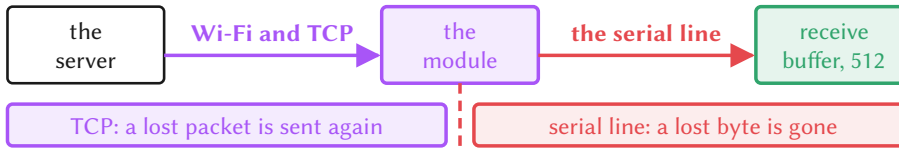


Figure 30.3: Where TCP's promise ends

The module acknowledges a packet to the server the moment **it** has it. From the server's point of view the data has been delivered, and it will never be sent again. Then the module pushes those bytes down the serial line at full speed, and in the usual arrangement it neither knows nor cares whether anybody is reading.

Now do the arithmetic. A TCP packet carries up to 1,460 bytes, which takes 127 ms to cross at 115,200 baud, and the Next's receive buffer holds 512, filled in 44 ms. If your program goes away for longer than that while a packet is arriving, perhaps to write a sector to the SD card, the buffer overflows and bytes are lost. TCP will not save you, because as far as TCP is concerned, everything went perfectly.

That leaves three defences, best first:

- **Never stop reading during a packet.** Read into memory, and do the slow work between packets. Status bit 2, the overflow flag, tells you it went wrong.
- **Add your own check.** If the data matters, put a checksum or CRC in the protocol above TCP and ask again when it fails. NextSync, the popular file transfer tool for the Next, puts checksums on every packet for exactly this reason.
- **Ask for the data yourself.** Newer firmware has a **passive receive mode**, `AT+CIPRECVMODE=1`. In this mode the module holds what arrives, tells you how much is waiting, and hands over a portion only when you request it with `AT+CIPRECVDATA`. As its own buffer fills, TCP's flow control slows the server down: the only true end-to-end flow control available here. Check with `AT+GMR` whether yours has it.

How fast it can go

At 115,200 baud a byte takes 86.8 µs and the line carries 11,520 bytes a second, which the framing barely dents: on a full packet the `+IPD,1460:` header is under one per cent. A 200K file takes about 18 seconds at best.

You can go faster, in three moves that must happen in this order:

1. Tell the module the new speed with `AT+UART_CUR=921600,8,1,0,0`. `CUR` lasts only until the next restart, which is safer than `AT+UART_DEF`: a module stuck at an unusual saved speed is a pain to recover.
2. Wait for its `OK`, which still comes at the old speed.
3. Program the Next's prescaler for the new rate, here 30, which gives 933,333 baud (see the UART chapter's section on the baud rate). Do this promptly, or you'll miss the start of whatever it sends next.

The next limit is your code. Add up the instructions in the reader above and each data byte costs 122 T-states when the buffer is not empty. So the reader manages about 28,700 bytes a second at 3.5 MHz, and 229,500 at 28 MHz, comfortable at any CPU speed for 115,200 baud. Faster lines are another matter:

- At **921,600 baud** (really 933,333), 93,333 bytes a second, one every 10.7 µs: the reader needs a 14 MHz CPU.
- At **2,000,000 baud**, 200,000 bytes a second, one every 5 µs: it needs 28 MHz and nothing else happening, and the receive buffer fills in just 2.6 ms.



KLIVE IDE Counting the reader's T-states

Klive's unit tests report each test's T-states. Run the previous note's test with 5 and with 105 data bytes: the difference divided by 100 is one byte's cost with the stub. Take off what the stub costs, call included, add the real `GetByte`'s 76, and you should get the 122 above.

NextSync reports a best of about 48 KB/s for a single large file, four times what 115,200 baud carries.

Firmware is the fine print

Everything above the serial line belongs to the firmware, and not all modules run the same one. Versions 1.x are the commonest, and the details differ:

- Older firmware answers `AT+CIPSTART` with `OK` and `Linked` rather than `CONNECT` and `OK`.
- Passive receive does not exist in the older versions, and the speed command has changed its name over time: `AT+UART` on some firmware, `AT+UART_CUR` and `AT+UART_DEF` on others.
- `AT+CIPMUX=1` allows several connections at once, and it changes the tag to `+IPD,<id>,<length>:`.

My rule of thumb: match on the shortest thing that is stable, such as `OK`, `ERROR` and `+IPD,`, and treat every other line as decoration.

The Next's distribution ships several layers built on all this, so you often need not write any of it. `UART.DRV` is an interrupt-driven driver with a 16K buffer, which solves the overflow problem for programs that use it. The ESPAT driver serves BASIC programs, and the `.http` dot command fetches a file.

In short. The ESP8266 on UART 0 does all the networking and takes text commands, and a server's data arrives after `+IPD,` and a length, so it must be counted, never searched. TCP's guarantee stops at the module, which acknowledges a packet of up to 1,460 bytes the moment it has it, then sends it down the line whether or not anybody reads. The 512-byte receive buffer fills in 44 ms, so never stop reading during a packet, and add your own checksum if the data matters. After a module reset, wait, then clear UART 0 with `$98`, `$18`.

CHAPTER 31

Real-Time Clock: A Watchman on Two Wires

Switch your Next off, put it in a drawer for a month, and switch it on again: it knows the date. The FPGA forgot everything the moment the power went, the SRAM is blank, the core has been loaded from flash all over again. So who was counting the seconds in the dark? The answer is the Next's **Real-Time Clock**, the **RTC**: a clock that keeps the time and the date whether the machine is running or not.

File dates, a puzzle of the day and a random seed all need a clock that survives the power switch, and the Spectrum never had one.

Chapter 3 explains how the Next decodes its ports and how a NextReg switches a group of them off. Chapter 28 shows a serial bus that the hardware clocks for you; here you do the clocking yourself.

- Where the clock chip lives, and how two wires connect it
- What the rules of the conversation on those wires are
- How a program speaks those rules itself, and the speed limit it must respect
- How to read and set the time, and use the chip's spare memory

A clock outside the machine

The Next's clock is a small chip of its own, a **DS1307**, sitting on the board beside the FPGA. I like to call it **the watchman**: it stays awake on its own lamp while the house sleeps, and when you come down in the morning it tells you what time it is. It is not part of the core and no reset of the FPGA reaches it.

Three things keep it going:

- **A 32,768 Hz crystal.** The number is 2^{15}, so fifteen halvings make exactly one pulse a second. One 20 parts in a million off gains or loses about 1.7 seconds a day, so expect to set the clock now and then.
- **A battery input** for a 3 V lithium cell, between 2.0 and 3.5 V. When the board's own supply falls below the battery's voltage, the chip switches over and keeps

counting on less than half a microamp. A 48 mAh cell keeps it alive for more than ten years.

- **A guard at the door.** While the supply is below 1.25 times the battery's voltage, the chip ignores the bus altogether, so a machine that is powering up or down cannot scribble garbage over the time.



KLIVE IDE The watchman in Klive

Klive emulates the DS1307 and keeps it like the real one: it shows the real time, and a time a program sets stays set when Klive starts again. The options are in **Machine › Configure ZX Spectrum Next...**, in the **Real-time clock** group.

Two wires, and nobody pushes

The clock talks over **I2C**, the inter-integrated circuit bus: two wires that many chips can share. One wire, **SCL**, is the clock, which says *when* a bit is valid. The other, **SDA**, is the data, which says *what* the bit is.

The trick that makes sharing possible is that no chip ever drives a wire high. Each wire has a **pull-up**, a resistor to the supply, that holds it high when it is left alone. A chip can only do two things to a wire: pull it low, or let go. This is called **open drain**, and it means two chips can never fight, one pushing up while the other pulls down. If anybody pulls, the wire is low; it is high only when nobody pulls. Figure 31.1 shows the Next's I2C bus.

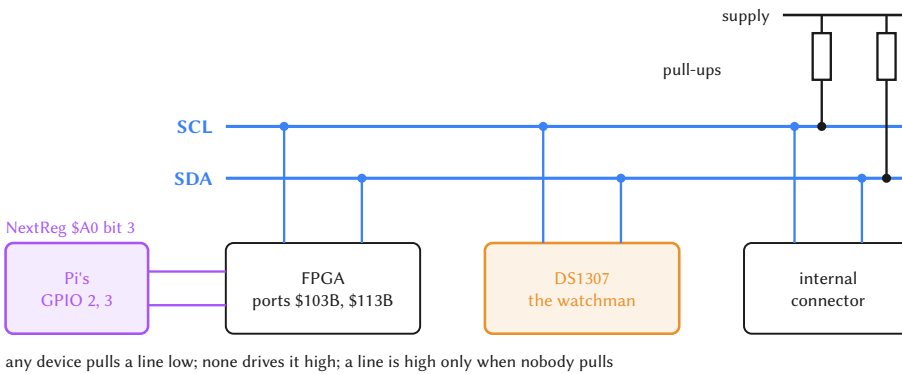


Figure 31.1: The Next's I2C bus

The FPGA, the DS1307 and an internal connector share the bus. Bit 3 of the **PI Peripheral Enable register, NextReg `$A0`**, joins the Pi's GPIO 2 (data) and GPIO 3 (clock)

too, but not by wire: the FPGA pulls the Pi's pins whenever it pulls its own, and the lines your program reads are the AND of both sides.

There is one more rule: **the Next is always the master**. It makes the clock, it starts every conversation and it ends it. The clock only answers.

Two ports, one bit each

Each wire has its own port, and only bit 0 of each counts:

- **Port `$103B`** is SCL, the clock.
- **Port `$113B`** is SDA, the data.

A write of bit 0 = 0 makes the FPGA pull the wire low; bit 0 = 1 lets it go. A read gives `$FE` or `$FF`: bits 7–1 are always 1, and bit 0 is **the wire itself**, not what you wrote. That difference is the whole point. Write 1 to `$113B` and read it back, and you read 0 when the clock is pulling the wire low: that is how you hear it. The same goes for the clock wire, so a program can see if another chip holds SCL low to ask for more time.

A reset, soft or hard, lets go of both wires. Bit 2 of the **Internal Port Decoding Enables register, NextReg `$83`**, switches both ports off: they then read `$FF` and ignore writes.

The rules of the conversation

Since both wires rest high and anyone may pull, the chips need rules about who touches what, and when. There is one simple rule and two exceptions.

The rule: SDA changes only while SCL is low. The master puts a bit on SDA, raises SCL, and the listener reads SDA while SCL is high. Then the master lowers SCL and may change SDA for the next bit.

The exceptions are the brackets around a message, and they break the rule on purpose:

- **START**: SDA falls while SCL is high. Every chip on the bus wakes up and listens.
- **STOP**: SDA rises while SCL is high. The message is over and the bus is free.

Because a normal bit never changes SDA while SCL is high, nobody can mistake data for a bracket. A START inside a message, before any STOP, is a **repeated START**: it begins a new message without letting anybody else grab the bus in between.

Bytes go **most significant bit first**, and each byte gets a ninth clock pulse, the **acknowledge**. For that pulse the sender lets go of SDA and the receiver answers: pulled low is an **ACK**, "got it"; left high is a **NACK**, "no", or "no more". The first byte after a START is the **address**: seven bits saying which chip the master wants, and an eighth saying the direction, 0 for write and 1 for read. The watchman's address is `$68`, so

`$D0` means "watchman, listen" and `$D1` means "watchman, talk". Figure 31.2 shows START, the address byte `$D0` and STOP.

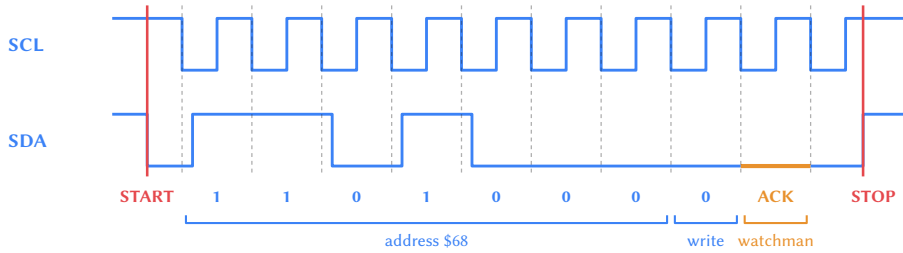


Figure 31.2: START, the address byte $D0 and STOP

A chip that is not addressed answers nothing, and nothing means a high SDA on the ninth pulse, a NACK. So the address byte doubles as a roll call: if nobody acknowledges `$D0`, there is no clock on your bus.

Speaking the protocol yourself

The Next has no hardware that sends an I2C byte for you. Your program raises and lowers the two wires one `OUT` at a time, a technique called **bit-banging**. It sounds primitive, and it is, but it takes a page of code, and here it is. The routines use BC, DE, A and the flags, and keep HL.

START and STOP are three or four writes each. Both begin with the wires in a known state:

```
; START (also a repeated START): SDA, SCL high,
; SDA low while SCL is high, then SCL low
I2cStart:
        ld bc,$113b             ; SDA
        ld a,1
        out (c),a               ; let SDA go
        ld b,$10                ; SCL
        out (c),a               ; let SCL go
        ld b,$11
        xor a
        out (c),a               ; SDA low: START
        ld b,$10
        out (c),a               ; SCL low
        ret
```

```
; STOP: SDA low, SCL high, then SDA high
I2cStop:
        ld bc,$113b
        xor a
        out (c),a               ; SDA low
        ld b,$10
        inc a
        out (c),a               ; let SCL go
        ld b,$11
        out (c),a               ; SDA high: STOP
        ret
```

Sending a byte is eight bits and a listen:

```
; send A, MSB first; A = the ACK bit (0 = ACK)
I2cWrite:
        ld e,a
        ld d,8                  ; eight bits
WrBit:  ld bc,$113b
        xor a
        rl e                    ; next bit into carry
        rla                     ; and into bit 0
        out (c),a               ; SDA = the bit
        ld b,$10
        ld a,1
        out (c),a               ; SCL high: read me
        xor a
        out (c),a               ; SCL low
        dec d
        jr nz,WrBit
        ld bc,$113b
        ld a,1
        out (c),a               ; let SDA go
        ld b,$10
        out (c),a               ; the ninth pulse
        ld b,$11
        in a,(c)                ; the answer
        and 1
        ld e,a
        ld b,$10
        xor a
        out (c),a               ; SCL low
        ld a,e
        ret
```

And receiving one is the mirror image: you let go of SDA, clock eight times and read the wire at every pulse, then answer yourself.

```
; receive a byte into A; answer NACK if carry is
; set, ACK otherwise
I2cRead:
        push af                 ; keep the carry
        ld bc,$113b
        ld a,1
        out (c),a               ; let SDA go
        ld d,8
        ld e,0
RdBit:  ld b,$10
        ld a,1
        out (c),a               ; SCL high
        ld b,$11
        in a,(c)                ; read SDA
        rra                     ; bit 0 into carry
        rl e                    ; and into E
        ld b,$10
        xor a
        out (c),a               ; SCL low
        dec d
        jr nz,RdBit
        pop af
        ld a,0
        rla                     ; A = carry: 0 ACK, 1 NACK
        ld b,$11
        out (c),a               ; our answer on SDA
        ld b,$10
        ld a,1
        out (c),a               ; the ninth pulse
        xor a
        out (c),a
        ld b,$11
        inc a
        out (c),a               ; let SDA go
        ld a,e
        ret
```

How it works, the parts worth a second look:

- **For the acknowledge, the sender lets go.** `I2cWrite` writes 1 to SDA before the ninth pulse, which does not make the wire high: it only stops pulling. The `IN` then reads whatever the clock does.
- **`I2cRead` writes 1 to SDA once, at the start**, and never touches it during the eight bits. The clock pulls the wire for every 0 it sends; the 1s are the pull-up's work.

- **The carry chooses the answer.** Clear for ACK, "go on"; set for NACK, "that was the last one". The clock stops talking after a NACK and leaves SDA to you for the STOP.

How fast is too fast

The DS1307 is a 100 kHz chip: it runs I2C's standard mode only, never the faster ones. For every pulse it asks for at least **4.0 µs with SCL high** and **4.7 µs with SCL low**. A START needs SCL high for 4.7 µs before SDA falls and 4.0 µs after; a STOP needs 4.7 µs of SCL high before SDA rises; and between a STOP and the next START the bus must rest for 4.7 µs.

Now look at `I2cWrite` at 3.5 MHz. A bit takes 96 T-states, 27.4 µs, which is about 36,500 bits a second, well under the limit. The tight spot is the clock's high time: from the `OUT` that raises SCL to the one that lowers it there is an `XOR A` and an `OUT`, 16 T-states, **4.6 µs**. That clears the 4.0 µs, but not by much. The ACK pulse of `I2cRead` is as short; no gap in these routines is shorter.

Here is the catch: the same 16 T-states last 2.3 µs at 7 MHz and **0.57 µs at 28 MHz**, seven times too short. A Next running at full speed outruns the clock with the very same code. My advice is the boring one: switch the CPU to 3.5 MHz with the **CPU Speed register, NextReg `$07`**, before you talk to the bus, and switch it back afterwards. A read of the whole time takes under 3 ms, so you lose almost nothing. If you would rather keep the speed, keep SCL high for at least 112 T-states and low for at least 132 at 28 MHz, and measure what you wrote.



KLIVE IDE Where Klive is kinder

Klive's I2C bus reacts to the edges of SCL and SDA only, and does not check the DS1307's 4.0 and 4.7 µs. So these routines work in Klive at 28 MHz, where a real clock would miss bits: test the timing on a real Next, or slow the CPU.

The watchman's registers

The chip has 64 bytes, numbered `$00` to `$3F`. The first eight hold the time and a control byte; the other **56 bytes are RAM**, kept by the battery with the time.

The time is in **BCD**, binary-coded decimal: each 4-bit half of a byte holds one decimal digit, so 45 seconds is `$45`, not `$2D`. It is wasteful, but a program that prints the time only has to print two hex digits. The spare bits carry switches:

```
reg   7    6    5    4    3    2    1    0
$00   CH  [ tens of sec ][ seconds          ]
$01   0   [ tens of min ][ minutes          ]
$02   0   12   PM/20 10h [ hours            ]
$03   0    0    0    0    0  [ day 1-7      ]
```

```
$04   0    0  [ tens   ][ date            ]
$05   0    0    0   10m  [ month            ]
$06  [ tens of year     ][ year             ]
$07   OUT  0    0   SQWE  0    0   RS1  RS0
```

- **CH, clock halt, bit 7 of the seconds.** When it is 1 the crystal stops and so does the time. A fresh chip comes up with CH set, so the first thing any program that sets the clock does is clear it.
- **The hours, bit 6, choose 12- or 24-hour mode.** With bit 6 = 0, bit 5 is the second tens digit, for 20 to 23. With bit 6 = 1, bit 5 is **PM**, and the hours count 12, 1, 2 to 11; PM flips when 11 becomes 12, and the date moves on when 11 PM becomes 12 AM. After changing bit 6, write the hours again: the chip does not convert them.
- **The day of the week, 1 to 7**, steps at midnight; which day is 1 is the software's choice.
- **The date, month and year** roll over as a calendar should: 30 or 31 days, February 28 or 29, and the year from 99 to 00. The year is two digits, and every year divisible by four is a leap year, which is right for every year from 2000 to 2099. Somebody in 2100 will have to fix it.
- **The control register** drives the chip's square-wave pin: SQWE = 1 makes a square wave of 1 Hz, 4,096 Hz, 8,192 Hz or 32,768 Hz by RS1–RS0 (00 to 11); with SQWE = 0, the pin follows OUT. On the Next the clock reaches the FPGA only through the two bus wires, so that pin cannot interrupt the Z80.

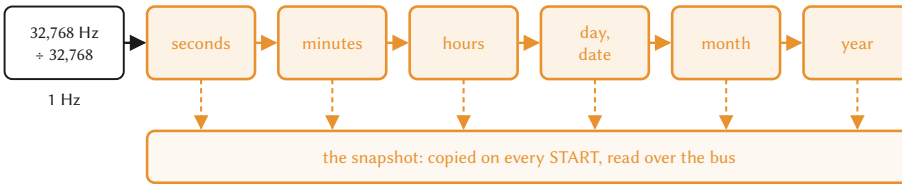
Reading the time

Reading a clock has a classic trap. Suppose it is 12:59:59 and you read the seconds, then the minutes, then the hours, and between the minutes and the hours the clock ticks: you get 13:59:59, an hour out. Every programmer of an old PC clock has met this bug.

The watchman avoids it with a **snapshot**. Its counters run on their own, and on every START it copies them into a second set of registers. Everything you read over the bus comes from the snapshot, so all seven bytes belong to the same moment, however slowly you read them, while the real counters keep running underneath. Figure 31.3 shows what happens inside the watchman.

A read is two messages joined by a repeated START. The first, addressed to `$D0`, writes one byte: the **register pointer**, the number of the register to start from. The second, addressed to `$D1`, reads: the clock sends the register at the pointer, steps the pointer, and goes on for as long as you acknowledge. Your NACK on the last byte tells it to stop.

Here it is as code, on the routines above. It reads the seconds to the year into `Time`, in the registers' order:



the counters keep running while a read takes its bytes from the snapshot

Figure 31.3: Inside the watchman

```
; read registers $00-$06 into Time (7 bytes)
ReadTime:
        call I2cStart
        ld a,$d0                ; watchman, listen
        call I2cWrite
        xor a                   ; the pointer: $00
        call I2cWrite
        call I2cStart           ; repeated START
        ld a,$d1                ; watchman, talk
        call I2cWrite
        ld hl,Time
        ld b,6                  ; six bytes with ACK
RdNext: push bc
        or a                    ; carry clear: ACK
        call I2cRead
        ld (hl),a
        inc hl
        pop bc
        djnz RdNext
        scf                     ; carry set: NACK
        call I2cRead            ; the year
        ld (hl),a
        jp I2cStop
```

At 3.5 MHz the whole read takes 10,058 T-states, 2.9 ms, about a seventh of a frame. On Friday, 18 September 2026 at 12:34:56, in 24-hour mode with Sunday as day 1, `Time` holds:

```
$56 $34 $12 $06 $18 $09 $26
sec min hr  day dt  mon yr
```



KLIVE IDE A clock that waits for the debugger

Set **The clock follows** to **Emulated time**, and the clock stops while the debugger pauses the machine. Stepping through `ReadTime` then reads the same seconds every run, and the Memory view at `Time` shows the snapshot.

▶ TRY IT Clock

Part IV, key 4. The demo reads registers `$00`–`$06` five times a second with this chapter's routines at 3.5 MHz, and shows them raw, then as a time, a date and a day of the week, with the CH and 12/24-hour bits, until a key is pressed. It writes only the register pointer, so the clock and its RAM stay as they were. The code is in `examples/code/demos/4-world/clock.kz80.asm`. Its `ClReadTime` is `ReadTime` with the roll call added:

```
ClReadTime:
    call ClStart
    ld a,$D0                ; watchman, listen
    call ClWrite
    or a
    jr nz,ClNoAck           ; SDA stayed high: nobody is there
    xor a                   ; the pointer: $00
    call ClWrite
    call ClStart            ; repeated START
    ld a,$D1                ; watchman, talk
    call ClWrite
```

The rest is `ReadTime` as above. If no chip acknowledges `$D0`, the demo says so instead. Watch the Reads counter and the seconds: about five reads go by for every tick.

Setting the clock

Setting is one message: START, `$D0`, the pointer, and then the bytes, each written to the register at the pointer before the pointer steps. With `Time` holding the seven bytes in the same order:

```
; write Time (7 bytes) into registers $00-$06
SetTime:
        call I2cStart
        ld a,$d0                ; watchman, listen
        call I2cWrite
        xor a                   ; from register $00
        call I2cWrite
        ld hl,Time
        ld b,7
WrNext: push bc
        ld a,(hl)
        call I2cWrite
        inc hl
        pop bc
        djnz WrNext
        jp I2cStop
```

It takes 8,711 T-states, 2.5 ms. Two details make this the right way to do it:

- **Writing the seconds restarts the second.** The divider that halves the crystal fifteen times is cleared, so the next tick comes a whole second after the write. Set the clock at 12:34:56 and it says 12:34:57 exactly one second later, not at some random moment within the second.
- **Write everything in one message.** A tick between your seconds and your hours could carry into minutes you have not written yet; one message from register `$00` takes 2.5 ms. The bytes are stored as each one is acknowledged, so a message broken off halfway keeps what it wrote.

And remember CH: the seconds byte you write has bit 7 clear, so the clock runs. Write `$80` or more and it stops, which is also how you would freeze the time on purpose.



KLIVE IDE Undoing an experiment

A program that sets CH keeps Klive's clock stopped, even between sessions: Klive does not move a stopped clock. **Set to host time**, in the same settings group, brings back the real time, running.

Fifty-six bytes that remember

Registers `$08` to `$3F` are plain RAM, 56 bytes, read and written like the time and kept by the same battery. It is not much, but it is the only memory in the Next that survives the power switch without a file system: a high score, a setting, the date a program last ran, with a byte or two as a signature.

The pointer has two habits worth knowing:

- **It wraps from `$3F` to `$00`.** Read four bytes from `$3E` and you get `$3E`, `$3F`, then the seconds and the minutes.
- **It stays where it was between messages.** A read that starts with `$D1` straight after the START, with no pointer written first, continues from the last pointer: a **current address read**.



KLIVE IDE RAM that outlives Klive

Klive keeps the 56 bytes for each Next model in `zxnext-rtc.json`, so they survive closing it, as long as **Keep the time and the clock's RAM between sessions** is on, as it is by default. **Forget** clears them and resets the time, like a new battery.

Resets

No Next reset touches the clock. A soft reset, a hard reset, loading a new core: the time keeps running and the RAM keeps its bytes, because the chip is not part of the FPGA and has its own power.

The FPGA's side is another matter. A reset lets go of both wires, so the bus goes high, and it does not care what the clock was doing at that moment. If it was in the middle of sending a 0, it is still holding SDA low, waiting for clock pulses that the reset will not send. So the first thing a program does on the bus: if SDA reads 0, pulse SCL until it reads 1 (nine pulses are always enough to finish any byte and its acknowledge), then send a STOP. After that the clock is listening again, and your START will be heard.

In short. The DS1307 sits outside the FPGA on its own battery, so no reset touches its BCD time or its 56 bytes of RAM, and a fresh chip stays halted until you clear bit 7 of the seconds. A program bit-bangs I2C itself, SCL on port `$103B` and SDA on port `$113B`, with `$D0` to write and `$D1` to read. Code timed for 3.5 MHz holds the clock high seven times too briefly at 28 MHz, so slow the CPU with NextReg `$07`. Every START takes a snapshot, so one read's seven bytes belong to one moment.

Summary: The Next in One Page

Four parts, four ideas. This is what to keep when the details fade.

The Machine. A Z80N CPU in an FPGA that loads its core again at every power-up. Memory is eight 8K slots, named by NextRegs `$50` to `$57`, and ports and NextRegs are how you talk to everything else. Fourteen interrupt sources share `/INT`, so end every handler with `RETI`. The DMA moves a byte in six T-states where `LDIR` needs 21, and the CTC divides the 28 MHz clock, so its rates do not depend on the CPU's speed.

The Picture. A television turns time into position, so every dot must arrive at the right moment. The Next feeds five producers into four layers and shows the first opaque dot, in the order NextReg `$15` chooses. Every dot uses the registers as the beam finds them: write one mid-frame and the picture changes from that dot on. A line interrupt can wander by 44 dots; the Copper hits the same dot every frame, at no cost to the CPU.

The Sound. The beeper, three AY chips, four 8-bit DACs and the Pi add up in one mixer, and a silent Next sits at 1,536. AY chips play by themselves; the beeper and the DACs play only what the CPU, the DMA or the Copper writes, on time. The audio jack takes the lowest 12 bits of the sum, so keep it below 4,096 or it wraps and crackles. HDMI carries all 13 bits and never does.

Talking to the World. The SD card, the two UARTs and the clock are slow devices behind ports: SPI on `$E7` and `$EB`, UART 0 for the Wi-Fi module and UART 1 for the Pi, and I2C on `$103B` and `$113B`. Three habits keep them working. Never start an SPI exchange before the last one ends, so use `OTIR` and `INIR`. Drain a UART's 512-byte buffer every time you look: it fills in 44 ms at 115,200 baud. Slow the CPU with NextReg `$07` when you bit-bang a bus.

For every part. A soft reset keeps many settings, so set every register you depend on. When a loop's length matters, read the core version and count T-states at the speed you really have. And when a detail is in doubt, run the chapter's Try it demo in Klive IDE and look inside the machine.