

dastaZ80 Mark III
Programmer's Reference Guide

Disclaimer

The products described in this manual are intended for educational purposes, and should not be used for controlling any machinery, critical component in life support devices or any system in which failure could result in personal injury if any of the described here products fail.

These products are subject to continuous development and improvement. All information of a technical nature and particulars of the products and its use are given by the author in good faith. However, it is acknowledged that there may be errors or omissions in this manual. Therefore, the author cannot accept any liability for any loss or damage arising from the use of any information or particulars in this manual.

Licenses

Hardware is licensed under the **Creative Commons Attribution-ShareAlike 4.0 International License**

<http://creativecommons.org/licenses/by-sa/4.0/>

Software is licensed under **The MIT License**

<https://opensource.org/licenses/MIT>

Documentation is licensed under the **Creative Commons Attribution-ShareAlike 4.0 International License**

<http://creativecommons.org/licenses/by-sa/4.0/>

Document Conventions

The following conventions are used in this manual:

MUST	MUST denotes that the definition is an absolute requirement.
SHOULD	SHOULD denotes that it is recommended, but that there may exist valid reasons to ignore it.
DEVICE	Device names are displayed in bold all upper case letters, and refer to hardware devices.
<code>Courier</code>	Text appearing in the <code>Courier</code> font represents either an OS System Variable, a Z80 CPU Register, or a Z80 Flag. OS System Variables are identifiers for specific MEMORY addresses that can be used to read statuses and to pass information between routines or programs.
<code>0x14B0</code>	Numbers prefixed by 0x indicate an Hexadecimal value. Unless specified, memory addresses are always expressed in Hexadecimal.
<code>F_abcdef</code>	Text starting with <code>F_</code> refers to the name of an OS routine that can be called via Jumpblocks.
<code>jp abcdef</code>	Refers to the Z80 mnemonic for <i>jump</i> , which transfers the CPU Program Counter to a specific MEMORY address.

The SD card is referred as **DISK**, while the Floppy Disk Drive is referred as **DISK** or as **FDD**.

The 80 column text VGA output is referred as **CONSOLE** or as **High Resolution Display**.

The 40 column graphics Composite Video output is referred as **Low Resolution Display** or **VDP screen** or simply **VDP**.

The Operating System may be referred as **DZOS**, **dzOS** or simply **OS**.

MEMORY refers to both **ROM** and **RAM**.

Memory used by the **Low Resolution Display** is referred as **VRAM** (Video RAM).

The sound chip may be referred as **Sound Chip** or **PSG** (Programmable Sound Generator).

The Real-Time Clock is referred as **RTC**.

In the list of subroutines, the **Destroys** row lists the **CPU** registers and OS System Variables that are destroyed by the subroutine. And it is understood that the listed register or variable value is overwritten within the subroutine.

Related Documentation

- dastaZ80 User's Manual[[1](#)]
- dastaZ80 Technical Reference Manual[[2](#)]
- dzOS Github Repository[[3](#)]

Contents

1	Memory Map	1
1.1	ROM	1
1.2	RAM	2
1.3	VDP	6
2	I/O Map	8
3.1	DISK Routines	9
3	BIOS Jumpblocks	9
3.2	General Routines	12
3.3	Dual Joystick Routines	12
3.4	Non-Maskable Interrupt (NMI)	12
3.5	PSG Routines	13
3.6	Real-Time Clock Routines	14
3.7	Serial Routines	15
3.8	VDP Routines	16
4.1	General Routines	20
4.2	Serial Routines	20
4	Kernel Jumpblocks	20
4.3	DZFS (file system) Routines	22
4.4	Math Routines	26
4.5	String manipulation Routines	27
4.6	Conversion Routines	30
4.7	MEMORY Routines	32
4.8	Real-Time Clock Routines	33
4.9	VDP Routines	34
5	dastaZ80 File System (DZFS)	36
5.1	DZFS characteristics	36
5.2	DISK anatomy	36
5.3	How Volume Serial Number is calculated	39
5.4	How Dates (creation/last modified) are calculated	39
5.5	How Times (creation/last modified) are calculated	39
5.6	Block Number, Sector Number and Addresses	40
6	How To	41
6.1	Read data from DISK	41
6.2	Write data to DISK	41
6.3	Convert between HEX and DEC and ASCII	41
6.4	How to display Sprites	42
6.5	Develop software for dzOS	43
7	Appendixes	46
7.1	ANSI Terminal colours	46
7.2	VDP Composite colours	46
7.3	VDP Screen resolutions	46
7.4	VDP Limitations	48
7.5	Jiffy Counter	49
7.6	OS Boot Sequence	49
7.7	dzOS Programming Style	51

1 Memory Map

1.1 ROM

The **ROM** is a 16KB EEPROM, and is divided as follows:

Address		Description		Size (bytes)
0x0000	0x0007	Cold Boot	BIOS	8
0x0008	0x0215	init SIO/2		526
0x0216	0x0FFF	BIOS code		3,562
0x1000	0x26C7	Kernel code	Kernel	5,832
0x26B7	0x26C7	dzOS version build		17
0x26C8	0x3A88	CLI code	CLI	5,057
0x3A89	0x3AAB	Bootstrap	BOOTSTRAP	35
0x3AAC	0x3D9B	8x6 Font Pattern set (alphanumeric only 0-Z)		752
0x3E20	0x3F0F	BIOS Jumpblock	Jumpblocks	240
0x3F10	0x3FFF	Kernel Jumpblock		240

1.1.1 BIOS

The Basic Input/Output System (BIOS) is the part of the DZOS operating system that contains the subroutines that communicate directly with the hardware. It is, therefore, hardware dependent.

1.1.2 Kernel

The Kernel's main task is to facilitate interactions between any software and the BIOS. It is, therefore, hardware independent.

Though it is entirely possible for any software to either communicate directly with the BIOS via it's public functions (aka [Jumpblocks](#)) or even directly with the hardware via it's ports, it is highly recommended to use the Kernel public functions (aka [Kernel Jumpblocks](#)). The reason being that the Kernel subroutines are well tested and often already contain error handling, but more importantly because some functions set/unset values in the so called [System Variables \(SYSVARS\)](#). Failing to set/unset those values, could result on undesired behaviour of the OS or of some software dependant on it.

1.1.3 CLI

The Command-Line Interface is the the part of the OS that interacts with the user, by providing commands that can be entered via the keyboard and information provided via the screen.

A full list of available CLI commands is provided in the *dastaZ80 User's Manual*.

1.1.4 BOOTSTRAP

The bootstrap is a part of the BIOS that is executed first when the computer is powered ON or reset (cold boot). The one and only purpose of this code is to copy the contents of the **ROM** into **RAM** and then disable the **ROM** chip, so that the whole OS resides in **RAM**.

It does this by first copying all bytes (16 KB) from the **ROM** into the so called **High RAM** (starting at 0x8000). Then electronically disables the **ROM** chip, and finally copies all 16 KB bytes from **High RAM** into **Low RAM** (starting at 0x0000). Then it gives control to back to the BIOS' cold boot subroutine, which starts executing code from **Low RAM**.

1.1.5 Jumpblocks

1.2 RAM

The **RAM** is a 64KB SRAM, and is divided as follows:

Address		Description	Size (bytes)
0x4000	0x401F	Stack	32
0x4020	0x4174	System Variables	389
0x41A5	0x421F	Reserved for future use	123
0x4220	0x441F	DISK Buffer	512
0x4420	0xFFFF	Free RAM	48,096

1.2.1 Stack

A *Stack* is a list of words (2 bytes) that uses Last In First Out (LIFO) access method. It is used by the **CPU** to keep track of **MEMORY** addresses when executing a *call* instruction.

The programmer can also store (*PUSH*) or retrieve (*POP*) values on/from the top of the stack.

Usage of the Stack requires very careful attention. doing (*PUSH*) without the corresponding (*POP*) or vice versa, will set the CPU on the wrong path of execution. Most of the time just hanging the computer, but also potentially destroying information if an access to disk is triggered by the wrong call.

1.2.2 System Variables (SYSVARS)

The area of **RAM** called *System Variables (SYSVARS)* is an area heavily used by the OS, but it can also be used by a program to communicate with the OS.

The area has been *split* as follows:

- **SIO**

- 0x4020 - **SIO_CH_A_BUFFER** (64 bytes): Buffer for SIO Channel A.
- 0x4060 - **SIO_CH_A_IN_PTR** (2 bytes)
- 0x4062 - **SIO_CH_A_RD_PTR** (2 bytes)
- 0x4064 - **SIO_CH_A_BUFFER_USED** (1 byte)
- 0x4065 - **SIO_CH_A_LASTCHAR** (1 bytes)
- 0x4066 - **SIO_CH_B_BUFFER** (64 bytes): Buffer for SIO Channel B.
- 0x40A6 - **SIO_CH_B_IN_PTR** (2 bytes)
- 0x40A6 - **SIO_CH_B_RD_PTR** (2 bytes)
- 0x40AA - **SIO_CH_B_BUFFER_USED** (1 byte)

- **DISK Superblock**

- 0x40AB - **DISK_is_formatted** (1 byte): tells to the OS if the **DISK** can be used.
 - * 0xFF = formatted with *DZFS*.
 - * 0x00 = not formatted.
- 0x40AC - **DISK_show_deleted** (1 byte)
 - * 0x00 = do not show deleted files in *cat* command results.

- * 0x01 = show also deleted files in *cat* command results.

- 0x40AD - **DISK_cur_sector** (2 bytes): current Sector being used by the OS.

- **DISK BAT**

- 0x40AF - **DISK_cur_file_name** (14 bytes): Filename of file currently being load or saved.

- 0x40BD - **DISK_cur_file_attr** (1 byte): Attributes of file currently being load or saved.

- * Bit 0: if set, file is Read Only.

- * Bit 1: if set, file is Hidden (it does not display in *cat* command results).

- * Bit 2: if set, file is System (it does not display in *cat* command results).

- * Bit 3: if set, file is Executable.

- * Bits 4-7: not used.

- 0x40BE - **DISK_cur_file_time_created** (2 bytes): time when currently being load or saved file was created.

- 0x40C0 - **DISK_cur_file_date_created** (2 bytes): date when currently being load or saved file was created.

- 0x40C2 - **DISK_cur_file_time_modified** (2 bytes): time when currently being load or saved file was last modified.

- 0x40C4 - **DISK_cur_file_date_modified** (2 bytes): date when currently being load or saved file was last modified.

- 0x40C6 - **DISK_cur_file_size_bytes** (2 bytes): size in bytes of file currently being load or saved.

- 0x40C8 - **DISK_cur_file_size_sectors** (1 byte): size in sectors of file currently being load or saved.

- 0x40C9 - **DISK_cur_file_entry_number** (2 bytes): entry number in the BAT, of file currently being load or saved.

- 0x40CB - **DISK_cur_file_1st_sector** (2 bytes): sector number, of the first sector, where the bytes of file currently being load or saved are stored in the **DISK**.

- 0x40CD - **DISK_cur_file_load_addr** (2 bytes): address where the bytes of file currently being load will be stored in **RAM**.

- **CLI**: buffers used by CLI to store temporary data.

- 0x40CF - **CLI_prompt_addr** (2 bytes): The address of the CLI Prompt subroutine. Programs that need to return control to CLI on exit, MUST jump to the address stored here.

- 0x40D1 - **CLI_buffer** (6 bytes): generic buffer.

- 0x40D7 - **CLI_buffer_cmd** (16 bytes): when a user enters a command and its parameters, the command alone is stored here.

- 0x40E7 - **CLI_buffer_parm1_val** (16 bytes): when a user enters a command and its parameters, the first parameter is stored here.

- 0x40F7 - **CLI_buffer_parm2_val** (16 bytes): when a user enters a command and its parameters, the second parameter is stored here.

- 0x4107 - **CLI_buffer_pgm** (32 bytes): generic buffer.

- 0x4127 - **CLI.buffer_full_cmd** (64 bytes): when a user enters a command and its parameters, the entire line entered by the user is stored here. This is useful for passing parameters to programs called with *run* command.

- **RTC**

- 0x4167 - **RTC_hour** (1 byte): 24h format, in hexadecimal (0x00-0x17).
- 0x4168 - **RTC_minutes** (1 byte): in hexadecimal (0x00-0x3B).
- 0x4169 - **RTC_seconds** (1 byte): in hexadecimal (0x00-0x3B).
- 0x416A - **RTC_century** (1 byte): 20 part of year 20xx, in hexadecimal (0x14 = 20).
- 0x416B - **RTC_year** (1 byte): xx part of year 20xx, in hexadecimal (e.g. 0x16 = 22). The **RTC** supports until 2079, therefore maximum value is 0x4F.
- 0x416C - **RTC_year4** (2 bytes): four digit year, in hexadecimal (e.g. 0x07E6 = 2022). The **RTC** supports until 2079, therefore maximum value is 0x081F.
- 0x416E - **RTC_month** (1 byte): in hexadecimal (0x00-0x0C).
- 0x416F - **RTC_day** (1 byte): in hexadecimal (0x00-0x1F).
- 0x4170 - **RTC_day_of_the_week** (1 byte): 0x00=Sunday, 0x01=Monday, 0x02=Tuesday, 0x03=Wednesday, 0x04=Thursday, 0x05= Friday, 0x06=Saturday

- **Math**

- 0x4171 - **MATH_CRC** (2 bytes): CRC-16 CRC.
- 0x4173 - **MATH_polynomial** (2 bytes): CRC-16 Polynomial.

- **Generic**

- 0x4175 - **FDD_detected** (1 byte): 1=FDD detected, 0=Not detected
- 0x4176 - **SD_images_num** (1 byte): number of Disk Image Files found by **ASMDC**.
- 0x4177 - **DISK_current** (1 byte): current **DISK** unit active. All disk operations will be on this **DISK**.
- 0x4178 - **DISK_status** (1 byte): status of the **FDD**.
 - * Low Nibble (0x00 if all OK)
 - bit 0 = not used.
 - bit 1 = not used.
 - bit 2 = set if last command resulted in error.
 - bit 3 = not used.
 - * High Nibble: error code of last operation.
- 0x4179 - **DISK_file_type** (1 byte): File Type when creating (*save*) next file.
- 0x417A - **DISK_loadsave_addr** (2 bytes): see [Read data from DISK](#) and [Write data to DISK](#).
- 0x417C - **tmp_addr1** (2 bytes): temporary storage for an address.
- 0x417E - **tmp_addr2** (2 bytes): temporary storage for an address.
- 0x4180 - **tmp_addr3** (2 bytes): temporary storage for an address.

- 0x4182 - **tmp_byte** (1 byte): temporary storage for a byte.
- 0x4183 - **tmp_byte2** (1 byte): temporary storage for a byte.

- **VDP**

- 0x4184 - **NMI_enable**: Enable (1) / Disable (0) the execution of the [NMI subroutine](#).
- 0x4185 - **NMI_usr_jump**: Enable (1) / Disable (0) the user configurable *BIOS_NMI_JP* jump of the [NMI subroutine](#).
- 0x4186 - **VDP_cur_mode**:
 - * 0 = Text Mode
 - * 1 = Graphics I Mode
 - * 2 = Graphics II Mode
 - * 3 = Multicolour Mode
 - * 4 = Graphics II Mode Bitmapped
- 0x4187 - **VDP_cursor_x** (1 byte): Current horizontal position of the cursor on the **VDP** screen.
- 0x4188 - **VDP_cursor_y** (1 byte): Current vertical position of the cursor on the **VDP** screen.
- 0x4189 - **VDP_PTRNTAB_addr** (2 bytes): Address of current Mode's Pattern Table.
- 0x418B - **VDP_NAMETAB_addr** (2 bytes): Address of current Mode's Name Table.
- 0x418D - **VDP_COLRTAB_addr** (2 bytes): Address of current Mode's Colour Table.
- 0x418F - **VDP_SPRPTAB_addr** (2 bytes): Address of current Mode's Sprite Pattern Table.
- 0x4191 - **VDP_SPRATAB_addr** (2 bytes): Address of current Mode's Sprite Attribute Table.
- 0x4193 - **VDP_jiffy_byte1** (1 byte): [Jiffy Counter](#)'s byte 1.
- 0x4194 - **VDP_jiffy_byte2** (1 byte): [Jiffy Counter](#)'s byte 2.
- 0x4195 - **VDP_jiffy_byte3** (1 byte): [Jiffy Counter](#)'s byte 3.

- **System Colour Scheme**

These are the default colours used by messages displayed on the **High Resolution Screen** (VGA), and can be re-defined by the user by changing each byte value.

- 0x4196 - **col_kernel_debug** (1 byte): default is Cyan.
- 0x4197 - **col_kernel_disk** (1 byte): default is Magenta.
- 0x4198 - **col_kernel_error** (1 byte): default is Red.
- 0x4199 - **col_kernel_info** (1 byte): default is Green.
- 0x419A - **col_kernel_notice** (1 byte): default is Yellow.
- 0x419B - **col_kernel_warning** (1 byte): default is Magenta.
- 0x419C - **col_kernel_welcome** (1 byte): default is Blue.
- 0x419D - **col_CLi_debug** (1 byte): default is Cyan.
- 0x419E - **col_CLi_disk** (1 byte): default is Magenta.
- 0x419F - **col_CLi_error** (1 byte): default is Red.

- 0x41A0 - **col.CLI_info** (1 byte): default is Green.
- 0x41A1 - **col.CLI_input** (1 byte): default is White.
- 0x41A2 - **col.CLI_notice** (1 byte): default is Yellow.
- 0x41A3 - **col.CLI_prompt** (1 byte): default is Blue.
- 0x41A4 - **col.CLI_warning** (1 byte): default is Magenta.

1.2.3 DISK Buffer

Read and Write operations on **DISK** are done Sector by Sector (i.e 512 Bytes).

When loading a file, dzOS asks **ASMDC** for the first 512 bytes of the file, and stores it in this buffer. After the bytes are moved to **RAM**, dzOS asks **ASMDC** for the next 512 bytes, and so on until the file is read entirely.

When saving a file, dzOS copies the first 512 bytes of the file from **RAM** to this buffer. After sending the bytes to **ASMDC**, dzOS copies the next 512 bytes of the file, and so on until the file is saved entirely.

When doing a *cat* of a **DISK**, dzOS asks **ASMDC** for the first 512 bytes of the BAT, and stores it in this buffer. After the list of files is shown on the screen, dzOS asks **ASMDC** for the next 512 bytes, and so on until the entire catalogue has been shown.

1.3 VDP

1.3.1 Text Mode

Pattern Table	0x0000
Name Table	0x0800
UNUSED	0x0BC0
	0x3FFF

1.3.2 Graphics I Mode

Sprites Patterns	0x0000
Pattern Table	0x0800
SPRITE ATTRIBUTTES	0x1000
UNUSED	0x1080
Name Table	0x1400
UNUSED	0x1800
COLOUR TABLE	0x2000
UNUSED	0x2020
	0x3FFF

1.3.3 Graphics II Mode

Pattern Table	0x0000
Sprites Patterns	0x1800
COLOUR TABLE	0x2000
Name Table	0x3800
SPRITE ATTRIBUTTES	0x3B00
UNUSED	0x3C00
	0x3FFF

1.3.4 Multicolour Mode

Sprites Patterns	0x0000
Pattern Table	0x0800
UNUSED	0x0E00
SPRITE ATTRIBUTTES	0x1000
UNUSED	0x1080
Name Table	0x1400
	0x1700
UNUSED	0x3FFF

2 I/O Map

VDP	0x10	Mode 0 (VRAM)
	0x11	Mode 1 (Register)
PSG	0x20	PSG Register
	0x21	PSG Data
ROM / RAM	0x38	ROM Paging
Joystick Ports	0x40	Joystick 1
	0x41	Joystick 2
RTC	0x50	RTC I/O
SIO	0x80	Channel A Control
	0x81	Channel A Data
	0x82	Channel B Control
	0x83	Channel B Data

3 BIOS Jumpblocks

3.1 DISK Routines

3.1.1 F_BIOS_SD_BUSY_WAIT

Action	Calls ASMDC to check if the DISK is busy, and loops until it is not busy.
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.2 F_BIOS_SD_GET_STATUS

Action	Calls ASMDC to check the status of the SD Card module.
Entry	None
Exit	<i>SD_status</i> bit 0 = set if SD card was not found bit 1 = set if image file was not found bit 2 = set if last command resulted in error
Destroys	A
Calls	F_BIOS_SD_BUSY F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.3 F_BIOS_SD_PARK_DISKS

Action	Tells ASMDC to close the Image File
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SD_BUSY F_BIOS_SERIAL_CONOUT_B

3.1.4 F_BIOS_SD_MOUNT_DISKS

Action	Tells ASMDC to open the Image File
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SD_BUSY F_BIOS_SERIAL_CONOUT_B

3.1.5 F_BIOS_DISK_READ_SEC

Action	Reads a Sector (512 bytes), from the DISK and places the bytes into the CF_BUFFER_START
Entry	E = sector address LBA 0 (bits 0-7) D = sector address LBA 1 (bits 8-15) C = sector address LBA 2 (bits 16-23) B = sector address LBA 3 (bits 24-27) BC are not used (set to zero), because max sector is 65,535
Exit	CF_BUFFER_START contains the 512 bytes read
Destroys	A, B, HL, DISK_BUFFER_START
Calls	F_BIOS_SD_BUSY F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.6 F_BIOS_DISK_WRITE_SEC

Action	Writes a Sector (512 bytes), from the DISK_BUFFER_START into the DISK
Entry	E = sector address LBA 0 (bits 0-7) D = sector address LBA 1 (bits 8-15) C = sector address LBA 2 (bits 16-23) B = sector address LBA 3 (bits 24-27) BC are not used (set to zero), because max sector is 65,535
Exit	DISK_BUFFER_START contains the 512 bytes written
Destroys	A, HL, DISK_BUFFER_START
Calls	F_BIOS_SD_BUSY F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.7 F_BIOS_FDD_GET_STATUS

Action	Calls ASMDC to check the status of the Floppy Disk Drive.
Entry	None
Exit	<i>SD_status</i> bit 0 = set if FDD was not detected bit 1 = Not used bit 2 = set if last command resulted in error
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.8 F_BIOS_FDD_BUSY_WAIT

Action	Calls ASMDC to check if the FDD is busy, and loops until it is not busy.
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.9 F_BIOS_FDD_CHANGE

Action	Tells the ASMDC that the current DISK for operations is now the FDD .
Entry	None
Exit	DISK_status is updated
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B

3.1.10 F_BIOS_FDD_LOWLVL_FORMAT

Action	Tells the ASMDC to low-level format a DISK in the FDD . This function does not set up any file system. It just fills with 0xF6 all bytes of all sectors.
Entry	None
Exit	A = 0x00 if everything OK. Bit 2 set if command resulted in error.
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.11 F_BIOS_FDD_MOTOR_ON

Action	Tells the ASMDC to switch the FDD motor on. It is a recommended practice to switch the motor on and off manually if multiple sectors are to read or written.
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B

3.1.12 F_BIOS_FDD_MOTOR_OFF

Action	Tells the ASMDC to switch the FDD motor off. It is a recommended practice to switch the motor on and off manually if multiple sectors are to read or written.
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B

3.1.13 F_BIOS_FDD_CHECK_DISKIN

Action	Asks the ASMDC to check if a Floppy Disk is inside the FDD .
Entry	None
Exit	A = 0x00 yes / 0xFF no
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.1.14 F_BIOS_FDD_CHECK_WPROTECT

Action	Asks the ASMDC to check if the Floppy Disk is write protected.
Entry	None
Exit	A = 0x00 yes / 0xFF no
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_B F_BIOS_SERIAL_CONIN_B

3.2 General Routines

3.2.1 F_BIOS_WBOOT

Action	Warm Boot. Executed after SIO/2 initialisation, or after a <i>reset</i> command
Entry	None
Exit	None
Destroys	None
Calls	<i>jp</i> F_KRN_START

3.2.2 F_BIOS_SYSHALT

Action	Halts the computer.
Entry	None
Exit	Disables both Maskable and Non-Maskable interrupts, and then executes a <i>halt</i> command.
Destroys	None
Calls	None

3.3 Dual Joystick Routines

3.3.1 F_BIOS_JOYS_GET_STAT

Action	Get status of Joysticks.
Entry	A = Joystick Port to get status from (1=JOY1, 2=JOY2).
Exit	A 0x00 = None 0x01 = Up 0x02 = Down 0x04 = Left 0x08 = Right 0x10 = Fire
Destroys	A, C
Calls	None

3.4 Non-Maskable Interrupt (NMI)

When the chip used for the generation of the Composite Video (the *Texas Instruments TMS9918A VDP*) is done drawing the screen, it enters the so called *vertical refresh mode* and issues an interrupt that gives the **CPU** a window of 4.3 miliseconds (4300 μ s). This interrupt occurs about every 1/60th second.

But this chip doesn't have the *priority daisy-chain* feature of other Zilog chips, and when raising an interrupt to the **CPU** pin */INT* could create bus contention¹. Therefore, the interrupt pin */INT* of the TMS9918A is connected to the */NMI* pin of the **CPU**.

This means that 1) there is no standard way² to programatically disable these interrupts, and 2) that every 1/60th second the **CPU** will receive a Non-Maskable Interrupt and therefore, store the current Program Counter (PC) in the stack and jump to the location 0x0066.

At this address, dzOS contains a small piece of code that allows programs to enable and disable a jump to their own subroutine. For example, a video game playing a tune will need to update the **PSG** in an interrupt basis.

¹Bus contention occurs when all devices communicate directly with each other through a single shared channel (Address and Data buses), and more than one device attempts to place values on the channel at the same time.

²By design the **CPU** doesn't offer an instruction to enable/disable this type of interrupts, hence are called *non-maskable*. But this has been implemented programatically within dzOS, and therefore NMI can be enabled/disabled via the funtions [F_BIOS_VDP_EI](#) and [F_BIOS_VDP_DI](#)

This code works as follows:

- All **CPU** registers are saved (with *PUSH*).
- The [Jiffy Counter](#) is incremented.
- If the flag *NMI_usr_jump* is enabled (1), the subroutine jumps to whatever address is in bytes 2 and 3 of *BIOS_NMI_JP*
- If the flag is disabled (0), **CPU** registers are restored, and the subroutine ends.

The end of your subroutine **MUST** be a *jp F_BIOS_NMI_END*. This is the part that restores the previously saved **CPU** registers and ends the subroutine with *RETN*. Otherwise the system will certainly crash.

When writing a subroutine that will be called at each interrupt, remember that the window given for **CPU** time is 4.3 milliseconds (4300 μ s). The current NMI routine takes 35.81 μ s. After this window, the **VDP** will start drawing again in the screen.

3.4.1 F_BIOS_NMI_END

Action	Performs <i>POP</i> instructions for all CPU registers. Reads the VDP Status Register , to acknowledge the interrupt and allow more to happen, and performs a return from non maskable interrupt (<i>RETN</i>).
Entry	None
Exit	None
Destroys	Restores CPU registers AF, BC,DE, HL, IX and IY to the values they had before the NMI was triggered.
Calls	F_BIOS_VDP_READ_STATREG

3.4.2 BIOS_NMI_JP

This is the start address of three bytes corresponding to the instruction *jp BIOS_NMI_END*. The first byte (*C3*) **MUST** not be changed. The next two bytes are the ones a program can change to make the interrupt jump to a desired subroutine.

3.5 PSG Routines

3.5.1 F_BIOS_PSG_SET_REGISTER

Action	Set a value to a PSG Register.
Entry	A = register number to set, E = value to set.
Exit	None
Destroys	C
Calls	None

3.5.2 F_BIOS_PSG_READ_REGISTER

Action	Read the value of a PSG Register.
Entry	A = register number to read.
Exit	A = value of the register.
Destroys	C
Calls	None

3.5.3 F_BIOS_PSG_INIT

Action	Initialises the PSG to: Noise OFF, Audio OFF, I/O Port as Output.
Entry	None
Exit	None
Destroys	A, B, HL, DE
Calls	F_BIOS_PSG_SET_REGISTER

3.5.4 F_BIOS_PSG_BEEP

Action	Makes a short beep-like sound.
Entry	None
Exit	None
Destroys	A, B, HL, E
Calls	F_BIOS_VDP_VBLANK_WAIT F_BIOS_PSG_SET_REGISTER

3.6 Real-Time Clock Routines

3.6.1 F_BIOS_RTC_INIT

Action	Initialises the RTC to 24 hours format and sets the oscillator clock as running, so that the time keeps ticking when running on battery.
Entry	None
Exit	None
Destroys	A
Calls	None

3.6.2 F_BIOS_RTC_GET_TIME

Action	Gets the current time from the RTC , and stores hour, minutes and seconds as hexadecimal values in SYSVARS .
Entry	None
Exit	RTC_hour, RTC_minutes, RTC_seconds
Destroys	A
Calls	F_KRN_BCD_TO_BIN

3.6.3 F_BIOS_RTC_GET_DATE

Action	Gets the current date from the RTC , and stores day, month, year and day of the week as hexadecimal values in SYSVARS .
Entry	None
Exit	RTC_day, RTC_month, RTC_year, RTC_day_of_the_week
Destroys	A, HL
Calls	F_KRN_BCD_TO_BIN

3.6.4 F_BIOS_RTC_SET_TIME

Action	Tells the RTC to store a new hours, minutes and seconds.
Entry	RTC_hour, RTC_minutes, RTC_seconds
Exit	None
Destroys	A
Calls	F_KRN_BIN_TO_BCD4

3.6.5 F_BIOS_RTC_SET_DATE

Action	Tells the RTC to store a new day, month, year and day of the week.
Entry	RTC_day, RTC_month, RTC_year, RTC_day_of_the_week
Exit	None
Destroys	A
Calls	F_KRN_BIN_TO_BCD4

3.6.6 F_BIOS_CHECK_BATTERY

Action	Asks the RTC if the battery is healthy or has to be replaced.
Entry	None
Exit	Z Flag set if battery unhealthy
Destroys	A
Calls	None

3.7 Serial Routines**3.7.1 F_BIOS_SERIAL_INIT**

Action	Initialises SIO/2 : sets Channels A and B as 115,000 bps, 8N1, Interrupt in all characters Configures the interrupt vector to 0x60 Sets the CPU to Interrupt Mode 2 Enables Interrupts
Entry	None
Exit	None
Destroys	A, HL
Calls	<i>jp</i> F_BIOS_WBOOT

3.7.2 F_BIOS_SERIAL_CONIN_A

Action	Reads a character from the SIO/2 Channel A
Entry	None
Exit	A = character read
Destroys	A
Calls	None

3.7.3 F_BIOS_SERIAL_CONIN_B

Action	Reads a character from the SIO/2 Channel B
Entry	None
Exit	A = character read
Destroys	A
Calls	None

3.7.4 F_BIOS_SERIAL_CONOUT_A

Action	Sends a character to the SIO/2 Channel A
Entry	A = character to be send
Exit	None
Destroys	None
Calls	None

3.7.5 F_BIOS_SERIAL_CONOUT_B

Action	Sends a character to the SIO/2 Channel B
Entry	A = character to be send
Exit	None
Destroys	None
Calls	None

3.8 VDP Routines**3.8.1 F_BIOS_VDP_SET_ADDR_WR**

Action	Set a VRAM address for writting.
Entry	HL = address to be set
Exit	None
Destroys	C, H
Calls	None

3.8.2 F_BIOS_VDP_SET_ADDR_RD

Action	Set a VRAM address for reading.
Entry	HL = address to be read
Exit	None
Destroys	A, C
Calls	None

3.8.3 F_BIOS_VDP_SET_REGISTER

Action	Set a value to a VDP register.
Entry	A = register number B = value to set
Exit	None
Destroys	C
Calls	None

3.8.4 F_BIOS_VDP_EI

Action	Enable VDP Interrupts. This is independent of the value (bit 5) in the VDP Register 1 . What this does is that the NMI subroutine reads the VDP Status Register again in each run, and therefore it does allow more interrupts to happen.
Entry	None
Exit	None
Destroys	A
Calls	F_BIOS_VDP_READ_STATREG

3.8.5 F_BIOS_VDP_DI

Action	Disable VDP Interrupts. This is independent of the value (bit 5) in the VDP Register 1 . What this does is that the NMI subroutine does not read the VDP Status Register anymore, and therefore does not allow more interrupts to happen. IMPORTANT: Disabling VDP Interrupts will stop the Jiffy Counter .
Entry	None
Exit	None
Destroys	A
Calls	None

3.8.6 F_BIOS_VDP_READ_STATREG

Action	Read the read-only VDP Status Register . IMPORTANT: Reading the VDP Status Register clears (acknowledges) the VDP Interrupt. This is already done by the BIOS' NMI subroutine, so this function MUST not be used, unless NMI subroutines have been disabled with F_BIOS_VDP_DI
Entry	None
Exit	A = Status Register byte.
Destroys	A, C
Calls	None

3.8.7 F_BIOS_VDP_VRAM_CLEAR

Action	Set all cells of the VRAM (0x0000- 0x3FFF) to zero.
Entry	None
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR

3.8.8 F_BIOS_VDP_VRAM_TEST

Action	Set a value to each VRAM cell and then reads it back. If the value is not the same, something went wrong.
Entry	None
Exit	C Flag set if an error occurred.
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_ADDR_RD

3.8.9 F_BIOS_VDP_SET_MODE_TXT

Action	Set VDP to <i>Text Mode</i> display.
Entry	B = Sprite size (0=8×8, 1=16×16) C = Sprite magnification (0=no magnification, 1=magnification)
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_REGISTER

3.8.10 F_BIOS_VDP_SET_MODE_G1

Action	Set VDP to <i>Graphics I Mode</i> display.
Entry	B = Sprite size (0=8×8, 1=16×16) C = Sprite magnification (0=no magnification, 1=magnification)
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_REGISTER

3.8.11 F_BIOS_VDP_SET_MODE_G2

Action	Set VDP to <i>Graphics II Mode</i> display.
Entry	B = Sprite size (0=8×8, 1=16×16) C = Sprite magnification (0=no magnification, 1=magnification)
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_REGISTER

3.8.12 F_BIOS_VDP_SET_MODE_G2BM

Action	Set VDP to <i>Graphics II Bit-mapped Mode</i> display.
Entry	B = Sprite size (0=8×8, 1=16×16) C = Sprite magnification (0=no magnification, 1=magnification)
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_REGISTER

3.8.13 F_BIOS_VDP_SET_MODE_MULTICLR

Action	Set VDP to <i>Multicolour Mode</i> display.
Entry	B = Sprite size (0=8×8, 1=16×16) C = Sprite magnification (0=no magnification, 1=magnification)
Exit	None
Destroys	A, BC, D, HL
Calls	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_SET_REGISTER

3.8.14 F_BIOS_VDP_BYTE_TO_VRAM

Action	Writes a byte to currently pointed VRAM cell. The VDP autoincrements the VRAM address whenever a Read or a Write to VRAM is performed.
Entry	A = byte to be written
Exit	VRAM address autoincremented
Destroys	C
Calls	None

3.8.15 F_BIOS_VDP_VRAM_TO_BYTE

Action	Read a byte from VRAM . The VDP autoincrements the VRAM address whenever a Read or a Write to VRAM is performed.
Entry	None
Exit	A = read byte, VRAM address autoincremented.
Destroys	A, C
Calls	None

3.8.16 F_BIOS_VDP_JIFFY_COUNTER

Action	Increments the Jiffy Counter .
Entry	None
Exit	None
Destroys	A, IX, VDP_jiffy_byte1, VDP_jiffy_byte2, VDP_jiffy_byte3
Calls	None

3.8.17 F_BIOS_VDP_VBLANK_WAIT

Action	Test if the SYSVARS <i>VDP_jiffy_byte1</i> has changed. If not, continues waiting for the change in a loop. Otherwise, exits. See the VDP Limitations section on the <i>Appendixes</i> of this Guide for a possible bug on the VDP 's Status Register.
Entry	None
Exit	None
Destroys	A, B
Calls	None

3.8.18 F_BIOS_VDP_LDIR_VRAM

Action	Block transfer from RAM to VRAM .
Entry	BC = Block length (total number of bytes to copy) HL = Start address of VRAM DE = Start address of RAM
Exit	None
Destroys	A, BC, DE, HL, tmp_byte
Call	F_KRN_DIV1616 F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_BYTE_TO_VRAM

3.8.19 F_BIOS_VDP_CHAROUT_ATXY

Action	Print a character in the Low Resolution display , at the <i>VDP_cursor_x</i> , <i>VDP_cursor_y</i> position. <i>VDP_cursor_x</i> is incremented by 1, and if it has reached the maximum width (Mode 0 = 40, others = 32), resets it to zero and increases <i>VDP_cursor_y</i> by 1.
Entry	A = Character to be printed, in Hexadecimal ASCII.
Exit	None
Destroys	A, BC, DE, HL, IX, VDP_cursor_x, VDP_cursor_y
Call	F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_BYTE_TO_VRAM

4 Kernel Jumpblocks

4.1 General Routines

4.1.1 F_KRN_SYSHALT

Action	Prepares the computer for a <i>HALT</i> .
Entry	None.
Exit	None
Destroys	A, HL
Calls	F_BIOS_SD_PARK_DISKS F_KRN_SERIAL_WRSTRCLR

4.2 Serial Routines

4.2.1 F_KRN_SERIAL_SETFGCOLR

Action	Set the colour that will be used for the foreground (text). The colour will remain until a different one is set.
Entry	A = Colour number (as listed in Appendixes section)
Exit	None
Destroys	B, DE
Calls	F_BIOS_SERIAL_CONOUT_A <i>jp</i> F_KRN_SERIAL_SEND_ANSI_CODE

4.2.2 F_KRN_SERIAL_WRSTR

Action	Outputs a string, terminated with Carriage Return to the CONSOLE .
Entry	HL = address in MEMORY where the first character of the string to be output is.
Exit	None
Destroys	A, HL
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.3 F_KRN_SERIAL_WRSTRCLR

Action	Outputs a string, terminated with Carriage Return to the CONSOLE , with a specific foreground colour.
Entry	A = Colour number (as listed in Appendixes section) HL = address in MEMORY where the first character of the string to be output is.
Exit	None
Destroys	B, DE
Calls	F_KRN_SERIAL_SETFGCOLR <i>jp</i> F_KRN_SERIAL_WRSTR

4.2.4 F_KRN_SERIAL_WR6DIG_NOLZEROS

Action	Outputs to the CONSOLE a string of ASCII characters representing a number, without outputting the leading zeros. (e.g. 30 30 31 32 30 34 is 001204, but the output will be 1024)
Entry	IX = address in MEMORY where the ASCII characters are stored.
Exit	None
Destroys	A, B, DE, IX
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.5 F_KRN_SERIAL_RDCHARECHO

Action	Reads with echo. Reads a character from the SIO/2 Channel A, and outputs it to the CONSOLE .
Entry	None
Exit	A = read character.
Destroys	None
Calls	F_BIOS_SERIAL_CONIN_A F_BIOS_SERIAL_CONOUT_A

4.2.6 F_KRN_SERIAL_EMPTYLINES

Action	Outputs n number of empty lines to the CONSOLE .
Entry	B = number (n) of empty lines to output.
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.7 F_KRN_SERIAL_PRN_NIBBLE

Action	Outputs a single hexadecimal nibble in hexadecimal notation.
Entry	A = nibble to output. Nibble will be the less significant 4 bits of the byte.
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.8 F_KRN_SERIAL_PRN_BYTE

Action	Outputs a single hexadecimal byte in hexadecimal notation.
Entry	A = byte to output.
Exit	None
Destroys	A
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.9 F_KRN_SERIAL_PRN_BYTES

Action	Outputs n number of bytes as ASCII characters.
Entry	B = number (n) of bytes to output. HL = address in MEMORY where the first byte to output is.
Exit	None
Destroys	A, HL
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.10 F_KRN_SERIAL_PRN_WORD

Action	Outputs the 4 hexadecimal digits of a word in hexadecimal notation.
Entry	HL = word to be output.
Exit	None
Destroys	A
Calls	F_KRN_SERIAL_PRN_BYTE

4.2.11 F_KRN_SERIAL_SEND_ANSI_CODE

Action	Writes an ANSI code to the SIO/2 Channel A.
Entry	DE = address in MEMORY where the first byte of ANSI escape code is. B = number of bytes in the ANSI escape code.
Exit	None
Destroys	A, DE
Calls	F_BIOS_SERIAL_CONOUT_A

4.2.12 F_KRN_SERIAL_CLR_SIOCHA_BUFFER

Action	Clear (sets to zeros) the SIO Channel A Buffer.
Entry	None
Exit	None
Destroys	A, B, HL, SIO_CH_A_BUFFER_USED, SIO_CH_A_IN_PTR, SIO_CH_A_RD_PTR
Calls	None

4.3 DZFS (file system) Routines**4.3.1 F_KRN_DZFS_READ_SUPERBLOCK**

Action	Reads 512 bytes from Sector 0 (corresponding to the DZFS <i>Superblock</i>) into the disk buffer in MEMORY . If the <i>Superblock</i> does not contain the correct DZFS signature, <code>DISK_is_formatted</code> is set to 0x00. Otherwise, is set to 0x01.
Entry	None
Exit	None
Destroys	A, DE, <code>DISK_is_formatted</code>
Calls	F_BIOS_SD_READ_SEC

4.3.2 F_KRN_DZFS_READ_BAT_SECTOR

Action	Reads a BAT Sector from DISK into MEMORY .
Entry	<code>DISK_cur_sector</code> holds the sector number for the BAT.
Exit	<code>DISK Buffer</code> contains the BAT sector.
Destroys	HL
Calls	F_KRN_DZFS_SEC_TO_BUFFER

4.3.3 F_KRN_DZFS_BATENTRY_TO_BUFFER

Action	Extracts the data of a BAT entry from the <code>DISK Buffer</code> in MEMORY and populates the values into System variables.
Entry	A = BAT entry number to extract data from.
Exit	<code>DISK BAT System Variables</code> are populated. See RAM Memory Map for details.
Destroys	A, BC, DE, HL, IX, <code>tmp_addr1</code>
Calls	F_KRN_MULTIPLY816_SLOW

4.3.4 F_KRN_DZFS_SEC_TO_BUFFER

Action	Loads a Sector (512 bytes) from the DISK and copies the bytes into the <code>DISK Buffer</code> in MEMORY .
Entry	HL = Sector number to load.
Exit	<code>DISK Buffer</code> contains the bytes of Sector loaded.
Destroys	DE, HL
Calls	F_BIOS_SD_READ_SEC

4.3.5 F_KRN_DZFS_GET_FILE_BATENTRY

Action	Gets the BAT's entry number of a specified filename.
Entry	HL = Address where the filename to check is stored
Exit	BAT Entry values are stored in the SYSVARS . DE = \$0000 if filename found. Otherwise, whatever value had at start.
Destroys	A, B, DE, HL, tmp_byte, tmp_addr2, tmp_addr3
Calls	F_KRN_DZFS_SEC_TO_BUFFER F_KRN_DZFS_BATENTRY_TO_BUFFER F_KRN_STRLLENMAX F_KRN_STRCMP

4.3.6 F_KRN_DZFS_LOAD_FILE_TO_RAM

Action	Load a file from DISK . Copies the bytes stored in the DISK into MEMORY . If SYSVARS <i>DISK_loadsave_addr</i> is not zero, then loads file to this address. If zero, then if SYSVARS <i>DISK_cur_file_load_addr</i> is not zero, then loads file to this address. If also zero, then loads file to start of Free RAM .
Entry	DE = 1st sector number in the DISK . IX = file length in sectors.
Exit	None
Destroys	BC, DE, HL, IX, tmp_addr1
Calls	F_BIOS_SD_READ_SEC

4.3.7 F_KRN_DZFS_DELETE_FILE

Action	Marks a file as deleted. The mark is done by changing the first character of the filename to 0x7E (~)
Entry	DE = BAT Entry number.
Exit	None
Destroys	A, DE, HL,
Calls	F_KRN_MULTIPLY816_SLOW F_KRN_DZFS_SECTOR_TO_SD

4.3.8 F_KRN_DZFS_CHGATTR_FILE

Action	Changes the attributes (RHSE) of a file.
Entry	DE = BAT Entry number. A = attributes mask byte.
Exit	None
Destroys	DE, HL,
Calls	F_KRN_MULTIPLY816_SLOW F_KRN_DZFS_SECTOR_TO_SD

4.3.9 F_KRN_DZFS_RENAME_FILE

Action	Changes the name of a file.
Entry	IY = MEMORY address where the new filename is stored. DE = BAT Entry number.
Exit	None
Destroys	A, BC, DE, HL, IY
Calls	F_KRN_MULTIPLY816_SLOW F_KRN_DZFS_SECTOR_TO_SD

4.3.10 F_KRN_DZFS_FORMAT_DISK

Action	Formats a DISK with DZFS.
Entry	HL = MEMORY address where the disk label is stored.
Exit	None
Destroys	A, BC, DE, HL, IX, IY, tmp_addr1, tmp_byte
Calls	F_KRN_SERIAL_WRSTR F_KRN_DZFS_CALC_SN F_KRN_RTC_GET_DATE F_BIOS_RTC_GET_TIME F_KRN_BCD_TO_ASCII F_KRN_BIN_TO_BCD4 F_KRN_BIN_TO_BCD6 F_KRN_DZFS_SECTOR_TO_SD F_KRN_SETMEMRNG F_BIOS_SERIAL_CONOUT_A F_BIOS_SD_PARK_DISKS F_BIOS_SD_MOUNT_DISKS

4.3.11 F_KRN_DZFS_CALC_SN

Action	Calculates the Serial Number (4 bytes) for a DISK .
Entry	IX = MEMORY address where the serial number will be stored.
Exit	None
Destroys	A, BC, DE, HL, IX
Calls	F_BIOS_RTC_GET_DATE F_BIOS_RTC_GET_TIME F_KRN_MULTIPLY816_SLOW

4.3.12 F_KRN_DZFS_SECTOR_TO_DISK

Action	Calls the BIOS subroutine that will store the data (512 bytes) currently in DISK Buffer in MEMORY , to the DISK .
Entry	DISK_cur_sector = the sector number in the DISK that will be written.
Exit	None
Destroys	BC, DE
Calls	F_BIOS_SD_WRITE_SEC

4.3.13 F_KRN_DZFS_GET_BAT_FREE_ENTRY

Action	Get number of available BAT entry.
Entry	None
Exit	DISK_cur_file_entry_number = entry number.
Destroys	A, IY, CF_cur_sector, CF_cur_file_entry_number
Calls	F_KRN_DZFS_READ_BAT_SECTOR F_KRN_DZFS_BATENTRY_TO_BUFFER

4.3.14 F_KRN_DZFS_ADD_BAT_ENTRY

Action	Adds a BAT entry into the DISK .
Entry	DE = BAT entry number. DISK_cur_sector = Sector number where the BAT Entry is in the DISK . DISK_BUFFER_START = Sector (512 bytes) containing the BAT where the entry is. DISK BAT = BAT Entry data that will be saved to DISK .
Exit	None
Destroys	A, BC, DE, HL
Calls	F_KRN_MULTIPLY816_SLOW

4.3.15 F_KRN_DZFS_CREATE_NEW_FILE

Action	Creates a new file (and its corresponding BAT Entry) in the DISK , from bytes stored in MEMORY .
Entry	HL = MEMORY address of the first byte to be stored. BC = number of bytes to be stored in the DISK . IX = MEMORY address where the filename is stored.
Exit	None
Destroys	A, BC, DE, HL, IX, tmp_addr1, tmp_addr2, tmp_addr3, tmp_byte
Calls	F_KRN_DZFS_GET_BAT_FREE_ENTRY F_KRN_DIV1616 F_KRN_MULTIPLY1616 F_KRN_COPYMEM512 F_KRN_CLEAR_MEMAREA F_KRN_CLEAR_DISKBUFFER F_KRN_DZFS_SECTOR_TO_SD F_BIOS_SD_BUSY_WAIT F_KRN_SERIAL_WRSTRCLR F_KRN_DZFS_CALC_FILETIME F_KRN_DZFS_CALC_FILEDATE F_KRN_DZFS_SEC_TO_BUFFER F_KRN_DZFS_ADD_BAT_ENTRY

4.3.16 F_KRN_DZFS_CALC_FILETIME

Action	Packs current Real-Time Clock time into two bytes, which is the format used to store times (created/modified) for files in the DISK . The formula used is: $2048 * hours + 32 * minutes + seconds/2$
Entry	None
Exit	HL = RTC time
Destroys	A, DE, HL
Calls	F_BIOS_RTC_GET_TIME

4.3.17 F_KRN_DZFS_CALC_FILEDATE

Action	Packs current Real-Time Clock date into two bytes, which is the format used to store dates (created/modified) for files in the DISK . The formula used is: $512 * (year - 2000) + month * 32 + day$
Entry	None
Exit	HL = RTC date
Destroys	A, DE, HL
Calls	F_BIOS_RTC_GET_DATE

4.3.18 F_KRN_DZFS_SHOW_DISKINFO_SHORT

Action	Outputs to the CONSOLE some information of the DISK : volume label, serial number, date/time creation.
Entry	None
Exit	None
Destroys	A, BC, DE, HL
Calls	F_KRN_SERIAL_WRSTRCLR F_KRN_SERIAL_PRN_BYTE F_KRN_SERIAL_PRN_BYTES F_BIOS_SERIAL_CONOUT_A F_KRN_SERIAL_EMPTYLINES

4.3.19 F_KRN_DZFS_SHOW_DISKINFO

Action	Outputs to the CONSOLE all information of the DISK : volume label, serial number, date/time creation, file system ID, number of partitions, number of bytes per sector, number of sectors per block.
Entry	None
Exit	None
Destroys	A, BC, DE, HL, tmp_addr1
Calls	F_KRN_DZFS_SHOW_DISKINFO_SHORT F_KRN_SERIAL_WRSTRCLR F_KRN_SERIAL_PRN_BYTE F_KRN_SERIAL_PRN_BYTES F_BIOS_SERIAL_CONOUT_A F_KRN_SERIAL_EMPTYLINES

4.3.20 F_KRN_DZFS_CHECK_FILE_EXISTS

Action	Checks if a specified filename exists in the DISK . The filename MUST be terminated by a zero.
Entry	HL = MEMORY address where the filename to check is stored.
Exit	Z Flag set if filename is not found.
Destroys	A, DE, tmp_addr3
Calls	F_KRN_DZFS_GET_FILE_BATENTRY

4.4 Math Routines**4.4.1 F_KRN_MULTIPLY816_SLOW**

Action	Multiplies an 8-bit number by a 16-bit number (HL = A * DE). It does a slow multiplication by adding the multiplier to itself as many times as multiplicand (e.g. 8 * 4 = 8+8+8+8).
Entry	A = Multiplicand DE = Multiplier
Exit	HL = Product
Destroys	B, HL
Calls	None

4.4.2 F_KRN_MULTIPLY1616

Action	Multiplies two 16-bit numbers (HL = HL * DE)
Entry	HL = Multiplicand DE = Multiplier
Exit	HL = Product
Destroys	A, BC, DE, HL
Calls	None

4.4.3 F_KRN_DIV1616

Action	Divides two 16-bit numbers (BC = BC / DE, HL = remainder)
Entry	BC = Dividend DE = Divisor
Exit	BC = Quotient HL = Remainder
Destroys	A, BC, HL
Calls	None

4.4.4 F_KRN_CRC16_INI

Action	Initialises the CRC to 0 and the polynomial to the appropriate bit pattern, to generate a CRC-16/BUYPASS1
Entry	None
Exit	MATH_CRC = 0 (initial CRC value) MATH_polynomial = CRC polynomial
Destroys	HL
Calls	None

CRC-16/BUYPASS1: A 16-bit cyclic redundancy check (CRC) based on the IBM Binary Synchronous Communications protocol (BSC or Bisync). It uses the polynomial $X^{16} + X^{15} + X^2 + 1$.

4.4.5 F_KRN_CRC16_GEN

Action	Combines the previous CRC with the CRC generated from the current data byte, to generate a CRC-16/BUYPASS1.
Entry	A = current data byte. MATH_CRC = previous CRC MATH_polynomial = CRC polynomial
Exit	MATH_CRC = CRC with current data byte included
Destroys	A, BC, DE, HL
Calls	None

4.5 String manipulation Routines**4.5.1 F_KRN_IS_PRINTABLE**

Action	Checks if a character is a printable ASCII character.
Entry	A = character to check.
Exit	C Flag is set if character is printable.
Destroys	None
Calls	None

4.5.2 F_KRN_IS_NUMERIC

Action	Checks if a character is numeric (0, 1, 2, 3, 4, 5, 6, 7, 8 or 9).
Entry	A = character to check.
Exit	C Flag is set if character is numeric.
Destroys	None
Calls	None

4.5.3 F_KRN_TOUPPER

Action	Converts a charcater to uppercase (e.g. <i>a</i> is converted to <i>A</i>).
Entry	A = character to convert.
Exit	A = uppercased character.
Destroys	None
Calls	None

4.5.4 F_KRN_STRCMP

Action	Compares two strings.
Entry	A = length of string 1. HL = MEMORY address where the first byte of string 1 is located. B = length of string 2. DE = MEMORY address where the first byte of string 2 is located.
Exit	if str1 = str 2, Z Flag set and C Flag not set. if str1 != str 2 and str1 longer than str2, Z Flag not set and C Flag not set. if str1 != str 2 and str1 shorter than str2, Z Flag not set and C Flag set.
Destroys	A, BC, DE, HL
Calls	None

4.5.5 F_KRN_STRCPY

Action	Copies <i>n</i> characters from string 1 to string 2.
Entry	HL = MEMORY address where the first byte of string 1 is located. DE = MEMORY address where the first byte of string 2 is located. B = number of characters to copy.
Exit	None
Destroys	A, DE, HL
Calls	None

4.5.6 F_KRN_STRLEN

Action	Gets the length of a string that is terminated with a specified character.
Entry	HL = MEMORY address where the first byte of the string is located. A = terminating character.
Exit	B = lenght of the string.
Destroys	BC, HL
Calls	None

4.5.7 F_KRN_STRLENMAX

Action	Gets the length of a string that is terminated with a specified character, but only check up to a maximum of characters.
Entry	HL = MEMORY address where the first byte of the string is located. A = terminating character. B = maximum length to be checked.
Exit	B = length of the string.
Destroys	BC, DE, HL
Calls	None

4.5.8 F_KRN_INSTR

Action	Locates the first occurrence of a character within a string.
Entry	HL = MEMORY address where the first byte of the string is located. B = character to search in string. D = terminating character.
Exit	E = position of character in string. Carry Flag = Set if character was found.
Destroys	A, C, E
Calls	None

4.5.9 F_KRN_STRCHR

Action	Finds the first occurrence of a character in a string terminated by a specified character.
Entry	HL = MEMORY address where the first byte of the string is located. D = terminating character. E = character to search in string.
Exit	HL = MEMORY address to the character found. Carry Flag = Set if character was found.
Destroys	A, HL
Calls	None

4.5.10 F_KRN_STRCHRNTNTH

Action	Finds the <i>nth</i> occurrence of a character in a string terminated by a specified character.
Entry	HL = MEMORY address where the first byte of the string is located. D = terminating character. E = character to search in string. B = occurrence number (<i>nth</i>).
Exit	HL = MEMORY address to the character found. Carry Flag = Set if character was found.
Destroys	A, B, HL
Calls	None

4.6 Conversion Routines

4.6.1 F_KRN_ASCIIADR_TO_HEX

Action	Convert an address (or any 2 bytes) from hex ASCII to its hexadecimal value (e.g. 32 35 37 30 are converted into 2570).
Entry	IX = MEMORY address where the first byte is located.
Exit	HL = hexadecimal converted value.
Destroys	HL
Calls	F_KRN_ASCII_TO_HEX

4.6.2 F_KRN_ASCII_TO_HEX

Action	Converts two ASCII characters (representing two hexadecimal digits) ; to one byte in hexadecimal (e.g. 0x33 and 0x45 are converted into 3E).
Entry	H = Most significant ASCII digit. L = Less significant ASCII digit.
Exit	A = Converted value.
Destroys	A, BC
Calls	None

4.6.3 F_KRN_HEX_TO_ASCII

Action	Converts one byte in hexadecimal to two ASCII printable characters (e.g. 0x3E is converted into 33 and 45, which are the ASCII values of 3 and E).
Entry	A = Byte to convert.
Exit	H = Most significant ASCII digit. L = Less significant ASCII digit.
Destroys	A, BC, HL
Calls	None

4.6.4 F_KRN_BCD_TO_BIN

Action	Converts a byte of BCD to a byte of hexadecimal (e.g. 12 is converted into 0x0C).
Entry	A = BCD.
Exit	A = Hexadecimal.
Destroys	A, BC
Calls	None

4.6.5 F_KRN_BIN_TO_BCD4

Action	Converts a byte of unsigned integer hexadecimal to 4-digit BCD (e.g. 0x80 is converted into 0128).
Entry	A = Unsigned integer to convert.
Exit	H = Hundreds digits. L = Tens digits.
Destroys	A, BC, HL
Calls	None

4.6.6 F_KRN_BIN_TO_BCD6

Action	Converts two bytes of unsigned integer hexadecimal to 6-digit BCD (e.g. 0xFFFF is converted into 065535).
Entry	HL = Unsigned integer to convert.
Exit	C = Thousands digits. D = Hundreds digits. E = Tens digits.
Destroys	A, BC, DE, HL
Calls	None

4.6.7 F_KRN_BCD_TO_ASCII

Action	Converts 6-digit BCD to hexadecimal ASCII string (e.g. 512 is converted into 30 30 30 35 31 32).
Entry	DE = MEMORY address where the converted string will be stored. C = first two digits of the 6-digit BCD to convert. H = next two digits of the 6-digit BCD to convert. L = last two digits of the 6-digit BCD to convert.
Exit	None
Destroys	A, DE
Calls	None

4.6.8 F_KRN_BITEXTRACT

Action	Extracts a group of bits from a byte and returns the group in the LSB position.
Entry	E = byte from where to extract bits. D = number of bits to extract. A = start extraction at bit number.
Exit	A = extracted group of bits
Destroys	A, BC, DE, HL
Calls	None

4.6.9 F_KRN_BIN_TO_ASCII

Action	Converts a 16-bit signed binary number (-32768 to 32767) to ASCII data (e.g. 32767 is converted into 33 32 37 36 37).
Entry	D = High byte of value to convert. E = Low byte of value to convert.
Exit	CLI_buffer_pgm = converted ASCII data. First byte us the length.
Destroys	A, BC, DE, HL, CLI_buffer_pgm
Calls	None

4.6.10 F_KRN_DEC_TO_BIN

Action	Converts an ASCII string consisting of the length of the number (in bytes), a possible ASCII - or + sign, and a series of ASCII digits to two bytes of binary data. Note that the length is an ordinary binary number, not an ASCII number. (e.g. 05 33 32 37 36 37 is converted into 7FFF).
Entry	HL = MEMORY address where the string to be converted is.
Exit	HL = converted bytes.
Destroys	A, BC, DE, HL, tmp_byte
Calls	None

4.6.11 F_KRN_PKEDDATE_TO_DMY

Action	Extracts day, month and year from a packed date (used by DZFS to store dates).
Entry	HL = packed date.
Exit	A = day. B = month. C = year.
Destroys	A, BC, HL, tmp_addr1
Calls	None

4.6.12 F_KRN_PKEDTIME_TO_HMS

Action	Extracts hour, minutes and seconds from a packed time (used by DZFS to store times).
Entry	HL = packed time.
Exit	A = hour. B = minutes. C = seconds.
Destroys	A, BC, HL, tmp_addr1
Calls	None

4.7 MEMORY Routines**4.7.1 F_KRN_SETMEMRNG**

Action	Sets (changes) a value in a MEMORY position range.
Entry	HL = MEMORY start position (first byte). BC = number of bytes to set. A = value to set.
Exit	None
Destroys	BC, HL
Calls	None

4.7.2 F_KRN_COPYMEM512

Action	Copies bytes from one area of MEMORY to another, in group of 512 bytes (i.e. max. 512 bytes). If less than 512 bytes are to be copied, the rest will be filled with zeros.
Entry	HL = MEMORY origin position (from where to copy the bytes). DE = MEMORY destination position (to where to copy the bytes). BC = number of bytes to copy (MUST be less or equal to 512).
Exit	None
Destroys	A, BC, DE, HL
Calls	None

4.7.3 F_KRN_SHIFT_BYTES_BY1

Action	Moves bytes (by one) to the right and replaces first byte with bytes counter.
Entry	HL = MEMORY address of last byte to move. BC = number of bytes to move.
Exit	None
Destroys	A, DE, HL
Calls	None

4.7.4 F_KRN_CLEAR_MEMAREA

Action	Clears (with zeros) a number of bytes, starting at a specified MEMORY address. Maximum 256 bytes can be cleared.
Entry	IX = MEMORY address of first byte to clear. B = number of bytes to clear.
Exit	None
Destroys	A, BC, IX
Calls	None

4.7.5 F_KRN_CLEAR_DISKBUFFER

Action	Clears (with zeros) the MEMORY area of the DISK buffer.
Entry	None
Exit	None
Destroys	BC, IX
Calls	F_KRN_CLEAR_MEMAREA

4.8 Real-Time Clock Routines**4.8.1 F_KRN_RTC_GET_DATE**

Action	Calls the BIOS function to get date from the RTC, and then calculates the year in four digits.
Entry	None
Exit	RTC_year4
Destroys	A, DE, HL
Calls	F_KRN_MULTIPLY816_SLOW

4.8.2 F_KRN_RTC_SHOW_TIME

Action	Sends to the Serial Channel A the values of hour, minutes and seconds from SYSVARS , as hh:mm:ss
Entry	None
Exit	None
Destroys	A, BC, DE, tmp_addr1
Calls	F_KRN_BIN_TO_BCD4 F_KRN_BCD_TO_ASCII F_BIOS_SERIAL_CONOUT_A

4.8.3 F_KRN_RTC_SHOW_DATE

Action	Sends to the Serial Channel A the values of day, month, year (4 digits) and day of the week (3 letters) from SYSVARS , as dd/mm/yyyy www
Entry	None
Exit	None
Destroys	A, BC, DE, tmp_addr1
Calls	F_KRN_BIN_TO_BCD4 F_KRN_BIN_TO_BCD6 F_KRN_BCD_TO_ASCII F_BIOS_SERIAL_CONOUT_A

4.8.4 F_KRN_RTC_SET_TIME

Action	Converts ASCII values to Hexadecimal, RTC_hour, RTC_minutes, RTC_seconds and calls the BIOS function to change time via ASMDC .
Entry	IX = MEMORY address where the new time is stored in ASCII format.
Exit	None
Destroys	A, HL, RTC_hour, RTC_minutes, RTC_seconds
Calls	F_KRN_ASCII_TO_HEX F_KRN_BCD_TO_BIN F_BIOS_RTC_SET_TIME

4.8.5 F_KRN_RTC_SET_DATE

Action	Converts ASCII values to Hexadecimal, RTC_year, RTC_month, RTC_day, RTC_day_of_the_week, and calls the BIOS function to change date via ASMDC .
Entry	IX = MEMORY address where the new date is stored in ASCII format.
Exit	None
Destroys	A, HL, RTC_year, RTC_month, RTC_day, RTC_day_of_the_week
Calls	F_KRN_ASCII_TO_HEX F_KRN_BCD_TO_BIN F_BIOS_RTC_SET_DATE

4.9 VDP Routines

4.9.1 F_KRN_VDP_WRSTR

Action	Displays a text in the VDP screen, starting at a specified XY position. The text MUST be a zero terminated string.
Entry	B = Cursor X (horizontal) start position. C = Cursor Y (vertical) start position. HL = RAM address of a zero terminated string.
Exit	None
Destroys	A, VDP_cursor_x, VDP_cursor_y, HL
Calls	F_BIOS_VDP_CHAROUT_ATXY

4.9.2 F_KRN_VDP_GET_CURSOR_ADDR

Action	Returns the VRAM address of a specific XY position on the screen.
Entry	B = Cursor X (horizontal) position. C = Cursor Y (vertical) position.
Exit	HL = VRAM address.
Destroys	A, B, DE, HL, IX
Calls	F_BIOS_VDP_CHAROUT_ATXY

4.9.3 F_KRN_VDP_CLEARSCREEN

Action	Clears the VDP screen.
Entry	None
Exit	None
Destroys	A, B, DE, HL
Calls	F_KRN_SERIAL_WRSTRCLR F_BIOS_VDP_SET_ADDR_WR F_BIOS_VDP_BYTE_TO_VRAM

4.9.4 F_KRN_VDP_CHG_COLOUR_FGBG

Action	Changes the Foreground and Background colours of the VDP screen. For <i>Text Mode</i> also sets the border colour to the same as the Background colour.
Entry	A = Foreground colour. B = Background colour.
Exit	None
Destroys	A, B
Calls	F_BIOS_VDP_SET_REGISTER

4.9.5 F_KRN_VDP_CHG_COLOUR_BORDER

Action	Changes the Border colour of the VDP screen, for screen modes other than <i>Text Mode</i> . In <i>Text Mode</i> the Border (backdrop) colour is the same as the Background colour.
Entry	B = Border colour.
Exit	None
Destroys	A
Calls	F_BIOS_VDP_SET_REGISTER

5 dastaZ80 File System (DZFS)

In summary, a file system is a layer of abstraction to store, retrieve and update a set of files.

A file system manages access to the data and the metadata of the files, and manages the available space of the device, dividing the storage area into units of storage and keeping a map of every storage unit of the device.

DZFS main goal is to be very simple to implement. As the free **MEMORY**³ of the dastaZ80 is about 48,096 bytes, it makes no sense to have files bigger than that, as will not fit. Therefore, DZFS defines that *a Block can store only a single file*.

dastaZ80 access the **DISK** via Logical Block Addressing (LBA), which is a particularly simple linear addressing schema, in which each sector is assigned a unique number rather than referring to a cylinder, head, and sector (CHS) to access the disk.

A typical LBA scheme uses a 28-bit value that allows up to 8.4 GB of data storage capacity. DZFS schema is as follows:

LBA 3	LBA 2	LBA 1	LBA 0
XXXX	XXXX XXXX	BBBB BBBB	BBSS SSSS

Where:

- S is Sector (6 bits)
- B is Block (10 bits)
- X not used (12 bits)

5.1 DZFS characteristics

- **Bytes per Sector:** 512
- **Sectors per Block:** 64
- **Bytes per Block:** 32,768 (64 * 512). This also defines the maximum size of a file and the BAT maximum size.
- **Bytes per BAT entry:** 32
- **BAT entries:** 1024 (32,768 / 32). This also defines the maximum number of files per **DISK**.
- **Maximum bytes per File:** 1 Block (32,768 bytes)
- **Maximum bytes per DISK:** 1024 Blocks (1 Block = 1 File) * 32,768 bytes per Block = 33,554,432 bytes (33.5 MB)

5.2 DISK anatomy

A **DISK** is divided into areas:

- **Superblock** = 512 bytes (1 Sector)
- **Block Allocation Table (BAT)** = 1 Block (64 Sectors = 32,768 bytes)
- **Data Area** = 1023 Blocks (65,472 Sectors = 33,521,664 bytes)

³Free **MEMORY** is the **RAM** that is not used by the OS, the System variables and the buffers, and hence available to use for the user and programs.

5.2.1 Superblock

The first 512 bytes on the **DISK** contain fundamental information about the geometry, and is used by the OS to know how to access every other information on the **DISK**. On IBM PC-compatibles, this is known as the *Master Boot Record* or *MBR* for short. In DZFS, it is called *Superblock*, as it is an orphan sector that doesn't belong to any block.

Offset	Length (bytes)	Description	Example
0x00	2	Signature. Used to check that this is a Superblock. Set to 0xABBA	AB BA
0x02	1	Not used	00
0x03	8	File System Identifier. ASCII values for human-readable. Padded with spaces.	DZFSV1
0x0B	4	Volume Serial Number	35 2A 15 F2
0x0F	1	Not used.	00
0x10	16	Volume Label. ASCII values. Padded with spaces.	dastaZ80 Main
0x20	8	Volume Date Creation. ASCII values (ddmmyyyy).	03102022
0x28	6	Volume Time Creation. ASCII values (hhmmss).	142232
0x2E	2	Bytes per Sector (in Hexadecimal little-endian)	00 02
0x30	1	Sectors per Block (in Hexadecimal)	40
0x31	1	Not used.	00
0x32 - 0x64	51	Copyright notice (ASCII value)	Copyright 2022David Asta The MIT License (MIT)
0x65 - 0x1FF	411	Not used (filled with 0x00)	00 00 00 00 00 00 00

5.2.2 Block Allocation Table (BAT)

The BAT is an area of 32,768 bytes (i.e. 1 Block) on the **DISK** used to store the details about the files saved in the *Data Area*, and is comprised of file descriptors called *entry*. Each entry holds information about a single file, and is 32 bytes in length.

For simplicity, each entry works also as index. The first entry describes the first file on the **DISK**, the second entry describes the second file, and so on.

Offset	Length (bytes)	Description	Example
0x00	14	Filename Padded with spaces at the end. (only allowed A to Z and 0 to 9. No spaces allowed. Cannot start with a number.) First character also indicates 00=available, 7E=deleted (will appear as ~)	46 49 4C 45 30 30 30 30 31 20 20 20 20 20
		Attributes (0=Inactive / 1=Active)	Read Only, System file, Executable = 1101 = 0D

Offset	Length (bytes)	Description	Example
		Bit 0 = Read Only Bit 1 = Hidden Bit 2 = System Bit 3 = Executable Bit 4-7 = File Type (see below)	
0x0F	2	Time created 5 bits for hour (binary number 0-23) 6 bits for minutes (binary number 0-59) 5 bits for seconds (binary number seconds / 2)	F5 9A
0x11	2	Date created 7 bits for year since 2000 (max. is year 2127) 4 bits for month (binary number 0-12) 5 bits for day (binary number 0-31)	69 1B
0x13	2	Time last modified (same formula as Time created)	F5 9A
0x15	2	Date last modified (same formula as Date created)	69 1B
0x17	2	File size in bytes (little-endian)	26 00
0x19	1	File size in sectors (little-endian)	01
0x1A	2	Entry number (little-endian)	00 00
0x1C	2	1st Sector (where the file data starts) It is calculated when the file is created. The formula is: $65 + 64 * \text{entry_number}$	41 00
0x1E	2	Load address (The start address little-endian where it will be loaded in RAM)	68 25

The value of the bits 4 to 7 of the *Attributes* field define the *File Type*:

Bits 4-7	File Type	Description
0x00	USR	User defined
0x01	EXE	Executable binary
0x02	BIN	Binary (non-executable) data
0x03	BAS	BASIC code
0x04	TXT	Plain ASCII Text file
0x05	SC1	Screen 1 (Graphics I Mode) Picture
0x06	FN6	Font (8x6) for Text Mode
0x07	SC2	Screen 2 (Graphics II Mode) Picture
0x08	FN8	Font (8x8) for Graphics Modes
0x09	SC3	Screen 3 (Multicolour Mode) Picture
0x0A		Not used
0x0B		Not used
0x0C		Not used
0x0D		Not used
0x0E		Not used
0x0F		Not used

5.2.3 Data Area

The Data Area is the area of the **DISK** used to store file data (e.g. programs, documents).

It is divided into Blocks of 64 Sectors each.

5.3 How Volume Serial Number is calculated

Calculated by combining the date and time at the point of format:

- first byte is calculated as follows:
 - day + milliseconds (converted to hexadecimal)
 - e.g. $3 + 50 = 53$ (0x35)
- second byte is calculated as follows:
 - month + seconds (converted to hexadecimal)
 - e.g. $10 + 32 = 42$ (0x2A)
- last two bytes are calculated as follows:
 - (hours [if pm + 12] * 256) + minutes + year (converted to hexadecimal)
 - e.g. $(2 + 12 = 14 * 256 = 3584) + 22 + 2012 = 5618$ (0x15 0xF2)

5.4 How Dates (creation/last modified) are calculated

Dates (day, month, 4-digit year) are converted into two bytes as follows:

- Remove century from year (e.g. $2013 - 2000 = 13$)
- Convert resulting number to hexadecimal (e.g. $13 = 0x0D$)
- Bitwise Shift Left 9 positions (e.g. $0x0D \ll 9 = 0x1A00$)
- Convert month to hexadecimal (e.g. $11 = 0x0B$)
- Bitwise Shift Left 5 positions (e.g. $0x0B \ll 5 = 0x0160$)
- Add converted month to converted year (e.g. $0x1A00 + 0x0160 = 0x1B60$)
- Convert day to hexadecimal (e.g. $9 = 0x09$)
- Add converted day to the sum of converted month and converted year (e.g. $0x1B60 + 0x09 = 0x1B69$)

5.5 How Times (creation/last modified) are calculated

Times (hours, minutes, seconds) are converted into two bytes as follows:

- Convert hours to hexadecimal (e.g. $19 = 0x13$)
-
- Bitwise Shift Left 3 positions (e.g. $0x13 \ll 3 = 0x98$)
- Convert minutes to hexadecimal (e.g. $23 = 0x17$)
- Bitwise Shift Left 5 positions (e.g. $0x17 \ll 5 = 0x02E0$)
- Logical OR most significant byte (MSB) of converted minutes with less significant byte (LSB) of converted hours (e.g. $0x02 \vee 0x98 = 0x9A$)
- Logical OR LSB of converted minutes with MSB of converted hours (e.g. $0xE0 \vee 0x00 = 0xE0$)
- Convert seconds to hexadecimal (e.g. $42 = 0x2A$)

- Divide the converted seconds by 2 (e.g. $0x2A / 2 = 0x15$)
- Add converted seconds to ORed converted hours and minutes (e.g. $0x9AE0 + 0x15 = 0x9AF5$)

5.6 Block Number, Sector Number and Addresses

To locate files in a Disk Image File it is useful to know how Blocks and Sector Numbers relate to the Address in the disk.

Given a Sector Number (*SecNum*), multiply it by the number of Bytes per Sector (512) to obtain the address where the data will start.

Below is provided a table for quick reference:

Block	SecNum	Address
0	1 (0x0000)	0x00000200
1	65 (0x0041)	0x00008200
2	129 (0x0081)	0x00010200
3	193 (0x00C1)	0x00018200
4	257 (0x0101)	0x00020200
5	321 (0x0141)	0x00028200
6	385 (0x0181)	0x00030200
7	449 (0x01C1)	0x00038200
8	513 (0x0201)	0x00040200
9	577 (0x0241)	0x00048200
10	641 (0x0281)	0x00050200
11	705 (0x02C1)	0x00058200
12	705 (0x0301)	0x00060200
13	833 (0x0341)	0x00068200
14	897 (0x0381)	0x00070200
15	961 (0x03C1)	0x00078200
16	1025 (0x0401)	0x00080200
17	1089 (0x0441)	0x00088200
18	1153 (0x0481)	0x00090200
19	1217 (0x04C1)	0x00098200
20	1281 (0x0501)	0x000A0200
21	1345 (0x0541)	0x000A8200
22	1409 (0x0581)	0x000B0200
23	1473 (0x05C1)	0x000B8200
...
1023	65473 (0xFFC1)	0x01FF8200

6 How To

6.1 Read data from DISK

Given `DISK_is_formatted` is equal to `0xFF` (i.e. **DISK** is formatted with DZFS file system), call `F_KRN_DZFS_LOAD_FILE_TO_RAM` with `DE` equal to first sector (512 bytes) to read and `IX` equal to how many sectors to read.

Read bytes will be copied into **MEMORY**, following these rules:

- if `DISK_loadsave_addr` \neq 0, load bytes to this address.
- if `DISK_loadsave_addr` = 0,
 - if `DISK_cur_file_load_addr` \neq 0, load bytes to this address.
 - if `DISK_cur_file_load_addr` = 0, load bytes to start of **Free RAM** (`0x4420`).

6.2 Write data to DISK

Given `DISK_is_formatted` is equal to `0xFF` (i.e. **DISK** is formatted with DZFS file system):

- Store the filename (in ASCII) somewhere in **MEMORY**.
- call `F_KRN_DZFS_GET_FILE_BATENTRY`, with `HL` equal to the **MEMORY** address where the filename is stored. If a file with the specified filename does not exist, flag `z` will be set to indicate that it is OK to save the file.
- call `F_KRN_DZFS_CREATE_NEW_FILE`, with:
 - `HL` equal to the address in **MEMORY** of first byte to be stored.
 - `BC` equal to the total number of bytes to be stored.
 - `IX` equal to the address in **MEMORY** where the filename is stored.
 - `DISK_loadsave_addr` equal to:
 - * zero, will use the address in **MEMORY** of first byte as the load address when loading the file (i.e. `DISK_loadsave_addr`).
 - * non zero, will use this number as the load address when loading the file (i.e. `DISK_loadsave_addr`).

6.3 Convert between HEX and DEC and ASCII

In many situations your programs will need to convert between different number notations (hexadecimal, decimal, ASCII). For example, all characters typed by the user are read by the function `F_BIOS_SERIAL_CONIN_A`, which stores the ASCII value of the pressed key in the `A` register. In order to do manipulations of data, our program will need to convert this ASCII data into either hexadecimal or decimal notation.

Take as an example the CLI command for saving files to disk (*save*). As shown in the *dastaZ80 User's Manual* section 5.3 *Disk Commands*, this command takes two parameters: `<start_address>`, which is expressed in hexadecimal, and `<number_of_bytes>`, which is expressed in decimal. But in both cases, `F_BIOS_SERIAL_CONIN_A` will give us (in the `A` register) the ASCII representation of the numbers typed by the user.

Before we can set a pointer to the memory address specified by `<start_address>`, and set our counter to `<number_of_bytes>`, we need to convert those ASCII numbers into hexadecimal and decimal respectively.

The Kernel, offers a series of functions to help the programmer with the conversions:

- `F_KRN_ASCIIADR_TO_HEX`: Converts ASCII 4 chars to HEX 2 bytes. (e.g. 32 35 37 30 to `0x2570`)

- [F_KRN_ASCII_TO_HEX](#): Converts ASCII 2 chars to HEX 1 byte. (e.g. 33 45 to 0x3E)
- [KRN_HEX_TO_ASCII](#): Converts HEX 1 byte to ASCII 2 chars. (e.g. 0x3E to 33 45)
- [F_KRN_BCD_TO_BIN](#): Converts a byte of BCD to a byte of hexadecimal. (e.g. 12 is converted into 0x0C).
- [F_KRN_BIN_TO_BCD4](#): Converts HEX 1 byte to DEC 4 digits. (e.g. 0x80 to 0128)
- [F_KRN_BIN_TO_BCD6](#): Converts HEX 2 bytes to DEC 6 digits. (e.g. 0xFFF to 065535)
- [F_KRN_BCD_TO_ASCII](#): Converts DEC 6 digits to ASCII 6 chars. (e.g. 512 to 30 30 30 35 31 32)
- [F_KRN_BIN_TO_ASCII](#): Converts HEX 2 bytes to ASCII string. (e.g. 0x7FFF to 33 32 37 36 37)
- [F_KRN_DEC_TO_BIN](#): Converts HEX n bytes to ASCII string. First byte tells the number of bytes to convert (e.g. 05 33 32 37 36 37 to 0x7FFF)

6.4 How to display Sprites

A *sprite* is a two-dimensional bitmap that can be made to move and change shape in the screen with very little programming effort, thanks to the **VDP** support for hardware sprites.

The **VDP** has 32 sprite planes each of which can contain a single sprite.

Sprites can be of two sizes; 8x8 pixels (Size 0) or 16x16 (Size 1) pixels. There is also the possibility to magnify a sprite, thus a 8x8 sprite becomes 16x16, and a 16x16 sprite becomes 32x32. Unfortunately, the sprite resolution is cut in half.

Two tables are required in **VRAM** in order to display a sprite; the *Sprite Attribute Table* and the *Sprite Pattern Table*.

The address of these tables will change depending on which mode we are using. Refer to the [VDP Memory Map](#) to know the addresses.

The *Sprite Pattern Table* defines the shape of each sprite. It takes 8 bytes to define the pattern of Size 0 (8x8) sprite and 32 bytes for a Size 1 (16x16) sprite. The table has a maximum length of 2048 bytes, therefore a maximum of 256 patterns can be defined for Size 0 and 64 for Size 1 sprites.

The *Sprite Attribute Table* contains four bytes of information for every sprite. The table is ordered sequentially, where the first four bytes contain the information for sprite 0, the next 4 bytes contain the information for sprite 1, and so on.

The information of the four bytes in the *Sprite Attribute Table* is as follows:

- Vertical coordinate: determines the distance (in pixels) the sprite will be offset from the top of the screen. The position is measured relative to the upper left hand corner of the sprite. A value of 0xFF will put the sprite at the top of the screen. A value of 0xBF will put the sprite at the bottom of the screen. But because the position is measured relative to the upper left hand corner of the sprite, it will not appear.
- Horizontal coordinate: determines the distance (in pixels) the sprite will be offset from the left hand side of the screen. A value of 0x00 will put the sprite at the left hand side of the screen. A value of 0xFF will put the sprite at the right hand side of the screen. But because the position is measured relative to the upper left hand corner of the sprite, it will not appear.
- Sprite Name Pointer: the value in this byte determines which pattern from the *Sprite Pattern Table* will be used as the sprite's shape. This highly simplify the production of sprite animations, as just the pointer needs to be changed.
- Colour and Early Clock Bit:

- The lower nibble define the sprite colour, which can be any of the [VDP Composite colours](#).
- The MSB of the higher nibble is called the *Early Clock Bit*, and when set as 1 it shifts the position of the sprite to the left by 32 pixels.

6.4.1 Example

Lets assume we are working in mode 2 (Graphics II Mode). Following the [VDP Memory Map](#), we can see that the Sprite Pattern Table is located at 0x1800 and the Sprite Attribute Table at 0x3B00.

First we will fill the patterns of Sprite 0 with the 8x8 sprite from the *Video Display Processors Programmer's Guide*[\[4\]](#). Hence we need to assign the following values:

- 0x1800 = 0x10
- 0x1801 = 0x10
- 0x1802 = 0xFE
- 0x1803 = 0x7C
- 0x1804 = 0x38
- 0x1805 = 0x6C
- 0x1806 = 0x44
- 0x1807 = 0x00

At this point, most probably (if just started the computer) we won't see anything yet, beacuse the colour is set to 0x00 (Transparent).

Lets change the colour byte for sprite 0 in the Sprite Attribute Table (0x3B03) to 0x03 (Light Green).

You should be seeing a little green star at the top left of the screen.

By changing the bytes corresponding to the Y position 0x3B00 and X position 0x3B01, we can move the sprite around the screen. Lets try for example 0x3B00 = 0x5F and 0x3B01 = 0x7F to display the sprite at the center of the screen.

This can be easily tested from the command line, by using the command *vpoke* to change the bytes. For example, *vpoke 1800,10*.

6.5 Develop software for dzOS

6.5.1 Available RAM

Programs can be loaded from disk to any area of **RAM**. Nevertheless, addresses below 0x4420 SHOULD not be used, at these contain the Operating System's variables. Modifying these without the proper care will result in undesired behaviour, system crash or even lost of data on the disk. Therefore, taking in consideration that the free RAM area starts at 0x4420 and ends at 0xFFFF, the programmer can load programs of maximum 48,095 bytes (48 KB).

6.5.2 Storing your variables

Variables for programs can be store anywhere in the free **RAM** space.

The OS is having its own internal variables that can be accessed by the user. Also, some variables are used only by CLI and therefore could be re-used during the execution of a program.

Refer to the section [System Variables \(SYSVARS\)](#) on this guide to know the exact locations.

- The **DISK Superblock** and **DISK BAT** areas can be re-used if you are not using **DISK** routines.
- The **CLI** area can safely be re-used in your program, as the CLI is not running meanwhile your program is.
- The **RTC** area can be re-used if you are not calling any **RTC** routines.
- The **Math** area can be re-used if you are not calling any **Math** routines.
- The **SIO**, **Generic** and **VDP** areas MUST not be touched.

All in all, you may end up having some extra 700 bytes here.

6.5.3 Receiving parameters from CLI

When a user types a command in CLI, the entered command is stored in an area of 64 bytes in the [System Variables \(SYSVARS\)](#) called *CLI_buffer_full_cmd*. From there, you can read the full command, which will be the name of your binary program, and the parameter or parameters.

6.5.4 Returning to CLI

If your program allows the user to return to CLI, it must then jump to the loop subroutine known as (CLI Prompt). The address of this subroutine is stored in the [System Variables \(SYSVARS\)](#) *CLI_prompt_addr*.

Simply make your program to load the value stored at that location and jump (*jp*) to it.

6.5.5 Developing with Z80 Assembler

In order for dzOS to know where to load the program in **RAM**, the executable code must provide the load address. For compatibility with SDCC ⁴, we will store it in the bytes 3 and 4 of the executable.

For programs developed in Z80 Assembler, add the following at the top of the source code:

```
.ORG      $4420                ; start of code at
                                ; start of free RAM
jp        $4425                ; first instruction
                                ; must jump to the
                                ; executable code
.BYTE     $20 , $44            ; load address
                                ; (values must be
                                ; same as .org above)

.ORG      $4425                ; start of program
                                ; (must be same as jp above)

; your program here
```

The first *.ORG* (*.ORG \$4420*) indicates the start address used for creating the binary file after compilation.

0x4420 is where the [Free RAM](#) starts, giving you 48 KB for your program. Programs SHOULD not be loaded at a lower address, for the reason explained before.

The first instruction MUST be a jump (*jp*) instruction to the actual executable code (i.e. your program code) The *.BYTE* instruction just inserts the two bytes after the jump instruction. The values MUST be in hexadecimal little-endian format.

⁴Small Device C Compiler (SDCC) is a retargettable, optimizing Standard C (ANSI C89, ISO C99, ISO C11) compiler suite that targets (amongst others) the Zilog Z80 based MCUs. (<http://sdcc.sourceforge.net/>)

Because the `jp` instruction in Z80 is translated as *C3 nn nn* (where *nn* are the bytes where to jump), this will use the first three bytes (0x00, 0x01, 0x02) in the binary, therefore we store the load address at bytes 3 and 4 and your program can start just after, at byte 0x05.

Once assembled, the binary will be loaded by dzOS at the load address, and when executed, the first thing that will happen is a *jp* instruction and then the execution will continue from the executable code of your program.

If your program allows the user to return to CLI, add the following on your source code:

```
ld      HL, (CLI_prompt_addr)    ; return control
jp      (HL)                    ; to CLI
```

For convenience, two files are provided in the Github repository ⁵: *_header.inc* and *_template.asm*

6.5.6 Developing with SDCC

In the Github repository, there is a file (*crt0.s*) that sets:

- the start address for the binary at 0x4420
- the values 0x20 and 0x44 in the binary at bytes 5 and 6.
- first instruction of your program to be started located at 0x4425

Therefore, by using this file all programs will be loaded at the correct address.

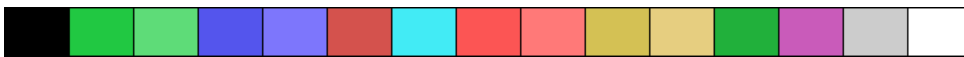
⁵<https://github.com/dasta400/dzSoftware>

7 Appendixes

7.1 ANSI Terminal colours

- `ANSI_COLR_BLK` - Black
- `ANSI_COLR_RED` - Red
- `ANSI_COLR_GRN` - Green
- `ANSI_COLR_YLW` - Yellow
- `ANSI_COLR_BLU` - Blue
- `ANSI_COLR_MGT` - Magenta
- `ANSI_COLR_CYA` - Cyan
- `ANSI_COLR_WHT` - White

7.2 VDP Composite colours



- `VDP_COLR_TRNSP` (Transparent) = \$00
- `VDP_COLR_BLACK` (Black) = \$01
- `VDP_COLR_M_GRN` (Medium Green) = \$02
- `VDP_COLR_L_GRN` (Light Green) = \$03
- `VDP_COLR_D_BLU` (Dark Blue) = \$04
- `VDP_COLR_L_BLU` (Light Blue) = \$05
- `VDP_COLR_D_RED` (Dark Red) = \$06
- `VDP_COLR_CYAN` (Cyan) = \$07
- `VDP_COLR_M_RED` (Medium Red) = \$08
- `VDP_COLR_L_RED` (Light Red) = \$09
- `VDP_COLR_D_YLW` (Dark Yellow) = \$0A
- `VDP_COLR_L_YLW` (Light Yellow) = \$0B
- `VDP_COLR_D_GRN` (Dark Green) = \$0C
- `VDP_COLR_MGNTA` (Magenta) = \$0D
- `VDP_COLR_GREY` (Grey) = \$0E
- `VDP_COLR_WHITE` (White) = \$0F

7.3 VDP Screen resolutions

7.3.1 Mode 0: Text Mode

- Screen is divided into 960 pattern positions each of which is capable of displaying a character. There are 40 characters in each row and 24 rows on the screen.
- Each character is 8x6 pixels.

- Each character can have 2 colours (Foreground and Background).
- Sprites cannot be used.
- The Pattern Table starts at **VRAM** address 0x0000, for a length of 2048 bytes (from 0x0000 to 0x07FFF).
 - This table contains the character sets, for a maximum of 256 characters per set.
 - Up to 7 different character sets can be held in the **VRAM** at the same time. Each set **MUST** be located starting at an 0x0800 boundary (i.e. 0x0000, 0x1000, 0x1800, 0x2000, 0x2800, 0x3000 and 0x3800). Note that 0x0800 is not listed because that address is used by the Name Table.
 - Ideally, the patterns follow the ASCII table definitions and order, so that the Name Table can be easily used to display text by for example assigning the value 0x41 to a byte in the Name Table to display the character *A*.
- The Name Table starts at **VRAM** address 0x0800. for a length of 960 bytes (from 0x0800 to 0x0BBF).
 - Each entry in the table is 1 byte long and therefore can specify one of 256 patterns (from 0x00 to 0xFF).
 - Each entry represents a pattern position on the screen. Position 0 is in the top left of the screen. Position 39 is in the top right of the screen. The second row ranges from 40 to 79, and so on.

7.3.2 Mode 1: Graphics I Mode

- Screen is divided into 768 blocks of 8x8 pixels each. There are 32 blocks in a row and 24 rows on the screen.
- Sprites can be used.
- The Name Table starts at **VRAM** address 0x1400.
 - This table has 768 entries, one for each block on the screen.
 - If the Pattern Table is loaded with with a full ASCII character set, the entry of any ASCII value in the Name Table will result in the corresponding character being displayed on the screen.
- The Pattern Table starts at **VRAM** address 0x0800.
- The Colour Table starts at **VRAM** address 0x2000.
 - This table has 32 entries, each entry defining 2 colours (Foreground and Background) out of 15 colours available, for a block of 8 characters. In other words, colours cannot be assigned independently to each character in the screen, but instead to groups of 8 consecutive characters.

7.3.3 Mode 2: Graphics II Mode

- Also known as **Bitmap Mode**.
- Screen is divided into 768 blocks of 8x8 pixels each. There are 32 blocks in a row and 24 rows on the screen.
- Sprites can be used.
- The Name Table starts at **VRAM** address 0x3800.
 - This table is divided into three subtables of 256 each.

- The Pattern Table starts at **VRAM** address 0x0000.
- The Colour Table starts at **VRAM** address 0x2000.
 - Each entry in the Colour Table is 8 bytes and each byte defines the 2 colours (Foreground and Background) of each of the 8 rows of the character, from a total of 15 colours plus transparent available.

7.3.4 Mode 3: Multicolour Mode

- Screen is divided into 768 blocks of 2x2 squares. Each square is 4 pixels. There are 32 blocks in each row and 4 rows in each section. There are 6 sections, for a total of 24 rows on the screen.
- Blocks are arranged in columns with 4 blocks in each column.
- Columns are arranged in sections, with 32 columns in each section.
- There are a total of 6 sections on the screen.
- In summary:
 - 32 columns * 6 sections = 192 columns
 - 192 columns * 4 blocks = 768 blocks
- No characters for text can be used.
- Sprites can be used.
- The Name Table starts at **VRAM** address 0x1400.
- The Colour Table is not used. Instead, the colour of the boxes are defined in the Pattern Table.
- The Pattern Table starts at **VRAM** address 0x0800.
 - Each entry in the table is 8 bytes, but only 2 bytes are used to define the colours of the 4 boxes that make up a character.

7.3.5 Mode 4: Graphics II Mode Bitmapped

- Same as Mode 2, but screen is bitmapped for addressing every pixel individually.

7.4 VDP Limitations

The maximum resolutions are: 240x192 pixels in Text Mode, 256x192 pixels in Graphics Modes (I, II, II Bit-mapped), and 512x384 in Multicolour Mode.

The maximum number of colours is 15 plus a transparent colour.

In Graphics I Mode, each entry in the Colour Table defines the colour for a group of eight patterns. Hence, individual character colouring is not possible.

In Graphics II Bit-mapped Mode, individual pixels can be addressed but individual colours cannot. Therefore it is not possible to assign different colours for each pixel.

Bug?: After some tests, and confirmed with some information found on the Internet, reading continuously the Status Register can lead to miss the flag. This happens when the register is read and the VDP is about to set it, because as specified in the *Video Display Processors Programmer's Guide*[4], *the Status Register is reset after it's read*. Therefore, the subroutine implemented in dzOS for waiting for the VBLANK (*BIOS_VDP_VBLANK_WAIT*) that initially was reading the **VDP's** Status Register and looping until the MSB changed to 1, has been changed to instead check for a change on the [Jiffy Counter](#).

7.4.1 Sprites

A maximum of 32 sprites can be shown on the screen, of sizes either 8x8 or 16x16 pixels. Though sprites can be magnified, thus showing as 16x16 or 32x32 respectively.

The location of a sprite is defined by the top left-hand corner of the sprite pattern.

When more than one sprite is located at the same screen coordinate, the sprite on the higher priority plane will be shown.

A maximum of 4 sprites can be displayed on the same horizontal line. If this rule is violated, the four highest priority sprites on the line are displayed normally, but the fifth and subsequent sprites are not displayed.

The *Coincidence Flag* (collision detection) only indicates that any two sprites have overlapping bits, but it does not tell which sprites are. This must be calculated programatically.

7.5 Jiffy Counter

A *Jiffy* is the time between two ticks of the system timer interrupt. On the dastaZ80, this timer is generated by the TMS9918A (**VDP**) at roughly each 1/60th second.

The counter is made of 3 bytes. Byte 1 is incremented in each **VDP** interrupt. Once it rolls over to zero (256 increments), the byte 2 is incremented. Once the byte 2 rolls over, the byte 3 is incremented. Once the three bytes together (24-bit) reach the value 0x4F1A00, the three bytes are initialised to zero.

0x4F1A00 (5,184,000 in decimal) is the number of jiffies in 24 hours: 24 hours x 60 minutes in an hour x 60 seconds in a minute x 60 jiffies in a second.

IMPORTANT: This counter **MUST** not be interpreted as an accurate clock, because when transferring data to the **VRAM** the OS disables the [NMI](#)⁶, and therefore the counter stops for a while.

7.6 OS Boot Sequence

After power on or after pressing the **RESET** button:

- **Bootstrap**

- Copy contents of the ROM into High RAM (0x8000 - 0xFFFF).
- Disable ROM chip and enable Low RAM (0x0000 - 0x7FFF). Therefore, all **MEMORY** is RAM from now on.
- Copy the copy of ROM in High RAM to Low RAM. Bootstrap code is not copied.
- Transfer control to BIOS (jp F_BIOS_SERIAL_INIT).

- **Initialise SIO/2** (F_BIOS_SERIAL_INIT)

- Initialise SIO/2.
 - * Set Channel A as 115,000 bps, 8N1, Interrupt in all received characters.
 - * Set Channel B as 115,000 bps, 8N1, Interrupt in all received characters.
 - * Set Interrupt Vector to 0x60.
- Set CPU to Interrupt Mode 2.
- jp F_BIOS_WBOOT

⁶It is also highly recommended that in your programs you also disable the [NMI](#) when copying large amounts of data. Otherwise, the process will be interrupted 60 times per second, and therefore slow it down.

- **BIOS Boot** (F_BIOS_WBOOT)
 - Set SIO/2 Channel A as primary I/O.
 - Transfer control to Kernel (jp F_KRN_START).
- **Kernel Boot** (F_KRN_START)
 - Display dzOS welcome message.
 - Display dzOS release version.
 - Display Kernel version.
 - Display available **RAM**.
 - Initialise **VDP**.
 - * Test write/read **VRAM**.
 - * Set **Low Resolution Display** as *Graphics II Bit-mapped Mode*.
 - * Show dastaZ80 Logo in the **Low Resolution Display**.
 - Initialise **PSG**.
 - * Set Noise OFF, Audio OFF, I/O Port as Output.
 - * Make a beep.
 - Initialise **FDD**.
 - Initialise **SD Card**.
 - * Detect **SD Card**.
 - * Display number of available Disk Image Files.
 - * Display disk unit and name of each Disk Image File.
 - Initialise **Real-Time Clock (RTC)**.
 - * Display current date and time.
 - * Display **RTC**'s battery status.
 - Initialise **SYSVARS**.
 - * Set show deleted files with *cat* command as OFF.
 - * Set default File Type as 0 (USR = User defined).
 - * Set default loadsave address to 0x0000 (i.e. will save/load starting from **Free RAM** (0x4420)).
 - Set default **DISK** as 1 (i.e. first Disk Image File in the **SD card**).
 - Transfer control to Command-line Interpreter (CLI) (jp F_CLI_START).
- **CLI** (F_CLI_START)
 - Display CLI version.
 - Clear command buffers
 - Display prompt (>).
 - Read command entered by user.

- Parse command.
- Execute corresponding subroutine.
- Loop back to Display prompt.

7.7 dzOS Programming Style

When writing dzOS and software for dzOS, the following style has been followed:

- All CPU registers are written in uppercase (e.g. *A*, *BC*, *HL*, *IX*, *SP*).
- All CPU flags are written in lowercase (e.g. *z*, *nz*, *c*, *nc*, *m*, *p*).
- All assembly mnemonics are written in lowercase (e.g. *ld A,0*).
- Labels for subroutines that will be public (i.e. called via a Jumpblock) are written in uppercase.
- Labels are written in a line, with no mnemonics.
- Public subroutines contain comments specifying:
 - Short description.
 - Input CPU registers or variables ([SYSVARS](#)).
 - Output CPU registers or variables ([SYSVARS](#)).
- All hexadecimal values are written with a dollar sign as prefix.
- Tabulation (Tabs) are written as 4 spaces.
- Mnemonics start after 2 tabs (8 spaces).
- When possible, comments are written in column 41. Otherwise in next closest Tab.
- Source code is heavily commented. Mostly on each line.
- *The Telemark Assembler* (TASM) specific:
 - *.BYTE* is used instead of *.DB*
 - *.WORD* is used instead of *.DW*

References

- [1] David Asta. *dastaZ80 User's Manual*, 2023.
- [2] David Asta. *dastaZ80 Technical Reference Manual*, 2023.
- [3] David Asta. dzos github repository. <https://github.com/dasta400/dzOS>, 2022.
- [4] Texas Instruments. *Video Display Processors Programmer's Guide*.