ZX Spectrum Next

Assembly Developer Guide

Tomaž Kragelj

Z80 Undocumented by
Jan Wilmans
Sean Young

ZX Spectrum Next Assembly Developer Guide

Tomaž Kragelj

15 July 2022

REVISIONS
2022-07-15
2022-11-11
2021-07-16

Acknowlegements

A lot of work has gone into this book. But I also learned a lot. Not just about Z80 and Next, also LaTeX which basics I learned in secondary school and then all but forgotten. In this regard, I'd like to thank Jan Wilmans and Sean Young for creating the original Z80 Undocumented that I took as a basis for this book. I also need to thank countless kind folks on `https://tex.stackexchange.com/`, those who asked questions and those who answered them; I found solutions to most issues I encountered there (and I was one very frequent visitor at times :) I'm very grateful to all members of `z80-hardcore` Spectrum Next Discord channel who pointed out errors and shortcomings of the book but especially Peter Ped Helcmanovsky and Alvin Albrecht (aka Allen Albright) for help in fact-checking and pull request contributions. And last but not least, my family for being patient with my frequent long after-hours. Oh, also special shoutout to Bibi, our small JRT companion with the biggest heart :)

Contents

Chapter 1

Introduction

1.1 Where to get this document

ZX Spectrum Next Assembly Developer Guide is available as coil bound printed book on

`https://bit.ly/zx-next-assembly-dev-guide`

You can also download it as PDF document from GitHub where you can also find its source LaTeX form so you can edit it to your preference

`https://github.com/tomaz/zx-next-dev-guide`

1.2 Companion Source Code

GitHub repository also includes companion source code. Sample projects were created in a cross-platform environment on Windows so instructions on the following page are written with these in mind. All programs mentioned are available on Linux and macOS; you should be able to run everything on those platforms too, but likely with some deviations. Regardless, these are merely suggestions, you should be able to use your preferred editor or tools.

Visual Studio Code (`https://code.visualstudio.com/`)

My code editor of choice! I use it with the following plugins:

> **DeZog plugin** (`https://github.com/maziac/DeZog`)
>
> Essential plugin; features list is too large to even attempt to enumerate here but essentially turns VS Code into a fully-fledged debugging environment.
>
> **Z80 Macro-Assembler** (`https://github.com/mborik/z80-macroasm-vscode`)
>
> Another must-have plugin for the Z80 assembly developer; syntax highlighting, code formatting and code completion, renaming etc.
>
> **Z80 Instruction Set** (`https://github.com/maziac/z80-instruction-set`)
>
> Adds mouse hover action above any Z80N instruction for quick info.
>
> **Z80 Assembly meter** (`https://github.com/theNestruo/z80-asm-meter-vscode`)
>
> Shows the sum of clock cycles and machine code bytes for all instructions in the current selection.

sjasmplus 1.18.2 (`https://github.com/z00m128/sjasmplus`)

Source code includes sjasmplus specific directives for creating `nex` files at the top and bottom of `main.asm` files; if you use a different compiler, you may need to tweak or comment them out.

VS Code projects are set up to expect binaries in a specific folder. You will need to download and copy so that `sjasmplus.exe` is located in `Tools/sjasmplus`.

CSpect 2.13.0 (`http://cspect.org`)

Similar to sjasmplus, CSpect binaries are expected in a specific folder. To install, download and copy so that `CSpect.exe` is located in `Tools/CSpect` folder.

CSpect Next Image (`http://www.zxspectrumnext.online/#sd`)

You will also need to download the ZX Spectrum Next image file and copy it to the folder where `CSpect.exe` is located. I use a 2GB image, hence VS Code project file is configured for that. If you use a different image, make sure to update `.vscode/tasks.json` file.

DeZog CSpect plugin (`https://github.com/maziac/DeZogPlugin`)

DeZog requires this plugin to be installed to work with CSpect. To install, download and copy to the same folder where `CSpect.exe` is located. Make sure the plugin version matches the DeZog version!

Note: you need to have CSpect launched before you can run the samples. I created couple tasks[1] for it: open VS Code command palette (`Ctrl+Shift+P` shortcut on my installation) and select *Tasks: Run Task* option, then select *Launch CSpect* from list. This is only needed once. Afterwards, use *Run > Start Debugging* from the main menu to compile and launch the program.

Note: default DeZog port of 11000 doesn't work on my computer, so I changed it to 13000. This needs to be managed in 2 places: `.vscode/launch.json` and on the plugin side. Companion code repository already includes the setup needed, including `DeZogPlugin.dll.config` file, so it should work out of the box.

Note: sample projects are ready for ZEsarUX as well, select the option from debugging panel in VS Code.

[1] Workspace tasks seem to not be supported in some later VS Code versions. If this is the case for you, copy them to user tasks (shared between projects): open `.vscode/tasks.json` file from any of the sample projects, scroll down a little and copy `Launch CSpect` and `Launch ZEsarUX` tasks to user tasks. You can do this all from within VS Code. To open the user tasks file, open the command palette and start typing `open user tasks`, then select the option from the drop-down menu.

1.3 Background, Contact & Feedback

My first computer was ZX Spectrum 48K. Initially, it was only used to play games, but my creative mind soon set me on the path of building simple games of my own in BASIC. While too young to master assembler at that point, the idea stayed with me. ZX Spectrum Next revived my wish to learn Z80 and return to writing games for the platform.

My original intent was to have coil bound list of all ZX Next instructions so I can quickly compare. However, after finding Z80 Undocumented online, it felt like a perfect starting point. And with additional information included, it also encouraged me to extend the mere instructions list with the Next specific chapters. In a way, this book represents my notes as I was learning those topics. That being said, I did my best to present information as a reference to keep the book relevant.

During the process, I wanted to tweak or unify the look of various elements. For example instruction tables. Original LaTeX code required applying changes to each and every instance. As LaTeX is all but a programming language, I extracted individual elements into reusable commands which allowed me to tweak appearance in a single place and apply it to the whole document. I also converted almost all drawings from `picture` to `tikz` as it's far more adaptable. With this, I almost completely restructured the original LaTeX code. While this took a lot of time and effort, it allowed me to quickly iterate later on. I really love this aspect of LaTeX!

English is not my native tongue. And our mind is not the best tool to correct our own work either. Since I can't afford a professional proofreader, mistakes are a matter of fact I'm afraid. If you spot something or want to contribute, feel free to open an issue on GitHub. Pull requests are also welcome! If you want to contribute, but are unsure of what, check the accompanying readme file on GitHub for ideas. If you want to discuss in advance, or for anything else, you can find me on email `tkragelj AT gmail DOT com` or Twitter `@tomsbarks`.

That being said, I hope you'll enjoy reading this document as much as I did writing it!

Sincerely, Tomaž

1.4 Z80 Undocumented

As the saying "standing on the shoulders of giants" goes, this book is also based on pre-existing work from Jan and Sean. While my work is ZX Spectrum Next developer-oriented, their original project was more focused on hardware perspective, for Z80 emulator developers.

If interested, you can find it at `http://www.myquest.nl/z80undocumented/`.

Jan

`http://www.myquest.nl/z80undocumented/`
Email `jw AT dds DOT nl`
Twitter `@janwilmans`

Interested in emulation for a long time, but a few years after Sean started writing this document, I have also started writing my own MSX emulator in 2003 and I've used this document quite a lot. Now (2005) the Z80 emulation is nearing perfection, I decided to add what extra I have learned and comments various people have sent to Sean, to this document.

I have restyled the document (although very little) to fit my personal needs and I have checked a lot of things that were already in here.

Sean

`http://www.msxnet.org/`

Ever since I first started working on an MSX emulator, I've been very interested in getting the emulation absolutely correct - including the undocumented features. Not just to make sure that all games work, but also to make sure that if a program crashes, it crashes exactly the same way if running on an emulator as on the real thing. Only then is perfection achieved.

I set about collecting information. I found pieces of information on the Internet, but not everything there is to know. So I tried to fill in the gaps, the results of which I put on my website. Various people have helped since then; this is the result of all those efforts and to my knowledge, this document is the most complete.

1.5 ChangeLog

15 July 2022 Corrections, updates and improvements. Main focus on making instruction up close chapter more useful. Each instruction now includes description of effects on flags and where makes sense, includes additional description or code examples.

11 November 2021 Corrections and updates based on community comments - with special thanks to Peter Ped Helcmanovsky and Alvin Albrecht. Restructured and updated many ZX Next chapters: added sample code to ports, completely restructured memory map and paging, added new palette chapter including 9-bit palette handling, updated ULA with shadow screen info and added Next extended keyboard, DMA, Copper and Hardware IM2 sections. Other than some cosmetic changes: redesigned title, copyright pages etc. Also, many behind the scenes improvements like splitting previous huge single LaTeX file into multiple per-chapter/section. This is not only more manageable but can also compile much faster.

16 July 2021 Added ZX Spectrum Next information and instructions and restructured text for better maintainability and readability.

18 September 2005 Corrected a textual typo in the R register and memory refresh section, thanks to David Aubespin. Corrected the contradiction in the `DAA` section saying the **NF** flag was both affected and unchanged :) thanks to Dan Meir. Added an error in official documentation about the way Interrupt Mode 2 works, thanks to Aaldert Dekker.

15 June 2005 Corrected improper notation of `JP x,nn` mnemonics in opcode list, thanks to Laurens Holst. Corrected a mistake in the `INI`, `INIR`, `IND`, `INDR` section and documented a mistake in official Z80 documentation concerning Interrupt Mode 2, thanks to Boris Donko. Thanks to Aaldert Dekker for his ideas, for verifying many assumptions and for writing instruction exercisers for various instruction groups.

18 May 2005 Added an alphabetical list of instructions for easy reference and corrected an error in the 16-bit arithmetic section, `SBC HL,nn` sets the NF flag just like other subtraction instructions, thanks to Fredrik Olssen for pointing that out.

4 April 2005 I (Jan `jw AT dds DOT nl`) will be maintaining this document from this version on. I restyled the document to fix the page numbering issues, corrected an error in the I/O Block Instructions section, added graphics for the `RLD` and `RRD` instructions and corrected the spelling in several places.

20 November 2003 Again, thanks to Ramsoft, added PV flag to `OUTI`, `INI` and friends. Minor fix to `DAA` tables, other minor fixes.

13 November 2003 Thanks to Ramsoft, add the correct tables for the `DAA` instruction (section 2.3.7, page 19). Minor corrections & typos, thanks to Jim Battle, David Sutherland and most of all Fred Limouzin.

September 2001 Previous documents I had written were in plain text and Microsoft Word, which I now find very embarrassing, so I decided to combine them all and use LaTeX. Apart from a full re-write, the only changed information is “Power on defaults” (section 2.1.3, page 10) and the algorithm for the CF and HF flags for `OTIR` and friends (section 2.3.3, page 17).

This page intentionally left empty

Chapter 2

Zilog Z80

2.1 Overview

2.1.1 History of the Z80

In 1969 Intel was approached by a Japanese company called Busicom to produce chips for Busicom's electronic desktop calculator. Intel suggested that the calculator should be built around a single-chip generalized computing engine and thus was born the first microprocessor - the 4004. Although it was based on ideas from a much larger mainframe and mini-computers the 4004 was cut down to fit onto a 16-pin chip, the largest that was available at the time, so that its data bus and address bus were each only 4-bits wide.

Intel went on to improve the design and produced the 4040 (an improved 4-bit design) the 8008 (the first 8-bit microprocessor) and then in 1974 the 8080. This last one turned out to be a very useful and popular design and was used in the first home computer, the Altair 8800, and CP/M.

In 1975 Federico Faggin who had worked at Intel on the 4004 and its successors left the company and joined forces with Masatoshi Shima to form Zilog. At their new company, Faggin and Shima designed a microprocessor that was compatible with Intel's 8080 (it ran all 78 instructions of the 8080 in almost the same way that Intel's chip did)[1] but had many more abilities (an extra 120 instructions, many more registers, simplified connection to hardware). Thus was born the mighty Z80, and thus was the empire forged!

The original Z80 was first released in July 1976, coincidentally Jan was born in the very same month. Since then newer versions have appeared with much of the same architecture but running at higher speeds. The original Z80 ran with a clock rate of 2.5MHz, the Z80A runs at 4MHz, the Z80B at 6MHz and the Z80H at 8Mhz.

Many companies produced machines based around Zilog's improved chip during the 1970s and 80's and because the chip could run 8080 code without needing any changes to the code the perfect choice of the operating system was CP/M.

Also, Zilog has created a Z280, an enhanced version of the Zilog Z80 with a 16-bit architecture, introduced in July 1987. It added an MMU to expand addressing to 16Mb, features for multitasking, a 256-byte cache, and a huge number of new opcodes (giving a total of over 2000!). Its internal clock runs at 2 or 4 times the external clock (e.g. a 16MHz CPU with a 4MHz bus.

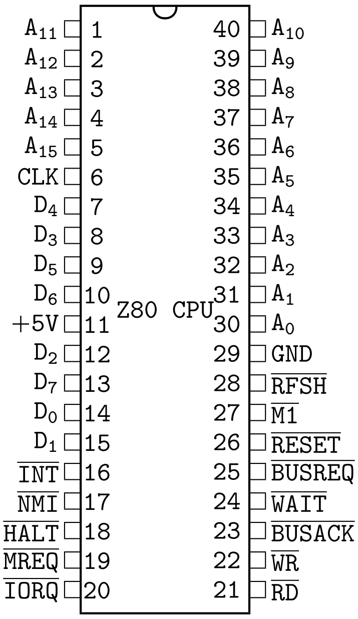
The Z380 CPU incorporates advanced architectural while maintaining Z80/Z180 object code compatibility. The Z380 CPU is an enhanced version of the Z80 CPU. The Z80 instruction set has been retained, adding a full complement of 16-bit arithmetic and logical operations, multiply and divide, a complete set of register-to-register loads and exchanges, plus 32-bit load and exchange, and 32-bit arithmetic operations for address calculations.

The addressing modes of the Z80 have been enhanced with Stack pointer relative loads and stores, 16-bit and 24-bit indexed offsets and more flexible indirect register addressing. All of the addressing modes allow access to the entire 32-bit addressing space.

[1] Thanks to Jim Battle (`frustum AT pacbell DOT net`): the 8080 always puts the parity in the PF flag; VF does not exist and the timing is different. Possibly there are other differences.

2.1.2 Pin Descriptions [7]

This section might be relevant even if you don't do anything with hardware; it might give you insight into how the Z80 operates. Besides, it took me hours to draw this.


Z80 CPU
A_{11} 1
A_{12} 2
A_{13} 3
A_{14} 4
A_{15} 5
CLK 6
D_4 7
D_3 8
D_5 9
D_6 10
+5V 11
D_2 12
D_7 13
D_0 14
D_1 15
$\overline{\mathtt{INT}}$ 16
$\overline{\mathtt{NMI}}$ 17
$\overline{\mathtt{HALT}}$ 18
$\overline{\mathtt{MREQ}}$ 19
$\overline{\mathtt{IORQ}}$ 20
40 A_{10}
39 A_9
38 A_8
37 A_7
36 A_6
35 A_5
34 A_4
33 A_3
32 A_2
31 A_1
30 A_0
29 GND
28 $\overline{\mathtt{RFSH}}$
27 $\overline{\mathtt{M1}}$
26 $\overline{\mathtt{RESET}}$
25 $\overline{\mathtt{BUSREQ}}$
24 $\overline{\mathtt{WAIT}}$
23 $\overline{\mathtt{BUSACK}}$
22 $\overline{\mathtt{WR}}$
21 $\overline{\mathtt{RD}}$

A_{15}-A_0 *Address bus* (output, active high, 3-state). This bus is used for accessing the memory and for I/O ports. During the refresh cycle the IR register is put on this bus.

$\overline{\mathtt{BUSACK}}$ *Bus Acknowledge* (output, active low). Bus Acknowledge indicates to the requesting device that the CPU address bus, data bus, and control signals $\overline{\mathtt{MREQ}}$, $\overline{\mathtt{IORQ}}$, $\overline{\mathtt{RD}}$ and $\overline{\mathtt{WR}}$ have been entered into their high-impedance states. The external device now control these lines.

$\overline{\mathtt{BUSREQ}}$ *Bus Request* (input, active low). Bus Request has a higher priority than $\overline{\mathtt{NMI}}$ and is always recognised at the end of the current machine cycle. $\overline{\mathtt{BUSREQ}}$ forces the CPU address bus, data bus and control signals $\overline{\mathtt{MREQ}}$, $\overline{\mathtt{IORQ}}$, $\overline{\mathtt{RD}}$ and $\overline{\mathtt{WR}}$ to go to a high-impedance state so that other devices can control these lines. $\overline{\mathtt{BUSREQ}}$ is normally wired-OR and requires an external pullup for these applications. Extended $\overline{\mathtt{BUSREQ}}$ periods due to extensive DMA operations can prevent the CPU from refreshing dynamic RAMs.

D_7-D_0 *Data Bus* (input/output, active low, 3-state). Used for data exchanges with memory, I/O and interrupts.

$\overline{\mathtt{HALT}}$ *Halt State* (output, active low). Indicates that the CPU has executed a `HALT` instruction and is waiting for either a maskable or nonmaskable interrupt (with the mask enabled) before operation can resume. While halted, the CPU stops increasing the PC so the instruction is re-executed, to maintain memory refresh.

$\overline{\mathtt{INT}}$ *Interrupt Request* (input, active low). Interrupt Request is generated by I/O devices. The CPU honours a request at the end of the current instruction if `IFF1` is set. $\overline{\mathtt{INT}}$ is normally wired-OR and requires an external pullup for these applications.

$\overline{\mathtt{IORQ}}$ *Input/Output Request* (output, active low, 3-state). Indicates that the address bus holds a valid I/O address for an I/O read or write operation. $\overline{\mathtt{IORQ}}$ is also generated concurrently with $\overline{\texttt{M1}}$ during an interrupt acknowledge cycle to indicate that an interrupt response vector can be placed on the databus.

$\overline{\texttt{M1}}$ *Machine Cycle One* (output, active low). $\overline{\texttt{M1}}$, together with $\overline{\texttt{MREQ}}$, indicates that the current machine cycle is the opcode fetch cycle of an instruction execution. $\overline{\texttt{M1}}$, together with $\overline{\texttt{IORQ}}$, indicates an interrupt acknowledge cycle.

$\overline{\texttt{MREQ}}$ *Memory Request* (output, active low, 3-state). Indicates that the address holds a valid address for a memory read or write cycle operations.

$\overline{\texttt{NMI}}$ *Non-Maskable Interrupt* (input, negative edge-triggered). $\overline{\texttt{NMI}}$ has a higher priority than $\overline{\texttt{INT}}$. $\overline{\texttt{NMI}}$ is always recognised at the end of an instruction, independent of the status of the interrupt flip-flops and automatically forces the CPU to restart at location $0066.

$\overline{\texttt{RD}}$ *Read* (output, active low, 3-state). Indicates that the CPU wants to read data from memory or an I/O device. The addressed I/O device or memory should use this signal to place data onto the data bus.

$\overline{\texttt{RESET}}$ *Reset* (input, active low). Initializes the CPU as follows: it resets the interrupt flip-flops, clears the PC and IR registers, and set the interrupt mode to 0. During reset time, the address bus and data bus go to a high-impedance state, and all control output signals go to the inactive state. Note that $\overline{\texttt{RESET}}$ must be active for a minimum of three full clock cycles before the reset operation is complete. Note that Matt found that SP and AF are set to $FFFF.

$\overline{\texttt{RFSH}}$ *Refresh* (output, active low). $\overline{\texttt{RFSH}}$, together with $\overline{\texttt{MREQ}}$, indicates that the IR registers are on the address bus (note that only the lower 7 bits are useful) and can be used for the refresh of dynamic memories.

$\overline{\texttt{WAIT}}$ *Wait* (input, active low). Indicates to the CPU that the addressed memory or I/O device are not ready for data transfer. The CPU continues to enter a wait state as long as this signal is active. Note that during this period memory is not refreshed.

$\overline{\texttt{WR}}$ *Write* (output, active low, 3-state). Indicates that the CPU wants to write data to memory or an I/O device. The addressed I/O device or memory should use this signal to store the data on the data bus.

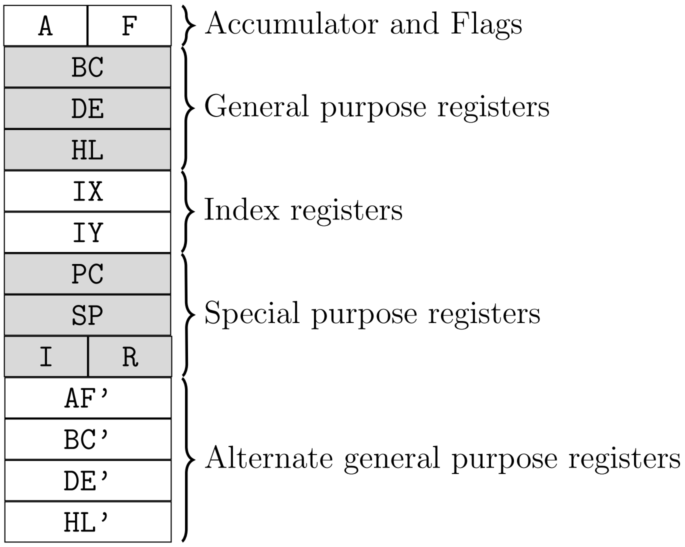
2.1.3 Power on Defaults

Matt[2] has done some excellent research on this. He found that AF and SP are always set to `$FFFF` after a reset, and all other registers are undefined (different depending on how long the CPU has been powered off, different for different Z80 chips). Of course, the PC should be set to 0 after a reset, and so should the `IFF1` and `IFF2` flags (otherwise strange things could happen). Also since the Z80 is 8080 compatible, the interrupt mode is probably 0.

Probably the best way to simulate this in an emulator is to set `PC`, `IFF1`, `IFF2`, `IM` to 0 and set all other registers to `$FFFF`.

[2] `redflame AT xmission DOT com`

2.1.4 Registers


A
F
BC
DE
HL
IX
IY
PC
SP
I
R
AF'
BC'
DE'
HL'
Accumulator and Flags
General purpose registers
Index registers
Special purpose registers
Alternate general purpose registers

2.1.5 Flags

The conventional way of denoting the flags is with one letter, "C" for the carry flag for example. It could be confused with the C register, so I've chosen to use the "CF" notation for flags (except "P" which uses "PV" notation due to having dual-purpose, either as parity or overflow). And for YF and XF the same notation is used in MAME[3].

bit	7	6	5	4	3	2	1	0
flag	SF	ZF	YF	HF	XF	PF	NF	CF

SF Set if the 2-complement value is negative; simply a copy of the most significant bit.

ZF Set if the result is zero.

YF A copy of bit 5 of the result.

HF The half-carry of an addition/subtraction (from bit 3 to 4). Needed for BCD correction with `DAA`.

XF A copy of bit 3 of the result.

PV This flag can either be the parity of the result (PF), or the 2-complement signed overflow (VF): set if 2-complement value doesn't fit in the register.

NF Shows whether the last operation was an addition (0) or a subtraction (1). This information is needed for `DAA`.[4]

CF The carry flag, set if there was a carry after the most significant bit.

[3] `http://www.mame.net/`

[4] Wouldn't it be better to have separate instructions for `DAA` after addition and subtraction, like the 80x86 has instead of sacrificing a bit in the flag register?

2.2 Undocumented Opcodes

There are quite a few undocumented opcodes/instructions. This section should describe every possible opcode so you know what will be executed, whatever the value of the opcode is.

The following prefixes exist: CB, ED, DD, FD, DDCB and FDCB. Prefixes change the way the following opcodes are interpreted. All instructions without a prefix (not a value of one the above) are single-byte opcodes (without the operand, that is), which are documented in the official documentation.

2.2.1 CB Prefix [5]

An opcode with a CB prefix is a rotate, shift or bit test/set/reset instruction. A few instructions are missing from the official list, for example SLL (Shift Logical Left). It works like SLA, for one exception: it sets bit 0 (SLA resets it).

CB30	SLL B	CB34	SLL H
CB31	SLL C	CB35	SLL L
CB32	SLL D	CB36	SLL (HL)
CB33	SLL E	CB37	SLL A

2.2.2 DD Prefix [5]

In general, the instruction following the DD prefix is executed as is, but if the HL register is supposed to be used the IX register is used instead. Here are the rules:

- Any usage of HL is treated as an access to IX (except EX DE,HL and EXX and the ED prefixed instructions that use HL).
- Any access to (HL) is changed to (IX+d), where “d” is a signed displacement byte placed after the main opcode - except JP (HL), which isn’t indirect anyway. The mnemonic should be JP HL.
- Any access to H is treated as an access to IX_h (the high byte of IX) except if (IX+d) is used as well.
- Any access to L is treated as an access to IX_l (the low byte of IX) except if (IX+d) is used as well.
- A DD prefix before a CB selects a completely different instruction set, see section 2.2.5, page 14.

Some examples:

Without	With DD prefix	Without	With DD prefix
LD H, (HL)	LD H, (IX+d)	JP (HL)	JP (IX)
LD H, A	LD IXH, A	LD DE, 0	LD DE, 0
LD L, H	LD IXL, IXH	LD HL, 0	LD IX, 0

2.2.3 ED Prefix [5]

There are a number of undocumented EDxx instructions, of which most are duplicates of documented instructions. Any instruction not listed here has no effect (same as 2 NOPs). ** indicates undocumented instruction:

ED40	IN B, (C)	ED50	IN D, (C)
ED41	OUT (C), B	ED51	OUT (C), D
ED42	SBC HL, BC	ED52	SBC HL, DE
ED43	LD (nn), BC	ED53	LD (nn), DE
ED44	NEG	ED54	NEG**
ED45	RETN	ED55	RETN**
ED46	IM 0	ED56	IM 1
ED47	LD I, A	ED57	LD A, I
ED48	IN C, (C)	ED58	IN E, (C)
ED49	OUT (C), C	ED59	OUT (C), E
ED4A	ADC HL, BC	ED5A	ADC HL, DE
ED4B	LD BC, (nn)	ED5B	LD DE, (nn)
ED4C	NEG**	ED5C	NEG**
ED4D	RETI	ED5D	RETN**
ED4E	IM 0**	ED5E	IM 2
ED4F	LD R, A	ED5F	LD A, R

ED60	IN H, (C)	ED70	IN (C) / IN F, (C)**
ED61	OUT (C), H	ED71	OUT (C), 0**
ED62	SBC HL, HL	ED72	SBC HL, SP
ED63	LD (nn), HL	ED73	LD (nn), SP
ED64	NEG**	ED74	NEG**
ED65	RETN**	ED75	RETN**
ED66	IM 0**	ED76	IM 1**
ED67	RRD	ED77	NOP**
ED68	IN L, (C)	ED78	IN A, (C)
ED69	OUT (C), L	ED79	OUT (C), A
ED6A	ADC HL, HL	ED7A	ADC HL, SP
ED6B	LD HL, (nn)	ED7B	LD SP, (nn)
ED6C	NEG**	ED7C	NEG**
ED6D	RETN**	ED7D	RETN**
ED6E	IM 0**	ED7E	IM 2**
ED6F	RLD	ED7F	NOP**

The ED70 instruction reads from I/O port C, but does not store the result. It just affects the flags like the other IN x,(C) instructions. ED71 simply outs the value 0 to I/O port C.

The ED63 is a duplicate of the 22 opcode (LD (nn),HL) and similarly ED6B is a duplicate of the 2A opcode (LD HL,(nn)). Of course the timings are different. These instructions are listed in the official documentation.

According to Gerton Lunter[5]:

> The instructions ED 4E and ED 6E are IM 0 equivalents: when FF was put on the bus (physically) at interrupt time, the Spectrum continued to execute normally, whereas when an EF (RST $28) was put on the bus it crashed, just as it does in that case when the Z80 is in the official interrupt mode 0. In IM 1 the Z80 just executes a RST $38 (opcode FF) no matter what is on the bus.

All the RETI/RETN instructions are the same, all like the RETN instruction. So they all, including RETI, copy IFF2 to IFF1. See section 2.4.3, page 21 for more information on RETI and RETN and IM x.

2.2.4 FD Prefix [5]

This prefix has the same effect as the DD prefix, though IY is used instead of IX. Note LD IXL, IYH is not possible: only IX or IY is accessed in one instruction, never both.

2.2.5 DDCB Prefix

The undocumented DDCB instructions store the result (if any) of the operation in one of the seven all-purpose registers. Which one depends on the lower 3 bits of the last byte of the opcode (not operand, so not the offset).

000	B	100	H
001	C	101	L
010	D	110	(none: documented opcode)
011	E	111	A

The documented DDCB0106 is RLC (IX+$01). So, clear the lower three bits (DDCB0100) and something is done to register B. The result of the RLC (which is stored in (IX+$01)) is now also stored in register B. Effectively, it does the following:

```
    LD B, (IX+$01)
    RLC B
    LD (IX+$01), B
```

So you get double value for money. The result is stored in B and (IX+$01). The most common notation is: RLC (IX+$01), B

I've once seen this notation:

```
    RLC (IX+$01)
    LD B, (IX+$01)
```

That's not correct: B contains the rotated value, even if (IX+$01) points to ROM.

[5] gerton AT math.rug DOT nl

The DDCB SET and RES instructions do the same thing as the shift/rotate instructions:

```
DDCB10C0    SET 0, (IX+$10), B
DDCB10C1    SET 0, (IX+$10), C
DDCB10C2    SET 0, (IX+$10), D
DDCB10C3    SET 0, (IX+$10), E
DDCB10C4    SET 0, (IX+$10), H
DDCB10C5    SET 0, (IX+$10), L
DDCB10C6    SET 0, (IX+$10) - documented instruction
DDCB10C7    SET 0, (IX+$10), A
```

So for example with the last instruction, the value of (IX+$10) with bit 0 set is also stored in register A.

The DDCB BIT instructions do not store any value; they merely test a bit. That's why the undocumented DDCB BIT instructions are no different from the official ones:

```
DDCB d 78   BIT 7, (IX+d)
DDCB d 79   BIT 7, (IX+d)
DDCB d 7A   BIT 7, (IX+d)
DDCB d 7B   BIT 7, (IX+d)
DDCB d 7C   BIT 7, (IX+d)
DDCB d 7D   BIT 7, (IX+d)
DDCB d 7E   BIT 7, (IX+d) - documented instruction
DDCB d 7F   BIT 7, (IX+d)
```

2.2.6 FDCB Prefixes

Same as for the DDCB prefix, though IY is used instead of IX.

2.2.7 Combinations of Prefixes

This part may be of some interest to emulator coders. Here we define what happens if strange sequences of prefixes appear in the instruction cycle of the Z80.

If CB or ED is encountered, that byte plus the next make up an instruction. FD or DD should be seen as prefix setting a flag which says "use IX or IY instead of HL", and not an instruction. In a large sequence of DD and FD bytes, it is the last one that counts. Also any other byte (or instruction) resets this flag.

```
FD DD 00 21 00 10    NOP NOP NOP LD HL, $1000
```

2.3 Undocumented Effects

2.3.1 BIT Instructions

`BIT n,r` behaves much like `AND r,2`^{n} with the result thrown away, and CF flag unaffected. Compare `BIT 7,A` with `AND $80`: flag YF and XF are reset, SF is set if bit 7 was actually set; ZF is set if the result was 0 (bit was reset), and PV is effectively set if ZF is set (the result of the AND leaves either no bits set (PV set - parity even) or one bit set (PV reset - parity odd). So the rules for the flags are:

SF flag Set if n = 7 and tested bit is set.

ZF flag Set if the tested bit is reset.

YF flag Set if n = 5 and tested bit is set.

HF flag Always set.

XF flag Set if n = 3 and tested bit is set.

PV flag Set just like ZF flag.

NF flag Always reset.

CF flag Unchanged.

This is where things start to get strange. With the `BIT n,(IX+d)` instructions, the flags behave just like the `BIT n,r` instruction, except for YF and XF. These are not copied from the result but from something completely different, namely bit 5 and 3 of the high byte of `IX+d` (so `IX` plus the displacement).

Things get more bizarre with the `BIT n,(HL)` instruction. Again, except for YF and XF, the flags are the same. YF and XF are copied from some sort of internal register. This register is related to 16-bit additions. Most instructions do not change this register. Unfortunately, I haven't tested all instructions yet, but here is the list so far:

`ADD HL, xx`	Use high byte of `HL`, ie. `H` before the addition.
`LD r, (IX+d)`	Use high byte of the resulting address `IX+d`.
`JR d`	Use high byte target address of the jump.
`LD r, r'`	Doesn't change this register.

Any help here would be most appreciated!

2.3.2 Memory Block Instructions [1]

The LDI/LDIR/LDD/LDDR instructions affect the flags in a strange way. At every iteration, a byte is copied. Take that byte and add the value of register A to it. Call that value n. Now, the flags are:

YF flag A copy of bit 1 of n.

HF flag Always reset.

XF flag A copy of bit 3 of n.

PV flag Set if BC not 0.

SF, ZF, CF flags These flags are unchanged.

And now for CPI/CPIR/CPD/CPDR. These instructions compare a series of bytes in memory to register A. Effectively, it can be said they perform CP (HL) at every iteration. The result of that comparison sets the HF flag, which is important for the next step. Take the value of register A, subtract the value of the memory address, and finally subtract the value of HF flag, which is set or reset by the hypothetical CP (HL). So, n=A-(HL)-HF.

SF, ZF, HF flags Set by the hypothetical CP (HL).

YF flag A copy of bit 1 of n.

XF flag A copy of bit 3 of n.

PV flag Set if BC is not 0.

NF flag Always set.

CF flag Unchanged.

2.3.3 I/O Block Instructions

These are the most bizarre instructions, as far as the flags are concerned. Ramsoft found all of the flags. The “out” instructions behave differently than the “in” instructions, which doesn’t make the CPU very symmetrical.

First of all, all instructions affect the following flags:

SF, ZF, YF, XF flags Affected by decreasing register B, as in DEC B.

NF flag A copy of bit 7 of the value read from or written to an I/O port.

And now the for `OUTI/OTIR/OUTD/OTDR` instructions. Take the state of the `L` after the increment or decrement of HL; add the value written to the I/O port; call that `k` for now. If `k` > 255, then the CF and HF flags are set. The PV flag is set like the parity of `k` bitwise and'ed with 7, bitwise xor'ed with `B`.

HF and CF Both set if `((HL) + L > 255)`

PV The parity of `((((HL) + L) ∧ 7) ⊻ B)`

`INI/INIR/IND/INDR` use the `C` register instead of the `L` register. There is a catch though, because not the value of `C` is used, but `C + 1` if it's `INI/INIR` or `C - 1` if it's `IND/INDR`. So, first of all `INI/INIR`:

HF and CF Both set if `((HL) + ((C + 1) ∧ 255) ⊻ 255)`

PF The parity of `(((HL) + ((C + 1) ∧ 255)) ∧ 7) ⊻ B)`

And last `IND/INDR`:

HF and CF Both set if `((HL) + ((C - 1) ∧ 255) > 255)`

PF The parity of `(((HL) + ((C - 1) ∧ 255)) ∧ 7) ⊻ B)`

2.3.4 16 Bit I/O ports

Officially the Z80 has an 8-bit I/O port address space. When using the I/O ports, the 16 address lines are used. And in fact, the high 8 bits do have some value, so you can use 65536 ports after all. `IN r, (C)`, `OUT (C), r`, and the block I/O instructions actually place the entire `BC` register on the address bus. Similarly `IN A, (n)` and `OUT (n), A` put `A ×256 + n` on the address bus.

The `INI`, `INIR`, `IND` and `INDR` instructions use `BC` before decrementing `B`, and the `OUTI`, `OTIR`, `OUTD` and `OTDR` instructions use `BC` after decrementing.

2.3.5 Block Instructions

The repeated block instructions simply decrement the `PC` by two so the instruction is simply re-executed. So interrupts can occur during block instructions. So, `LDIR` is simply `LDI` + if `BC` is not `0`, decrement `PC` by `2`.

2.3.6 16 Bit Additions

The 16-bit additions are a bit more complicated than the 8-bit ones. Since the Z80 is an 8-bit CPU, 16-bit additions are done in two stages: first, the lower bytes are added, then the two higher bytes. The SF, YF, HF, XF flags are affected by the second (high) 8-bit addition. ZF is set if the whole 16-bit result is 0.

2.3.7 DAA Instruction

This instruction is useful when you're using BCD values. After addition or subtraction, DAA corrects the value back to BCD again. Note that it uses the CF flag, so it cannot be used after INC and DEC.

Stefano Donati from Ramsoft[6] has found the tables which describe the DAA operation. The input is the A register and the CF, NF, HF flags. The result is as follows:

Depending on the NF flag, the "diff" from this table must be added (NF is reset) or subtracted (NF is set) to A:

	high		low	
CF	nibble	HF	nibble	diff
0	0-9	0	0-9	00
0	0-9	1	0-9	06
0	0-8	*	A-F	06
0	A-F	0	0-9	60
1	*	0	0-9	60
1	*	1	0-9	66
1	*	*	A-F	66
0	9-F	*	A-F	66
0	A-F	1	0-9	66

CF flag is affected:

	high	low	
CF	nibble	nibble	CF'
0	0-9	0-9	0
0	0-8	A-F	0
0	9-F	A-F	1
0	A-F	0-9	1
1	*	*	1

HF flag is affected:

		low	
NF	HF	nibble	HF'
0	*	0-9	0
0	*	A-F	1
1	0	*	0
1	1	6-F	0
1	1	0-5	1

SF, YF, XF are copies of bit 7, 5, 3 of the result respectively; ZF is set according to the result and NF is always unchanged.

2.4 Interrupts

There are two types of interrupts, maskable and non-maskable. The maskable type is ignored if IFF1 is reset. Non-maskable interrupts (NMI) will are always accepted, and they have a higher priority, so if both are requested at the same time, the NMI will be accepted first.

For the interrupts, the following things are important: interrupt Mode (set with the IM 0, IM 1, IM 2 instructions), the interrupt flip-flops (IFF1 and IFF2), and the I register. When a maskable interrupt is accepted, the external device can put a value on the data bus.

Both types of interrupts increase the R register by one when accepted.

2.4.1 Non-Maskable Interrupts (NMI)

When an NMI is accepted, IFF1 is reset. At the end of the routine, IFF1 must be restored (so the running program is not affected). That's why IFF2 is there; to keep a copy of IFF1.

An NMI is accepted when the $\overline{\mathtt{NMI}}$ pin on the Z80 is made low (edge-triggered). The Z80 responds to the change of the line from +5 to 0 - so the interrupt line doesn't have a state, it's just a pulse. When this happens, a call is done to address $0066 and IFF1 is reset so the

[6] http://www.ramsoft.bbk.org/

routine isn't bothered by maskable interrupts. The routine should end with an `RETN` (RETurn from Nmi) which is just a usual `RET` but also copies `IFF2` to `IFF1`, so the IFFs are the same as before the interrupt.

You can check whether interrupts were disabled or not during an NMI by using the `LD A,I` or `LD A,R` instruction. These instructions copy `IFF2` to the PV flag.

Accepting an NMI costs 11 t-states.

2.4.2 Maskable Interrupts (INT)

If the $\overline{\texttt{INT}}$ line is low and `IFF1` is set, a maskable interrupt is accepted - whether or not the last interrupt routine has finished. That's why you should not enable interrupts during such a routine, and make sure that the device that generated it has put the $\overline{\texttt{INT}}$ line up again before ending the routine. So unlike NMI interrupts, the interrupt line has a state; it's not a pulse.

When an interrupt is accepted, both `IFF1` and `IFF2` are cleared, preventing another interrupt from occurring which would end up as an infinite loop (and overflowing the stack). What happens next depends on the Interrupt Mode.

A device can place a value on the data bus when the interrupt is accepted. Some computer systems do not utilize this feature, and this value ends up being `$FF`.

Interrupt Mode 0 This is the 8080 compatibility mode. The instruction on the bus is executed (usually an `RST` instruction, but it can be anything). `I` register is not used. Assuming it's a `RST` instruction, accepting this takes 13 t-states.

Interrupt Mode 1 This is the 8080 compatibility mode. The instruction on the bus is executed (usually an `RST` instruction, but it can be anything). `I` register is not used. Assuming it's a `RST` instruction, accepting this takes 13 t-states.

Interrupt Mode 2 A call is made to the address read from memory. What address is read from is calculated as follows: $(\mathit{I\ register}) \times 256 + (\mathit{value\ on\ bus})$. Zilog's user manual states (very convincingly) that the least significant bit of the address is always 0, so they calculate the address that is read from as: $(\mathit{I\ register}) \times 256 + (\mathit{value\ on\ bus} \wedge \texttt{\$FE})$. I have tested this and it's not correct. Of course, a word (two bytes) is read, making the address where the call is made to. In this way, you can have a vector table for interrupts. Accepting this interrupt type costs 19 t-states.

At the end of a maskable interrupt, the interrupts should be enabled again. You can assume that was the state of the IFFs because otherwise the interrupt wasn't accepted. So, an interrupt routine always ends with an `EI` and a `RET` (`RETI` according to the official documentation, more about that later):

```
InterruptRoutine:
    ...
    EI
    RETI or RET
```

Note a fact about EI: a maskable interrupt isn't accepted directly after it, so the next opportunity for an interrupt is after the RETI. This is very useful; if the $\overline{\texttt{INT}}$ line is still low, an interrupt is accepted again. If this happens a lot and the interrupt is generated before the RETI, the stack could overflow (since the routine would be called again and again). But this property of EI prevents this.

DI is not necessary at the start of the interrupt routine: the interrupt flip-flops are cleared when accepting the interrupt.

You can use RET instead of RETI, depending on the hardware setup. RETI is only useful if you have something like a Z80 PIO to support daisy-chaining: queuing interrupts. The PIO can detect that the routine has ended by the opcode of RETI, and let another device generate an interrupt. That is why I called all the undocumented EDxx RET instructions RETN: All of them operate alike, the only difference of RETI is its specific opcode which the Z80 PIO recognises.

2.4.3 Things Affecting the Interrupt Flip-Flops

All the IFF related things are:

Accept NMI	IFF1	IFF2	
CPU reset	0	0	
DI	0	0	
EI	1	1	
Accept INT	0	0	
Accept NMI	0	-	
RETI/N	IFF2	-	All the EDxx RETI/N instructions
LD A,I / LD A,R	-	-	Copies IFF2 into PV flag

If you're working with a Z80 system without NMIs (like the MSX), you can forget all about the two separate IFFs; since an NMI isn't ever generated, the two will always be the same.

Some documentation says that when an NMI is accepted, IFF1 is first copied into IFF2 before IFF1 is cleared. If this is true, the state of IFF2 is lost after a nested NMI, which is undesirable. Have tested this in the following way: make sure the Z80 is in EI mode, generate an NMI. In the NMI routine, wait for another NMI before executing RETN. In the second NMI IFF2 was still set, so IFF1 is *not* copied to IFF2 when accepting an NMI.

Another interesting fact: I was trying to figure out whether the undocumented ED RET instructions were RETN or RETI. I tested this by putting the machine in EI mode, wait for an NMI and end with one of the ED RET instructions. Then execute a HALT instruction. If IFF1 was not restored, the machine would hang but this did not happen with any of the instructions, including the documented RETI!

Since every interrupt routine must end with EI followed by RETI officially, It does not matter that RETI copies IFF2 into IFF1; both are set anyway.

2.4.4 HALT Instruction

The HALT instruction halts the Z80; it does not increase the PC so that the instruction is re-executed until a maskable or non-maskable interrupt is accepted. Only then does the Z80 increase the PC again and continues with the next instruction. During the HALT state, the HALT line is set. The PC is increased before the interrupt routine is called.

2.4.5 Where interrupts are accepted

During the execution of instructions, interrupts won't be accepted. Only *between* instructions. This is also true for prefixed instructions.

Directly after an `EI` or `DI` instruction, interrupts aren't accepted. They're accepted again after the instruction after the `EI` (`RET` in the following example). So for example, look at this MSX2 routine that reads a scanline from the keyboard:

```
    LD C, A
    DI
    IN A, ($AA)
    AND $F0
    ADD A, C
    OUT ($AA), A
    EI
    IN A, ($A9)
    RET
```

You can assume that there never is an interrupt after the `EI`, before the `IN A,($A9)` - which would be a problem because the MSX interrupt routine reads the keyboard too.

Using this feature of `EI`, it is possible to check whether it is true that interrupts are never accepted during instructions:

```
    DI
    make sure interrupt is active
    EI
    insert instruction to test
InterruptRoutine:
    store PC where interrupt was accepted
    RET
```

And yes, for all instructions, including the prefixed ones, interrupts are never accepted during an instruction. Only after the tested instruction. Remember that block instructions simply re-execute themselves (by decreasing the PC with 2) so an interrupt is accepted after each iteration.

Another predictable test: at the "insert instruction to test" insert a large sequence of `EI` instructions. Of course, during the execution of the `EI` instructions, no interrupts are accepted.

But now for the interesting stuff. ED or CB make up instructions, so interrupts are accepted after them. But DD and FD are prefixes, which only slightly affects the next opcode. If you test a large sequence of DDs or FDs, the same happens as with the EI instruction: no interrupts are accepted during the execution of these sequences.

This makes sense if you think of DD and FD as a prefix that sets the "use IX instead of HL" or "use IY instead of HL" flag. If an interrupt was accepted after DD or FD, this flag information would be lost, and:

```
    DD 21 00 00    LD IX, 0
```

could be interpreted as a simple LD HL,0 if the interrupt was after the last DD. Which never happens, so the implementation is correct. Although I haven't tested this, as I imagine the same holds for NMI interrupts.

Also see section 3.12, page 119 for details on handling interrupts on ZX Spectrum Next.

2.5 Timing and R register

2.5.1 R register and memory refresh

During every first machine cycle (beginning of instruction or part of it - prefixes have their own M1 two), the memory refresh cycle is issued. The whole IR register is put on the address bus, and the $\overline{\texttt{RFSH}}$ pin is lowered. It's unclear whether the Z80 increases the R register before or after putting IR on the bus.

The R register is increased at every first machine cycle (M1). Bit 7 of the register is never changed by this; only the lower 7 bits are included in the addition. So bit 7 stays the same, but it can be changed using the LD R,A instruction.

Instructions without a prefix increase R by one. Instructions with an ED, CB, DD, FD prefix, increase R by two, and so do the DDCBxxxx and FDCBxxxx instructions (weird enough). Just a stray DD or FD increases the R by one. LD A,R and LD R,A access the R register after it is increased by the instruction itself.

Remember that block instructions simply decrement the PC with two, so the instructions are re-executed. So LDIR increases R by BC × 2 (note that in the case of BC = 0, R is increased by $10000 × 2, effectively 0).

Accepting a maskable or non-maskable interrupt increases the R by one.

After a hardware reset, or after power on, the R register is reset to 0.

That should cover all there is to say about the R register. It is often used in programs for a random value, which is good but of course not truly random.

2.6 Errors in Official Documentation

Some official Zilog documentation contains errors. Not every documentation has all of these mistakes, so your milage may vary, but these are just things to look out for.

- The flag affection summary table shows that `LDI/LDIR/LDD/LDDR` instructions leave the SF and ZF in an undefined state. This is not correct; the SF and ZF flags are unaffected.
- Similarly, the same table shows that `CPI/CPIR/CPD/CPDR` leave the SF and HF flags in an undefined state. Not true, they are affected as defined elsewhere in the documentation.
- Also, the table says about `INI/OUTD`/etc "Z=0 if B <> 0 otherwise Z=0"; of course the latter should be Z=1.
- The `INI/INIR/IND/INDR/OUTI/OUTD/OTIR/OTDR` instructions do affect the CF flag (some official documentation says they leave it unaffected, important!) and the NF flag isn't always set but may also be reset (see 2.3.3, page 17 for exact operation).
- When an NMI is accepted, the `IFF1` isn't copied to `IFF2`. Only `IFF1` is reset.
- In the 8-bit Load Group, the last two bits of the second byte of the `LD r,(IX + d)` opcode should be 10 and not 01.
- In the 16-bit Arithmetic Group, bit 6 of the second byte of the `ADD IX,pp` opcode should be 0, not 1.
- `IN x,(C)` resets the HF flag, it never sets it. Some documentation states it is set according to the result of the operation; this is impossible since no arithmetic is done in this instruction.

Note: In zilog's own z80cpu_um.pdf document, there are a lot of errors, some are very confusing, so I'll mention the ones I have found here:

- Page 21, figure 2 says "the Alternative Register Set contains 2 `B'` registers"; this should of course be `B'` and `C'`.
- Page 26, figure 16 shows very convincingly that "the least significant bit of the address to read for Interrupt Mode 2 is always 0". I have tested this and it is not correct, it can also be 1, in my test case the bus contained `$FF` and the address that was read did not end in `$FE` but was `$FF`.

Chapter 3

ZX Spectrum Next

With increased CPU speeds, more memory, better graphics, hardware sprites and tiles, to mention just some of the most obvious, ZX Spectrum Next is an exciting platform for the retro programmer.

This chapter represents the bulk of the book. Each topic is discussed in its own section. While the sections are laid out in order - later sections sometimes rely on, or refer to the topics discussed earlier, there's no need to go through them in an orderly fashion. Each section should be quite usable by itself as well. All topics discussed elsewhere are referenced, so you can quickly jump there if needed. If using PDF you can click on the section number to go straight to it. With a printed book though, turning the pages gives you a chance to land on something unrelated, but equally interesting, thus learning something new almost by accident.

One more thing worth mentioning, before leaving you on to explore, are ports and Next registers. You will find the full list in the next section. But that's just a list with a couple of examples on how to read and write them. Still, many ports and registers are described in detail at the end of the sections in which they are first mentioned. Those registers that are relevant for multiple topics are described in the first section they are mentioned in, and then referenced from other sections. Additionally, each port and register that's described in detail, has the reference to that section in the list on the following pages as a convenience. I thought for a while about how to approach this. One way would be to describe the ports and references in a single section, together with their list, and then only reference them elsewhere. But ultimately decided on the format described above for two reasons: descriptions are not comprehensive, only relevant ports and registers have this "honour". And secondly, I wanted to keep all relevant material as close together as possible.

3.1 Ports and Registers

3.1.1 Mapped Spectrum Ports

RW	Addr	Mask	Description
RW	$103B	%0001 0000 0011 1011	Sets and reads the I2C SCL line
RW	$113B	%0001 0001 0011 1011	Sets and reads the I2C SDA line
RW	$123B	%0001 0010 0011 1011	Enables layer 2 and controls paging of layer 2 screen into lower memory (see section 3.6.8, page 81)
RW	$133B	%0001 0011 0011 1011	Sends byte to serial port. Read tells if data is available in RX buffer
RW	$143B	%0001 0100 0011 1011	Reads data from serial port, write sets the baud rate
RW	$153B	%0001 0101 0011 1011	Configuration of UART interfaces
-W	$1FFD	%0001 ---- ---- --0-	Controls ROM paging and special paging options from the +2a/+3 (see section 3.2.7, page 41)
RW	$243B	%0010 0100 0011 1011	Selects active port for TBBlue/Next feature configuration (see section 3.1.3, page 34)
RW	$253B	%0010 0101 0011 1011	Reads and/or writes the selected TBBlue control register (see section 3.1.3, page 34)
RW	$303B	%0011 0000 0011 1011	Sets active sprite-attribute index and pattern-slot index, reads sprite status (see section 3.8.7, page 100)
-W	$7FFD	%01-- ---- ---- --0-	Selects active RAM, ROM, and displayed screen (see section 3.2.7, page 41)
-W	$BFFD	%10-- ---- ---- --0-	Writes to the selected register of the selected sound chip (see section 3.10.4, page 113)
-W	$DFFD	%1101 1111 1111 1101	Provides additional bank select bits for extended memory (see section 3.2.7, page 41)
R-	$FADF	%---- ---0 --0- ----	Reads buttons on Kempston Mouse
R-	$FBDF	%---- -0-1 --0- ----	X coordinate of Kempston Mouse, 0-255
R-	$FFDF	%---- -1-1 --0- ----	Y coordinate of Kempston Mouse, 0-192
-W	$FFFD	%11-- ---- ---- --0-	Controls stereo channels and selects active sound chip and sound chip channel (see section 3.10.4, page 113)

RW	Addr	Mask	Description
RW	$xx0B	%---- ---- 0000 1011	Controls Z8410 DMA chip via MB02 standard (see section 3.3.12, page 60)
R-	$xx1F	%---- ---- 0001 1111	Reads movement of joysticks using Kempston interface
RW	$xx37	%---- ---- ---- ----	Kempston interface second joystick variant and controls joystick I/O
-W	$xx57	%---- ---- 0101 0111	Uploads sprite positions, visibility, colour type and effect flags (see section 3.8.7, page 100)
-W	$xx5B	%---- ---- 0101 1011	Used to upload the pattern of the selected sprite (see section 3.8.7, page 100)
RW	$xx6B	%---- ---- 0110 1011	Controls zxnDMA chip (see section 3.3.12, page 60)
-W	$xxDF	%---- ---- --01 1111	Output to SpecDrum DAC
RW	$xxFE	%xxxx xxxx ---- ---0	Reading with particular high bytes returns keyboard status (see section 3.11.3, page 118), write changes border colour and base Spectrum audio settings (see section 3.5.6, page 71)
RW	$xxFF	%---- ---- ---- ----	Controls Timex Sinclair video modes and colours in hi-res mode. Readable when bit 2 of **Peripheral 3** $08 (page 115) is set

3.1.2 Next/TBBlue Feature Control Registers

Specific features of the Next are controlled via these register numbers, accessed through ports **TBBlue Register Select** $243B and **TBBlue Register Access** $253B (see page 34 for details). All registers can also be written to directly with the NEXTREG instruction.

RW	Port	Description
R-	$0	Identifies TBBlue board type. Should always be 10 on Next
R-	$1	Identifies core (FPGA image) version
RW	$2	Identifies type of last reset. Can be written to force reset
RW	$3	Identifies timing and machine type
-W	$4	In config mode, allows RAM to be mapped to ROM area
RW	$5	Sets joystick mode, video frequency and Scandoubler
RW	$6	Enables CPU Speed key, DivMMC, Multiface, Mouse and AY audio (see section 3.10.4, page 115)
RW	$7	Sets CPU Speed, reads actual speed
RW	$8	ABC/ACB Stereo, Internal Speaker, SpecDrum, Timex Video Modes, Turbo Sound Next, RAM contention and (un)lock 128k paging (see section 3.10.4, page 115)
RW	$9	Sets scanlines, AY mono output, sprite-id lockstep, resets DivMMC mapram and disables HDMI audio (see section 3.8.7, page 101)
RW	$0A	Mouse buttons and DPI config
R-	$0E	Identifies core (FPGA image) version (sub minor number)
RW	$10	Used within the Anti-brick system
RW	$11	Sets video output timing variant (see section 3.3.12, page 60)
RW	$12	Sets the bank number where Layer 2 video memory begins (see section 3.6.8, page 81)
RW	$13	Sets the bank number where the Layer 2 shadow screen begins
RW	$14	Sets the transparent colour for Layer 2, ULA and LoRes pixel data
RW	$15	Enables/disables sprites and Lores Layer, and chooses priority of sprites and Layer 2 (see section 3.7.6, page 89)
RW	$16	Sets X pixel offset used for drawing Layer 2 graphics on the screen (see section 3.6.8, page 82)
RW	$17	Sets Y offset used when drawing Layer 2 graphics on the screen (see section 3.6.8, page 83)
RW	$18	Sets and reads clip-window for Layer 2 (see section 3.6.8, page 83)
RW	$19	Sets and reads clip-window for Sprites (see section 3.8.7, page 101)
RW	$1A	Sets and reads clip-window for ULA/LoRes layer
RW	$1B	Sets and reads clip-window for Tilemap (see section 3.7.6, page 89)
RW	$1C	Controls (resets) the clip-window registers indices (see section 3.6.8, page 83)
R-	$1E	Holds the MSB of the raster line currently being drawn
R-	$1F	Holds the eight LSBs of the raster line currently being drawn

RW	Port	Description
RW	$22	Controls the timing of raster interrupts and the ULA frame interrupt (see section 3.12.4, page 125)
RW	$23	Holds the eight LSBs of the line on which a raster interrupt should occur (see section 3.12.4, page 125)
RW	$26	Pixel X offset (0-255) to use when drawing ULA Layer
RW	$27	Pixel Y offset (0-191) to use when drawing ULA Layer
RW	$28	PS/2 Keymap address MSB, read (pending) first byte of palette colour
-W	$29	PS/2 Keymap address LSB
-W	$2A	High data to PS/2 Keymap (MSB of data in bit 0)
-W	$2B	Low eight LSBs of PS/2 Keymap data
RW	$2C	DAC B mirror, read current I2S left MSB
RW	$2D	SpecDrum port 0xDF / DAC A+D mirror, read current I2S LSB
RW	$2E	DAC C mirror, read current I2S right MSB
RW	$2F	Sets the pixel offset (two high bits) used for drawing Tilemap graphics on the screen (see section 3.7.6, page 90)
RW	$30	Sets the pixel offset (eight low bits) used for drawing Tilemap graphics on the screen (see section 3.7.6, page 90)
RW	$31	Sets the pixel offset used for drawing Tilemap graphics on the screen (see section 3.7.6, page 90)
RW	$32	Pixel X offset (0-255) to use when drawing LoRes Layer
RW	$33	Pixel Y offset (0-191) to use when drawing LoRes Layer
RW	$34	Selects sprite index 0-127 to be affected by writes to other Sprite ports (and mirrors) (see section 3.8.7, page 101)
-W	$35	Writes directly into byte 1 of port $xx57 (see section 3.8.7, page 102)
-W	$36	Writes directly into byte 2 of port $xx57 (see section 3.8.7, page 102)
-W	$37	Writes directly into byte 3 of port $xx57 (see section 3.8.7, page 102)
-W	$38	Writes directly into byte 4 of port $xx57 (see section 3.8.7, page 102)
-W	$39	Writes directly into byte 5 of port $xx57 (see section 3.8.7, page 103)
RW	$40	Chooses a palette element (index) to manipulate with (see section 3.4.5, page 63)
RW	$41	Use to set/read 8-bit colours of the ULANext palette (see section 3.4.5, page 64)
RW	$42	Specifies mask to extract ink colour from attribute cell value in ULANext mode
RW	$43	Enables or disables Enhanced ULA interpretation of attribute values and toggles active palette (see section 3.4.5, page 65)
RW	$44	Sets 9-bit (2-byte) colours of the Enhanced ULA palette, or to read second byte of colour (see section 3.4.5, page 65)

RW	Port	Description
RW	$4A	8-bit colour to be used when all layers contain transparent pixel (see section 3.4.5, page 66)
RW	$4B	Index of transparent colour in sprite palette (see section 3.8.7, page 104)
RW	$4C	Index of transparent colour in Tilemap palette (see section 3.7.6, page 90)
RW	$50	Selects the 8k-bank stored in 8k-slot 0 (see section 3.2.7, page 42)
RW	$51	Selects the 8k-bank stored in 8k-slot 1 (see section 3.2.7, page 42)
RW	$52	Selects the 8k-bank stored in 8k-slot 2 (see section 3.2.7, page 42)
RW	$53	Selects the 8k-bank stored in 8k-slot 3 (see section 3.2.7, page 42)
RW	$54	Selects the 8k-bank stored in 8k-slot 4 (see section 3.2.7, page 42)
RW	$55	Selects the 8k-bank stored in 8k-slot 5 (see section 3.2.7, page 42)
RW	$56	Selects the 8k-bank stored in 8k-slot 6 (see section 3.2.7, page 42)
RW	$57	Selects the 8k-bank stored in 8k-slot 7 (see section 3.2.7, page 42)
-W	$60	Used to upload code to the Copper (see section 3.9.4, page 109)
RW	$61	Holds low byte of Copper control bits (see section 3.9.4, page 109)
RW	$62	Holds high byte of Copper control flags (see section 3.9.4, page 109)
-W	$63	Used to upload code to the Copper in 16-bit chunks (see section 3.9.4, page 109)
RW	$64	Offset numbering of raster lines in copper/interrupt/active register (see section 3.12.4, page 125)
RW	$68	Disable ULA, controls ULA mixing/blending, enable ULA+ (see section 3.7.6, page 91)
RW	$69	Layer2, ULA shadow, Timex $FF port (see section 3.6.8, page 84)
RW	$6A	LoRes Radastan mode
RW	$6B	Controls Tilemap mode (see section 3.7.6, page 91)
RW	$6C	Default tile attribute for 8-bit only maps (see section 3.7.6, page 92)
RW	$6E	Base address of the 40x32 or 80x32 tile map (see section 3.7.6, page 92)
RW	$6F	Base address of the tiles' graphics (see section 3.7.6, page 92)
RW	$70	Layer 2 resolution, palette offset (see section 3.6.8, page 84)
RW	$71	Sets pixel offset for drawing Layer 2 graphics on the screen (see section 3.9.4, page 109)
-W	$75	Same as register $35 plus increments $34 (see section 3.8.7, page 104)
-W	$76	Same as register $36 plus increments $34 (see section 3.8.7, page 104)
-W	$77	Same as register $37 plus increments $34 (see section 3.8.7, page 104)
-W	$78	Same as register $38 plus increments $34 (see section 3.8.7, page 104)
-W	$79	Same as register $39 plus increments $34 (see section 3.8.7, page 104)
RW	$7F	8-bit storage for user
RW	$80	Expansion bus enable/config
RW	$81	Expansion bus controls

RW	Port	Description
RW	$82	Enabling internal ports decoding bits 0-7 register
RW	$83	Enabling internal ports decoding bits 8-15 register
RW	$84	Enabling internal ports decoding bits 16-23 register
RW	$85	Enabling internal ports decoding bits 24-31 register
RW	$86	When expansion bus is enabled: internal ports decoding mask bits 0-7
RW	$87	When expansion bus is enabled: internal ports decoding mask bits 8-15
RW	$88	When expansion bus is enabled: internal ports decoding mask bits 16-23
RW	$89	When expansion bus is enabled: internal ports decoding mask bits 24-31
RW	$8A	Monitoring internal I/O or adding external keyboard
RW	$8C	Enable alternate ROM or lock 48k ROM
RW	$8E	Control classic Spectrum memory mapping
RW	$90-93	Enables GPIO pins output
RW	$98-9B	GPIO pins mapped to Next Register
RW	$A0	Enable Pi peripherals: UART, Pi hats, I2C, SPI
RW	$A2	Pi I2S controls
RW	$A3	Pi I2S clock divide in master mode
RW	$A8	ESP WiFi GPIO output
RW	$A9	ESP WiFi GPIO read/write
R-	$B0	Read Next keyboard compound keys separately (see section 3.11.3, page 118)
R-	$B1	Read Next keyboard compound keys separately (see section 3.11.3, page 118)
RW	$B2	DivMMC trap configuration
RW	$B4	DivMMC trap configuration
RW	$C0	Interrupt Control (see section 3.12.4, page 126)
RW	$C2	NMI Return Address LSB (see section 3.12.4, page 126)
RW	$C3	NMI Return Address MSB (see section 3.12.4, page 126)
RW	$C4	Interrupt Enable 0 (see section 3.12.4, page 126)
RW	$C5	Interrupt Enable 1 (see section 3.12.4, page 126)
RW	$C6	Interrupt Enable 2 (see section 3.12.4, page 127)
RW	$C8	Interrupt Status 0 (see section 3.12.4, page 127)
RW	$C9	Interrupt Status 1 (see section 3.12.4, page 127)
RW	$CA	Interrupt Status 2 (see section 3.12.4, page 128)
RW	$CC	DMA Interrupt Enable 0 (see section 3.12.4, page 128)
RW	$CD	DMA Interrupt Enable 1 (see section 3.12.4, page 128)
RW	$CE	DMA Interrupt Enable 2 (see section 3.12.4, page 129)
-W	$FF	Turns debug LEDs on and off on TBBlue implementations that have them

3.1.3 Accessing Registers

Writing to Spectrum Ports

When writing to one of the lower 256 ports, `OUT (n),A` instruction is used. For example to write the value of `43` to peripheral device mapped to port `$15`:

```
    LD A, 43           ; we want to write 43
    OUT ($15), A       ; writes value of A to port $15
```

To write using full 16-bit address, `OUT (C),r` instruction is used instead. Example of writing a byte to serial port using **UART TX** `$133B`:

```
    LD A, 42           ; we want to write 42
    LD BC, $133B       ; we want to write to port $133B
    OUT (C), A
```

The difference between the two speed-wise is tangible: first example requires only 18 t-states (7+11) while second 29 (7+10+12).

Reading from Spectrum Ports

Reading also uses the same approach as on original Spectrums - for the lower 256 ports `IN A,(n)` is used. For example reading a byte from port `$15`:

```
    LD A, 0      ; perhaps not strictly required, but good idea
    IN A, ($15)  ; read byte from port $15 to A
```

Note how the accumulator `A` is cleared before accessing the port. With `IN A,(n)`, the 16-bit address is composed from `A` forming high byte and `n` low byte.

Let's see how we can use this for reading from 16-bit ports - we have two options: we can either use `IN A,(n)` or `IN r,(C)`. Example of both, reading a byte from serial port:

```
LD BC, $143B  ; read $143B port
IN A, (C)     ; read byte to A
```

```
LD A, $14     ; high byte
IN A, ($3B)   ; read byte to A
```

Both have the same result. The difference speed-wise is 22 t-states (10+12) vs 18 (7+11). Not by a lot, but it may add up if used frequently. However, the intent of the first code is clearer as the port address is provided in full instead of being split between two instructions.

This example nicely demonstrates a common dilemma when programming: frequently we can have readable but not as optimal code, or vice versa. But I also thought this was worth pointing out to avoid possible confusion in case you will encounter different ways in someone else's code.

Writing to Next registers

Writing values to Next/TBBlue registers occurs through **TBBlue Register Select** `$243B` and **TBBlue Register Access** `$253B` ports. It's composed from 2 steps: first we select the register via write to port `$243B`, then write the value through port `$253B`. For example writing value of `5` to Next register `$16`:

```
LD A, $16     ; register $16
LD BC, $243B  ; port $243B
OUT (C), A

LD A, 5       ; write 5
LD BC, $253B  ; to port $254B
OUT (C), A
```

```
LD A, $16     ; register $16
LD BC, $243B  ; port $243B
OUT (C), A

LD A, 5       ; write 5
INC B         ; to port $253B
OUT (C), A
```

Quite involving, isn't it? Speed-wise, first example requires 58 t-states ((7+10+12)×2) and second 6 t-states less: 52 ((7+10+12)+(7+4+12)).

The second code relies on the fact that the only difference between two port addresses is the high byte (`$24` vs `$25`). So given we already assigned `$243B` to `BC`, we can simply increment `B` to get `$253B`. Again, the intent of the first example is clearer. And again, I thought it was worth pointing out in case you will encounter both approaches and wonder...

However, we can do better. Much better, in fact, using Next `NEXTREG` instruction, which allows direct writes to given Next registers. So above examples could simply be changed to either:

```
LD A, 5         ; write 5
NEXTREG $16, A  ; to reg $16
```

```
NEXTREG $16, 5 ; write 5 to reg $16
```

The first example requires 24 t-states (7+17) while the second 20. So less than half compared to using ports. In fact, using `NEXTREG` is the preferred method of writing to Next registers!

Reading from Next Registers

Reading values from Next/TBBlue registers also occurs through `$243B` and `$253B` ports. Similar to write, read is also composed from 2 steps: first select the register with port `$243B`, then read the value from port `$253B`. For example reading a byte from Next register `$B0`:

```
LD A, $16     ; register $16
LD BC, $243B  ; port $243B
OUT (C), A    ; set port

LD BC, $253B  ; port $253B
IN A, (C)     ; read to A
```

```
LD A, $16     ; register $16
LD BC, $243B  ; port $243B
OUT (C), A    ; set port

INC B         ; port $253B
IN A, (C)     ; read to A
```

The difference is small: 51 t-states ((7+10+12)+(10+12)) vs 45 ((7+10+12)+(4+12)).

Unfortunately, we don't have faster means of reading Next registers directly as we do for writing; there is no `NEXTREG` alternative for reads.

3.2 Memory Map and Paging

ZX Spectrum Next comes with 1024K (expanded version with 2048K) of memory. But it can't see it all at once.

3.2.1 Banks and Slots

Due to its 16-bit address bus, Next can only address $2^{16} = 65.536$ bytes or 64K of memory at a time. To get access to all available memory, it's divided into smaller chunks called "banks"[1].

Next supports two interchangeable memory management models. One is inherited from the original Spectrums and clones and uses 16K banks. The other is unique to Next and uses 8K banks. Hence, addressable 64K is also divided into 16K or 8K "slots" into which banks are swapped in and out. In a way, slots are like windows into available memory.

Banks are selected by their number - the first bank is `0`, second `1` and so on. If you ever worked with arrays, banks and their numbers work the same as array data and indexes. Both 16K and 8K banks start with number `0` at the same address. So if 16K bank n is selected, then the two corresponding 8K bank numbers would be $n \times 2$ and $n \times 2 + 1$.

After startup, addressable 64K space is mapped like this:

Address	**Slots**		**Banks**		**Description**
	16K	**8K**	**16K**	**8K**	
`$0000-$1FFF`	0	0	ROM	ROM	ROM, R/W redirect by L2, IRQ, NMI
`$2000-$3FFF`		1		ROM	ROM, R/W redirect by Layer 2
`$4000-$5FFF`	1	2	5	10	Normal/shadow ULA screen, Tilemap
`$6000-$7FFF`		3		11	ULA extended attribute/graphics, Tilemap
`$8000-$9FFF`	2	4	2	4	Free RAM
`$A000-$BFFF`		5		5	Free RAM
`$C000-$DFFF`	3	6	0	0	Free RAM
`$E000-$FFFF`		7		1	Free RAM

3.2.2 Default Bank Traits

First few addressable banks have certain uses and traits:

Banks		**Description**
16K	**8K**	
0	0-1	Standard RAM, maybe used by EsxDOS. Initially mapped to `$C000-$FFFF`
1	2-3	Standard RAM, contended on 128, may be used by EsxDOS, RAM disk on NextZXOS

[1] You may also see the term "page" used instead of "bank" (in fact, that's why the process of swapping banks into slots is usually called "paging"). I also noticed sometimes 64K addressable memory is referred to as "bank". In this book, I will keep naming consistent to avoid confusion.

.

Banks		Description
16K	8K	
2	4-5	Standard RAM. Initially mapped to `$8000-$BFFF`
3	6-7	Standard RAM, contended on 128, may be used by EsxDOS, RAM disk on NextZXOS
4	8-9	Standard RAM, contended on +2/+3, RAM disk on NextZXOS
5	10-11	ULA Screen, contended except on Pentagon, cannot be used by NextBASIC commands. Initially mapped to `$4000-$7FFF`
6	12-13	Standard RAM, contended on +2/+3, RAM disk on NextZXOS
7	14-15	ULA Shadow Screen, contended except on Pentagon, NextZXOS Workspace, cannot be used by NextBASIC commands
8	16-17	Next RAM, Default Layer 2, NextZXOS screen and extra data, cannot be used by NextBASIC commands
9-10	18-21	Next RAM, Rest of default Layer 2
11-13	22-27	Next RAM, Default Layer 2 Shadow Screen

3.2.3 Memory Map

As hinted before, not all available memory is addressable by programs. The first 256K is always reserved for ROMs and firmware. Hence bank `0` starts at absolute address `$40000`:

		16K bank	8K bank	Size	Absolute Address	Description
Expanded Next	Unexpanded Next	-	-	64K	`$000000-$00FFFF`	ZX Spectrum ROM
		-	-	16K	`$010000-$013FFF`	EsxDOS ROM
		-	-	16K	`$014000-$017FFF`	Multiface ROM
		-	-	16K	`$018000-$01BFFF`	Multiface Extra ROM
		-	-	16K	`$01C000-$01FFFF`	Multiface RAM
		-	-	128K	`$020000-$03FFFF`	DivMMC RAM
		0-7	0-15	128K	`$040000-$05FFFF`	Standard 128K RAM
		8-15	16-31	128K	`$060000-$07FFFF`	Extra RAM
		16-47	32-95	512K	`$080000-$0FFFFF`	1st Extra IC RAM
		48-79	96-159	512K	`$080000-$0FFFFF`	1st Extra IC RAM
		80-111	160-223	512K	`$080000-$0FFFFF`	2st Extra IC RAM

So when swapping in, for example:

- 16K bank 20 to slot 3 and writing 10 bytes to memory `$C000` (start of 16K slot 3), we're effectively writing to absolute memory `$90000-$90009` (`$40000 + 20 × 16384`)
- 8K bank 30 to slot 5 and writing 10 bytes to memory `$A000` (start of 8K slot 5), we're effectively writing to absolute memory `$7C000-$7C009` (`$40000 + 30 × 8192`)

.

3.2.4 Legacy Paging Modes

As mentioned, Next inherits the memory management models from the Spectrum 128K/+2/+3 models and Pentagon clones. It's unlikely you will use these modes for Next programs, as Next own model is much simpler to use. They are still briefly described here though in case you will encounter them in older programs. All legacy models use 16K slots and banks.

128K Mode

Slot	0	1	2	3
Start End	`$0000` `$3FFF`	`$4000` `$7FFF`	`$8000` `$BFFF`	`$C000` `$FFFF`

↑ `ROM 0-1` (slot 0)

↑ `BANK 0-7` on 128K
`BANK 0-127` on Next (slot 3)

Allows selecting:

- 16K ROM to be visible in the bottom 16K slot (0) from 2 possible banks
- 16K RAM to be visible in the top 16K slot (3) from 8 possible banks (128 banks on Next)

Ports involved:

- **Memory Paging Control** `$7FFD` bit 4 selects ROM bank for slot 0
- **Memory Paging Control** `$7FFD` bits 2-0 select one of 8 RAM banks for slot 3
- **Next Memory Bank Select** `$DFFD` bits 3-0 are added as MSB to 2-0 from `$7FFD` to form 128 banks for slot 3 (Next specific)

See page 41 for details on ports `$7FFD` and `$DFFD`. If you are using the standard interrupt handler or OS routines, then any time you write to **Memory Paging Control** `$7FFD` you should also store the value at `$5B5C`.

+3 Normal Mode

Slot	0	1	2	3
Start End	`$0000` `$3FFF`	`$4000` `$7FFF`	`$8000` `$BFFF`	`$C000` `$FFFF`

↑ `ROM 0-3`

↑ `BANK 0-7` on 128K
`BANK 0-127` on Next

Allows selecting:

- 16K ROM to be visible in the bottom 16K slot (0) from 4 possible banks
- 16K RAM to be visible in the top 16K slot (3) from 8 possible banks (128 banks on Next)

Ports involved:

- **+3 Memory Paging Control** `$1FFD` bit 2 as LSB selects ROM bank for slot 0
- **Memory Paging Control** `$7FFD` bit 4 forms MSB selects ROM bank for slot 0
- **Memory Paging Control** `$7FFD` bits 2-0 select one of 8 RAM banks for slot 3
- **Next Memory Bank Select** `$DFFD` bits 3-0 are added as MSB to 2-0 from `$7FFD` to form 128 banks for slot 3 (Next specific)

See page 41 for details on ports `$1FFD`, `$7FFD` and `$DFFD`. If you are using the standard interrupt handler or OS routines, then any time you write to **+3 Memory Paging Control** `$1FFD` you should also store the same value at `$5B67` and any time your write to **Memory Paging Control** `$7FFD` you should also store the value at `$5B5C`.

+3 All-RAM Mode

Slot	0	1	2	3
Start	`$0000`	`$4000`	`$8000`	`$C000`
End	`$3FFF`	`$7FFF`	`$BFFF`	`$FFFF`

```
          ↑        ↑        ↑        ↑
00 =   BANK 0   BANK 1   BANK 2   BANK 3
01 =   BANK 4   BANK 5   BANK 6   BANK 7
10 =   BANK 4   BANK 5   BANK 6   BANK 3
11 =   BANK 4   BANK 7   BANK 6   BANK 3
|↓
↓Lo bit = bit 1 from $1DDF
Hi bit = bit 2 from $1DDF
```

Also called “Special Mode” or “CP/M Mode”. Allows selecting all 4 slots from limited selection of banks as shown in the table above.

Ports involved:

- **+3 Memory Paging Control** `$1FFD` bit 0 enables All-RAM (if `1`) or normal mode (`0`)
- **+3 Memory Paging Control** `$1FFD` bits 2-1 select memory configuration

See page 41 for details on port `$1FFD`. If you are using the standard interrupt handler or OS routines, then any time you write to **+3 Memory Paging Control** `$1FFD` you should also store the same value at `$5B67`.

Pentagon 512K/1024K Mode

Next also supports paging implementation from Pentagon spectrums. It’s unlikely you will ever use it on Next, so just mentioning for completness sake. You can find more information on Next Dev Wiki[2] or internet if interested.

[2] https://wiki.specnext.dev/Next_Memory_Bank_Select

3.2.5 Next MMU Paging Mode

Next MMU based paging mode is much more flexible in that it allows mapping 8K banks into any 8K slot of memory available to the CPU. This is the only mode that allows paging in all 2048K on extended Next. It's also the simplest to use - a single `NEXTREG` instruction assigning bank number to desired MMU slot register.

In this mode, 64K memory accessible to the CPU is divided into 8 slots called MMU0 through MMU7, as shown in the diagram below. Physical memory is thus divided into 96 (or 224 on expanded Next) 8K banks.

16K Slot	0		1		2		3	
8K Slot	0	1	2	3	4	5	6	7
Start	$0000	$2000	$4000	$6000	$8000	$A000	$C000	$E000
End	$1FFF	$3FFF	$5FFF	$7FFF	$9FFF	$BFFF	$DFFF	$FFFF
	↑	↑	↑	↑	↑	↑	↑	↑
	BANK	BANK	BANK	BANK	BANK	BANK	BANK	BANK
	0-255	0-255	0-255	0-255	0-255	0-255	0-255	0-255

Bank selection is set via Next registers (see page 42 for details):

- **Memory Management Slot 0 Bank** $50
- **Memory Management Slot 1 Bank** $51
- **Memory Management Slot 2 Bank** $52
- **Memory Management Slot 3 Bank** $53
- **Memory Management Slot 4 Bank** $54
- **Memory Management Slot 5 Bank** $55
- **Memory Management Slot 6 Bank** $56
- **Memory Management Slot 7 Bank** $57

While not absolutely required, it's good practice to store original slot values and then restore before exiting program or returning from subroutines.

Example of writing 10 bytes (`00 01 02 03 04 05 06 07 08 09`) to 8K bank 30 swapped in to slot 5. As mentioned before, this will effectively write to absolute memory `$7C000-$7C009`:

```
    NEXTREG $55, 30   ; swap bank 30 to slot 5

    LD DE, $A000      ; slot 5 starts at $A000
    LD A, 0           ; starting data to write
    LD B, 10          ; number of bytes to write
next:
    LD (DE), A        ; write next byte
    INC A             ; increment source byte
    INC DE            ; increment destination location
    DJNZ next
```

Note: registers **Memory Management Slot 0 Bank** $50 and **Memory Management Slot 1 Bank** $51 (page 42) have extra "functionality": ROM can be automatically paged in if otherwise nonexistent 8K page $FF is set. Low or high 8K ROM bank is automatically determined based on which 8K slot is used. This may be useful if temporarily paging RAM into the bottom 16K region and then wanting to restore back to ROM.

3.2.6 Interaction Between Paging Modes

As mentioned, legacy and Next paging modes are interchangeable. Changing banks in one will be reflected in the other. The most recent change always has priority. Again, keep in mind that legacy modes use 16K banks, therefore single bank change will affect 2 8K banks.

Paging Out ROM

ROM is usually mapped to the bottom 16K slot, addresses $0000-$3FFF. This area can only be remapped using +3 All-RAM or Next MMU-based mode. Beware though that some programs may expect to find ROM routines at fixed addresses between $0000 and $3FFF. And if default interrupt mode (IM 1) is set, Z80 will expect to find interrupt handler at the address $0038.

ULA

ULA always reads content from 16K bank 5. This is mapped to 16K slot 1 by default, addresses $4000-$7FFF. ULA will always use bank 5, regardless of which bank is mapped to slot 1, or which slot bank 5 is mapped to (or if it is mapped into any slot at all).

You can redirect ULA to read from 16K bank 7 instead (the "shadow screen", used for double-buffering), using bit 3 of **Memory Paging Control** $7FFD (page 41). However, you still need to map bank 7 into one of the slots if you want to read or write to it (that's 8K banks 14 and 15 if using MMU for paging). Read more in ULA chapter, section 3.5, page 67.

Layer 2 Paging

Layer 2 uses specific ports and registers that are can be used to change memory mapping. For example, the bottom 16K slot can be set for write and read access. Layer 2 also supports a double-buffering scheme of its own. Though any other mapping mode discussed here can be used as well. See Layer 2 chapter, section 3.6, page 73 for more details.

3.2.7 Paging Mode Ports and Registers

+3 Memory Paging Control `$1FFD`

Bit	Effect
`7-3`	Unused, use 0
`2`	In normal mode high bit of ROM selection. With low bit from bit 4 of `$7FFD`: `00` ROM0 = 128K editor and menu system `01` ROM1 = 128K syntax checker `10` ROM2 = +3DOS `11` ROM3 = 48K BASIC In special mode: high bit of memory configuration number
`1`	In special mode: low bit of memory configuration number
`0`	Paging mode: 0 = normal, 1 = special

Memory Paging Control `$7FFD`

Bit	Effect
`7-6`	Extra two bits for 16K RAM bank if in Pentagon 512K/1024K mode (see **Next Memory Bank Select** `$DFFD` below)
`5`	`1` locks pages; cannot be unlocked until next reset on regular ZX128)
`4`	128K: ROM select (`0` = 128K editor, `1` = 48K BASIC) +2/+3: low bit of ROM select (see **+3 Memory Paging Control** `$1FFD` above)
`3`	ULA layer shadow screen toggle (0 = bank 5, 1 = bank 7)
`2-0`	Bank number for slot 4 (`$C000`)

Next Memory Bank Select `$DFFD`

Bit	Effect
`7`	1 to set Pentagon 512K/1024K mode
`3-0`	Most significant bits of the 16K RAM bank selected in **Memory Paging Control** `$7FFD`

Memory Management Slot 0 Bank $50

Memory Management Slot 1 Bank $51

Memory Management Slot 2 Bank $52

Memory Management Slot 3 Bank $53

Memory Management Slot 4 Bank $54

Memory Management Slot 5 Bank $55

Memory Management Slot 6 Bank $56

Memory Management Slot 7 Bank $57

Bit	Effect
7-0	Selects 8K bank stored in corresponding 8K slot

Memory Mapping $8E

Bit	Effect
7	Access to bit 0 of **Next Memory Bank Select** $DFFD
6-4	Access to bits 2-0 of **Memory Paging Control** $7FFD
3	Read will always return 1 Write 1 to change RAM bank, 0 for no change to MMU6,7, $7FFD and $DFFD
2	0 for normal paging mode, 1 for special all-RAM mode
1	Access to bit 2 of **+3 Memory Paging Control** $1FFD
0	If bit 2 = 0 (normal mode): bit 4 of **Memory Paging Control** $7FFD If bit 2 = 1 (special mode): bit 1 of **+3 Memory Paging Control** $1FFD

Acts as a shortcut for reading and writing **+3 Memory Paging Control** $1FFD, **Memory Paging Control** $7FFD and **Next Memory Bank Select** $DFFD all at once. Mainly to simplify classic Spectrum memory mapping. Though, as mentioned, Next specific programs should prefer MMU based memory mapping.

3.3 DMA

The ZX Spectrum Next DMA (zxnDMA) is a single channel direct memory access device that implements a subset of the Z80 DMA functionality. The subset is large enough to be compatible with common uses of the similar Datagear interface available for standard ZX Spectrums but without the idiosyncracies and requirements on the order of commands.

zxnDMA defines two "ports", called "A" and "B" (port is just a word used for referring to both, source and destination). Either one can be used as a source, the other as the destination. They can be memory location or I/O port, auto-increment, auto-decrement or stay fixed. zxnDMA can operate in continuous or burst mode and implements a special feature that can force each byte transfer to take a fixed amount of time, which can be used to deliver sampled audio.

3.3.1 Programming

Since core 3.1.2, zxnDMA is mapped to **zxnDMA Port** `$xx6B` and legacy Zilog DMA to **MB02 DMA Port** `$xx0B` (see page 60 for details).

Similar to Z80 DMA, zxnDMA also has 7 write registers named `WR0-WR6`. Some of the bits are used to identify a register, while the rest represent the payload (`x` in the table below):

Reg.	Bitmask	Description
`WR0`	`0xxxxx01`	Direction, operation and port A configuration
`WR1`	`0xxxx100`	Port A configuration
`WR2`	`0xxxx000`	Port B configuration
`WR3`	`1xxxxx00`	Activation
`WR4`	`1xxxxx01`	Port B, timing and interrupt configuration
`WR5`	`10xxx010`	Ready and stop configuration
`WR6`	`1xxxxx11`	Command register

Each register can include zero or more parameters. Most often specific bits in the payload define whether and which parameters are used. Each parameter is one byte long. Parameters are written immediately after the base register byte. If multiple parameters are used, the order is specified by the bit position within the base payload. The order is from right to left, the parameter associated with bit `0` is written first, the one from bit `7` last.

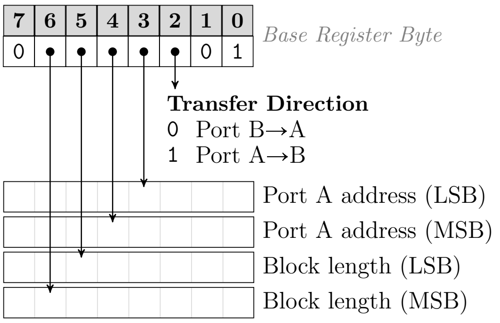
Sometimes it's also possible that a specific configuration of parameter byte requires additional bytes to be inserted. If so, these bytes are inserted immediately after their "parent" parameter byte following the same rules as indicated above. Only after all child parameters are written, we continue with other parents' parameters, if there are more. This forms a sort of recursive pattern.

After all parameters are written, we can start with a new register byte again. This is then repeated until zxnDMA program is started using `WR3` or (preferably) `WR6`. The same registers can be repeated multiple times within the same DMA program.

DMA programs can be up to 256 bytes long (but the data being transferred can be up to 64K).

3.3.2 Registers at a Glance

WR0



7	6	5	4	3	2	1	0	
0	●	●	●	●	●	0	1	*Base Register Byte*

Transfer Direction

0	Port B→A
1	Port A→B

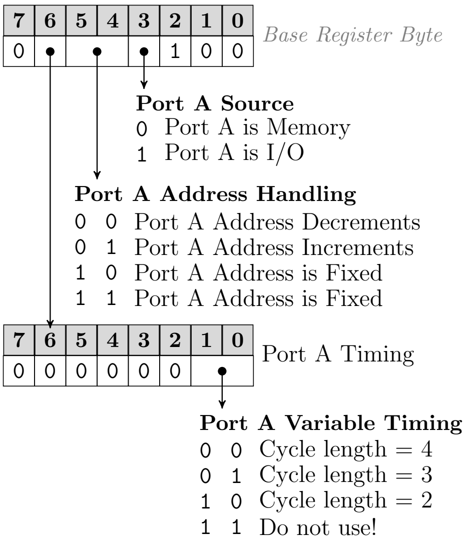
Port A address (LSB)

Port A address (MSB)

Block length (LSB)

Block length (MSB)

WR1



7	6	5	4	3	2	1	0	
0	●	●		●	1	0	0	*Base Register Byte*

Port A Source

0	Port A is Memory
1	Port A is I/O

Port A Address Handling

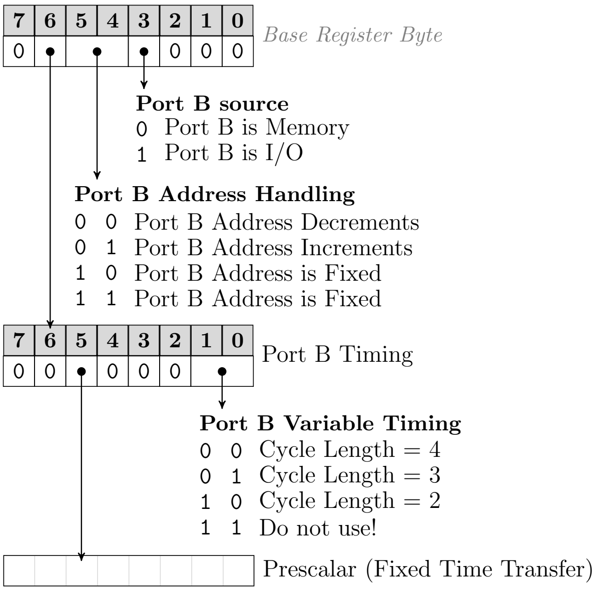
0	0	Port A Address Decrements
0	1	Port A Address Increments
1	0	Port A Address is Fixed
1	1	Port A Address is Fixed

7	6	5	4	3	2	1	0	
0	0	0	0	0	0	●		Port A Timing

Port A Variable Timing

0	0	Cycle length = 4
0	1	Cycle length = 3
1	0	Cycle length = 2
1	1	Do not use!

WR2



7	6	5	4	3	2	1	0	
0	●	●		●	0	0	0	*Base Register Byte*

Port B source

0	Port B is Memory
1	Port B is I/O

Port B Address Handling

0	0	Port B Address Decrements
0	1	Port B Address Increments
1	0	Port B Address is Fixed
1	1	Port B Address is Fixed

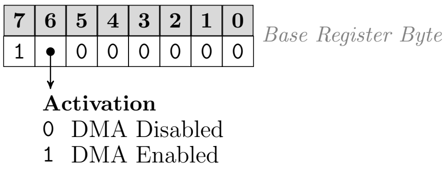
7	6	5	4	3	2	1	0	
0	0	●	0	0	0	●		Port B Timing

Port B Variable Timing

0	0	Cycle Length = 4
0	1	Cycle Length = 3
1	0	Cycle Length = 2
1	1	Do not use!

Prescalar (Fixed Time Transfer)

WR3

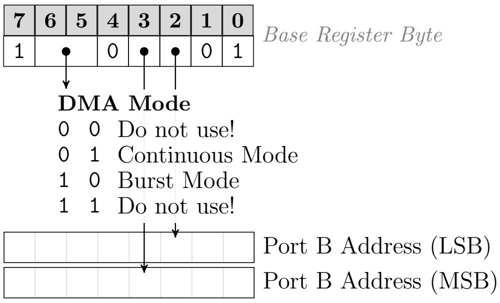


7	6	5	4	3	2	1	0	
1	●	0	0	0	0	0	0	*Base Register Byte*

Activation

0	DMA Disabled
1	DMA Enabled

WR4



7	6	5	4	3	2	1	0	
1	●		0	●	●	0	1	*Base Register Byte*

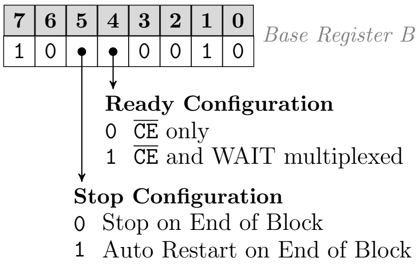
DMA Mode

0	0	Do not use!
0	1	Continuous Mode
1	0	Burst Mode
1	1	Do not use!

Port B Address (LSB)

Port B Address (MSB)

WR5



7	6	5	4	3	2	1	0	
1	0	●	●	0	0	1	0	*Base Register Byte*

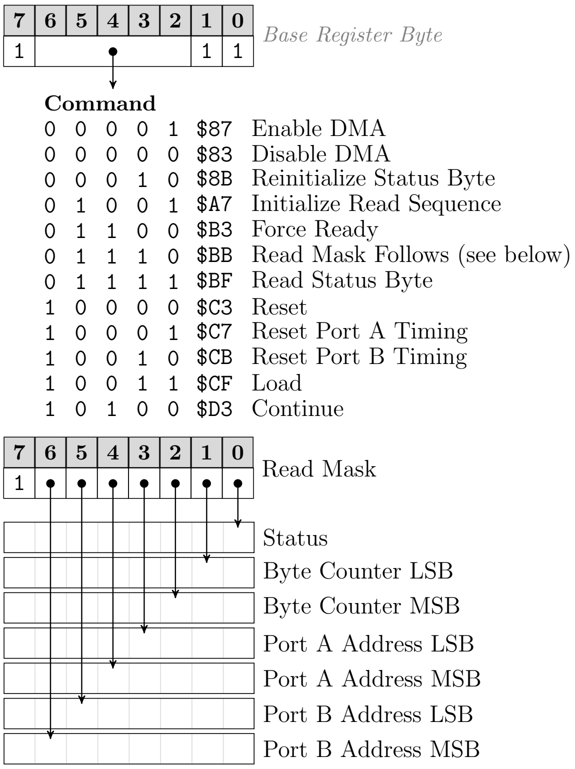
Ready Configuration

0	$\overline{\text{CE}}$ only
1	$\overline{\text{CE}}$ and WAIT multiplexed

Stop Configuration

0	Stop on End of Block
1	Auto Restart on End of Block

WR6



7	6	5	4	3	2	1	0	
1			●			1	1	*Base Register Byte*

Command

0	0	0	0	1	$87	Enable DMA
0	0	0	0	0	$83	Disable DMA
0	0	0	1	0	$8B	Reinitialize Status Byte
0	1	0	0	1	$A7	Initialize Read Sequence
0	1	1	0	0	$B3	Force Ready
0	1	1	1	0	$BB	Read Mask Follows (see below)
0	1	1	1	1	$BF	Read Status Byte
1	0	0	0	0	$C3	Reset
1	0	0	0	1	$C7	Reset Port A Timing
1	0	0	1	0	$CB	Reset Port B Timing
1	0	0	1	1	$CF	Load
1	0	1	0	0	$D3	Continue

7	6	5	4	3	2	1	0	
1	●	●	●	●	●	●	●	Read Mask

Status

Byte Counter LSB

Byte Counter MSB

Port A Address LSB

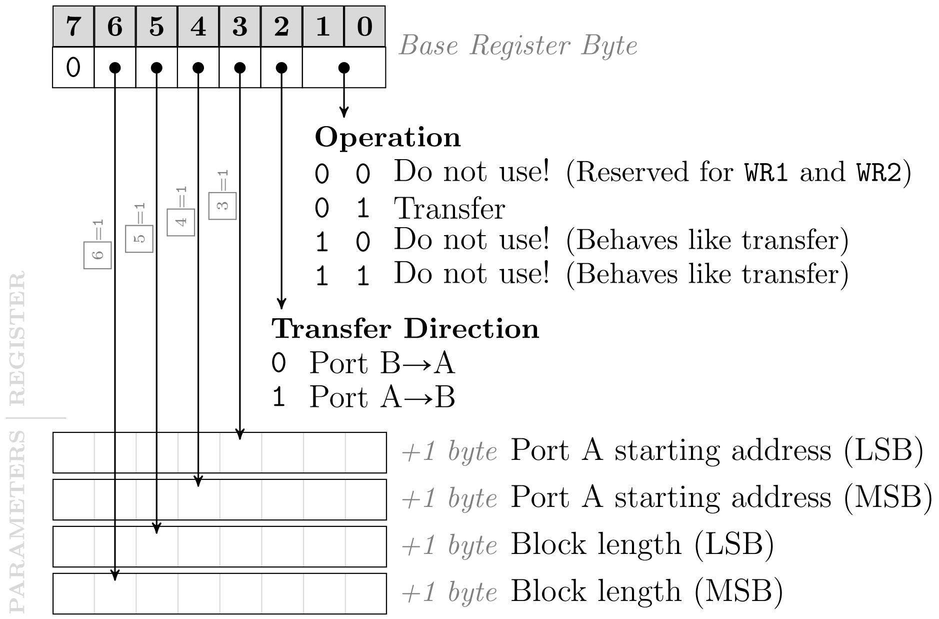
Port A Address MSB

Port B Address LSB

Port B Address MSB

3.3.3 WR0 - Direction, Operation, Port A Configuration

`WR0` specifies the direction of the transfer, the length of the data that will be transferred and port A address. Base register byte can be followed by up to four parameter bytes:


7 6 5 4 3 2 1 0
0
Base Register Byte
6=1
5=1
4=1
3=1
REGISTER
PARAMETERS
+1 byte Port A starting address (LSB)
+1 byte Port A starting address (MSB)
+1 byte Block length (LSB)
+1 byte Block length (MSB)

Operation
0 0 Do not use! (Reserved for WR1 and WR2)
0 1 Transfer
1 0 Do not use! (Behaves like transfer)
1 1 Do not use! (Behaves like transfer)

Transfer Direction
0 Port B→A
1 Port A→B

Base Register Byte

Bits `1-0`: Operation The combination of these two bits defines the type of operation the DMA program will use. While all four combinations are listed, zxnDMA only supports one at the moment - `01`:

- `00` Don't use, it conflicts with `WR1` and `WR2`.
- `01` Transfer; this is the only supported operation on zxnDMA.
- `10` Not recommended for compatibility reasons: at the moment `10` behaves exactly like `01` (transfer) on zxnDMA, but on Z80 it's "search" instead. So there is a possibility this will change in future cores.
- `11` Similar to `10`; at the moment it behaves like `01` (transfer) on zxnDMA, but on Z80 it's "search/transfer". Again, this may change in future cores.

Bit `2`: Transfer Direction Provides the destination of the data transfer:

- `0` Port B is source, port A destination.
- `1` Port A is source, port B destination.

Either port can act as source, while the other becomes the destination.

Bits `4-3`: Port A Starting Address Regardless of whether port A acts as a source or destination, we have to define its source address. To do so, set both bits to `1`. The address is then entered immediately after this byte. The address is interpreted either as memory or I/O port, based on the configuration from `WR1`.

Bits `6-5`: Transfer Length All DMA operations must have a length defined. Therefore this parameter is also required - set both bits to `1`. The length is a 16-bit value and needs to be entered at the end of the data for `WR0` register.

Port A Starting Address

If bit 3 of the base register byte is set, then LSB of port A starting address is expected as the first parameter after the base. If bit 4 is set, then MSB byte is expected. If both bits are set, then LSB needs to be written first, followed by MSB. This is in fact the most common setup because port A starting address is required for each DMA operation. And since little-endian format is used, the value forming the 16-bit starting address of port A can be written directly. For example:

```
    DW $253B         ; Port A starts at $253B
```

Or, if we have a label that points to the start of the memory:

```
    DW StartLabel    ; Port A starts on memory at this label
```

Whether the address represents a memory location or I/O port is defined with `WR1`.

Transfer Length

Setting bit 5 of the base register byte is associated with LSB of transfer length and bit 6 with MSB. If both bits are set, which is the usual case, then LSB is written first, followed by MSB. Again, 16-bit length can be written directly in this case, as shown above with port A starting address.

Notes

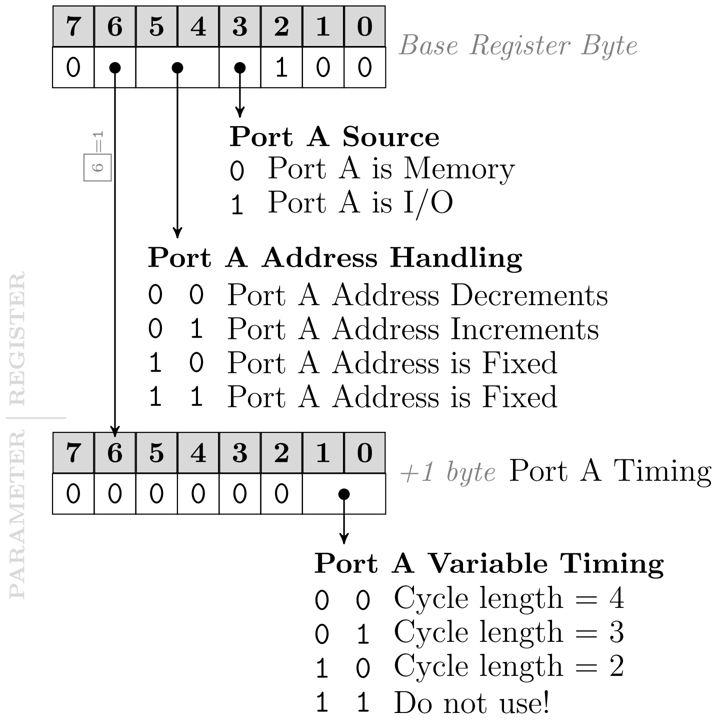
Since all parameters in `WR0` are important for DMA transfer, they are typically always included. The configuration would most often look something like this:

```
    DB %01111101     ; WR0 - append length and port A address, A->B
    DW $253B         ; Port A starts at $253B
    DW 1500          ; We will transfer 1500 bytes
```

Usually, the address and length are provided dynamically, and frequently “self-modifying code” approach is used to fill this data on the fly. We’ll discuss this in the example section later on.

3.3.4 WR1 - Port A Configuration

This is where we provide details about port A. Base register byte may be followed by one parameter, if bit 6 of the base byte is set:



7	6	5	4	3	2	1	0
0	●	●		●	1	0	0

Base Register Byte

Port A Source
0 Port A is Memory
1 Port A is I/O

Port A Address Handling
0 0 Port A Address Decrements
0 1 Port A Address Increments
1 0 Port A Address is Fixed
1 1 Port A Address is Fixed

7	6	5	4	3	2	1	0
0	0	0	0	0	0	●	

+1 byte Port A Timing

Port A Variable Timing
0 0 Cycle length = 4
0 1 Cycle length = 3
1 0 Cycle length = 2
1 1 Do not use!

Base Register Byte

Bit 3: Port A Source Specifies the type of the address for port A (the address is written with `WR0`):

- `0` Port A address is memory location.
- `1` Port A address is I/O port.

Bits 5-4: Port A Address Handling The combination of these two bits determines how port A address will be handled after each byte is transferred:

- `00` Address is decremented after byte is transferred.
- `01` Address is incremented after byte is transferred.
- `10` The address remains fixed at its starting value. This is typically used when port A is an I/O port; all bytes are sent to the same port in this case.
- `11` The same as `10`, address is fixed.

In case of decrementing or incrementing, the first byte is read from or written to the starting address for port A from `WR0`, then decrementing or incrementing begins for the second and subsequent bytes.

Bit 6: Port A Timing If bit 6 is set, the next byte written to the DMA after `WR1` base register byte is used to define port A variable timing. If bit 6 is `0`, standard Z80 timing for read and write cycles is used.

Port A Timing

This byte is expected if bit 6 of the base register byte is set. Only bits 1 and 0 are used, the rest must be set to `0`.

Bits `1-0`: Port A Timing Specifies variable timing configuration for port A that allows shortening the length of port A read or write cycles:

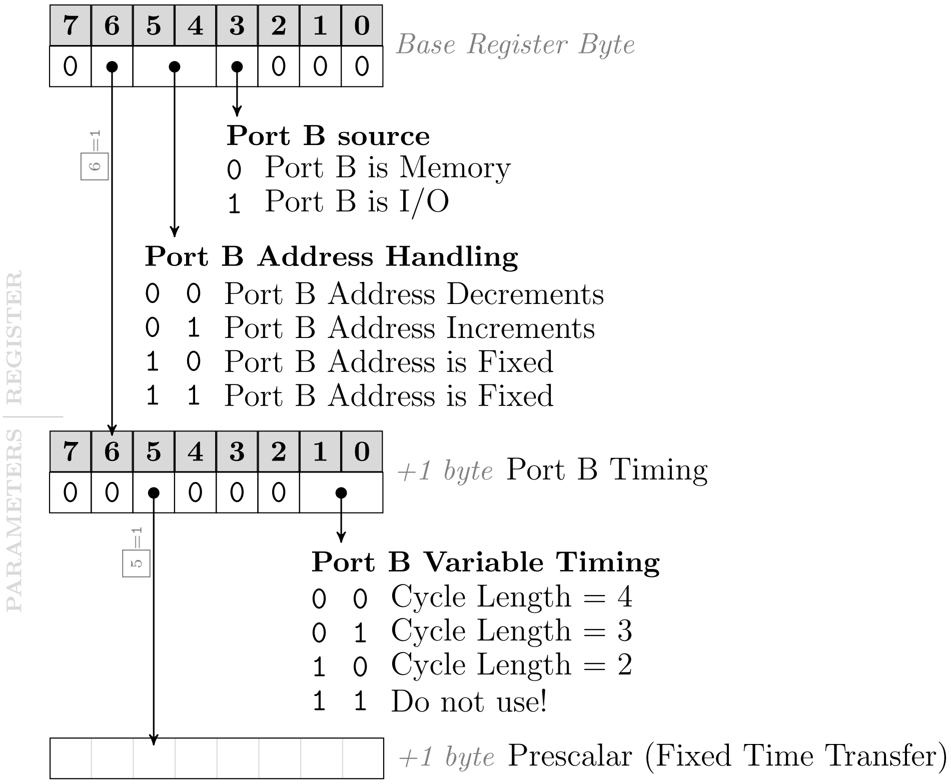
`00`	Cycle length is 4.
`01`	Cycle length is 3.
`10`	Cycle length is 2.
`11`	Do not use!

The cycle lengths are intended to selectively slow down read or write cycles for hardware that can't operate at the DMA full speed.

In contrast with Z80 DMA, zxnDMA doesn't support half-cycle timing for control signals.

3.3.5 WR2 - Port B Configuration

This register is similar to WR1 except it sets the configuration for port B. If bit 6 is set, base register byte needs to be followed by one parameter. And the configuration of this parameter may in turn require one more parameter byte to be appended.



Base Register Byte

Bit 3: Port B Source
Specifies the type of the address for port B (the address is written with WR4):

0 Port B address is memory location.
1 Port B address is I/O port.

Bits 5-4: Port B Address Handling
The combination of the two bits determines how port B address will be handled after each byte is transferred:

00 Address is decremented after byte is transferred.
01 Address is incremented after byte is transferred.
10 The address remains fixed at its starting value. This is typically used when port B is an I/O port; all bytes are sent to the same port in this case.
11 The same as 10, address is fixed.

In case of decrementing or incrementing, the first byte is read from or written to the starting address for port B from WR4, then decrementing or incrementing begins for the second and subsequent bytes.

Bit 6: Port B Timing
If bit 6 is set, the next byte written to the DMA after WR2 base register byte is used to define port B variable timing. If bit 6 is 0, standard Z80 timing for read and write cycles is used.

Port B Timing

This byte is expected if bit 6 of the base register byte is set. Only bits 5, 1 and 0 are used, the rest must be set to `0`.

Bits `1-0`: Port A Timing Specifies variable timing configuration for port A that allows shortening the length of port A read or write cycles:

- `00` Cycle length is 4.
- `01` Cycle length is 3.
- `10` Cycle length is 2.
- `11` Do not use!

The cycle lengths are intended to selectively slow down read or write cycles for hardware that can't operate at the DMA full speed.

Bit `5`: Enable Prescalar If set, then additional byte is expected with prescalar value used for fixed time transfers. See below for details.

In contrast with Z80 DMA, zxnDMA doesn't support half-cycle timing for control signals.

Prescalar - Fixed Time Transfer

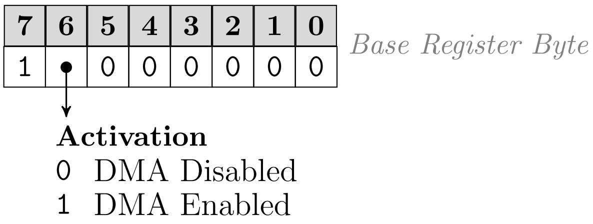
This byte is expected if bit 5 of "Port B Timing" is set. This is a feature of zxnDMA not present in Z80. If non-zero value is written, a delay will be inserted after each byte is transferred. The delay is calculated so that the total amount of time for each byte, including time required for actual transfer, is determined by the prescalar value.

To calculate prescalar value, use formula: $prescalar = 875kHz/rate$ where rate is desired frequency. For example to have a constant rate of 16kHz, prescalar needs to be set to 55 (calculation gives 54.6875 which needs to be rounded). The constant 875kHz is assumed for 28MHz system clock. But exact system clock depends on video timing selected by user (HDMI, VGA). The actual value should therefore be adjusted according to exact frequency as described with **Video Timing `$11`** (page 60).

Fixed time transfer works in continuous and burst mode (see `WR4`). In burst mode, DMA will give up any waiting time to the CPU so it can run while DMA is idle.

3.3.6 WR3 - Activation

Compared to Z80 DMA, Next only uses one bit:



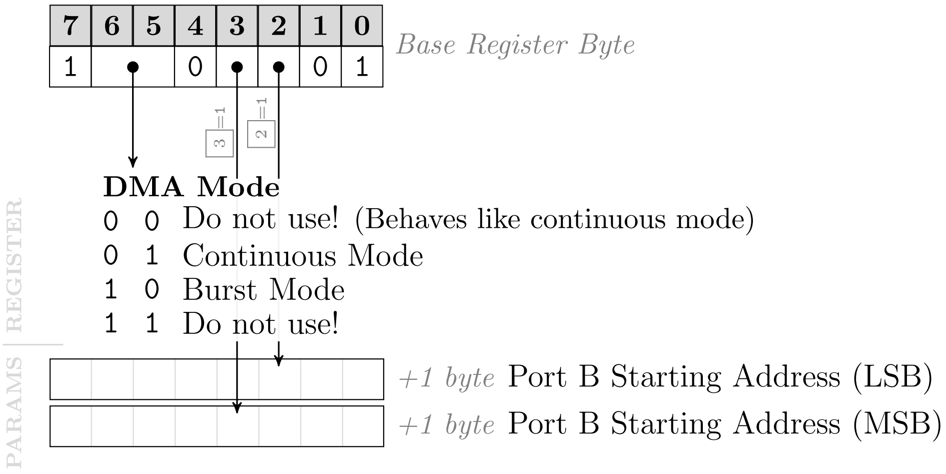
Base Register Byte

Bit 6: Enable DMA If set, DMA will be enabled, otherwise disabled. DMA should be enabled only after all other bytes are written.

Note: while enabling/disabling through `WR3` works on Next, it's recommended to use `WR6` instead.

3.3.7 WR4 - Port B, Timing, Interrupt Control

Another register where zxnDMA uses only a small portion of functionality from Z80 DMA:



Base Register Byte

Bits 3-2: Port B Starting Address Similar to bits 4-3 of `WR0`, these two bits indicate that port B address will follow. Bit 3 enables LSB and bit 2 MSB of the address. Each byte can be enabled separately, though most commonly, both are used. The address is interpreted either as memory or I/O port, based on the configuration from `WR2`.

Bits 6-5: DMA Mode Specifes operating mode for DMA:

`00` Not recommended for compatibility reasons: at the moment `00` behaves exactly like `01` (continuous mode) on zxnDMA, but on Z80 it's "byte" mode. So there is a possibility this will change in future cores.

01 Continuous mode. When the CPU starts DMA, it will run to completion without allowing the CPU to run. The CPU will only execute the next instruction after DMA completes.

10 Burst mode. In this mode, DMA will let the CPU run if either port is not ready. With zxnDMA, this condition can only occur when operating in fixed time transfer mode, as described with `WR2`.

11 Do not use!

Port B Starting Address

This works exactly the same as port A starting address in bits 4-3 of `WR0`, but for port B. Bit 3 of base register enables LSB and bit 4 MSB of the port B address. If both are set, then LSB is written before MSB. As port B address is also required for each DMA operation, both bytes are usually provided. And because the 16-bit value uses little-endian notation, it can be written directly. For example:

```
    DW $253B        ; Port B starts at $253B
```

Whether the address represents memory or I/O port is defined with `WR2`.

3.3.8 WR5 - Ready, Stop Configuration

7	6	5	4	3	2	1	0
1	0	●	●	0	0	1	0

Base Register Byte

Ready Configuration

0 $\overline{\mathtt{CE}}$ only

1 $\overline{\mathtt{CE}}$ and $\overline{\mathtt{WAIT}}$ multiplexed

Stop Configuration

0 Stop on End of Block

1 Auto Restart on End of Block

Base Register Byte

Bit 4: Ready Configuration

0 $\overline{\mathtt{CE}}$/$\overline{\mathtt{WAIT}}$ line functions only as chip enable line, allowing CPU to read and write control and status bytes when DMA is not owning the bus.

1 This mode is implemented but currently not used in zxnDMA. This mode has an external device using the DMA's $\overline{\mathtt{CE}}$ pin to insert wait states during the DMA's transfer.

Bit 5: Stop Configuration

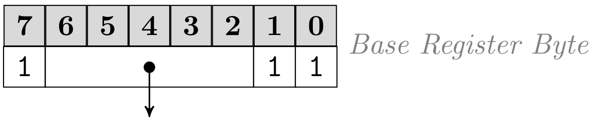
Specifies what happens when DMA reaches the end of the block:

0 DMA stops once the end of the block is reached.

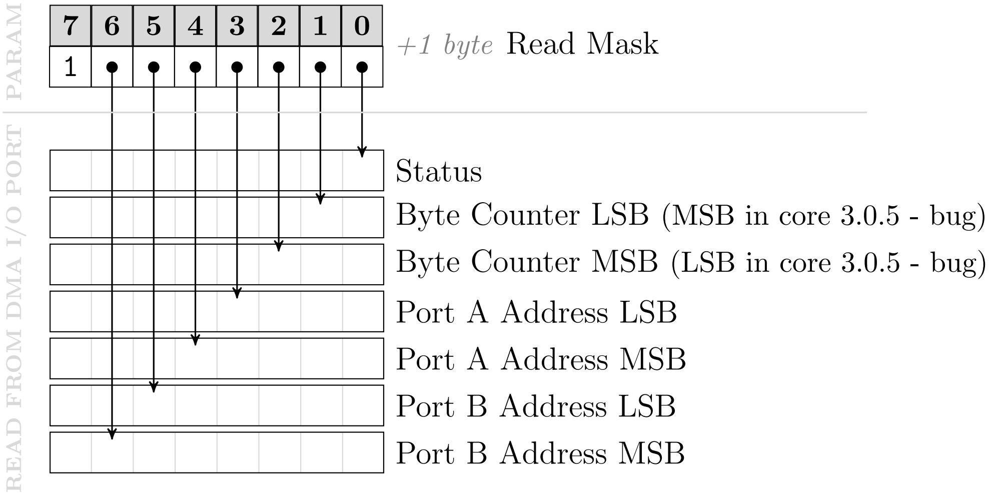
1 DMA will auto repeat at the end of the block.

3.3.9 WR6 - Command Register

This register is the most complex, but usually only a small subset of functionality is needed:



Command						
0	0	0	0	1	$87	Enable DMA
0	0	0	0	0	$83	Disable DMA
0	0	0	1	0	$8B	Reinitialize Status Byte
0	1	0	0	1	$A7	Initialize Read Sequence
0	1	1	0	0	$B3	Force Ready (irrelevant for zxnDMA)
0	1	1	1	0	$BB	Read Mask Follows (see below)
0	1	1	1	1	$BF	Read Status Byte
1	0	0	0	0	$C3	Reset
1	0	0	0	1	$C7	Reset Port A Timing
1	0	0	1	0	$CB	Reset Port B Timing
1	0	0	1	1	$CF	Load
1	0	1	0	0	$D3	Continue



Base Register Byte

WR6 uses a slightly different configuration than other registers. Bits 7, 1 and 0 are always set while bits 6-2 are used together to specify the command to execute. Only a subset of possbible bit combinations are implemented. Some of the most relevant ones:

Enable DMA This should be written as the last command of the DMA program to start executing.

Disable DMA It's recommended to use this command at the start to ensure the whole program will be written before executing.

Load This is required for DMA to copy port A and B addresses into its internal pointers. Usually, this is used just before **Enable DMA**.

Read Mask This is not one of the frequently used commands, but it's mentioned because it works differently: instead of indicating parameter bytes that will follow the register as part of the DMA program, it specifies which bytes will be read from the DMA port in the "future". See below for more details.

Read Mask

This parameter provides a bit mask that defines which bytes will be read from DMA. While read mask parameter byte has to be written immediately after base register byte, as any other parameter, reading happens separately as I/O read from the DMA port.

Bit `0`: Status Indicates that the status byte will be read from the DMA I/O port. Status byte has the following format: `00E1101T` where:

`E` is `0` if the total block length was transferred at least once, `1` otherwise.
`T` is `0` if no byte transferred yet, `1` if at least one byte was transferred.

Bits `1-2`: Byte Counter Indicates that the 16-bit number of bytes transferred so far will be read from the DMA I/O port. Bit 1 indicates read for LSB and bit 2 for MSB of the value (note: core 3.0.5 has a bug that reverses the bytes, so bit 1 indicates read for MSB and bit 2 for LSB).

Bits `3-4`: Port A Address Indicates that the current internal 16-bit port A address will be read from the I/O port. Bit 3 indicates LSB and bit 4 MSB of the value will be read. Usually the address it read as the whole 16-bit value, so both bits are set.

Bits `5-6`: Port B Address Similar to bits 3-4 above; indicates that the current internal 16-bit port B address will be read from the I/O port. Bit 3 indicates LSB and bit 4 MSB of the value will be read.

Reading From DMA I/O Port

Registers can be read via an I/O read from the DMA port `$6B` after setting the read mask. Register values are the current internal DMA values. Or in other words: the values that will be used for the next transfer operation. Once the end of the read mask is reached, further reads loop around to the first one.

3.3.10 Examples

DMA is another subject that may sound quite complicated when attempting to explain, but is really quite straighforward in practice. As you'll see in the following pages, most programs can use the same pattern with only slight changes. You can also find use of DMA to transfer data in various projects in companion code, for example `sprites` and `copper`.

Copy Data From Memory to Memory

Source is at memory address `$C000`, destination `$D000`, size 2048 bytes. We'll use port A as source, B as destination.

```
 1 CopyMemory:
 2     LD HL, .dmaProgram   ; HL = pointer to DMA program
 3     LD B, .dmaProgramSize; B = size of the code
 4     LD C, $6B            ; C = $6B (zxnDMA port)
 5     OTIR                 ; upload DMA program
 6     RET
 7
 8 .dmaProgram:
 9     DB %1’00000’11       ; WR6 - disable DMA
10
11     DB %0’11’11’1’01     ; WR0 - append length + port A address, A->B
12     DW $C000             ; WR0 par 1&2 - port A start address
13     DW 2048              ; WR0 par 3&4 - transfer length
14
15     DB %0’0’01’0’100     ; WR1 - A incr., A=memory
16
17     DB %0’0’01’0’000     ; WR2 - B incr., B=memory
18
19     DB %1’01’0’11’01     ; WR4 - continuous, append port B addres
20     DW $D000             ; WR4 par 1&2 - port B address
21
22     DB %10’0’0’0010      ; WR5 - stop on end of block, CE only
23
24     DB %1’10011’11       ; WR6 - load addresses into DMA counters
25     DB %1’00001’11       ; WR6 - enable DMA
26 .dmaProgramSize = $-.dmaProgram
```

The code includes both, the routine that copies the program into DMA memory and the program itself.

Copy routine relies on `OTIR` to upload the program to DMA memory through port `$xx6B`. `OTIR` does all the heavy lifting for us - continuously copies bytes until `B` becomes 0. Perhaps worth noting: `.dmaProgramSize` is used to establish the length of the program that will be sent. This way we can safely add or remove instructions and it will continue to work (as long as the size stays within 256 bytes).

The program itself, lines 9-25, should be self-explanatory. But we'll go through it anyway since this is how most DMA programs look like:

- Line 9 uses `WR6` with bits 6-2 set to `%00000` which corresponds to command "Disable DMA". It's good practice to disable DMA before uploading the rest of the program.
- Next comes `WR0` register in line 11; from least to most significant:
  - Bits 1-0 have the value `%01`, so "transfer" operation will be used. On zxnDMA this is the only allowed combination anyway.

  - Bit 2 establishes the direction of transfer A→B with value `1`.
  - Next two bits, 4-3, are both set, which tells DMA we want to append both bytes of the port A starting address.
  - Similarly bits 6-5 tell DMA to expect both bytes of transfer length.

- Since we enabled all 4 parameters with `WR0` register, we have to append all 4 bytes immediately afterwards. In this case, it’s a 16-bit port A starting address first, followed by the 16-bit transfer length. Since little-endian format is expected (LSB first, then MSB), we can simply use a `DW` and write the exact address in a natural and readable way. Lines 12-13 is where the parameters are written.

  Note: we could also use `DB` to write each byte separately. It emphasizes the fact two bytes are written, but obfuscates the address. Regardless, the program would then look like this:

```
12      DB $00
13      DB $C0    ; $C000
14      DB $00
15      DB $08    ; $0800 (=2048)
```

- Line 15 specifies the configuration for port A with `WR1` and line 17 for port B with `WR2`. Both use the same:
  - Bit 3 is `0` meaning port is memory.
  - Bits 5-4 are `%01` so address will increment after each byte.
  - Bit 6 is `0` so we use default cycle length.
- Next comes `WR4` in line 19 with which we specify additional configuration for port B:
  - Bits 3-2 are set, therefore both bytes of port B address are expected after the register.
  - Bits 6-5 are `%01` therefore DMA will run the program in continuous mode.
- 16-bit port B starting address follows in line 20.
- Line 22 has `WR5` register with bit 4 and 5 reset. Bit 4 tells DMA to use $\overline{\texttt{CE}}$ only and bit 5 to stop at the end of the block, after all bytes are transferred.
- Line 24 uses `WR6` with bits 6-2 set to `%10011` which tells DMA to load source and destination addresses into its own internal pointers. This command must always be included before the end of the program, otherwise DMA internal pointers may point to old values.
- And finally, once everything is configured, line 25 includes another `WR6` with bits 6-2 set to `%00001`. This corresponds to “Enable DMA”. After encountering this command, DMA will start running the program.

Note the use of apostrophes to delimit bits. This is special syntax for binary values, `sjasmplus` will strip them out, so `%1’00000’11` will become `%10000011` for example. I used it to emphasize individual bits and bit groups. You can use whichever you prefer though.

Perhaps one more thing: the dot in front of `.dmaProgram` and `.dmaProgramSize` labels makes them private within last “normal” label (`CopyMemory` in our case). This is nice if your editor supports code completion; it will not offer these labels outside this scope.

Copy Data From Memory to I/O Port

Source is at memory address `$9000`, destination is port `$5B`, size 16384 bytes. We’ll use port A as source, B as destination.

```
1  CopyMemory:
2      LD HL, .dmaProgram   ; HL = pointer to DMA program
3      LD B, .dmaProgramSize; B = size of the code
4      LD C, $6B            ; C = $6B (zxnDMA port)
5      OTIR                 ; upload DMA program
6      RET
7
8  .dmaProgram:
9      DB %1’00000’11       ; WR6 - disable DMA
10
11     DB %0’11’11’1’01     ; WR0 - append length + port A address, A->B
12     DW $9000             ; WR0 par 1&2 - port A start address
13     DW 16384             ; WR0 par 3&4 - transfer length
14
15     DB %0’0’01’0’100     ; WR1 - A incr., A=memory
16
17     DB %0’0’10’1’000     ; WR2 - B fixed, B=I/O
18
19     DB %1’01’0’11’01     ; WR4 - continuous, append port B addres
20     DW $005B             ; WR4 par 1&2 - port B address
21
22     DB %10’0’0’0010      ; WR5 - stop on end of block, CE only
23
24     DB %1’10011’11       ; WR6 - load addresses into DMA counters
25     DB %1’00001’11       ; WR6 - enable DMA
26 .dmaProgramSize = $-.dmaProgram
```

Apart from the obvious differences - source and destination addresses and size in lines 12, 13 and 20, the program is almost the same as before.

The only other difference is port B setup in line 17. Because we’re sending the bytes to an I/O port, which exists on a specific address and doesn’t change, we need to tweak `WR2` register data:

- Bit 3 is now set to indicate port B is an I/O port.
- Bits 5-4 are `%10` to indicate port B address is fixed.

With this change, port B address will remain fixed at the given value of `$5B` throughout the whole transfer, while port A address will be incremented after each byte.

While the result is quite different, the two programs share a lot of similarities. Even more so if we want to use this program multiple times, for transferring data to the same port but from another memory address or of a different size. As it stands now, we’d have to repeat the above program, only replace port A source and transfer length in lines 12-13.

Well, we can use a neat trick to have a reusable DMA program and pass in the address and length as parameters! And it’s right on the next page for easy comparison.

Using Generic Routine for DMA Transfer

The routine gets the address of the source memory through `HL` and length in `BC` register pair:

```
 1 CopyMemory:
 2     LD (.dmaPortAAddress), HL   ; modify program with actual address
 3     LD (.dmaTransferLength), BC ; modify program with actual length
 4
 5     LD HL, .dmaProgram   ; HL = pointer to DMA program
 6     LD B, .dmaProgramSize; B = size of the code
 7     LD C, $6B            ; C = $6B (zxnDMA port)
 8     OTIR                 ; upload DMA program
 9     RET
10
11 .dmaProgram:
12     DB %1’00000’11       ; WR6 - disable DMA
13
14     DB %0’11’11’1’01     ; WR0 - append length + port A address, A->B
15 .dmaPortAAddress:
16     DW 0                 ; WR0 par 1&2 - port A start address
17 .dmaTransferLength:
18     DW 0                 ; WR0 par 3&4 - transfer length
19
20     DB %0’0’01’0’100     ; WR1 - A incr., A=memory
21
22     DB %0’0’10’1’000     ; WR2 - B fixed, B=I/O
23
24     DB %1’01’0’11’01     ; WR4 - continuous, append port B address
25     DW $005B             ; WR4 par 1&2 - port B address
26
27     DB %10’0’0’0010      ; WR5 - stop on end of block, CE only
28
29     DB %1’10011’11       ; WR6 - load addresses into DMA counters
30     DB %1’00001’11       ; WR6 - enable DMA
31 .dmaProgramSize = $-.dmaProgram
```

DMA program is exactly the same, except for the two `WR0` parameters.

The labels in lines 15 and 17 are the first part of the puzzle. They are used to get the exact memory address of the first byte of the 16-bit value declared with `DW 0`. Lines 2 and 3 are where the magic happens. We take the address from the labels and load the value from the corresponding register pair over it.

That’s all there is to it. This is one of the most frequent implementations of DMA routines, you likely saw it in various code listings.

The technique is called “self-modifying code”, for obvious reasons: we are modifying the program loaded into memory. It’s a useful trick to keep in our bag, and this is just one of the simplest implementations. However, we should take care - it’s very easy to modify wrong parts which can lead to all sorts of issues. So use with care!

3.3.11 Miscellaneous

Operating Speed

The zxnDMA operates at the same speed as the CPU.

Note: at the moment of this writing, Next Dev Wiki[3] states that DMA automatically slows down if the speed exceeds 7MHz and Layer 2 display is being generated and then automatically resumes in high speed operation once Layer 2 display finishes generating. However this is no longer true on latest cores. Thanks to Alvin Albrecht for clarifying this!

DMA and Interrupts

When zxnDMA controls the bus (when transfer is in progress), Z80 can't respond to interrupts. Here's detailed explanation from Next Dev Wiki[4]:

> On the Z80, the NMI interrupt is edge triggered so if an NMI occurs the fact that it occurred is stored internally in the Z80 so that it will respond when it is woken up. On the other hand, maskable interrupts are level triggered. That is, the Z80 must be active to regularly sample the $\overline{\texttt{INT}}$ line to determine if a maskable interrupt is occurring. On the Spectrum and the ZX Next, the ULA (and line interrupt) are only asserted for a fixed amount of time, ~30 cycles at 3.5MHz. If the DMA is executing a transfer while the interrupt is asserted, the CPU will not be able to see this and it will most likely miss the interrupt.

However, when operating in burst mode, with large enough prescalar, the CPU will be able to respond to interrupts.

It's also possible to allow or prevent specific interrupters from interrupting DMA using Next specific Hardware IM2 mode. This is configured with Next registers **DMA Interrupt Enable 0** `$CC`, **DMA Interrupt Enable 1** `$CD` and **DMA Interrupt Enable 2** `$CE` (pages 128-129). See section 3.12, page 119 for more details on interrupts.

[3] `https://wiki.specnext.dev/DMA#Operating_speed`

[4] `https://wiki.specnext.dev/DMA#The_DMA_and_Interrupts`

3.3.12 DMA Ports and Registers

MB02 DMA Port $xx0B

Core 3.1.2+: Always in Zilog DMA mode for uploading legacy DMA programs. For zxnDMA, use **zxnDMA Port** $xx6B.

Older cores: this port is not supported, set bit 6 of **Peripheral 2** $06 (page 115) and use **zxnDMA Port** $xx6B.

zxnDMA Port $xx6B

Core 3.1.2+: Always in zxnDMA mode. For legacy Zilog DMA, use **MB02 DMA Port** $xx0B.

Older cores: this port behaves either in zxnDMA or legacy Zilog DMA mode based on bit 6 of **Peripheral 2** $06 (page 115): if bit 6 is 0, zxnDMA mode is enabled, otherwise Zilog DMA.

Peripheral 2 $06

See description under Sound, section 3.10.4, page 115.

Video Timing $11

Bit	Effect		
7-3	Reserved, must be 0		
2-0	Video signal timing variant		
	000	clk28=28000000	Base VGA timing
	001	clk28=28571429	VGA setting 1
	010	clk28=29464286	VGA setting 2
	011	clk28=30000000	VGA setting 3
	100	clk28=31000000	VGA setting 4
	101	clk28=32000000	VGA setting 5
	110	clk28=33000000	VGA setting 6
	111	clk28=27000000	HDMI

DMA Interrupt Enable 0 $CC

DMA Interrupt Enable 1 $CD

DMA Interrupt Enable 2 $CE

See description under Interrupts, section 3.12.4, pages 128-129.

3.4 Palette

Next greatly enhances ZX Spectrum video capabilities by offering several new ways to draw graphics on a screen. We'll see how to program each in later chapters, but let's check common behaviour first - colour management.

To draw a pixel on a screen, we need to set its colour as data in memory. Next shares implementation to other contemporary 8-bit computers - all possible colours are stored together in a palette, as an array of RGB values, and each pixel is simply an index into this array. This approach requires less memory and allows creating efficient effects such as fade to/from black, transitions from day to night, water animations etc.

3.4.1 Palette Selection

Each graphics mode or layer has not one but two palettes, each of which can be changed independently. Of course, only one of two can be active at any given time for each mode. The other can be initialized with alternate colours and can be quickly activated to achieve colour animation effects.

Active palette is set with **Enhanced ULA Control** `$43` (page 65) for ULA, Layer 2 and Sprites and **Tilemap Control** `$6B` (page 91) for Tilemap.

3.4.2 Palette Editing

Next palettes have 256 colours. All are initialized with default values, so they are usable out of the box. But it's also possible to change every colour. Regardless of the palette, the procedure to read or write colours is:

1. **Enhanced ULA Control** `$43` (page 65) selects palette which colours you want to edit
2. **Palette Index** `$40` (page 63) selects colour index that will be read or written
3. **Palette Value** `$41` (page 64) or **Enhanced ULA Palette Extension** `$44` (page 65) reads or writes data for selected colour

When writing colours, we can chose to automatically increment colour indexes after each write. Bit 7 of **Enhanced ULA Control** `$43` is used for that purpose. This works the same for both write registers (`$41` and `$44`). Colour RGB values can either be 8-bit `RRRGGGBB`, or 9-bit `RRRGGGBBB` values. Use **Palette Value** `$41` for 8-bit and **Enhanced ULA Palette Extension** `$44` for 9-bit.

Note: **Enhanced ULA Control** `$43` has two roles when working with palettes - it selects the active palette for display (out of two available - only for ULA, Layer 2 and Sprites) and selects palette for editing (for all layers, including Tilemap). Therefore care needs to be taken when updating colour entries to avoid accidentally changing the active palette for display at the same time. Depending on our program, we may first need to read the value and then only change bits affecting the palette for editing to ensure the rest of the data remains unaffected.

3.4.3 8 Bit Colours

8-bit colours are stored as `RRRGGGBB` values with 3 bits per red and green and 2 bits per blue component. Each colour is therefore stored as a single byte. **Palette Value** `$41` (page 64) is used to read or write the value.

Here's a reusable subroutine for copying `B` number of colours stored as a contiguous block in memory addressed by `HL` register, starting at the currently selected colour index:

```
Copy8BitPalette:
    LD A, (HL)             ; Load RRRGGGBB into A
    INC HL                 ; Increment to next colour entry
    NEXTREG $41, A         ; Send colour data to Next HW
    DJNZ Copy8BitPalette   ; Repeat until B=0
```

And we'd call the subroutine with something like:

```
    NEXTREG $43, %00010000 ; Auto increment, Layer 2 first palette for read/write
    NEXTREG $40, 0         ; Start copying into index 0
    LD HL, palette         ; Address to copy RRRGGGBB values from
    LD B, 255              ; Copy 255 colours
    CALL Copy8BitPalette
```

Note: we could also use DMA to transfer all the bytes. I won't show it here, but feel free to implement it as an excercise - see section 3.3 for details on programming the DMA.

3.4.4 9 Bit Colours

With 9 bits per colour, each RGB component uses full 3 bits, thus greatly increasing the available colour gamut. However, each colour needs 2 bytes in memory instead of 1. To read or write we use **Enhanced ULA Palette Extension** `$44` (page 65) register instead of `$41`. It works similarly to `$41` except that each colour requires two writes: first one stores `RRRGGGBB` part and second least significant bit of blue component. Subroutine for copying 9-bit colours:

```
Copy9BitPalette:
    LD A, (HL)             ; Load RRRGGGBB into A
    INC HL                 ; Increment to next byte
    NEXTREG $44, A         ; Send colour data to Next HW
    LD A, (HL)             ; Load remaining byte with LSB of B component into A
    INC HL                 ; Increment to next colour entry
    NEXTREG $44, A         ; Send colour data to Next HW and increment index
    DJNZ Copy9BitPalette   ; Repeat until B=0
```

Note: subroutine requires that colours are stored in 2 bytes with first containing `RRRGGGBB` part and second least significant bit of blue. Which is how typically drawing programs store a 9-bit palette anyways. The code for calling this subroutine is exactly the same as for the 8-bit colours above.

3.4.5 Palette Registers

Palette Index $40

Bit	Effect
7-0	Reads or writes palette colour index to be manipulated

Writing an index 0-255 associates it with colour set through **Palette Value** $41 or **Enhanced ULA Palette Extension** $44 (page 65) of currently selected pallette in **Enhanced ULA Control** $43 (page 65). Write also resets value of **Enhanced ULA Palette Extension** $44 (page 65) so next write will occur for first colour of the palette

While Tilemap, Layer 2 and Sprites palettes use all 256 distinct colours (with some caveats, as described in specific chapters), ULA modes work like this:

Classic ULA

Index	Colours
0-7	Ink
8-15	Bright ink
16-23	Paper
24-31	Bright paper

Border is taken from paper colours.

ULA+

Index	Colours
0-64	Ink

Paper and border are taken from **Transparency Colour Fallback** $4A (page 66).

ULANext normal mode

Index	Colours
0-127	Ink (only a subset)
128-255	Paper (only a subset)

Border is taken from paper colours. The number of active indices depends on the number of attribute bits assigned to ink and paper out of the attribute byte by **Enhanced ULA Ink Colour Mask** $42.

ULANext full-ink mode

Index	Colours
0-255	Ink

Paper and border are taken from **Transparency Colour Fallback** $4A (page 66).

Palette Value $41

Bit	Effect
7-0	Reads or writes 8-bit colour data

Format is:

7	6	5	4	3	2	1	0
R_2	R_1	R_0	G_2	G_1	G_0	B_2	B_1
Red			Green			Blue	

Least significant bit of blue is set to OR between B_2 and B_1.

Writing the value will automatically increment index in **Palette Index** $40, if auto-increment is enabled in **Enhanced ULA Control** $43 (page 65). Read doesn't auto-increment index.

Enhanced ULA Ink Colour Mask $42

Bit	Effect
7-0	The number for last ink colour entry in the palette. Only used when ULANext mode is enabled (see **Enhanced ULA Control** $43 (page 65)). Only the following values are allowed, harware behavior is unpredictable for other values:

1	Ink and paper only use 1 colour each on indices 0 and 128 respectively
3	Ink and paper use 4 colours each, on indices 0-3 and 128-131
7	Ink and paper use 8 colours each, on indices 0-7 and 128-135
15	Ink and paper use 16 colours each, on indices 0-15 and 128-143
31	Ink and paper use 32 colours each, on indices 0-31 and 128-159
63	Ink and paper use 64 colours each, on indices 0-63 and 128-191
127	Ink and paper use 128 colours each, on indices 0-127 and 128-255
255	Enables full-ink colour mode where all indices are ink. In this mode paper and border are taken from **Transparency Colour Fallback** $4A (page 66)

Default value is 7 for core 3.0 and later, 15 for older cores.

Enhanced ULA Control `$43`

Bit	Effect
`7`	`1` to disable palette index auto-increment, `0` to enable
`6-4`	Selects palette for read or write `000` ULA first palette `100` ULA second palette `001` Layer 2 first palette `101` Layer 2 second palette `010` Sprites first palette `110` Sprites second palette `011` Tilemap first palette `111` Tilemap second palette
`3`	Selects active Sprites palette (`0` = first palette, `1` = second palette)
`2`	Selects active Layer 2 palette (`0` = first palette, `1` = second palette)
`1`	Selects active ULA palette (`0` = first palette, `1` = second palette)
`0`	Enables ULANext mode if `1` (`0` after reset)

Write will also reset the index of **Enhanced ULA Palette Extension** `$44` so next write there will be considered as first byte of first colour.

Enhanced ULA Palette Extension `$44`

Bit	Effect
`7-0`	Reads or writes 9-bit colour definition

Two consequtive writes are needed:

First write:

7	6	5	4	3	2	1	0
R_2	R_1	R_0	G_2	G_1	G_0	B_2	B_1
Red			Green			Blue	

Second write:

7	6	5	4	3	2	1	0
P_r	-						B_0
$L2$	Reserved, set to `0`						B

Bit 7 of the second write must be `0` except for Layer 2 palettes where it specifies colour priority. If set to `1`, then the colour will always be on top, above all other layers, regardless of priority set with **Sprite and Layers System** `$15` (page 89). So if you need exactly the same colour with priority and non-priority, you will need to set the same data twice, to different indexes, once with priority bit `1` and then with `0`.

After the second write palette colour index in **Palette Index** `$40` (page 63) is automatically increment, if auto-increment is enabled in **Enhanced ULA Control** `$43`.

Note: reading will always return the second byte of the colour (least significant bit of blue) and will not auto-increment index. You can read `RRRGGGBB` part with **Palette Value** `$41` (page 64).

Transparency Colour Fallback $4A

Bit	Effect
7-0	8-bit colour to be used when all layers contain transparent pixel. Format is RRRGGGBB

This colour is also used for paper and border when ULANext full-ink mode is enabled - see **Enhanced ULA Ink Colour Mask $42** (page 64).

3.5 ULA Layer

Original ZX Spectrum didn't have a dedicated graphics chip. To keep the price as low as possible, screen rendering was performed by ULA ("Uncommitted Logic Array") chip.

ZX Spectrum Next inherits ULA mode. The resolution of the screen in this mode is 256×192 pixels. If we translate this to 8×8 pixels characters, it gives us 32 character columns in 24 character rows.

ULA always reads from 16K bank 5 which is assigned to the second 16K slot at addresses `$4000-$7FFF` by default. Similar to the memory configuration of other contemporary computers, pixel memory is separate from attributes/colour memory. If using default memory configuration:

ROM	RAM				
16K	16K			16K	16K
	Pixels	Attributes	(free)		
	`$4000-$57FF`	`$5800-$5AFF`	`$5B00-$7FFF`		

3.5.1 Pixel Memory

Each screen pixel is represented by a single bit, meaning 1 byte holds 8 screen pixels. So, for each line of 256 pixels, 32 bytes are needed. However, for sake of efficiency, the original Spectrum optimized screen memory layout for speed but made it inconvenient for programming.

Pixel memory is not linear but is instead divided to fill character rows line by line. The first 32 bytes of memory represent the first line of the first character row, followed by 32 bytes representing the first line of the second character row and so on until the first line of 8 character rows is filled. Then next 32 bytes of screen memory represent the second line of the first character row, again followed by the second line of the second character row, until all 8 character rows are covered:

Addr.	Ln.	Ch.	Addr.	Ln.	Ch.	Addr.	Ln.	Ch.	
$4000	0	0/0	$4100	1	0/1	$4200	2	0/2	
$4020	8	1/0	$4120	9	1/1	$4220	10	1/2	
$4040	16	2/0	$4140	17	2/1	$4240	18	2/2	
$4060	24	3/0	$4160	25	3/1	$4260	26	3/2	
$4080	32	4/0	$4180	32	4/1	$4280	33	4/2	…
$40A0	40	5/0	$41A0	41	5/1	$42A0	42	5/2	
$40C0	48	6/0	$41C0	49	6/1	$42C0	50	6/2	
$40E0	56	7/0	$41E0	57	7/1	$42E0	58	7/2	

Ln. Screen line (0-191) **Ch.** Character `<row>/<line>` (0-23/0-7)

But this is not the end of the peculiarities of Spectrum ULA mode. If you attempt to fill the screen memory byte by byte, you'll realize the top third of the screen fills in first, then middle third and lastly bottom third. The reason is, ULA mode divides the screen into 3 banks. Each bank covers 8 character rows, so 8×8×32 or 2048 bytes:

Memory Range	Screen Lines	Char. Rows
$4000 - $47FF	0 - 63	0 - 8
$4800 - $4FFF	64 - 127	9 - 16
$5000 - $57FF	128 - 191	17 - 23

In fact, to calculate the address of memory for any given (x,y) coordinate, we'd need to prepare a 16-bit value like this:

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	1	0	Y_7	Y_6	Y_2	Y_1	Y_0	Y_5	Y_4	Y_3	X_7	X_6	X_5	X_4	X_3
0	1	0	Y								X				

As you can see, X is straightforward; we simply need to take the upper 5 bits and fill them into the lower 5 bits of a 16-bit register pair. Y coordinate requires all 8 bits written into bits 12-5 of 16-bit register pair. However, notice how individual bits are scrambled. It makes incrementing address for next character row simple operation of `INC H` (assuming `HL` stores the address of the previous row), which is likely one of the reasons for such implementation. But imagine for a second how complex a Z80 program would need to be to handle all of this. Sure, nothing couple shifts and masking operations couldn't handle but still, lots of wasted CPU cycles. However, on ZX Spectrum Next we have 3 new instructions that take care of all of the complexity for us:

- `PIXELAD` calculates the address of a pixel with coordinates from `DE` register pair where `D` is Y and `E` is X coordinate and stores the memory location address into `HL` register pair for ready consumption
- `PIXELDN` takes the address of a pixel in `HL` and updates it to point to the same X coordinate but one screen line down
- `SETAE` takes X coordinate from `E` register and prepares mask in register `A` for reading or writing to ULA screen

Furthermore; each instruction only uses 8 t-states, which is far less than the corresponding Z80 assembly program would require. Somewhat naive program for drawing vertical line write from the pixel at coordinate (16,32) to (16,50):

```
    LD DE, $1020    ; Y=16, X=32
    PIXELAD         ; HL=address of pixel (E,D)
loop:
    SETAE           ; A=pixel mask
    OR (HL)         ; we'll write the pixel
    LD (HL), A      ; actually write the pixel

    INC D           ; Y=Y+1
    LD A, D         ; copy new Y coordinate to A
    CP 51           ; are we at 51 already?
    RET NC          ; yes, return

    PIXELDN         ; no, update HL to next line
    JR loop         ; continue with next pixel
```

Note: because we're updating our Y coordinate in `D` register within the loop, we could also use `PIXELAD` instead of `PIXELDN` in line 13. Both instructions require 8 T states for execution, so there's no difference performance-wise.

If we instead wanted to check if the pixel at the given coordinate is set or not, we would use `AND (HL)` instead of `OR (HL)`. For example:

```
    LD DE, $1020     ; Y=16, X=32
    PIXELAD          ; HL=address of pixel (E,D)
    SETAE            ; A=pixel mask
    AND (HL)         ; we'll read the pixel
    RET Z            ; exit if pixel is not set
```

3.5.2 Attributes Memory

Now that we know how to draw individual pixels, it's time to handle colour. Memory wise, it's stored immediately after pixel RAM, at memory locations `$5800 - $5AFF`. Each byte represents colour and attributes for 8×8 pixel block on the screen. Byte contents are as follows:

7	6	5	4	3	2	1	0
F	B	P_2	P_1	P_0	I_2	I_1	I_0
F	B	Paper			Ink		

- Bit 7: `1` to enable flashing, `0` to disables it
- Bit 6: `1` to enable bright colours, `0` for normal colours
- Bits 5-3: paper colour `0-7`
- Bits 2-0: ink colour `0-7`

Colour value `0-7` corresponds to:

Value	Binary	Colour	Bright
0	000	Black	Black
1	001	Blue	Bright blue
2	010	Red	Bright red
3	011	Magenta	Bright magenta
4	100	Green	Bright green
5	101	Cyan	Bright cyan
6	110	Yellow	Bright yellow
7	111	Gray	White

Spectrum only requires 768 bytes to configure colour and attributes for the whole screen. And memory is contiguous so it's simple to manage. However, it comes at expense of restricting to only 2 colours per character block - the reason for the (in)famous colour clash.

Note: on Next, default ULA colours can be changed, see Palette chapter 3.4, page 61 for details.

3.5.3 Border

Next inherits Spectrum border colour handling through **ULA Control Port Write** `$xxFE` (page 71). The bottom 3 bits are used to specify one of 8 possible colours (see table on the previous page for full list). Example:

```
    LD A, 1         ; Select blue colour
    OUT ($FE), A    ; Set border colour from A
```

Note: border colour is set the same way regardless of graphics mode used. However, some Layer 2 modes and Tileset may partially or fully cover the border, effectively making it invisible to the user.

3.5.4 Shadow Screen

As mentioned, ULA uses 16K bank 5 by default to determine what to show on the screen. However, it's possible to change this to bank 7 instead by using bit 3 of **Memory Paging Control** `$7FFD` (page 41). Bank 7 mode is called the "shadow" screen. It gives us two separate memory spaces for rendering ULA data and means for quickly swapping between them. It allows always drawing into inactive bank and only swapping it in when ready thus help eliminating flicker.

Note: **Memory Paging Control** `$7FFD` (page 41) only controls which of the two possible banks is being used by ULA, but it doesn't map the bank into any of the memory slots. This needs to be done by one of the paging modes as described in the Memory Map and Paging chapter, section 3.2, page 35. Using MMU, we could do something like:

```
    LD HL, $5800       ; we'll be swapping colours

    NEXTREG $52, 10    ; swap first half of 16K bank 5 to 8K slot 2
    LD A, %00000000    ; paper=black, ink=black
    LD (HL), A         ; write data to screen (immediately visible)

    NEXTREG $52, 14    ; swap first half of 16K bank 7 to 8K slot 2
    LD A, %00000101    ; paper=black, ink=cyan
    LD (HL), A         ; write to 16K bank 7 (not visible)

    LD BC, $7FFD       ; prepare port for changing layers
    LD A, %00001000    ; activate shadow layer
    OUT (C), A         ; top left char now has black background

    LD A, %00000000    ; deactivate shadow layer
    OUT (C), A         ; top left char now has cyan background
```

Remember: 16K bank 7 corresponds to 8K banks 14 and 15. And because pixel and attributes combined fit within single 8K, only single bank needs to be swapped in.

3.5.5 Enhanced ULA Modes

ZX Spectrum Next also supports several enhanced ULA modes like Timex Sinclair Double Buffering, Timex Sinclair Hi-Res and Hi-Colour, etc. However, with the presence of Layer 2 and Tilemap modes, it's unlikely these will be used when programming new software on Next. Therefore they are not described here. If interested, read more on:

`https://wiki.specnext.dev/Video_Modes`

3.5.6 ULA Registers

ULA Control Port Write `$xxFE`

Bit	Effect
`7-5`	Reserved, use `0`
`4`	EAR output (connected to internal speaker)
`3`	MIC output (saving to tape via audio jack)
`2-0`	Border colour

Note: when read with certain high byte values, **ULA Control Port Read** `$xxFE` will read keyboard status. See Keyboard, section 3.11.3, page 118 for details.

Memory Paging Control `$7FFD`

See description under Memory Map and Paging, section 3.2.7, page 41.

Palette Index `$40`

Palette Value `$41`

Enhanced ULA Ink Colour Mask `$42`

Enhanced ULA Control `$43`

Enhanced ULA Palette Extension `$44`

Transparency Colour Fallback `$4A`

See description under Palette, section 3.4.5, pages 63-66.

This page intentionally left empty

3.6 Layer 2

As we saw in the previous section, drawing with ULA graphics is much simplified on Next. But it can't eliminate the colour clash. Well, not with ULA mode at least. However, Next brings a couple of brand new graphic modes to the table, hidden behind a somewhat casual name "Layer 2". But don't let its name deceive you; Layer 2 raises Next graphics capabilities to a whole new level!

Layer 2 may appear behind or above the ULA layer. It supports different resolutions with every pixel coloured independently and memory organized sequentially, line by line, pixel by pixel. Consequently, Layer 2 requires more memory compared to ULA; each mode needs multiple 16K banks. But of course, Next has far more memory than the original Speccy ever did!

Resolution	**Colours**	**BPP**	**Memory Organization**
256×192	256	8	48K, 3 horizontal banks of 64 lines
320×256	256	8	80K, 5 vertical banks of 64 columns[5]
640×256	16	4	80K, 5 vertical banks of 128 columns[5]

3.6.1 Initialization

Drawing on Layer 2 is much simpler than ULA. But in contrast with ULA, which is always "on", Layer 2 needs to be explicitly enabled. This is done by setting bit `1` of **Layer 2 Access Port** `$123B` (page 81).

By default, Layer 2 will use 256×192 with 256 colours, supported across all Next core versions. You can select another resolution with **Layer 2 Control** `$70` (page 84). In this case you will also have to set up clip window correctly with **Clip Window Layer 2** `$18` (page 83).

3.6.2 Paging

After Layer 2 is enabled, we can start writing into memory banks. As mentioned, Layer 2 requires 3-5 contiguous 16K banks. Upon boot, Next assigns it 16K banks 8-10. However, that gets modified by NextZXOS to 9-11 soon afterwards. You can use this configuration, but it's a good idea to set it up manually to future proof our programs.

There are two pieces to the "puzzle" of Layer 2 paging: first, we need to tell the hardware which banks are used. Since banks are always contiguous we only need to write the starting 16K bank number into **Layer 2 RAM Page** `$12` (page 81). Then we need to swap the banks into one or more slots to write or read the data. Any supported mode can be used for paging, as described in section 3.2, page 35. But the recommended and simplest is MMU mode. It's recommended to only use 16K banks 9 or greater for Layer 2.

16K slot 3 (`$C000`-`$FFFF`, MMU7 and 8) is typically used for Layer 2 banks. But any other slot will work. You can use MMU registers to swap banks. Alternatively **Layer 2 Access Port** `$123B` (page 81) also allows setting up paging for Layer 2. Either way, make sure paging is reset before passing control back from Layer 2 handling code.

[5] Core 3.0.6+ only

Similar to ULA, Layer 2 can also be set up to use a double-buffering scheme. **Layer 2 RAM Shadow Page** $13 (page 82) defines starting 16K bank number for "shadow screen" (back buffer) in this case. This is mainly used when paging is set up through **Layer 2 Access Port** $123B (page 81); bit 3 is used to switch configuration between normal and shadow banks. Or we can use MMU registers instead. If we track shadow banks manually, we don't have to use register $13 at all. We still need to assign starting shadow screen bank to register **Layer 2 RAM Page** $12 (page 81) in order to make it visible on screen.

3.6.3 Drawing

In general, drawing pixels requires the programmer to:

- Determine and select bank to write to
- Calculate address of the pixel within the bank
- Write byte with colour data

All Layer 2 modes use the same approach when drawing pixels. Each pixel uses one byte (except 640×320 where each byte contains data for 2 pixels). The value is simply an index into the palette entries list. Similar to other layers, Layer 2 also has two palettes, of which only one can be active at any given time. **Enhanced ULA Control** $43 (page 65) is used to select active palette. See Palette chapter 3.4, page 61 for details on how to program palettes.

See specific modes in the following pages for examples of writing pixel data.

3.6.4 Effects

Sprite and Layers System $15 (page 89) can be used to change Layer 2 priority, effectively moving Layer 2 above or below other layers - see Tilemap chapter, section 3.7.6, page 89 for details.

We can even be more specific and only prioritize specific colours, so only pixels using those colours will appear on top while other pixels below other layers. This way we can achieve a simple depth effect. Per-pixel priority is available when writing a custom palette with **Enhanced ULA Palette Extension** $44 (page 65) (9-bit colours). See description under Palette chapter, section 3.4, page 61 for details on how to program palette.

We can also use both Layer 2 palettes to achieve simple effects. For example, certain colours can be marked with the priority flag on one palette but not on the other. When swapping palettes, pixels drawn with these colours would appear on top or below other layers. Another simple effect using both palettes could be colour animation, though it can't be very smooth with only two states.

Global Transparency $14 (page 82) register can be used to alter the transparent colour of Layer 2. This same register also affects ULA, LoRes and 1-bit ("text mode") tilemap.

Scrolling effects can be achieved by writing pixel offsets to registers **Layer 2 X Offset MSB** $71 (page 84), **Layer 2 X Offset** $16 (page 82) and **Layer 2 Y Offset** $17 (page 83).

3.6.5 256×192 256 Colour Mode

3 horizontal banks:

	0 ... 255	
0	16K BANK 0	8K BANK 0 0...31
⋮ 63		8K BANK 1 32 ...63
64	16K BANK 1	8K BANK 2 64 ...95
⋮ 127		8K BANK 3 96 ...127
128	16K BANK 2	8K BANK 4 128 ...159
⋮ 191		8K BANK 5 160 ...191

8BPP:

7	6	5	4	3	2	1	0
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
Colour index							

Banking Setup:

15	14	13	12-8	7-0
Y				X
16K		Y_{5-0}		X
8K			Y_{4-0}	X

This mode is the closest to ULA, resolution wise, so is perhaps the simplest to grasp. It's also supported across all Next core versions. Pixels are laid out from left to right and top to bottom. Each pixel uses one byte that represents an 8-bit index into the palette. 3 16K banks are needed to cover the whole screen, each holding data for 64 lines. Or, if using 8K, 6 banks, 32 lines each. Combined, colour data requires 48K of memory.

Each (x,y) coordinate pair requires 16-bits. If the upper byte is used for Y and lower for the X coordinate, together they will form exact memory location offset from the top of the first bank. But to account for bank swapping; for 16K banks, the most significant 2 bits of Y correspond to bank number and for 8K banks, top 3 bits. The rest of Y + X is memory location within the bank.

Example of filling the screen with a vertical rainbow:

```
START_16K_BANK = 9
START_8K_BANK  = START_16K_BANK*2

    ; Enable Layer 2
    LD BC, $123B
    LD A, 2
    OUT (C), A

    ; Setup starting Layer2 16K bank
    NEXTREG $12, START_16K_BANK

    LD D, 0                 ; D=Y, start at top of the screen

nextY:
    ; Calculate bank number and swap it in
    LD A, D                 ; Copy current Y to A
    AND %11100000           ; 32100000 (3 MSBs = bank number)
    RLCA                    ; 21000003
```

```
19      RLCA                    ; 10000032
20      RLCA                    ; 00000321
21      ADD A, START_8K_BANK    ; A=bank number to swap in
22      NEXTREG $56, A          ; Swap bank to slot 6 ($C000-$DFFF)
23
24      ; Convert DE (yx) to screen memory location starting at $C000
25      PUSH DE                 ; (DE) will be changed to bank offset
26      LD A, D                 ; Copy current Y to A
27      AND %00011111           ; Discard bank number
28      OR $C0                  ; Screen starts at $C000
29      LD D, A                 ; D=high byte for $C000 screen memory
30
31      ; Loop X through 0..255; we don't have to deal with bank swapping
32      ; here because it only occurs when changing Y
33      LD E, 0
34  nextX:
35      LD A, E                 ; A=current X
36      LD (DE), A              ; Use X as colour index
37      INC E                   ; Increment to next X
38      JR NZ, nextX            ; Repeat until E rolls over
39
40      ; Continue with next line or exit
41      POP DE                  ; Restore DE to coordinates
42      INC D                   ; Increment to next Y
43      LD A, D                 ; A=current Y
44      CP 192                  ; Did we just complete last line?
45      JP C, nextY             ; No, continue with next linee
```

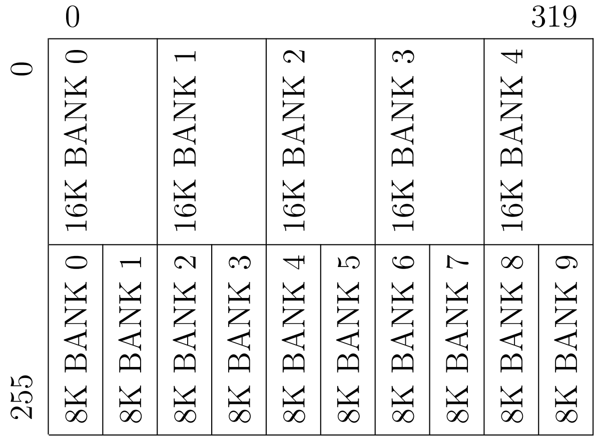
Worth noting: MMU page 6 (next register `$56`) covers memory `$C000 - $DFFF`. As we swap different 8K banks there, we're effectively changing 8K banks that are readable and writable at those memory addresses. That's why we `OR $C0` in line 24; we need to convert zero based address to `$C000` based.

We don't have to handle bank swapping on every iteration; once per 32 rows would do for this example. But the code is more versatile this way and could be easily converted into a reusable pixel setting routine.

You can find fully working example in companion code on GitHub in folder `layer2-256x192`.

3.6.6 320×256 256 Colour Mode

5 vertical banks:



16K bank contains 64 columns
8K bank contains 32 columns

8BPP:

7	6	5	4	3	2	1	0
I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
Colour index							

Banking Setup:

16	**15-8**				**7-0**
16	**15**	**14**	**13**	**12-8**	**7-0**
X_8	X_{7-0}				Y
16K			X_{5-0}		Y
8K				X_{4-0}	Y

320×256 mode is only available on Next core 3.0.6 or later. Pixels are laid out from top to bottom and left to right. Each pixel uses one byte that represents an 8-bit index into the palette. To cover the whole screen, 5 16K banks of 64 columns or 10 8K banks of 32 columns are needed. Together colour data requires 80K of memory.

In contrast with 256×192, this mode allows drawing to the whole screen, including border. In fact, you can think of it as the regular 256×192 mode with additional 32 pixel border around (32 + 256 + 32 = 320 and 32 + 192 + 32 = 256).

Addressing is more complicated though. As we need 9 bits for X and 8 for Y, we can't address all screen pixels with single 16-bit register pair. But we can use 16-bit register pair to address all pixels within each bank. From this perspective, the setup is similar to 256×192 mode, except that X and Y are reversed: if the upper byte is used for X and lower for Y, then most significant 2 bits of 16-bit register pair represent lower 2 bits of 16K bank number. And for 8K banks, the most significant 3 bits correspond to the lower 3 bits of 8K bank number. In either case, the most significant bit of the bank number arrives from the 9th bit of the X coordinate (X_8 in the table above). The rest of the X + Y is memory location within the bank.

To use this mode, we must explicitly select it with **Layer 2 Control** `$70` (page 84). We must also not forget to set clip window correctly with **Clip Window Layer 2** `$18` (page 83) and **Clip Window Control** `$1C` (page 83), as demonstrated in example below:

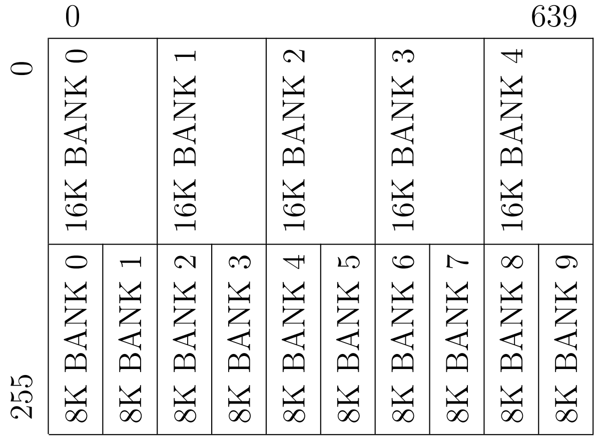
```
START_16K_BANK = 9
START_8K_BANK  = START_16K_BANK*2

RESOLUTION_X   = 320
RESOLUTION_Y   = 256

BANK_8K_SIZE   = 8192
NUM_BANKS      = RESOLUTION_X * RESOLUTION_Y / BANK_8K_SIZE
BANK_X         = BANK_8K_SIZE / RESOLUTION_Y
```

```

    ; Enable Layer 2
    LD BC, $123B
    LD A, 2
    OUT (C), A

    ; Setup starting Layer2 16K bank
    NEXTREG $12, START_16K_BANK
    NEXTREG $70, %00010000 ; 320x256 256 colour mode

    ; Setup window clip for 320x256 resolution
    NEXTREG $1C, 1         ; Reset Layer 2 clip window reg index
    NEXTREG $18, 0         ; X1; X2 next line
    NEXTREG $18, RESOLUTION_X / 2 - 1
    NEXTREG $18, 0         ; Y1; Y2 next line
    NEXTREG $18, RESOLUTION_Y - 1

    LD B, START_8K_BANK    ; Bank number
    LD H, 0                ; Colour index
nextBank:
    ; Swap to next bank, exit once all 5 are done
    LD A, B                ; Copy current bank number to A
    NEXTREG $56, A         ; Swap bank to slot 6 ($C000-$DFFF)

    ; Fill in current bank
    LD DE, $C000           ; Prepare starting address
nextY:
    ; Fill in 256 pixels of current line
    LD A, H                ; Copy colour index to A
    LD (DE), A             ; Write colour index into memory
    INC E                  ; Increment Y
    JR NZ, nextY           ; Continue with next Y until we wrap to next X

    ; Prepare for next line until bank is full
    INC H                  ; Increment colour
    INC D                  ; Increment X
    LD A, D                ; Copy X to A
    AND %00111111          ; Clear $C0 to get pure X coordinate
    CP BANK_X              ; Did we reach next bank?
    JP NZ, nextY           ; No, continue with next Y

    ; Prepare for next bank
    INC B                  ; Increment to next bank
    LD A, B                ; Copy bank to A
    CP START_8K_BANK+NUM_BANKS; Did we fill last bank?
    JP NZ, nextBank        ; No, proceed with next bank
```

You can find fully working example in companion code on GitHub in folder `layer2-320x256`.

3.6.7 640×256 16 Colour Mode

5 vertical banks:



16K bank contains 128 columns
8K bank contains 64 columns

4BPP:

7	6	5	4	3	2	1	0
I_3	I_2	I_1	I_0	I_3	I_2	I_1	I_0
Colour 1				Colour 2			

Banking Setup:

16	15-8				7-0
16	**15**	**14**	**13**	**12-8**	**7-0**
X_8	X_{7-0}				Y
16K			X_{5-0}		Y
8K				X_{4-0}	Y

640×256 mode is very similar to 320×256, except that each byte represents 2 colours instead of 1. It's also available on Next core 3.0.6 or later only. Pixels are laid out from top to bottom and left to right. Each pixel takes 4 bits, so each byte contains data for 2 pixels. Therefore division by 2 should be used to convert screen coordinate to X for address, multiplication by 2 for the other way around.

To cover the whole screen, 5 16K banks of 128 columns or 10 8K banks of 64 columns are needed. Together colour data requires 80K of memory. Similar to 320×256, this mode also covers the whole screen, including the border.

Addressing wise, this mode is the same as 320×256. Using 16-bit register pair we can't address all pixels on the screen, but we can address all pixels within each bank. Again, assuming upper byte of 16-bit register pair is used for X and lower for Y and using 9th bit of X coordinate (bit X_8 in the table above) as the most significant bit of bank number, then most significant 2 bits of 16-bit register pair represent lower 2 bits of 16K bank number. And for 8K banks, the most significant 3 bits correspond to the lower 3 bits of 8K bank number. The rest of the X + Y is memory location within the bank. Don't forget: each colour byte represents 2 screen pixels, so the memory X coordinate (as described above) needs to be multiplied by 2 to convert to screen X coordinate.

To use this mode, we must explicitly select it with **Layer 2 Control** `$70` (page 84). We must also not forget to set clip window correctly with **Clip Window Layer 2** `$18` (page 83) and **Clip Window Control** `$1C` (page 83), as demonstrated in example below:

```
START_16K_BANK = 9
START_8K_BANK  = START_16K_BANK*2

RESOLUTION_X   = 640
RESOLUTION_Y   = 256

BANK_8K_SIZE   = 8192
```

```
NUM_BANKS      = RESOLUTION_X * RESOLUTION_Y / BANK_8K_SIZE / 2
BANK_X         = BANK_8K_SIZE / RESOLUTION_Y

    ; Enable Layer 2
    LD BC, $123B
    LD A, 2
    OUT (C), A

    ; Setup starting Layer2 16K bank
    NEXTREG $12, START_16K_BANK
    NEXTREG $70, %00100000 ; 640x256 16 colour mode

    NEXTREG $1C, 1          ; Reset Layer 2 clip window reg index
    NEXTREG $18, 0
    NEXTREG $18, RESOLUTION_X / 4 - 1
    NEXTREG $18, 0
    NEXTREG $18, RESOLUTION_Y - 1

    LD B, START_8K_BANK     ; Bank number
    LD H, 0                 ; Colour index for 2 pixels
nextBank:
    ; Swap to next bank, exit once all 5 are done
    LD A, B                 ; Copy current bank number to A
    NEXTREG $56, A          ; Swap bank to slot 6 ($C000-$DFFF)

    ; Fill in current bank
    LD DE, $C000            ; Prepare starting address
nextY:
    ; Fill in 256 pixels of current line
    LD A, H                 ; Copy colour indexes for 2 pixels to A
    LD (DE), A              ; Write colour indexes into memory
    INC E                   ; Increment Y
    JR NZ, nextY            ; Continue with next Y until we wrap to next X

    ; Prepare for next line until bank is full
    INC H                   ; Increment colour index for both colours
    INC D                   ; Increment X
    LD A, D                 ; Copy X to A
    AND %00111111           ; Clear $C0 to get pure X coordinate
    CP BANK_X               ; Did we reach next bank?
    JP NZ, nextY            ; No, continue with next Y

    ; Prepare for next bank
    INC B                   ; Increment to next bank
    LD A, B                 ; Copy bank to A
    CP START_8K_BANK+NUM_BANKS; Did we fill last bank?
    JP NZ, nextBank         ; No, proceed with next bank
```

You can find fully working example in companion code on GitHub in folder `layer2-640x256`.

3.6.8 Layer 2 Registers

Layer 2 Access Port $123B

Bit	Effect
7-6	Video RAM bank select 00 First 16K of layer 2 in the bottom 16K slot 01 Second 16K of layer 2 in the bottom 16K slot 10 Third 16K of layer 2 in the bottom 16K slot 11 First 48K of layer 2 in the bottom 48K - 16K slots 0-2 (core 3.0+)
5	Reserved, use 0
4	0 (see below)
3	Use Shadow Layer 2 for paging 0 Map **Layer 2 RAM Page** $12 1 Map **Layer 2 RAM Shadow Page** $13
2	Enable Layer 2 read-only paging on 16K slot 0 (core 3.0+)
1	Layer 2 visible, see **Layer 2 RAM Page** $12 Since core 3.0 this bit has mirror in **Display Control 1** $69 (page 84)
0	Enable Layer 2 write-only paging on 16K slot 0

Note: bits 0 and 2 can be combined to get both read and write access to selected bank(s). If bit 3 is set, then paging uses shadow screen banks instead.

Since core 3.0.7, write with bit 4 set was also added:

Bit	Effect
7-5	Reserved, use 0
4	1
3	Reserved, use 0
2-0	16K bank relative offset (+0..+7) applied to Layer 2 memory mapping

With this, all 5 banks needed for 320×256 and 640×256 modes can be selected for reading or writing to 16K slot 0 (or first three slots). To use this mode, port $123B is typically written to twice, first without bit 4 to switch on read/write access, then with bit 4 set to select bank offset.

Note: read and write access to Layer 2 banks (or any other bank for that matter) can also be achieved using Next MMU registers which is arguably simpler to use. Regardless, don't forget to reset banks back to the original after you're done handling Layer 2!

Layer 2 RAM Page $12

Bit	Effect
7	Reserved, must be 0
6-0	Starting 16K bank of Layer 2

Default 256×192 mode requires 3 16K banks while new, 320×256 and 640×256 modes require 5

16K banks. Banks need to be contiguous in memory, so here we only specify the first one. Valid bank numbers are therefore `0 - 45` (`109` for 2MB RAM models) for standard mode and `0 - 43` (`107` for 2MB RAM models) for new modes.

Changes to this registers are immediately visible on screen.

Note: this register uses 16K bank numbers. If you're using 8K banks, you have to multiply this value by 2. For example, 16K bank 9 corresponds to 8K banks 18 and 19.

Layer 2 RAM Shadow Page `$13`

Bit	Effect
`7`	Reserved, must be `0`
`6-0`	Starting 16K bank of Layer 2 shadow screen

Similar to **Layer 2 RAM Page** `$12` except this register sets up starting 16K bank for Layer 2 shadow screen. The other difference is that changes to this registers are not immediately visible on screen.

Note: this register doesn't affect the shadow screen in any way besides when bit 3 is set in **Layer 2 Access Port** `$123B`. We can circumvent it completely if a manual paging scheme is used to swap banks for reading and writing.

Global Transparency `$14`

Bit	Effect
`7-0`	Sets index of transparent colour for Layer 2, ULA and LoRes pixel data (`$E3` after reset).

Layer 2 X Offset `$16`

Bit	Effect
`7-0`	Writes or reads X pixel offset used for drawing Layer 2 graphics on the screen.

This can be used for creating scrolling effects. For 320×256 and 640×256 modes, 9 bits are required; use **Layer 2 X Offset MSB** `$71` (page 84) to set it up.

Layer 2 Y Offset `$17`

Bit	Effect
`7-0`	Writes or reads Y pixel offset used for drawing Layer 2 graphics on the screen.

Valid range is:

- 256×192: `191`
- 320×256: `255`
- 640×256: `255`

Clip Window Layer 2 `$18`

Bit	Effect
`7-0`	Reads and writes clip-window coordinates for Layer 2

4 coordinates need to be set: X1, X2, Y1 and Y2. Which coordinate gets set, is determined by index. As each write to this register will also increment index, the usual flow is to reset the index to `0` in **Clip Window Control** `$1C`, then write all 4 coordinates in succession. Positions are inclusive. Furthermore, X positions are doubled for 320×256 mode, quadrupled for 640×256. Therefore, to view the whole of Layer 2, the values are:

		256×192	**320×256**	**640×256**
0	X1 position	`0`	`0`	`0`
1	X2 position	`255`	`159`	`159`
2	Y1 position	`0`	`0`	`0`
3	Y2 position	`191`	`255`	`255`

Clip Window Control `$1C`

Write:

Bit	Effect
`7-4`	Reserved, must be `0`
`3`	`1` to reset Tilemap clip-window register index
`2`	`1` to reset ULA/LoRes clip-window register index
`1`	`1` to reset Sprite clip-window register index
`0`	`1` to reset Layer 2 clip-window register index

Read:

Bit	Effect
`7-6`	Current Tilemap clip-window register index
`5-4`	Current ULA/LoRes clip-window register index
`3-2`	Current Sprite clip-window register index
`1-0`	Current Layer 2 clip-window register index

Palette Index `$40`

Palette Value `$41`

Enhanced ULA Control `$43`

Enhanced ULA Palette Extension `$44`

See description under Palette, section 3.4.5, pages 63-65.

Display Control 1 `$69`

Bit	Effect
7	1 to enable Layer 2 (alias for bit 1 in **Layer 2 Access Port** `$123B`, page 81)
6	1 to enable ULA shadow display (alias for bit 3 in **Memory Paging Control** `$7FFD`, page 41)
5-0	Alias for bits 5-0 in Timex Sinclair Video Mode Control `$xxFF`

ULA shadow screen from Bank 7 has higher priority than Timex modes.

Layer 2 Control `$70`

Bit	Effect
7-6	Reserved, must be 0
5-4	Layer 2 resolution (0 after soft reset) 00 256×192, 8BPP 01 320×256, 8BPP 10 640×256, 4BPP
3-0	Palette offset (0 after soft reset)

Layer 2 X Offset MSB `$71`

Bit	Effect
7-1	Reserved, must be 0
0	MSB for X pixel offset

This is only used for 320×256 and 640×256 modes. Together with **Layer 2 X Offset** `$16` (page 82) full 319 pixels offsets are available. For 640×256 only 2 pixel offsets are possible.

3.7 Tilemap

Tilemap is fast and effective way of displaying 8x8 pixel blocks on the screen. There are two possible resolutions available: 40x32 or 80x32 tiles. Tilemap layer overlaps ULA by 32 pixels on each side. Or in other words, similar to 320x256 and 640x256 modes of Layer 2, tilemap also covers the whole of the screen, including the border.

Tilemap is defined by 2 data structures: tile definitions and tilemap data itself.

3.7.1 Tile Definitions

Tiles are 8x8 pixels with each pixel representing an index of the colour from the currently selected tilemap palette.

Each pixel occupies 4-bits, meaning tiles can use 16 colours. However, as we'll see in the next section, it's possible to specify a 4-bit palette offset for each tile which allows us to reach all 256 colours from the palette.

A maximum of 256 tile definitions are possible, but this can be extended to 512 if needed using **Tilemap Control** `$6B` (page 91).

All tiles definitions are specified in a contiguous memory block. The offset of tile definitions memory address relative to the start of bank 5 needs to be specified with **Tile Definitions Base Address** `$6F` (page 92).

3.7.2 Tilemap Data

Tilemap data requires 2 bytes per tile:

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Palette Offset				X Mirror	Y Mirror	Rotate	ULA Mode	Tile Index							

Palette Offset 4-bit palette offset for this tile. This allows shifting colours to other 16-colour "banks" thus allowing us to reach the whole 256 colours from the palette.

X Mirror If `1`, this tile will be mirrored in X direction.

Y Mirror If `1`, this tile will be mirrored in Y direction.

Rotate If `1`, this tile will be rotated 90°clockwise.

ULA Mode If `1`, this tile will be rendered on top, if `0` below ULA display. However in 512 tile mode, this is the 8th bit of tile index.

Tile Index 8-bit tile index within the tile definitions.

However, it's possible to eliminate attributes byte by setting bit 5 in **Tilemap Control** `$6B` (page 91). This only leaves an 8-bit tile index. Tileset then only occupies half the memory. But we lose the option to specify attributes for each tile separately. Instead attributes for all tiles are taken from **Default Tilemap Attribute** `$6C` (page 92).

The offset of the tilemap data memory address relative to the start of bank 5 needs to be specified with **Tilemap Base Address** `$6E` (page 92).

3.7.3 Memory Organization

The Tilemap layer is closely tied with ULA. Memory wise, it always exists in 16K slot 5. By default, this page is loaded into 16K slot 1 `$4000-$7FFF` (examples here will assume this configuration, if you load into a different slot, you will have to adjust addresses accordingly).

If both ULA and tilemap are used, memory should be arranged to avoid overlap. Given ULA pixel and attributes memory occupied memory addresses `$4000-$5AFF`, this leaves `$5B00-$7FFF` for tilemap. If we also take into account various system variables that reside on top of ULA attributes, `$6000` should be used for starting address. This leaves us:

	40x32		**80x32**	
Bytes per tile	1	2	1	2
Bytes per tileset	1280	2560	2560	5120
Max Tile Definitions	215	175	175	95

We as programmers need to tell hardware where in the memory tilemap and tile definitions are stored. **Tilemap Base Address** `$6E` and **Tile Definitions Base Address** `$6F` registers (page 92) are used for that.

Both addresses are provided as most significant byte of the offset into memory slot 5 (which starts at `$4000`). This means we can only store data at multiples of 256 bytes. For example, if data is stored at `$6000`, the MSB offset value would be `$20` (`$6000 - $4000 = $2000`).

Generic formula to calculate MSB of the offset is: `(Address - $4000) >> 8`.

3.7.4 Combining ULA and Tilemap

ULA and Tilemap can be combined in two ways:

- Standard mode: uses bit `0` from tile's attribute byte to determine if a tile is above or below ULA. If tilemap uses 2 bytes per tile, we can specify the priority for each tile separately, otherwise we specify it for all tiles. Transparent pixels are taken into account - if the top layer is transparent, the bottom one is visible through.
- Stencil mode: only used if both, ULA and tileset are enabled. The final pixel is transparent if both, ULA and tilemap pixels are transparent. Otherwise final pixel is `AND` of both colour bits. This mode allows one layer to act as a cut-out for the other.

3.7.5 Examples

Using tilemaps is very simple. The most challenging part of my experience was finding a drawing program that would export to required formats in full. In my experience, Next Graphics[6] is the most feature-complete application for generating data for the Next. It takes images from your preferred editor and converts them into a format that can be easily consumed by Next hardware. It supports many different configurations too.

Alternatively, Remy's Sprite, Tile and Palette editor website[7] is the readily available editor and exporter. However, at the time of this writing, export is limited.

Regardless of the editor, we need 3 pieces of data: palette, tile definitions and tileset itself. In this example, they are included as binary files:

```
tilemap:
    INCBIN "tiles.map"
tilemapLength = $-tilemap

tiles:
    INCBIN "tiles.spr"
tilesLength = $-tiles

palette:
    INCBIN "tiles.pal"
paletteLength = $-palette
```

With all data in place, we can start setting up tilemap:

```
START_OF_BANK_5   = $4000
START_OF_TILEMAP  = $6000    ; Just after ULA attributes and system vars
START_OF_TILES    = $6600    ; Just after 40x32 tilemap

OFFSET_OF_MAP     = (START_OF_TILEMAP - START_OF_BANK_5) >> 8
OFFSET_OF_TILES   = (START_OF_TILES - START_OF_BANK_5) >> 8

    ; Enable tilemap mode
    NEXTREG $6B, %10100001    ; 40x32, 8-bit entries
    NEXTREG $6C, %00000000    ; palette offset, visuals

    ; Tell hardware where to find tiles
    NEXTREG $6E, OFFSET_OF_MAP ; MSB of tilemap in bank 5
    NEXTREG $6F, OFFSET_OF_TILES ; MSB of tilemap definitions
```

Above code uses couple neat preprocessing tricks to automatically calculate MSB for tilemap and tile definitions offsets. The rest is simply setting up desired behaviour using Next registers.

[6] https://github.com/infromthecold/Next-Graphics

[7] https://zx.remysharp.com/sprites/

The only remaining piece is to actually copy all the data to expected memory locations:

```
    ; Setup tilemap palette
    NEXTREG $43, %00110000    ; Auto increment, select first tilemap palette

    ; Copy palette
    LD HL, palette            ; Address of palette data in memory
    LD B, 16                  ; Copy 16 colours
    CALL Copy8BitPalette      ; Call routine for copying

    ; Copy tile definitions to expected memory
    LD HL, tiles              ; Address of tiles in memory
    LD BC, tilesLength        ; Number of bytes to copy
    CALL CopyTileDefinitions  ; Copy all tiles data

    ; Copy tilemap to expected memory
    LD HL, tilemap            ; Addreess of tilemap in memory
    CALL CopyTileMap40x32     ; Copy 40x32 tilemaps
```

We already know `Copy8BitPalette` routine from Layer 2 chapter, the other two are straightforward `LDIR` loops:

```
CopyTileDefinitions:
    LD DE, START_OF_TILES
    LDIR
    RET

CopyTileMap40x32:
    LD BC, 40*32        ; This variant always loads 40x32
    JR copyTileMap

CopyTileMap80x32:
    LD BC, 80*32        ; This variant always loads 80x32

CopyTileMap:
    LD DE, START_OF_TILEMAP
    LDIR
    RET
```

You can find fully working example in companion code on GitHub in folder `tilemap`.

3.7.6 Tilemap Registers

Sprite and Layers System `$15`

Bit	Effect
7	`1` to enable lo-res layer, `0` disable it
6	`1` to flip sprite rendering priority, i.e. sprite 0 is on top (`0` after reset)
5	`1` to change clipping to "over border" mode (doubling X-axis coordinates of clip window, `0` after reset)

4-2	Layers priority and mixing
	`000` `S L U` (Sprites are at top, Layer 2 under, Enhanced ULA at bottom)
	`001` `L S U`
	`010` `S U L`
	`011` `L U S`
	`100` `U S L`
	`101` `U L S`
	`110` Core 3.1.1+: `(U|T)S(T|U)(B+L)` blending layer and Layer 2 combined
	Older cores: `S(U+L)` colours from ULA and L2 added per R/G/B channel
	`111` Core 3.1.1+: `(U|T)S(T|U)(B+L-5)` blending layer and Layer 2 combined
	Older cores: `S(U+L-5)` similar as `110`, but per R/G/B channel `(U+L-5)`
	`110` and `111` modes: colours are clamped to `[0,7]`

1	`1` to enable sprites over border (`0` after reset)
0	`1` to enable sprite visibility (`0` after reset)

Clip Window Tilemap `$1B`

Bit	Effect
7-0	Reads and writes clip-window coordinates for Tilemap

4 coordinates need to be set: X1, X2, Y1 and Y2. Tilemap will only be visible within these coordinates. X coordinates are internally doubled for 40x32 or quadrupled for 80x32 mode. Positions are inclusive. Default values are `0`, `159`, `0`, `255`. Origin (0,0) is located 32 pixels to the top-left of ULA top-left coordinate.

Which coordinate gets set, is determined by index. As each write to this register will also increment index, the usual flow is to reset the index to 0 in **Clip Window Control** `$1C` (page 83), then write all 4 coordinates in succession.

Clip Window Control `$1C`

See description under Layer 2, section 3.6.8, page 83.

Tilemap Offset X MSB `$2F`

Bit	Effect
`7-2`	Reserved, use `0`
`1-0`	Most significant bit(s) of X offset

In 40x32 mode, meaningful range is `0-319`, for 80x32 `0-639`. Low 8-bits are stored in **Tilemap Offset X LSB** `$30`.

Tilemap Offset X LSB `$30`

Bit	Effect
`7-0`	X offset for drawing tilemap in pixels

Tilemap X offset in pixels. Meaningful range is `0-319` for 40x32 and `0-639` for 80x32 mode. To write values larger than 255, **Tilemap Offset X MSB** `$2F` is used to store MSB.

Tilemap Offset Y `$31`

Bit	Effect
`7-0`	Y offset for drawing tilemap in pixels

Y offset is `0-255`.

Palette Index `$40`

Palette Value `$41`

Enhanced ULA Control `$43`

Enhanced ULA Palette Extension `$44`

See description under Palette, section 3.4.5, pages 63-65.

Tilemap Transparency Index `$4C`

Bit	Effect
`7-5`	Reserved, must be `0`
`4-0`	Index of transparent colour into tilemap palette

The pixel index from tile definitions is compared before palette offset is applied to the upper 4 bits, so there's always one index between `0` and `15` that works as transparent colour.

ULA Control `$68`

Bit	Effect
7	`1` to disable ULA output (`0` after soft reset)
6-5	(Core 3.1.1+) Blending in SLU modes 6 & 7 `00` ULA as blend colour `01` No blending `10` ULA/tilemap as blend colour `11` Tilemap as blend colour
4	(Core 3.1.4+) Cancel entries in 8x5 matrix for extended keys
3	`1` to enable ULA+ (`0` after soft reset)
2	`1` to enable ULA half pixel scroll (`0` after soft reset)
1	Reserved, set to `0`
0	`1` to enable stencil mode when both the ULA and tilemap are enabled.

See **Sprite and Layers System** `$15` (page 89) for different priorities and mixing of ULA, Layer 2 and Sprites.

Tilemap Control `$6B`

Bit	Effect
7	`1` to enable tilemap, `0` disable tilemap
6	`1` for 80x32, `0` 40x32 mode
5	`1` to eliminate attribute byte in tilemap
4	`1` for second, `0` for first tilemap palette
3	`1` to activate "text mode"[1]
2	Reserved, set to `0`
1	`1` to activate 512, `0` for 256 tile mode
0	`1` to force tilemap on top of ULA

[1] In the text mode, tiles are defined as 1-bit B&W bitmaps, same as original Spectrum UDGs. Each tile only requires 8 bytes. In this mode, the tilemap attribute byte is also interpreted differently: bit 0 is still ULA over Tilemap (or 9th bit of tile data index) but the top 7 bits are extended palette offset (the least significant bit is the value of the pixel itself). In this mode, transparency is checked against **Global Transparency** `$14` (page 82) colour, not against the four-bit tilemap colour index.

Default Tilemap Attribute $6C

If single byte tilemap mode is selected (bit 5 of **Tilemap Control** $6B), this register defines attributes for all tiles.

Bit	Effect
7-4	Palette offset
3	1 to mirror tiles in X direction
2	1 to mirror tiles in Y direction
1	1 rotate tiles 90°clockwise
0	In 512 tile mode, bit 8 of tile index 1 for ULA over tilemap, 0 for tilemap over ULA

Tilemap Base Address $6E

Bit	Effect
7-6	Ignored, set to 0
5-0	Most significant byte of tilemap data offset in bank 5

Tile Definitions Base Address $6F

Bit	Effect
7-6	Ignored, set to 0
5-0	Most significant byte of tile definitions offset in bank 5

3.8 Sprites

One of the frequently used "my computer is better" arguments from owners and developers of contemporary systems such as Commodore 64 was hardware supported sprites. To be fair, they had a point - poor old Speccy had none. But Next finally rectifies this with a sprite system that far supersedes even later 16-bit era machines such as Amiga. And as we'll see, it's really simple to program too!

Some of the capabilities of Next sprites:

- 128 simultaneous sprites
- 16x16 pixels per sprite
- Magnification of 2x, 4x or 8x horizontally and vertically
- Mirroring and rotation
- Sprite grouping to form larger objects
- 512 colours from 2 256 colour palettes
- Per sprite palette
- Built-in sprite editor

So lots of reasons to get excited! Let's dig in!

3.8.1 Editing

Before describing how sprites hardware works, it would be beneficial to know how to draw them. As mentioned, Next comes with a built-in sprite editor. To use it, change to desired folder, then enter `.spredit <filename>` in BASIC or command line. The editor is quite capable and can even be used with a mouse if you have one attached to your Next (or in the emulator).

Alternatively, if you're developing cross-platform, the most feature-complete application for converting images to sprite data in my experience is Next Graphics[8]. It takes images from your preferred editor and converts them into a format that can be easily consumed by Next hardware. Other options I found are UDGeed-Next[9] or Remy's Sprite, Tile and Palette editor[10]. In contrast to Next Graphics, the latter two are not just exporters but also editors and share very similar feature sets. So try them out and decide for yourself.

3.8.2 Patterns

Next sprites have a fixed size of 16x16 pixels. Their display surface is 320x256, overlapping the ULA by 32 pixels on each side. Or in other words, to draw the sprite fully on-screen, we need to position it to (32,32) coordinate. And the last coordinate where the sprite is fully visible at

[8] `https://github.com/infromthecold/Next-Graphics`

[9] `http://zxbasic.uk/files/UDGeedNext-current.rar`

[10] `https://zx.remysharp.com/sprites/`

the bottom-right edge is (271,207). This allows sprites to be animated in and out of the visible area. Sprites can be made visible or invisible when over the border as well as rendered on top or below Layer 2 and ULA, all specified by **Sprite and Layers System** `$15` (page 89). It's also possible to further restrict sprite visibility within provided clip window using **Clip Window Sprites** `$19` (page 101).

Sprite patterns (or pixel data) are stored in Next FPGA internal 16K memory. As mentioned, sprites are always 16x16 pixels but can be 8-bit or 4-bit.

- 8-bit sprites use full 8-bits to specify colour, so each pixel can be of any of 256 colours from the sprite palette of which one acts as transparent. Hence each sprite occupies 256 bytes of memory and 64 sprites can be stored.
- 4-bit sprites use only 4-bits for colour, so each pixel can only choose from 16 colours, one of which is reserved for transparency. However this allows us to store 2 colours per byte, so these sprites take half the memory of 8-bit ones: 128 bytes each, meaning 128 sprites can be stored in available memory.

3.8.3 Palette

Each sprite can specify its own palette offset. This allows sprites to share image data but use different colours. 4 bits are used for palette offset, therefore the final colour index within the current sprite palette (as defined by **Enhanced ULA Control** `$43` (page 65)) is determined using the following formula:

8-bit sprites

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	P_3	P_2	P_1	P_0	0	0	0	0
+	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
=	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

If default palette offset and default palette are used, sprite colour index can be interpretted as RGB332 colour.

4-bit sprites

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
	P_3	P_2	P_1	P_0	0	0	0	0
+	0	0	0	0	S_3	S_2	S_1	S_0
=	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0

Palette offset can be thought of as if selecting one of 16 different 16-colour palettes.

P_n is palette offset bit, S_n sprite colour index bit and C_n final colour index.

Transparent colour is defined with **Sprites Transparency Index** `$4B` (page 104).

3.8.4 Combined Sprites

Anchor Sprites

These are “normal” 16x16 pixel sprites, as described in previous sections. They act as standalone sprites.

The reason they are called “anchors” is because multiple sprites can be grouped together to form larger sprites. In such case “anchor” acts as a parent and all its “relative” sprites are tied to it. In order to combine sprites, anchor needs to be defined first, immediately followed by all its relative sprites. The group ends with the next anchor sprite which can either be another standalone sprite, or an anchor for another sprite group. For example, if sprite 5 is setup as an anchor, its relative sprites must be followed at 6, 7, 8... until another sprite that’s setup as “anchor”.

There are 2 types of relative sprites: composite and unified sprites.

Composite Relative Sprites

Composite sprites inherit certain attributes from their anchor.

Inherited attributes:

- Visibility
- X
- Y
- Palette offset
- Pattern number
- 4 or 8-bit pattern

NOT inherited:

- Rotation
- X & Y mirroring
- X & Y scaling

Relative sprites only have 8-bits for X and Y coordinates (ninth bits are used for other purposes). But as the name suggests, these coordinates are relative to their parent anchor sprite so they are usually positioned close by. When the anchor sprite is moved to a different position on the screen, all its relatives are also moved by the same amount.

Visibility of relative sprites is determined as `AND` between anchor visibility and relative sprite visibility. This way individual relative sprites can be made invisible independently from their anchor, but if the anchor is invisible, then all its relative sprites will also be invisible.

Relative sprites inherit 4 or 8-bit setup from their anchor. They can’t use a different type but can use a different palette offset than its anchor.

It’s also possible to tie relative sprite’s pattern number to act as an offset on top of its anchor’s pattern number and thus easily animate the whole sprite group simply by changing the anchor’s pattern number.

Unified Relative Sprites

Unified relative sprites are an extension of the composite type. Everything described above applies here as well.

The main difference is the hardware will automatically adjust relative sprites X, Y, rotation, mirroring and scaling attributes according to changes in anchor. So relatives will rotate, mirror and scale around the anchor as if it was a single larger sprite.

3.8.5 Attributes

Attributes are 4 or 5 bytes that define where and how the sprite is drawn. The data can be set either by selecting sprite index with **Sprite Status/Slot Select** `$303B` and then continuously sending bytes to **Sprite Attribute Upload** `$xx57` (details on page 100) which automatically increments sprite index after all data for single sprite is transferred or by calling individual direct access Next registers `$35-$39` or their auto-increment variants `$75-$79`. See ports and registers section 3.8.7, page 100 for a description of individual bytes:

- Byte 0: **Sprite Port-Mirror Attribute 0** `$35` (page 102)
- Byte 1: **Sprite Port-Mirror Attribute 1** `$36` (page 102)
- Byte 2: **Sprite Port-Mirror Attribute 2** `$37` (page 102)
- Byte 3: **Sprite Port-Mirror Attribute 3** `$38` (page 102)
- Byte 4: **Sprite Port-Mirror Attribute 4** `$39` (page 103)

3.8.6 Examples

Reading about sprites may seem complicated, but in practice, it's quite simple. The following pages include sample code for working with sprites.

To preserve space, only partial code demonstrating relevant parts is included. You can find fully working example in companion code on GitHub in folder `sprites`. Besides demonstrating anchor and relative sprites, it also includes some very crude animations as a bonus.

Loading Patterns into FPGA Memory

Before we can use sprites, we need to load their data into FPGA memory. This example introduces a generic routine that uses DMA[11] to copy from given memory to FPGA. Don't worry if it seems like magic - it's implemented as a reusable routine, just copy it to your project. Routine requires 3 parameters:

- `HL` Source address of sprites to copy from
- `BC` Number of bytes to copy
- `A` Starting sprite number to copy to

```
LoadSprites:
    LD (.dmaSource), HL   ; Copy sprite sheet address from HL
    LD (.dmaLength), BC   ; Copy length in bytes from BC
    LD BC, $303B          ; Prepare port for sprite index
    OUT (C), A            ; Load index of first sprite
    LD HL, .dmaProgram    ; Setup source for OTIR
    LD B, .dmaProgramLength ; Setup length for OTIR
    LD C, $6B             ; Setup DMA port
    OTIR                  ; Invoke DMA code
    RET
.dmaProgram:
    DB %10000011          ; WR6 - Disable DMA
    DB %01111101          ; WR0 - append length + port A address, A->B
.dmaSource:
    DW 0                  ; WR0 par 1&2 - port A start address
.dmaLength:
    DW 0                  ; WR0 par 3&4 - transfer length
    DB %00010100          ; WR1 - A incr., A=memory
    DB %00101000          ; WR2 - B fixed, B=I/O
    DB %10101101          ; WR4 - continuous, append port B address
    DW $005B              ; WR4 par 1&2 - port B address
    DB %10000010          ; WR5 - stop on end of block, CE only
    DB %11001111          ; WR6 - load addresses into DMA counters
    DB %10000111          ; WR6 - enable DMA
.dmaProgramLength = $-.dmaProgram
```

See section 3.3, page 43 for details on how to program the zxnDMA.

[11] https://wiki.specnext.dev/DMA

Loading Sprites

Using `loadSprites` routine is very simple. This example assumes you've edited sprites with one of the editors and saved them as `sprites.spr` file in the same folder as the assembler code:

```
    LD HL, sprites        ; Sprites data source
    LD BC, 16*16*5        ; Copy 5 sprites, each 16x16 pixels
    LD A, 0               ; Start with first sprite
    CALL LoadSprites      ; Load sprites to FPGA

sprites:
    INCBIN "sprites.spr"  ; Sprite sheets file
```

Enabling Sprites

After sprites are loaded into FPGA memory, we need to enable them:

```
    NEXTREG $15, %01000001   ; Sprite 0 on top, SLU, sprites visible
```

Displaying a Sprite

Sprites are now loaded into FPGA memory, they are enabled, so we can start displaying them. This example displays the same sprite pattern twice, as two separate sprites:

```
    NEXTREG $34, 0           ; First sprite
    NEXTREG $35, 100         ; X=100
    NEXTREG $36, 80          ; Y=80
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %10000000   ; Visible, no byte 4, pattern 0

    NEXTREG $34, 1           ; Second sprite
    NEXTREG $35, 86          ; X=86
    NEXTREG $36, 80          ; Y=80
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %10000000   ; Visible, no byte 4, pattern 0
```

Displaying Combined Sprites

Even handling combined sprites is much simpler in practice than in theory! This example combines 4 sprites into a single one using unified relative sprites. Note use of “inc” register $79 which auto-increments sprite index for next sprite:

```
    NEXTREG $34, 2           ; Select third sprite
    NEXTREG $35, 150         ; X=150
    NEXTREG $36, 80          ; Y=80
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %11000001   ; Visible, use byte 4, pattern 1
    NEXTREG $79, %00100000   ; Anchor with unified relatives, no scaling

    NEXTREG $35, 16          ; X=AnchorX+16
    NEXTREG $36, 0           ; Y=AnchorY+0
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %11000010   ; Visible, use byte 4, pattern 2
    NEXTREG $79, %01000000   ; Relative sprite

    NEXTREG $35, 0           ; X=AnchorX+0
    NEXTREG $36, 16          ; Y=AnchorY+16
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %11000011   ; Visible, use byte 4, pattern 3
    NEXTREG $79, %01000000   ; Relative sprite

    NEXTREG $35, 16          ; X=AnchorX+16
    NEXTREG $36, 16          ; Y=AnchorY+16
    NEXTREG $37, %00000000   ; Palette offset, no mirror, no rotation
    NEXTREG $38, %11000100   ; Visible, use byte 4, pattern 4
    NEXTREG $79, %01000000   ; Relative sprite
```

Because we use combined sprite, we only need to update the anchor to change all its relatives. And because we set it up as unified relative sprites, even rotation, mirroring and scaling is inherited as if it was a single sprite!

```
    NEXTREG $34, 1           ; Select second sprite
    NEXTREG $35, 200         ; X=200
    NEXTREG $36, 100         ; Y=100
    NEXTREG $37, %00001010   ; Palette offset, mirror X, rotate
    NEXTREG $38, %11000001   ; Visible, use byte 4, pattern 1
    NEXTREG $39, %00101010   ; Anchor with unified relatives, scale X$Y
```

3.8.7 Sprite Ports and Registers

Sprite Status/Slot Select $303B

Write: sets active sprite attribute and pattern slot index used by **Sprite Attribute Upload** $xx57 and **Sprite Pattern Upload** $xx5B (see below).

Bit	Effect
7	Set to 1 to offset reads and writes by 128 bytes
6-0	0-63 for pattern slots and 0-127 for attribute slots

Read: returns sprite status information

Bit	Effect
7-2	Reserved
1	1 if sprite renderer was not able to render all sprites; read will reset to 0
0	1 when collision between any 2 sprites occurred; read will reset to 0

Sprite Attribute Upload $xx57

Uploads the attributes for the currently selected sprite slot. Attributes require 4 or 5 bytes. After all bytes are sent, the sprite index slot automatically increments. See the following Next registers that directly set the value for specific bytes:

- Byte 0: **Sprite Port-Mirror Attribute 0** $35 (page 102)
- Byte 1: **Sprite Port-Mirror Attribute 1** $36 (page 102)
- Byte 2: **Sprite Port-Mirror Attribute 2** $37 (page 102)
- Byte 3: **Sprite Port-Mirror Attribute 3** $38 (page 102)
- Byte 4: **Sprite Port-Mirror Attribute 4** $39 (page 103)

Sprite Pattern Upload $xx5B

Uploads sprite pattern data. 256 bytes are needed for each sprite. For 8-bit sprites, each pattern slot contains a single sprite. For 4-bit sprites, it contains 2 128 byte sprites. After 256 bytes are sent, the target pattern slot is auto-incremented.

Bit	Effect
7-0	Next byte of pattern data for current sprite

Peripheral 4 $09

Bit	Effect
7	1 to enable AY2 “mono” output (A+B+C is sent to both R and L channels, makes it a bit louder than stereo mode)
6	1 to enable AY1 “mono” output, 0 default
5	1 to enable AY0 “mono” output (0 after hard reset)
4	1 to lockstep **Sprite Port-Mirror Index** $34 (page 101) and **Sprite Status/Slot Select** $303B (page 100)
3	1 to reset mapram bit in DivMMC
2	1 to silence HDMI audio (0 after hard reset) (since core 3.0.5)
1-0	Scanlines weight (0 after hard reset)

	Core 3.1.1+	**Older cores**
00	Scanlines off	Scalines off
01	Scanlines 50%	Scanlines 75%
10	Scanlines 50%	Scanlines 25%
11	Scanlines 25%	Scanlines 12.5%

Sprite and Layers System $15

See description under Tilemap, section 3.7.6, page 89.

Clip Window Sprites $19

Bit	Effect
7-0	Reads or writes clip-window coordinates for Sprites

4 coordinates need to be set: X1, X2, Y1 and Y2. Sprites will only be visible within these coordinates. Positions are inclusive. Default values are 0, 255, 0, 191. Origin (0,0) is located 32 pixels to the top-left of ULA top-left coordinate.

Which coordinate gets set, is determined by index. As each write to this register will also increment index, the usual flow is to reset the index to 0 with **Clip Window Control** $1C (page 83), then write all 4 coordinates in succession.

When “over border” mode is enabled (bit 1 of **Sprite and Layers System** $15, page 89), X coordinates are doubled internally.

Clip Window Control $1C

See description under Layer 2, section 3.6.8, page 83.

Sprite Port-Mirror Index $34

If sprite id lockstep in **Peripheral 4** $09 (page 101) is enabled, write to this registers has same effect as writing to **Sprite Status/Slot Select** $303B (page 100).

Bit	Effect
7	Set to 1 to offset reads and writes by 128 bytes
6-0	0-63 for pattern slots and 0-127 for attribute slots

Sprite Port-Mirror Attribute 0 $35

Bit	Effect
7-0	Low 8 bits of X position

Sprite Port-Mirror Attribute 1 $36

Bit	Effect
7-0	Low 8 bits of Y position

Sprite Port-Mirror Attribute 2 $37

Bit	Effect
7-4	Palette offset
3	1 to enable X mirroring, 0 to disable
2	1 to enable Y mirroring, 0 to disable
1	1 to rotate sprite 90°clockwise, 0 to disable
0	Anchor sprite: most significant bit of X coordinate Relative sprite: 1 to add anchor palette offset, 0 to use independent palette offset

Sprite Port-Mirror Attribute 3 $38

Bit	Effect
7	1 to make sprite visible, 0 to hide it
6	1 to enable optional byte 4, 0 to disable it
5-0	Pattern index 0-63 (7th, MSB for 4-bit sprites is configured with byte 4)

Sprite Port-Mirror Attribute 4 $39

For anchor sprites:

Bit	Effect
7-6	H+N6 where H is 4/8-bit data selector and N6 is sub-pattern selector for 4-bit sprites 00 Anchor sprite, 8-bit 10 Anchor sprite, 4-bit using bytes 0-127 of pattern slot 11 Anchor sprite, 4-bit using bytes 128-255 of pattern slot
5	0 if this anchor's relative sprites are composite, 1 for unified sprite
4-3	X axis scale factor 00 1x 01 2x 10 4x 11 8x
2-1	Y axis scale factor, see above
0	Most significant bit of Y coordinate

For composite relative sprites:

Bit	Effect
7-6	01 needs to be used for relative sprites
5	4-bit mode: N6, 1 to use bytes 0-127, 0 to use bytes 128-255 of pattern slot 8-bit mode: not used, set to 0
4-3	X axis scale factor, see below
2-1	Y axis scale factor, see below
0	1 to enable relative pattern offset, 0 to use independent pattern index

For unified relative sprites

Bit	Effect
7-6	01 needs to be used for relative sprites
5	4-bit mode: N6, 1 to use bytes 0-127, 0 to use bytes 128-255 of pattern slot 8-bit mode: not used, set to 0
4-1	Set to 0; scaling is defined by anchor sprite
0	1 to enable relative pattern offset, 0 to use independent pattern index

Palette Index $40

Palette Value $41

Enhanced ULA Control $43

Enhanced ULA Palette Extension $44

See description under Palette, section 3.4.5, pages 63-65.

Sprites Transparency Index $4B

Bit	Effect
7-0	Sets index of transparent colour inside sprites palette.

For 4-bit sprites, low 4 bits of this register are used.

Sprite Port-Mirror Attribute 0 (With Increment) $75

Sprite Port-Mirror Attribute 1 (With Increment) $76

Sprite Port-Mirror Attribute 2 (With Increment) $77

Sprite Port-Mirror Attribute 3 (With Increment) $78

Sprite Port-Mirror Attribute 4 (With Increment) $79

This set of registers work the same as their non-inc counterpart in $35-$39; writes byte 0-4 of Sprite attributes for currently selected sprite, except $7X variants also increment **Sprite Port-Mirror Index** $34 (page 101) after write. When batch updating multiple sprites, typically the first sprite is selected explicitly, then $3X registers are used until the last write, which occurs through $7X register. This way we'll also increment the sprite index for the next iteration.

3.9 Copper

Copper stands for "co-processor". If the name sounds familiar, there's a reason - it functions similarly to the Copper from the Commodore Amiga Agnus chip. It allows changing a subset of Next registers at certain scanline positions, which frees the Z80 processor for other tasks.

Copper uses 2K of dedicated write-only memory for its programs. A program consists of a series of instructions. Instructions are 16-bits in size, meaning we can store up to 1024 instructions. Internally, Copper uses a 10-bit program counter (`CPC`) that by default auto-increments and wraps around from the last to the first instruction. But we can change this behaviour if needed.

Timing for Copper is 14MHz on core 2.0 and 28MHz on core 3.0. If interested, this document describes the timing and many other details for core 2.0 in great detail[12].

3.9.1 Instructions

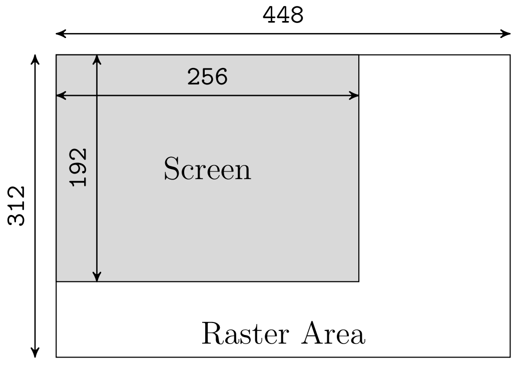
There are only two types of instructions ZX Next Copper understands, but each type has a special case, so in total, we can say there are four operations:

Op.	Bit Pattern	Effect	Dur.
`WAIT`	`1HHHHHHV VVVVVVVV`	Wait for raster line **V** (`0-311`) and horizontal position **H** (`0-55`)	1 cycle
`HALT`	`11111111 11111111`	Special case of `WAIT`; works as "halt"	1 cycle
`MOVE`	`0RRRRRRR VVVVVVVV`	Write value **V** to Next register **R**	2 cycles
`NOOP`	`00000000 00000000`	Special case of `MOVE`; works as "no operation"	1 cycle

WAIT

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	Horizontal (0-55)						Vertical (0-311)								

`WAIT` blocks Copper program until the current raster line reaches the 9-bit vertical position from bits 8-0. When the line matches, it further waits until the given 6-bit horizontal position is reached.



The raster area addressable by Copper is 448×312 pixels. So for the standard 256×192 resolution, one horizontal position translates into 8 pixels. The visible portion of the screen is positioned top-left within the raster area like shown in this drawing. This means Copper line 0 corresponds with the top line of the screen, just below the border.

[12] `https://gitlab.com/thesmog358/tbblue/blob/master/docs/extra-hw/copper/COPPER-v0.1c.TXT`

HALT

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

`HALT` is a special case of `WAIT` instruction that tells Copper to wait for the vertical position 511 and horizontal 63. As these are unreachable positions, it will effectively stop all further Copper processing until Next is reset.

However, when mode `11` is used (bits 7-6 of **Copper Control High Byte** `$62`, page 109), Copper will auto-wrap to the first instruction on every vertical blank. This allows us to use `HALT` to mark the end of the Copper program without having to fill in the remaining bytes with `0`. Quite convenient! In fact, I'd imagine this would be the most commonly used mode. If you're used to Copper on Amiga, this is also how it behaved there.

MOVE

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	Next Register (0-127)							Value (0-255)							

`MOVE` writes the given 8-bit value to the given Next register. Any register between `1` and `127` (`$7F`) can be written to. Register `0` is a special case, see `NOOP` below.

`MOVE` can be used for all sorts of neat effects. For example: change Layer 2 offsets to achieve parallax scrolling effect, change palette at specific screen coordinates to achieve sky gradient or simulate above and under-water colours etc.

NOOP

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

`NOOP` is a special case of `MOVE` that effectively does nothing for a period of one horizontal position. It can be used to fine-tune timing, align colour and display changes etc.

3.9.2 Configuration

To load a program, we need to send it, byte by byte, through **Copper Data** `$60` or **Copper Data 16-bit Write** `$63` registers (page 109). As instructions are 16-bits in size, two writes are required. The difference between the two registers is that `$60` sends bytes immediately while `$63` only after both bytes of an instruction are provided, thus preventing half-written instructions from executing.

Copper is controlled through 16-bit control word accessible through **Copper Control High Byte** `$62` and **Copper Control Low Byte** `$61` registers (page 109):

Copper Control High Byte `$62` (page 109)								**Copper Control Low Byte** `$61` (page 109)							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Mode		0	0	0	Index for program upload										

Mode can be one of the following:

- `00` Stops the Copper, `CPC` keeps its current value. This is useful during program upload, to prevent Copper from executing incomplete instructions and programs.
- `01` Resets `CPC` to `0`, then starts Copper. From here on, Copper will start executing the first instruction in the program and continue until `CPC` reaches 1023, then wrap around back to first. However it will stop if `HALT` instruction is encountered.
- `10` Starts or resumes Copper from current `CPC`. Similar to `01`, except that `CPC` is not changed. Instead, Copper resumes execution from current instruction.
- `11` Same as `01`, but also auto-resets `CPC` to 0 on vertical blank. In this mode we can use `HALT` to mark the end of the program and still repeat it without having to fill-in `NOOP`s from the last instruction of our program to the end of Copper 2K memory.

The other value we set is the index for program upload. This is 11-bit value (`0`-`2047`) specifying the byte offset for write commands with **Copper Data** `$60` or **Copper Data 16-bit Write** `$63` registers (page 109). In other words: this is the index for the location into which data will be uploaded, not the value of the `CPC`. We can't change `CPC` programmatically, apart from resetting it to `0`.

3.9.3 Example

Enough theory, let's see how it works in practice, Copper program first. It changes palette colour to green at the top of the screen and then to red in the middle:

```
CopperList:
    DB $80, 0         ; Wait line 0
    DB $41, %00011100 ; Set palette entry to green
    DB $80, 96        ; Wait line 96
    DB $41, %11100000 ; Set palette entry to red
    DB $FF, $FF       ; HALT
CopperListSize = $-CopperList
```

In case you may be wondering: we should also ensure we update the correct colour with

Enhanced ULA Control $43 (page 65) and **Palette Index** $40 (page 63). But I wanted to keep the program simple for demonstration purposes.

With Copper program in place, we can upload it to Copper memory. We can use DMA or directly upload values through Next registers. The code here demonstrates later, but companion code implements both so you can compare:

```
    ; Stop Copper and set data upload index to 0
    NEXTREG $61, %00000000
    NEXTREG $62, %00000000

    ; Copy list into Copper memory
    LD HL, CopperList        ; HL points to start of copper list
    LD B, CopperListSize     ; B = size of our Copper list in bytes
.nextByte:
    LD A, (HL)               ; Load current byte to A
    NEXTREG $63, A           ; Copy it to Copper memory
    INC HL                   ; Increment HL to next byte
    DJNZ .nextByte           ; Repeat or continue

    ; Start Copper in mode %11 - reset on every vertical blank
    NEXTREG $61, %00000000
    NEXTREG $62, %11000000
```

Again, this is an overly simplified example. It only works for lists that are less than 256 bytes long.

The next step is... Well, there is no next step - if we enabled Layer 2 and filled it with the colour we're changing in our Copper list, we should see the screen divided into two halves with top in green and bottom red colour.

Not the most impressive display of Copper capabilities, I give you that. You can find a more complex example in companion code on GitHub, folder `copper` with couple additional points of interest:

- Upload routine that supports programs of arbitrary size (within 1024 instructions limit)
- Example of upload routine using DMA
- Using DMA to fill in Layer 2 banks
- Macros that hopefully make Copper programs easier to read and write
- Usage of **Copper Data** $60 (page 109) to dynamically update individual bytes of the program in memory to achieve couple effects

3.9.4 Copper Registers

Copper Data `$60`

Bit	Effect
`7-0`	Data to upload to Copper memory

The data is written to the index specified with **Copper Control Low Byte** `$61` and **Copper Control High Byte** `$62` registers. After the write, the index is auto-incremented to the next memory position. The index wraps to `0` when the last byte of the program memory is written to position `2047`. Since Copper instructions are 16-bits in size, two writes are required to complete each one.

Copper Control Low Byte `$61`

Bit	Effect
`7-0`	Least significant 8 bits of Copper list index

Copper Control High Byte `$62`

Bit	Effect
`7-6`	Control mode `00` Stops the Copper, `CPC` keeps its current value `01` Resets `CPC` to `0`, then starts Copper `10` Starts or resumes Copper from current `CPC` `11` Same as `01`, but also auto-resets `CPC` to 0 on vertical blank
`5-3`	Reserved, must be `0`
`2-0`	Most significant 3 bits of Copper list index

When control mode is identical to current one, it's ignored. This allows change of the upload index without restarting the program.

Copper Data 16-bit Write `$63`

Bit	Effect
`7-0`	Data to upload to Copper memory

Similar to **Copper Data** `$60` except that writes are only committed to Copper memory after two bytes are written. This prevents half-written instructions to be executed.

The first write to this register is for MSB of the Copper instruction or even instruction address and second write for LSB or odd instruction address.

This page intentionally left empty

3.10 Sound

Next inherits the same 3 AY-3-8912 chips setup as used in 128K Spectrums. This allows us to reuse many of the pre-existing applications and routines to play sound effects and music.

3.10.1 AY Chip Registers

AY chip has 3 sound channels, called A, B and C. Combined with 3 chips, this allows us to produce 9 channel music. Programming wise, each of the 3 chips needs to be selected first via **Turbo Sound Next Control** `$FFFD` (page 113) register. Afterwards, we can set various parameters through **Peripheral 3** `$08` (page 115) and **Peripheral 4** `$09` (page 101) registers.

AY chip is controlled by 14 internal registers. To program them, we first need to select the register with **Turbo Sound Next Control** `$FFFD` (page 113) and then write the value with **Sound Chip Register Write** `$BFFD` (page 113).

3.10.2 Editing and Players

Several applications can produce sounds or music compatible with the AY chip. For sounds, Shiru's AYFX Player[13] can be used. This program also includes a Z80 native player that can directly load and play sound effects. Alternatively, Remy's AY audio generator website[14] can produce exactly the same results and is fully compatible with AYFX Player.

A different way of playing sounds is to convert the WAV file into 1, 2 or 4-bit per sample sound with the ChibiWave application. Sounds take a bit more memory this way but are much easier to create. You can find the application, as well as tutorial and playback source code on Chibi Akumas website[15]. While there, definitely check other tutorials too - they're all high quality and available as both, written posts and YouTube videos.

For creating music there are also several options. NextDAW[16] is native composer that runs on ZX Spectrum Next itself. Or if you prefer cross-platform, Arkos Tracker[17] or Vortex Tracker[18] should do the job. All include "drivers"; Z80 code you can include in your program that can load and play created music.

[13] `https://shiru.untergrund.net/software.shtml#old`

[14] `https://zx.remysharp.com/audio/`

[15] `https://www.chibiakumas.com/z80/platform4.php#LessonP35`

[16] `https://nextdaw.biasillo.com/`

[17] `https://www.julien-nevo.com/arkostracker/`

[18] `https://bulba.untergrund.net/vortex_e.htm`

3.10.3 Examples

Before we can start playing sounds, we need to enable the sound hardware. While this is usually enabled by default, it's nonetheless a good idea to ensure our program will always run under the same conditions.

```
    ; Setup Turbo Sound chip
    LD BC, $FFFD            ; Turbo Sound Next Control Register
    LD A, %11111101         ; Enable left+right audio, select AY1
    OUT (C), A

    ; Setup mapping of chip channels to stereo channels
    NEXTREG $08, %00010010 ; Use ABC, enable internal speaker $turbosound
    NEXTREG $09, %11100000 ; Enable mono for AY1-3
```

Programming AY consists of writing various values to its registers. As mentioned, this is a two-step process: first select register number, then write the value. Multiple writes are required for each tone to set period, volume etc. To make it simpler, I created a subroutine. It takes 2 parameters: `A` for register number (`0-13`) and `D` with value to write.

```
WriteDToAYReg:
    ; Select desired register
    LD BC, $FFFD
    OUT (C), A

    ; Write given value
    LD A, D
    LD BC, $BFFD
    OUT (C), A

    RET
```

Companion code on GitHub, folder `sound` includes expanded code as well as a simple player that plays multiple tones in sequence. For the purposes of this book, I used Remy's AY audio generator website to load one of the example effects, then manually copied raw values into the source code. Laborious process to say the least - this is not how effects should be handled in real life. But I wanted to learn and demonstrate how to program AY chip, not how to use ready-made drivers to play effects or music. Furthermore, my "player" blocks the main loop; ideally, sound effects and music would play on the interrupt handler. This could be a nice homework for the reader - example in section 3.12, page 119 should give you an idea of how to achieve this - happy coding!

3.10.4 Sound Ports and Registers

Turbo Sound Next Control `$FFFD`

When bit `7` is `1`:

Bit	Effect
`7`	`1`
`6`	`1` to enable left audio
`5`	`1` to enable right audio
`4-2`	Must be `1`
`1-0`	Selects active chip: `00` Unused `01` AY3 `10` AY2 `11` AY1

When bit `7` is `0`:

Bit	Effect
`7`	`0`
`6-0`	Selects given AY register number for read or write from active sound chip

Sound Chip Register Write `$BFFD`

Bit	Effect
`7-0`	Writes given value to currently selected register:

0 - Channel A tone, low byte

7	6	5	4	3	2	1	0
A tone							

1 - Channel A tone, high 4-bits

7	6	5	4	3	2	1	0
0	0	0	0	A tone high			

2 - Channel B tone, low byte

7	6	5	4	3	2	1	0
B tone							

3 - Channel B tone, high 4-bits

7	6	5	4	3	2	1	0
0	0	0	0	B tone high			

4 - Channel C tone, low byte

7	6	5	4	3	2	1	0
C tone							

5 - Channel C tone, high 4-bits

7	6	5	4	3	2	1	0
0	0	0	0	C tone high			

6 - Noise period

7	6	5	4	3	2	1	0
0	0	0	Noise Period				

7 - Flags

7	6	5	4	3	2	1	0
0	0	C	B	A	C	B	A
0	0	Noise			Tone		

8 - Channel A volume/envelope

7	6	5	4	3	2	1	0
0	0	0	0	A Volume			

9 - Channel B volume/envelope

7	6	5	4	3	2	1	0
0	0	0	0	B Volume			

10 - Channel C volume/envelope

7	6	5	4	3	2	1	0
0	0	0	0	C Volume			

Note: Registers `8-10` work as volume control if bit `4` is `0`, otherwise envelop generator is used (see registers `11-13`). In this case bits `3-0` are ignored.

11 - Envelope period fine

7	6	5	4	3	2	1	0
Envelope bits 7-0							

12 - Envelope period coarse

7	6	5	4	3	2	1	0
Envelope bits 15-8							

13 - Envelope shape

7	6	5	4	3	2	1	0
0	0	0	0	C	A_t	A_l	H

H “Hold”

- `1` envelope generator performs 1 cycle then holds the end value
- `0` cycles continuously

A_l “Alternate”

If “hold” set

- `1` the value held is initial value
- `0` the value held is the final value

If “hold” not set

- `1` envelope generator alters direction after each cycle
- `0` resets after each cycle

A_t “Attack”

- `1` the generator counts up
- `0` the generator counts down

C “Continue”

- `1` “hold” is followed
- `0` the envelope generator performs one cycle then drops volume to 0 and stays there, overriding “hold”

Peripheral 2 $06

Bit	Effect
7	1 to enable CPU speed mode key ”F8”, 0 to disable (1 after soft reset)
6	Core 3.1.2+: Divert BEEP-only to internal speaker (0 after hard reset) Pre core 3.1.2: DMA mode, 0 zxnDMA, 1 Z80 DMA (0 after hard reset)
5	Core 2.0+: 1 to enable ”F3” key (50/60 Hz switch) (1 after soft reset) Pre core 2.0: ”Enable Lightpen”
4	1 to enable DivMMC automap and DivMMC NMI by DRIVE button (0 after hard reset)
3	1 to enable multiface NMI by M1 button (0 after hard reset)
2	1 to set primary device to mouse in PS/2 mode, 0 to set to keyboard
1-0	Audio chip mode: 00 YM 01 AY 10 Disabled 11 Core 3.0+: Hold all AY in reset

Peripheral 3 $08

Bit	Effect
7	1 unlock / 0 lock port **Memory Paging Control** $7FFD (page 41) paging
6	1 to disable RAM and I/O port contention (0 after soft reset)
5	AY stereo mode (0 = ABC, 1 = ACB) (0 after hard reset)
4	Enable internal speaker (1 after hard reset)
3	Enable 8-bit DACs (A,B,C,D) (0 after hard reset)
2	Enable port $FF Timex video mode read (0 after hard reset)
1	Enable Turbosound (currently selected AY is frozen when disabled) (0 after hard reset)
0	Implement Issue 2 keyboard (port $FE reads as early ZX boards) (0 after hard reset)

Peripheral 4 $09

See description under Sprites, section 3.8.7, page 101.

This page intentionally left empty

3.11 Keyboard

Next inherits ZX Spectrum keyboard handling, so all legacy programs will work out of the box. Additionally, it allows reading the status of extended keys.

3.11.1 Legacy Keyboard Status

ZX Spectrum uses 8×5 matrix for reading keyboard status. This means 40 distinct keys can be represented. The keyboard is read from **ULA Control Port Read** `$xxFE` (page 118) with particular high bytes. There are 8 possible bytes, each will return the status of 5 associated keys. If a key is pressed, the corresponding bit is set to `0` and vice versa.

Example for checking if `P` or `I` is pressed:

```
    LD BC, $DFFE   ; We want to read keys..... YUIOP
    IN A, (C)      ; A holds values in bits... 43210
checkP:
    BIT 0, A       ; test bit 0 of A (P key)
    JR NZ checkI   ; if bit0=1, P not pressed
    ...            ; P is pressed
checkI:
    BIT 2, A       ; test bit 2 of A (I key)
    JR NZ continue ; if bit2=1, I not pressed
    ...            ; I is pressed
continue:
```

As mentioned in Ports chapter, section 3.1.3, page 33, we can slightly improve performance if we replace first two lines with:

```
    LD A, $DF
    IN ($FE)
```

Reading the port in first example requires 22 t-states (10+12) vs. 18 (7+11). The difference is small, but it can add up as typically keyboard is read multiple times per frame.

The first program is more understandable at a glance - the port address is given as a whole 16-bit value, as usually provided in the documentation. The second program splits it into 2 8-bit values, so intent may not be immediately apparent. Of course, one learns the patterns with experience, but it nonetheless demonstrates the compromise between readability and speed.

3.11.2 Next Extended Keys

Next uses larger 8×7 matrix for keyboard, with 10 additional keys. By default, hardware is translating keys from extra two columns into the existing 8×5 set. But you can turn this off with bit 4 of **ULA Control** `$68` (page 91). Extra keys can be read separately via **Extended Keys 0** `$B0` and **Extended Keys 1** `$B1`, details on page 118.

3.11.3 Keyboard Ports and Registers

ULA Control Port Read $xxFE

Returns keyboard status when read with certain high byte values:

xx	4	3	2	1	0
$7F	B	N	M	Symb	Space
$BF	H	J	K	L	Enter
$DF	Y	U	I	O	P
$EF	6	7	8	9	0
$F7	5	4	3	2	1
$FB	T	R	E	W	Q
$FD	G	F	D	S	A
$FE	V	C	X	Z	Caps

Bits are reversed: if a key is pressed, the corresponding bit is 0, if a key is not pressed, bit is 1.

Note: when written to, **ULA Control Port Write** $xxFE is used to set border colour and audio devices. See ULA Layer, section 3.5.6, page 71 for details.

ULA Control $68

See description under Tilemap, section 3.7.6, page 91.

Extended Keys 0 $B0

Extended Keys 1 $B1

Bit	Effect	$B0	$B1
7	0 if key pressed, 1 otherwise	;	Delete
6	0 if key pressed, 1 otherwise	"	Edit
5	0 if key pressed, 1 otherwise	,	Break
4	0 if key pressed, 1 otherwise	.	Inv Video
3	0 if key pressed, 1 otherwise	Up	True Video
2	0 if key pressed, 1 otherwise	Down	Graph
1	0 if key pressed, 1 otherwise	Left	Caps Lock
0	0 if key pressed, 1 otherwise	Right	Extend

Available since core 3.1.5

3.12 Interrupts on Next

Like many other functionalities, Next also inherits interrupts handling from ZX Spectrum. As described in the Z80 chapter, section 2.4, page 19, Interrupt Mode 0 is used for a short period of time after booting up, before ROM activates Mode 1. IM1 remains active unless we explicitly change it. To replace the default IM1 interrupt handler, we can page out ROM or change to Interrupt Mode 2.

Both IM1 and IM2 are triggered by standard Spectrum ULA on every video frame. However timing can be adjusted using **Video Line Interrupt Control** $22 and **Video Line Interrupt Value** $23 registers (page 125). This allows disabling standard ULA interrupt, or add an extra interrupt that occurs on a particular video line.

Interrupts are most frequently used for driving music playback. Though they can also be used for other purposes such as synchronizing game state etc.

3.12.1 Interrupt Mode 1

In Interrupt Mode 1, an interrupt handler is expected on address $0038 in ROM. Default handler updates frame counter system variable, scans the keyboard and updates keyboard state system variables. We can replace it with our routine by paging out ROM.

Let's see how to do this with an example. We will use the interrupt routine to update an 8-bit counter. The example uses sjasmplus directives to establish paging. If you are using a different assembler, you may need to change! First, let's prepare the groundwork - declare the variable and main part of the program:

```
    DEVICE ZXSPECTRUMNEXT ; this is Next program - important for paging setup!

    ORG $8000           ; set PC to $8000
Start:
    DI                  ; disable interrupts while we page out ROM
    NEXTREG $50, 28     ; page out ROM in slot 0 ($000-$1FFF) with page 28
    IM 1                ; enable Interrupt Mode 1
    EI                  ; enable interrupts

.loop:
    JP .loop            ; infinite loop
    RET                 ; this is never reached...

counter: DB 0           ; counter variable
```

The first directive tells sjasmplus to generate the code for Next. This will become important later on. Then we set the program counter to $8000; all subsequent instructions and data will be assembled starting at this address.

Next, we are paging out ROM in 8K slot 0 with bank 28. We'll write our interrupt handler subroutine into this bank later on. Afterwards, we enable Interrupt Mode 1. This is optional;

by default, Next will run in this mode already, but being explicit is more future proof. Maybe another program would change to IM2. Note how we disable interrupts during this setup to prevent undesired side effects.

The remaining part is simply an infinite loop to prevent the program from exiting. And finally, we declare our counter variable.

The interrupt subroutine is expected at $0038. There are several ways to achieve this with sjasmplus. I opted for explicit slot/bank technique so we're in control of the paging:

```
1      SLOT 6             ; activate slot 6
2      PAGE 28            ; load page 28K to active slot
3      ORG $C038          ; set PC to $C038
4
5  InterruptHandler:
6      LD HL, counter     ; load address of counter var
7      INC (HL)           ; increment it
8      EI                 ; enable interrupts
9      RETI               ; return from interrupt
```

This is where the previously mentioned DEVICE directive comes to play - sjasmplus decides slot and bank configuration based on it. ZXSPECTRUMNEXT assumes 8K slot and bank size. Lines 1-3 are the key to ensure our interrupt handler will be present on $0038:

- SLOT directive tells sjasmplus we want the following section to be loaded into 8K slot 6, addresses $C000-$DFFF.
- Next, we specify 8K bank number to load into the selected slot with PAGE directive. Subsequent code will be assembled into this bank. This step is optional; default bank would be used otherwise, 0 in this case (see section 3.2, page 35). Being explicit is more future proof though. I chose bank 28 but feel free to use others. What's important is to use the same bank with NEXTREG instruction when paging out ROM!
- Since we selected slot 6, we also need to set the program counter to the corresponding address - we want the code to be assembled into the selected bank that we will page into ROM slot 0 at runtime. Because interrupt handler is expected on $0038, we need to start at $C038 (slot 6 starts at $C000, but this will be loaded into slot 0 at $0000).

Not that hard, right!? It may take a while to wrap the head around those SLOT, PAGE and ORG directives and addresses, but it's quite straightforward otherwise. While at it: this same technique can be used to load assets into specific banks and then page them in during runtime!

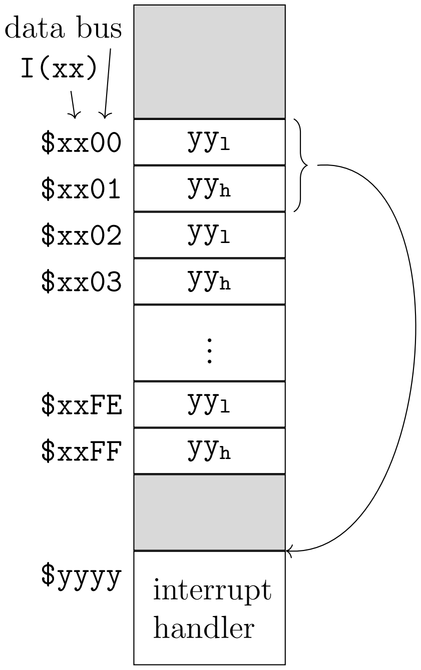
You can find the full source code for this example in companion code, folder im1. Feel free to run it, set breakpoints and see how the counter is being changed.

Note there's one potentially deal-breaking issue with this approach: since we are paging out ROM, we can't use any of its functionality. For example, ROM routines like print text at $203C. But we can do this with IM2 - read on!

3.12.2 Interrupt Mode 2

Once Interrupt Mode 2 is activated, mode 1 handler at `$0038` isn't called anymore. Instead, when an interrupt occurs, Z80 performs the following steps:

- First, a 16-bit address is formed where the value for the most significant byte is taken from the `I` register and the value for the least significant byte from the current value of the data bus.
- Two bytes are read from this address (little endian format is assumed - low byte first, then high byte); these 2 bytes form another 16-bit address.
- It's this second address that the CPU treats as an interrupt routine and starts executing from.



In other words:

Because the LSB for the vector table address is effectively a random number, we need to allocate 256 bytes in the memory, on 256-byte boundary (meaning any address with low byte `$00` - `$xx00`). This gives us a chunk of memory where low byte starts at `$00` up to `$FF`, thus covering all possible 8-bit values the data bus can "throw" at us. This chunk, called "vector table", is 256 bytes long and consists of 128 16-bit addresses (vectors), all pointing to the IM2 interrupt handler routine.

We are responsible for assigning the high byte of the vector table address to `I` register though. Together 16-bit address for vector table lookup looks like this:

15-8	7-0
`I` register	Data Bus

Phew, a lot of words and concepts to grasp. But it's simpler than it sounds - let's rewrite the IM1 example, but this time with IM2. The main program first:

```
   ORG $8000             ; set PC to $8000
Start:
   DI                    ; disable interrupts while we setup interrupts
   CALL SetupInterruptVectors
   IM 2                  ; enable Interrupt Mode 2
   EI                    ; enable interrupts
   (the rest is the same as in IM1 example)
```

Almost the same except for the `NEXTREG` replaced with `SetupInterruptVectors` subroutine call that initializes vector table:

```
SetupInterruptVectors:
   LD DE, InterruptHandler     ; prepare pointer to IM2 routine
```

```
    LD HL, InterruptVectorTable ; prepare pointer to vector table
    LD B, 128                   ; we need to fill 128 addresses
.loop:
    LD (HL), E                  ; copy low byte
    INC HL                      ; increment vector table pointer
    LD (HL), D                  ; copy high byte
    INC HL                      ; increment vector table pointer
    DJNZ .loop                  ; repeat until the end of the vector table
    LD A, InterruptVectorTable >> 8
    LD I, A                     ; I now holds high byte of vector table
    RET
```

The subroutine fills in all 128 entries of the vector table with the address of our actual interrupt routine. The only remaining piece is the declaration of the vector table itself; I chose `.ALIGN 256` directive that fills in bytes until 256-byte boundary is reached:

```
    .ALIGN 256                  ; section must start on 256-byte boundary $xx00
InterruptVectorTable:           ; I register should therefore have value $xx
    DEFS 128*2                  ; 128 16-bit vectors
```

The interrupt handler itself can reside anywhere in the memory. Code-wise it's the same as for IM1 mode previously.

So there's some more work when using IM2 mode, but it leaves ROM untouched so we can rely on its routines and other functionality. I only demonstrated relevant bits here. You can find a fully working example in companion code, folder `im2` (make sure to **read the next section**!).

What About Odd Data Bus Values?

Sharp-eyed readers may be wondering what would happen if the value from the data bus is odd? Given our IM2 example from above, our interrupt handler happens to be on address `$802A`. Therefore the vector table would have the following contents:

$xx00	$xx01	$xx02	$xx03	...	$xxFE	$xxFF
$2A	$80	$2A	$80	...	$2A	$80

If the bus value is even, for example `0`, `2`, `4` etc, then the address of the interrupt handler would be correctly read as `$802A` (remember, Z80 is little-endian). But what if the value is odd, `1`, `3`, `5` etc? In this case, 16-bit interrupt handler address would be wrong - `$2A80`! During my tests, this didn't happen - whether by luck, the data bus always had an even number, or, maybe Z80 reset bit 0 before constructing vector lookup address. But we can't rely on luck!

In fact, Next Dev Wiki[19] recommends that the interrupt handler routine is always placed on an address where high and low bytes are equal. It's best to follow this advice to avoid potential issues. The changes to the code are minimal, so I won't show it here - it could be a nice exercise though! Just a tip - to be extra safe, make vector table 257 bytes long in case the data bus has `$FF`. You can find a working example in companion code, folder `im2safe`.

[19] https://wiki.specnext.dev/Interrupts

3.12.3 Hardware Interrupt Mode 2

In addition to regular IM2, Next also supports a "Hardware IM2" mode. This mode works similar to legacy IM2 in that we still need to provide vector tables. But it properly implements the Z80 IM2 scheme with daisy chain priority. So when particular interrupt subroutine is called, we know exactly which device caused it without having to figure it out as needed in legacy IM2. Let's go over the details with an example. First, the initialization:

```
1      DI                           ; disable interrupts
2      NEXTREG $C0, (InterruptVectorTable & %11100000) | %00000001
3      NEXTREG $C4, %10000001       ; enable expansion bus INT and ULA interrupts
4      NEXTREG $C5, %00000000       ; disable all CTC channel interrupts
5      NEXTREG $C6, %00000000       ; disable UART interrupts
6
7      LD A, InterruptVectorTable >> 8
8      LD I, A                      ; I now holds high byte of vector table
9
10     IM 2                         ; enable HW Interrupt Mode 2
11     EI                           ; enable interrupts
```

Line 3 is where hardware IM2 mode is established. The crucial part is **Interrupt Control** `$C0` (page 126) register: firstly, we are supplying the top 3 bits of LSB of the vector table to bits 7-5, and secondly, we enable IM2 mode by setting bit 0.

Lines 4-6 enable or disable specific interrupters with **Interrupt Enable 0** `$C4` (page 126), **Interrupt Enable 1** `$C5` (page 126) and **Interrupt Enable 2** `$C6` (page 127).

Lines 8-12 are the same as with legacy IM2 mode - we assign the LSB of the vector table address to `I` register. Then we enable IM2 mode and interrupts.

Interrupt vectors table is quite a bit different (in a good way though):

```
    .ALIGN 32
InterruptVectorTable:
    DW InterruptHandler     ; 0 = line interrupt (highest priority)
    DW InterruptHandler     ; 1 = UART0 Rx
    DW InterruptHandler     ; 2 = UART1 Rx
    DW InterruptHandler     ; 3 = CTC channel 0
    DW InterruptHandler     ; 4 = CTC channel 1
    DW InterruptHandler     ; 5 = CTC channel 2
    DW InterruptHandler     ; 6 = CTC channel 3
    DW InterruptHandler     ; 7 = CTC channel 4
    DW InterruptHandler     ; 8 = CTC channel 5
    DW InterruptHandler     ; 9 = CTC channel 6
    DW InterruptHandler     ; 10 = CTC channel 7
    DW InterruptHandlerULA  ; 11 = ULA
    DW InterruptHandler     ; 12 = UART0 Tx
    DW InterruptHandler     ; 13 = UART1 Tx (lowest priority)
    DW InterruptHandler
    DW InterruptHandler
```

As you can see, each interrupter gets its own vector. In the above example, all except ULA are pointing to the same interrupt routine. Since there are only a handful, we can use a much clearer declaration with `DW <routine address>`. This way we don't need any initialization code that would fill in the routine address as we used with legacy IM2 mode.

The thing to note: `.ALIGN 32` is used to ensure the interrupt table is placed on a 32-byte boundary; the least significant 5 bits of the address must be `0` ($2^5 = 32 =$ `%100000`). This is important due to how hardware IM2 vector table lookup address is formed during interrupt:

High Byte								Low Byte							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`I` register: MSB								`$C0`: LSB 7-5			Interrupt specific				

As with legacy IM2 mode, the most significant byte of the vector table address is assigned to `I` register. But the least significant byte is not random. We provide the top 3 bits through **Interrupt Control** `$C0` (page 126) Next register, the rest are filled in based on interrupt type:

`0`	Line interrupt (highest priority)
`1`	UART0 Rx
`2`	UART1 Rx
`3-10`	CTC Channels 0-7
`11`	ULA
`12`	UART0 Tx
`13`	UART1 Tx (lowest priority)

Interrupt routines can reside anywhere in the memory.

You can find fully working example in companion code, folder `im2hw`.

With hardware IM2 mode it's possible to interrupt DMA operations. The choice of interrupters that can interrupt DMA is made with **DMA Interrupt Enable 0** `$CC` (page 128), **DMA Interrupt Enable 1** `$CD` (page 128) and **DMA Interrupt Enable 2** `$CE` (page 129) registers. Here's what Alvin Albrecht wrote about this possibility on Next Discord server:

> In this mode, it is also possible to program interrupters to interrupt DMA operations. Interrupting a DMA operation comes with a new caveat since the Z80 cannot see an interrupt until the end of an instruction. So if the DMA gives up the bus temporarily for an interrupt, then the Z80 will execute one instruction in the main program until the interrupt is seen. The interrups subroutine will execute and `RETI` will return control to the DMA. Anyway there is another rabbit hole here.

For details and more, see this discussion on Discord[20], highly recommended!

Very special thanks to the folks from Next Discord server: Alvin Albrecht for mentioning[21] hardware IM2 mode and `@varmfskii` whos discussion[22] and sample code was the basis for my exploration into this mode!

[20] `https://discord.com/channels/556228195767156758/692885312296190102/894284968614854749`

[21] `https://discord.com/channels/556228195767156758/692885312296190102/865955247552462848`

[22] `https://discord.com/channels/556228195767156758/692885353161293895/817807486744526886`

3.12.4 Interrupt Registers

Video Line Interrupt Control $22

Bit	Effect
7	Read: $\overline{\texttt{INT}}$ signal (even when Z80N has interrupts disabled) (1 = interrupt is requested) Write: Reserved, must be 0
6-3	Reserved, must be 0
2	1 disables original ULA interrupt (0 after reset)
1	1 enables Line Interrupt (0 after reset)
0	MSB of interrupt line value (0 after reset)

Line value starts with 0 for the first line of pixels. But the line-interrupt happens already when the previous line's pixel area is finished (i.e. the raster-line counter still reads ”previous line” and not the one programmed for interrupt). The $\overline{\texttt{INT}}$ signal is raised while display beam horizontal position is between 256-319 standard pixels, precise timing of interrupt handler execution then depends on how-quickly/if the Z80 will process the $\overline{\texttt{INT}}$ signal.

Video Line Interrupt Value $23

Bit	Effect
7-0	LSB of interrupt line value (0 after reset)

On core 3.1.5+ line numbering can be offset by **Vertical Video Line Offset** $64.

Vertical Video Line Offset $64

Bit	Effect
7-0	Vertical line offset value 0-255 added to Copper, Video Line Interrupt and Active Video Line readings.

Core 3.1.5+ only.

Normally the ULA's pixel row 0 aligns with vertical line count 0. With a non-zero offset, the ULA's pixel row 0 will align with the vertical line offset. For example, if the offset is 32 then video line 32 will correspond to the first pixel row in the ULA and video line 0 will align with the first pixel row of the Tilemap and Sprites (in the top border area).

Since a change in offset takes effect when the ULA reaches row 0, the change can take up to one frame to occur.

Interrupt Control $C0

Bit	Effect
7-5	Programmable portion of IM2 vector
4	Reserved, must be 0
3	1 enables, 0 disabled stackless NMI response
2-1	Reserved, must be 0
0	Maskable interrupt mode: 1 IM2, 0 pulse

If bit 3 is set, the return address pushed during an NMI acknowledge cycle will be written to **NMI Return Address LSB** $C2 and **NMI Return Address MSB** $C3 instead of the memory (the stack pointer will be decremented). The first RETN after the NMI acknowledge will then take its return address from the registers instead of memory (the stack pointer will be incremented). If bit 3 is 0, and in other circumstances (if there is no NMI first), RETN functions normally.

NMI Return Address LSB $C2

NMI Return Address MSB $C3

Bit	Effect
7-0	LSB or MSB of the return address written during an NMI acknowledge cycle

Interrupt Enable 0 $C4

Bit	Effect
7	Expansion bus $\overline{\texttt{INT}}$ (1 after reset)
6-2	Reserved, must be 0
1	1 enables, 0 disables line interrupt (0 after reset)
0	1 enables, 0 disabled ULA interrupt (1 after reset)

Interrupt Enable 1 $C5

Bit	Effect
7	1 enables, 0 disables CTC channel 7 interrupt
6	1 enables, 0 disables CTC channel 6 interrupt
5	1 enables, 0 disables CTC channel 5 interrupt
4	1 enables, 0 disables CTC channel 4 interrupt
3	1 enables, 0 disables CTC channel 3 interrupt
2	1 enables, 0 disables CTC channel 2 interrupt
1	1 enables, 0 disables CTC channel 1 interrupt
0	1 enables, 0 disables CTC channel 0 interrupt

All bits 0 after reset.

Interrupt Enable 2 `$C6`

Bit	Effect
7	Reserved, must be `0`
6	UART1 Tx empty
5	UART1 Rx half full
4	UART1 Rx available
3	Reserved, must be `0`
2	UART0 Tx empty
1	UART0 Rx half full
0	UART0 Rx available

All bits `0` after reset. Rx half full overrides Rx available.

Interrupt Status 0 `$C8`

Bit	Effect
7-2	Reserved, must be `0`
1	Line interrupt
0	ULA interrupt

Read indicates whether the given interrupt was generated in the past. Write with `1` clears given bit unless in IM2 mode (bit 0 set on **Interrupt Control** `$C0` (page 126)) with interrupts enabled.

Interrupt Status 1 `$C9`

Bit	Effect
7	CTC channel 7 interrupt
6	CTC channel 6 interrupt
5	CTC channel 5 interrupt
4	CTC channel 4 interrupt
3	CTC channel 3 interrupt
2	CTC channel 2 interrupt
1	CTC channel 1 interrupt
0	CTC channel 0 interrupt

Read indicates whether the given interrupt was generated in the past. Write with `1` clears given bit unless in IM2 mode (bit 0 set on **Interrupt Control** `$C0`, page 126) with interrupts enabled.

Interrupt Status 2 $CA

Bit	Effect
7	Reserved, must be 0
6	UART1 Tx empty interrupt
5	UART1 Rx half full interrupt
4	UART1 Rx available interrupt
3	Reserved, must be 0
2	UART0 Tx empty interrupt
1	UART0 Rx half full interrupt
0	UART0 Rx available interrupt

Read indicates whether the given interrupt was generated in the past. Write with 1 clears given bit unless in IM2 mode (bit 0 set on **Interrupt Control** $C0, page 126) with interrupts enabled.

DMA Interrupt Enable 0 $CC

Bit	Effect
7-2	Reserved, must be 0
1	1 allows, 0 prevents line interrupt from interrupting DMA
0	1 allows, 0 prevents ULA interrupt from interrupting DMA

DMA Interrupt Enable 1 $CD

Bit	Effect
7	1 allows, 0 prevents CTC channel 7 interrupt from interrupting DMA
6	1 allows, 0 prevents CTC channel 6 interrupt from interrupting DMA
5	1 allows, 0 prevents CTC channel 5 interrupt from interrupting DMA
4	1 allows, 0 prevents CTC channel 4 interrupt from interrupting DMA
3	1 allows, 0 prevents CTC channel 3 interrupt from interrupting DMA
2	1 allows, 0 prevents CTC channel 2 interrupt from interrupting DMA
1	1 allows, 0 prevents CTC channel 1 interrupt from interrupting DMA
0	1 allows, 0 prevents CTC channel 0 interrupt from interrupting DMA

DMA Interrupt Enable 2 `$CE`

Bit	Effect
7	Reserved, must be `0`
6	`1` allows, `0` prevents UART1 Tx empty from interrupting DMA
5	`1` allows, `0` prevents UART1 Rx half full from interrupting DMA
4	`1` allows, `0` prevents UART1 Rx available from interrupting DMA
3	Reserved, must be `0`
2	`1` allows, `0` prevents UART0 Tx empty from interrupting DMA
1	`1` allows, `0` prevents UART0 Rx half full from interrupting DMA
0	`1` allows, `0` prevents UART0 Rx available from interrupting DMA

This page intentionally left empty

Chapter 4

Instructions at a Glance

This chapter presents all instructions at a glance for quick info and to easily compare them when choosing the most optimal combination for the task at hand. Instructions are grouped into logical sections based on the area they operate on.

Instruction Execution

B Number of bytes instruction uses in RAM
Mc Number of machine cycles instruction takes to complete
Ts Number of clock periods instruction requires to complete

Flags (copied from section 2.1.5, page 11 as convenience)

SF **Sign** Set if 2-complement value is negative.
ZF **Zero** Set if the result is zero.
HY **Half-Carry** The half-carry of an addition/subtraction (from bit 3 to 4)*.
PV **Parity/Overflow** This flag can either be the parity of the result (`PF`), or 2-complement signed overflow (`VF`).
NF **Add/Subtract** Indicates the last operation was an addition (`0`) or a subtraction (`1`)*.
CF **Carry** Set if there was a carry from the most significant bit.

* Primarily used for BCD operations.

Effects

`0/1` Flag is set to `0` or `1`
↕ Flag is modified according to operation
`-` Flag is not affected
`?` Effect on flag is unpredictable
`VF` P/V flag is used as overflow
`PF` P/V flag is used as parity
• Special case, see description under the table or in chapter 5, page 147

Notes

`YF` and `XF` flags are not represented; they're irrelevant from the programmer point of view.

I used 4 sources for comparing effects: Z80 undocumented[1], Programming the Z80 third edition[2], Zilog Z80 manual[3] and Next Dev Wiki[4]. Where different and I couldn't verify, I opted for variant that matches most sources with slightly greater precedence for Next Dev Wiki side.

[1] `http://www.myquest.nl/z80undocumented/`
[2] `http://www.z80.info/zaks.html`
[3] `https://www.zilog.com/docs/z80/um0080.pdf`
[4] `https://wiki.specnext.dev/Extended_Z80_instruction_set`

4.1 8-Bit Arithmetic and Logical

	Symbolic	Flags						Opcode					
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76 543 210	Hex	B	Mc	Ts	Comments
ADD A,r	A←A+r	↕	↕	↕	VF	0	↕	10 $\boxed{000}$ ⇤r⇥	..	1B	1	4	r ⇤r⇥ B 000 C 001 D 010 E 011 H 100 L 101 A 111
ADD A,p	A←A+p	↕	↕	↕	VF	0	↕	11 011 101	DD	2B	2	8	
								10 $\boxed{000}$ ⇤p⇥	..				
ADD A,q	A←A+q	↕	↕	↕	VF	0	↕	11 111 101	FD	2B	2	8	
								10 $\boxed{000}$ ⇤q⇥	..				
ADD A,n	A←A+n	↕	↕	↕	VF	0	↕	11 $\boxed{000}$ 110	C6	2B	2	7	
								⇤--- n ---⇥	..				
ADD A,(HL)	A←A+(HL)	↕	↕	↕	VF	0	↕	10 $\boxed{000}$ 110	86	1B	2	7	
ADD A,(IX+d)	A←A+(IX+d)	↕	↕	↕	VF	0	↕	11 011 101	DD	3B	5	19	p ⇤p⇥ B 000 C 001 D 010 E 011 IX$_h$ 100 IX$_l$ 101 A 111
								10 $\boxed{000}$ 110	86				
								⇤--- d ---⇥	..				
ADD A,(IY+d)	A←A+(IY+d)	↕	↕	↕	VF	0	↕	11 111 101	FD	3B	5	19	
								10 $\boxed{000}$ 110	86				
								⇤--- d ---⇥	..				
								↑					
ADC A,s[2]	A←A+s+CF	↕	↕	↕	VF	0	↕	.. $\boxed{001}$ ...					
SUB s[2]	A←A-s	↕	↕	↕	VF	1	↕	.. $\boxed{010}$ ...					q ⇤q⇥ B 000 C 001 D 010 E 011 IY$_h$ 100 IY$_l$ 101 A 111
SBC A,s[2]	A←A-s-CF	↕	↕	↕	VF	1	↕	.. $\boxed{011}$ ...					
AND s[2]	A←A∧s	↕	↕	1	PF	0	0	.. $\boxed{100}$ ...					
XOR s[2]	A←A⊻s	↕	↕	0	PF	0	0	.. $\boxed{101}$ ...					
OR s[2]	A←A∨s	↕	↕	0	PF	0	0	.. $\boxed{110}$ ...					
CP s[1,2]	A-s	↕	↕	↕	VF	1	↕	.. $\boxed{111}$ ...					
INC r	r←r+1	↕	↕	↕	VF[4]	0	-	00 ⇤r⇥ $\boxed{100}$	..	1B	1	4	
INC p	p←p+1	↕	↕	↕	VF[4]	0	-	11 011 101	DD	2B	2	8	
								00 ⇤p⇥ $\boxed{100}$	..				
INC q	q←q+1	↕	↕	↕	VF[4]	0	-	11 111 101	FD	2B	2	8B	
								00 ⇤q⇥ $\boxed{100}$	..				
INC (HL)	(HL)←(HL)+1	↕	↕	↕	VF[4]	0	-	00 110 $\boxed{100}$	34	1B	3	11	
INC (IX+d)	(IX+d)←(IX+d)+1	↕	↕	↕	VF[4]	0	-	11 011 101	DD	3B	6	23	
								00 110 $\boxed{100}$	34				
								⇤--- d ---⇥					
INC (IY+d)	(IY+d)←(IY+d)+1	↕	↕	↕	VF[4]	0	-	11 111 101	FD	3B	6	23	
								00 110 $\boxed{100}$	34				
								⇤--- d ---⇥					
								↑					
DEC m[3]	m←m-1	↕	↕	↕	VF[5]	1	-	.. ... $\boxed{101}$					

Notes:

[1] YF and XF flags are copied from the operand s, not the result A-s

[2] s is any of r, p, q, n, (HL), (IX+d), (IY+d) as shown for ADD. Replace $\boxed{000}$ in the ADD set above. Ts also the same

[3] m is any of r, p, q, n, (HL), (IX+d), (IY+d) as shown for INC. Replace $\boxed{100}$ with $\boxed{101}$ in opcode. Ts also the same

[4] PV set if value was $7F before incrementing

[5] PV set if value was $80 before decrementing

4.2 16-Bit Arithmetic

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
ADC HL,rr	HL←HL+rr+CF	↕[1]	↕[1]	↕[2]	VF[1]	0	↕[1]	11	101	101	ED	2B	4	15	rr rr BC 00 DE 01 HL 10 SP 11
								01	rr1	010	..				
SBC HL,rr	HL←HL-rr-CF	↕[1]	↕[1]	↕[2]	VF[1]	1	↕[1]	11	101	101	ED	2B	4	15	
								01	rr0	010	..				
ADD HL,rr	HL←HL+rr	-	-	↕[2]	-	0	↕[1]	00	rr1	001	..	1B	3	11	
ADD IX,pp	IX←IX+pp	-	-	↕[2]	-	0	↕[1]	11	011	101	DD	2B	4	15	pp pp BC 00 DE 01 IX 10 SP 11
								00	pp1	001	..				
ADD IY,qq	IY←IY+qq	-	-	↕[2]	-	0	↕[1]	11	111	101	FD	2B	4	15	
								00	qq1	001	..				
INC rr	rr←rr+1	-	-	-	-	-	-	00	rr0	011	..	1B	1	6	
INC IX	IX←IX+1	-	-	-	-	-	-	11	011	101	DD	2B	2	10	qq qq BC 00 DE 01 IY 10 SP 11
								00	100	011	23				
INC IY	IY←IY+1	-	-	-	-	-	-	11	111	101	FD	2B	2	10	
								00	100	011	23				
DEC rr	rr←rr-1	-	-	-	-	-	-	00	rr1	011	..	1B	1	6	
DEC IX	IX←IX-1	-	-	-	-	-	-	11	011	101	DD	2B	2	10	
								00	101	011	2B				
DEC IY	IY←IY-1	-	-	-	-	-	-	11	111	101	FD	2B	2	10	
								00	101	011	2B				

Notes: [1] Flag is set by carry from bit 15
[2] Flag is set by carry from bit 11 (half carry in high byte)

4.3 8-Bit Load

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
LD r,r'	r←r'	–	–	–	–	–	–	01	⇤r⇥	⇤r'⇥	..	1B	1	4	r ⇤r⇥ r' ⇤r'⇥ B 000 C 001 D 010 E 011 H 100 L 101 A 111
LD p,p'	p←p'	–	–	–	–	–	–	11	011	101	DD	2B	2	8	
								01	⇤p⇥	⇤p'⇥	..				
LD q,q'	q←q'	–	–	–	–	–	–	11	111	101	FD	2B	2	8	
								01	⇤q⇥	⇤q'⇥	..				
LD r,n	r←n	–	–	–	–	–	–	00	⇤r⇥	110	..	2B	2	7	
								⇤--- n ---⇥			..				
LD p,n	p←n	–	–	–	–	–	–	11	011	101	DD	3B	3	11	p ⇤p⇥ p' ⇤p'⇥ B 000 C 001 D 010 E 011 IX_h 100 IX_l 101 A 111
								00	⇤p⇥	110	..				
								⇤--- n ---⇥			..				
LD q,n	q←n	–	–	–	–	–	–	11	111	101	FD	3B	3	11	
								00	⇤q⇥	110	..				
								⇤--- n ---⇥			..				
LD r,(HL)	r←(HL)	–	–	–	–	–	–	01	⇤r⇥	110	..	1B	2	7	
LD r,(IX+d)	r←(IX+d)	–	–	–	–	–	–	11	011	101	DD	3B	5	19	q ⇤q⇥ q' ⇤q'⇥ B 000 C 001 D 010 E 011 IY_h 100 IY_l 101 A 111
								01	⇤r⇥	110	..				
								⇤--- d ---⇥			..				
LD r,(IY+d)	r←(IY+d)	–	–	–	–	–	–	11	111	101	FD	3B	5	19	
								01	⇤r⇥	110	..				
								⇤--- d ---⇥			..				
LD (HL),r	(HL)←r	–	–	–	–	–	–	01	110	⇤r⇥	..	1B	2	7	
LD (IX+d),r	(IX+d)←r	–	–	–	–	–	–	11	011	101	DD	3B	5	19	
								01	110	⇤r⇥	..				
								⇤--- d ---⇥			..				
LD (IY+d),r	(IY+d)←r	–	–	–	–	–	–	11	111	101	FD	3B	5	19	
								01	110	⇤r⇥	..				
								⇤--- d ---⇥			..				
LD (HL),n	(HL)←n	–	–	–	–	–	–	00	110	110	36	2B	3	10	
								⇤--- n ---⇥			..				
LD (IX+d),n	(IX+d)←n	–	–	–	–	–	–	11	011	101	DD	4B	5	19	
								00	110	110	36				
								⇤--- d ---⇥			..				
								⇤--- n ---⇥			..				
LD (IY+d),n	(IY+d)←n	–	–	–	–	–	–	11	111	101	FD	4B	5	19	
								00	110	110	36				
								⇤--- d ---⇥			..				
								⇤--- n ---⇥			..				
LD A,(BC)	A←(BC)	–	–	–	–	–	–	00	001	010	0A	1B	2	7	
LD A,(DE)	A←(DE)	–	–	–	–	–	–	00	011	01	1A	1B	2	7	
LD A,(nm)	A←(nm)	–	–	–	–	–	–	00	111	010	3A	3B	4	13	
								⇤--- m ---⇥			..				
								⇤--- n ---⇥			..				

(continued on next page)

	Symbolic	Flags						Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
`LD (BC),A`	`(BC)←A`	-	-	-	-	-	-	00	000	010	02	1B	2	7	
`LD (DE),A`	`(DE)←A`	-	-	-	-	-	-	00	010	010	12	1B	2	7	
`LD (nm),A`	`(nm)←A`	-	-	-	-	-	-	00	110	010	32	3B	4	13	
								⇤--- m ---⇥			..				
								⇤--- n ---⇥			..				
`LD A,I`	`A←I`	↕	↕	0	IFF2	0	-	11	101	101	ED	2B	2	9	
								01	010	111	57				
`LD A,R`	`A←R`	↕	↕	0	IFF2	0	-	11	101	101	ED	2B	2	9	
								01	011	111	5F				
`LD I,A`	`I←A`	-	-	-	-	-	-	11	101	101	ED	2B	2	9	
								01	000	111	47				
`LD R,A`	`R←A`	-	-	-	-	-	-	11	101	101	ED	2B	2	9	
								01	001	111	4F				

4.4 General-Purpose Arithmetic and CPU Control

	Symbolic	Flags						Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
`DAA`		↕	↕	↕	PF	-	↕	00	100	111	27	1B	1	4	
`CPL`	`A←` $\overline{\texttt{A}}$	-	-	1	-	1	-	00	101	111	2F	1B	1	4	
`NEG`	`A←-A`	↕	↕	↕	PF	1	↕	11	101	101	ED	2B	2	8	
								01	000	100	44				
`CCF`[1]	`CF←` $\overline{\texttt{CF}}$	-	-	•[2]	-	0	↕	00	111	111	3F	1B	1	4	
`SCF`[1]	`CF←1`	-	-	0	-	0	1	00	110	111	37	1B	1	4	
`NOP`		-	-	-	-	-	-	00	000	000	00	1B	1	4	
`HALT`		-	-	-	-	-	-	01	110	110	76	1B	1	4	
`DI`[3]	`IFF1←0` `IFF2←0`	-	-	-	-	-	-	11	110	011	F3	1B	1	4	
`EI`[3]	`IFF1←1` `IFF2←1`	-	-	-	-	-	-	11	111	011	FB	1B	1	4	
`IM 0`[4]		-	-	-	-	-	-	11	101	101	ED	2B	2	8	
								01	000	110	46				
`IM 1`[4]		-	-	-	-	-	-	11	101	101	ED	2B	2	8	
								01	010	110	56				
`IM 2`[4]		-	-	-	-	-	-	11	101	101	ED	2B	2	8	
								01	011	110	5E				

Notes:

[1] YF and XF are copied from register `A`

[2] Documentation says original value of CF is copied to HF, but my tests show that **HF** remains unchanged

[3] No interrupts are accepted directly after `EI` or `DI`

[4] This instruction has other undocumented opcodes

4.5 16-Bit Load

	Symbolic			Flags				Opcode					
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76 543 210	Hex	B	Mc	Ts	Comments
LD rr,nm	rr←nm	–	–	–	–	–	–	00 rr0 001 ←--- m ---→ ←--- n ---→	.. .. ..	3B	3	10	rr rr BC 00 DE 01 HL 10 SP 11
LD IX,nm	IX←nm	–	–	–	–	–	–	11 011 101 00 100 001 ←--- m ---→ ←--- n ---→	DD 21	4B	4	14	
LD IY,nm	IX←nm	–	–	–	–	–	–	11 111 101 00 100 001 ←--- m ---→ ←--- n ---→	FD 21	4B	4	14	
LD HL,(nm)	H←(nm+1) L←(nm)	–	–	–	–	–	–	00 101 010 ←--- m ---→ ←--- n ---→	2A	3B	5	16	
LD rr,(nm)	$\mathrm{rr_h}$←(nm+1) $\mathrm{rr_l}$←(nm)	–	–	–	–	–	–	11 101 101 01 rr1 011 ←--- m ---→ ←--- n ---→	ED	4B	6	20	
LD IX,(nm)	$\mathrm{IX_h}$←(nm+1) $\mathrm{IX_l}$←(nm)	–	–	–	–	–	–	11 011 101 00 101 010 ←--- m ---→ ←--- n ---→	DD 2A	4B	6	20	
LD IY,(nm)	$\mathrm{IY_h}$←(nm+1) $\mathrm{IY_l}$←(nn)	–	–	–	–	–	–	11 111 101 00 101 010 ←--- n ---→ ←--- n ---→	FD 2A	4B	6	20	
LD (nm),HL	(nn+1)←H (nm)←L	–	–	–	–	–	–	00 100 010 ←--- m ---→ ←--- n ---→	22	3B	5	16	
LD (nm),rr	(nm+1)←$\mathrm{rr_h}$ (nm)←$\mathrm{rr_l}$	–	–	–	–	–	–	11 101 101 01 rr0 011 ←--- m ---→ ←--- n ---→	ED	4B	6	20	
LD (nm),IX	(nm+1)←$\mathrm{IX_h}$ (nm)←$\mathrm{IX_l}$	–	–	–	–	–	–	11 011 101 00 100 010 ←--- m ---→ ←--- n ---→	DD 22	4B	6	20	
LD (nm),IY	(nm+1)←$\mathrm{IY_h}$ (nm)←$\mathrm{IY_l}$	–	–	–	–	–	–	11 111 101 00 100 010 ←--- m ---→ ←--- n ---→	FD 22	4B	6	20	
LD SP,HL	SP←HL	–	–	–	–	–	–	11 111 001	F9	1B	1	6	
LD SP,IX	SP←IX	–	–	–	–	–	–	11 011 101 11 111 001	DD F9	2B	2	10	
LD SP,IY	SP←IY	–	–	–	–	–	–	11 111 101 11 111 001	FD F9	2B	2	10	

4.6 Stack

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
POP pp	pp$_h$←(SP+1) pp$_l$←(SP) SP←SP+2	-	-	-	-	-	-	11	pp0	001	..	1B	3	10	pp pp BC 00 DE 01 HL 10
POP AF	A←(SP+1) F←(SP) SP←SP+2	↕^{1}	↕^{1}	↕^{1}	↕^{1}	↕^{1}	↕^{1}	11	110	001	F1	1B	3	10	
POP IX	IX$_h$←(SP+1) IX$_l$←(SP) SP←SP+2	-	-	-	-	-	-	11 11	011 100	101 001	DD E1	2B	4	14	
POP IY	IY$_h$←(SP+1) IY$_l$←(SP) SP←SP+2	-	-	-	-	-	-	11 11	111 100	101 001	FD E1	2B	4	14	
PUSH rr	(SP-2)←rr$_l$ (SP-1)←rr$_h$ SP←SP-2	-	-	-	-	-	-	11	rr0	101	..	1B	3	11	rr rr BC 00 DE 01 HL 10 AF 11
PUSH IX	(SP-2)←IX$_l$ (SP-1)←IX$_h$ SP←SP-2	-	-	-	-	-	-	11 11	011 100	101 101	DD E5	2B	4	15	
PUSH IY	(SP-2)←IY$_l$ (SP-1)←IY$_h$ SP←SP-2	-	-	-	-	-	-	11 11	111 100	101 101	FD E5	2B	4	15	

Notes: [1] Flags set directly to low 8-bits of the value from stack SP

4.7 Exchange

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
EX AF,AF'	AF↔AF'	•^{1}	•^{1}	•^{1}	•^{1}	•^{1}	•^{1}	00	001	000	08	1B	1	4	
EX DE,HL	DE↔HL	-	-	-	-	-	-	11	101	011	EB	1B	1	4	
EX (SP),HL	H↔(SP+1) L↔(SP)	-	-	-	-	-	-	11	100	011	E3	1B	5	19	
EX (SP),IX	IX$_h$↔(SP+1) IX$_l$↔(SP)	-	-	-	-	-	-	11 11	011 100	101 011	DD E3	2B	6	2	
EX (SP),IY	IY$_h$↔(SP+1) IY$_l$↔(SP)	-	-	-	-	-	-	11 11	111 100	101 011	FD E3	2B	6	23	
EXX	BC↔BC' DE↔DE' HL↔HL'	-	-	-	-	-	-	11	011	001	D9	1B	1	4	

Notes: [1] Flags set directly from the value of F'

4.8 Bit Set, Reset and Test

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
BIT b,r	ZF← $\overline{r_b}$	?[1]	↕	1	?[1]	0	–	11	001	011	CB	2B	2	8	r ⇤r⇥
								01	⇤b⇥	⇤r⇥	..				B 000 C 001
BIT b,(HL)	ZF← $\overline{(HL)_b}$	?[1]	↕	1	?[1]	0	–	11	001	011	CB	2B	3	12	D 010
								01	⇤b⇥	110	..				E 011 H 100
BIT b,(IX+d)[2]	ZF← $\overline{(IX+d)_b}$	?[1]	↕	1	?[1]	0	–	11	011	101	DD	4B	5	20	L 101
								11	001	011	CB				A 111
								⇤--- d ---⇥			..				
								01	⇤b⇥	110	..				
BIT b,(IY+d)[2]	ZF← $\overline{(IY+d)_b}$	?[1]	↕	1	?[1]	0	–	11	111	101	FD	4B	5	20	b ⇤b⇥
								11	001	011	CB				0 000
								⇤--- d ---⇥			..				1 001
								01	⇤b⇥	110	..				2 010 3 011
SET b,r	r_b ←1	–	–	–	–	–	–	11	001	011	CB	2B	2	8	4 100 5 101
								$\boxed{11}$	⇤b⇥	⇤r⇥	..				6 110 7 111
SET b,(HL)	$(HL)_b$ ←1	–	–	–	–	–	–	11	001	011	CB	2B	4	15	
								$\boxed{11}$	⇤b⇥	110	..				
SET b,(IX+d)	$(IX+d)_b$ ←1	–	–	–	–	–	–	11	011	101	DD	4B	6	23	
								11	001	011	CB				
								⇤--- d ---⇥			..				
								$\boxed{11}$	⇤b⇥	110	..				
SET b,(IY+d)	$(IY+d)_b$ ←1	–	–	–	–	–	–	11	111	101	FD	4B	6	23	
								11	001	011	CB				
								⇤--- d ---⇥			..				
								$\boxed{11}$	⇤b⇥	110	..				
SET b,(IX+d),r	r←(IX+d)	–	–	–	–	–	–	11	011	101	DD	4B	6	23	
	r_b ←1							11	001	011	CB				
	(IX+d)←r							⇤--- d ---⇥			..				
								$\boxed{11}$	⇤b⇥	⇤r⇥	..				
SET b,(IY+d),r	r←(IY+d)	–	–	–	–	–	–	11	111	101	FD	4B	6	23	
	r_b ←1							11	001	011	CB				
	(IY+d)←r							⇤--- d ---⇥			..				
								$\boxed{11}$	⇤b⇥	⇤r⇥	..				
								↑							
RES b,m[3]	m_b ←0	–	–	–	–	–	–	$\boxed{10}$	...	...					

Notes: [1] See section 2.3.1, page 16 for complete description

[2] Instruction has other undocumented opcodes

[3] m is one of r, (HL), (IX+d), (IY+d). To form RES instruction, replace $\boxed{11}$ with $\boxed{10}$. Ts also the same

4.9 Rotate and Shift

	Symbolic	Flags						Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
`RLC r`	CF←7←0←	↕	↕	0	PF	0	↕	11	001	011	CB	2B	2	8	`r` ⊢r⊣ `B` 000 `C` 001 `D` 010 `E` 011 `H` 100 `L` 101 `A` 111
								00	$\boxed{000}$	⊢r⊣	..				
`RLC (HL)`	CF←7←0←	↕	↕	0	PF	0	↕	11	001	011	CB	2B	4	15	
								00	$\boxed{000}$	110	06				
`RLC (IX+d)`	CF←7←0←	↕	↕	0	PF	0	↕	11	011	101	DD	4B	6	23	
								11	001	011	CB				
								⊢---	d	---⊣	..				
								00	$\boxed{000}$	110	06				
`RLC (IY+d)`	CF←7←0←	↕	↕	0	PF	0	↕	11	111	101	FD	4B	6	23	
								11	001	011	CB				
								⊢---	d	---⊣	..				
								00	$\boxed{000}$	110	06				
`RLC r,(IX+d)`	`r←(IX+d)`	↕	↕	0	PF	0	↕	11	011	101	DD	4B	6	23	
	`RLC r`							11	001	011	CB				
	`(IX+d)←r`							⊢---	d	---⊣	..				
								00	$\boxed{000}$	⊢r⊣	..				
`RLC r,(IY+d)`	`r←(IY+d)`	↕	↕	0	PF	0	↕	11	111	101	FD	4B	6	23	
	`RLC r`							11	001	011	CB				
	`(IY+d)←r`							⊢---	d	---⊣	..				
								00	$\boxed{000}$	⊢r⊣	..				
									↑						
`RRC m`[1]	→7→0→CF	↕	↕	0	PF	0	↕	..	$\boxed{001}$	...					
`RL m`[1]	CF←7←0←	↕	↕	0	PF	0	↕	..	$\boxed{010}$	...					
`RR m`[1]	→7→0→CF	↕	↕	0	PF	0	↕	..	$\boxed{011}$	...					
`SLA m`[1]	CF←7←0←0	↕	↕	0	PF	0	↕	..	$\boxed{100}$	...					
`SRA m`[1]	→7→0→CF	↕	↕	0	PF	0	↕	..	$\boxed{101}$	...					
`SLI m`[1,2]	CF←7←0←1	↕	↕	0	PF	0	↕	..	$\boxed{110}$	...					
`SRL m`[1]	0→7→0→CF	↕	↕	0	PF	0	↕	..	$\boxed{111}$	...					
`SLL m`[3]															
`RLA`	CF←7←0←	-	-	0	-	0	↕	00	010	111	17	1B	1	4	
`RLCA`	CF←7←0←	-	-	0	-	0	↕	00	000	111	07	1B	1	4	
`RRA`	→7→0→CF	-	-	0	-	0	↕	00	011	111	1F	1B	1	4	
`RRCA`	→7→0→CF	-	-	0	-	0	↕	00	001	111	0F	1B	1	4	
`RLD`	`A 7-4 3-0  7-4 3-0 (HL)`	↕	↕	0	PF	0	-	11	101	101	ED	2B	5	18	
								01	101	111	6F				
`RRD`	`A 7-4 3-0  7-4 3-0 (HL)`	↕	↕	0	PF	0	-	11	101	101	ED	2B	5	18	
								01	100	111	67				

Notes: [1] `m` is one of `r`, `(HL)`, `(IX+d)`, `(IY+d)`. To form new opcode replace $\boxed{000}$ of `RLC`s with shown code. Ts also the same

[2] Some assemblers may also allow `SL1` to be used instead of `SLI`

[3] Shift Left Logical; no associated opcode, there is no difference between logical and arithmetic shift left, use `SLA` for both. Some assemblers will allow `SLL` as equivalent, but unfortunately some will assemble it as `SLI`, so it's best avoiding

4.10 Jump

	Symbolic	Flags						Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
JP nm	PC←nm	-	-	-	-	-	-	11	000	011	C3	3B	3	10	c ⇤c⇥
								⇤---	m	---⇥	..				NZ 000
								⇤---	n	---⇥	..				Z 001
															NC C10
JP (HL)	PC←HL	-	-	-	-	-	-	11	101	001	E9	1B	1	4	C 011
															PO 100
JP (IX)	PC←IX	-	-	-	-	-	-	11	011	101	DD	2B	2	8	PE 101
								11	101	001	E9				P 110
															M 111
JP (IY)	PC←IY	-	-	-	-	-	-	11	111	101	FD	2B	2	8	
								11	101	001	E9				
JP c,nm	if c=true: JP nm	-	-	-	-	-	-	11	⇤c⇥	010	..	3B	3	10	p pp
								⇤---	m	---⇥	..				NZ 00
								⇤---	n	---⇥	..				Z 01
															NC 10
JR e	PC←PC+e	-	-	-	-	-	-	00	011	000	18	2B	3	12	C 11
								⇤--	e-2	--⇥	..				
JR p,e	if p=true: JR e	-	-	-	-	-	-	00	1pp	000	..	2B	2	7	if p=false
								⇤--	e-2	--⇥	..		3	12	if p=true
DJNZ e	B←B-1	-	-	-	-	-	-	00	010	000	10	2B	2	8	if B=0
	if B≠0: JR e							⇤--	e-2	--⇥	..		3	13	if B≠0

Notes: e is a signed two-complement in the range -127, 129.
e-2 in the opcode provides an effective number of PC+e as PC is incremented by two prior to the addition of e.

4.11 Call and Return

	Symbolic	Flags						Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
CALL nm	(SP-1)←PC$_h$	-	-	-	-	-	-	11	001	101	CD	3B	5	17	
	(SP-2)←PC$_l$							⇤--- m ---⇥			..				
	SP←SP-2							⇤--- n ---⇥			..				
	PC←nm														
CALL c,nm	if c=true: CALL nm	-	-	-	-	-	-	11	⇤c⇥	100	..	3B	3	10	if c=false
								⇤--- m ---⇥			..		5	17	if c=true
								⇤--- n ---⇥			..				
RET	PC$_l$←(SP)	-	-	-	-	-	-	11	001	001	C9	1B	3	10	
	PC$_h$←(SP+1)														
	SP←SP+2														
RET c	if c=true: RET	-	-	-	-	-	-	11	⇤c⇥	000	..	1B	1	5	if c=false
													3	11	if c=true
RETI[1]	PC$_l$←(SP)	-	-	-	-	-	-	11	101	101	ED	2B	4	14	c ⇤c⇥ NZ 000 Z 001 NC 010 C 011 PO 100 PE 101 P 110 M 111
	PC$_h$←(SP+1)							01	001	101	4D				
	SP←SP+2														
RETN[2]	PC$_l$←(SP)	-	-	-	-	-	-	11	101	101	ED	2B	4	14	
	PC$_h$←(SP+1)							01	000	101	45				
	SP←SP+2														
	IFF1←IFF2														
RST p	(SP-1)←PC$_h$	-	-	-	-	-	-	11	⇤p⇥	111	..	1B	3	11	p ⇤p⇥ $0 000 $8 001 $10 010 $18 011 $20 100 $28 101 $30 110 $38 111
	(SP-2)←PC$_l$														
	SP←SP-2														
	PC←p														

Notes:

[1] RETI also copies IFF2 into IFF1, like RETN

[2] This instruction has other undocumented opcodes

4.12 Block Transfer, Search

	Symbolic			Flags				Opcode							
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
CPD	A-(HL)	$\updownarrow^{1}$	$\updownarrow^{2}$	$\updownarrow^{1}$	$\bullet^{3}$	1	-	11	101	101	ED	2B	4	16	
	HL←HL-1							10	101	001	A9				
	BC←BC-1														
CPDR	do CPD	$\updownarrow^{1}$	$\updownarrow^{2}$	$\updownarrow^{1}$	$\bullet^{3}$	1	-	11	101	101	ED	2B	4	16	if A=(HL)
	while A≠(HL)∧BC>0							10	111	001	B9				or BC=0
													5	21	if A≠(HL)
															and BC≠0
CPI	A-(HL)	$\updownarrow^{1}$	$\updownarrow^{2}$	$\updownarrow^{1}$	$\bullet^{3}$	1	-	11	101	101	ED	2B	4	16	
	HL←HL+1							10	100	001	A1				
	BC←BC-1														
CPIR	do CPI	$\updownarrow^{1}$	$\updownarrow^{2}$	$\updownarrow^{1}$	$\bullet^{3}$	1	-	11	101	101	ED	2B	4	16	if A=(HL)
	while A≠(HL)∧BC>0							10	110	001	B1				or BC=0
													5	21	if A≠(HL)
															and BC≠0
LDD	(DE)←(HL)	-	-	0	$\bullet^{3}$	0	-	11	101	101	ED	2B	4	16	
	DE←DE-1							10	101	000	A8				
	HL←HL-1														
	BC←BC-1														
LDDR	do LDD	-	-	0	0^{4}	0	-	11	101	101	ED	2B	4	16	if BC=0
	while BC>0							10	111	000	B8		5	21	if BC≠0
LDI	(DE)←(HL)	-	-	0	$\bullet^{3}$	0	-	11	101	101	ED	2B	4	16	
	DE←DE+1							10	100	000	A0				
	HL←HL+1														
	BC←BC-1														
LDIR	do LDI	-	-	0	0^{4}	0	-	11	101	101	ED	2B	4	16	if BC=0
	while BC>0							10	110	000	B0		5	21	if BC≠0

Notes:
[1] See section 2.3.2, page 17 for a description
[2] ZF is 1 if A=(HL), otherwise 0
[3] PV is 1 if BC≠0 after execution, otherwise 0
[4] PV is 0 only at the completion of the instruction

4.13 Input

	Symbolic			Flags					Opcode						
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
IN A,(n)[1]	A←(n)	-	-	-	-	-	-	11	011	011	DB	2B	3	11	r ⇤r⇥
								⇤--- n ---⇥			..				B 000
IN r,(C)[2]	r←(BC)	↕	↕	0	PF	0	-	11	101	101	ED	2B	3	12	C 001 D 010
								01	⇤r⇥	000	..				E 011
IN (C)[2,3]	(BC)	↕	↕	0	PF	0	-	11	101	101	ED	2B	3	12	H 100 L 101
								01	110	000	70				A 111
IND	(HL)←(BC)	●[5]	●[4]	●[5]	●[5]	1	-	11	101	101	ED	2B	4	16	
	HL←HL-1							10	101	010	AA				
	B←B-1														
INDR	do IND	●[5]	1	●[5]	●[5]	1	-	11	101	101	ED	2B	4	16	if B=0
	while B>0							10	111	010	BA		5	21	if B≠0
INI	(HL)←(BC)	●[5]	●[4]	●[5]	●[5]	1	-	11	101	101	ED	2B	4	16	
	HL←HL+1							10	100	010	A2				
	B←B-1														
INIR	do INI	●[5]	1	●[5]	●[5]	1	-	11	101	101	ED	2B	4	16	if B=0
	while B>0							10	110	010	B2		5	21	if B≠0

Notes: [1] Some assemblers allow `IN (n)` to be used instead of `IN A,(n)`
[2] Some assemblers allow instruction to be written with `(BC)` instead of `(C)`
[3] Performs the input without storing the result. Some assemblers allow `IN F,(C)` to be used instead of `IN (C)`
[4] Flag is 1 if B=0 after execution, otherwise 0; similar to `DEC B`
[5] On Next this flag is destroyed, for other Z80 computers see section 2.3.3, page 17

4.14 Output

	Symbolic			Flags					Opcode						
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
OUT (n),A	(n)←A	-	-	-	-	-	-	11	010	011	D3	2B	3	11	r ⇤r⇥
								⇤--- n ---⇥			..				B 000
OUT (C),r	(BC)←r	-	-	-	-	-	-	11	101	101	ED	2B	3	12	C 001 D 010
								01	⇤r⇥	001	..				E 011
OUT (C),0	(BC)←0	-	-	-	-	-	-	11	101	101	ED	2B	3	12	H 100 L 101
								01	110	001	71				A 111
OUTI	B←B-1	●[2]	●[1]	●[2]	●[2]	1	-	11	101	101	ED	2B	4	16	
	(BC)←(HL)							10	100	011	A3				
	HL←HL+1														
OTIR	do OUTI	●[2]	1	●[2]	●[2]	1	-	11	101	101	ED	2B	4	16	if B=0
	while B>0							10	110	011	B3		5	21	if B≠0
OUTD	B←B-1	●[2]	●[1]	●[2]	●[2]	1	-	11	101	101	ED	2B	4	16	
	(BC)←(HL)							10	101	011	AB				
	HL←HL-1														
OTDR	do OUTD	●[2]	1	●[2]	●[2]	1	-	11	101	101	ED	2B	4	16	if B=0
	while B>0							10	111	011	BB		5	21	if B≠0

Notes: [1] Flag is 1 if B=0 after execution, otherwise 0
[2] On Next this flag is destroyed, for other Z80 computers see section 2.3.3, page 17.

4.15 ZX Spectrum Next Extended

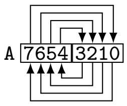
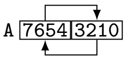
	Symbolic			Flags				Opcode						
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76 543 210	Hex	B	Mc	Ts	Comments
ADD rr,A	rr←rr+A	-	-	-	-	-	?[1]	11 101 101	ED	2B	2	8	rr rr
								00 110 0rr	..				HL 01 DE 10
ADD pp,nm	pp←pp+nm	-	-	-	-	-	-	11 101 101	ED	2B	4	16	BC 11
								00 110 1pp	..				pp pp
								⇤--- m ---⇥	..				HL 00
								⇤--- n ---⇥	..				DE 01 BC 10
BSLA DE,B[2]	DE←DE<<(B∧$1F)	-	-	-	-	-	-	11 101 101	ED	2B	2	8	
								00 101 000	28				
BSRA DE,B[2]	DE←signed(DE)...	-	-	-	-	-	-	11 101 101	ED	2B	2	8	
	...>>(B∧$1F)							00 101 001	29				
BSRL DE,B[2]	DE←unsigned(DE)...	-	-	-	-	-	-	11 101 101	ED	2B	2	8	
	...>>(B∧$1F)							00 101 010	2A				
BSRF DE,B[2]	DE←∼(unsigned(∼DE)...	-	-	-	-	-	-	11 101 101	ED	2B	2	8	
	...>>(B∧$1F))							00 101 011	2B				
BRLC DE,B[2]	DE←DE<<(B∧$0F or	-	-	-	-	-	-	11 101 101	ED	2B	2	8	
	DE←DE>>(16-B∧$0F)							00 101 100	2C				
JP (C)	PC←PC∧$C000+IN(C)<<6	?	?	?	?	?	?	11 101 101	ED	2B	3	13	
								10 011 000	98				
LDDX	if (HL)≠A: (DE)←(HL)	-	-	-	-	-	-	11 101 101	ED	2B	4	16	
	DE←DE+1							10 101 100	AC				
	HL←HL-1												
	BC←BC-1												
LDDRX	do LDDX	-	-	-	-	-	-	11 101 101	ED	2B	4	16	if BC=0
	while BC>0							10 111 100	BC		5	21	if BC≠0
LDIX	if (HL)≠A: (DE)←(HL)	-	-	-	-	-	-	11 101 101	ED	2B	4	16	
	DE←DE+1							10 100 100	A4				
	HL←HL+1												
	BC←BC-1												
LDIRX	do LDIX	-	-	-	-	-	-	11 101 101	ED	2B	4	16	if BC=0
	while BC>0							10 110 100	B4		5	21	if BC≠0
LDPIRX	do	-	-	-	-	-	-	11 101 101	ED	2B	4	16	if BC=0
	t←(HL∧$FFF8+E∧7)							10 110 111	B7		5	21	if BC≠0
	if t≠A: (DE)←t												
	DE←DE+1												
	BC←BC-1												
	while BC>0												
LDWS	(DE)←(HL)	↕	↕	↕	VF[3]	0	-	11 101 101	ED	2B	4	16	
	INC L							10 100 101	A5				
	INC D												

Notes: [1] CF is undefined, recently discovered, thanks to Peter Ped Helcmanovsky https://discord.com/channels/556228195767156758/695180116040351795/888099852725133412

[2] Core v2+ only

[3] PV set to 1 if D was $7F before increment, otherwise 0

(continued on next page)

	Symbolic			Flags					Opcode						
Mnemonic	Operation	SF	ZF	HF	PV	NF	CF	76	543	210	Hex	B	Mc	Ts	Comments
`MIRROR A`		-	-	-	-	-	-	11 00	101 100	101 100	ED 24	2B	2	8	
`MUL D,E`	`DE←D×E`	-	-	-	-	-	-	11 00	101 110	101 000	ED 30	2B	2	8	
`NEXTREG n,A`	`HwNextReg[n]←A`	-	-	-	-	-	-	11 10 |←---	101 010 n	101 010 ---→|	ED 92 ..	3B	4	17	
`NEXTREG n,m`	`HwNextReg[n]←m`	-	-	-	-	-	-	11 10 |←--- |←---	101 010 n m	101 001 ---→| ---→|	ED 91	3B	5	20	
`OUTINB`	`(BC)←(HL)` `HL←HL+1`	?	?	?	?	?	?	11 10	101 010	101 000	ED 90	2B	4	16	
`PIXELAD`	`HL←$4000...` `...+((D∧$C0)<<5)` `...+((D∧$07)<<8)` `...+((D∧$38)<<2)` `...+(E>>3)`	-	-	-	-	-	-	11 10	101 010	101 100	ED 94	3B	2	8	
`PIXELDN`	`if (HL∧$700)≠$700` `  HL←HL+256` `else if (HL∧$E0)≠$E0` `  HL←HL∧$F8FF+$20` `else` `  HL←HL∧$F81F+$800`	-	-	-	-	-	-	11 10	101 010	101 011	ED 93	3B	2	8	
`PUSH nm`	`(SP-2)←m` `(SP-1)←n` `SP←SP-2`	-	-	-	-	-	-	11 10 |←--- |←---	101 001 n[1] m[1]	101 010 ---→| ---→|	ED 8A	3B	6	23	
`SETAE`	`A←unsigned($80)>>(E∧7)`	-	-	-	-	-	-	11 10	101 010	101 101	ED 95	3B	2	8	
`SWAPNIB`		-	-	-	-	-	-	11 00	101 100	101 011	ED 23	2B	2	8	
`TEST n`	`A∧n`	↕	↕	1	PF	?	0	11 00	101 100	101 111	ED 27	3B	3	11	

Notes: [1] This is not mistake, `nm` operand is in fact encoded in big-endian

Chapter 5

Instructions up Close

The following pages describe all instructions in detail. Alphabetical order is used as much as possible, but some deviations were made to better fit to pages. Each instruction includes:

- Mnemonic
- Symbolic operation for quick info on what instruction does
- All variants (where applicable)
- Description with further details
- Effects on flags
- Timing table with machine cycles, T states and time required for execution on different CPU speeds

Where possible, multiple variants of same instruction are grouped together and where multiple timings are possible, timing table is sorted from quickest to slowest.

Flags

SF **Sign Flag** is set to twos-complement of the most-significant bit (bit 7) of the result of an instruction. If the result is positive (bit 7 is 0), **SF** is set, and if the result is negative (bit 7 is 1), **SF** is reset. This leaves bits 0-6 to represent the value. Positive numbers range from 0 to 127 and negative from -1 to -128.

ZF **Zero Flag** depends on whether the result of an instruction is 0. **ZF** is set if the result if 0 and reset otherwise.

HF **Half Carry Flag** represents a carry or borrow status between bits 3 and 4 of an 8-bit arithmetic operation (bits 11 and 12 for 16-bit operations). Set if:

- A carry from bit 3 to bit 4 occurs during addition (bit 11 to 12 for 16-bit operations)
- A borrow from bit 4 occurs during subtraction (from bit 12 for 16-bit operations)

PV **Parity/Overflow Flag** value depends on the type of the operation.

For arithmetic operations, **PV** indicates an overflow. The flag is set when the sign of the result is different from the sign of the operands:

- all operands are positive but the result is negative or
- all operands are negative but the result is positive

For logical and rotate operations, **PV** indicates the parity of the result. The number of set bits in the result are counted. If the total is an even value, **PV** is set. If the total is odd, **PV** is reset.

NF **Add/Subtract Flag** is used primarily for `DAA` instruction to distinguish between add and subtract operations. But other instructions may also affect it as described in the following pages.

CF **Carry Flag** represents a carry or borrow status for arithmetic operations. **CF** is set if add instruction generates a carry, or subtract generates a borrow.

For rotate and shift instructions, **CF** is used:

- as a link between least-significat and most significant bit for `RLA`, `RL`, `RRA` and `RR`
- contains the value shifted out of bit 7 for `RLC`, `RLCA` and `SLA`
- contains the value shifted out of bit 0 for `RRC`, `RRCA`, `SRA` and `SRL`

Finally, some instructions directly affect the value of **CF**:

- reset with `AND`, `OR` and `XOR`
- set with `SCF`
- completed with `CCF`

Effects

`0`	Flag is set to `0`
`1`	Flag is set to `1`
↕	Flag is modified according to operation
`-`	Flag is not affected
`?`	Effect on flag is unpredictable
•	Special case, see notes below effects table
PⓋ	P/V flag is used as overflow
ⓅV	P/V flag is used as parity
PV	P/V is undefined or indicates other result

Abbreviations

`r`	8-bit register `A-L`
`n`	8-bit immediate value
`rr`	16-bit register pair `AF`, `BC`, `DE`, `HL`, `IX`, `IY`, `SP` (note in some cases particular register pairs may use different timing from the rest; if so, those will be explicitly indicated in their own line; `rr` may still be used, though in those cases it will cover the remaining registers only)
`nn`	16-bit immediate value
`s`	Placeholder for argument when multiple variants are possible
`d`	If instruction takes 2 operands, `d` indicates destination and `s` source
**	Indicates undocumented instruction
$^{\text{ZX}}$	Indicates ZX Spectrum Next extended instruction

This page intentionally left empty

ADC d,s

ADd with Carry

`d←d+s+CF`

8 bit	8 bit	8 bit	8 bit	16 bit
`ADC A,A`	`ADC A,E`	`ADC A,(HL)`	`ADC A,IXH`**	`ADC HL,BC`
`ADC A,B`	`ADC A,H`	`ADC A,(IX+d)`	`ADC A,IXL`**	`ADC HL,DE`
`ADC A,C`	`ADC A,L`	`ADC A,(IY+d)`	`ADC A,IYH`**	`ADC HL,HL`
`ADC A,D`	`ADC A,n`		`ADC A,IYL`**	`ADC HL,SP`

Adds source operand `s` or contents of the memory location addressed by `s` and value of carry flag to destination `d`. Result is then stored to destination `d`.

Effects

	SF	ZF		HF		PⓋ	NF	CF
8-bit	↕	↕		↕		↕	0	↕
16-bit	↕	↕		↕		↕	0	↕

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
HF	set if:	carry from bit 3
PV	set if:	• both operands positive and result negative • both operands negative and result positve
CF	set if:	carry from bit 7

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`A,r`	1	4	1,1μs	0,57μs	0,29μs	0,14μs
`A,n`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`A,(HL)`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`HL,rr`	4	15	4,3μs	2,14μs	1,07μs	0,54μs
`A,(IX+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs
`A,(IY+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs

ADD d,s

ADD

`d←d+s`

8-bit	8-bit	16-bit	16-bit	ZX Next
ADD A,A	ADD A,(HL)	ADD IX,BC	ADD HL,BC	ADD BC,A^{ZX}
ADD A,B	ADD A,(IX+d)	ADD IX,DE	ADD HL,DE	ADD DE,A^{ZX}
ADD A,C	ADD A,(IY+d)	ADD IX,IX	ADD HL,HL	ADD HL,A^{ZX}
ADD A,D	ADD A,IXH**	ADD IX,SP	ADD HL,SP	ADD BE,nn^{ZX}
ADD A,E	ADD A,IXL**	ADD IY,BC		ADD DE,nn^{ZX}
ADD A,H	ADD A,IYH**	ADD IY,DE		ADD HL,nn^{ZX}
ADD A,L	ADD A,IYL**	ADD IY,IY		
ADD A,n		ADD IY,SP		

Similar to `ADC` except carry flag is not used in calculation: adds operand `s` or contents of the memory location addressed by `s` to destination `d`. Result is then stored to destination `d`.

ZX Next Extended instructions for adding `A` to 16-bit register pair, zero extend `A` to 16-bits.

Effects	SF	ZF		HF		PⓋ	NF	CF
8-bit	↕	↕		↕		↕	0	↕
16-bit	-	-		↕		-	0	↕

SF 8-bit only, set if: result is negative (bit 7 is set)

ZF 8-bit only, set if: result is 0

HF set if: carry from bit 3 (bit 11 for 16-bit)

PV 8-bit only, set if:

- both operands positive and result negative
- both operands negative and result positve

CF set if: carry from bit 7 (bit 15 for 16-bit)

Effects	SF	ZF		HF		PV	NF	CF
ADD rr,A^{ZX}	-	-		-		-	-	?
ADD rr,nn^{ZX}	-	-		-		-	-	-

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
A,r	1	4	1,1µs	0,57µs	0,29µs	0,14µs
A,n	2	7	2,0µs	1,00µs	0,50µs	0,25µs
A,(HL)	2	7	2,0µs	1,00µs	0,50µs	0,25µs
rr,A^{ZX}	2	8	2,3µs	1,14µs	0,57µs	0,29µs
HL,rr	3	11	3,1µs	1,57µs	0,79µs	0,39µs
IX,rr	4	15	4,3µs	2,14µs	1,07µs	0,54µs
IY,rr	4	15	4,3µs	2,14µs	1,07µs	0,54µs
rr,nn^{ZX}	4	16	4,6µs	2,29µs	1,14µs	0,57µs
A,(IX+d)	5	19	5,4µs	2,71µs	1,36µs	0,68µs
A,(IY+d)	5	19	5,4µs	2,71µs	1,36µs	0,68µs

AND s

bitwise AND

A←A∧s

AND A
AND B
AND C
AND D
AND E
AND H
AND L
AND n
AND (HL)
AND (IX+d)
AND (IY+d)
AND IXH**
AND IXL**
AND IYH**
AND IYL**

Performs bitwise AND between accumulator A and the given operand. The result is then stored back to the accumulator. Individual bits are AND'ed as shown on the right:

A	s	Result
0	0	0
0	1	0
1	0	0
1	1	1

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		1		↕	0	0

SF set if: result is negative (bit 7 is set)

ZF set if: result is 0

PV set if: result has even number of bits set

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	4	1,1µs	0,57µs	0,29µs	0,14µs
n	2	7	2,0µs	1,00µs	0,50µs	0,25µs
(HL)	2	7	2,0µs	1,00µs	0,50µs	0,25µs
(IX+d)	5	19	5,4µs	2,71µs	1,36µs	0,68µs
(IY+d)	5	19	5,4µs	2,71µs	1,36µs	0,68µs

BIT b,s

test BIT

ZF← $\overline{s_b}$

BIT b,A
BIT b,B
BIT b,C
BIT b,D
BIT b,E
BIT b,H
BIT b,L
BIT b,(HL)
BIT b,(IX+d)
BIT b,(IY+d)

Tests specified bit b (0-7) of the given register s or contents of memory addressed by s and sets zero flag according to result; if bit was 1, **ZF** is 0 and vice versa.

Effects

SF	ZF		HF		PV	NF	CF
?	↕		1		?	0	-

ZF set if: bit b of the given source argument is 0

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
b,r	2	8	2,3µs	1,14µs	0,57µs	0,29µs
b,(HL)	3	12	3,4µs	1,71µs	0,86µs	0,43µs
b,(IX+d)	5	20	5,7µs	2,86µs	1,43µs	0,71µs
b,(IY+d)	5	20	5,7µs	2,86µs	1,43µs	0,71µs

`BRLC, BSLA, BSRA, BSRF, BSRL` See pages 155 and 156

`CALL nn`

CALL subroutine

(SP-1)←PC_h
(SP-2)←PC_l
SP←SP-2
PC←nn

Pushes program counter `PC` to stack and calls subroutine at the given location `nn` by changing `PC` to point to address `nn`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
–	–		–		–	–	–

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
5	17	4,9μs	2,43μs	1,21μs	0,61μs

`CALL c,nn`

CALL subroutine conditionally

`if c=true: CALL nn`

`CALL C,nn`	calls if **CF** is set	`CALL M,nn`	calls if **SF** is set
`CALL NC,nn`	calls if **CF** is reset	`CALL P,nn`	calls if **SF** is reset
`CALL Z,nn`	calls if **ZF** is set	`CALL PE,nn`	calls if **PV** is set
`CALL NZ,nn`	calls if **ZF** is reset	`CALL PO,nn`	calls if **PV** is reset

If the given condition is met, `CALL nn` is performed, as described above.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
–	–		–		–	–	–

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
c=false	3	10	2,9μs	1,43μs	0,71μs	0,36μs
c=true	5	17	4,9μs	2,43μs	1,21μs	0,61μs

BRLC DE,B^{ZX} Barrel Rotate Left Circular

DE←DE<<(B∧$0F or
DE←DE>>(16-B∧$0F)

Rotates value in register pair DE left for the amount given in bits 3-0 (low nibble) of register B. To rotate right, use formula: B=16-places. The result is stored in DE.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

BSLA DE,B^{ZX} Barrel Shift Left Arithmetic

DE←DE<<(B∧$1F)

Performs shift left of the value in register pair DE for the amount given in lower 5 bits of register B. The result is stored in DE.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

BSRA DE,B^{ZX} Barrel Shift Right Arithmetic

DE←signed(DE)>>(B∧$1F)

Performs arithmetical shift right of the value in register pair DE for the amount given in lower 5 bits of register B. The result is stored in DE.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

BSRF DE,B^{ZX} — **B**arrel **S**hift **R**ight **F**ill-one

`DE←~(unsigned(~DE)>>(B∧$1F))`

Performs fill-one-way shift right of the value in register pair `DE` for the amount given in lower 5 bits of register `B`. The result is stored in `DE`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

BSRL DE,B^{ZX} — **B**arrel **S**hift **R**ight **L**ogical

`DE←unsigned(DE)>>(B∧$1F)`

Performs logical shift right of the value in register pair `DE` for the amount given in lower 5 bits of register `B`. The result is stored in `DE`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

CALL

See page 154

CCF — **C**omplement **C**arry **F**lag

`CF←` $\overline{\texttt{CF}}$

Complements (inverts) carry flag **CF**; if **CF** was `0` it's now `1` and vice versa. Previous value of **CF** is copied to **HF**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		•		-	0	↕

HF Documentation says original value of **CF** is copied to **HF**, however under my tests **HF** remained unchanged

CF if **CF** was `0` it's now `1` and vice versa

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

CP s

ComPare

`A-s`

CP A
CP B
CP C
CP D

CP E
CP H
CP L
CP n

CP (HL)
CP (IX+d)
CP (IY+d)

CP IXH**
CP IXL**
CP IYH**
CP IYL**

Operand `s` or content of the memory location addressed by `s` is subtracted from accumulator `A`. Status flags are updated according to the result, but the result is then discarded (value of `A` is not changed).

Effects

SF	ZF		HF		PⓋ	NF	CF
↕	↕		↕		↕	1	↕

HF set if: borrow from bit 4

PV set if:
- `A` and `s` positive and `A-s` result negative
- `A` and `s` negative and `A-s` result positve

Other flags are set like this when `A` is greater than, equal or less than `s`:

	SF	ZF		HF		PⓋ	NF	CF
A>s	0	0		↕		↕	1	0
A=s	0	1		↕		↕	1	0
A<s	1	0		↕		↕	1	1

With this in mind, we can derive the programs for common comparisons:

`A=s`

```
    CP s
    JP Z, true ; A=s?
false: ; A!=s
true:  ; A=s
```

`A≠s`

```
    CP s
    JP NZ, true ; A!=s?
false: ; A=s
true:  ; A!=s
```

`A⩽s`

```
    CP s
    JP M, true ; A<s?
    JP Z, true ; A=s?
false: ; A>s
true:  ; A<=s
```

`A<s`

```
    CP s
    JP M, true ; A<s?
false: ; A>=s
true:  ; A<s
```

`A⩾s`

```
    CP s
    JP M, false ; A<s?
true:  ; A>=s
false: ; A<s
```

`A>s`

```
    CP s
    JP M, false ; A<s?
    JP Z, false ; A=s?
true:  ; A>s
false: ; A<=s
```

Note: the examples use two labels to emphasize both results. But only one is needed. Furthermore, depending on the actual needs, the programs can also use no label at all. For example, we can use `RET` instead of `JP` when used within a subroutine, and the desired outcome is to return if the condition is not met:

`A=s`

```
    CP s
    RET NZ ; A!=s?
    ; A=s
```

`A≠s`

```
    CP s
    RET Z ; A=s?
    ; A!=s
```

`A⩽s`

```
    CP s
    JP M, true ; A<s?
    JP Z, true ; A=s?
    RET        ; A>s
true:  ; A<=s
```

`A<s`

```
    CP s
    RET P ; A>=s?
    ; A<s
```

`A⩾s`

```
    CP s
    RET M ; A<s?
    ; A>=s
```

`A>s`

```
    CP s
    RET M ; A<s?
    RET Z ; A=s?
    ; A>s
```

Note: some of the comparisons are reversed. And `A⩽s` still requires a label because the condition is true if either **SF** or **ZF** is set.

Note: I opted to use **SF** for some of the comparisons. It makes more sense to me this way. But you can just as well use **CF** instead. As evident from the table on the previous page, both flags are updated the same way, so you could use `JP C` or `RET C` instead of `M` and `JP NZ` or `RET NZ` instead of `P`. This is due to `CP` performing a subtraction `A-s` internally. So when `s` is greater than `A`, the result of the subtraction is negative, meaning the sign flag is set. At the same time a borrow is needed, so the carry is set too. I thought it's worth mentioning since you may find examples using the carry flag elsewhere and wonder why.

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r`	1	4	1,1μs	0,57μs	0,29μs	0,14μs
`n`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`(HL)`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`(IX+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs
`(IY+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs

CPD

ComPare and Decrement

```
A-(HL)
HL←HL-1
BC←BC-1
```

Subtracts contents of memory location addressed by `HL` register pair from accumulator `A`. Result is then discarded. Afterwards both `HL` and `BC` are decremented.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		↕		•	1	-

SF set if: `A<(HL)` before `HL` is decremented
ZF set if: `A=(HL)` before `HL` is decremented
PV set if: `BC≠0` after execution

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

CPDR

ComPare and Decrement Repeated

```
do CPD
while A≠(HL)∧BC>0
```

Repeats `CPD` until either `A=(HL)` or `BC=0`. If `BC` is set to `0` before instruction execution, it loops through 64KB if no match is found. See `CPIR` for example.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		↕		•	1	-

SF set if: `A<(HL)` before `HL` is decremented
ZF set if: `A=(HL)` before `HL` is decremented
PV set if: `BC≠0` after execution

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0 or A=(HL)	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0 and A≠(HL)	5	21	6,0μs	3,00μs	1,50μs	0,75μs

CPI

ComPare and Increment

```
A-(HL)
HL←HL+1
BC←BC-1
```

Subtracts contents of memory location addressed by HL register pair from accumulator A. Result is then discarded. Afterwards HL is incremented and BC decremented.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		↕		•	1	-

SF	set if:	A<(HL) before HL is decremented
ZF	set if:	A=(HL) before HL is decremented
PV	set if:	BC≠0 after execution

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

CPIR

ComPare and Decrement Repeated

```
do CPI
while A≠(HL)∧BC>0
```

Repeats CPI until either A=(HL) or BC=0. If BC is set to 0 before instruction execution, it loops through 64KB if no match is found.

Example, searching for $AB in memory from $0000-$999:

CPIR = finding first occurrence:

```
LD HL, $0000
LD BC, $0999
LD A, $AB
CPIR
```

CPDR = finding last occurrence:

```
LD HL, $0999
LD BC, $0999
LD A, $AB
CPDR
```

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		↕		•	1	-

SF	set if:	A<(HL) before HL is decremented
ZF	set if:	A=(HL) before HL is decremented
PV	set if:	BC≠0 after execution

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0 or A=(HL)	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0 and A≠(HL)	5	21	6,0μs	3,00μs	1,50μs	0,75μs

`CPL`

See next page

`DAA`

Decimal Adjust Accumulator

Updates accumulator `A` for BCD correction after arithmetic operations using the following algorithm:

1. The least significant 4 bits of accumulator `A` (low nibble) are checked first. If they contain invalid BCD number (greater than 9), or **HF** is set, the value of `A` is adjusted based on the value of **NF**: if it's reset, `$06` is added to `A`, if set, `$06` is removed from `A`.
2. Then 4 most significant bits of accumulator `A` (high nibble) are checked in a similar fashion. If they contain invalid BCD number, or **CF** is set, the value of `A` is adjusted: if **NF** is not set, `$60` is added to `A`, if **NF** is set, `$60` is removed from `A`.
3. Finally flags are changed accordingly, as described below.

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		↕		↕	-	↕

SF set if: `A` is negative (bit 7 is set) after operation

ZF set if: `A` is 0 after operation

HF depends on: input values of **NF**, **HF** and bits 0-3 of `A`:

NF	HF	A[3-0]	→HF
0	*	0-9	0
0	*	A-F	1
1	0	*	0
1	1	0-5	1
1	1	6-F	0

PV set if: `A` has even number of bits set after operation

CF depends on: input values of **CF** and both nibbles of `A`:

CF	A[7-4]	A[3-0]	→CF
0	0-9	0-9	0
0	0-8	A-F	0
0	9-F	A-F	1
0	A-F	0-9	1
1	*	*	1

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

CPL

ComPLement accumulator

A← $\overline{\texttt{A}}$

Complements (inverts) all bits of the accumulator A and stores the result back to A.

Effects

SF	ZF		HF		PV	NF	CF
-	-		1		-	1	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

DEC s

DECrement

s←s-1

8-bit	8-bit	16-bit
DEC A	DEC (HL)	DEC BC
DEC B	DEC (IX+d)	DEC DE
DEC C	DEC (IY+d)	DEC HL
DEC D	DEC IXH**	DEC IX
DEC E	DEC IXL**	DEC IY
DEC H	DEC IYH**	DEC SP
DEC L	DEC IYL**	

Decrements the operand s or memory addressed by s by 1.

Effects	SF	ZF		HF		PⓋ	NF	CF
8-bit	↕	↕		↕		↕	1	-
16-bit (no effect)	-	-		-		-	-	-

SF	8-bit only, set if:	result is negative (bit 7 is set)
ZF	8-bit only, set if:	result is 0
HF	8-bit only, set if:	borrow from bit 4
PV	8-bit only, set if:	value was $80 before decrementing

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	4	1,1μs	0,57μs	0,29μs	0,14μs
rr	1	6	1,7μs	0,86μs	0,43μs	0,21μs
IX	2	10	2,9μs	1,43μs	0,71μs	0,36μs
IY	2	10	2,9μs	1,43μs	0,71μs	0,36μs
(HL)	3	11	3,1μs	1,57μs	0,79μs	0,39μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

DI

Disable Interrupts

```
IFF1←0
IFF2←0
```

Disables all maskable interrupts (mode 1 and 2). Interrupts are disabled after execution of the instruction following `DI`. See sections 2.4, page 19 and 3.12, page 119 for more details on interrupts.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
	1	4	1,1μs	0,57μs	0,29μs	0,14μs

DJNZ e

Decrement B and Jump if Not Zero

```
B←B-1
if B≠0: JR e
```

Decrements `B` register and jumps to the given relative address if `B≠0`. Given offset is added to the value of `PC` after parsing `DJNZ` instruction, so effective offset it `-126` to `+129`. Assembler automatically subtracts `2` from offset value `e` to generate opcode.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
B=0	2	8	2,3μs	1,14μs	0,57μs	0,29μs
B≠0	3	13	3,7μs	1,86μs	0,93μs	0,46μs

EI

Enable Interrupts

```
IFF1←1
IFF2←1
```

Enables maskable interrupts (mode 1 and 2). Interrupts are enabled after execution of the instruction following `EI`; typically `RETI` or `RETN`. See sections 2.4, page 19 and 3.12, page 119 for more details on interrupts.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
	1	4	1,1μs	0,57μs	0,29μs	0,14μs

EX d,s

EXchange register pair

`d↔s`

`EX AF,AF’`	`EX (SP),HL`
`EX DE,HL`	`EX (SP),IX`
	`EX (SP),IY`

Exchanges contents of two register pairs or register pair and last value pushed to stack. For example:

BEFORE			AFTER
Reg	Value		
HL	$ABCD		$3412
SP	$0B00		$0B00
Mem	Value	→ `EX (SP),HL` →	
$0B00	$12		$CD
$0B01	$34		$AB

Effects	SF	ZF		HF		PV	NF	CF
`EX AF,AF’`	•	•		•		•	•	•
Other variants no effect	-	-		-		-	-	-

- `EX AF,AF’` sets flags directly from the value of `F’`

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`rr,rr`	1	4	1,1μs	0,57μs	0,29μs	0,14μs
`(SP),HL`	5	19	5,4μs	2,71μs	1,36μs	0,68μs
`(SP),IX`	6	23	6,6μs	3,29μs	1,64μs	0,82μs
`(SP),IY`	6	23	6,6μs	3,29μs	1,64μs	0,82μs

EXX

EXchange alternate registers

`BC↔BC’`
`DE↔DE’`
`HL↔HL’`

Exchanges contents of registers `BC`, `DE` and `HL` with shadow registers `BC’`, `DE’` and `HL’`. The most frequent use is in interrupt handlers as an alternative to using the stack for saving and restoring register values. If using outside interrupt handlers, interrupts must be disabled before using this instruction.

Effects	SF	ZF		HF		PV	NF	CF
No effect on flags	-	-		-		-	-	-

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
	1	4	1,1μs	0,57μs	0,29μs	0,14μs

HALT

HALT

Suspends CPU and executes NOPs (to continue memory refresh cycles) until the next interrupt or reset. This effectively creates a delay. You can chain HALTs. But make sure that there will be an interrupt, otherwise HALT will run forever.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

IM n

Interrupt Mode

```
IM 0
IM 1
IM 2
```

Sets the interrupt mode. All 3 interrupts are maskable, meaning they can be disabled using DI instruction. See sections 2.4, page 19 and 3.12, page 119 for details and example.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

IN r,(s)

INput from port

r←(s)

IN A,(n)
IN A,(C)
IN B,(C)
IN C,(C)
IN D,(C)
IN E,(C)
IN H,(C)
IN L,(C)
IN (C)**
IN F,(C)**

Reads peripheral device addressed by BC or combination of A and immediate value and stores result in given register. The address is provided as follows:

	Address Bits	
Variant	15-8	7-0
IN A,(n)	A	n
IN r,(C)	B	C

So these two have the same result (though, as mentioned in section 3.11, page 117, variant on the right is slightly faster, 18 vs 22 T states):

```
LD BC, $DFFE
IN A, (C)
```

```
LD A, $DF
IN A, ($FE)
```

Effects

	SF	ZF		HF		PV	NF	CF
IN r,(C)	↕	↕		0		↕	0	-
IN A,(n) no effect	-	-		-		-	-	-

SF	IN r,(C), set if:	input data is negative (bit 7 is set)
ZF	IN r,(C), set if:	input data is 0
PV	IN r,(C), set if:	input data has even number of bits set

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r,(n)	3	11	3,1μs	1,57μs	0,79μs	0,39μs
r,(C)	3	12	3,4μs	1,71μs	0,86μs	0,43μs

Note: IN (C) (or its alternative form IN F,(C)) performs an input, but does not store the result, only sets the flags.

Note: some assemblers also allow (BC) to be used instead of (C).

INC s

INCrement

s←s+1

8-bit	8-bit	16-bit
`INC A`	`INC (HL)`	`INC BC`
`INC B`	`INC (IX+d)`	`INC DE`
`INC C`	`INC (IY+d)`	`INC HL`
`INC D`	`INC IXH`**	`INC IX`
`INC E`	`INC IXL`**	`INC IY`
`INC H`	`INC IYH`**	`INC SP`
`INC L`	`INC IYL`**	

Increments the operand `s` or memory addressed by `s` by 1.

Effects	SF	ZF		HF		PⓋ	NF	CF
8-bit	↕	↕		↕		↕	0	-
16-bit (no effect)	-	-		-		-	-	-

SF	8-bit only, set if:	result is negative (bit 7 is set)
ZF	8-bit only, set if:	result is 0
HF	8-bit only, set if:	carry from bit 3
PV	8-bit only, set if:	value was `$7F` before incrementing

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	6	1,7μs	0,86μs	0,43μs	0,21μs
rr	1	6	1,7μs	0,86μs	0,43μs	0,21μs
IX	2	10	2,9μs	1,43μs	0,71μs	0,36μs
IY	2	10	2,9μs	1,43μs	0,71μs	0,36μs
(HL)	3	11	3,1μs	1,57μs	0,79μs	0,39μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

This page intentionally left empty

IND

INput and Decrement

```
(HL)←(BC)
HL←HL-1
B←B-1
```

Reads peripheral device addressed by `BC` and stores the result in memory addressed by `HL` register pair. Then decrements `HL` and `B`.

Effects

SF	ZF		HF		PV	NF	CF
●	●		●		●	1	-

SF		destroyed on Next, Z80 see 2.3.3, page 17
ZF	set if:	`B` becomes zero after decrementing
HF		destroyed on Next, Z80 see 2.3.3, page 17
PV		destroyed on Next, Z80 see 2.3.3, page 17

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
	4	16	4,6μs	2,29μs	1,14μs	0,57μs

INDR

INput and Decrement Repeated

```
do IND
while B>0
```

Repeats `IND` until `B=0`.

Effects

SF	ZF		HF		PV	NF	CF
●	1		●		●	1	-

SF	destroyed on Next, Z80 see 2.3.3, page 17
HF	destroyed on Next, Z80 see 2.3.3, page 17
PV	destroyed on Next, Z80 see 2.3.3, page 17

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
B=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
B≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

INI

INput and Increment

```
(HL)←(BC)
HL←HL+1
B←B-1
```

Reads peripheral device addressed by BC and stores the result in memory addressed by HL register pair. Then increments HL and decrements B.

Effects

SF	ZF		HF		PV	NF	CF
•	•		•		•	1	-

SF		destroyed on Next, Z80 see 2.3.3, page 17
ZF	set if:	B becomes zero after decrementing
HF		destroyed on Next, Z80 see 2.3.3, page 17
PV		destroyed on Next, Z80 see 2.3.3, page 17

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

INIR

INput and Increment Repeated

```
do INI
while B>0
```

Repeats INI until B=0.

Effects

SF	ZF		HF		PV	NF	CF
•	1		•		•	1	-

SF	destroyed on Next, Z80 see 2.3.3, page 17
HF	destroyed on Next, Z80 see 2.3.3, page 17
PV	destroyed on Next, Z80 see 2.3.3, page 17

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
B=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
B≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

JP nn

JumP

`PC←nn`

```
JP nn                 JP (IX)
JP (HL)               JP (IY)
```

Unconditionally jumps (changes program counter `PC` to point) to the given absolute address or the memory location addressed by register pair. Unconditional jumps are the fastest way of changing program counter, even faster than `JR`, but they take more bytes.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`(HL)`	1	4	1,1μs	0,57μs	0,29μs	0,14μs
`(IX)`	2	8	2,3μs	1,14μs	0,57μs	0,29μs
`(IY)`	2	8	2,3μs	1,14μs	0,57μs	0,29μs
`nn`	3	10	2,9μs	1,43μs	0,71μs	0,36μs

JP c,nn

JumP conditionally

`if c=true: JP nn`

`JP C,nn`	jumps if **CF** is set	`JP M,nn`	jumps if **SF** is set
`JP NC,nn`	jumps if **CF** is reset	`JP P,nn`	jumps if **SF** is reset
`JP Z,nn`	jumps if **ZF** is set	`JP PE,nn`	jumps if **PV** is set
`JP NZ,nn`	jumps if **ZF** is reset	`JP PO,nn`	jumps if **PV** is reset

Conditionally jumps to the given absolute address. See `CP` on page 157 for more details on comparisons.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
3	10	2,9μs	1,43μs	0,71μs	0,36μs

JP (C)^{ZX} **JumP**

`PC←PC∧$C000+IN(C)<<6`

Sets bottom 14 bits of current program counter `PC`* to value read from I/O port: `PC[13-0] = (IN (C) << 6)`. Can be used to execute code block read from a disk stream.

*"Current `PC`" is the address of the next instruction after `JP (C)`; `PC` was already advanced after fetching `JP (C)` instruction from memory. If `JP (C)` instruction is located at the very end of 16K memory block (`$..FE` or `$..FF` address), then the new `PC` value will land into the following 16K block.

Effects

SF	ZF		HF		PV	NF	CF
?	?		?		?	?	?

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
3	13	3,7μs	1,86μs	0,93μs	0,46μs

JR e **Jump Relative**

`PC←PC+e`

Unconditionally performs relative jump. Offset `e` is added to the value of program counter `PC` as signed value to allow jumps forward and backward. Offset is added to `PC` after `JR` instruction is read (aka `PC+2`), so offset is in the range of `-126` to `129`. Assembler automatically subtracts 2 from offset value `e` to generate opcode.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
3	12	3,4μs	1,71μs	0,86μs	0,43μs

JR c,n **Jump Relative conditionally**

`if c=true: JR n`

`JR C,e`	jumps if **CF** is set	`JR Z,e`	jumps if **ZF** is set
`JR NC,e`	jumps if **CF** is reset	`JR NZ,e`	jumps if **ZF** is reset

Conditionally performs relative jump. Note: in contrast to `JP`, `JR` only supports above 4 conditions. See `CP` on page 157 for more details on conditions.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
c=false	2	7	2,0μs	1,00μs	0,50μs	0,25μs
c=true	3	12	3,4μs	1,71μs	0,86μs	0,43μs

LD d,s

LoaD

d←s

Loads source s into destination d. The following combinations are allowed (source s is represented horizontally, destination d vertically):

	A	B	C	D	E	H	L	I	R	IXH	IXL	IYH	IYL	BC	DE	HL	SP	IX	IY	(BC)	(DE)	(HL)	(IX+d)	(IY+d)	n	nn	(nn)
A	●	●	●	●	●	●	●	●	●	●	●	●	●							●	●	●	●	●	●		●
B	●	●	●	●	●	●	●			●	●	●	●									●	●	●	●		
C	●	●	●	●	●	●	●			●	●	●	●									●	●	●	●		
D	●	●	●	●	●	●	●			●	●	●	●									●	●	●	●		
E	●	●	●	●	●	●	●			●	●	●	●									●	●	●	●		
H	●	●	●	●	●	●	●															●	●	●	●		
L	●	●	●	●	●	●	●															●	●	●	●		
I	●																										
R	●																										
IXH	●	●	●	●	●					●	●														●		
IXL	●	●	●	●	●					●	●														●		
IYH	●	●	●	●	●							●	●												●		
IYL	●	●	●	●	●							●	●												●		
BC																										●	●
DE																										●	●
HL																										●	●
SP																●		●	●							●	●
IX																										●	●
IY																										●	●
(BC)	●																										
(DE)	●																										
(HL)	●	●	●	●	●	●	●																		●		
(IX+d)	●	●	●	●	●	●	●																		●		
(IY+d)	●	●	●	●	●	●	●																		●		
(nn)	●													●	●	●	●	●	●								

Effects	SF	ZF		HF		PV	NF	CF
LD A,I and LD A,R	↕	↕		0		IFF2	0	-
Other variants	-	-		-		-	-	-

Timing 8-bit	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r,r	1	4	1,1μs	0,57μs	0,29μs	0,14μs
r,n	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(rr),A	2	7	2,0μs	1,00μs	0,50μs	0,25μs
A,(rr)	2	7	2,0μs	1,00μs	0,50μs	0,25μs
r,(HL)	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(HL),r	2	7	2,0μs	1,00μs	0,50μs	0,25μs
A,I	2	9	2,6μs	1,29μs	0,64μs	0,32μs
A,R	2	9	2,6μs	1,29μs	0,64μs	0,32μs
I,A	2	9	2,6μs	1,29μs	0,64μs	0,32μs
R,A	2	9	2,6μs	1,29μs	0,64μs	0,32μs
(HL),n	3	10	2,9μs	1,43μs	0,71μs	0,36μs
A,(nn)	4	13	3,7μs	1,86μs	0,93μs	0,46μs
(nn),A	4	13	3,7μs	1,86μs	0,93μs	0,46μs
r,(IX+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs
r,(IY+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IX+d),r	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IX+d),n	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IY+d),r	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IY+d),n	5	19	5,4μs	2,71μs	1,36μs	0,68μs

Timing 16-bit	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
SP,HL	1	6	1,7μs	0,86μs	0,43μs	0,21μs
SP,IX	2	10	2,9μs	1,43μs	0,71μs	0,36μs
SP,IY	2	10	2,9μs	1,43μs	0,71μs	0,36μs
rr,nn	3	10	2,9μs	1,43μs	0,71μs	0,36μs
IX,nn	4	14	4,0μs	2,00μs	1,00μs	0,50μs
IY,nn	4	14	4,0μs	2,00μs	1,00μs	0,50μs
(HL),nn	5	16	4,6μs	2,29μs	1,14μs	0,57μs
(nn),HL	5	16	4,6μs	2,29μs	1,14μs	0,57μs
(IX),nn	6	20	5,7μs	2,86μs	1,43μs	0,71μs
(IY),nn	6	20	5,7μs	2,86μs	1,43μs	0,71μs
rr,(nn)	6	20	5,7μs	2,86μs	1,43μs	0,71μs
(nn),rr	6	20	5,7μs	2,86μs	1,43μs	0,71μs

LDD

LoaD and Decrement

```
(DE)←(HL)
DE←DE-1
HL←HL-1
BC←BC-1
```

Loads contents of memory location addressed by HL to memory location addressed by DE. Then decrements DE, HL and BC register pairs.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		•	0	-

PV set if: BC≠0 after execution

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

LDDX^{ZX}

LoaD and Decrement eXtended

```
if (HL)≠A: (DE)←(HL)
DE←DE+1
HL←HL-1
BC←BC-1
```

Works similar to LDD except:

- Byte is only copied if it's different from the accumulator A
- DE is incremented instead of decremented
- Doesn't change flags

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

LDI

LoaD and Increment

```
(DE)←(HL)
DE←DE+1
HL←HL+1
BC←BC-1
```

Same as LDD, except it increments DE and HL.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		•	0	-

PV set if: BC≠0 after execution

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

LDIX^{ZX}

LoaD and Increment eXtended

```
if (HL)≠A: (DE)←(HL)
DE←DE+1
HL←HL+1
BC←BC-1
```

Works similar to LDI except:

- Byte is only copied if it's different from the accumulator A
- Doesn't change flags

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

LDWS^{ZX}

LoaD Wasp Special

```
(DE)←(HL)
INC L
INC D
```

Copies the byte pointed to by HL to the address pointed to by DE. Then increments L and D. Used for vertically copying bytes to Layer 2 display. Flags are identical to what the INC D instrution would produce.

Effects

SF	ZF		HF		PⓋ	NF	CF
↕	↕		↕		↕	0	-

SF	set if:	D is negative (bit 7 is set)
ZF	set if:	result is 0
HF	set if:	carry from bit 3
PV	set if:	D was $7F before incrementing

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	14	4,0μs	2,00μs	1,00μs	0,50μs

Note: the source data are read from a single 256B (aligned) block of memory, because only L is incremented, not the whole HL pair.

LDDR

LoaD and Decrement Repeated

```
do LDD:
   (DE)←(HL)
   DE←DE-1: HL←HL-1: BC←BC-1
while BC>0
```

Repeats LDD until BC=0. LDDR can be used for block transfer. See LDIR on page 177 for an example and comparison of both instructions.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		0	0	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

LDDRX^{ZX}

LoaD and Decrement Repeated eXtended

```
do LDDX:
   if (HL)≠A: (DE)←(HL)
   DE←DE+1: HL←HL-1: BC←BC-1
while BC>0
```

Works similar to LDDR except the differences noted at LDDX above.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

LDIR

LoaD and Increment Repeated

```
do LDI:
   (DE)←(HL)
   DE←DE+1: HL←HL+1: BC←BC-1
while BC>0
```

Repeats LDI until BC=0. Example of copying 100 bytes from source to destination with LDIR and LDDR:

LDIR = copy forward

```
LD HL, source
LD DE, destination
LD BC, 100
LDIR
```

LDDR = copy backwards

```
LD HL, source+99
LD DE, destination+99
LD BC, 100
LDDR
```

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		0	0	-

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

LDIRX^{ZX} **LoaD and Increment Repeated eXtended**

```
do LDIX:
   if (HL)≠A: (DE)←(HL)
   DE←DE+1: HL←HL+1: BC←BC-1
while BC>0
```

Works similar to `LDIR` except the differences noted at `LDIX` on previous page.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

LDPIRX^{ZX} **LoaD Pattern fill and Increment Repeated eXtended**

```
do
  t←(HL∧$FFF8+E∧7)
  if t≠A: (DE)←t
  DE←DE+1: BC←BC-1
while BC>0
```

Similar to `LDIRX` except the source byte address is not just `HL`, but is obtained by using the top 13 bits of `HL` and lower 3 bits of `DE`. Furthermore `HL` is not incremented during the loop; it serves as the base address of the aligned 8-byte lookup table. `DE` works as destination and also wrapping index 0..7 into the table. This instruction is intended for "pattern fill" functionality.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
BC=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
BC≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

MUL D,E^{ZX}

MULtiply

`DE←D×E`

Multiplies `D` by `E`, storing 16-bit result into `DE`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

NEG

NEGate

`A←-A`

Negates contents of the accumulator `A` and stores result back to `A`. You can also think of the operation as subtracting the value of `A` from `0` (`A←0-A`). This way it might be easier to understand effects on flags.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		↕		↕	1	↕

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is `0`
HF	set if:	borrow from bit 4
PV	set if:	`A` was `$80` before operation
CF	set if:	`A` was not `$00` before operation

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

NEXTREG n,s^{ZX}

set NEXT REGister value

`HwNextReg[n]←s`

`NEXTREG n,A`   `NEXTREG n,n'`

Directly sets the Next Feature Control Registers without going through ports **TBBlue Register Select** `$243B` and **TBBlue Register Access** `$253B` (page 34). See section 3.1.2, page 29 for registers list.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r,A`	4	17	4,9μs	2,43μs	1,21μs	0,61μs
`r,n`	5	20	5,7μs	2,86μs	1,43μs	0,71μs

NOP

No OPeration

Does nothing for 4 cycles.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

OR s

bitwise OR

A←A∨s

OR A
OR B
OR C
OR D

OR E
OR H
OR L
OR n

OR (HL)
OR (IX+d)
OR (IY+d)

OR IXH**
OR IXL**
OR IYH**
OR IYL**

Performs bitwise or between the accumulator A and operand s or contents of memory addressed by s. Then stores the result back to A. Individual bits are OR'ed as shown on the right:

A	s	Result
0	0	0
0	1	1
1	0	1
1	1	1

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	0

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	4	1,1μs	0,57μs	0,29μs	0,14μs
n	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(HL)	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(IX+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IY+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs

OTDR

OuTput and DecRement

```
do OUTD
while B>0
```

Repeats `OUTD` (see page 182) until `B=0`. Similar to `OTIR` except `HL` is decremented instead of incremented.

Effects

SF	ZF		HF		PV	NF	CF
●	1		●		●	1	-

SF destroyed on Next, Z80 see 2.3.3, page 17

HF destroyed on Next, Z80 see 2.3.3, page 17

PV destroyed on Next, Z80 see 2.3.3, page 17

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
B=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
B≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

OTIR

OuTput and IncRement

```
do OUTI
while B>0
```

Repeats `OUTI` (see page 182) until `B=0`. Similar to `OTDR` except `HL` is incremented instead of decremented.

Effects

SF	ZF		HF		PV	NF	CF
●	1		●		●	1	-

SF destroyed on Next, Z80 see 2.3.3, page 17

HF destroyed on Next, Z80 see 2.3.3, page 17

PV destroyed on Next, Z80 see 2.3.3, page 17

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
B=0	4	16	4,6μs	2,29μs	1,14μs	0,57μs
B≠0	5	21	6,0μs	3,00μs	1,50μs	0,75μs

OUT

See page 183

OUTD

OUTput and Decrement

```
B←B-1
(BC)←(HL)
HL←HL-1
```

Outputs the value from contents of memory addressed by `HL` to port on address `BC`. Then decrements both, `HL` and `B`.

Effects

SF	ZF		HF		PV	NF	CF
•	•		•		•	1	-

SF		destroyed on Next, Z80 see 2.3.3, page 17
ZF	set if:	`B=0` after decrement
HF		destroyed on Next, Z80 see 2.3.3, page 17
PV		destroyed on Next, Z80 see 2.3.3, page 17

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

OUTI

OUTput and Increment

```
B←B-1
(BC)←(HL)
HL←HL+1
```

Similar to `OUTD` except `HL` is incremented.

Effects

SF	ZF		HF		PV	NF	CF
•	•		•		•	1	-

SF		destroyed on Next, Z80 see 2.3.3, page 17
ZF	set if:	`B=0` after decrement
HF		destroyed on Next, Z80 see 2.3.3, page 17
PV		destroyed on Next, Z80 see 2.3.3, page 17

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

`OUT (d),s` **OUTput to port**

`(d)←s`

`OUT (n),A`	`OUT (C),A`	`OUT (C),E`
	`OUT (C),B`	`OUT (C),H`
	`OUT (C),C`	`OUT (C),L`
	`OUT (C),D`	`OUT (C),0`**

Writes the value of operand `s` to the port at address `d`. Port addresses are always 16-bit values defined like this:

	Address Bits	
Variant	15-8	7-0
`OUT (n),A`	A	n
`OUT (C),r`	B	C

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`(n),A`	3	11	3,1μs	1,57μs	0,79μs	0,39μs
`(C),r`	3	12	3,4μs	1,71μs	0,86μs	0,43μs

Note: on the Next FPGA `OUT (C),0` variant outputs `0` to the port at address `BC`, but some Z80 chips may output different value like `$FF`, so it is not recommended to use `OUT (C),0` if you want to reuse your code on original ZX Spectrum also.

`OUTINB`^{ZX} **OUTput and Increment with No B**

`(BC)←(HL)`
`HL←HL+1`

Similar to `OUTI` except it doesn't decrement `B`.

Effects

SF	ZF		HF		PV	NF	CF
?	?		?		?	?	?

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	16	4,6μs	2,29μs	1,14μs	0,57μs

PIXELAD^{ZX}

PIXEL ADdress

```
HL←$4000+((D∧$C0)<<5)+((D∧$07)<<8)+((D∧$38)<<2)+(E>>3)
```

Takes `E` and `D` as the (x,y) coordinates of a point and calculates the address of the byte containing this pixel in the pixel area of standard ULA screen 0. Result is stored in `HL`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

PIXELDN^{ZX}

PIXEL DowN

```
if (HL∧$700)≠$700
  HL←HL+256
else if (HL∧$E0)≠$E0
  HL←HL∧$F8FF+$20
else
  HL←HL∧$F81F+$800
```

Updates the address in `HL` (likely from prior `PIXELAD` or `PIXELDN`) to move down by one line of pixels of standard ULA screen 0.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

POP rr

POP from stack

$\texttt{rr}_{\texttt{h}}$←(SP+1)
$\texttt{rr}_{\texttt{l}}$←(SP)
SP←SP+2

```
POP AF                  POP IX
POP BC                  POP IY
POP DE
POP HL
```

Copies 2 bytes from stack pointer `SP` into contents of the given register pair `ss` and increments `SP` by 2.

Effects

	SF	ZF		HF		PV	NF	CF
POP AF	↕	↕		↕		↕	↕	↕
Other variants no effect	-	-		-		-	-	-

- `POP AF` flags set directly to low 8-bits of the value from `SP`

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
rr	3	10	2,9μs	1,43μs	0,71μs	0,36μs
IX	4	14	4,0μs	2,00μs	1,00μs	0,50μs
IY	4	14	4,0μs	2,00μs	1,00μs	0,50μs

PUSH ss

PUSH on stack

(SP-2)←$\texttt{ss}_{\texttt{l}}$
(SP-1)←$\texttt{ss}_{\texttt{h}}$
SP←SP-2

PUSH AF — PUSH IX — PUSH nn^{ZX}
PUSH BC — PUSH IY
PUSH DE
PUSH HL

Copies contents of a register pair to the top of the stack pointer `SP`, then decrements `SP` by 2. Next extended `PUSH nn` also allows pushing immediate 16-bit value.

Effects

	SF	ZF		HF		PV	NF	CF
No effect on flags	-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
rr	3	11	3,1μs	1,57μs	0,79μs	0,39μs
IX	4	15	4,3μs	2,14μs	1,07μs	0,54μs
IY	4	15	4,3μs	2,14μs	1,07μs	0,54μs
nn	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RES b,s

<u>RES</u>et bit

$s_b \leftarrow 0$

```
RES b,A          RES b,(IX+d),A**     RES b,(IY+d),A**
RES b,B          RES b,(IX+d),B**     RES b,(IY+d),B**
RES b,C          RES b,(IX+d),C**     RES b,(IY+d),C**
RES b,D          RES b,(IX+d),D**     RES b,(IY+d),D**
RES b,E          RES b,(IX+d),E**     RES b,(IY+d),E**
RES b,H          RES b,(IX+d),H**     RES b,(IY+d),H**
RES b,L          RES b,(IX+d),L**     RES b,(IY+d),L**
RES b,(HL)
RES b,(IX+d)
RES b,(IY+d)
```

Resets bit `b` (`0-7`) of the given register `s` or memory location addressed by operand `s`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r`	2	8	2,3μs	1,14μs	0,57μs	0,29μs
`(HL)`	4	15	4,3μs	2,14μs	1,07μs	0,54μs
`(IX+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs
`(IY+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RET

RETurn from subroutine

$PC_l \leftarrow (SP)$
$PC_h \leftarrow (SP+1)$
$SP \leftarrow SP+2$

Returns from subroutine. The contents of program counter `PC` is `POP`-ed from stack so next instruction will be loaded from there.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
	3	10	2,9μs	1,43μs	0,71μs	0,36μs

RET c

RETurn from subroutine conditionally

`if c=true: RET`

`RET C,nn`	returns if **CF** is set	`RET M,nn`	returns if **SF** is set
`RET NC,nn`	returns if **CF** is reset	`RET P,nn`	returns if **SF** is reset
`RET Z,nn`	returns if **ZF** is set	`RET PE,nn`	returns if **PV** is set
`RET NZ,nn`	returns if **ZF** is reset	`RET PO,nn`	returns if **PV** is reset

If given condition is met, `RET` is performed, as described above.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
c=false	1	5	1,4μs	0,71μs	0,36μs	0,18μs
c=true	3	11	3,1μs	1,57μs	0,79μs	0,39μs

RETI

RETurn from Interrupt

```
PCl←(SP)
PCh←(SP+1)
SP←SP+2
```

Returns from maskable interrupt; restores stack pointer `SP` and signals to I/O device that interrupt routine is completed.

Note that `RETI` doesn't re-enable interrupts that were disabled when interrupt routine started - `EI` should be called before `RETI` to do that.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	14	4,0μs	2,00μs	1,00μs	0,50μs

RETN

RETurn from Non-maskable interrupt

```
PCl←(SP)
PCh←(SP+1)
SP←SP+2
IFF1←IFF2
```

Returns from non-maskable interrupt; restores stack pointer `SP` and copies state of `IFF2` back to `IFF1` so that maskable interrupts are re-enabled.

Effects

No effect on flags

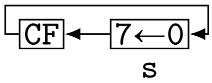
SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
4	14	4,0μs	2,00μs	1,00μs	0,50μs

RL s

Rotate Left



```
RL A              RL (IX+d),A**         RL (IY+d),A**
RL B              RL (IX+d),B**         RL (IY+d),B**
RL C              RL (IX+d),C**         RL (IY+d),C**
RL D              RL (IX+d),D**         RL (IY+d),D**
RL E              RL (IX+d),E**         RL (IY+d),E**
RL H              RL (IX+d),H**         RL (IY+d),H**
RL L              RL (IX+d),L**         RL (IY+d),L**
RL (HL)
RL (IX+d)
RL (IY+d)
```

Performs 9-bit left rotation of the value of the operand `s` or memory addressed by `s` through the carry flag **CF** so that contents of **CF** are moved to bit `0` and bit `7` to **CF**. Result is then stored back to `s`.

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		0		↕	0	↕

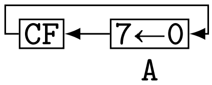
SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set
CF	set to:	bit 7 of the original value

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r`	2	8	2,3μs	1,14μs	0,57μs	0,29μs
`(HL)`	4	15	4,3μs	2,14μs	1,07μs	0,54μs
`(IX+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs
`(IY+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RLA

Rotate Left Accumulator



Performs `RL A`, but twice faster and preserves **SF**, **ZF** and **PV**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		-	0	↕

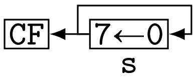
CF	set to:	bit 7 of the original value

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

RLC s

Rotate Left Circular



```
RLC A
RLC B
RLC C
RLC D
RLC E
RLC H
RLC L
RLC (HL)
RLC (IX+d)
RLC (IY+d)
RLC (IX+d),A**
RLC (IX+d),B**
RLC (IX+d),C**
RLC (IX+d),D**
RLC (IX+d),E**
RLC (IX+d),H**
RLC (IX+d),L**
RLC (IY+d),A**
RLC (IY+d),B**
RLC (IY+d),C**
RLC (IY+d),D**
RLC (IY+d),E**
RLC (IY+d),H**
RLC (IY+d),L**
```

Performs 8-bit rotation to the left. Bit 7 is moved to carry flag **CF** as well as to bit 0. Result is then stored back to s.

Note: undocumented variants work slightly differently:

RLC r,(IX+d):

```
r←(IX+d)
RLC r
(IX+d)←r
```

RLC r,(IY+d):

```
r←(IY+d)
RLC r
(IY+d)←r
```

Effects

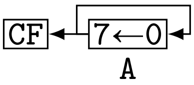
SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	↕

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set
CF	set to:	bit 7 of the original value

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RLCA

Rotate Left Circular Accumulator



Performs `RLC A`, but twice faster and preserves **SF**, **ZF** and **PV**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		-	0	↕

CF set to: bit 7 of the original value

Timing

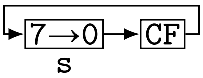
Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

RLD

See page 193

RR s

Rotate Right



```
RR A              RR (IX+d),A**         RR (IY+d),A**
RR B              RR (IX+d),B**         RR (IY+d),B**
RR C              RR (IX+d),C**         RR (IY+d),C**
RR D              RR (IX+d),D**         RR (IY+d),D**
RR E              RR (IX+d),E**         RR (IY+d),E**
RR H              RR (IX+d),H**         RR (IY+d),H**
RR L              RR (IX+d),L**         RR (IY+d),L**
RR (HL)
RR (IX+d)
RR (IY+d)
```

Performs 9-bit right rotation of the contents of the operand `s` or memory addressed by `s` through carry flag **CF** so that contents of **CF** are moved to bit 7 and bit 0 to **CF**. Result is then stored back to `s`.

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		0		↕	0	↕

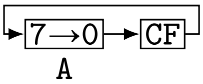
SF set if: result is negative (bit 7 is set)
ZF set if: result is 0
PV set if: result has even number of bits set
CF set to: bit 0 of the original value

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RRA

Rotate Right Accumulator



Performs `RR A`, but twice faster and preserves **SF**, **ZF** and **PV**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		-	0	↕

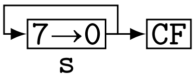
CF set to: bit 0 of the original value

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

RRC s

Rotate Right Circular



RRC A	RRC (IX+d),A**	RRC (IY+d),A**
RRC B	RRC (IX+d),B**	RRC (IY+d),B**
RRC C	RRC (IX+d),C**	RRC (IY+d),C**
RRC D	RRC (IX+d),D**	RRC (IY+d),D**
RRC E	RRC (IX+d),E**	RRC (IY+d),E**
RRC H	RRC (IX+d),H**	RRC (IY+d),H**
RRC L	RRC (IX+d),L**	RRC (IY+d),L**
RRC (HL)		
RRC (IX+d)		
RRC (IY+d)		

Performs 8-bit rotation of the source `s` to the right. Bit 0 is moved to **CF** as well as to bit 7. Result is then stored back to `s`.

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		0		↕	0	↕

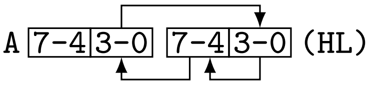
SF set if: result is negative (bit 7 is set)
ZF set if: result is 0
PV set if: result has even number of bits set
CF set to: bit 0 of the original value

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

RLD

Rotate Left bcd Digit



Performs leftward 12-bit rotation of 4-bit nibbles where 2 least significant nibbles are stored in memory location addressed by `HL` and most significant digit as lower 4 bits of the accumulator `A`.

If used with BCD numbers: as the shift happens by 1 digit to the left, this effectively results in multiplication with `10`. `A` acts as a sort of decimal carry in the operation. Example of multiplying multi-digit BCD number by 10:

```
MultiplyBy10:       ; number=0123
    LD HL, number+digits-1
    LD B, digits    ; number of repeats
    XOR A           ; reset "carry"
lp: RLD             ; multiply by 10
    DEC HL          ; prev 2 digits
    DJNZ lp         ; number=1230, A=0

number:
    DB $01, $23
digits = $-number ;(2)
```

Progression

line	number	A	B
2-4 ↓	$01\underbrace{23}_{\text{(HL)}}$	0	2
5-7 ↻	$0\underbrace{13}_{\text{(HL)}}0$	2	1
5-7 ↻	$\underbrace{12}_{\text{(HL)}}30$	0	0

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	-

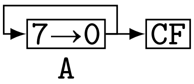
Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
5	18	5,1µs	2,57µs	1,29µs	0,64µs

Note: instruction doesn't assume any format of the data; it simply rotates nibbles. So while it's most frequently associated with BCD numbers, it can be used for shifting hexadecimal values or any other content.

RRCA

Rotate Right Circular Accumulator



Performs `RRC A`, but twice faster and preserves **SF**, **ZF** and **PV**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		-	0	↕

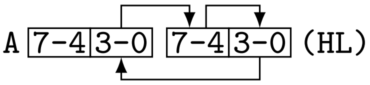
CF set to: bit `0` of the original value

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1µs	0,57µs	0,29µs	0,14µs

RRD

Rotate Right bcd Digit



Similar to RLD except rotation is to the right. If used with BCD values, this operation effectively divides 3-digit BCD number by 10 and stores remainder in A. Taking the example from RLD, we can easily convert it to division by 10 simply by using RRD. Note however we also need to change the order - we start from MSB now (which is exactly how division would be performed by hand):

```
1  DivideBy10:
2      LD HL, number   ; number=0123
3      LD B, digits    ; number of repeats
4      XOR A           ; reset "carry"
5  lp: RRD             ; divide by 10
6      INC HL          ; next 2 digits
7      DJNZ lp         ; number=0012, A=3
8
9  number:
10     DB $01, $23
11 digits = $-number  ;(2)
```

Progression

line	number	A	B
2-4 ↓	$\underbrace{01}_{\text{(HL)}}23$	0	2
5-7 ↻	$00\underbrace{23}_{\text{(HL)}}$	1	1
5-7 ↻	$0012\underbrace{\quad}_{\text{(HL)}}$	3	0

Effects

SF	ZF		HF		ⓅV	NF	CF
↕	↕		0		↕	0	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
5	18	5,1μs	2,57μs	1,29μs	0,64μs

Note: similar to RLD, this instruction also doesn't assume any format of the data; it simply rotates nibbles. So while it's most frequently associated with BCD numbers, it can be used for shifting hexadecimal values or any other content.

RST n

ReSTart

(SP-1)←PC_h
(SP-2)←PC_l
SP←SP-2
PC←n

```
RST $00          RST $20
RST $08          RST $28
RST $10          RST $30
RST $18          RST $38
```

Restarts at the zero page address `s`. Only above addresses are possible, all in page `0` of the memory, therefore the most significant byte of the program counter `PC` is loaded with `$00`. The instruction may be used as a fast response to an interrupt.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
3	11	3,1μs	1,57μs	0,79μs	0,39μs

SBC

See page 201

SCF

Set Carry Flag

CF←1

Sets carry flag **CF**.

Effects

SF	ZF		HF		PV	NF	CF
-	-		0		-	0	1

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
1	4	1,1μs	0,57μs	0,29μs	0,14μs

SET b,s

SET bit

$\mathtt{s_b} \leftarrow 1$

```
SET b,A          SET b,(IX+d),A**     SET b,(IY+d),A**
SET b,B          SET b,(IX+d),B**     SET b,(IY+d),B**
SET b,C          SET b,(IX+d),C**     SET b,(IY+d),C**
SET b,D          SET b,(IX+d),D**     SET b,(IY+d),D**
SET b,E          SET b,(IX+d),E**     SET b,(IY+d),E**
SET b,H          SET b,(IX+d),H**     SET b,(IY+d),H**
SET b,L          SET b,(IX+d),L**     SET b,(IY+d),L**
SET b,(HL)
SET b,(IX+d)
SET b,(IY+d)
```

Sets bit `b` (`0-7`) of operand `s` or memory location addressed by `s`.

Note: undocumented variants work slightly differently:

<u>`SET b,(IX+d),r`</u>:

```
r←(IX+d)
rb ←1
(IX+d)←r
```

<u>`SET b,(IY+d),r`</u>:

```
r←(IY+d)
rb ←1
(IY+d)←r
```

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

SETAE^{ZX}

SET Accumulator from E

`A←unsigned($80)>>(E∧7)`

Takes the bit number to set from `E` (only the low 3 bits) and sets the value of the accumulator `A` to the value of that bit, but counted from top to bottom (`E=0` will produce `A←$80`, `E=7` will produce `A←$01` and so on). This works as pixel mask for ULA bitmap modes, when `E` represents x-coordinate `0-255`.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
-	-		-		-	-	-

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

SLA s

Shift Left Arithmetic

$\boxed{\text{CF}} \leftarrow \underset{\texttt{s}}{\boxed{7 \leftarrow 0}} \leftarrow 0$

```
SLA A
SLA B
SLA C
SLA D
SLA E
SLA H
SLA L
SLA (HL)
SLA (IX+d)
SLA (IY+d)
SLA (IX+d),A**
SLA (IX+d),B**
SLA (IX+d),C**
SLA (IX+d),D**
SLA (IX+d),E**
SLA (IX+d),H**
SLA (IX+d),L**
SLA (IY+d),A**
SLA (IY+d),B**
SLA (IY+d),C**
SLA (IY+d),D**
SLA (IY+d),E**
SLA (IY+d),H**
SLA (IY+d),L**
```

Performs arithmetic shift left of the operand `s` or memory location addressed by `s`. Bit `0` is forced to `0` and bit `7` is moved to **CF**.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	↕

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set
CF	set to:	bit 7 of the original value

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

SLL

Shift Left Logical

This mnemonic has no associated opcode on Next. There is no difference between logical and arithmetic shift left, use `SLA` for both. Some assemblers will allow `SLL` as equivalent, but unfortunately, some will assemble it as `SLI`, so it's best avoiding.

SLI s** — **Shift Left and Increment**

SL1 s** — **Shift Left and add 1**

$$\boxed{\texttt{CF}} \leftarrow \underset{\texttt{s}}{\boxed{7 \leftarrow 0}} \leftarrow \texttt{1}$$

```
SLI A              SLI (IX+d),A**       SLI (IY+d),A**
SLI B              SLI (IX+d),B**       SLI (IY+d),B**
SLI C              SLI (IX+d),C**       SLI (IY+d),C**
SLI D              SLI (IX+d),D**       SLI (IY+d),D**
SLI E              SLI (IX+d),E**       SLI (IY+d),E**
SLI H              SLI (IX+d),H**       SLI (IY+d),H**
SLI L              SLI (IX+d),L**       SLI (IY+d),L**
SLA (HL)
SLA (IX+d)
SLA (IY+d)
```

Undocumented instruction. Similar to `SLA` except `1` is moved to bit `0`.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	↕

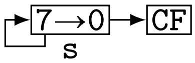
SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set
CF	set to:	bit 7 of the original value

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

Note: most assemblers will accept both variants: `SLI` or `SL1`, but some may only accept one or the other, while some may expect `SLL` instead.

SRA s

Shift Right Arithmetic



SRA A
SRA B
SRA C
SRA D
SRA E
SRA H
SRA L

SRA (HL)
SRA (IX+d)
SRA (IY+d)

SRA (IX+d),A**
SRA (IX+d),B**
SRA (IX+d),C**
SRA (IX+d),D**
SRA (IX+d),E**
SRA (IX+d),H**
SRA (IX+d),L**

SRA (IY+d),A**
SRA (IY+d),B**
SRA (IY+d),C**
SRA (IY+d),D**
SRA (IY+d),E**
SRA (IY+d),H**
SRA (IY+d),L**

Performs arithmetic shift right of the operand s or memory location addressed by s. Bit 0 is moved to **CF** while bit 7 remains unchanged (on the assumption that it's the sign bit).

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	↕

SF set if: result is negative (bit 7 is set)

ZF set if: result is 0

PV set if: result has even number of bits set

CF set to: bit 0 of the original value

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	2	8	2,3μs	1,14μs	0,57μs	0,29μs
(HL)	4	15	4,3μs	2,14μs	1,07μs	0,54μs
(IX+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs
(IY+d)	6	23	6,6μs	3,29μs	1,64μs	0,82μs

SRL s

Shift Right Logical

$0 \rightarrow \underset{\texttt{s}}{\boxed{7 \rightarrow 0}} \rightarrow \boxed{CF}$

```
SRL A    SRL (HL)         SRL (IX+d),A**       SRL (IY+d),A**
SRL B    SRL (IX+d)       SRL (IX+d),B**       SRL (IY+d),B**
SRL C    SRL (IY+d)       SRL (IX+d),C**       SRL (IY+d),C**
SRL D                     SRL (IX+d),D**       SRL (IY+d),D**
SRL E                     SRL (IX+d),E**       SRL (IY+d),E**
SRL H                     SRL (IX+d),H**       SRL (IY+d),H**
SRL L                     SRL (IX+d),L**       SRL (IY+d),L**
```

Performs logical shift right of the operand `s` or memory location addressed by `s`. Bit `0` is moved to **CF** while `0` is moved to bit `7`.

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	↕

SF	set if:	result is negative (bit `7` is set)
ZF	set if:	result is `0`
PV	set if:	result has even number of bits set
CF	set to:	bit 0 of the original value

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r`	2	8	2,3μs	1,14μs	0,57μs	0,29μs
`(HL)`	4	15	4,3μs	2,14μs	1,07μs	0,54μs
`(IX+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs
`(IY+d)`	6	23	6,6μs	3,29μs	1,64μs	0,82μs

SBC d,s

SuBtract with Carry

`d←d-s-CF`

8 bit	8 bit	16 bit
`SBC A,A`	`SBC A,IXH`**	`SBC HL,BC`
`SBC A,B`	`SBC A,IXL`**	`SBC HL,DE`
`SBC A,C`	`SBC A,IYH`**	`SBC HL,HL`
`SBC A,D`	`SBC A,IYL`**	`SBC HL,SP`
`SBC A,E`	`SBC A,(HL)`	
`SBC A,H`	`SBC A,(IX+d)`	
`SBC A,L`	`SBC A,(IY+d)`	
`SBC A,n`		

Subtracts source operand `s` or contents of the memory location addressed by `s` and carry flag **CF** from destination `d`. Result is then stored to destination `d`.

Effects

	SF	ZF		HF		PⓋ	NF	CF
8-bit	↕	↕		↕		↕	1	↕
16-bit	↕	↕		↕		↕	1	↕

SF set if: result is negative (bit 7 is set)

ZF set if: result is 0

HF set if: borrow from bit 4 (bit 12 for 16-bit)

PV set if:
- both operands positive and result negative
- both operands negative and result positve

CF set if: borrow from bit 8 (bit 16 for 16-bit)

Timing

	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
`r`	1	4	1,1μs	0,57μs	0,29μs	0,14μs
`n`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`(HL)`	2	7	2,0μs	1,00μs	0,50μs	0,25μs
`HL,rr`	4	15	4,3μs	2,14μs	1,07μs	0,54μs
`(IX+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs
`(IY+d)`	5	19	5,4μs	2,71μs	1,36μs	0,68μs

SUB s

SUBtract

A←A-s

SUB A
SUB B
SUB C
SUB D
SUB E
SUB H
SUB L
SUB n
SUB (HL)
SUB (IX+d)
SUB (IY+d)
SUB IXH**
SUB IXL**
SUB IYH**
SUB IYL**

Subtracts 8-bit immediate value, operand `s` or memory location addressed by `s` from accumulator `A`. Then stores result back to `A`.

Effects

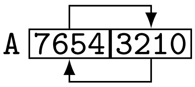
SF	ZF		HF		PⓋ	NF	CF
↕	↕		↕		↕	1	↕

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
HF	set if:	borrow from bit 4 (bit 12 for 16-bit)
PV	set if:	• both operands positive and result negative • both operands negative and result positve
CF	set if:	borrow from bit 8 (bit 16 for 16-bit)

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	4	1,1μs	0,57μs	0,29μs	0,14μs
n	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(HL)	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(IX+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IY+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs

SWAPNIB^{ZX}

SWAP NIBbles



Swaps the high and low nibbles of the accumulator A.

Effects

No effect on flags

SF	ZF		HF		PV	NF	CF
–	–		–		–	–	–

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
2	8	2,3μs	1,14μs	0,57μs	0,29μs

TEST n^{ZX}

TEST

$A \wedge n$

Similar to CP (page 157), but performs an AND instead of a subtraction. Again, AND is performed between the accumulator A and value n. Status flags are updated according to the result, but the result is then discarded (value of A is not changed).

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		1		↕	?	0

SF set if: result is negative (bit 7 is set)

ZF set if: result is 0 (no bits matched)

PV set if: result has even number of bits set

Timing

Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
3	11	3,1μs	1,57μs	0,79μs	0,39μs

XOR s **bitwise eXclusive OR**

`A←A⊻s`

`XOR A`
`XOR B`
`XOR C`
`XOR D`
`XOR E`
`XOR H`
`XOR L`
`XOR n`

`XOR (HL)`
`XOR (IX+d)`
`XOR (IY+d)`

`XOR IXH`**
`XOR IXL`**
`XOR IYH`**
`XOR IYL`**

Performs exclusive or between accumulator `A` and operand `s` or memory location addressed by `s`. Result is then stored back to `A`. Individual bits are XOR'ed as shown on the right:

A	s	Result
0	0	0
0	1	1
1	0	1
1	1	0

Effects

SF	ZF		HF		PV	NF	CF
↕	↕		0		↕	0	0

SF	set if:	result is negative (bit 7 is set)
ZF	set if:	result is 0
PV	set if:	result has even number of bits set

Timing	Mc	Ts	3.5MHz	7MHz	14MHz	28MHz
r	1	4	1,1μs	0,57μs	0,29μs	0,14μs
n	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(HL)	2	7	2,0μs	1,00μs	0,50μs	0,25μs
(IX+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs
(IY+d)	5	19	5,4μs	2,71μs	1,36μs	0,68μs

Appendix A

Instructions Sorted by Mnemonic

Instructions marked with ** are undocumented.
Instructions marked with ^{ZX} are ZX Spectrum Next extended.

ADC A,A	8F
ADC A,B	88
ADC A,C	89
ADC A,D	8A
ADC A,E	8B
ADC A,H	8C
ADC A,L	8D
ADC A,n	CE n
ADC A,(HL)	8E
ADC A,(IX+d)	DD8E d
ADC A,(IY+d)	FD8E d
ADC A,IXH**	DD8C
ADC A,IXL**	DD8D
ADC A,IYH**	FD8C
ADC A,IYL**	FD8D
ADC HL,BC	ED4A
ADC HL,DE	ED5A
ADC HL,HL	ED6A
ADC HL,SP	ED7A
ADD A,A	87
ADD A,B	80
ADD A,C	81
ADD A,D	82
ADD A,E	83
ADD A,H	84
ADD A,L	85
ADD A,n	C6 n
ADD A,(HL)	86
ADD A,(IX+d)	DD86 d
ADD A,(IY+d)	FD86 d
ADD A,IXH**	DD84
ADD A,IXL**	DD85
ADD A,IYH**	FD84
ADD A,IYL**	FD85
ADD BC,A^{ZX}	ED33
ADD BC,nm^{ZX}	ED36 m n
ADD DE,A^{ZX}	ED32
ADD DE,nm^{ZX}	ED35 m n
ADD HL,A^{ZX}	ED31
ADD HL,BC	09
ADD HL,DE	19
ADD HL,HL	29
ADD HL,SP	39
ADD HL,nm^{ZX}	ED34 m n
ADD IX,BC	DD09
ADD IX,DE	DD19
ADD IX,IX	DD29
ADD IX,SP	DD39
ADD IY,BC	FD09
ADD IY,DE	FD19
ADD IY,IY	FD29
ADD IY,SP	FD39
AND A	A7
AND B	A0
AND C	A1
AND D	A2
AND E	A3
AND H	A4
AND L	A5
AND n	E6 n
AND (HL)	A6
AND (IX+d)	DDA6 d
AND (IY+d)	FDA6 d
AND IXH**	DDA4
AND IXL**	DDA5
AND IYH**	FDA4
AND IYL**	FDA5
BIT 0,A	CB47
BIT 0,B	CB40
BIT 0,C	CB41
BIT 0,D	CB42
BIT 0,E	CB43
BIT 0,H	CB44
BIT 0,L	CB45
BIT 0,(HL)	CB46
BIT 0,(IX+d)	DDCB d 46
BIT 0,(IX+d)**	DDCB d 40
BIT 0,(IX+d)**	DDCB d 41
BIT 0,(IX+d)**	DDCB d 42
BIT 0,(IX+d)**	DDCB d 43
BIT 0,(IX+d)**	DDCB d 44
BIT 0,(IX+d)**	DDCB d 45
BIT 0,(IX+d)**	DDCB d 47
BIT 0,(IY+d)	FDCB d 46
BIT 0,(IY+d)**	FDCB d 40
BIT 0,(IY+d)**	FDCB d 41
BIT 0,(IY+d)**	FDCB d 42
BIT 0,(IY+d)**	FDCB d 43
BIT 0,(IY+d)**	FDCB d 44
BIT 0,(IY+d)**	FDCB d 45
BIT 0,(IY+d)**	FDCB d 47
BIT 1,A	CB4F
BIT 1,B	CB48
BIT 1,C	CB49
BIT 1,D	CB4A
BIT 1,E	CB4B
BIT 1,H	CB4C
BIT 1,L	CB4D
BIT 1,(HL)	CB4E
BIT 1,(IX+d)	DDCB d 4E
BIT 1,(IX+d)**	DDCB d 48
BIT 1,(IX+d)**	DDCB d 49
BIT 1,(IX+d)**	DDCB d 4A
BIT 1,(IX+d)**	DDCB d 4B
BIT 1,(IX+d)**	DDCB d 4C
BIT 1,(IX+d)**	DDCB d 4D
BIT 1,(IX+d)**	DDCB d 4F

BIT 1,(IY+d)	FDCB d 4E
BIT 1,(IY+d)**	FDCB d 48
BIT 1,(IY+d)**	FDCB d 49
BIT 1,(IY+d)**	FDCB d 4A
BIT 1,(IY+d)**	FDCB d 4B
BIT 1,(IY+d)**	FDCB d 4C
BIT 1,(IY+d)**	FDCB d 4D
BIT 1,(IY+d)**	FDCB d 4F
BIT 2,A	CB57
BIT 2,B	CB50
BIT 2,C	CB51
BIT 2,D	CB52
BIT 2,E	CB53
BIT 2,H	CB54
BIT 2,L	CB55
BIT 2,(HL)	CB56
BIT 2,(IX+d)	DDCB d 56
BIT 2,(IX+d)**	DDCB d 50
BIT 2,(IX+d)**	DDCB d 51
BIT 2,(IX+d)**	DDCB d 52
BIT 2,(IX+d)**	DDCB d 53
BIT 2,(IX+d)**	DDCB d 54
BIT 2,(IX+d)**	DDCB d 55
BIT 2,(IX+d)**	DDCB d 57
BIT 2,(IY+d)	FDCB d 56
BIT 2,(IY+d)**	FDCB d 50
BIT 2,(IY+d)**	FDCB d 51
BIT 2,(IY+d)**	FDCB d 52
BIT 2,(IY+d)**	FDCB d 53
BIT 2,(IY+d)**	FDCB d 54
BIT 2,(IY+d)**	FDCB d 55
BIT 2,(IY+d)**	FDCB d 57
BIT 3,A	CB5F
BIT 3,B	CB58
BIT 3,C	CB59
BIT 3,D	CB5A
BIT 3,E	CB5B
BIT 3,H	CB5C
BIT 3,L	CB5D
BIT 3,(HL)	CB5E
BIT 3,(IX+d)	DDCB d 5E
BIT 3,(IX+d)**	DDCB d 58
BIT 3,(IX+d)**	DDCB d 59
BIT 3,(IX+d)**	DDCB d 5A
BIT 3,(IX+d)**	DDCB d 5B
BIT 3,(IX+d)**	DDCB d 5C
BIT 3,(IX+d)**	DDCB d 5D
BIT 3,(IX+d)**	DDCB d 5F
BIT 3,(IY+d)	FDCB d 5E
BIT 3,(IY+d)**	FDCB d 58
BIT 3,(IY+d)**	FDCB d 59
BIT 3,(IY+d)**	FDCB d 5A
BIT 3,(IY+d)**	FDCB d 5B
BIT 3,(IY+d)**	FDCB d 5C
BIT 3,(IY+d)**	FDCB d 5D
BIT 3,(IY+d)**	FDCB d 5F
BIT 4,A	CB67
BIT 4,B	CB60
BIT 4,C	CB61
BIT 4,D	CB62
BIT 4,E	CB63
BIT 4,H	CB64
BIT 4,L	CB65
BIT 4,(HL)	CB66
BIT 4,(IX+d)	DDCB d 66
BIT 4,(IX+d)**	DDCB d 60
BIT 4,(IX+d)**	DDCB d 61
BIT 4,(IX+d)**	DDCB d 62
BIT 4,(IX+d)**	DDCB d 63
BIT 4,(IX+d)**	DDCB d 64
BIT 4,(IX+d)**	DDCB d 65
BIT 4,(IX+d)**	DDCB d 67
BIT 4,(IY+d)	FDCB d 66
BIT 4,(IY+d)**	FDCB d 60
BIT 4,(IY+d)**	FDCB d 61
BIT 4,(IY+d)**	FDCB d 62
BIT 4,(IY+d)**	FDCB d 63
BIT 4,(IY+d)**	FDCB d 64
BIT 4,(IY+d)**	FDCB d 65
BIT 4,(IY+d)**	FDCB d 67
BIT 5,A	CB6F
BIT 5,B	CB68
BIT 5,C	CB69
BIT 5,D	CB6A
BIT 5,E	CB6B
BIT 5,H	CB6C
BIT 5,L	CB6D
BIT 5,(HL)	CB6E
BIT 5,(IX+d)	DDCB d 6E
BIT 5,(IX+d)**	DDCB d 68
BIT 5,(IX+d)**	DDCB d 69
BIT 5,(IX+d)**	DDCB d 6A
BIT 5,(IX+d)**	DDCB d 6B
BIT 5,(IX+d)**	DDCB d 6C
BIT 5,(IX+d)**	DDCB d 6D
BIT 5,(IX+d)**	DDCB d 6F
BIT 5,(IY+d)	FDCB d 6E
BIT 5,(IY+d)**	FDCB d 68
BIT 5,(IY+d)**	FDCB d 69
BIT 5,(IY+d)**	FDCB d 6A
BIT 5,(IY+d)**	FDCB d 6B
BIT 5,(IY+d)**	FDCB d 6C
BIT 5,(IY+d)**	FDCB d 6D
BIT 5,(IY+d)**	FDCB d 6F
BIT 6,A	CB77
BIT 6,B	CB70
BIT 6,C	CB71
BIT 6,D	CB72
BIT 6,E	CB73
BIT 6,H	CB74
BIT 6,L	CB75
BIT 6,(HL)	CB76
BIT 6,(IX+d)	DDCB d 76
BIT 6,(IX+d)**	DDCB d 70
BIT 6,(IX+d)**	DDCB d 71
BIT 6,(IX+d)**	DDCB d 72
BIT 6,(IX+d)**	DDCB d 73
BIT 6,(IX+d)**	DDCB d 74
BIT 6,(IX+d)**	DDCB d 75
BIT 6,(IX+d)**	DDCB d 77
BIT 6,(IY+d)	FDCB d 76
BIT 6,(IY+d)**	FDCB d 70
BIT 6,(IY+d)**	FDCB d 71
BIT 6,(IY+d)**	FDCB d 72
BIT 6,(IY+d)**	FDCB d 73
BIT 6,(IY+d)**	FDCB d 74
BIT 6,(IY+d)**	FDCB d 75
BIT 6,(IY+d)**	FDCB d 77
BIT 7,A	CB7F
BIT 7,B	CB78
BIT 7,C	CB79
BIT 7,D	CB7A
BIT 7,E	CB7B
BIT 7,H	CB7C
BIT 7,L	CB7D
BIT 7,(HL)	CB7E
BIT 7,(IX+d)	DDCB d 7E
BIT 7,(IX+d)**	DDCB d 78
BIT 7,(IX+d)**	DDCB d 79
BIT 7,(IX+d)**	DDCB d 7A
BIT 7,(IX+d)**	DDCB d 7B
BIT 7,(IX+d)**	DDCB d 7C
BIT 7,(IX+d)**	DDCB d 7D
BIT 7,(IX+d)**	DDCB d 7F
BIT 7,(IY+d)	FDCB d 7E
BIT 7,(IY+d)**	FDCB d 78
BIT 7,(IY+d)**	FDCB d 79
BIT 7,(IY+d)**	FDCB d 7A
BIT 7,(IY+d)**	FDCB d 7B
BIT 7,(IY+d)**	FDCB d 7C
BIT 7,(IY+d)**	FDCB d 7D
BIT 7,(IY+d)**	FDCB d 7F
BRLC DE,B^{ZX}	ED2C
BSLA DE,B^{ZX}	ED28
BSRA DE,B^{ZX}	ED29
BSRF DE,B^{ZX}	ED2B
BSRL DE,B^{ZX}	ED2A
CALL nm	CD m n
CALL C,nm	DC m n
CALL M,nm	FC m n
CALL NC,nm	D4 m n
CALL NZ,nm	C4 m n
CALL P,nm	F4 m n
CALL PE,nm	EC m n
CALL PO,nm	E4 m n
CALL Z,nm	CC m n
CCF	3F

CP A	BF
CP B	B8
CP C	B9
CP D	BA
CP E	BB
CP H	BC
CP L	BD
CP n	FE n
CP (HL)	BE
CP (IX+d)	DDBE d
CP (IY+d)	FDBE d
CP IXH**	DDBC
CP IXL**	DDBD
CP IYH**	FDBC
CP IYL**	FDBD
CPDR	EDB9
CPD	EDA9
CPIR	EDB1
CPI	EDA1
CPL	2F
DAA	27
DEC (HL)	35
DEC (IX+d)	DD35 d
DEC (IY+d)	FD35 d
DEC A	3D
DEC B	05
DEC C	0D
DEC D	15
DEC E	1D
DEC H	25
DEC L	2D
DEC BC	0B
DEC DE	1B
DEC HL	2B
DEC IX	DD2B
DEC IXH**	DD25
DEC IXL**	DD2D
DEC IY	FD2B
DEC IYH**	FD25
DEC IYL**	FD2D
DEC SP	3B
DI	F3
DJNZ (PC+e)	10 e
EI	FB
EX (SP),HL	E3
EX (SP),IX	DDE3
EX (SP),IY	FDE3
EX AF,AF’	08
EX DE,HL	EB
EXX	D9
HALT	76
IM 0**	ED4E
IM 0**	ED66
IM 0**	ED6E
IM 0	ED46
IM 1**	ED76
IM 1	ED56
IM 2**	ED7E
IM 2	ED5E
IN A,(C)	ED78
IN A,(n)	DB n
IN B,(C)	ED40
IN C,(C)	ED48
IN D,(C)	ED50
IN E,(C)	ED58
IN F,(C)**	ED70
IN H,(C)	ED60
IN L,(C)	ED68
IN (C)**	ED70
INC (HL)	34
INC (IX+d)	DD34 d
INC (IY+d)	FD34 d
INC A	3C
INC B	04
INC C	0C
INC D	14
INC E	1C
INC H	24
INC L	2C
INC BC	03
INC DE	13
INC HL	23
INC IX	DD23
INC IXH**	DD24
INC IXL**	DD2C
INC IY	FD23
INC IYH**	FD24
INC IYL**	FD2C
INC SP	33
INDR	EDBA
IND	EDAA
INIR	EDB2
INI	EDA2
JP (C)^{ZX}	ED98
JP (HL)	E9
JP (IX)	DDE9
JP (IY)	FDE9
JP nm	C3 m n
JP C,nm	DA m n
JP M,nm	FA m n
JP NC,nm	D2 m n
JP NZ,nm	C2 m n
JP P,nm	F2 m n
JP PE,nm	EA m n
JP PO,nm	E2 m n
JP Z,nm	CA m n
JR e	18 e
JR C,e	38 e
JR NC,e	30 e
JR NZ,e	20 e
JR Z,e	28 e
LD (BC),A	02
LD (DE),A	12
LD (HL),A	77
LD (HL),B	70
LD (HL),C	71
LD (HL),D	72
LD (HL),E	73
LD (HL),H	74
LD (HL),L	75
LD (HL),n	36 n
LD (IX+d),A	DD77 d
LD (IX+d),B	DD70 d
LD (IX+d),C	DD71 d
LD (IX+d),D	DD72 d
LD (IX+d),E	DD73 d
LD (IX+d),H	DD74 d
LD (IX+d),L	DD75 d
LD (IX+d),n	DD36 d n
LD (IY+d),A	FD77 d
LD (IY+d),B	FD70 d
LD (IY+d),C	FD71 d
LD (IY+d),D	FD72 d
LD (IY+d),E	FD73 d
LD (IY+d),H	FD74 d
LD (IY+d),L	FD75 d
LD (IY+d),n	FD36 d n
LD (nm),A	32 m n
LD (nm),BC	ED43 m n
LD (nm),DE	ED53 m n
LD (nm),HL	22 m n
LD (nm),HL	ED63 m n
LD (nm),IX	DD22 m n
LD (nm),IY	FD22 m n
LD (nm),SP	ED73 m n
LD A,A	7F
LD A,B	78
LD A,C	79
LD A,D	7A
LD A,E	7B
LD A,H	7C
LD A,I	ED57
LD A,L	7D
LD A,R	ED5F
LD A,n	3E n
LD A,(BC)	0A
LD A,(DE)	1A
LD A,(HL)	7E
LD A,(IX+d)	DD7E d
LD A,(IY+d)	FD7E d
LD A,(nm)	3A m n
LD A,IXH**	DD7C
LD A,IXL**	DD7D
LD A,IYH**	FD7C
LD A,IYL**	FD7D
LD B,A	47
LD B,B	40
LD B,C	41
LD B,D	42
LD B,E	43

LD B,H	44
LD B,L	45
LD B,n	06 n
LD B,(HL)	46
LD B,(IX+d)	DD46 d
LD B,(IY+d)	FD46 d
LD B,IXH**	DD44
LD B,IXL**	DD45
LD B,IYH**	FD44
LD B,IYL**	FD45
LD BC,(nm)	ED4B m n
LD BC,nm	01 m n
LD C,A	4F
LD C,B	48
LD C,C	49
LD C,D	4A
LD C,E	4B
LD C,H	4C
LD C,L	4D
LD C,n	0E n
LD C,(HL)	4E
LD C,(IX+d)	DD4E d
LD C,(IY+d)	FD4E d
LD C,IXH**	DD4C
LD C,IXL**	DD4D
LD C,IYH**	FD4C
LD C,IYL**	FD4D
LD D,A	57
LD D,B	50
LD D,C	51
LD D,D	52
LD D,E	53
LD D,H	54
LD D,L	55
LD D,n	16 n
LD D,(HL)	56
LD D,(IX+d)	DD56 d
LD D,(IY+d)	FD56 d
LD D,IXH**	DD54
LD D,IXL**	DD55
LD D,IYH**	FD54
LD D,IYL**	FD55
LD DE,(nm)	ED5B m n
LD DE,nm	11 m n
LD E,A	5F
LD E,B	58
LD E,C	59
LD E,D	5A
LD E,E	5B
LD E,H	5C
LD E,L	5D
LD E,n	1E n
LD E,(HL)	5E
LD E,(IX+d)	DD5E d
LD E,(IY+d)	FD5E d
LD E,IXH**	DD5C
LD E,IXL**	DD5D
LD E,IYH**	FD5C
LD E,IYL**	FD5D
LD H,A	67
LD H,B	60
LD H,C	61
LD H,D	62
LD H,E	63
LD H,H	64
LD H,L	65
LD H,n	26 n
LD H,(HL)	66
LD H,(IX+d)	DD66 d
LD H,(IY+d)	FD66 d
LD HL,(nm)	2A m n
LD HL,(nm)	ED6B m n
LD HL,nm	21 m n
LD I,A	ED47
LD IX,(nm)	DD2A m n
LD IX,nm	DD21 m n
LD IXH,A**	DD67
LD IXH,B**	DD60
LD IXH,C**	DD61
LD IXH,D**	DD62
LD IXH,E**	DD63
LD IXH,IXH**	DD64
LD IXH,IXL**	DD65
LD IXH,n**	DD26 n
LD IXL,A**	DD6F
LD IXL,B**	DD68
LD IXL,C**	DD69
LD IXL,D**	DD6A
LD IXL,E**	DD6B
LD IXL,IXH**	DD6C
LD IXL,IXL**	DD6D
LD IXL,n**	DD2E n
LD IY,(nm)	FD2A m n
LD IY,nm	FD21 m n
LD IYH,A**	FD67
LD IYH,B**	FD60
LD IYH,C**	FD61
LD IYH,D**	FD62
LD IYH,E**	FD63
LD IYH,IYH**	FD64
LD IYH,IYL**	FD65
LD IYH,n**	FD26 n
LD IYL,A**	FD6F
LD IYL,B**	FD68
LD IYL,C**	FD69
LD IYL,D**	FD6A
LD IYL,E**	FD6B
LD IYL,IYH**	FD6C
LD IYL,IYL**	FD6D
LD L,A	6F
LD L,B	68
LD L,C	69
LD L,D	6A
LD L,E	6B
LD L,H	6C
LD L,L	6D
LD L,n	2E n
LD IYL,n**	FD2E n
LD L,(HL)	6E
LD L,(IX+d)	DD6E d
LD L,(IY+d)	FD6E d
LD R,A	ED4F
LD SP,(nm)	ED7B m n
LD SP,HL	F9
LD SP,IX	DDF9
LD SP,IY	FDF9
LD SP,nm	31 m n
LDD	EDA8
LDDR	EDB8
LDDX^{ZX}	EDAC
LDDRX^{ZX}	EDBC
LDI	EDA0
LDIR	EDB0
LDIX^{ZX}	EDA4
LDIRX^{ZX}	EDB4
LDPIRX^{ZX}	EDB7
LDWS^{ZX}	EDA5
MIRROR A^{ZX}	ED24
MUL D,E^{ZX}	ED30
NEG**	ED4C
NEG**	ED54
NEG**	ED5C
NEG**	ED64
NEG**	ED6C
NEG**	ED74
NEG**	ED7C
NEG	ED44
NEXTREG r,n^{ZX}	ED91 r n
NEXTREG r,A^{ZX}	ED92 r
NOP	00
OR A	B7
OR B	B0
OR C	B1
OR D	B2
OR E	B3
OR H	B4
OR L	B5
OR n	F6 n
OR (HL)	B6
OR (IX+d)	DDB6 d
OR (IY+d)	FDB6 d
OR IXH**	DDB4
OR IXL**	DDB5
OR IYH**	FDB4
OR IYL**	FDB5
OTDR	EDBB
OTIR	EDB3
OUT (C),0**	ED71

OUT (C),A	ED79
OUT (C),B	ED41
OUT (C),C	ED49
OUT (C),D	ED51
OUT (C),E	ED59
OUT (C),H	ED61
OUT (C),L	ED69
OUT (n),A	D3 n
OUTD	EDAB
OUTI	EDA3
OUTINB^{ZX}	ED90
PIXELAD^{ZX}	ED94
PIXELDN^{ZX}	ED93
POP AF	F1
POP BC	C1
POP DE	D1
POP HL	E1
POP IX	DDE1
POP IY	FDE1
PUSH AF	F5
PUSH BC	C5
PUSH DE	D5
PUSH HL	E5
PUSH IX	DDE5
PUSH IY	FDE5
PUSH nm^{ZX}	ED8A n m
RES 0,A	CB87
RES 0,B	CB80
RES 0,C	CB81
RES 0,D	CB82
RES 0,E	CB83
RES 0,H	CB84
RES 0,L	CB85
RES 0,(HL)	CB86
RES 0,(IX+d)	DDCB d 86
RES 0,(IX+d),A**	DDCB d 87
RES 0,(IX+d),B**	DDCB d 80
RES 0,(IX+d),C**	DDCB d 81
RES 0,(IX+d),D**	DDCB d 82
RES 0,(IX+d),E**	DDCB d 83
RES 0,(IX+d),H**	DDCB d 84
RES 0,(IX+d),L**	DDCB d 85
RES 0,(IY+d)	FDCB d 86
RES 0,(IY+d),A**	FDCB d 87
RES 0,(IY+d),B**	FDCB d 80
RES 0,(IY+d),C**	FDCB d 81
RES 0,(IY+d),D**	FDCB d 82
RES 0,(IY+d),E**	FDCB d 83
RES 0,(IY+d),H**	FDCB d 84
RES 0,(IY+d),L**	FDCB d 85
RES 1,A	CB8F
RES 1,B	CB88
RES 1,C	CB89
RES 1,D	CB8A
RES 1,E	CB8B
RES 1,H	CB8C
RES 1,L	CB8D
RES 1,(HL)	CB8E
RES 1,(IX+d)	DDCB d 8E
RES 1,(IX+d),A**	DDCB d 8F
RES 1,(IX+d),B**	DDCB d 88
RES 1,(IX+d),C**	DDCB d 89
RES 1,(IX+d),D**	DDCB d 8A
RES 1,(IX+d),E**	DDCB d 8B
RES 1,(IX+d),H**	DDCB d 8C
RES 1,(IX+d),L**	DDCB d 8D
RES 1,(IY+d)	FDCB d 8E
RES 1,(IY+d),A**	FDCB d 8F
RES 1,(IY+d),B**	FDCB d 88
RES 1,(IY+d),C**	FDCB d 89
RES 1,(IY+d),D**	FDCB d 8A
RES 1,(IY+d),E**	FDCB d 8B
RES 1,(IY+d),H**	FDCB d 8C
RES 1,(IY+d),L**	FDCB d 8D
RES 2,A	CB97
RES 2,B	CB90
RES 2,C	CB91
RES 2,D	CB92
RES 2,E	CB93
RES 2,H	CB94
RES 2,L	CB95
RES 2,(HL)	CB96
RES 2,(IX+d)	DDCB d 96
RES 2,(IX+d),A**	DDCB d 97
RES 2,(IX+d),B**	DDCB d 90
RES 2,(IX+d),C**	DDCB d 91
RES 2,(IX+d),D**	DDCB d 92
RES 2,(IX+d),E**	DDCB d 93
RES 2,(IX+d),H**	DDCB d 94
RES 2,(IX+d),L**	DDCB d 95
RES 2,(IY+d)	FDCB d 96
RES 2,(IY+d),A**	FDCB d 97
RES 2,(IY+d),B**	FDCB d 90
RES 2,(IY+d),C**	FDCB d 91
RES 2,(IY+d),D**	FDCB d 92
RES 2,(IY+d),E**	FDCB d 93
RES 2,(IY+d),H**	FDCB d 94
RES 2,(IY+d),L**	FDCB d 95
RES 3,A	CB9F
RES 3,B	CB98
RES 3,C	CB99
RES 3,D	CB9A
RES 3,E	CB9B
RES 3,H	CB9C
RES 3,L	CB9D
RES 3,(HL)	CB9E
RES 3,(IX+d)	DDCB d 9E
RES 3,(IX+d),A**	DDCB d 9F
RES 3,(IX+d),B**	DDCB d 98
RES 3,(IX+d),C**	DDCB d 99
RES 3,(IX+d),D**	DDCB d 9A
RES 3,(IX+d),E**	DDCB d 9B
RES 3,(IX+d),H**	DDCB d 9C
RES 3,(IX+d),L**	DDCB d 9D
RES 3,(IY+d)	FDCB d 9E
RES 3,(IY+d),A**	FDCB d 9F
RES 3,(IY+d),B**	FDCB d 98
RES 3,(IY+d),C**	FDCB d 99
RES 3,(IY+d),D**	FDCB d 9A
RES 3,(IY+d),E**	FDCB d 9B
RES 3,(IY+d),H**	FDCB d 9C
RES 3,(IY+d),L**	FDCB d 9D
RES 4,A	CBA7
RES 4,B	CBA0
RES 4,C	CBA1
RES 4,D	CBA2
RES 4,E	CBA3
RES 4,H	CBA4
RES 4,L	CBA5
RES 4,(HL)	CBA6
RES 4,(IX+d)	DDCB d A6
RES 4,(IX+d),A**	DDCB d A7
RES 4,(IX+d),B**	DDCB d A0
RES 4,(IX+d),C**	DDCB d A1
RES 4,(IX+d),D**	DDCB d A2
RES 4,(IX+d),E**	DDCB d A3
RES 4,(IX+d),H**	DDCB d A4
RES 4,(IX+d),L**	DDCB d A5
RES 4,(IY+d)	FDCB d A6
RES 4,(IY+d),A**	FDCB d A7
RES 4,(IY+d),B**	FDCB d A0
RES 4,(IY+d),C**	FDCB d A1
RES 4,(IY+d),D**	FDCB d A2
RES 4,(IY+d),E**	FDCB d A3
RES 4,(IY+d),H**	FDCB d A4
RES 4,(IY+d),L**	FDCB d A5
RES 5,A	CBAF
RES 5,B	CBA8
RES 5,C	CBA9
RES 5,D	CBAA
RES 5,E	CBAB
RES 5,H	CBAC
RES 5,L	CBAD
RES 5,(HL)	CBAE
RES 5,(IX+d)	DDCB d AE
RES 5,(IX+d),A**	DDCB d AF
RES 5,(IX+d),B**	DDCB d A8
RES 5,(IX+d),C**	DDCB d A9
RES 5,(IX+d),D**	DDCB d AA
RES 5,(IX+d),E**	DDCB d AB
RES 5,(IX+d),H**	DDCB d AC
RES 5,(IX+d),L**	DDCB d AD
RES 5,(IY+d)	FDCB d AE
RES 5,(IY+d),A**	FDCB d AF
RES 5,(IY+d),B**	FDCB d A8
RES 5,(IY+d),C**	FDCB d A9
RES 5,(IY+d),D**	FDCB d AA
RES 5,(IY+d),E**	FDCB d AB

RES 5,(IY+d),H**	FDCB d AC
RES 5,(IY+d),L**	FDCB d AD
RES 6,A	CBB7
RES 6,B	CBB0
RES 6,C	CBB1
RES 6,D	CBB2
RES 6,E	CBB3
RES 6,H	CBB4
RES 6,L	CBB5
RES 6,(HL)	CBB6
RES 6,(IX+d)	DDCB d B6
RES 6,(IX+d),A**	DDCB d B7
RES 6,(IX+d),B**	DDCB d B0
RES 6,(IX+d),C**	DDCB d B1
RES 6,(IX+d),D**	DDCB d B2
RES 6,(IX+d),E**	DDCB d B3
RES 6,(IX+d),H**	DDCB d B4
RES 6,(IX+d),L**	DDCB d B5
RES 6,(IY+d)	FDCB d B6
RES 6,(IY+d),A**	FDCB d B7
RES 6,(IY+d),B**	FDCB d B0
RES 6,(IY+d),C**	FDCB d B1
RES 6,(IY+d),D**	FDCB d B2
RES 6,(IY+d),E**	FDCB d B3
RES 6,(IY+d),H**	FDCB d B4
RES 6,(IY+d),L**	FDCB d B5
RES 7,A	CBBF
RES 7,B	CBB8
RES 7,C	CBB9
RES 7,D	CBBA
RES 7,E	CBBB
RES 7,H	CBBC
RES 7,L	CBBD
RES 7,(HL)	CBBE
RES 7,(IX+d)	DDCB d BE
RES 7,(IX+d),A**	DDCB d BF
RES 7,(IX+d),B**	DDCB d B8
RES 7,(IX+d),C**	DDCB d B9
RES 7,(IX+d),D**	DDCB d BA
RES 7,(IX+d),E**	DDCB d BB
RES 7,(IX+d),H**	DDCB d BC
RES 7,(IX+d),L**	DDCB d BD
RES 7,(IY+d)	FDCB d BE
RES 7,(IY+d),A**	FDCB d BF
RES 7,(IY+d),B**	FDCB d B8
RES 7,(IY+d),C**	FDCB d B9
RES 7,(IY+d),D**	FDCB d BA
RES 7,(IY+d),E**	FDCB d BB
RES 7,(IY+d),H**	FDCB d BC
RES 7,(IY+d),L**	FDCB d BD
RET C	D8
RET M	F8
RET NC	D0
RET NZ	C0
RET PE	E8
RET PO	E0
RET P	F0
RET Z	C8
RETI	ED4D
RETN**	ED55
RETN**	ED5D
RETN**	ED65
RETN**	ED6D
RETN**	ED75
RETN**	ED7D
RETN	ED45
RET	C9
RL A	CB17
RL B	CB10
RL C	CB11
RL D	CB12
RL E	CB13
RL H	CB14
RL L	CB15
RL (HL)	CB16
RL (IX+d)	DDCB d 16
RL (IX+d),A**	DDCB d 17
RL (IX+d),B**	DDCB d 10
RL (IX+d),C**	DDCB d 11
RL (IX+d),D**	DDCB d 12
RL (IX+d),E**	DDCB d 13
RL (IX+d),H**	DDCB d 14
RL (IX+d),L**	DDCB d 15
RL (IY+d)	FDCB d 16
RL (IY+d),A**	FDCB d 17
RL (IY+d),B**	FDCB d 10
RL (IY+d),C**	FDCB d 11
RL (IY+d),D**	FDCB d 12
RL (IY+d),E**	FDCB d 13
RL (IY+d),H**	FDCB d 14
RL (IY+d),L**	FDCB d 15
RLA	17
RLC A	CB07
RLC B	CB00
RLC C	CB01
RLC D	CB02
RLC E	CB03
RLC H	CB04
RLC L	CB05
RLC (HL)	CB06
RLC (IX+d)	DDCB d 06
RLC (IX+d),A**	DDCB d 07
RLC (IX+d),B**	DDCB d 00
RLC (IX+d),C**	DDCB d 01
RLC (IX+d),D**	DDCB d 02
RLC (IX+d),E**	DDCB d 03
RLC (IX+d),H**	DDCB d 04
RLC (IX+d),L**	DDCB d 05
RLC (IY+d)	FDCB d 06
RLC (IY+d),A**	FDCB d 07
RLC (IY+d),B**	FDCB d 00
RLC (IY+d),C**	FDCB d 01
RLC (IY+d),D**	FDCB d 02
RLC (IY+d),E**	FDCB d 03
RLC (IY+d),H**	FDCB d 04
RLC (IY+d),L**	FDCB d 05
RLCA	07
RLD	ED6F
RR A	CB1F
RR B	CB18
RR C	CB19
RR D	CB1A
RR E	CB1B
RR H	CB1C
RR L	CB1D
RR (HL)	CB1E
RR (IX+d)	DDCB d 1E
RR (IX+d),A**	DDCB d 1F
RR (IX+d),B**	DDCB d 18
RR (IX+d),C**	DDCB d 19
RR (IX+d),D**	DDCB d 1A
RR (IX+d),E**	DDCB d 1B
RR (IX+d),H**	DDCB d 1C
RR (IX+d),L**	DDCB d 1D
RR (IY+d)	FDCB d 1E
RR (IY+d),A**	FDCB d 1F
RR (IY+d),B**	FDCB d 18
RR (IY+d),C**	FDCB d 19
RR (IY+d),D**	FDCB d 1A
RR (IY+d),E**	FDCB d 1B
RR (IY+d),H**	FDCB d 1C
RR (IY+d),L**	FDCB d 1D
RRA	1F
RRC A	CB0F
RRC B	CB08
RRC C	CB09
RRC D	CB0A
RRC E	CB0B
RRC H	CB0C
RRC L	CB0D
RRC (HL)	CB0E
RRC (IX+d)	DDCB d 0E
RRC (IX+d),A**	DDCB d 0F
RRC (IX+d),B**	DDCB d 08
RRC (IX+d),C**	DDCB d 09
RRC (IX+d),D**	DDCB d 0A
RRC (IX+d),E**	DDCB d 0B
RRC (IX+d),H**	DDCB d 0C
RRC (IX+d),L**	DDCB d 0D
RRC (IY+d)	FDCB d 0E
RRC (IY+d),A**	FDCB d 0F
RRC (IY+d),B**	FDCB d 08
RRC (IY+d),C**	FDCB d 09
RRC (IY+d),D**	FDCB d 0A
RRC (IY+d),E**	FDCB d 0B
RRC (IY+d),H**	FDCB d 0C
RRC (IY+d),L**	FDCB d 0D
RRCA	0F

RRD	ED67
RST 0H	C7
RST 10H	D7
RST 18H	DF
RST 20H	E7
RST 28H	EF
RST 30H	F7
RST 38H	FF
RST 8H	CF
SBC A,A	9F
SBC A,B	98
SBC A,C	99
SBC A,D	9A
SBC A,E	9B
SBC A,H	9C
SBC A,L	9D
SBC A,n	DE n
SBC A,(HL)	9E
SBC A,(IX+d)	DD9E d
SBC A,(IY+d)	FD9E d
SBC A,IXH**	DD9C
SBC A,IXL**	DD9D
SBC A,IYH**	FD9C
SBC A,IYL**	FD9D
SBC HL,BC	ED42
SBC HL,DE	ED52
SBC HL,HL	ED62
SBC HL,SP	ED72
SCF	37
SET 0,A	CBC7
SET 0,B	CBC0
SET 0,C	CBC1
SET 0,D	CBC2
SET 0,E	CBC3
SET 0,H	CBC4
SET 0,L	CBC5
SET 0,(HL)	CBC6
SET 0,(IX+d)	DDCB d C6
SET 0,(IX+d),A**	DDCB d C7
SET 0,(IX+d),B**	DDCB d C0
SET 0,(IX+d),C**	DDCB d C1
SET 0,(IX+d),D**	DDCB d C2
SET 0,(IX+d),E**	DDCB d C3
SET 0,(IX+d),H**	DDCB d C4
SET 0,(IX+d),L**	DDCB d C5
SET 0,(IY+d)	FDCB d C6
SET 0,(IY+d),A**	FDCB d C7
SET 0,(IY+d),B**	FDCB d C0
SET 0,(IY+d),C**	FDCB d C1
SET 0,(IY+d),D**	FDCB d C2
SET 0,(IY+d),E**	FDCB d C3
SET 0,(IY+d),H**	FDCB d C4
SET 0,(IY+d),L**	FDCB d C5
SET 1,A	CBCF
SET 1,B	CBC8
SET 1,C	CBC9
SET 1,D	CBCA
SET 1,E	CBCB
SET 1,H	CBCC
SET 1,L	CBCD
SET 1,(HL)	CBCE
SET 1,(IX+d)	DDCB d CE
SET 1,(IX+d),A**	DDCB d CF
SET 1,(IX+d),B**	DDCB d C8
SET 1,(IX+d),C**	DDCB d C9
SET 1,(IX+d),D**	DDCB d CA
SET 1,(IX+d),E**	DDCB d CB
SET 1,(IX+d),H**	DDCB d CC
SET 1,(IX+d),L**	DDCB d CD
SET 1,(IY+d)	FDCB d CE
SET 1,(IY+d),A**	FDCB d CF
SET 1,(IY+d),B**	FDCB d C8
SET 1,(IY+d),C**	FDCB d C9
SET 1,(IY+d),D**	FDCB d CA
SET 1,(IY+d),E**	FDCB d CB
SET 1,(IY+d),H**	FDCB d CC
SET 1,(IY+d),L**	FDCB d CD
SET 2,A	CBD7
SET 2,B	CBD0
SET 2,C	CBD1
SET 2,D	CBD2
SET 2,E	CBD3
SET 2,H	CBD4
SET 2,L	CBD5
SET 2,(HL)	CBD6
SET 2,(IX+d)	DDCB d D6
SET 2,(IX+d),A**	DDCB d D7
SET 2,(IX+d),B**	DDCB d D0
SET 2,(IX+d),C**	DDCB d D1
SET 2,(IX+d),D**	DDCB d D2
SET 2,(IX+d),E**	DDCB d D3
SET 2,(IX+d),H**	DDCB d D4
SET 2,(IX+d),L**	DDCB d D5
SET 2,(IY+d)	FDCB d D6
SET 2,(IY+d),A**	FDCB d D7
SET 2,(IY+d),B**	FDCB d D0
SET 2,(IY+d),C**	FDCB d D1
SET 2,(IY+d),D**	FDCB d D2
SET 2,(IY+d),E**	FDCB d D3
SET 2,(IY+d),H**	FDCB d D4
SET 2,(IY+d),L**	FDCB d D5
SET 3,A	CBDF
SET 3,B	CBD8
SET 3,C	CBD9
SET 3,D	CBDA
SET 3,E	CBDB
SET 3,H	CBDC
SET 3,L	CBDD
SET 3,(HL)	CBDE
SET 3,(IY+d)	FDCB d DE
SET 3,(IX+d),A**	DDCB d DF
SET 3,(IX+d),B**	DDCB d D8
SET 3,(IX+d),C**	DDCB d D9
SET 3,(IX+d),D**	DDCB d DA
SET 3,(IX+d),E**	DDCB d DB
SET 3,(IX+d),H**	DDCB d DC
SET 3,(IX+d),L**	DDCB d DD
SET 3,(IY+d)	FDCB d DE
SET 3,(IY+d),A**	FDCB d DF
SET 3,(IY+d),B**	FDCB d D8
SET 3,(IY+d),C**	FDCB d D9
SET 3,(IY+d),D**	FDCB d DA
SET 3,(IY+d),E**	FDCB d DB
SET 3,(IY+d),H**	FDCB d DC
SET 3,(IY+d),L**	FDCB d DD
SET 4,A	CBE7
SET 4,B	CBE0
SET 4,C	CBE1
SET 4,D	CBE2
SET 4,E	CBE3
SET 4,H	CBE4
SET 4,L	CBE5
SET 4,(HL)	CBE6
SET 4,(IY+d)	FDCB d E6
SET 4,(IX+d),A**	DDCB d E7
SET 4,(IX+d),B**	DDCB d E0
SET 4,(IX+d),C**	DDCB d E1
SET 4,(IX+d),D**	DDCB d E2
SET 4,(IX+d),E**	DDCB d E3
SET 4,(IX+d),H**	DDCB d E4
SET 4,(IX+d),L**	DDCB d E5
SET 4,(IY+d)	FDCB d E6
SET 4,(IY+d),A**	FDCB d E7
SET 4,(IY+d),B**	FDCB d E0
SET 4,(IY+d),C**	FDCB d E1
SET 4,(IY+d),D**	FDCB d E2
SET 4,(IY+d),E**	FDCB d E3
SET 4,(IY+d),H**	FDCB d E4
SET 4,(IY+d),L**	FDCB d E5
SET 5,A	CBEF
SET 5,B	CBE8
SET 5,C	CBE9
SET 5,D	CBEA
SET 5,E	CBEB
SET 5,H	CBEC
SET 5,L	CBED
SET 5,(HL)	CBEE
SET 5,(IX+d)	DDCB d EE
SET 5,(IX+d),A**	DDCB d EF
SET 5,(IX+d),B**	DDCB d E8
SET 5,(IX+d),C**	DDCB d E9
SET 5,(IX+d),D**	DDCB d EA
SET 5,(IX+d),E**	DDCB d EB
SET 5,(IX+d),H**	DDCB d EC
SET 5,(IX+d),L**	DDCB d ED
SET 5,(IY+d)	FDCB d EE
SET 5,(IY+d),A**	FDCB d EF
SET 5,(IY+d),B**	FDCB d E8

SET 5,(IY+d),C**	FDCB d E9
SET 5,(IY+d),D**	FDCB d EA
SET 5,(IY+d),E**	FDCB d EB
SET 5,(IY+d),H**	FDCB d EC
SET 5,(IY+d),L**	FDCB d ED
SET 6,A	CBF7
SET 6,B	CBF0
SET 6,C	CBF1
SET 6,D	CBF2
SET 6,E	CBF3
SET 6,H	CBF4
SET 6,L	CBF5
SET 6,(HL)	CBF6
SET 6,(IX+d)	DDCB d F6
SET 6,(IX+d),A**	DDCB d F7
SET 6,(IX+d),B**	DDCB d F0
SET 6,(IX+d),C**	DDCB d F1
SET 6,(IX+d),D**	DDCB d F2
SET 6,(IX+d),E**	DDCB d F3
SET 6,(IX+d),H**	DDCB d F4
SET 6,(IX+d),L**	DDCB d F5
SET 6,(IY+d)	FDCB d F6
SET 6,(IY+d),A**	FDCB d F7
SET 6,(IY+d),B**	FDCB d F0
SET 6,(IY+d),C**	FDCB d F1
SET 6,(IY+d),D**	FDCB d F2
SET 6,(IY+d),E**	FDCB d F3
SET 6,(IY+d),H**	FDCB d F4
SET 6,(IY+d),L**	FDCB d F5
SET 7,A	CBFF
SET 7,B	CBF8
SET 7,C	CBF9
SET 7,D	CBFA
SET 7,E	CBFB
SET 7,H	CBFC
SET 7,L	CBFD
SET 7,(HL)	CBFE
SET 7,(IX+d)	DDCB d FE
SET 7,(IX+d),A**	DDCB d FF
SET 7,(IX+d),B**	DDCB d F8
SET 7,(IX+d),C**	DDCB d F9
SET 7,(IX+d),D**	DDCB d FA
SET 7,(IX+d),E**	DDCB d FB
SET 7,(IX+d),H**	DDCB d FC
SET 7,(IX+d),L**	DDCB d FD
SET 7,(IY+d)	FDCB d FE
SET 7,(IY+d),A**	FDCB d FF
SET 7,(IY+d),B**	FDCB d F8
SET 7,(IY+d),C**	FDCB d F9
SET 7,(IY+d),D**	FDCB d FA
SET 7,(IY+d),E**	FDCB d FB
SET 7,(IY+d),H**	FDCB d FC
SET 7,(IY+d),L**	FDCB d FD
SETAE^{ZX}	ED95
SLA A	CB27
SLA B	CB20
SLA C	CB21
SLA D	CB22
SLA E	CB23
SLA H	CB24
SLA L	CB25
SLA (HL)	CB26
SLA (IX+d)	DDCB d 26
SLA (IX+d),A**	DDCB d 27
SLA (IX+d),B**	DDCB d 20
SLA (IX+d),C**	DDCB d 21
SLA (IX+d),D**	DDCB d 22
SLA (IX+d),E**	DDCB d 23
SLA (IX+d),H**	DDCB d 24
SLA (IX+d),L**	DDCB d 25
SLA (IY+d)	FDCB d 26
SLA (IY+d),A**	FDCB d 27
SLA (IY+d),B**	FDCB d 20
SLA (IY+d),C**	FDCB d 21
SLA (IY+d),D**	FDCB d 22
SLA (IY+d),E**	FDCB d 23
SLA (IY+d),H**	FDCB d 24
SLA (IY+d),L**	FDCB d 25
SLI (HL)**	CB36
SLI A**	CB37
SLI B**	CB30
SLI C**	CB31
SLI D**	CB32
SLI E**	CB33
SLI H**	CB34
SLI L**	CB35
SLI (IX+d)**	DDCB d 36
SLI (IX+d),A**	DDCB d 37
SLI (IX+d),B**	DDCB d 30
SLI (IX+d),C**	DDCB d 31
SLI (IX+d),D**	DDCB d 32
SLI (IX+d),E**	DDCB d 33
SLI (IX+d),H**	DDCB d 34
SLI (IX+d),L**	DDCB d 35
SLI (IY+d)**	FDCB d 36
SLI (IY+d),A**	FDCB d 37
SLI (IY+d),B**	FDCB d 30
SLI (IY+d),C**	FDCB d 31
SLI (IY+d),D**	FDCB d 32
SLI (IY+d),E**	FDCB d 33
SLI (IY+d),H**	FDCB d 34
SLI (IY+d),L**	FDCB d 35
SRA A	CB2F
SRA B	CB28
SRA C	CB29
SRA D	CB2A
SRA E	CB2B
SRA H	CB2C
SRA L	CB2D
SRA (HL)	CB2E
SRA (IX+d)	DDCB d 2E
SRA (IX+d),A**	DDCB d 2F
SRA (IX+d),B**	DDCB d 28
SRA (IX+d),C**	DDCB d 29
SRA (IX+d),D**	DDCB d 2A
SRA (IX+d),E**	DDCB d 2B
SRA (IX+d),H**	DDCB d 2C
SRA (IX+d),L**	DDCB d 2D
SRA (IY+d)	FDCB d 2E
SRA (IY+d),A**	FDCB d 2F
SRA (IY+d),B**	FDCB d 28
SRA (IY+d),C**	FDCB d 29
SRA (IY+d),D**	FDCB d 2A
SRA (IY+d),E**	FDCB d 2B
SRA (IY+d),H**	FDCB d 2C
SRA (IY+d),L**	FDCB d 2D
SRL A	CB3F
SRL B	CB38
SRL C	CB39
SRL D	CB3A
SRL E	CB3B
SRL H	CB3C
SRL L	CB3D
SRL (HL)	CB3E
SRL (IX+d)	DDCB d 3E
SRL (IX+d),A**	DDCB d 3F
SRL (IX+d),B**	DDCB d 38
SRL (IX+d),C**	DDCB d 39
SRL (IX+d),D**	DDCB d 3A
SRL (IX+d),E**	DDCB d 3B
SRL (IX+d),H**	DDCB d 3C
SRL (IX+d),L**	DDCB d 3D
SRL (IY+d)	FDCB d 3E
SRL (IY+d),A**	FDCB d 3F
SRL (IY+d),B**	FDCB d 38
SRL (IY+d),C**	FDCB d 39
SRL (IY+d),D**	FDCB d 3A
SRL (IY+d),E**	FDCB d 3B
SRL (IY+d),H**	FDCB d 3C
SRL (IY+d),L**	FDCB d 3D
SUB A	97
SUB B	90
SUB C	91
SUB D	92
SUB E	93
SUB H	94
SUB L	95
SUB n	D6 n
SUB (HL)	96
SUB (IX+d)	DD96 d
SUB (IY+d)	FD96 d
SUB IXH**	DD94
SUB IXL**	DD95
SUB IYH**	FD94
SUB IYL**	FD95
SWAPNIB^{ZX}	ED23
TEST n^{ZX}	ED27 n

XOR A	AF	XOR H	AC	XOR (IY+d)	FDAE d
XOR B	A8	XOR L	AD	XOR IXH**	DDAC
XOR C	A9	XOR n	EE n	XOR IXL**	DDAD
XOR D	AA	XOR (HL)	AE	XOR IYH**	FDAC
XOR E	AB	XOR (IX+d)	DDAE d	XOR IYL**	FDAD

This page intentionally left empty

Appendix B

Instructions Sorted by Opcode

Instructions marked with ** are undocumented.
Instructions marked with $^{\text{ZX}}$ are ZX Spectrum Next extended.

00	NOP
01 m n	LD BC,nm
02	LD (BC),A
03	INC BC
04	INC B
05	DEC B
06 n	LD B,n
07	RLCA
08	EX AF,AF’
09	ADD HL,BC
0A	LD A,(BC)
0B	DEC BC
0C	INC C
0D	DEC C
0E n	LD C,n
0F	RRCA
10 e	DJNZ (PC+e)
11 m n	LD DE,nm
12	LD (DE),A
13	INC DE
14	INC D
15	DEC D
16 n	LD D,n
17	RLA
18 e	JR e
19	ADD HL,DE
1A	LD A,(DE)
1B	DEC DE
1C	INC E
1D	DEC E
1E n	LD E,n
1F	RRA
20 e	JR NZ,e
21 m n	LD HL,nm
22 m n	LD (nm),HL
23	INC HL
24	INC H
25	DEC H
26 n	LD H,n
27	DAA
28 e	JR Z,e
29	ADD HL,HL
2A m n	LD HL,(nm)
2B	DEC HL
2C	INC L
2D	DEC L
2E n	LD L,n
2F	CPL
30 e	JR NC,e
31 m n	LD SP,nm
32 m n	LD (nm),A
33	INC SP
34	INC (HL)
35	DEC (HL)
36 n	LD (HL),n
37	SCF
38 e	JR C,e
39	ADD HL,SP
3A m n	LD A,(nm)
3B	DEC SP
3C	INC A
3D	DEC A
3E n	LD A,n
3F	CCF
40	LD B,B
41	LD B,C
42	LD B,D
43	LD B,E
44	LD B,H
45	LD B,L
46	LD B,(HL)
47	LD B,A
48	LD C,B
49	LD C,C
4A	LD C,D
4B	LD C,E
4C	LD C,H
4D	LD C,L
4E	LD C,(HL)
4F	LD C,A
50	LD D,B
51	LD D,C
52	LD D,D
53	LD D,E
54	LD D,H
55	LD D,L
56	LD D,(HL)
57	LD D,A
58	LD E,B
59	LD E,C
5A	LD E,D
5B	LD E,E
5C	LD E,H
5D	LD E,L
5E	LD E,(HL)
5F	LD E,A
60	LD H,B
61	LD H,C
62	LD H,D
63	LD H,E
64	LD H,H
65	LD H,L
66	LD H,(HL)
67	LD H,A
68	LD L,B
69	LD L,C
6A	LD L,D
6B	LD L,E

6C	LD L,H
6D	LD L,L
6E	LD L,(HL)
6F	LD L,A
70	LD (HL),B
71	LD (HL),C
72	LD (HL),D
73	LD (HL),E
74	LD (HL),H
75	LD (HL),L
76	HALT
77	LD (HL),A
78	LD A,B
79	LD A,C
7A	LD A,D
7B	LD A,E
7C	LD A,H
7D	LD A,L
7E	LD A,(HL)
7F	LD A,A
80	ADD A,B
81	ADD A,C
82	ADD A,D
83	ADD A,E
84	ADD A,H
85	ADD A,L
86	ADD A,(HL)
87	ADD A,A
88	ADC A,B
89	ADC A,C
8A	ADC A,D
8B	ADC A,E
8C	ADC A,H
8D	ADC A,L
8E	ADC A,(HL)
8F	ADC A,A
90	SUB B
91	SUB C
92	SUB D
93	SUB E
94	SUB H
95	SUB L
96	SUB (HL)
97	SUB A
98	SBC A,B
99	SBC A,C
9A	SBC A,D
9B	SBC A,E
9C	SBC A,H
9D	SBC A,L
9E	SBC A,(HL)
9F	SBC A,A
A0	AND B
A1	AND C
A2	AND D
A3	AND E
A4	AND H
A5	AND L
A6	AND (HL)
A7	AND A
A8	XOR B
A9	XOR C
AA	XOR D
AB	XOR E
AC	XOR H
AD	XOR L
AE	XOR (HL)
AF	XOR A
B0	OR B
B1	OR C
B2	OR D
B3	OR E
B4	OR H
B5	OR L
B6	OR (HL)
B7	OR A
B8	CP B
B9	CP C
BA	CP D
BB	CP E
BC	CP H
BD	CP L
BE	CP (HL)
BF	CP A
C0	RET NZ
C1	POP BC
C2 m n	JP NZ,nm
C3 m n	JP nm
C4 m n	CALL NZ,nm
C5	PUSH BC
C6 n	ADD A,n
C7	RST 0H
C8	RET Z
C9	RET
CA m n	JP Z,nm
CB00	RLC B
CB01	RLC C
CB02	RLC D
CB03	RLC E
CB04	RLC H
CB05	RLC L
CB06	RLC (HL)
CB07	RLC A
CB08	RRC B
CB09	RRC C
CB0A	RRC D
CB0B	RRC E
CB0C	RRC H
CB0D	RRC L
CB0E	RRC (HL)
CB0F	RRC A
CB10	RL B
CB11	RL C
CB12	RL D
CB13	RL E
CB14	RL H
CB15	RL L
CB16	RL (HL)
CB17	RL A
CB18	RR B
CB19	RR C
CB1A	RR D
CB1B	RR E
CB1C	RR H
CB1D	RR L
CB1E	RR (HL)
CB1F	RR A
CB20	SLA B
CB21	SLA C
CB22	SLA D
CB23	SLA E
CB24	SLA H
CB25	SLA L
CB26	SLA (HL)
CB27	SLA A
CB28	SRA B
CB29	SRA C
CB2A	SRA D
CB2B	SRA E
CB2C	SRA H
CB2D	SRA L
CB2E	SRA (HL)
CB2F	SRA A
CB30	SLI B**
CB31	SLI C**
CB32	SLI D**
CB33	SLI E**
CB34	SLI H**
CB35	SLI L**
CB36	SLI (HL)**
CB37	SLI A**
CB38	SRL B
CB39	SRL C
CB3A	SRL D
CB3B	SRL E
CB3C	SRL H
CB3D	SRL L
CB3E	SRL (HL)
CB3F	SRL A
CB40	BIT 0,B
CB41	BIT 0,C
CB42	BIT 0,D
CB43	BIT 0,E
CB44	BIT 0,H
CB45	BIT 0,L
CB46	BIT 0,(HL)
CB47	BIT 0,A
CB48	BIT 1,B
CB49	BIT 1,C
CB4A	BIT 1,D
CB4B	BIT 1,E

CB4C	BIT 1,H
CB4D	BIT 1,L
CB4E	BIT 1,(HL)
CB4F	BIT 1,A
CB50	BIT 2,B
CB51	BIT 2,C
CB52	BIT 2,D
CB53	BIT 2,E
CB54	BIT 2,H
CB55	BIT 2,L
CB56	BIT 2,(HL)
CB57	BIT 2,A
CB58	BIT 3,B
CB59	BIT 3,C
CB5A	BIT 3,D
CB5B	BIT 3,E
CB5C	BIT 3,H
CB5D	BIT 3,L
CB5E	BIT 3,(HL)
CB5F	BIT 3,A
CB60	BIT 4,B
CB61	BIT 4,C
CB62	BIT 4,D
CB63	BIT 4,E
CB64	BIT 4,H
CB65	BIT 4,L
CB66	BIT 4,(HL)
CB67	BIT 4,A
CB68	BIT 5,B
CB69	BIT 5,C
CB6A	BIT 5,D
CB6B	BIT 5,E
CB6C	BIT 5,H
CB6D	BIT 5,L
CB6E	BIT 5,(HL)
CB6F	BIT 5,A
CB70	BIT 6,B
CB71	BIT 6,C
CB72	BIT 6,D
CB73	BIT 6,E
CB74	BIT 6,H
CB75	BIT 6,L
CB76	BIT 6,(HL)
CB77	BIT 6,A
CB78	BIT 7,B
CB79	BIT 7,C
CB7A	BIT 7,D
CB7B	BIT 7,E
CB7C	BIT 7,H
CB7D	BIT 7,L
CB7E	BIT 7,(HL)
CB7F	BIT 7,A
CB80	RES 0,B
CB81	RES 0,C
CB82	RES 0,D
CB83	RES 0,E
CB84	RES 0,H
CB85	RES 0,L
CB86	RES 0,(HL)
CB87	RES 0,A
CB88	RES 1,B
CB89	RES 1,C
CB8A	RES 1,D
CB8B	RES 1,E
CB8C	RES 1,H
CB8D	RES 1,L
CB8E	RES 1,(HL)
CB8F	RES 1,A
CB90	RES 2,B
CB91	RES 2,C
CB92	RES 2,D
CB93	RES 2,E
CB94	RES 2,H
CB95	RES 2,L
CB96	RES 2,(HL)
CB97	RES 2,A
CB98	RES 3,B
CB99	RES 3,C
CB9A	RES 3,D
CB9B	RES 3,E
CB9C	RES 3,H
CB9D	RES 3,L
CB9E	RES 3,(HL)
CB9F	RES 3,A
CBA0	RES 4,B
CBA1	RES 4,C
CBA2	RES 4,D
CBA3	RES 4,E
CBA4	RES 4,H
CBA5	RES 4,L
CBA6	RES 4,(HL)
CBA7	RES 4,A
CBA8	RES 5,B
CBA9	RES 5,C
CBAA	RES 5,D
CBAB	RES 5,E
CBAC	RES 5,H
CBAD	RES 5,L
CBAE	RES 5,(HL)
CBAF	RES 5,A
CBB0	RES 6,B
CBB1	RES 6,C
CBB2	RES 6,D
CBB3	RES 6,E
CBB4	RES 6,H
CBB5	RES 6,L
CBB6	RES 6,(HL)
CBB7	RES 6,A
CBB8	RES 7,B
CBB9	RES 7,C
CBBA	RES 7,D
CBBB	RES 7,E
CBBC	RES 7,H
CBBD	RES 7,L
CBBE	RES 7,(HL)
CBBF	RES 7,A
CBC0	SET 0,B
CBC1	SET 0,C
CBC2	SET 0,D
CBC3	SET 0,E
CBC4	SET 0,H
CBC5	SET 0,L
CBC6	SET 0,(HL)
CBC7	SET 0,A
CBC8	SET 1,B
CBC9	SET 1,C
CBCA	SET 1,D
CBCB	SET 1,E
CBCC	SET 1,H
CBCD	SET 1,L
CBCE	SET 1,(HL)
CBCF	SET 1,A
CBD0	SET 2,B
CBD1	SET 2,C
CBD2	SET 2,D
CBD3	SET 2,E
CBD4	SET 2,H
CBD5	SET 2,L
CBD6	SET 2,(HL)
CBD7	SET 2,A
CBD8	SET 3,B
CBD9	SET 3,C
CBDA	SET 3,D
CBDB	SET 3,E
CBDC	SET 3,H
CBDD	SET 3,L
CBDE	SET 3,(HL)
CBDF	SET 3,A
CBE0	SET 4,B
CBE1	SET 4,C
CBE2	SET 4,D
CBE3	SET 4,E
CBE4	SET 4,H
CBE5	SET 4,L
CBE6	SET 4,(HL)
CBE7	SET 4,A
CBE8	SET 5,B
CBE9	SET 5,C
CBEA	SET 5,D
CBEB	SET 5,E
CBEC	SET 5,H
CBED	SET 5,L
CBEE	SET 5,(HL)
CBEF	SET 5,A
CBF0	SET 6,B
CBF1	SET 6,C
CBF2	SET 6,D
CBF3	SET 6,E
CBF4	SET 6,H
CBF5	SET 6,L
CBF6	SET 6,(HL)

CBF7	SET 6,A
CBF8	SET 7,B
CBF9	SET 7,C
CBFA	SET 7,D
CBFB	SET 7,E
CBFC	SET 7,H
CBFD	SET 7,L
CBFE	SET 7,(HL)
CBFF	SET 7,A
CC m n	CALL Z,nm
CD m n	CALL nm
CE n	ADC A,n
CF	RST 8H
D0	RET NC
D1	POP DE
D2 m n	JP NC,nm
D3 n	OUT (n),A
D4 m n	CALL NC,nm
D5	PUSH DE
D6 n	SUB n
D7	RST 10H
D8	RET C
D9	EXX
DA m n	JP C,nm
DB n	IN A,(n)
DC m n	CALL C,nm
DD09	ADD IX,BC
DD19	ADD IX,DE
DD21 m n	LD IX,nm
DD22 m n	LD (nm),IX
DD23	INC IX
DD24	INC IXH**
DD25	DEC IXH**
DD26 n	LD IXH,n**
DD29	ADD IX,IX
DD2A m n	LD IX,(nm)
DD2B	DEC IX
DD2C	INC IXL**
DD2D	DEC IXL**
DD2E n	LD IXL,n**
DD34 d	INC (IX+d)
DD35 d	DEC (IX+d)
DD36 d n	LD (IX+d),n
DD39	ADD IX,SP
DD44	LD B,IXH**
DD45	LD B,IXL**
DD46 d	LD B,(IX+d)
DD4C	LD C,IXH**
DD4D	LD C,IXL**
DD4E d	LD C,(IX+d)
DD54	LD D,IXH**
DD55	LD D,IXL**
DD56 d	LD D,(IX+d)
DD5C	LD E,IXH**
DD5D	LD E,IXL**
DD5E d	LD E,(IX+d)
DD60	LD IXH,B**
DD61	LD IXH,C**
DD62	LD IXH,D**
DD63	LD IXH,E**
DD64	LD IXH,IXH**
DD65	LD IXH,IXL**
DD66 d	LD H,(IX+d)
DD67	LD IXH,A**
DD68	LD IXL,B**
DD69	LD IXL,C**
DD6A	LD IXL,D**
DD6B	LD IXL,E**
DD6C	LD IXL,IXH**
DD6D	LD IXL,IXL**
DD6E d	LD L,(IX+d)
DD6F	LD IXL,A**
DD70 d	LD (IX+d),B
DD71 d	LD (IX+d),C
DD72 d	LD (IX+d),D
DD73 d	LD (IX+d),E
DD74 d	LD (IX+d),H
DD75 d	LD (IX+d),L
DD77 d	LD (IX+d),A
DD7C	LD A,IXH**
DD7D	LD A,IXL**
DD7E d	LD A,(IX+d)
DD84	ADD A,IXH**
DD85	ADD A,IXL**
DD86 d	ADD A,(IX+d)
DD8C	ADC A,IXH**
DD8D	ADC A,IXL**
DD8E d	ADC A,(IX+d)
DD94	SUB IXH**
DD95	SUB IXL**
DD96 d	SUB (IX+d)
DD9C	SBC A,IXH**
DD9D	SBC A,IXL**
DD9E d	SBC A,(IX+d)
DDA4	AND IXH**
DDA5	AND IXL**
DDA6 d	AND (IX+d)
DDAC	XOR IXH**
DDAD	XOR IXL**
DDAE d	XOR (IX+d)
DDB4	OR IXH**
DDB5	OR IXL**
DDB6 d	OR (IX+d)
DDBC	CP IXH**
DDBD	CP IXL**
DDBE d	CP (IX+d)
DDCB d 00	RLC (IX+d),B**
DDCB d 01	RLC (IX+d),C**
DDCB d 02	RLC (IX+d),D**
DDCB d 03	RLC (IX+d),E**
DDCB d 04	RLC (IX+d),H**
DDCB d 05	RLC (IX+d),L**
DDCB d 06	RLC (IX+d)
DDCB d 07	RLC (IX+d),A**
DDCB d 08	RRC (IX+d),B**
DDCB d 09	RRC (IX+d),C**
DDCB d 0A	RRC (IX+d),D**
DDCB d 0B	RRC (IX+d),E**
DDCB d 0C	RRC (IX+d),H**
DDCB d 0D	RRC (IX+d),L**
DDCB d 0E	RRC (IX+d)
DDCB d 0F	RRC (IX+d),A**
DDCB d 10	RL (IX+d),B**
DDCB d 11	RL (IX+d),C**
DDCB d 12	RL (IX+d),D**
DDCB d 13	RL (IX+d),E**
DDCB d 14	RL (IX+d),H**
DDCB d 15	RL (IX+d),L**
DDCB d 16	RL (IX+d)
DDCB d 17	RL (IX+d),A**
DDCB d 18	RR (IX+d),B**
DDCB d 19	RR (IX+d),C**
DDCB d 1A	RR (IX+d),D**
DDCB d 1B	RR (IX+d),E**
DDCB d 1C	RR (IX+d),H**
DDCB d 1D	RR (IX+d),L**
DDCB d 1E	RR (IX+d)
DDCB d 1F	RR (IX+d),A**
DDCB d 20	SLA (IX+d),B**
DDCB d 21	SLA (IX+d),C**
DDCB d 22	SLA (IX+d),D**
DDCB d 23	SLA (IX+d),E**
DDCB d 24	SLA (IX+d),H**
DDCB d 25	SLA (IX+d),L**
DDCB d 26	SLA (IX+d)
DDCB d 27	SLA (IX+d),A**
DDCB d 28	SRA (IX+d),B**
DDCB d 29	SRA (IX+d),C**
DDCB d 2A	SRA (IX+d),D**
DDCB d 2B	SRA (IX+d),E**
DDCB d 2C	SRA (IX+d),H**
DDCB d 2D	SRA (IX+d),L**
DDCB d 2E	SRA (IX+d)
DDCB d 2F	SRA (IX+d),A**
DDCB d 30	SLI (IX+d),B**
DDCB d 31	SLI (IX+d),C**
DDCB d 32	SLI (IX+d),D**
DDCB d 33	SLI (IX+d),E**
DDCB d 34	SLI (IX+d),H**
DDCB d 35	SLI (IX+d),L**
DDCB d 36	SLI (IX+d)**
DDCB d 37	SLI (IX+d),A**
DDCB d 38	SRL (IX+d),B**
DDCB d 39	SRL (IX+d),C**
DDCB d 3A	SRL (IX+d),D**
DDCB d 3B	SRL (IX+d),E**

DDCB d 3C	SRL (IX+d),H**
DDCB d 3D	SRL (IX+d),L**
DDCB d 3E	SRL (IX+d)
DDCB d 3F	SRL (IX+d),A**
DDCB d 40	BIT 0,(IX+d)**
DDCB d 41	BIT 0,(IX+d)**
DDCB d 42	BIT 0,(IX+d)**
DDCB d 43	BIT 0,(IX+d)**
DDCB d 44	BIT 0,(IX+d)**
DDCB d 45	BIT 0,(IX+d)**
DDCB d 46	BIT 0,(IX+d)
DDCB d 47	BIT 0,(IX+d)**
DDCB d 48	BIT 1,(IX+d)**
DDCB d 49	BIT 1,(IX+d)**
DDCB d 4A	BIT 1,(IX+d)**
DDCB d 4B	BIT 1,(IX+d)**
DDCB d 4C	BIT 1,(IX+d)**
DDCB d 4D	BIT 1,(IX+d)**
DDCB d 4E	BIT 1,(IX+d)
DDCB d 4F	BIT 1,(IX+d)**
DDCB d 50	BIT 2,(IX+d)**
DDCB d 51	BIT 2,(IX+d)**
DDCB d 52	BIT 2,(IX+d)**
DDCB d 53	BIT 2,(IX+d)**
DDCB d 54	BIT 2,(IX+d)**
DDCB d 55	BIT 2,(IX+d)**
DDCB d 56	BIT 2,(IX+d)
DDCB d 57	BIT 2,(IX+d)**
DDCB d 58	BIT 3,(IX+d)**
DDCB d 59	BIT 3,(IX+d)**
DDCB d 5A	BIT 3,(IX+d)**
DDCB d 5B	BIT 3,(IX+d)**
DDCB d 5C	BIT 3,(IX+d)**
DDCB d 5D	BIT 3,(IX+d)**
DDCB d 5E	BIT 3,(IX+d)
DDCB d 5F	BIT 3,(IX+d)**
DDCB d 60	BIT 4,(IX+d)**
DDCB d 61	BIT 4,(IX+d)**
DDCB d 62	BIT 4,(IX+d)**
DDCB d 63	BIT 4,(IX+d)**
DDCB d 64	BIT 4,(IX+d)**
DDCB d 65	BIT 4,(IX+d)**
DDCB d 66	BIT 4,(IX+d)
DDCB d 67	BIT 4,(IX+d)**
DDCB d 68	BIT 5,(IX+d)**
DDCB d 69	BIT 5,(IX+d)**
DDCB d 6A	BIT 5,(IX+d)**
DDCB d 6B	BIT 5,(IX+d)**
DDCB d 6C	BIT 5,(IX+d)**
DDCB d 6D	BIT 5,(IX+d)**
DDCB d 6E	BIT 5,(IX+d)
DDCB d 6F	BIT 5,(IX+d)**
DDCB d 70	BIT 6,(IX+d)**
DDCB d 71	BIT 6,(IX+d)**
DDCB d 72	BIT 6,(IX+d)**
DDCB d 73	BIT 6,(IX+d)**
DDCB d 74	BIT 6,(IX+d)**
DDCB d 75	BIT 6,(IX+d)**
DDCB d 76	BIT 6,(IX+d)
DDCB d 77	BIT 6,(IX+d)**
DDCB d 78	BIT 7,(IX+d)**
DDCB d 79	BIT 7,(IX+d)**
DDCB d 7A	BIT 7,(IX+d)**
DDCB d 7B	BIT 7,(IX+d)**
DDCB d 7C	BIT 7,(IX+d)**
DDCB d 7D	BIT 7,(IX+d)**
DDCB d 7E	BIT 7,(IX+d)
DDCB d 7F	BIT 7,(IX+d)**
DDCB d 80	RES 0,(IX+d),B**
DDCB d 81	RES 0,(IX+d),C**
DDCB d 82	RES 0,(IX+d),D**
DDCB d 83	RES 0,(IX+d),E**
DDCB d 84	RES 0,(IX+d),H**
DDCB d 85	RES 0,(IX+d),L**
DDCB d 86	RES 0,(IX+d)
DDCB d 87	RES 0,(IX+d),A**
DDCB d 88	RES 1,(IX+d),B**
DDCB d 89	RES 1,(IX+d),C**
DDCB d 8A	RES 1,(IX+d),D**
DDCB d 8B	RES 1,(IX+d),E**
DDCB d 8C	RES 1,(IX+d),H**
DDCB d 8D	RES 1,(IX+d),L**
DDCB d 8E	RES 1,(IX+d)
DDCB d 8F	RES 1,(IX+d),A**
DDCB d 90	RES 2,(IX+d),B**
DDCB d 91	RES 2,(IX+d),C**
DDCB d 92	RES 2,(IX+d),D**
DDCB d 93	RES 2,(IX+d),E**
DDCB d 94	RES 2,(IX+d),H**
DDCB d 95	RES 2,(IX+d),L**
DDCB d 96	RES 2,(IX+d)
DDCB d 97	RES 2,(IX+d),A**
DDCB d 98	RES 3,(IX+d),B**
DDCB d 99	RES 3,(IX+d),C**
DDCB d 9A	RES 3,(IX+d),D**
DDCB d 9B	RES 3,(IX+d),E**
DDCB d 9C	RES 3,(IX+d),H**
DDCB d 9D	RES 3,(IX+d),L**
DDCB d 9E	RES 3,(IX+d)
DDCB d 9F	RES 3,(IX+d),A**
DDCB d A0	RES 4,(IX+d),B**
DDCB d A1	RES 4,(IX+d),C**
DDCB d A2	RES 4,(IX+d),D**
DDCB d A3	RES 4,(IX+d),E**
DDCB d A4	RES 4,(IX+d),H**
DDCB d A5	RES 4,(IX+d),L**
DDCB d A6	RES 4,(IX+d)
DDCB d A7	RES 4,(IX+d),A**
DDCB d A8	RES 5,(IX+d),B**
DDCB d A9	RES 5,(IX+d),C**
DDCB d AA	RES 5,(IX+d),D**
DDCB d AB	RES 5,(IX+d),E**
DDCB d AC	RES 5,(IX+d),H**
DDCB d AD	RES 5,(IX+d),L**
DDCB d AE	RES 5,(IX+d)
DDCB d AF	RES 5,(IX+d),A**
DDCB d B0	RES 6,(IX+d),B**
DDCB d B1	RES 6,(IX+d),C**
DDCB d B2	RES 6,(IX+d),D**
DDCB d B3	RES 6,(IX+d),E**
DDCB d B4	RES 6,(IX+d),H**
DDCB d B5	RES 6,(IX+d),L**
DDCB d B6	RES 6,(IX+d)
DDCB d B7	RES 6,(IX+d),A**
DDCB d B8	RES 7,(IX+d),B**
DDCB d B9	RES 7,(IX+d),C**
DDCB d BA	RES 7,(IX+d),D**
DDCB d BB	RES 7,(IX+d),E**
DDCB d BC	RES 7,(IX+d),H**
DDCB d BD	RES 7,(IX+d),L**
DDCB d BE	RES 7,(IX+d)
DDCB d BF	RES 7,(IX+d),A**
DDCB d C0	SET 0,(IX+d),B**
DDCB d C1	SET 0,(IX+d),C**
DDCB d C2	SET 0,(IX+d),D**
DDCB d C3	SET 0,(IX+d),E**
DDCB d C4	SET 0,(IX+d),H**
DDCB d C5	SET 0,(IX+d),L**
DDCB d C6	SET 0,(IX+d)
DDCB d C7	SET 0,(IX+d),A**
DDCB d C8	SET 1,(IX+d),B**
DDCB d C9	SET 1,(IX+d),C**
DDCB d CA	SET 1,(IX+d),D**
DDCB d CB	SET 1,(IX+d),E**
DDCB d CC	SET 1,(IX+d),H**
DDCB d CD	SET 1,(IX+d),L**
DDCB d CE	SET 1,(IX+d)
DDCB d CF	SET 1,(IX+d),A**
DDCB d D0	SET 2,(IX+d),B**
DDCB d D1	SET 2,(IX+d),C**
DDCB d D2	SET 2,(IX+d),D**
DDCB d D3	SET 2,(IX+d),E**
DDCB d D4	SET 2,(IX+d),H**
DDCB d D5	SET 2,(IX+d),L**
DDCB d D6	SET 2,(IX+d)
DDCB d D7	SET 2,(IX+d),A**
DDCB d D8	SET 3,(IX+d),B**
DDCB d D9	SET 3,(IX+d),C**
DDCB d DA	SET 3,(IX+d),D**
DDCB d DB	SET 3,(IX+d),E**
DDCB d DC	SET 3,(IX+d),H**
DDCB d DD	SET 3,(IX+d),L**
DDCB d DE	SET 3,(IX+d)
DDCB d DF	SET 3,(IX+d),A**
DDCB d E0	SET 4,(IX+d),B**

DDCB d E1	SET 4,(IX+d),C**
DDCB d E2	SET 4,(IX+d),D**
DDCB d E3	SET 4,(IX+d),E**
DDCB d E4	SET 4,(IX+d),H**
DDCB d E5	SET 4,(IX+d),L**
DDCB d E6	SET 4,(IX+d)
DDCB d E7	SET 4,(IX+d),A**
DDCB d E8	SET 5,(IX+d),B**
DDCB d E9	SET 5,(IX+d),C**
DDCB d EA	SET 5,(IX+d),D**
DDCB d EB	SET 5,(IX+d),E**
DDCB d EC	SET 5,(IX+d),H**
DDCB d ED	SET 5,(IX+d),L**
DDCB d EE	SET 5,(IX+d)
DDCB d EF	SET 5,(IX+d),A**
DDCB d F0	SET 6,(IX+d),B**
DDCB d F1	SET 6,(IX+d),C**
DDCB d F2	SET 6,(IX+d),D**
DDCB d F3	SET 6,(IX+d),E**
DDCB d F4	SET 6,(IX+d),H**
DDCB d F5	SET 6,(IX+d),L**
DDCB d F6	SET 6,(IX+d)
DDCB d F7	SET 6,(IX+d),A**
DDCB d F8	SET 7,(IX+d),B**
DDCB d F9	SET 7,(IX+d),C**
DDCB d FA	SET 7,(IX+d),D**
DDCB d FB	SET 7,(IX+d),E**
DDCB d FC	SET 7,(IX+d),H**
DDCB d FD	SET 7,(IX+d),L**
DDCB d FE	SET 7,(IX+d)
DDCB d FF	SET 7,(IX+d),A**
DDE1	POP IX
DDE3	EX (SP),IX
DDE5	PUSH IX
DDE9	JP (IX)
DDF9	LD SP,IX
DE n	SBC A,n
DF	RST 18H
E0	RET PO
E1	POP HL
E2 m n	JP PO,nm
E3	EX (SP),HL
E4 m n	CALL PO,nm
E5	PUSH HL
E6 n	AND n
E7	RST 20H
E8	RET PE
E9	JP (HL)
EA m n	JP PE,nm
EB	EX DE,HL
EC m n	CALL PE,nm
ED23	SWAPNIB^{ZX}
ED24	MIRROR A^{ZX}
ED27 n	TEST n^{ZX}
ED28	BSLA DE,B^{ZX}
ED28	BSLA DE,B^{ZX}
ED29	BSRA DE,B^{ZX}
ED2A	BSRL DE,B^{ZX}
ED2B	BSRF DE,B^{ZX}
ED2C	BRLC DE,B^{ZX}
ED30	MUL D,E^{ZX}
ED31	ADD HL,A^{ZX}
ED32	ADD DE,A^{ZX}
ED33	ADD BC,A^{ZX}
ED34 m n	ADD HL,nm^{ZX}
ED35 m n	ADD DE,nm^{ZX}
ED36 m n	ADD BC,nm^{ZX}
ED40	IN B,(C)
ED41	OUT (C),B
ED42	SBC HL,BC
ED43 m n	LD (nm),BC
ED44	NEG
ED45	RETN
ED46	IM 0
ED47	LD I,A
ED48	IN C,(C)
ED49	OUT (C),C
ED4A	ADC HL,BC
ED4B m n	LD BC,(nm)
ED4C	NEG**
ED4D	RETI
ED4E	IM 0**
ED4F	LD R,A
ED50	IN D,(C)
ED51	OUT (C),D
ED52	SBC HL,DE
ED53 m n	LD (nm),DE
ED54	NEG**
ED55	RETN**
ED56	IM 1
ED57	LD A,I
ED58	IN E,(C)
ED59	OUT (C),E
ED5A	ADC HL,DE
ED5B m n	LD DE,(nm)
ED5C	NEG**
ED5D	RETN**
ED5E	IM 2
ED5F	LD A,R
ED60	IN H,(C)
ED61	OUT (C),H
ED62	SBC HL,HL
ED63 m n	LD (nm),HL
ED64	NEG**
ED65	RETN**
ED66	IM 0**
ED67	RRD
ED68	IN L,(C)
ED69	OUT (C),L
ED6A	ADC HL,HL
ED6B m n	LD HL,(nm)
ED6C	NEG**
ED6D	RETN**
ED6E	IM 0**
ED6F	RLD
ED70	IN F,(C)**
ED70	IN (C)**
ED71	OUT (C),0**
ED72	SBC HL,SP
ED73 m n	LD (nm),SP
ED74	NEG**
ED75	RETN**
ED76	IM 1**
ED78	IN A,(C)
ED79	OUT (C),A
ED7A	ADC HL,SP
ED7B m n	LD SP,(nm)
ED7C	NEG**
ED7D	RETN**
ED7E	IM 2**
ED8A n m	PUSH nm^{ZX}
ED90	OUTINB^{ZX}
ED91 r n	NEXTREG r,n^{ZX}
ED92 n	NEXTREG r,A^{ZX}
ED93	PIXELDN^{ZX}
ED94	PIXELAD^{ZX}
ED95	SETAE^{ZX}
ED97	JP (C)^{ZX}
EDA0	LDI
EDA1	CPI
EDA2	INI
EDA3	OUTI
EDA4	LDIX^{ZX}
EDA5	LDWS^{ZX}
EDAC	LDDX^{ZX}
EDA8	LDD
EDA9	CPD
EDAA	IND
EDAB	OUTD
EDB0	LDIR
EDB1	CPIR
EDB2	INIR
EDB3	OTIR
EDB4	LDIRX^{ZX}
EDB7	LDPIRX^{ZX}
EDBC	LDDRX^{ZX}
EDB8	LDDR
EDB9	CPDR
EDBA	INDR
EDBB	OTDR
EE n	XOR n
EF	RST 28H
F0	RET P
F1	POP AF
F2 m n	JP P,nm
F3	DI
F4 m n	CALL P,nm
F5	PUSH AF

F6 n	OR n
F7	RST 30H
F8	RET M
F9	LD SP,HL
FA m n	JP M,nm
FB	EI
FC m n	CALL M,nm
FD09	ADD IY,BC
FD19	ADD IY,DE
FD21 m n	LD IY,nm
FD22 m n	LD (nm),IY
FD23	INC IY
FD24	INC IYH**
FD25	DEC IYH**
FD26 n	LD IYH,n**
FD29	ADD IY,IY
FD2A m n	LD IY,(nm)
FD2B	DEC IY
FD2C	INC IYL**
FD2D	DEC IYL**
FD2E n	LD IYL,n**
FD34 d	INC (IY+d)
FD35 d	DEC (IY+d)
FD36 d n	LD (IY+d),n
FD39	ADD IY,SP
FD44	LD B,IYH**
FD45	LD B,IYL**
FD46 d	LD B,(IY+d)
FD4C	LD C,IYH**
FD4D	LD C,IYL**
FD4E d	LD C,(IY+d)
FD54	LD D,IYH**
FD55	LD D,IYL**
FD56 d	LD D,(IY+d)
FD5C	LD E,IYH**
FD5D	LD E,IYL**
FD5E d	LD E,(IY+d)
FD60	LD IYH,B**
FD61	LD IYH,C**
FD62	LD IYH,D**
FD63	LD IYH,E**
FD64	LD IYH,IYH**
FD65	LD IYH,IYL**
FD66 d	LD H,(IY+d)
FD67	LD IYH,A**
FD68	LD IYL,B**
FD69	LD IYL,C**
FD6A	LD IYL,D**
FD6B	LD IYL,E**
FD6C	LD IYL,IYH**
FD6D	LD IYL,IYL**
FD6E d	LD L,(IY+d)
FD6F	LD IYL,A**
FD70 d	LD (IY+d),B
FD71 d	LD (IY+d),C
FD72 d	LD (IY+d),D
FD73 d	LD (IY+d),E
FD74 d	LD (IY+d),H
FD75 d	LD (IY+d),L
FD77 d	LD (IY+d),A
FD7C	LD A,IYH**
FD7D	LD A,IYL**
FD7E d	LD A,(IY+d)
FD84	ADD A,IYH**
FD85	ADD A,IYL**
FD86 d	ADD A,(IY+d)
FD8C	ADC A,IYH**
FD8D	ADC A,IYL**
FD8E d	ADC A,(IY+d)
FD94	SUB IYH**
FD95	SUB IYL**
FD96 d	SUB (IY+d)
FD9C	SBC A,IYH**
FD9D	SBC A,IYL**
FD9E d	SBC A,(IY+d)
FDA4	AND IYH**
FDA5	AND IYL**
FDA6 d	AND (IY+d)
FDAC	XOR IYH**
FDAD	XOR IYL**
FDAE d	XOR (IY+d)
FDB4	OR IYH**
FDB5	OR IYL**
FDB6 d	OR (IY+d)
FDBC	CP IYH**
FDBD	CP IYL**
FDBE d	CP (IY+d)
FDCB d 00	RLC (IY+d),B**
FDCB d 01	RLC (IY+d),C**
FDCB d 02	RLC (IY+d),D**
FDCB d 03	RLC (IY+d),E**
FDCB d 04	RLC (IY+d),H**
FDCB d 05	RLC (IY+d),L**
FDCB d 06	RLC (IY+d)
FDCB d 07	RLC (IY+d),A**
FDCB d 08	RRC (IY+d),B**
FDCB d 09	RRC (IY+d),C**
FDCB d 0A	RRC (IY+d),D**
FDCB d 0B	RRC (IY+d),E**
FDCB d 0C	RRC (IY+d),H**
FDCB d 0D	RRC (IY+d),L**
FDCB d 0E	RRC (IY+d)
FDCB d 0F	RRC (IY+d),A**
FDCB d 10	RL (IY+d),B**
FDCB d 11	RL (IY+d),C**
FDCB d 12	RL (IY+d),D**
FDCB d 13	RL (IY+d),E**
FDCB d 14	RL (IY+d),H**
FDCB d 15	RL (IY+d),L**
FDCB d 16	RL (IY+d)
FDCB d 17	RL (IY+d),A**
FDCB d 18	RR (IY+d),B**
FDCB d 19	RR (IY+d),C**
FDCB d 1A	RR (IY+d),D**
FDCB d 1B	RR (IY+d),E**
FDCB d 1C	RR (IY+d),H**
FDCB d 1D	RR (IY+d),L**
FDCB d 1E	RR (IY+d)
FDCB d 1F	RR (IY+d),A**
FDCB d 20	SLA (IY+d),B**
FDCB d 21	SLA (IY+d),C**
FDCB d 22	SLA (IY+d),D**
FDCB d 23	SLA (IY+d),E**
FDCB d 24	SLA (IY+d),H**
FDCB d 25	SLA (IY+d),L**
FDCB d 26	SLA (IY+d)
FDCB d 27	SLA (IY+d),A**
FDCB d 28	SRA (IY+d),B**
FDCB d 29	SRA (IY+d),C**
FDCB d 2A	SRA (IY+d),D**
FDCB d 2B	SRA (IY+d),E**
FDCB d 2C	SRA (IY+d),H**
FDCB d 2D	SRA (IY+d),L**
FDCB d 2E	SRA (IY+d)
FDCB d 2F	SRA (IY+d),A**
FDCB d 30	SLI (IY+d),B**
FDCB d 31	SLI (IY+d),C**
FDCB d 32	SLI (IY+d),D**
FDCB d 33	SLI (IY+d),E**
FDCB d 34	SLI (IY+d),H**
FDCB d 35	SLI (IY+d),L**
FDCB d 36	SLI (IY+d)**
FDCB d 37	SLI (IY+d),A**
FDCB d 38	SRL (IY+d),B**
FDCB d 39	SRL (IY+d),C**
FDCB d 3A	SRL (IY+d),D**
FDCB d 3B	SRL (IY+d),E**
FDCB d 3C	SRL (IY+d),H**
FDCB d 3D	SRL (IY+d),L**
FDCB d 3E	SRL (IY+d)
FDCB d 3F	SRL (IY+d),A**
FDCB d 40	BIT 0,(IY+d)**
FDCB d 41	BIT 0,(IY+d)**
FDCB d 42	BIT 0,(IY+d)**
FDCB d 43	BIT 0,(IY+d)**
FDCB d 44	BIT 0,(IY+d)**
FDCB d 45	BIT 0,(IY+d)**
FDCB d 46	BIT 0,(IY+d)
FDCB d 47	BIT 0,(IY+d)**
FDCB d 48	BIT 1,(IY+d)**
FDCB d 49	BIT 1,(IY+d)**
FDCB d 4A	BIT 1,(IY+d)**
FDCB d 4B	BIT 1,(IY+d)**
FDCB d 4C	BIT 1,(IY+d)**
FDCB d 4D	BIT 1,(IY+d)**
FDCB d 4E	BIT 1,(IY+d)

FDCB d 4F	BIT 1,(IY+d)**
FDCB d 50	BIT 2,(IY+d)**
FDCB d 51	BIT 2,(IY+d)**
FDCB d 52	BIT 2,(IY+d)**
FDCB d 53	BIT 2,(IY+d)**
FDCB d 54	BIT 2,(IY+d)**
FDCB d 55	BIT 2,(IY+d)**
FDCB d 56	BIT 2,(IY+d)
FDCB d 57	BIT 2,(IY+d)**
FDCB d 58	BIT 3,(IY+d)**
FDCB d 59	BIT 3,(IY+d)**
FDCB d 5A	BIT 3,(IY+d)**
FDCB d 5B	BIT 3,(IY+d)**
FDCB d 5C	BIT 3,(IY+d)**
FDCB d 5D	BIT 3,(IY+d)**
FDCB d 5E	BIT 3,(IY+d)
FDCB d 5F	BIT 3,(IY+d)**
FDCB d 60	BIT 4,(IY+d)**
FDCB d 61	BIT 4,(IY+d)**
FDCB d 62	BIT 4,(IY+d)**
FDCB d 63	BIT 4,(IY+d)**
FDCB d 64	BIT 4,(IY+d)**
FDCB d 65	BIT 4,(IY+d)**
FDCB d 66	BIT 4,(IY+d)
FDCB d 67	BIT 4,(IY+d)**
FDCB d 68	BIT 5,(IY+d)**
FDCB d 69	BIT 5,(IY+d)**
FDCB d 6A	BIT 5,(IY+d)**
FDCB d 6B	BIT 5,(IY+d)**
FDCB d 6C	BIT 5,(IY+d)**
FDCB d 6D	BIT 5,(IY+d)**
FDCB d 6E	BIT 5,(IY+d)
FDCB d 6F	BIT 5,(IY+d)**
FDCB d 70	BIT 6,(IY+d)**
FDCB d 71	BIT 6,(IY+d)**
FDCB d 72	BIT 6,(IY+d)**
FDCB d 73	BIT 6,(IY+d)**
FDCB d 74	BIT 6,(IY+d)**
FDCB d 75	BIT 6,(IY+d)**
FDCB d 76	BIT 6,(IY+d)
FDCB d 77	BIT 6,(IY+d)**
FDCB d 78	BIT 7,(IY+d)**
FDCB d 79	BIT 7,(IY+d)**
FDCB d 7A	BIT 7,(IY+d)**
FDCB d 7B	BIT 7,(IY+d)**
FDCB d 7C	BIT 7,(IY+d)**
FDCB d 7D	BIT 7,(IY+d)**
FDCB d 7E	BIT 7,(IY+d)
FDCB d 7F	BIT 7,(IY+d)**
FDCB d 80	RES 0,(IY+d),B**
FDCB d 81	RES 0,(IY+d),C**
FDCB d 82	RES 0,(IY+d),D**
FDCB d 83	RES 0,(IY+d),E**
FDCB d 84	RES 0,(IY+d),H**
FDCB d 85	RES 0,(IY+d),L**
FDCB d 86	RES 0,(IY+d)
FDCB d 87	RES 0,(IY+d),A**
FDCB d 88	RES 1,(IY+d),B**
FDCB d 89	RES 1,(IY+d),C**
FDCB d 8A	RES 1,(IY+d),D**
FDCB d 8B	RES 1,(IY+d),E**
FDCB d 8C	RES 1,(IY+d),H**
FDCB d 8D	RES 1,(IY+d),L**
FDCB d 8E	RES 1,(IY+d)
FDCB d 8F	RES 1,(IY+d),A**
FDCB d 90	RES 2,(IY+d),B**
FDCB d 91	RES 2,(IY+d),C**
FDCB d 92	RES 2,(IY+d),D**
FDCB d 93	RES 2,(IY+d),E**
FDCB d 94	RES 2,(IY+d),H**
FDCB d 95	RES 2,(IY+d),L**
FDCB d 96	RES 2,(IY+d)
FDCB d 97	RES 2,(IY+d),A**
FDCB d 98	RES 3,(IY+d),B**
FDCB d 99	RES 3,(IY+d),C**
FDCB d 9A	RES 3,(IY+d),D**
FDCB d 9B	RES 3,(IY+d),E**
FDCB d 9C	RES 3,(IY+d),H**
FDCB d 9D	RES 3,(IY+d),L**
FDCB d 9E	RES 3,(IY+d)
FDCB d 9F	RES 3,(IY+d),A**
FDCB d A0	RES 4,(IY+d),B**
FDCB d A1	RES 4,(IY+d),C**
FDCB d A2	RES 4,(IY+d),D**
FDCB d A3	RES 4,(IY+d),E**
FDCB d A4	RES 4,(IY+d),H**
FDCB d A5	RES 4,(IY+d),L**
FDCB d A6	RES 4,(IY+d)
FDCB d A7	RES 4,(IY+d),A**
FDCB d A8	RES 5,(IY+d),B**
FDCB d A9	RES 5,(IY+d),C**
FDCB d AA	RES 5,(IY+d),D**
FDCB d AB	RES 5,(IY+d),E**
FDCB d AC	RES 5,(IY+d),H**
FDCB d AD	RES 5,(IY+d),L**
FDCB d AE	RES 5,(IY+d)
FDCB d AF	RES 5,(IY+d),A**
FDCB d B0	RES 6,(IY+d),B**
FDCB d B1	RES 6,(IY+d),C**
FDCB d B2	RES 6,(IY+d),D**
FDCB d B3	RES 6,(IY+d),E**
FDCB d B4	RES 6,(IY+d),H**
FDCB d B5	RES 6,(IY+d),L**
FDCB d B6	RES 6,(IY+d)
FDCB d B7	RES 6,(IY+d),A**
FDCB d B8	RES 7,(IY+d),B**
FDCB d B9	RES 7,(IY+d),C**
FDCB d BA	RES 7,(IY+d),D**
FDCB d BB	RES 7,(IY+d),E**
FDCB d BC	RES 7,(IY+d),H**
FDCB d BD	RES 7,(IY+d),L**
FDCB d BE	RES 7,(IY+d)
FDCB d BF	RES 7,(IY+d),A**
FDCB d C0	SET 0,(IY+d),B**
FDCB d C1	SET 0,(IY+d),C**
FDCB d C2	SET 0,(IY+d),D**
FDCB d C3	SET 0,(IY+d),E**
FDCB d C4	SET 0,(IY+d),H**
FDCB d C5	SET 0,(IY+d),L**
FDCB d C6	SET 0,(IY+d)
FDCB d C7	SET 0,(IY+d),A**
FDCB d C8	SET 1,(IY+d),B**
FDCB d C9	SET 1,(IY+d),C**
FDCB d CA	SET 1,(IY+d),D**
FDCB d CB	SET 1,(IY+d),E**
FDCB d CC	SET 1,(IY+d),H**
FDCB d CD	SET 1,(IY+d),L**
FDCB d CE	SET 1,(IY+d)
FDCB d CF	SET 1,(IY+d),A**
FDCB d D0	SET 2,(IY+d),B**
FDCB d D1	SET 2,(IY+d),C**
FDCB d D2	SET 2,(IY+d),D**
FDCB d D3	SET 2,(IY+d),E**
FDCB d D4	SET 2,(IY+d),H**
FDCB d D5	SET 2,(IY+d),L**
FDCB d D6	SET 2,(IY+d)
FDCB d D7	SET 2,(IY+d),A**
FDCB d D8	SET 3,(IY+d),B**
FDCB d D9	SET 3,(IY+d),C**
FDCB d DA	SET 3,(IY+d),D**
FDCB d DB	SET 3,(IY+d),E**
FDCB d DC	SET 3,(IY+d),H**
FDCB d DD	SET 3,(IY+d),L**
FDCB d DE	SET 3,(IY+d)
FDCB d DF	SET 3,(IY+d),A**
FDCB d E0	SET 4,(IY+d),B**
FDCB d E1	SET 4,(IY+d),C**
FDCB d E2	SET 4,(IY+d),D**
FDCB d E3	SET 4,(IY+d),E**
FDCB d E4	SET 4,(IY+d),H**
FDCB d E5	SET 4,(IY+d),L**
FDCB d E6	SET 4,(IY+d)
FDCB d E7	SET 4,(IY+d),A**
FDCB d E8	SET 5,(IY+d),B**
FDCB d E9	SET 5,(IY+d),C**
FDCB d EA	SET 5,(IY+d),D**
FDCB d EB	SET 5,(IY+d),E**
FDCB d EC	SET 5,(IY+d),H**
FDCB d ED	SET 5,(IY+d),L**
FDCB d EE	SET 5,(IY+d)
FDCB d EF	SET 5,(IY+d),A**
FDCB d F0	SET 6,(IY+d),B**
FDCB d F1	SET 6,(IY+d),C**
FDCB d F2	SET 6,(IY+d),D**
FDCB d F3	SET 6,(IY+d),E**

FDCB d F4	SET 6,(IY+d),H**
FDCB d F5	SET 6,(IY+d),L**
FDCB d F6	SET 6,(IY+d)
FDCB d F7	SET 6,(IY+d),A**
FDCB d F8	SET 7,(IY+d),B**
FDCB d F9	SET 7,(IY+d),C**
FDCB d FA	SET 7,(IY+d),D**
FDCB d FB	SET 7,(IY+d),E**
FDCB d FC	SET 7,(IY+d),H**
FDCB d FD	SET 7,(IY+d),L**
FDCB d FE	SET 7,(IY+d)
FDCB d FF	SET 7,(IY+d),A**
FDE1	POP IY
FDE3	EX (SP),IY
FDE5	PUSH IY
FDE9	JP (IY)
FDF9	LD SP,IY
FE n	CP n
FF	RST 38H

This page intentionally left empty

Appendix C

Bibliography

[1] Mark Rison Z80 page for !CPC.
`http://www.acorn.co.uk/~mrison/en/cpc/tech.html`

[2] YAZE (Yet Another Z80 Emulator). This is a CPM emulator by Frank Cringle. It emulates almost every undocumented flag, very good emulator. Also includes a very good instruction exerciser and is released under the GPL.
`ftp://ftp.ping.de/pub/misc/emulators/yaze-1.10.tar.gz`
Note: the instruction exerciser zexdoc/zexall does not test I/O instructions and not all normal instructions (for instance LD A,(IX+n) is tested, but not with different values of n, just n=1, values above 128 (LD A,(IX-n) are not tested) but it still gives a pretty good idea of how well a simulated Z80 works.

[3] Z80 Family Official Support Page by Thomas Scherrer. Very good – your one-stop Z80 page.
`http://www.geocities.com/SiliconValley/Peaks/3938/z80_home.htm`

[4] Spectrum FAQ technical information.
`http://www.worldofspectrum.org/faq/`

[5] Gerton Lunter's Spectrum emulator (Z80). In the package there is a file TECHINFO.DOC, which contains a lot of interesting information. Note that the current version can only be unpacked in Windows.
`ftp://ftp.void.jump.org/pub/sinclair/emulators/pc/dos/z80-400.zip`

[6] Mostek Z80 Programming Manual – a very good reference to the Z80.

[7] Z80 Product Specification, from MSX2 Hardware Information.
`http://www.hardwareinfo.msx2.com/pdf/Zilog/z80.pdf`

[8] ZX Spectrum Next information.
`https://wiki.specnext.dev/`

This page intentionally left empty

Appendix D

GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or non-commercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

D.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose mark-up has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without mark-up, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

D.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

D.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

D.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
- Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

D.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

D.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

D.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

D.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

D.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

D.10 Future Revisions of This License

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.