MSX

Top Secret

3

THE FINAL COMPILATION OF MSX INFORMATION

1st Edition – 23 oct 2022

Edison Moraes [2019-2022]

AUTHOR'S NOTE

After the release of MSX Top Secret 2, in April 2004, I figured that there would be no need to update it anymore, since MSX is no longer commercially manufactured by large companies. In addition, the internet has evolved and a wide range of information has become available to everyone.

However, the information is sparse, leading to the need for multiple and tiring searches not always achieving complete success. That's why I thought it convenient to write this third – and final – edition of MSX Top Secret, gathering all the information I could find in only one place.

Good research!

Edison Antonio Pires de Moraes (author)

Sorry for my english mistakes. I'm not fluent in english.

Suggestions and information about errors are welcome.

Send it to:

eapmoraes2012@gmail.com

Summary

Chapter 1
INTRODUCTION TO THE SYSTEM

The MSX system was created in 1983 and officially announced on June 27 of that same year by Microsoft, holder of the standard at the time. MSX was created with open architecture, and any company can manufacture it without having to pay royalties. –

The specifications provided that all MSX computers would be compatible at strategic points, and that all versions that were created later would maintain compatibility with the original standard.

The MSX was commercially manufactured until the fourth version, the MSX turbo R, and in practice the compatibility has been maintained. In fact, there are always small changes due to technological development or the non-use of certain resources by programmers and users in general. Thus, from MSX1 to MSX2, the memory expansion in slots, which is complicated to handle, was replaced by an expansion called Memory Mapper. Its VDP (video processor) was also replaced by a much more powerful one, in addition to having several standard optional peripherals, such as MSX Audio, for example. From MSX2 to MSX2+ the main memory started to consist of the first 64 Kbytes of Memory Mapper, thus saving a slot, in addition to having some VDP functions changed. From the MSX2+ to the MSX turbo R, the changes were more radical: the cassette interface was eliminated, which had become completely obsolete, and a new 16-bit CPU, the R800, was introduced, fully compatible with the Z80, but very faster than this.

So, despite these small changes that would theoretically destroy compatibility, in practice the MSX turbo R is compatible with all previous models. The table on the next page illustrates the main features and differences between the four officially released versions of MSX:

	MSX1 June/83	MSX2 May/86	MSX2+ October/88	MSX turbo R October/90
CPU	Z80 3,58MHz	Z80 3,58MHz	Z80 3,58MHz	Z80 3,58MHz R800 7,16MHz
Min RAM	8 Kbytes	64 Kbytes	64 Kbytes	256 Kbytes
Max RAM	1 Mbyte *	64 Mbytes *	64 Mbytes *	64 Mbytes *
VRAM	16 Kbytes	64/128 Kb	128 Kbytes	128 Kbytes

VDP	TMS9918	V9938	V9958	V9958
Standard ROM	32K Main	32K Main 16K SubROM	32K Main 32K SubROM 16K DOS1	32K Main 48K SubROM 16K DOS1 48K DOS2
CAS interface	Standard	Standard	Standard	No
Printer interface	Optional	Standard	Standard	Standard
External Slots	1 ou 2	2	2	2
PSG	Standard	Standard	Standard	Standard
MSX Audio	No	Optional	Optional	Optional
FM Sound	No	Optional	Standard	Standard
PCM	No	Não	Não	Standard
Disk Drive	Optional	Optional	Standard 3½ DD	Standard 3½ DD
MSX Basic	Version 1.0	Version 2.0	Version 3.0	Version 4.0
MSX-DOS	Version 1.0	Version 1.0 2.0 Optional	Version 1.0 2.0 Optional	Version 1.0 2.0 standard

Maximum RAM refers to the maximum theoretical that could be connected to the CPU. In fact, in the case of the MSX1, only 64K of RAM can be plugged into each slot and as there are 16 slots in total, this results in a maximum theoretical of 1 Mbyte. In the case of MSX2 onwards, up to 4 Mbytes can be connected to each slot, through an expansion called Memory Mapper, resulting in a maximum theoretical of 64 Mbytes. However, most Mapper software will only be able to use 4 Mbytes linearly in a single slot (256 logical pages), although some of them will recognize Mapper in different slots.

1.1 – INTERNAL ARCHITECTURE

In MSX1, there are 4 central chips that perform specific functions on the PC. These chips are designated by the acronyms CPU, VDP, PSG and PPI. As each one, with the exception of the PPI, has a certain processing capacity, MSX is said to have multi-processing, a theory that is often contested.

1.1.1 – The CPU

The CPU (Center Procedure Unit) is responsible for executing the programs. The chip used is Zilog's veteran Z80A, clocked at 3.58 MHz. The processing capacity of this chip, however, proved to be insufficient.

Therefore, MSX turbocharged at 6 or 7 MHz appeared and from MSX2+ there was no longer a fixed clock requirement of 3.58 MHz in the basic specifications. Later, a specific CPU for the MSX was designed, the R800, 16 bits, which was much faster than the Z80A and which was used in the MSX turbo R models.

1.1.1.1 – Wait States

In MSX, a wait state is generated during the execution of the M1 cycle. Thus, if a Z80 instruction has 2 M cycles, 2 wait states will be generated during its execution.

1.1.2 – The VDP

The VDP (Video Display Processor) is the chip responsible for video processing. It is described in detail in Chapter 5 (Video and VDP). The VDP used on the MSX1 is the TMS9918A, from Texas Instruments, which was soon replaced by the V9938, specific to the MSX2, much more powerful. After, was developed the V9958, with some additional functions. But the V9958 proved too slow and limited. Yamaha started the development of a much more powerful chip, tentatively named V9978. However, it was never produced. Later, a chip with the nomenclature V9990 was released, but it was not fully compatible with the V9958. The V9990 was not used in any official MSX model, being used only in cartridge. All these chips are described with details in the Chapter 5 (Video and VDP).

1.1.3 – PSG

The PSG (Programmable Sound Generator) is the chip responsible for generating sounds. It also controls the joystick ports. The chip used is the AY-3-8910A, from General Instruments. Because it was very limited, alternatives soon emerged. The first was MSX-Audio (Y8950), which did not become popular. Then came the OPLL (YM2413), this one is quite popular, with a lot of software. In the same way came the SCC and then the PCM in the MSX turbo R models. A sound cartridge, the Moonsound, used the OPL4 (YMF278B), a very advanced chip with high sound quality. A rather inexpensive alternative was the Covox, which was plugged in the printer port. All these generators are described with details in the Chapter 6 (Audio Generators).

1.1.4 – The PPI

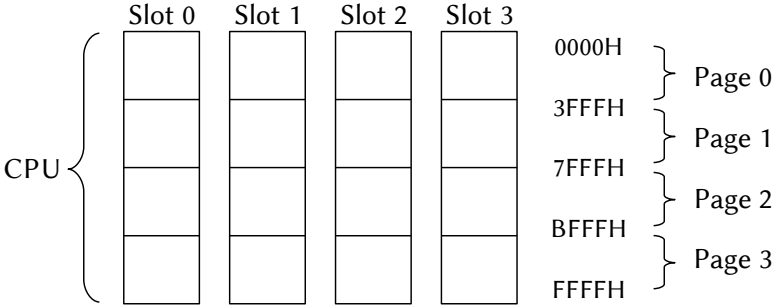
The PPI (Peripheral Programmable Interface) is the "touchstone" of MSX. Is it that makes MSX so flexible. Its two main tasks are keyboard control and the selection of memory slots and pages. The chip responsible is Intel's 8255A. Despite being built into the MSX Engine in the latest models, it works identically to the MSX1. The layout of pages and slots, controlled by the PPI, is described in detail in the chapter 2 (Slots and cartridges) and the keyboard interface is described in the chapter 8, item 2 (Keyboard interface).

Chapter 2
SLOTS AND CARTRIDGES

The Z80A CPU, which is used in MSX computers, is capable of directly addressing only 64 Kbytes of memory. On MSX computers, however, using the slots and pages technique, the Z80A can access up to 1 megabyte of memory, non-linearly.

2.1 – SLOTS

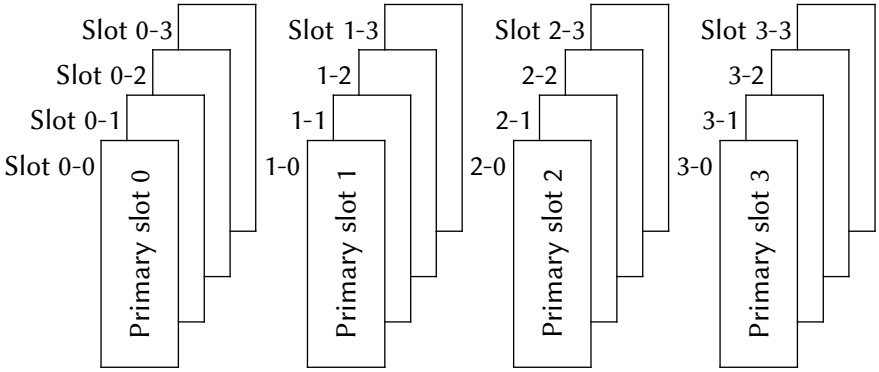
There are two types of slots: primary slots and secondary slots. The primary slots are four in number and are directly connected to the CPU. Each slot is divided into four parts of 16 Kbytes, making up the 64 Kbytes of the Z80's address space. These parts are called "pages". A page of the same number always occupies the same CPU address space, so only four pages of different numbers can be active at the same time, albeit in different slots. The illustration below shows how the slots and pages are arranged.



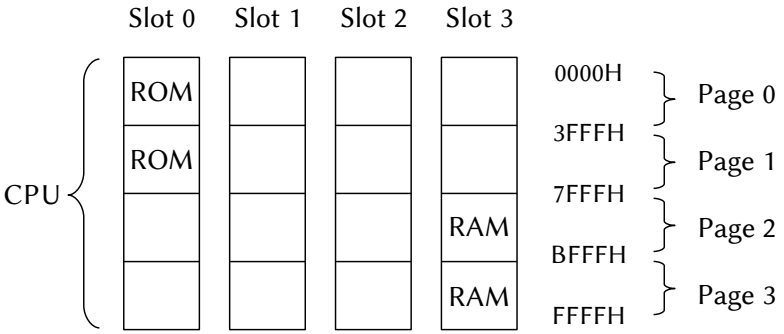
Each primary slot can support up to four secondary slots. The choice of pages is still possible in the same way as in the primary slots: only four pages can be active at the same time, albeit in different primary and secondary slots.

There can be a maximum of 16 slots, of which usually 8 are internal and reserved for system expansion and the other 8 are available to the user in the form of two cartridge connectors that can be expanded to four secondary connectors each. Below is illustrated how the pri-

mary and secondary slots are structured. The first number is the primary slot and the second is the secondary one, in the form "P-S Slot".

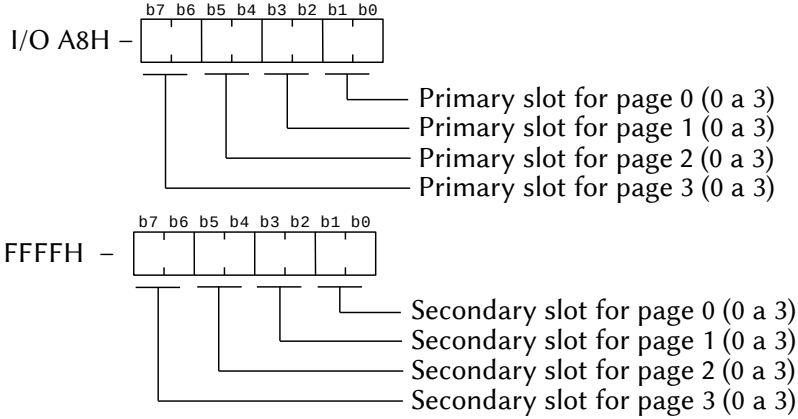


The most common initial memory selection is as follows:



When the MSX is powered on or reset, the address space is automatically plugged into slot 0. Therefore, the Main-ROM with BIOS is placed on the first two pages of slot 0-0. The RAM memory can be in other slots, but the most common is that it is on the last two pages of primary slot 3.

Normally, on a standard MSX1 machine, there are only primary slots. Each primary slot can be assigned up to four secondary slots. Slot and page selection is different for primary and secondary slots. For the primary slots it is done by the I/O port A8H and for the secondary slots it is done by the secondary slot register, which is the FFFFH address of memory. The format of slot registers is described below.

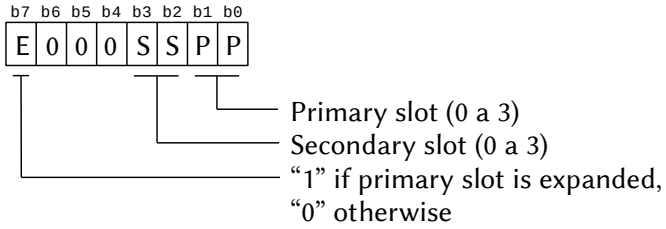


To get the correct value of the secondary slot at address FFFFH, it is necessary to do an inversion after reading (NOT statement in BASIC or CPL in Assembly). It is important to note that the value is only inverted when READ. When the value is WRITTEN, it is not inverted.

On MSX1, there are only the 32K of Main-Rom. For MSX2 onwards, there is also the Sub-ROM, which is placed in another slot. The slots where Main-ROM, Sub-ROM and RAM are installed depend on each machine. In many cases, it is necessary to know where the basic MSX memories are installed, as if you are running a program under DOS and you need to access the Main-ROM, for example. The slots where the Main-ROM and Sub-ROM are installed are specified in the following system variables:

EXPTBL (FCC1H) – Main-ROM Slot
EXBRSA (FAF8H) – Sub-ROM slot (0 for MSX1)

The structure of these registers is illustrated below.



2.1.1 – Inter-slot calls

When a program is running in a certain slot and must call some routine in another slot, it must make an inter-slot call.

Inter-slot calls were created to allow a given software to access routines in other slots in any memory area. Thus, software running under MSXDOS, which occupies all available linear RAM, can access the BIOS or the DOS Kernel residing on the disk interface. Starting with MSX2, there is a BIOS extension ROM called Sub-ROM. The calls that the BIOS itself makes to the Sub-ROM are also called inter-slot.

To facilitate calls and ensure compatibility, there is a group of BIOS routines called "inter-slot call group". Some of these routines are also available for MSXDOS, so it can access all BIOS routines. The routines available for MSXDOS are described in the chapter on the disk system.

The BIOS "inter-slot" routines are as follows:

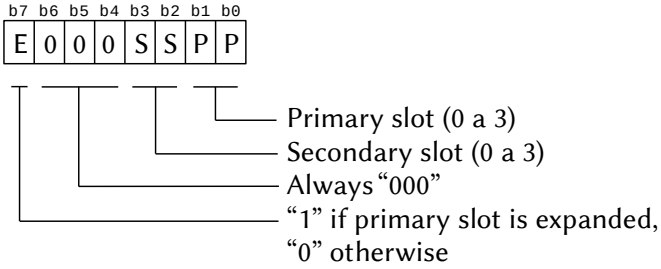
RDLST	(000CH)	Reads a byte from any slot
WRSLT	(0014H)	Write a byte to any slot
CALSLT	(001CH)	Calls a routine in any slot
ENASLT	(0024H)	Swap pages and slots
CALLF	(0030H)	Calls a routine in any slot
RSLREG	(0138H)	Read primary slot register
WSLREG	(013BH)	Writes to primary slot register
SUBROM	(015CH)	Calls a routine in the Sub-ROM
EXTROM	(015FH)	Calls a routine in the Sub-ROM

The complete description of each routine can be seen in the appendix "BIOS CALLS".

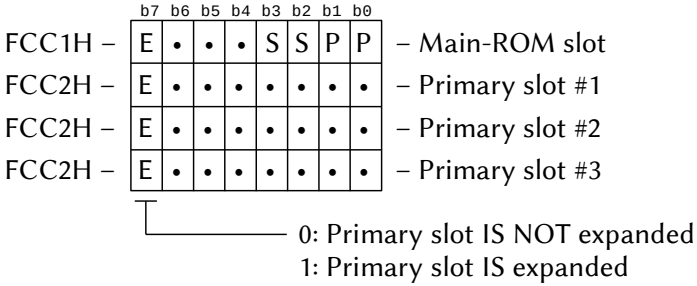
2.1.2 – Work Area

The work area that contains the system variables related to the slots are as follows (the complete description of the system variables can be seen in the appendix "Description of the system work area"):

EXBRSA (FAF8H,1) – SUB-ROM slot

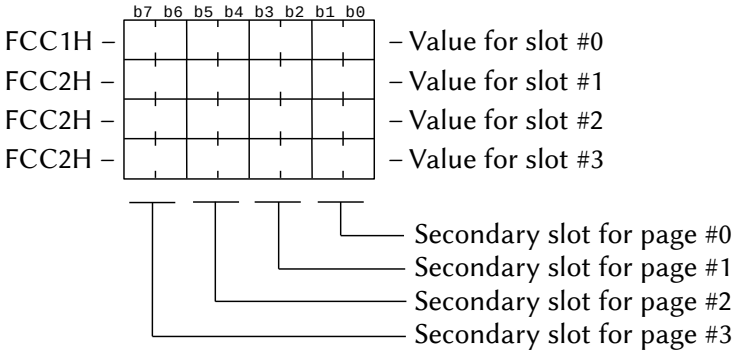


EXPTBL (FCC1H,4) – Indicates whether the primary slot is expanded.

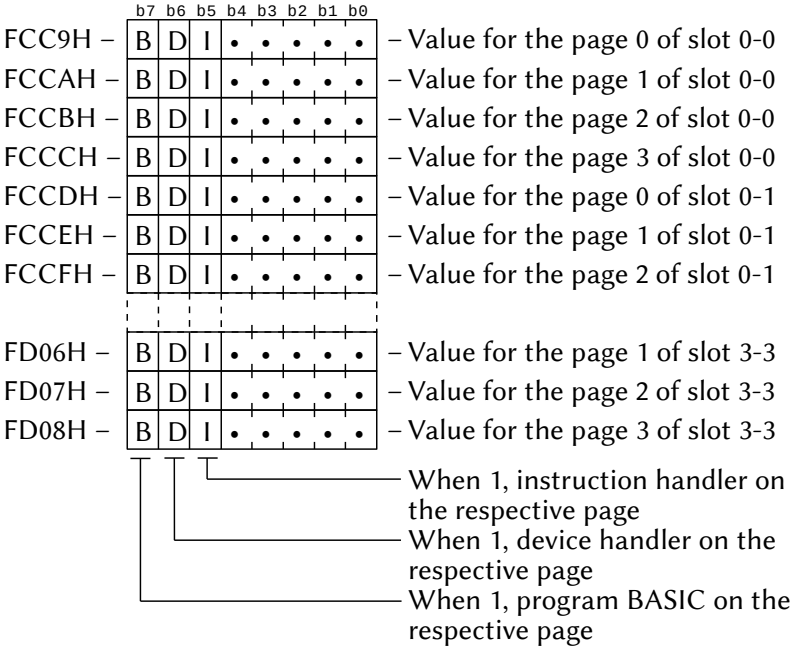


The first byte (FCC1H) also indicates the Main-ROM slot in its last four bits. For the other three bytes only bit 7 is valid.

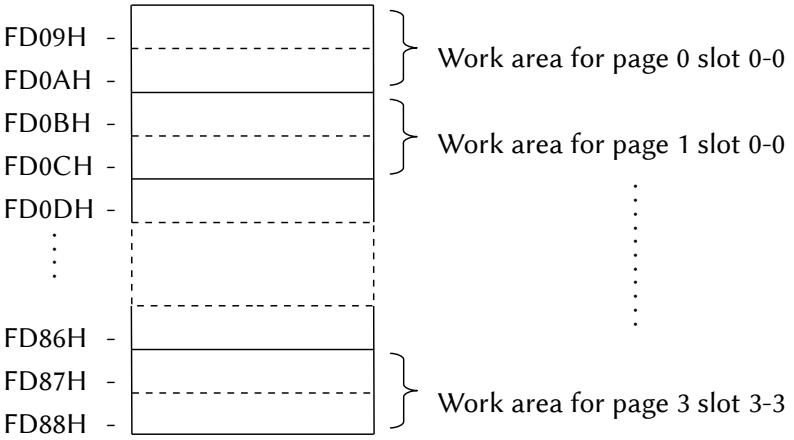
SLTTBL (FCC5H,4) – Area where the expansion values of each primary slot are stored.



SLTATR (FCC9H,64) – Indicates the existence of routines on any page in any slot.



SLTWRK (FD09H,128) – Work area for slots and pages, reserving two bytes for each page.



2.2 – DEVELOPING SOFTWARE FOR CARTRIDGES

Normally, MSX computers have two primary external slots where cartridges containing software, interfaces, etc. can be connected. BASIC or Assembler programs can be easily stored on cartridges containing a ROM or EPROM.

Cartridges must have the first 16 bytes reserved for the header. The header can start at addresses 4000H or 8000H, therefore only on pages 1 or 2. Cartridges cannot occupy the address area of pages 0 and 3. When the MSX is reset, the information contained in the cartridge header is automatically recognized to enable correct execution of the routines contained in it. The composition of the cartridge header is as follows:

+00H	ID	← 4000H ou 8000H
+02H	INIT	
+04H	STATEMENT	
+06H	DEVICE	
+08H	TEXT	
+0AH~+10H	RESERVED	Note: the reserved area must be filled in with bytes 00H.

ID – Two identification bytes. In the case of ROM cartridges, these bytes must have the code "AB" (bytes 41H and 42H) and in the case of Sub-ROM cartridges, the bytes must be "CD" (43H and 44H).

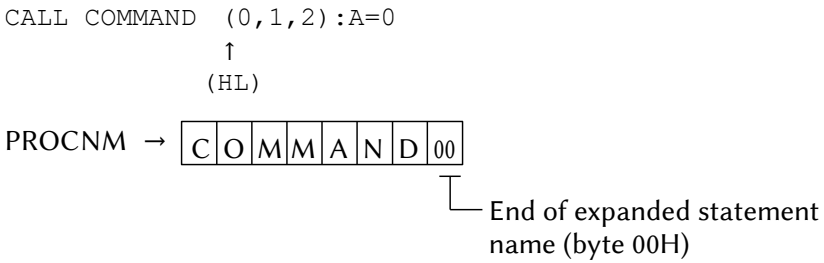
INIT – When it is necessary to initialize the work area or I/O, these two bytes must contain the address of the initialization routine; otherwise they must contain 0000H. After the initialization routine has been executed, the RET instruction returns control to the micro. All registers can be modified except the SP register. Assembler programs must also be executed directly by the INIT bytes.

STATEMENT – When the cartridge is to be accessed by the BASIC CALL instruction, these two bytes must contain the start address of the expansion routine, otherwise they must contain the value 0000H.

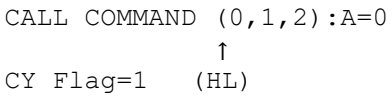
The CALL instruction has the following format:

CALL <name of expansion instruction> (argument)

The expansion instruction name can be up to 15 characters long. When the BASIC interpreter encounters a CALL command, the name of the expansion instruction is placed in the PROCNM (FD89H) and control is transferred to the routine whose start is indicated by the STATEMENT bytes. This routine must recognize the name of the instruction in PROCNM. The HL register points exactly to the first character after the expanded instruction, as shown in the illustration below:



When the expansion routine does not recognize the command, it must keep the value of HL, set the CY flag (CY=1) and return control to the interpreter (RET instruction). The interpreter will then search for other command expansion cartridges, if there are more than one, and the procedure will be the same. If at the end the instruction is not recognized as valid, the CY flag will be set and the message "Syntax Error" will be displayed. This procedure is illustrated below.



If the command was recognized as valid, the corresponding routine will be executed and when returning to the interpreter, the CY flag must be reset (CY=0) and the HL register must point to the first byte after the expanded instruction argument. The byte can be the value 00H (end of line) or 3AH (colon, statement separator). Processing will continue normally. The example below illustrates the described procedure.

```
CALL COMMAND (0,1,2):A=0
                      ↑
CY Flag = 0         (HL)
```

DEVICE – These two bytes can point to a device expansion routine in case the cartridge contains an I/O device; otherwise they must be 0000H. The routine for the expansion device must be between 4000H and 7FFFH. A cartridge can have up to four devices, and the name of each device can be up to 15 characters long.

When the interpreter encounters an undefined device, it stores its name in PROCNM (FD89H), places the value FFH in the A register, and passes control to the cartridge that has a device expansion.

To create a device expansion routine, the file descriptor must be identified in PROCNM (FD89H) first, and if it is not the correct device, control must be returned to the interpreter with the CY flag set (CY=1). The example below illustrates what has been described.

```
OPEN "XYZ:"...
```

Device name

Register A = FFH

CY flag = 1

PROCNM → | X | Y | Z | 00 |

End of device descriptor (byte 00H)

If the device descriptor is recognized, the respective routine must be processed and the device ID number, which varies from 0 to 3, must be placed in the A register; then the CY flag must be reset (CY=0) and control returned to the interpreter.

The interpreter searches for cartridge after cartridge, and if at the end the device name is not recognized (i.e. the CY flag is always 1), the error message "Bad file name" will be displayed.

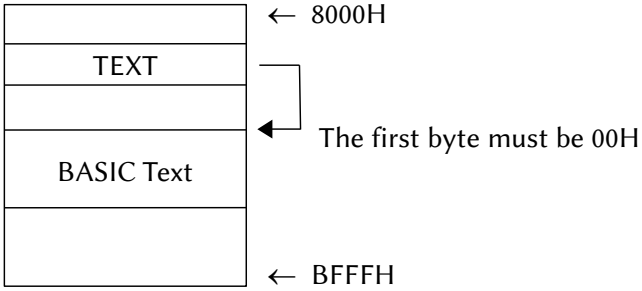
When the current I/O operation is processed, i.e. the device is valid, the interpreter puts the device name (device ID – 0 to 3) in the DEVICE system variable (FD99H) and sets the required device to A with

the values described in the table below, and then call the device expansion routine.

Reg. A	Device	Reg. A	Device
0	OPEN	10	LOC function
2	CLOSE	12	LOF function
4	Random access	14	EOF function
6	Sequential output	16	FPOS function
8	Sequential input	18	"Backup" char

TEXT – These two bytes point to a BASIC program recorded on a cartridge, which is auto-executed when the computer is reset or turned on. If there is no BASIC program, these two bytes must contain 0000H. The program size cannot exceed 16 Kbytes (8000H to BFFFH).

The interpreter examines the contents of TEXT, and if it contains an address, it starts the execution of the BASIC program contained at the indicated address. The first byte pointed to by TEXT must be 00H, which indicates the beginning of the BASIC text. The figure below shows how the BASIC text must be arranged on the ROM cartridge for correct execution.



To put BASIC programs in ROM some steps must be followed. First, change the starting address of the BASIC text to 8021H. The following commands can be used on a single line:

```
POKE &HF676,&H21:POKE &HF677,&H80:POKE &H8020,0:NEW
```

Then load the desired BASIC program and create the cartridge ID. The following commands can be used in direct mode:

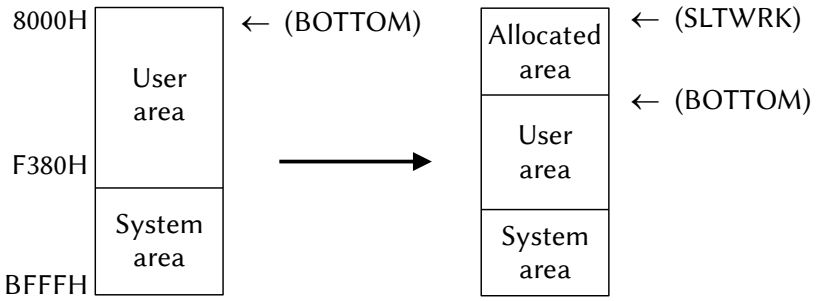
```
AD = &H8000
FOR I=0 TO 31:POKE AD+I,0:NEXT I
POKE &H8000,ASC("A")
POKE &H8001,ASC("B")
POKE &H8008,&H20
POKE &H8009,&H80
```

Then write the contents of page 2 (8000H to BFFFH) to a ROM cartridge.

2.2.1 – Allocating work area to cartridges

For programs that do not require software from other cartridges, such as games, the memory area below F380H can be used freely. But in programs that run using the BASIC interpreter or BIOS functions, there are three options:

1. Put RAM in the cartridge itself (the best way).
2. When only one or two bytes are required for the workspace, the corresponding two bytes in the SLTWRK system variable (FD09H~FD88H) can be used.
3. When more than two bytes are required, it is necessary to allocate RAM used by BASIC. To do this, it is necessary to place the contents of BOTTOM (FC48H) in the corresponding area in SLTWRK (FD09H~FD88H) and increment the value of BOTTOM up to the value required for the work area. This area can then be used as a work area by the cartridge. The figure below illustrates this method.



The area just below the work area can also be used, although this method is not recommended. When connecting a ROM with DOS Kernel (Disk System), a part of RAM is allocated just below F380H. The cartridge work area should be located below this for minimum security. With two logical drives connected for DOS1, an area of up to DA00H is occupied; in the case of DOS2, with eight logical drives connected, up to close to E100H. The allocated area should be just below. The best way to allocate it is by calling BASIC's CLEAR command, as many system variables have to be changed. For more details, see the section "CALLING COMMANDS IN BASIC" in chapter 3 (ROM MEMORY).

The following routine has been published in the MSX2 Technical Handbook as an example of work area allocation for cartridges.

```
;************************************************
;Subroutines to support slot for ROM in 1 page
;************************************************
;
RSLREG      EQU   00138H
EXPTBL      EQU   0FCC1H
BOTTOM      EQU   0FC48H
HIMEM       EQU   0FC4AH
SLTWRK      EQU   0FD09H
;
;-----------------------------------------------
;
;GTSL1      Get slot number of designated page
;Entry      None
;Return     A - Slot address as follows
;Modify     Flags
;
;     FxxxSSPP
;     |   ||||
;     |   ||++-- Primary slot # (0-3)
;     |   ++---- Secondary slot # (0-3)
;     |          00 if not expanded
;     +--------- 1 if secondary slot # specified
;
;This value can later be used as an input parameter
;for the RDSLT, WRSLT, CALSLT, ENASLT and 'RST 10H'
;
      PUBLIC      GTSL10
```

```
GETSL10:
      PUSH  HL          ;Save registers
      PUSH  DE

      CALL  RSLREG      ;Read primary slot #
      RRCA
      RRCA
      AND   11B         ;[A]=000000PP
      LD    E,A
      LD    D,0         ;[DE]=000000PP
      LD    HL,EXPTBL
      ADD   HL,DE       ;[HL]=EXPTBL+000000PP
      LD    E,A         ;[E]=000000PP
      LD    A,(HL)      ;A=(EXPTBL+000000PP)
      AND   80H         ;Use only MSB
      JR    Z,GTSL1NOEXP
      OR    E           ;[A]=F00000PP
      LD    E,A         ;Save primary slot number
      INC   HL          ;Point to SLTTBL entry
      INC   HL
      INC   HL
      INC   HL
      LD    A,(HL)      ;Get current expansion slot
                        ;Register
      RRCA
      RRCA
      AND   11B         ;[A] = 000000SS
      RLCA
      RLCA              ;[A] = 0000SS00
      OR    E           ;[A] = F000SSPP
;
GTSL1END:
      POP   DE
      POP   HL
      RET
GTSL1NOEXP:
      LD    A,E         ;[A] = 000000PP
      JR    GTSL1END

;-----------------------------------------------
;
;ASLW1      Get address of slot work
```

```
;Entry      None
;Return     HL    address of slot work
;Modify     None
;
      PUBLIC      ASLW10
ASLW10:
      PUSH  DE
      PUSH  AF
      CALL  GTSL10      ;[A] = F000SSPP, SS = 00 if
                        ;Not expanded
      AND   00001111B   ;[A] = 0000SSPP
      LD    L,A         ;[A] = 0000SSPP
      RLCA
      RLCA
      RLCA
      RLCA              ;[A] = SSPP0000
      AND   00110000B   ;[A] = 00PP0000
      OR    L           ;[A] = 00PPSSPP
      AND   00111100B   ;[A] = 00PPSS00
      OR    01B         ;[A] = 00PPSSBB
;
;Now, we have the sequence number for this cartridge
;as follows.
;
;     00PPSSBB
;       ||||||
;       ||||++-- Higher 2 bits of memory address (1)
;       ||++---- Seconday slot # (0..3)
;       ++------ Primary slot # (0..3)
;
      RLCA              ;*=2
      LD    E,A
      LD    D,0         ;[DE] = 0PPSSBB0
      LD    HL,SLTWRK
      ADD   HL,DE
      POP   AF
      POP   DE
      RET

;-----------------------------------------------
;
;RSLW1      Read slot work
```

```
;Entry      None
;Return     HL    Content of slot work
;Modify     None
;
      PUBLIC      RSLW10
RSLW10:
      PUSH  DE
      CALL  ASLW10      ;[HL] = Address of slot work
      LD    E,(HL)
      INC   HL
      LD    D,(HL)      ;[DE] = (Slot work)
      EX    DE,HL       ;[HL] = (Slot work)
      POP   DE
      RET

;-----------------------------------------------
;
;WSLW1      Write slot work
;Entry      HL    Data to write
;Return     None
;Modify     None
;
      PUBLIC      WSLW10
WSLW10:
      PUSH  DE
      EX    DE,HL       ;[DE] = Data to write
      CALL  ASLW10      ;[HL] = Address of slot work
      LD    (HL),E
      INC   HL
      LD    (HL),D
      EX    DE,HL       ;[HL] = Data to write
      POP   DE
      RET

;-----------------------------------------------
;
; How to allocate work area for cartridges.
; If the work area is greater than 2 bytes, make the
; SLTWRK point to the system variable BOTTOM
; (0FC48H), then update it by the amount of memory
; required. BOTTOM is set up by the initization
; code to point to the bottom of equipped RAM.
```

```
;
;     Ex, if the program is at 4000H...7FFFH.
;
;WORKB      Allocate work area from BOTTOM
;           (my slot work) <- (old BOTTOM)
;Entry            HL    required memory size
;Return     HL    start address of my work area
;                 = old BOTTOM
;                 0 if cannot allocate
;Modify     None
;
      PUBLIC      WORKB0
WORKB0:
      PUSH  DE
      PUSH  BC
      PUSH  AF

      EX    DE,HL       ;[DE] = Size
      LD    HL,(BOTTOM) ;Get current RAM bottom
      CALL  WSLW10      ;Save BOTTOM to slot work
      PUSH  HL          ;Save old BOTTOM
      ADD   HL,DE       ;[HL] = (BOTTOM) + SIZE
      LD    A,H         ;Beyond 0DFFFH?
      CP    0E0H
      JR    NC,NOROOM   ;Yes, cannot allocate this
                        ;much
      LD    (BOTTOM),HL ;Updtae (BOTTOM)
      POP   HL          ;[HL] = old BOTTOM
WORKBEND:
      POP   AF
      POP   BC
      POP   DE
      RET
;
;     BOTTOM became greater than 0DFFFH, there is
;     no RAM to be allocated.
;
NOROOM:
      LD    HL,0
      CALL  WSLW10      ;Clear slot work
      JR    WORKBEND    ;Return 0 in [HL]

      END
```

Chapter 3
THE ROM MEMORY

The ROM memory is vital for the functioning of any microcomputer. In the case of MSX, it incorporates the boot routine, the BIOS, the initial character table, the MSX-DOS (DOS Kernel), etc.

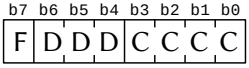
Also, there are some bytes at the beginning of the ROM that contain some important information that might be useful to the programmer. These bytes are:

0004H/0005H – Adress of the character table in ROM.

0006H – VDP data read port.

0007H – Port for writing data to the VDP.

002BH –

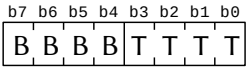


Charater table chart:
- 0 – japanese
- 1 – international
- 2 – korean

Date format:
- 0 – year/month/day
- 1 – month/day/year
- 2 – say/month/year

Interrupt frequency:
- 0 – 60 Hz
- 1 – 50 Hz

002CH –



Keyboard type:
- 0 – japanese
- 1 – international
- 2 – french
- 3 – united kingdom
- 4 – german

BASIC version:
- 0 – japanese
- 1 – international

002DH – Hardware version:
00H = MSX1
01H = MSX2
02H = MSX2
03H = MSX turbo R

3.1 – BIOS

Virtually every program, whether in assembler or high-level language, including the MSX resident BASIC interpreter itself, requires a set of primary functions in order to operate. These functions include screen triggers, printers, drivers, and other hardware-related functions. On MSX, these primary functions are performed by the BIOS routines, which stands for "Basic Input/Output System".

There are 137 BIOS routines available to the user, if the system is an MSX turbo R. For previous versions, the number of available routines is smaller.

There are two types of BIOS routines: those in the Main-ROM and those in the Sub-ROM. For MSX1 there is no Sub-ROM; for MSX2, MSX2+ and MSX turbo R there are 16K, 32K and 48K of Sub-ROM respectively. Main-ROM and Sub-ROM routines use different call sequences. For Main-ROM, a CALL or RST instruction can be used. The calls to the Sub-ROM are made with the help of the EXTROM (015FH) or SUBROM (015CH) routine of the Main-ROM, loading IX with the address of the Sub-ROM routine to be called, and proceeding according to the example below:

```
LD   IX,ROUTINE    ;Load ROUTINE adress on IX
CALL EXTROM        ;Executes the routine
...                ;Returns from routine
```

When the contents of IX must not be destroyed, the following calling sequence can be used:

```
INIROT: PUSH IX          ;Save IX
        LD   IX,ROUTINE  ;Load IX with adress
        JP   SUBROM      ;Executes the routine
        ...              ;Returns from INIROT call
```

All Main-ROM and Sub-ROM routines are described in detail in the Appendix, chapter 8 – BIOS Routines.

3.2 – THE MATH-PACK

The Math-Pack is a set of non-BIOS math routines that form the core of MSX-BASIC's math operations. These routines can be used by assembly programs, making floating point, arithmetic, logarithmic and trigonometric operations available, in addition to various special operations.

Operations involving real numbers with Math-Pack are performed in BCD (Binary Coded Decimal). Numbers can be 2-byte integers (–32 768 to +32 767), single precision (6 digits, with exponent from –63 to +63) occupying 4 bytes, or (14 digits with exponent from –63 to +63), occupying 8 bytes.

To store an integer, bit b15 is reserved for the sign, leaving 15 bits to store the module. Integer numbers have the following format:

b15	B14~b0	decimal
0	000000000000000	+0
0	000000000000001	+1
:	:	:
0	111111111111111	+32767
1	000000000000000	-32768
1	000000000000001	-32767
:	:	:
1	111111111111111	-1

A real number is made up of a mantissa, a sign, and an exponent. The sign of the mantissa is represented by 0 (positive) or 1 (negative). The exponent is a 7-bit binary expression that represents a power of 10 and can range from –63 to +63. The way floating point numbers are stored in memory is illustrated below.

The binary expression forming the exponent and the sign of the mantissa is illustrated below.

b7	b6	b5	b4	b3	b2	b1	b0	
+/-	exponent							
0	0	0	0	0	0	0	0	positive exponent
1	0	0	0	0	0	0	0	undefined (-0 ?)
x	0	0	0	0	0	0	0	-63rd power of 10
x	0	0	0	0	0	0	0	0th power of 10
x	1	1	1	1	1	1	1	+63rd power of 10

Example of single precision number:

123.456 → 0,123456E+6

DAC →

0	1	2	3
46	12	34	56

Example of double precision number:

123.456,78901234 → 0,12345678901234E+6

DAC →

0	1	2	3	4	5	6	7
46	12	34	56	78	90	12	34

The digits that make up the mantissa are always considered to be placed immediately after the comma.

3.2.1 – Work area

In order for Math-Pack to perform its functions, there are two reserved memory areas, which are the “DAC” (Decimal ACumulator, F7F6H) and the “ARG” (ARGument, F847H). For example, in a multiplication, the product of the numbers contained in DAC and ARG is calculated and the result is placed in DAC.

In the DAC, double-precision, single-precision or two-byte integers can be stored, in which case the two bytes representing the integer are stored in DAC+2 and DAC+3. In order for Math-Pack routines to distinguish which type of number is stored in DAC, the system variable

VALTYP (F663H) is used, which must contain the value 2 for integer numbers, 4 for single-precision numbers, and 8 for integer numbers. .

When using Math-Pack routines in assembly, special care must be taken. As these are routines used by the BASIC interpreter, if an error occurs (such as division by zero or overflow, for example), control is automatically transferred to the error handler, which then returns control to the interpreter. To prevent this from happening, the HERRO hook (FFB1H) can be used to trap the error before control is returned to the interpreter. The error code is stored in the "E" register and can also be used by the assembly program.

To use Math-Pack routines in assembly programs, proceed exactly the same way as BIOS routines are called. The appropriate values are placed in ARG, DAC and VALTYP and eventually in some CPU register and the routine is called through the CALL instruction or through the CALSLT or CALLF routines. The only observation to make is that very few routines preserve any registers; therefore it is always necessary to save on the stack the registers that must not be destroyed. Below and on the following page are described system variables used by Math-Pack.

VALTYP (F663H, 1 byte)
Number format contained in DAC (2, 4 or 8).

DAC (F7F6H, 16 bytes)
Floating point accumulator in BCD format.

ARG (F847H, 16 bytes)
Argument for use with DAC.

3.2.2 – Math-Pack Entries

3.2.2.1 – Floating point operations

Label	Adress	Function
DECSUB	268CH	DAC ← DAC – ARG (Double precision)
DECADD	2691H	DAC ← DAC + ARG (Double precision)
DECMUL	27E6H	DAC ← DAC * ARG (Double precision)
DECDIV	289FH	DAC ← DAC / ARG (Double precision)
SGNEXP	37C8H	DAC ← DAC ^ ARG (Single precision)

DBLEXP	37D7H	DAC ← DAC ^ ARG (Double precision)
COS	2993H	DAC ← COS (DAC)
SIN	29ACH	DAC ← SIN (DAC)
TAN	29FBH	DAC ← TAN (DAC)
ATN	2A14H	DAC ← ATN (DAC)
LOG	2A72H	DAC ← LOG (DAC)
SQR	2AFFH	DAC ← SQR (DAC)
EXP	2B4AH	DAC ← EXP (DAC)

3.2.2.2 – Integer numbers operations

Label	Adress	Function
UMULT	314AH	DE ← BC * DE (unsigned multiplication)
ISUB	3167H	HL ← DE – HL
IADD	3172H	HL ← DE + HL
IMULT	3193H	HL ← DE * HL
IDIV	31E6H	HL ← DE / HL
INTEXP	383FH	DAC ← DE ^ HL
IMOD	323AH	HL ← DE mod HL DE ← DE / HL

3.2.2.3 – Special functions

Label	Adress	Function
DECNRM	26FAH	Normalizes DAC[1]
DECROU	273CH	Rounds DAC
RND	2BDFH	DAC ← RND (DAC)
SIGN	2E71H	A ← Mantissa signal in DAC
ABSFN	2E82H	DAC ← ABS (DAC)
NEG	2E8DH	DAC ← NEG (DAC)
SGN	2E97H	DAC ← SGN (DAC)[2]
FRCINT	2F8AH	Converts DAC to 2 bytes integer[3]
FRCSNG	2FB2H	Converts DAC to single precision
FRCDBL	303AH	Converts DAC to double precision
FIXER	30BEH	DAC ← SGN(DAC) * INT(ABS(DAC))

1 Excessive zeros in the mantissa are removed (Ex. 0.00123 ⟶ 0.123E-2)
2 For the SGN function, the result is represented by a 2 byte integer.
3 The conversion result is stored in DAC +2 and +3.

3.2.2.4 – Movement

Label	Adress	Function	
MAF	2C4DH	ARG ← DAC	Double precision
MAM	2C50H	ARG ← (HL)	Double precision
MOV8DH	2C45H	(DE) ← (HL)	Double precision
MFA	2C59H	DAC ← ARG	Double precision
MFM	2C5CH	DAC ← (HL)	Double precision
MMF	2C67H	(HL) ← DAC	Double precision
MOV8HD	2D6AH	(HL) ← (DE)	Double precision
XTF	2C6FH	(SP) ↔ DAC	Double precision
PHA	2CC7H	ARG ← (SP)	Double precision
PHF	2CCCH	DAC ← (SP)	Double precision
PPA	2CDCH	(SP) ← ARG	Double precision
PPF	2CE1H	(SP) ← DAC	Double precision
PUSHF	2EB1H	DAC ← (SP)	Single precision
MOVFM	2EBEH	DAC ← (HL)	Single precision
MOVFR	2EC1H	DAC ← CBED	Single precision
MOVRF	2ECCH	CBED ← DAC	Single precision
MOVRMI	2ED6H	CBED ← (HL)	Single precision
MOVRM	2EDFH	BCDE ← (HL)	Single precision
MOVMF	2EE8H	(HL) ← DAC	Single precision
MOVE	2EEBH	(HL) ← (DE)	Single precision
VMOVAM	2EEFH	ARG ← (HL)	VALTYP
MOVVFM	2EF2H	(DE) ← (HL)	VALTYP
VMOVE	2EF3H	(HL) ← (DE)	VALTYP
VMOVFA	2F05H	DAC ← ARG	VALTYP
VMOVFM	2F08H	DAC ← (HL)	VALTYP
VMOVAF	2F0DH	ARG ← DAC	VALTYP
VMOVMF	2F10H	(HL) ← DAC	VALTYP

Note: (HL) and (DE) mean the memory addresses pointed to by HL and DE. Four register names together (CBED or BCDE) contain a single precision number (sign + exponent, 1st and 2nd digits, 3rd and 4th digits, 5th and 6th digits). When the object is VALTYP, the movement will be according to the type indicated by VALTYP (F663H), that is, 2, 4 or 8 bytes.

3.2.2.5 – Comparisons

Label	Adress		Esquerdo	Direito
ICOMP	2F4DH	Inteiro de 2 bytes	DE	HL
FCOMP	2F21H	Single precision	CBED	DAC
XDCOMP	2F5CH	Double precision	ARG	DAC

Note: When comparing single-precision real numbers, CBED must contain one of the single-precision operands (sign + exponent, 1st and 2nd digits, 3rd and 4th digits, 5th and 6th digits). The result of the comparison will be placed in register A, as shown below:

A = 1 ⟶ left < right

A = 0 ⟶ left = right

A = –1 ⟶ left > right

3.2.2.6 – Other floating point and I/O operations

FIN (3299H)

Function: Converts a string representing a real number to BCD format and stores it in DAC.
Input: HL – adress of the first character of the string.
Output: DAC – Real number in BCD.
C – FFH ⟶ no decimal point; 0 ⟶ with decimal point.
B – Number of digits after the decimal point.
D – Total number of digits.

FOUT (3225H)

Function: Converts a real number contained in DAC to an unformatted string.
Input: A – Always 0.
B – Number of digits before the decimal point.
C – Number of digits after the decimal point, including this one.
Output: HL – adress of the first character of the string.

PUFOUT (3426H)

Function: Converts a real number contained in DAC to a string, formatting.

Input: A – bit 7: 0 – unformatted 1 – formatted
bit 6: 0 – no commas 1 – with commas every 3 dig.
bit 5: 0 – meaningless 1 – fill spaces with "*"
bit 4: 0 – meaningless 1 – add "$" before the number
bit 3: 0 – meaningless 1 – put "+" for num. position
bit 2: 0 – meaningless 1 – sign after number
bit 1: 0 – not used
bit 0: 0 – fixed point 1 – floating point
B – Number of digits before the decimal point
C – Number of digits after the decimal point, including this one.
Output: HL – adress of the first character of the string.

FOUTB (371AH)
Function: Converts an integer to a binary expression.
Input: DAC+2 = integer
VALTYP = 2
Output: HL – adress of the first character of the string.

FOUTO (371EH)
Function: Converts an integer to an octal expression
Input: DAC+2 = integer
VALTYP = 2
Output: HL – adress of the first character of the string.

FOUTH (3722H)
Function: Converts an integer to a hexadecimal expression
Input: DAC+2 = integer
VALTYP = 2
Output: HL – adress of the first character of the string.

3.3 – THE BASIC INTERPRETER

Most of the BASIC interpreter resides on page 1 of the ROM. The text area of a BASIC program normally starts at address 8000H (which corresponds to the beginning of page 2) but can be changed by changing the system variable TXTTAB (F676H) which initially contains the value 8000H and indicates the beginning of the text area. BASIC text.

3.3.1 – The tokens

For each BASIC reserved word there is a corresponding code called a "token". A token is nothing more than a single byte representing a BASIC reserved word.

As you can see, BASIC text is not stored in ASCII form, but in a more compact form. The purpose of tokens is not only to make the BASIC text more compact, but also faster, since during processing, instead of decoding the entire ASCII sequence of the command, the interpreter only needs to decode one byte.

A BASIC command, for example "PRINT A", will be stored in the BASIC text area as follows:

byte 91H – PRINT command token
byte 20H – Space
byte 41H – ASCII code of variable 'A'

BASIC functions are stored in a slightly different way. function tokens are preceded by an FFH byte and have their bit 7 set. For example, a BASIC function type "X=SIN(A)" is stored as follows:

byte 58H – ASCII code of variable 'X'
byte EFH – Token of the '=' sign
byte FFH – function identifier
byte 89H – Set token of the SIN function
byte 28H – ASCII code of '('
byte 41H – ASCII code of variable 'A'
byte 29H – ASCII code of ')'

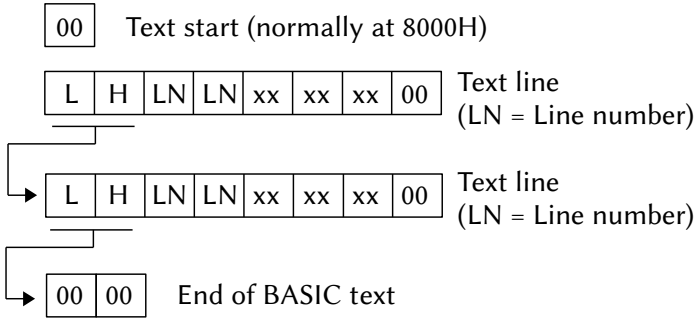
All BASIC commands and functions with their respective tokens can be seen in the section "CALLING COMMANDS IN BASIC".

3.3.2 – Structure of program lines

The way in which program lines are stored in the BASIC text area is quite simple.

The first two bytes (usually 8001H and 8002H) contain the start address of the next line; the next two contain the line number (which can vary from 0 to 65 529) and then come the bytes that store the line itself, which can be up to 254 bytes, the last one must be 00H, indicating

the end of the line. When the program ends, two more bytes 00H are added, indicating this fact. The way to store the lines is illustrated below.



3.3.3 – Number storage

The numbers are stored in a special way, in order to save as much memory as possible in the text area. Whole numbers are handled in a very peculiar way. They are divided into three groups: 0 to 9, 10 to 255 and 256 to 32767. For integers from 0 to 9, there is a kind of token that identifies it as such, as shown in the table below:

0 – 11H	1 – 12H	2 – 13H	3 – 14H	4 – 15H
5 – 16H	6 – 17H	7 – 18H	8 – 19H	9 – 20H

For integers from 10 to 255, an identification byte is placed before the number, which in this case is 0FH. Right after the identification byte, there is the byte that numerically represents the value, from 10 to 255. For integers from 256 to 32767, there is also an identification byte (1CH) followed by two bytes that store the number in the LSB–MSB form. If an integer is negative, it is preceded by the "–" sign token (F2H). See the examples:

4	15H	
–4	F2H 15H	
35	0FH 23H	(23H = 35 in decimal)
–35	F2H 0FH 23H	

1000	1CH	E8H	03H	(03E8H = 1000 in decimal)
–1000	F2H	1CH	E8H	03H

Single-precision numbers are stored in four bytes, in BCD form, preceded by the identification byte 1DH. Double-precision numbers are stored in eight bytes, also in BCD form, preceded by the identification byte 1FH.

Numbers stored in other bases (binary, octal and hexadecimal) also have their identification bytes. For a binary number, it's two ID bytes (26H and 42H, or "&B"), which is stored in ASCII form. The octal numbers have the byte 0BH as ID and are stored in the form LSB-MSB. For hexadecimal numbers, the ID byte is 0CH and the number is also stored in the form LSB-MSB.

The numbers that refer to program lines (in the GOTO and GOSUB instructions, for example) have a very peculiar treatment. During program typing, the line number is stored in two bytes in the form LSB-MSB, preceded by the identification byte 0EH. When the line is executed for the first time, the interpreter will change the ID byte to 0DH and the next two bytes will contain the start address of the respective line, and no longer the line number. This is done to speed up the execution of the program the next time it is run.

The different forms of storage are illustrated in the figure below.

Integer from 0 to 9	xx	(xx pode variar entre 11H a 1AH)							
Integer from 10 to 255	0FH	xx							
Integer 256 to 32 767	1CH	xx	xx						
Single precision	1DH	xx	xx	xx					
Double precision	1FH	xx	xx	xx	xx	xx	xx	xx	
Octal number	0BH	xx	xx						
Hexadecimal number	0CH	xx	xx						
Binary number	26H	42H	number in ASCII format (0-1)						
Line (before RUN)	0EH	xx	xx						
Line (after RUN)	0DH	xx	xx						

3.3.4 – Area of the interpreter variables

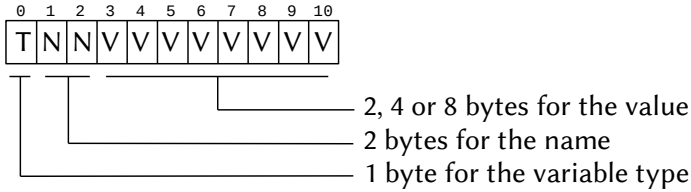
The memory area just above the end of the BASIC text is allocated to store program variables. This area starts at the address pointed to by VARTAB (F6C2H) and ends at the address pointed to by STREND (F6C6H)1. Each time a variable is consulted, the interpreter looks for it in the area delimited by VARTAB and STREND and, if it does not find it, it assumes the value 0 for numeric variables or null for string variables.

Whenever a new BASIC line is introduced, deleted or the CLEAR command is executed, the value of STREND is equaled to the value of VARTAB and consequently all program variables are cleared and become null.

There are 4 types of BASIC variables:

integer numbers: occupy 2 bytes
single-precision numbers: occupy 4 bytes
double-precision numbers: occupy 8 bytes
alphanumeric (strings): occupy 3 bytes

The syntax of the numeric variables is illustrated below:



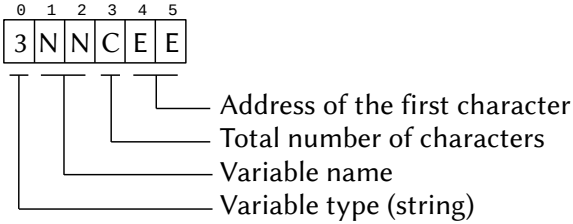
The first byte indicates the type of numeric variable that is stored: 2, 4 or 8. This value also already indicates the number of bytes occupied by the variable.

The interpreter defaults to double-precision variables, but the variable type can be changed by the DEFINT, DEFSNG, DEFDBL and DEFSTR commands. These commands have a table that starts with F6CAH and has 26 bytes, one for each letter of the alphabet, which indicates that the variable whose name starts with that letter must assume the indicated type:

02 – integer 04 – single precision
03 – string 08 – double precision

The variable type immediate identification signs (%, !, #, and $) take precedence over the values indicated by this table.

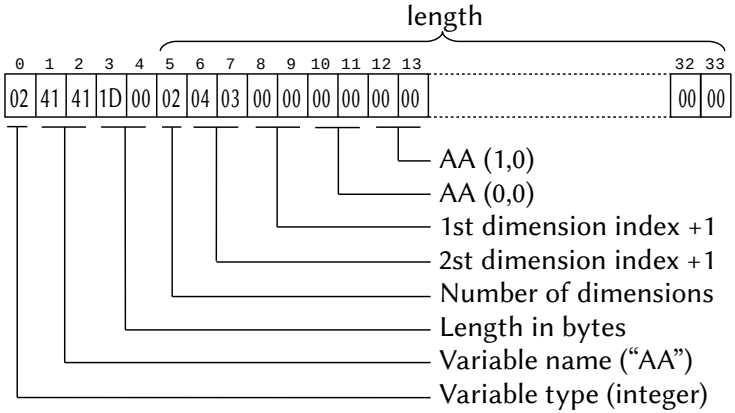
Alphanumeric variables (strings) have a slightly different storage form, whose syntax is described below:



The FRETOP system variable (F69BH) stores the address that will receive the last character of the string being stored.

If there is a direct assignment to an alphanumeric variable (type A$="XYZ"), the address that the pointer will indicate will be in the text area of the BASIC program and not in the area reserved for string variables, avoiding data duplication and saving memory. This also happens when reading data stored in "DATA" instructions: the pointer will indicate the text right after the "DATA" instruction, not transferring it to the string area. However, any operation performed with the variable that modifies it will cause the data represented by it to be transferred to the reserved area and the pointer will contain the respective address in this area.

Arrays have a different storage format, but the storage format is the same as for simple variables. First comes the ID byte followed by the variable name; then two bytes indicate the total length of the array (considering 3 bytes of the pointer for alphanumeric variables or 2, 4 or 8 bytes for numeric variables). The indicated length includes all values that follow. After the length comes a byte that indicates the number of dimensions in the array, followed by as many two-byte sequences as there are dimensions of the array. These two bytes are pointers to each of the dimensions of the array, plus 1. Then comes the storage of the variables themselves. Below is illustrated how the array AA%(2,3) is stored.



3.3.5 – Calling Assembly Programs in BASIC

To use assembly programs together with BASIC, there are 3 commands reserved for this purpose: USR, CMD and IPL. The most common usage is with the USR function; up to 10 routines can be defined with it. To use it, just follow the following three steps:

1 – Specify the routine execution address through the DEFUSR command;
2 – Call the routine through the USR command;
3 – To return control to the interpreter, use a RET instruction.

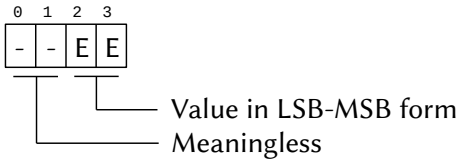
Any argument can be passed to the assembly program by the USR function. In this case, the A register will contain the type of variable passed and the DE register will contain the address of a pointer in the case of string variables or HL will contain the address of the variable itself, if it is numeric, as shown in the illustrations below.

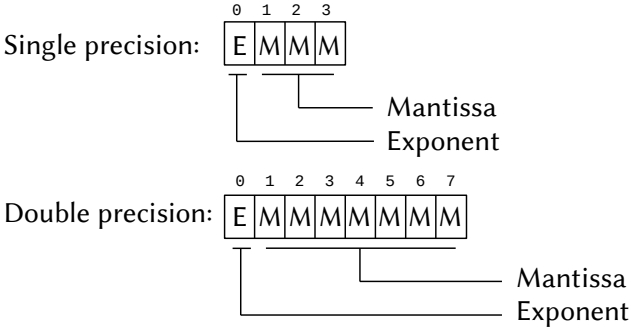
"A" values:

02 – integer numeric
03 – string
04 – single precision
08 – double precision

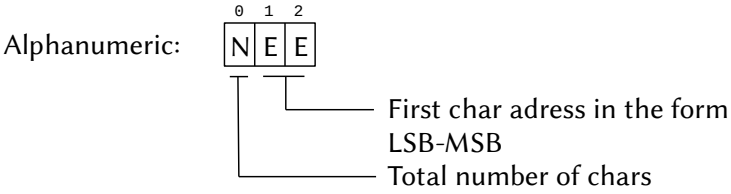
Addresses pointed out by HL:

Integer:





Adress pointed by DE:



The USR function also allows passing variables changed by the Assembly routine to BASIC. In this case, the HL and DE values must contain the starting address of the variable or pointer. Numerical variables can be freely typed, just by changing the VALTYP system variable (F663H) and respecting the way the variable is stored. However, string variables cannot be type-swapped, nor can they have changed the number of characters.

3.3.5.1 – Implementing new commands

New commands can also be implemented by using the reserved words CMD or IPL. To do so, just change the respective hook (FE0DH for CMD and FE03H for IPL) making it point to the Assembly routine.

If a POP AF instruction is inserted right at the beginning of the hook, if an error occurs in the execution of the command, there will be no error generation when returning to BASIC. In this case, the Assembly routine itself may print an internal error message.

In case any argument is passed to the assembly routine by these commands, the HL register will point to the first character after the command, as illustrated below:

```
CMD "ARGUMENT"            IPL "ARGUMENT"
    ↑             or          ↑
  (HL)                      (HL)
```

On return, the HL pair must point to the first flag after the implemented command, which can be 00H (end of line) or 3AH (colon, statement separator).

For more sophisticated implementations, arguments can be passed to the CMD or IPL commands. As the HL register points to the first character after the command when the ASM routine is called, it can interpret a complex string. In this case, there are several interpreter routines that can be very useful. There is a detailed description of these routines in the next item.

3.3.6 – Interpreter routines

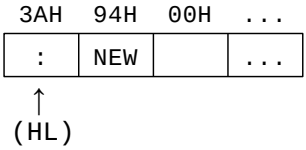
There are some standard interpreter routines that are available for assembly programs. They are listed below in the same way as the BIOS routines. As they are interpreter routines, if an error occurs, control will be transferred to the error handler and then returned to the interpreter. To prevent this from happening, the HERRO hook (FFB1H) can be used to trap the error. The error code is stored in the E register and can be used by the assembly routine. All registers are changed by these routines.

READYR (409BH/Main)
  Function: Returns to command level (warm start of BASIC).
  Input: None
  Output: None

CRUNCH (42B2H/Main)
  Function: Converts a BASIC text from ASCII to tokenized form.
  Input: HL – Address of the ASCII text to be converted, terminated by a 00H byte.
  Output: KBUF (F41FH) – converted BASIC text.

NEWSTT (4601H/Main)
  Function: Executes a BASIC text. The text must be in tokenized form.

Input: HL – Pointer to the beginning of the text to be executed. The text should be in the form illustrated below:



Output: None.

CHRGTR (4666H/Main) – from 0010H

Function: Extracts a character from BASIC text, starting with (HL)+1. Spaces are ignored.
Input: HL – starting address of the text
Output: HL – extracted character address
A – ASCII code of the extracted character
Z flag – "1" if end of line (00H or 3AH ":")
CY flag – "1" if it is a character from 0 to 9

FRMEVL (4C64H/Main)

Function: Evaluates an expression and returns the result.
Input: HL – Starting address of the expression in the BASIC text.
Output: HL – End address of expression +1.
VALTYP (F663H) – 2, 3, 4 or 8 (variable type)
DAC (F7F6H) – Result of the evaluated expression

GETBYT (521CH/Main)

Function: Evaluates an expression and return a 1-byte result. When the result extrapolates the value of 1 byte, an "Illegal Function" error will be generated and the execution will return to the command level.
Input: HL – Starting address of the expression to be evaluated
Output: HL – End address of expression +1.
A, E – evaluation result (A and E contain the same value)

FRMQNT (542FH/Main)

Function: Evaluates an expression and return a 2-byte (integer) result. When the result extrapolates the value of 2 bytes, an "Overflow" error will be generated and execution will return to the command level.

Input: HL – Starting address of the expression to be evaluated.
Output: HL – End address of expression +1.
DE – Evaluation result.

SYNCHR (558CH/Main) – from 0008H

Function: Tests whether the character pointed to by (HL) is the one specified. If not, generates "Syntax error"; otherwise it calls CHRGTR (4666H/Main).

Input: HL – Points to the character to be tested.
The character for comparison must be placed after an instruction "RST 0008H" in the form of an in-line parameter, as shown in the example below:

```
        LD   HL,CHAR
        RST  008H
        DEFB 'A'
        |
CHAR:   DEFB 'B'
```

Output: HL is incremented by one and A receives (HL). When the tested character is numeric, the CY flag is set. The end of declaration (00H or 3AH ":") sets the Z flag.

GETYPR (5597H/Main) – from 0028H

Function: Gets the type of operand contained in DAC.

Input: None

Output: CY, S, Z and P/V flags, as shown in the table below:

Inteiro:	C=1	S=1*	Z=0	P/V=1
Single precision:	C=1	S=0	Z=0	P/V=0*
Double precision:	C=0*	S=0	Z=0	P/V=1
String:	C=1	S=0	Z=1*	P/V=1

Note: The types can be recognized by only the flags marked with "*".

PTRGET (5EA4H/Main)

Function: Gets the address for storing a variable or matrix. The address is also taken when the variable has not been assigned. When the value of SUBFLG (F5A5H) is different from 0, the starting address of an array is taken; otherwise, the address of the array element will be obtained.

Input: HL – Starting address of the variable name in BASIC text.
SUBFLG (F6A5H) – 0: single variable, other value: array
Output: HL – adress after variable name
DE – adress where the contents of the variable are stored.

FRESTR (67D0H/Main)

Function: Registers the result of a string obtained by FRMEVL (4C64H) and obtains the respective descriptor. When evaluating a string, this routine is usually combined with FRMEVL as described below:

```
CALL FRMEVL
PUSH HL
CALL FRESTR
EX   DE,HL
POP  HL
LD   A,(DE)
...
```

Input: VALTYP (F663H) – variable type (must be 3).
DAC (F7F7H) – pointer to string descriptor.
Output: HL – Pointer to string descriptor.

3.3.7 – Calling interpreter commands

It is possible to use interpreter routines in assembly programs. However, when calling a BASIC command, you literally start working in BASIC, and two things must be taken into account. First: any accidental error or bug that occurs during the execution of the routine will cause control to be automatically returned to the BASIC command level. To prevent this from happening, the HERRO hook (FFB1H) can be used to trap the error. The error code is stored in the E register and can be used by the assembly routine. Second: a BASIC command should only be called if the algorithm to be used is very complex, such as CIRCLE, LINE, DRAW, PLAY instructions and others with complex execution. BIOS routines should always be preferred when they can do the same job, as they are much faster and easier to use than BASIC routines.

To call a BASIC command, it is usually enough to set in the HL register the address of a false BASIC line terminated by a 00H byte, pre-

ferably in tokenized form. However, some commands require more registers and even system variables to be loaded, but these commands are useless for assembly programs. To get the tokenized form of the command, there is a simple technique: just type the desired program line and then use a monitor (dump) program to observe the tokenized line.

After setting the HL register, the BIOS CALBAS routine must be used to execute the command. The routines CALSLT (001CH) or CALLF (0030H) can also be used, setting the Main-ROM slot to IY.

To use the command, it is necessary to know at which address the routine that executes it is. This can be done by consulting an address table that starts at 392EH, where the addresses it points to follow in ascending order of command token. Functions also have their table, starting with 39DEH. The addresses indicated by these tables, despite not being standardized, remained the same in all MSX models. It is concluded that there is no need to consult the table to ensure compatibility, just set the routine address directly. However, nothing prevents the table from being queried. Below is a list of all BASIC commands and functions with their respective tokens (including the set token preceded by FFH in the case of functions and the special tokens of the ELSE and REM commands), table address and entry points.

Command	Token	Function Token	Table Adress	Routine Adress
>	EEH	-	Afat	-
=	EFH	-	Afat	-
<	F0H	-	Afat	-
+	F1H	-	Afat	-
-	F2H	-	Afat	-
*	F3H	-	Afat	-
/	F4H	-	Afat	-
^	F5H	-	Afat	-
$	FCH	-	Afat	-
ABS	06H	FF86H	39E8H	2E82H
AND	F6H	-	Afat	-
ASC	15H	FF95H	3A06H	680BH
ATN	0EH	FF8EH	39F8H	2A14H
ATTR$	E9H	-	Afat	7C43H
AUTO	A9H	-	3973H	49B5H
BASE	C9H	-	39BEH	7B5AH

BEEP	C0H	-	39ACH	00C0H
BIN$	1DH	FF9DH	3A16H	6FFFH
BLOAD	CFH	-	39CAH	6EC6H
BSAVE	D0H	-	39CCH	6E92H
CALL	CAH	-	39C0H	55A8H
CDBL	20H	FFA0H	3A1CH	303AH
CHR$	16H	FF96H	3A08H	681BH
CINT	1EH	FF9EH	3A18H	2F8AH
CIRCLE	BCH	-	39A4H	5B11H
CLEAR	92H	-	3950H	64AFH
CLOAD	9BH	-	3962H	703FH
CLOSE	B4H	-	3994H	6C14H
CLS	9FH	-	396AH	00C3H
CMD	D7H	-	39DAH	7C34H
COLOR	BDH	-	39A6H	7980H
CONT	99H	-	395EH	6424H
COPY	D6H	-	39D8H	7C2FH
COS	0CH	FF8CH	39F4H	2993H
CSAVE	9AH	-	3960H	6FB7H
CSNG	1FH	FF9FH	3A1AH	2FB2H
CSRLIN	E8H	-	Afat	790AH
CVD	2AH	FFAAH	3A30H	7C70H
CVI	28H	FFA8H	3A2CH	7C66H
CVS	29H	FFA9H	3A2EH	7C6BH
DATA	84H	-	3934H	485BH
DEF	97H	-	395AH	501DH
DEFDBL	AEH	-	3988H	4721H
DEFINT	ACH	-	3984H	471BH
DEFSNG	ADH	-	3986H	471EH
DEFSTR	ABH	-	3982H	4718H
DELETE	A8H	-	397CH	53E2H
DIM	86H	-	3938H	5E9FH
DRAW	BEH	-	39A8H	5D6EH
DSKF	26H	FFA6H	3A28H	7C39H
DSKI$	EAH	-	Afat	7C3EH
DSKO$	D1H		39CEH	7C16H
ELSE	A1H	3AA1H	396EH	485DH
END	81H		396EH	63EAH
EOF	2BH	FFABH	3A32H	6D25H
EQV	F9H	-	Afat	-
ERASE	A5H		3976H	6477H
ERL	E1H	-	Afat	4E0BH

ERR	E2H	-	Afat	4DFDH
ERROR	A6H	-	3978H	49AAH
EXP	0BH	FF8BH	39F2H	2B4AH
FIELD	B1H	-	398EH	7C52H
FILES	B7H	-	39AAH	6C2FH
FIX	21H	FFA1H	3A1EH	30BEH
FN	DEH	-	Afat	5040H
FOR	82H	-	3920H	4524H
FPOS	27H	FFA7H	3A2AH	6D39H
FRE	0FH	FF8FH	39FAH	69F2H
GET	B2H	-	3990H	775BH
GOSUB	8DH	-	3948H	47B2H
GOTO	89H	-	393EH	47E8H
GO TO	89H	-	393EH	47E8H
HEX$	1BH	FF9BH	3A12H	65FAH
IF	8BH		3942H	49E5H
IMP	FAH	-	3A20H	7940H
INKEY$	ECH	-	Afat	7347H
INP	10H	FF90H	39FCH	4001H
INPUT	85H	-	3936H	4B6CH
INSTR	E5H	-	39F6H	29FBH
INT	05H	FF85H	39E6H	30CFH
IPL	D5H	-	39D6H	7C2AH
KEY	CCH	-	3964H	786CH
KILL	D4H	-	39D4H	7C25H
LEFT$	01H	FF81H	39DEH	6861H
LEN	12H	FF92H	3A00H	67FFH
LET	88H	-	393CH	4880H
LFILES	BBH	-	39A2H	6C2AH
LINE	AFH	-	398AH	4B0EH
LIST	93H	-	3952H	522EH
LLIST	9EH	-	3968H	5229H
LOAD	B5H	-	3996H	6B5DH
LOC	2CH	FFACH	3A34H	6D03H
LOCATE	D8H	-	39DCH	7766H
LOF	2DH	FFADH	3A36H	6D14H
LOG	0AH	FF8AH	39F0H	2A72H
LPOS	1CH	FF9CH	3A14H	4FC7H
LPRINT	9DH	-	394CH	4A1DH
LSET	B8H	-	399CH	7C48H
MAX	CDH	-	39C6H	7E4BH
MERGE	B6H	-	3998H	6B5EH

MID$	03H	FF83H	39E2H	689AH
MKD$	30H	FFB0H	3A3CH	7C61H
MKI$	2EH	FFAEH	3A38H	7C57H
MKS$	2FH	FFAFH	3A3AH	7C5CH
MOD	FBH	-	Afat	-
MOTOR	CEH	-	39C8H	73B7H
NAME	D3H	-	39D2H	7C20H
NEW	94H	-	3954H	6286H
NEXT	83H	-	3932H	6527H
NOT	E0H	-	Afat	-
OCT$	1AH	FF9AH	3A10H	7C70H
OFF	EBH	-	3A02H	3A02H
ON	95H	-	3956H	48E4H
OPEN	B0H	-	398CH	6AB7H
OR	F7H	-	Afat	-
OUT	9CH	-	3964H	4016H
PAD	25H	FFA5H	3A26H	7969H
PAINT	BFH	-	39AAH	59C5H
PDL	24H	FFA4H	3A24H	795AH
PEEK	17H	FF97H	3A0AH	541CH
PLAY	C1H	-	39AEH	73E5H
POINT	EDH	-	Afat	5803H
POKE	98H	-	395CH	5423H
POS	11H	FF91H	39FEH	4FCCH
PRESET	C3H	-	39B2H	57E5H
PRINT	91H	-	394EH	4A24H
PSET	C2H	-	39B0H	57EAH
PUT	B3H	-	3992H	7758H
READ	87H	-	393AH	4B9FH
REM	8FH	3A8FH	394AH	485DH
RENUM	AAH	-	3980H	5468H
RESTORE	8CH	-	3944H	63C9H
RESUME	A7H	-	397AH	495DH
RETURN	8EH	-	3948H	4821H
RIGHT$	02H	FF82H	39E0H	6891H
RND	08H	FF88H	39ECH	2BDFH
RSET	B9H	-	399EH	7C4DH
RUN	8AH	-	3940H	479EH
SAVE	BAH	-	39A0H	6BA3H
SCREEN	C5H	-	39B6H	79CCH
SET	D2H	-	39D0H	7C1BH
SGN	04H	FF84H	39E4H	2E97H

SIN	09H	FF89H	39EEH	29ACH
SOUND	C4H	-	39B4H	73CAH
SPACE$	19H	FF99H	3A0EH	6848H
SPC(	DFH	-	Afat	-
SPRITE	C7H	-	39BAH	7A48H
SQR	07H	FF87H	39EAH	2AFFH
STEP	DCH	-	Afat	-
STICK	22H	FFA2H	3A20H	7940H
STOP	90H	-	394CH	63E3H
STR$	13H	FF93H	3A02H	6604H
STRIG	23H	FFA3H	3A22H	794CH
STRING$	E3H	-	Afat	6829H
SWAP	A4H	-	3974H	643EH
TAB(	DBH	-	Afat	-
TAN	0DH	FF8DH	39F6H	29FBH
THEN	DAH	-	Afat	-
TIME	CBH	-	39C2H	7911H
TO	D9H	-	Afat	-
TROFF	A3H	-	3972H	6439H
TRON	A2H	-	3970H	6438H
USING	E4H	-	Afat	-
USR	DDH	-	Afat	4FD5H
VAL	14H	FF94H	3A04H	68BBH
VARPTR	E7H	-	39FAH	4E41H
VDP	C8H	-	39BCH	7B37H
VPEEK	18H	FF98H	3A0CH	7BF5H
VPOKE	C6H	-	39B8H	7BE2H
WAIT	96H	-	3958H	401CH
WIDTH	A0H	-	396CH	51C9H
XOR	F8H	-	Afat	-

Not all commands are in the ROM tables and some don't even have their own routines for execution. These commands are marked with the expression "Afat", as they are executed directly by the standard routine in 4DC7H (Factor Evaluator). In particular, ELSE and REM command tokens are preceded by the 3AH byte (":") and all function tokens (tokens smaller than 80H) have their bit 7 set and are preceded by the FFH byte in the BASIC text.

At the beginning of this section it was said that the BASIC line should preferably be in tokenized form. However, it is possible to use it in ASCII form. The only care in this case is to replace ten key characters

with the respective tokens, and the rest of the text can be in ASCII form. These characters with their respective tokens are:

```
' E6H    = EFH    + F1H    * F3H    ^ F5H

> EEH    < F0H    - F2H    / F4H    $ FCH
```

So, for example, a line of BASIC text type:

```
LINE (10,10)-(50,50),1
```

should be placed on the line in machine code as follows:

```
DEFB ´(10,10)´,0F2H,´(50,50),1´,000H
```

However, if any of the key characters are enclosed in double quotes in BASIC text, as in the DRAW or PLAY commands, they must be kept in their original form. For example:

```
PLAY "A-BC+"      in assembly will be:

DEFB '"A-BC+"',000H
```

A great place to put the text to be executed is in the system variable KBUF (F41FH), for two reasons: it's used by the interpreter for exactly that and it's on page 3, so it can be executed from DOS without problems. A practical example with the CIRCLE statement is illustrated below. This routine works under both DOS and BASIC, at any address.

```
CIRCLE: EQU  05B11H
INIGRP: EQU  00072H
CHGET:  EQU  0009FH
CALSLT: EQU  0001CH
SLTROM: EQU  0FCC1H
KBUF:   EQU  0F41FH
        LD   HL,LINBAS
        LD   DE,KBUF
        LD   BC,12 LDIR
        LD   IX,INIGRP
        LD   IY,(SLTROM+1)
        CALL CALSLT
        LD   HL,KBUF
        LD   IX,CIRCLE
```

```
        LD   IY,(SLTROM+1)
        CALL CALSLT
        LD   IX,CHGET
        LD   IY,(SLTROM+1)
        CALL CALSLT
        RET
LINBAS: DEFB ´(128,96),70´,000H
```

3.3.8 – Error messages

In the ROM, there is a table that contains a list of BASIC error messages, starting at 3D75H. Each of these is stored as text, terminated with a 00H byte. Associated error codes are shown below for reference only as they are not part of the table.

```
01 NEXT without FOR
02 Syntax error
03 RETURN without GOSUB
04 Out of DATA
05 Illegal function call
06 Overflow
07 Out of memory
08 Undefined line number
09 Subscript out of range
10 Redimensioned array
11 Division by zero
12 Illegal direct
13 Type mismatch
14 Out of string space
15 String too long
16 String formula too complex
17 Can't CONTINUE
18 Undefined user function
19 Device I/O error
20 Verify error
21 No RESUME
22 RESUME without error
23 Unprintable error
24 Missing operand
25 Line buffer overflow
50 FIELD overflow
51 Internal error
```

```
52 Bad file number
53 File not found
54 File already open
55 Input past end
56 Bad file name
57 Direct statement in file
58 Sequential I/O only
59 File not OPEN
```

Chapter 4
THE RAM MEMORY

The Z80 CPU can directly access a maximum of 64 Kbytes of memory. This amount of memory was already insufficient for many applications even in 1983 when the MSX standard was created. In view of this fact, some systems were developed to increase the amount of memory that the Z80 can access.

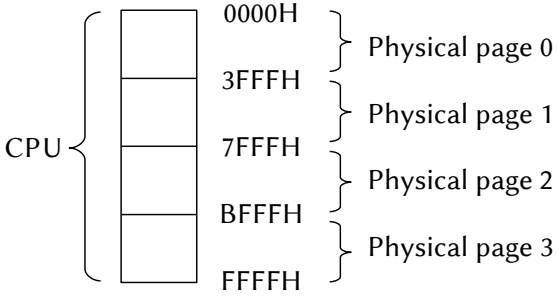
4.1 – MEMORY EXPANSIONS

The first memory expansion system that was developed for the MSX was the slots and pages scheme, which allowed the Z80 to access a theoretical maximum of 1 Mbyte. The slot and page system was great for upgrading hardware, but too complex to function as a memory expansion. Very few software came to use it.

In 1985, with the release of MSX2, a new concept of memory expansion was created, the Memory Mapper, easy to manipulate, which allowed the connection of up to 4 Mbytes in each slot.

4.1.1 – Memory Mapper

Memory Mapper uses the Z80's I/O ports to complement the address bus. Four ports are used, from FCH to FFH, one for each physical page. Physical pages are the four 16 Kbyte pages that can be active at the same time, each at different addresses, as shown in the illustration below:



For each physical page there is a corresponding I/O port, as illustrated below:

Physical page 0 ⟶ FCH port
Physical page 1 ⟶ FDH port
Physical page 2 ⟶ FEH port
Physical page 3 ⟶ FFH port

The value that can be written to a Z80 port ranges from 0 to 255 and each value defines a logical page. This way you can have up to 256 logical pages. In memory-mapped I/O ports are used to associate a logical page with a physical page; therefore each page, both physical and logical, has 16 Kbytes. So, to know the total that this method can handle, just do 16 Kbytes times 256, which gives 4 Mbytes in each slot. As each slot is handled differently and there can be up to 4 Mbytes of Memory Mapper to each one, we make 4 Mbytes times 16, which gives a maximum theoretical of 64 Mbytes.

On MSX2 a slot with 64 Kbytes of RAM is used and the Memory Mapper must be in another slot. From MSX2+ onwards, the 64 Kbytes of main RAM corresponds to the first 64 Kbytes of Memory Mapper. The initial default selection of pages is as follows:

Physical page 0 = logical page 3
Physical page 1 = logical page 2
Physical page 2 = logical page 1
Physical page 3 = logical page 0

Switching between physical and logical pages is very simple. Simply use a Z80 OUT instruction to position the desired logical page on the corresponding physical page. So, for the initial selection of 64 Kbytes, the following sequence of instructions can be used:

```
OUT 0FCH,3 ; put logical page 3 in the physical page 0
OUT 0FDH,2 ; put logical page 2 in the physical page 1
OUT 0FEH,1 ; put logical page 1 in the physical page 2
OUT 0FFH,0 ; put logical page 0 in the physical page 3
```

With this arrangement, logical page 0 (first page of the Memory Mapper) will contain the routines and system variables and is the only one that can never be moved.

As logical pages always have the same number, eventually a logical page can be on two or more physical pages at the same time. For example, instructions

```
OUT  0FDH,5
OUT  0FEH,5
```

puts logical page 5 on physical pages 1 and 2.

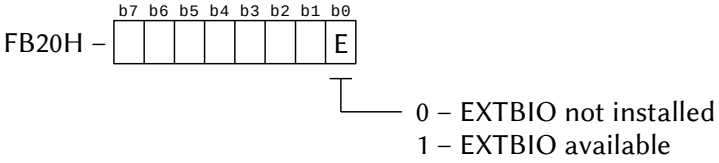
An important note is that the selection of slots and physical pages take precedence over the selection of logical pages. Therefore, when selecting a logical page, it is necessary to make sure that the corresponding physical page is enabled.

Normally only physical pages 1 and 2 are used for the selection of logical pages, since physical page 0 contains the BIOS and physical page 3 contains the system work area and cannot be turned off, otherwise the entire the system.

4.1.1.1 – Managing Memory Mapper

If MSXDOS2 is installed, several functions are available for manipulating the Memory Mapper. These functions are accessed through an additional input called EXTBIO, located at address FFCAH.

If the program is not loaded directly by MSXDOS2, it is necessary to check if the EXTBIO routine is available, which can be done by testing the HOKVLD flag at address FB20H, as follows:



If the program is loaded from MSXDOS2, the above step is unnecessary, as the Kernel already provides the input. To call the function, it is necessary to set the device number in the D register, the function in

the E register and the required parameters in the other registers. The EXTBIO call format can be as follows:

```
EXTBIO: EQU  0FFCAH
        XOR  A          ; A deve ser sempre 0
        LD   D,4        ; ID do dispositivo (mapper)
        LD   E,1        ; function GETMAP
        CALL EXTBIO     ; executa EXTBIO
```

The routine format is as follows:

EXTBIO (FFCAH/Sys)
Function: Access extended BIOS functions.
Input: A – Always 00H.
D – Device ID.
E – Function number.
Output: A – Primary mapper slot ID.
DE – Reserved.
BC, HL, BC', DE', HL', IX, IY – depends on device and function called.

The mapper handling device number is 4 and the functions are as follows:

GETMAP (Device: 04H, function: 01H)
Function: Returns the address of the mapper variable table.
Input: Via EXTBIO (A=00H, D=04H, E=01H).
Output: A – Primary mapper slot ID.
DE – Reserved.
HL – Starting address of the variable table, whose structure is as follows:

```
+00H Primary mapper slot ID.
+01H Total number of 16K segments.
+02H Number of free 16K segments.
+03H Number of 16K segments allocated
     by the system (minimum of 6 for the
     primary mapper).
+04H Number of 16K segments allocated
     to the user.
+05H~+07H Reserved (Always 00H).
```

```
+08H… entries for other mappers in
      other slots. If there is none, it
      will contain 00H.
```

GETSUP (Device: 04H, function: 02H)

Function: Returns several parameters related to the mapper

Input: Via EXTBIO (A=00H, D=04H, E=02H).

Output: A – Total number of segments (logical pages) of the primary mapper.

B – Primary mapper slot ID.

C – Number of free segments (logical pages) in the primary mapper.

DE – Reserved.

HL – Starting address of a table calling mapper support subroutines. The format of this table is as follows:

```
+00H ALL_SEG Allocates a 16K segment.
+03H FRE_SEG Frees a 16K segment.
+06H RD_SEG Read one byte from address
     A:HL to A.
+09H WR_SEG Write the contents of E
     at address A:HL.
+0CH CAL-SEG Inter-segment call
     by address Iyh:IX.
+0FH CALLS Inter-segment call. Inline
     parameters after the CALL statement.
+12H PUT_PH Puts a segment on the
     physical page (HL)
+15H GET_PH Returns the current thread
     to the physical page (HL).
+18H PUT_P0 Puts a segment on physical
     page 0
+1BH GET_P0 Returns the current segment
     of page 0.
+1EH PUT_P1 Puts a segment on physical
     page 1
+21H GET_P1 Returns the current segment
     of page 1.
+24H PUT_P2 Put a segment on physical
     page 2
+27H GET_P2 Returns the current segment
     of page 2.
```

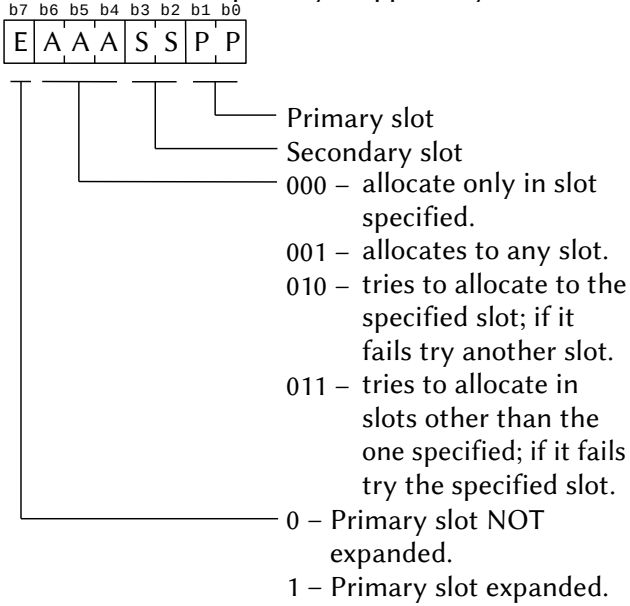
```
+2AH PUT_P3 Not supported as page 3
     cannot be switched. If called, it
     just returns.
+2DH GET_P3 Returns the current segment
     of page 3.
```

Once the starting address of the table is obtained, the extended BIOS subroutines can be called. They are described below.

ALL_SEG (00H/Ext)

Function: Allocate a 16K segment from the mapper.

Input: A – 00H – Allocates a user segment.
01H – Allocates a segment of system.
B – 00H – Allocates to primary mapper only.



Output: CY = 1 ⟶ There are no free segments.
CY = 0 ⟶ Segment allocated.
A = Segment number.
B = Segment Slot ID.

FRE_MON (03H/Ext)

Function: Release a 16K segment from the mapper.

Input: A – Segment number to be released.
B – If it is 00H, release it only on the primary mapper; if it is different from 00H, it releases in any other mapper than the primary one.
Output: CY = 0 ⟶ Segment released.
CY = 1 ⟶ Segment release error.

RD_MON (06H/Ext)
Function: Read a byte from the mapper.
Input: A – Segment number from which the byte will be read.
HL – Address to be read (0000H to 3FFFH).
Output: A – Byte read.
All other registers are preserved.

WR_MON (09H/Ext)
Function: Write a byte to the mapper.
Input: A – Segment number where the byte will be written.
HL – Address to be written (0000H to 3FFFH).
E – Value to write.
Output: A – Corrupted while writing.
All other registers are preserved.

CAL_SEG (0CH/Ext)
Function: Calls a routine in any area of the mapper.
Input: IYh – Segment number to be called.
IX – Address to be called (0000H to FFFFH).
AF, BC, DE and HL can contain parameters for the routine. Do not use AF', BC', DE' and HL' as they are corrupted during the call.
Output: AF, BC, DE, HL, IX and IY can contain valid return values. AF', BC', DE' and HL' return corrupted.

CALLS (0FH/Ext)
Function: Calls a routine in any area of the mapper through inline parameters.
Input: AF, BC, DE and HL can contain parameters for the routine. Do not use AF', BC', DE' and HL' as they are corrupted during the call. The call sequence must be in the following format:

```
        CALL CALLS
        DEFB SEGMENT
        DEFW ADRESS
```

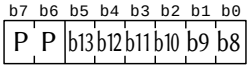
Output: AF, BC, DE, HL, IX and IY can contain valid return values. AF', BC', DE' and HL' return corrupted.

PUT_PH (12H/Ext)

Function: Enables a mapper segment on a physical page.

Input: A – Mapper segment

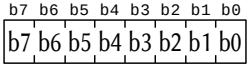
H –



Relative adress (MSB)*

Physical page (0 to 3)

L –



Relative adress (LSB)*

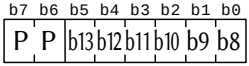
* The relative address is optional.

Output: None. All registers are preserved.

GET_PH (15H/Ext)

Function: Returns the current active segment on a physical page.0

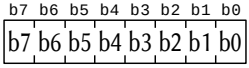
Input: H –



Relative adress (MSB)*

Physical page (0 to 3)

L –



Relative adress (LSB)*

* The relative address is optional.

Output: A – Segment number.
All other registers are preserved.

PUT_P0 (18H/Ext)

Function: Enables a mapper segment on physical page 0.

Input: A – Segment number to be enabled.

Output: None. All registers are preserved.

GET_P0 (1BH/Ext)
Function: Returns the active segment on physical page 0.
Input: None.
Output: A – Active segment number.
All other registers are preserved.

PUT_P1 (1EH/Ext)
Function: Enables a mapper segment on physical page 1.
Input: A – Segment number to be enabled.
Output: None. All registers are preserved.

GET_P1 (21H/Ext)
Function: Returns the active segment on physical page 1.
Input: None.
Output: A – Active segment number.
All other registers are preserved.

PUT_P2 (24H/Ext)
Function: Enables a mapper segment on the physical page 2.
Input: A – segment number to be enabled.
Output: None. All registers are preserved.

GET_P2 (27H/Ext)
Function: Returns the active segment on the physical page 2.
Input: None.
Output: A – Active segment number.
All other registers are preserved.

PUT_P3 (2AH/Ext)
Not supported as physical page 3 cannot be swapped.
A call to this function has no effect.

GET_P3 (2DH/Ext)
Function: Returns the active segment on physical page 0.
Input: None.
Output: A – active segment number.
All other registers are preserved.

4.1.2 – Megaram

Despite not being officially recognized as a memory expansion for MSX, Megaram is quite popular in Brazil. It was designed to be able to run megarom games without having to convert them to Memory Mapper.

Megaram also involves the concept of logical and physical pages, but its operation is more complicated than that of Memory Mapper. Each Megaram logical page has 8 Kbytes and as up to 256 logical pages can be defined, the maximum possible memory that can be plugged into each slot is 2 Mbytes.

The management of Megaram pages is done through the 08EH port of the Z80. To enable Megaram, the following instruction must first be executed:

```
OUT  (08EH),A
```

The value of A is unimportant. This instruction only tells Megaram that it will be used. As each Megaram logical page is only 8 Kbytes, two logical pages are required for each physical page. Each logical page can start at one of the following addresses:

```
4000H – 6000H – 8000H – A000H
```

After executing the instruction "OUT (08EH),A", you must load the desired number of the Megaram logical page in A and execute the instruction "LD (xxxxH),A", where "xxxxH" is the initial address of the logical page on the physical page. To place Megaram logical pages 0 and 1 on physical page 1 of memory, execute the following instructions:

```
OUT  (08EH),A   ;enables the megaram
LD   A,0        ;select logical page 0
LD   (04000H),A ;put logical page 0 in 4000H
LD   A,1        ;select logical page 1
LD   (06000H),A ;put logical page 1 in 6000H
```

Executing these instructions, Megaram's logical pages 0 and 1 will be occupying the physical page 1 of the PC, and will be ready to be read, but not to be written. In order to write data to Megaram, the instruction "IN A,(08EH)" must be executed. Below is the sequence of instructions that place Megaram pages 0 and 1 on physical page 1 and enable them to be read and written.

```
OUT   (08EH),A    ;enables the megaram
LD    A,0         ;select logical page 0
LD    (04000H),A ;put logical page 0 in 4000H
LD    A,1         ;select logical page 1
LD    (06000H),A ;put logical page 1 in 6000H
IN    A,(08EH)    ;enable writing and reading
```

When executed, this routine places Megaram's logical pages 0 and 1 on physical page 1 and enables them to be read and written. As with Memory Mapper, the selection of physical pages takes precedence over logical pages; therefore, to enable logical pages, the corresponding physical page must be enabled.

For Megaram there is no standard routine for its management. All access is direct and management is the responsibility of the programmer.

4.1.3 – Megaram x Memory Mapper

Both Megaram and Memory Mapper must be recognized by the software that uses it. There is no BIOS routine for directly handling these expansions. In BDOS there is a way to search for free logical pages of the Memory Mapper and select them.

A question that may arise for programmers is which memory expansion to use: Megaram or Memory Mapper. As already described, Memory Mapper is the standard MSX expansion; however Megaram is very popular in Brazil.

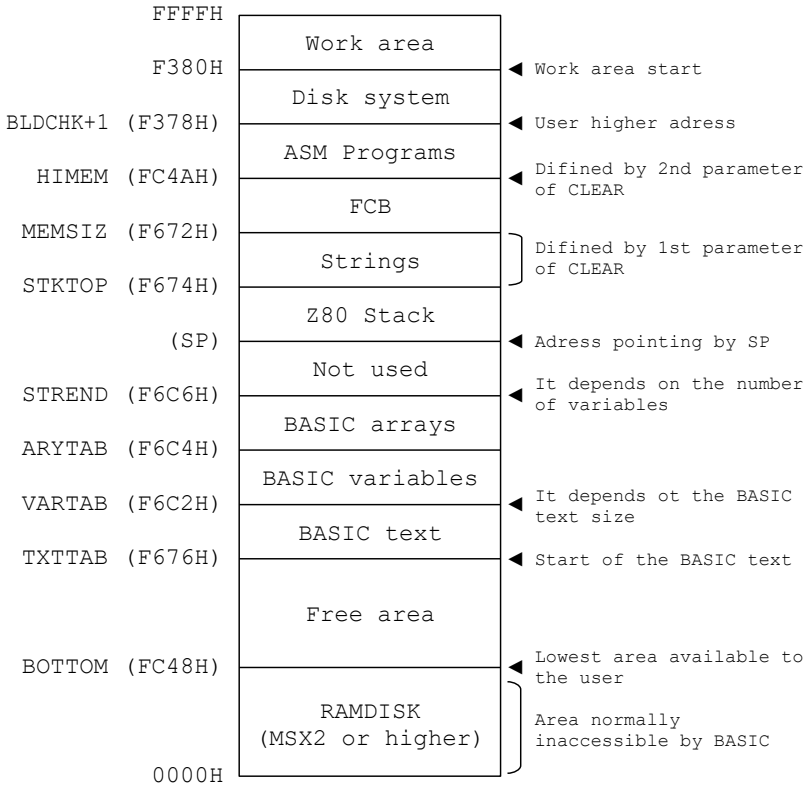
A reasonable solution to this issue is for the developed programs to recognize both expansions. First, the program should look for Memory Mapper, as it is the default expansion. If it is not found, a search is made for Megaram. It must be taken into account that Megaram is almost non-existent in other countries.

4.2 – RAM MAPPING

Regardless of memory slots, pages and expansions, there is a specific mapping for RAM, residing at the top of physical page 3. Although the lower addresses are also mapped, there are no problems switching between physical pages 0, 1 and 2, as long as that due care is

taken, such as, for example, not turning off the page where the program is running. Physical page 3 should never be turned off as it contains the system work area.

Upon entering BASIC, right after a reset, the RAM is mapped as illustrated below.



The hexadecimal value next to the system variables above are their addresses; are the variables, or the values contained in the respective addresses, that point to the actual address in the mapping shown above.

This is the default mapping of MSX2/2+/TR with disk drive. Without a disk drive, just disregard the respective area (the disk area will be described in detail in the chapter 7 – Mass Storage Systems). For

MSX1, RAMDISK should be disregarded. All areas related to BASIC (from TXTTAB to MEMSIZ) and the area for strings are described in chapter 1, section “BASIC Interpreter”. The area for assembly programs is set by the CLEAR command and is reserved for user routines; the interpreter will not interfere with it unless instructed to do so (USR function or command expansion).

4.2.1 – The FCB (File Control Block)

The FCB (File Control Block) is a 265-byte buffer used for communication with peripherals.

Standard peripherals have the following names:

A: to H: – drives (A: to F: for MSXDOS1)
AUX: – auxiliary (Ex. RS232C, DOS only)
CAS: – cassette interface
CON: – (DOS only) – keyboard or text screen
CRT: – text screen
GRP: – graphic display
LPT: – (BASIC) or PRN: / LST: (DOS) – printer
MEM: – Lower 32K ramdisk (BASIC, MSX2 or higher only)
NUL: – null (no effect, DOS only)

Every time BASIC or DOS refers to one of these devices, an FCB is opened. In BASIC, up to 15 FCBs can be opened simultaneously, specified by the MAXFILES command. In DOS, the maximum number depends on the available memory, and FCB's can be opened by the BUFFERS command. On reset, only one FCB is allocated for BASIC and five for DOS. The format of the FCB is illustrated below.

Offset	Label	Description
+0	FL.MOD	Open file mode
+1	FL.FCA	Pointer to the BDOS FCB (low)
+2	FL.LCA	Pointer to the BDOS FCB (high)
+3	FL.LSA	Backup character
+4	FL.DSK	Device number
+5	FL.SLB	Internal use of the interpreter
+6	FL.BPS	Location of FL.BUF

+7	FL.FLG	Information Flag
+8	FL.OPS	Virtual Head Information
+9...	FL.BUF	Start of 256-byte buffer

4.2.2 – The workspace

The system workspace ranges from adress F380H to FFFFH. The use of this area by the programmer must be well controlled, under penalty of undesirable alterations in the basic functions of the computer or even a total stoppage of the system. This area is mapped as shown in the illustration below.

FFFFH	Slot select
FFFDH ~ FFFEH	Reserved
FFFAH ~ FFFCH	V9958 registers
FFF8H ~ FFF9H	Reserved
FFF7H	MainROM slot
FFE7H ~ FFF6H	V9938 registers
FFCFH ~ FFE6H	BIOS expansion routines
FFCAH ~ FFCEH	BIOS expansion hooks
FD9AH ~ FFC9H	Hooks
F39AH ~ FD99H	System variables
F380H ~ F399H	Inter-slot routines

4.2.2.1 – Inter-slot subroutines

The area between F380H and F399H contains some routines for accessing other slots and pages. These routines can access any area as long as physical page 3 is active. They have remained the same from MSX1 to MSX turbo R and can be considered standard because of that. Are the following:

RDPRIM (F380H)
  Function: Read a byte of any adress from any slot.
  Input: A – Primary slot to be read.
  D – Current slot for return.
  Output: E – Byte read.

WRPRIM (F385H)
  Function: Write a byte to any adress of any slot.
  Input: A – Primary slot to be read.
  D – Current slot for return.
  E – Byte to be written.
  Output: None.

CLPRIM (F38CH)
  Function: Call an adress in any slot.
  Input: A – Primary slot that contains the routine.
  IX – Address to be called.
  PUSH AF – Current slot for return (at A).
  Output: Depends on the called routine.

Note that some input values are unconventional. This is because these routines were specifically written to be used by the standard BIOS routines RDSLT (000CH), WRSLT (0014H), and CALSLT (001CH). The ASM code of these routines is as follows:

```
F380H  RDPRIM: OUT  (0A8H),A  ;select primary slot
F382H          LD   E,(HL)    ;read byte
F383H          JR   WRPRM1    ;restore primary slot
F385H  WRPRIM: OUT  (0A8H),A  ;select primary slot
F387H          LD   (HL),E    ;write byte
F388H  WRPRM1: LD   A,D       ;A = previous slot
F389H          OUT  (0A8H),A  ;restore primary slot
F38CH  CLPRIM: OUT  (0A8H),A  ;select primary slot
F38EH          EX   AF,AF'    ;exchange AF with AF'
F38FH          CALL CLPRM1    ;calls routine
F392H          EX   AF,AF'    ;exchange AF with AF'
F393H          POP  AF        ;A = previous slot
F394H          OUT  (0A8H),A  ;restore primary slot
F396H          EX   AF,AF'    ;exchange AF with AF'
F397H          RET            ;return
F398H  CLPRM1: JP   (IX)      ;call adress (IX)
```

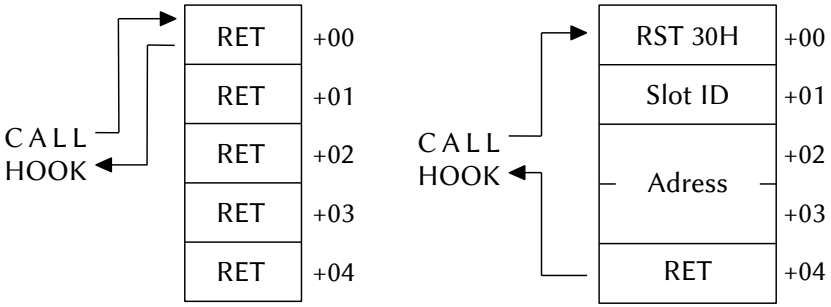
4.2.2.2 – The hooks

The work area between FD9AH and FFC9H is the area that contains the hooks. Each hook consists of five bytes that are normally filled with the value C9H (RET instruction).

The hooks are called strategic BIOS positions so that BIOS/Interpreter operations can be modified or extended. Each hook has enough space for a distant call to any slot. As they are initially filled with RET instructions, this only causes a return to the BIOS. However, it can be modified to call a routine in any slot.

It is not necessary for the hook to be populated in order to access routines in other slots. If the operation to be performed fits in five bytes, the hook, by itself, can constitute the code to be executed.

To access routines in other slots, the hooks must be modified as illustrated below.



A suggested ASM routine for changing hooks is illustrated below.

```
HOOK: EQU  XXXXH
      LD   A,0F7H          ; F7H = RST 030H statement
      LD   (HOOK),A
      LD   A,(ID_SLOT)     ; A <- slot ID of the routine
      LD   (HOOK+1),A
      LD   HL,(Adress)     ; HL <- Adress of the routine
      LD   (HOOK+2),HL
      LD   A,0C9H          ; C9H = RET statement
      LD   (HOOK+4),A
```

Chapter 5
THE VIDEO AND THE VDP

MSX machines, with their constant evolution, needed more and more graphics capacity. Thus, the MSX1 uses the VDP (Video Display Processor) TMS9918A or TMS9928A, which had only 16 colors and had few resources, representing a very limiting factor to the graphics processing of these machines. Just two years after its release, the MSX2 appeared in 1985, with the new VDP V9938, fully compatible with the TMS9918A. Later, in 1988, the MSX2+ appeared, with the V9958. Later, the V9990 was released, but it was not fully compatible with the V9958 and was not used in any MSX models, but was used in some video cartridges. The V9958 was also used in the MSX turbo R models, released in 1990 and 1991.

5.1 – THE MSX-VIDEO

The MSX1 uses the TMS9918A for video signal generation, whose main features are as follows:

- Maximum resolution of 256 x 192 pixels;
- Maximum of 16 colors displayed simultaneously;
- Text mode of 40 columns by 24 lines;
- Displays up to 32 monochrome sprites on the screen, with a maximum of 4 in a horizontal line;
- 16 Kbytes of VRAM

The V9938, used on the MSX2, has the following main features:

- Palette of 512 colors;
- Maximum resolution of 512 x 424 points with 16 colors;
- Maximum of 256 colors displayed simultaneously;
- Easy-to-handle bit-mapped graphics modes;
- Text mode of 80 characters per line with blink feature;
- Line, search and movement of areas implemented via hardware;
- Displays up to 32 sprites on the screen, with a maximum of 8 sprites on the same horizontal line;
- Each line of each sprite can have a different color;
- Memory addresses can be represented by graphical coordinates;
- Logic operation functions;

- Hardware thin vertical scroll;
- Internal scanning capability;
- Internal "superimpose" capability;
- Up to 128 Kbytes of VRAM.

The V9958, used on the MSX2+ and MSX turbo R, added new features to the video, including:

- Maximum of 19268 colors displayed simultaneously;
- External synchronization capability;
- Possibility of multiple MSX-VIDEO configurations;
- External color palettes can be added using color-bus output;
- Hardware vertical and horizontal fine scrolling;
- 5-bit DAC per primary color.

V9990 does not have Screens 0 to 4, and Screens 5 to 12 have been modified so that they are not fully compatible with V9958. However, it has many more modes, as well as being much faster (it is 9 to 17 times the speed of the V9958, depending on the mode) and accessing up to 512K of VRAM. Its main features are:

- Maximum resolution of 768 x 480 points (overscan mode);
- Maximum of 32768 colors displayed simultaneously;
- Palette of up to 64 colors selected from 32768;
- Smooth scroll in all directions;
- Displays up to 2 plans simultaneously;
- Internal scanning capability;
- Internal "superimpose" capability;
- Available hardware commands;
- Displays up to 125 sprites on the screen with up to 16 each horizontal line;
- Each point of each sprite can have a different color;
- Additional VGA resolution of 640 x 400 and 640 x 480 with up to 16 colors selected from 32768.

5.1.1 – Description of the registers (V9918/38/58)

The TMS9918A has 9, the V9938 has 49, and the V9958 has 52 internal registers to control video operations. These registers are divided into three groups. Control group and status group can be accessed di-

rectly by BASIC and BIOS. The third group, non-existent in the TMS9918A, is the palettes, and cannot be accessed directly.

The control register group is numbered from R#0 to R#7 for the TMS9918A, from R#0 to R#23 and from R#32 to R#46 for the V9938 and there are three more, R#25 to R #27 for the V9958. They are 8-bit write-only registers (to obtain their values, there is a copy of them on the system work area). The subgroup from R#0 to R#2710 are registers that control all screen modes. The other subgroup, R#32 through R#46, executes VDP hardware commands. These commands will be described in detail later. The tables below briefly describe the functions of each register in the control and status groups.

R#0	VDP(0)	Mode Register #0.
R#1	VDP(1)	Mode Register #1.
R#2	VDP(2)	Address of the pattern name table.
R#3	VDP(3)	Address of the color table (low).
R#4	VDP(4)	Address of the pattern generator table.
R#5	VDP(5)	Address of the the sprites attribute table (low).
R#6	VDP(6)	Address of the sprite pattern table.
R#7	VDP(7)	Border and character color in text mode.

The following registers are only available for V9938 or higher:

R#8	VDP(9)	Mode #2 Register.
R#9	VDP(10)	Mode #3 Register.
R#10	VDP(11)	Color table address (high).
R#11	VDP(12)	Address from the sprites attribute table (high).
R#12	VDP(13)	Character color for function blink.
R#13	VDP(14)	Blinking period.
R#14	VDP(15)	VRAM access address (high).
R#15	VDP(16)	Indirect specification for S#n (preset: 0).
R#16	VDP(17)	Indirect specification for P#n (preset: 0).
R#17	VDP(18)	Indirect specification for R#n (preset: 0).
R#18	VDP(19)	Screen adjustment (SET ADJUST).
R#19	VDP(20)	Examine line when interrupt occurs.
R#20	VDP(21)	Color burst for phase 0 (preset 00 000 000B).
R#21	VDP(22)	Color Burst for Phase 1/3 (preset 00 111 011B).
R#22	VDP(23)	Color Burst for Phase 2/3 (preset 00 000 101B).
R#23	VDP(24)	Vertical scroll.

R#24 Register R#24 does not exist.
R#25 VDP(26) Mode #4 register (V9958 only).
R#26 VDP(27) Horizontal scroll (only V9958).
R#27 VDP(28) Fine horizontal scroll (V9958 only).
R#xx Registers R#28 to R#31 does not exist.
R#32 VDP(33) SX: initial horizontal coordinate (low).
R#33 VDP(34) SX: initial horizontal coordinate (high).
R#34 VDP(35) SY: initial vertical coordinate (low).
R#35 VDP(36) SY: initial vertical coordinate (high).
R#36 VDP(37) DX: destination horizontal coordinate (low).
R#37 VDP(38) DX: destination horizontal coordinate (high).
R#38 VDP(39) DY: destination vertical coordinate (low).
R#39 VDP(40) DY: destination vertical coordinate (high).
R#40 VDP(41) NX: number of points to transfer in the direction horizontal (low).
R#41 VDP(42) NX: number of points to transfer in the direction horizontal (high).
R#42 VDP(43) NY: number of points to transfer in the direction vertical (low).
R#43 VDP(44) NY: number of points to transfer in the direction vertical (high).
R#44 VDP(45) CLR: data transfer to CPU.
R#45 VDP(46) ARGT: argument register.
R#46 VDP(47) CMR: send hardware command to VDP.

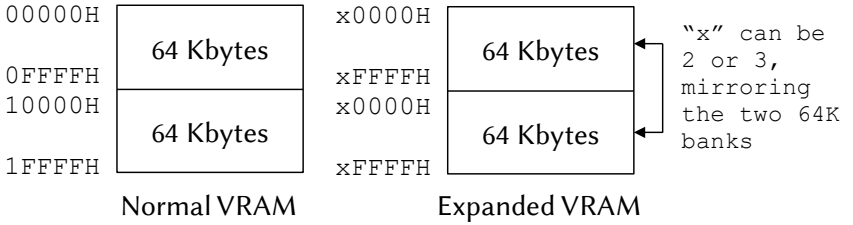
The next group is the state register group. They are read-only 8-bit registers designated by S#0 to S#9. The only one present on the TMS9918 is the S#0. The listing below briefly describes its functions:

S#0 VDP(8) Interrupt information.
S#1 VDP(–1) Interrupt information.
S#2 VDP(–2) Information and control register.
S#3 VDP(–3) Detected horizontal coordinate (low).
S#4 VDP(–4) Detected horizontal coordinate (high).
S#5 VDP(–5) Vertical coordinate detected (low).
S#6 VDP(–6) Vertical coordinate detected (high).
S#7 VDP(–7) Data obtained by some VDP command.
S#8 VDP(–8) Horizontal coordinate obtained by some search command (low).
S#9 VDP(–9) Vertical coordinate obtained by some search command (low).

5.1.2 – The VRAM (V9918/38/58)

The TMS9918A can only be connected to 16 Kbytes of memory. The V9938 can be connected to 64 or 128 Kbytes and the V9958 must be connected to 128 Kbytes, without which the added screen modes cannot be used. This memory is controlled by the VDP and cannot be accessed directly by the CPU; that's why it's called VRAM (Video RAM).

For the V9938 and V9958 one more bank of 64 Kbytes of expansion can be connected. However, this expansion is only for storing data, as its content is not displayed on the screen. Also, due to hardware limitations, it can only be used properly on screens up to 6. Screens from 7 and up require an access speed that the additional bank cannot support. For these screens, in the case of the standard 128K and due to the low speed of the memories, the VDP stores the even bytes of the image in a 64K bank and the odd bytes in another bank, making it impossible to use the additional 64K in this situation. The 64K expansion is mirrored, as shown in the illustration below:



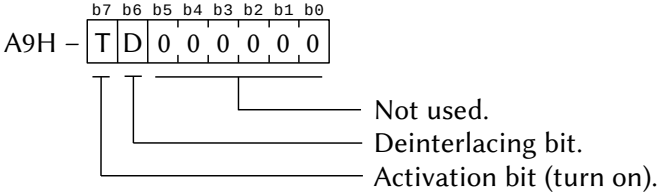
5.1.3 – ADVRAM (V9938/58)

The ADVRAM was a peripheral developed in Brazil that allows the CPU to directly access the VRAM. For this, it is connected directly to a slot, which in this case is where the Main-ROM is. It always works on page 2 (8000H to BFFFH) of that slot.

As a physical page has only 16 Kbytes, it is necessary to make a switch in order to access the 128 Kbytes of VRAM. This switching is done in exactly the same way as for Memory Mapper, by using the same I/O port (0FEH). ADVRAM is a special Memory Mapper, mirroring VRAM. The portion of VRAM accessed by the OUT command is described below:

```
OUT (0FEH),0    ; 00000H to 03FFFH
OUT (0FEH),1    ; 00000H to 03FFFH
 |    |   |        |        |
OUT (0FEH),7    ; 1C000H to 1FFFFH
```

To prevent application programs from confusing ADVRAM with normal Memory Mapper, it is disabled on reset. To activate it, there is a control register, which can be accessed through the I/O port 9AH. The structure of this register is illustrated below.



The enable bit must be set to enable ADVRAM. Its value at startup is 0; therefore on reset the ADVRAM is disabled. The deinterlacing bit is used to make the ADVRAM behave linearly when using the VDP interlaced mode, making programming easier.

To write data to the control register, the following sequence of instructions must be used:

```
LD   A,valor
IN   A,(09AH)
```

There was no error; an IN instruction is even used to write to the control register. The Z80, when reading data through an I/O port, places the contents of the A register on the address lines of A8~A15 and the device adress on the lines A0~A7. This feature allows writing one piece of data while reading another. In this case, ADVRAM reads the contents of the A register present in the lines of adress A8~15 and then writes it to the control register.

5.1.4 – VDP access ports (V9918/38/58)

The TMS9918 has two access ports while the V9938 and V9958 have four ports for CPU communication. The functions of these ports are listed in the table below. They are expressed by "**r**" and "**w**" and their

values are stored, respectively, in adresses 0006H and 0007H of the Main-ROM.

```
r = (0006H) = porta de leitura (RDVDP)
w = (0007H) = porta de escrita (WRVDP)
```

When communication with the VDP requires higher speed, these ports can be used to access the VDP directly. However, VDP is slow, requiring up to 8 µS interval between consecutive accesses. Therefore, it is good to avoid OTIR/OTDR type instructions, which can cause failures in the reading of data by the VDP. Prefer loops with OUTI/OTDR instructions. The ports are described below:

Port #0	(read)	r	Read data from VRAM (MSX1)
Port #0	(write)	w	Write data to VRAM (MSX1)
Port #1	(read)	r+1	Read status register (MSX1)
Port #1	(write)	w+1	Write in the control register (MSX1)
Port #2	(write)	w+2	Write in the palette registers (MSX2 or higher)
Port #3	(write)	w+3	Write in the indirectly specified register (MSX2 or higher)

Even though the VDP access ports have not been standardized, they are the same on all MSX models released. Are the following:

```
Port #0 - 98H
Port #1 - 99H
Port #2 - 9AH
Port #3 - 9BH
```

5.2 – ACCESS TO VRAM AND VDP (V9918/38/58)

The VDP and VRAM can be accessed directly through the I/O ports already described. This section describes how to do this.

5.2.1 – Access to control registers

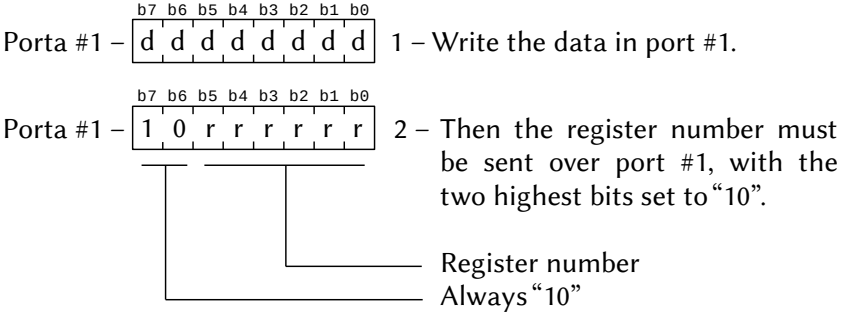
Control registers are write-only. However, the contents of the first subgroup (R#0 to R#27) can be obtained by BASIC's VDP(n) com-

mand because there is a copy of them in the system work area (Addresses F3DFH to F3E6H, FFE7H to FFF6H and FFFAH to FFFCH). This copy is done by the BIOS.

There are three ways to write data to the control registers, described below.

5.2.1.1 – Direct access

The first way is to specify the data and write it directly. The data is written first, to port #1, followed by the respective register number, as illustrated below:

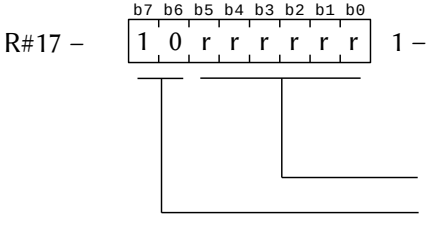


ASM code example:

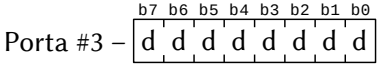
```
LD   A,x           ; x = data do be written
OUT  (099H),A      ; send data
LD   A,10000000B   ; sets b7-b6 in "10"
OR   y             ; y = register number (0~46)
OUT  (099H),A      ; send register
```

5.2.1.2 – Indirect access

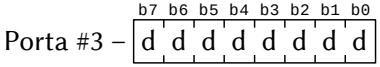
The second way is to write the data to the register specified by R#17. For this, it is necessary to use the direct method to place the desired register number in R#17, with the two highest bits set to "10". Then you can send data continuously through port #3 to the same register. This medium is useful for executing VDP hardware commands.



1 – The register number must be placed in R#17 with the highest two bits set to "10"



2 – The data sent through port #3 will be placed in the register specified in R#17.



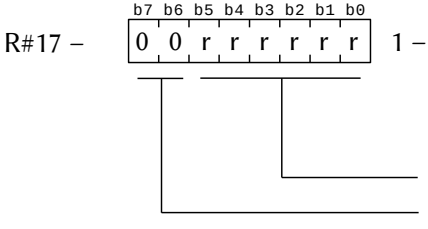
3 – Other data can be sent over port #3 to the same register indefinitely.

ASM code example:

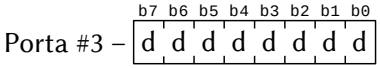
```
LD   A,10000000B  ; sets b7-b6 in "10"
OR   y            ; y = register number (0~46)
OUT  (099H),A     ; send register number
LD   A,10000000B  ; b7,b6 = 10 → no autoinc
OR   17           ; A = R#17 with b7,b6 = 10
OUT  (099H),A     ; send register to R#17
LD   A,x          ; x = data to be written
OUT  (09BH),A     ; send data
LD   A,x          ; x = data to be written
OUT  (09BH),A     ; send new data to the same
                  ; register, indefinitely
```

5.2.1.3 – Indirect access with autoincrement

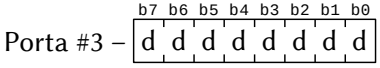
This method is similar to the previous one, with the difference that each time the data is written by port #3, R#17 is incremented by 1 and the next data sent will be written in the next register. To use it, it is necessary to put, by the direct method, the number of the first register in R#17 with the two highest bits set to "00". Then just send the data through port #3.



1 – The register number must be placed in R#17 with the highest two bits set to "00"



2 – The data sent through port #3 will be placed in the register specified by R#17.



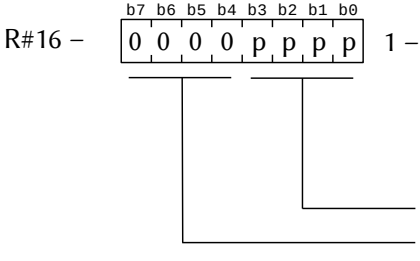
3 – The next data sent through port #3 will be placed in the next register, and so on.

ASM code example:

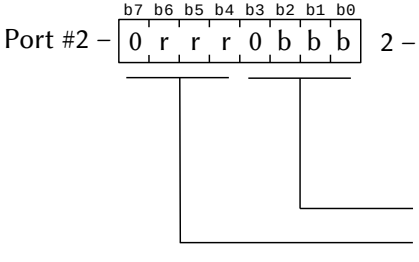
```
LD   A,10000000B  ; sets b7-b6 in "10"
OR   y            ; y = register number (0~46)
OUT  (099H),A     ; send register number
LD   A,17         ; A = R#17 with b7,b6 = 00
                  ; (Autoinc function active)
OUT  (099H),A     ; send the register to R#17
LD   A,x          ; x = data to be written
OUT  (09BH),A     ; send data to R#n
LD   A,x          ; x = data to be written
OUT  (09BH),A     ; send data to R#n+1
LD   A,x          ; x = data to be written
OUT  (09BH),A     ; send data to R#n+2
                  ; (and so on)
```

5.2.2 – Access to palette registers

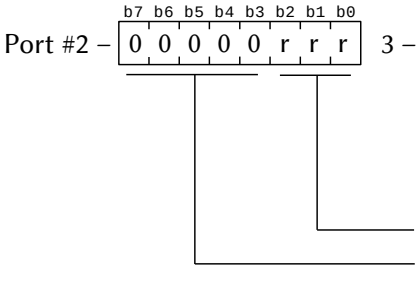
To write to the palette registers (P#0 to P#15), it is necessary to specify the palette number in the lower four bits of R#16 and send the data through port #2. As each register has 9 bits, the data must be sent by two consecutive bytes. After the two bytes are sent, R#16 is automatically incremented by 1, pointing to the next palette register. This feature makes it easier to initialize the palette.



1 – The palette register number must be placed in the lowest 4 bits of R#16, with the highest 4 bits in "0000".



2 – First the red and blue levels must be sent through port #2. The two unused bits must always be "0".



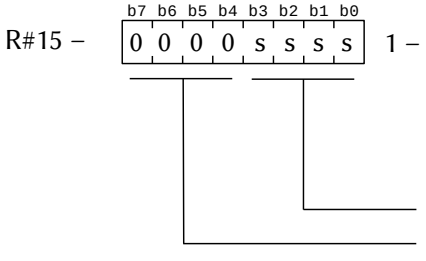
3 – Then the green level must also be sent through port #2. Once the green level is sent, R#16 will increment by 1 and point to the next palette register.

There are an ASM code example below.

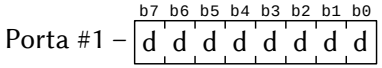
```
LD   A,x          ; x = palette reg num (0~15)
OUT  (099H),A     ; send register number
LD   A,16         ; R#16
OUT  (099H),A     ; send pal. register to R#16
LD   A,x          ; x = red/blue levels
OUT  (09AH),A     ; send red/blue to P#n
LD   A,x          ; x = green level
OUT  (09AH),A     ; send green to P#n
LD   A,x          ; x = red/blue levels
OUT  (09AH),A     ; send red/blue to P#n+1
LD   A,x          ; x = green level
OUT  (09AH),A     ; send green to P#n+1
                  ; (and so on)
```

5.2.3 – Reading the status registers

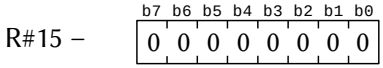
Status registers are read-only. Their contents can be read through port #1, putting in R#15 the number of the status register to be read. Interrupts must be disabled (DI) while reading status registers. Once read, register R#15 must be cleared before interrupts are re-enabled.



1 – The status register number must be placed in the lowest 4 bits of R#15. Interrupts must be disabled.



2 – The data is available to be read by port #1



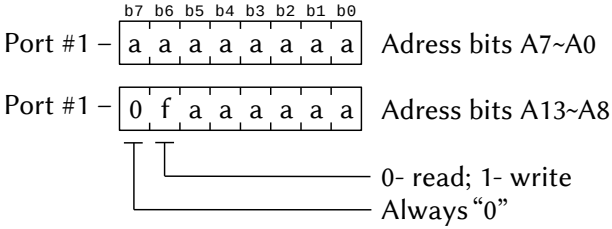
3 – R#15 must be set to zero before re-enabling interrupts.

There are an ASM code example below.

```
STA:  DEFB 00H

      DI                ; disable interrupt
      LD   A,x          ; x = status reg number(0~9)
      OUT  (099H),A     ; send register number
      LD   A,15         ; R#15
      OUT  (099H),A     ; send status reg to R#15
      IN   A,(099H)     ; A get status reg value
      LD   (STA),A      ; save value in STA
      XOR  A            ; A = 0
      OUT  (099H),A     ; send 0 value
      LD   A,15         ; R#15
      OUT  (099H),A     ; R#15 = 0
      EI                ; enable interrupt
```

5.2.4 – Acess to VRAM by CPU

To access the 16 Kbytes of the TMS9918A on the MSX1, a 14-bit adress bus is used. In order for the CPU to access the VRAM, these 14 bits must be sent out port #1 in two consecutive bytes. Bit 6 of the second byte is a flag to indicate read or write, as illustrated below:

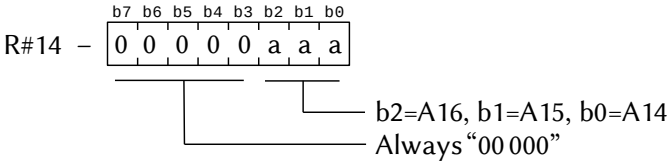


After setting the VRAM address, the data can be read or written through port #0. The read/write flag must be set as already described. The addresses counter is automatically incremented by 1 each time a byte is read or written through port #0, in order to facilitate continuous access to the VRAM.

ASM code example:

```
LD   HL,x          ; X = 0000H ~ 3FFFH
DI                 ; disable interrupt
LD   A,L           ; A = LSB 8 bits
OUT  (099H),A      ; send LSB 8 bits
LD   A,H           ; A = MSB 6 bits
OR   64            ; OR 01000000 sets write
OUT  (099H),A      ; send MSB 6 bits + flag
DI                 ; enable interrupt
LD  A,x            ; X = data to be sent
OUT  (098H),A      ; send data to VRAM
```

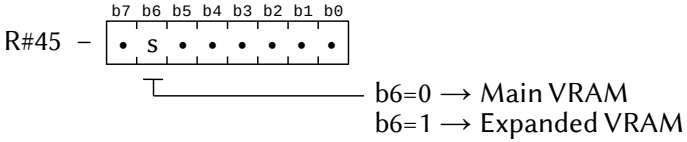
To access the V9938/V9958's 128 Kbytes of VRAM, a 17-bit address bus is used. Of these, the highest three bits are stored in R#14. In this way it is possible to select up to 8 pages of 16 Kbytes each.



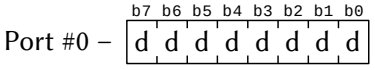
ASM code example:

```
LD   A,x          ; x = MSB 3 bits (pages)
OUT  (099H),A     ; send the page
LD   A,14         ; R#14
OUT  (099H),A     ; send page to R#14
LD   HL,x         ; X = 0000H ~ 3FFFH
DI                ; disable interrupt
LD   A,L          ; A = LSB 8 bits
OUT  (099H),A     ; send LSB 8 bits
LD   A,H          ; A = MSB 6 bits
OR   64           ; OR 01000000 sets write
OUT  (099H),A     ; send MSB 6 bits + flag
DI                ; enable interrupt
LD  A,x           ; X = data to be sent
OUT  (098H),A     ; send data to VRAM
```

The first 64 Kbytes of VRAM and the expanded VRAM occupy the same address space as the VDP. If the MSX does not have expanded VRAM, the main VRAM will always be selected. Bit 6 of R#45 can be used to switch between the two 64 Kbyte banks, as shown in the illustration below:



After setting the VRAM address and whether it is expanded or not, the data can be read or written through port #0. The read/write flag must be set as already described. The address counter is incremented by 1 each time a byte is read or written through port #0. This feature facilitates seamless access to the VRAM.



The data read or write is made through port #0.

Access times between reads/writes to VRAM depend on screen and VDP. Minimum times are listed below. The value followed by T means the number of T cycles of a Z80 working at the standard MSX clock (3.58 MHz).

		TMS9918/9928	V9938/V9958
Screen 0, width 40	Text 1	3,4 µS (12T)	5,6 µS (20T)
Screen 0, width 80	Text 2		5,6 µS (20T)
Screen 1	Graphic 1	8,2 µS (29T)	4,2 µS (15T)
Screen 2	Graphic 2	8,2 µS (29T)	4,2 µS (15T)
Screen 3	Multicor	3,7 µS (13T)	4,2 µS (15T)
Screen 4	Graphic 3		4,2 µS (15T)
Screen 5	Graphic 4		4,2 µS (15T)
Screen 6	Graphic 5		4,2 µS (15T)
Screen 7	Graphic 6		4,2 µS (15T)
Screen 8	Graphic 7		4,2 µS (15T)
Screen 10	Graphic 8		4,2 µS (15T)
Screen 11	Graphic 8		4,2 µS (15T)
Screen 12	Graphic 9		4,2 µS (15T)

Note that, in the case of the MSX turbo R, the access to the VDP is timed in 8 µS between consecutive accesses to the registers.

5.3 – SCREEN MODES (V9918/38/58)

The MSX1 has 4 screen modes. The MSX2 has six more modes and the MSX2+ and MSX turbo R, in addition to these, there are two more. In the table below the modes marked with "*" were added for MSX2 and those marked with "**" were added for MSX2+ and MSX turbo R. Along with the modes, there is a short description of them.

MODE	SCREEN	SHORT DESCRIPTION
TEXT 1	SCREEN 0 WIDTH 40	Up to 40 characters per line of text; one color for all characters
TEXT 2 *	SCREEN 0 WIDTH 80	Up to 80 characters per line of text; Blink function included
MULTICOLOR	SCREEN 3	Pseudographic; 64 x 48 points; 16 independent colors for each point

GRAPHIC 1	SCREEN 1	32 characters per line of text; multi-color characters available
GRAPHIC 2	SCREEN 2	256 x 192 points; 16 colors limited to 2 every 8 horizontal points; sprites mode 1
GRAPHIC 3*	SCREEN 4	Same as Graph 2, but uses mode 2 sprites
GRAPHIC 4*	SCREEN 5	256 x 212 points; 16 colors out of 512 for each point; mode 2 sprites
GRAPHIC 5*	SCREEN 6	512 x 212 points; 4 colors of 512 for each point; mode 2 sprites
GRAPHIC 6*	SCREEN 7	512 x 212 points; 16 colors out of 512 for each point; mode 2 sprites
GRAPHIC 7*	SCREEN 8	256 x 212 points; 256 colors for each point; mode 2 sprites
GRAPHIC 8**	SCREEN 10 SCREEN 11	256 x 212 points; 65 536 colors for every 4 horizontal points or 16 colors out of 512 for each point; maximum of 12 499 simultaneous colors; mode 2 sprites
GRAPHIC 9**	SCREEN 12	256 x 212 points; 131 072 colors for every 4 horizontal points; maximum of 19 268 simultaneous colors; mode 2 sprites

There are also some special settings that use some of the graphics modes for text. They are as follows:

Screen 9	Width 1~40	Graphic 4	Korean MSX2
	Width 41~80	Graphic 5	
Kanji0	Width 1~32	Graphic 4	MSX2+ MSX turbo R
	Width 33~64	Graphic 6	
Kanji1	Width 1~40	Graphic 4	
	Width 41~80	Graphic 6	
Kanji2	Width 1~32	Graphic 4 interlaced	
	Width 33~64	Graphic 6 interlaced	

Kanji3	Width 1~40	Graphic 4 interlaced	
	Width 41~80	Graphic 6 interlaced	

5.3.1 – Text mode 1

- 24 lines of up to 40 characters each;
- A background color and a character color, selected from 16 (MSX1) or 512 (MSX2 or higher);
- 256 characters available with a resolution of 6 horizontal by 8 vertical points;
- Requires 2048 bytes for the font and 960 bytes for the screen;
- Compatible with Screen 0, Width 1~40.

Text mode 1 is selected by registers R#0 and R#1, as illustrated below:

	b7	b6	b5	b4	b3	b2	b1	b0
R#0 –	•	•	•	•	0	0	0	•
R#1 –	•	•	•	1	0	•	•	•

The area where the character font is stored is called the pattern generator table. In it, each character is defined by 8 bytes, but the lowest two bits of each byte are not displayed. Therefore, the cell where each character is shown has 6 x 8 points. The font contains 256 distinct characters, numbered from 0 to 255.

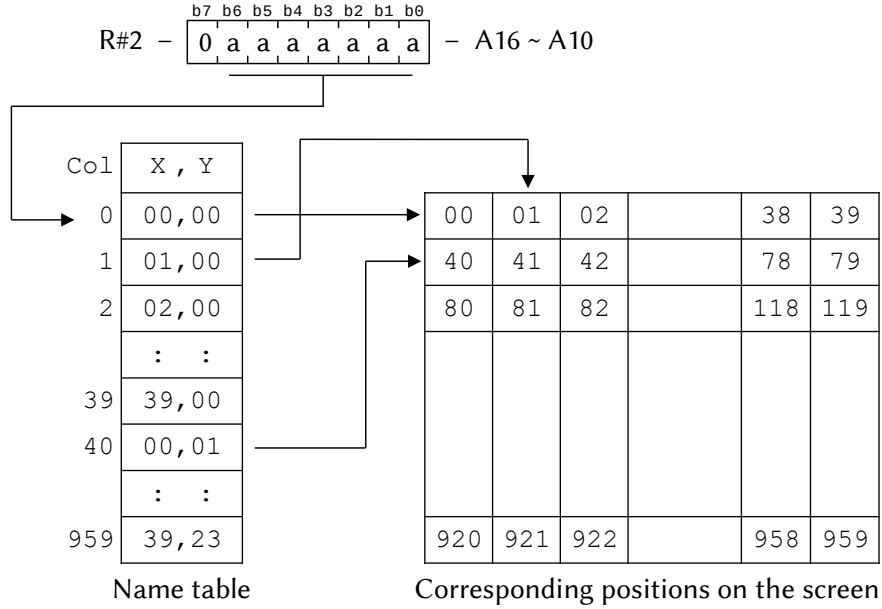
The location of the pattern generator table is specified in R#4. The lowest 11 bits (b10 ~ b0) are not specified; are always 0. Only the highest 6 bits can be specified. Therefore, the table always starts at a multiple of 2 Kbytes from 00000H. This address can be obtained from BASIC's BASE(2) system variable.

	b7	b6	b5	b4	b3	b2	b1	b0	
R#4 –	0	0	a	a	a	a	a	a	– A16 ~ A11

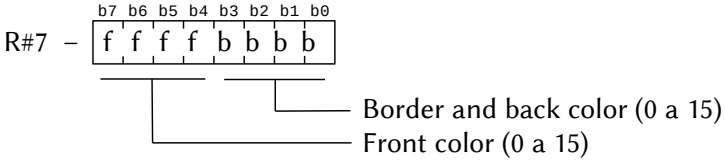
The pattern name table stores the position of each character that should be displayed on the screen. One byte is used for each cha-

racter; it contains the ASCII code of the character to be displayed in the respective position.

The starting address of the table is specified in R#2. The lowest 10 bits (b9 ~ b0) are not specified; are always 0. Only the highest 7 bits are specified. Therefore, the name table always starts at a multiple of 1 Kbyte from 00 000H. This address can be obtained from BASIC's BASE(0) system variable.



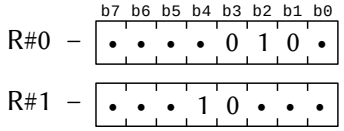
To specify character color and background color, register R#7 is used. The highest four bits of R#7 specify the character color (foreground color) and the lowest four bits specify the background and border color.



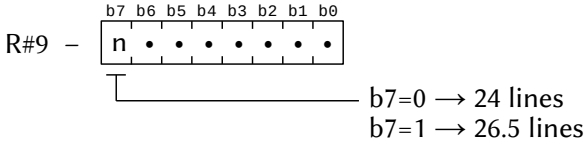
5.3.2 – Text mode 2

- 24 or 26.5 lines of up to 80 characters each;
- One background color and one color for characters, selected from 512;
- 256 characters available with a resolution of 6 horizontal by 8 vertical points;
- Independent blink function for each position on the screen;
- Requires 2048 bytes for the font;
- For 24 lines it requires 1920 bytes for the screen (80 characters x 24 lines) and 240 bytes (1920 bits) for the blinking attributes;
- For 26.5 lines it requires 2160 bytes for the screen (80 characters x 27 lines) and 270 bytes (2160 bits) for the blinking attributes;
- Compatible with Screen 0, Width 41~80.

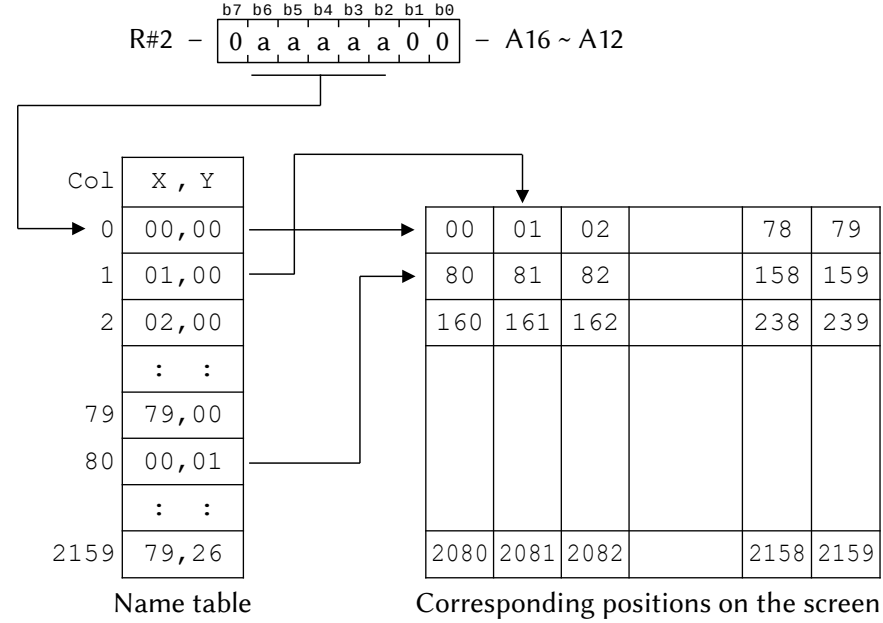
Text mode 2 is selected by registers R#0 and R#1, as illustrated below:



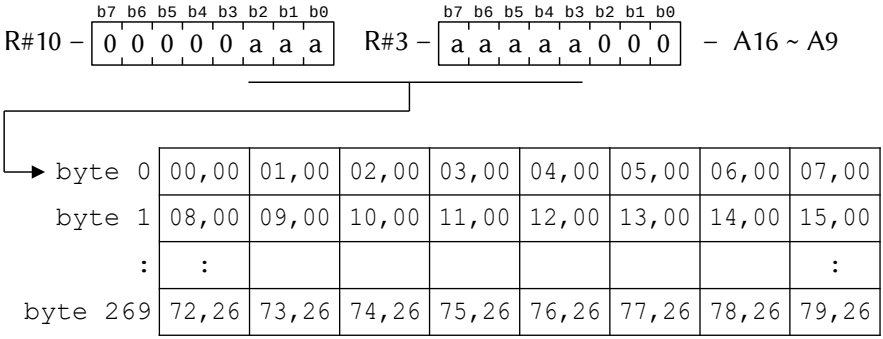
Text mode 2 can display 24 or 26.5 lines, depending on the value of bit 7 of R#9. On the last line of 26.5 lines mode, only the upper half of the characters are displayed. This mode is not supported by BASIC.



The pattern generator table of text mode 2 has the same structure, function and size as that of text mode 1. As the number of characters that can be displayed in this mode has been increased to a maximum of 2160 (80 x 27), memory maximum occupied by the name table is 2160 bytes. The starting address of the name table must be specified in R#2. Only the highest 5 bits are specified; the lowest 12 bits are always 0. Therefore, the starting adress of the name table is always a multiple of 4 Kbytes from 00 000H.

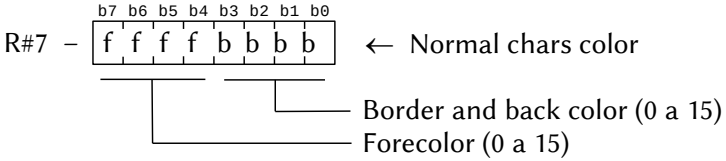


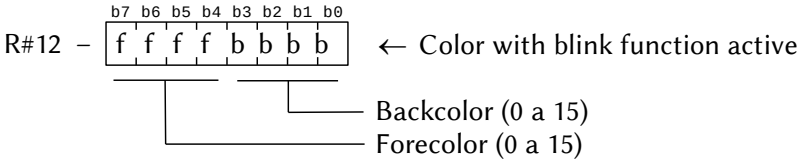
In text mode 2 it is possible to make characters blink. The blinking table stores the position of each character on the screen; one bit in the table corresponds to one character. When this bit is 1, the blink function is activated for the respective character. The adress of the table is stored in R#3 and R#10. The highest 8 bits specify the adress and the lowest 9 bits are always 0. Therefore, the Adress of the blinking table is always a multiple of 512 bytes from 00 000H.



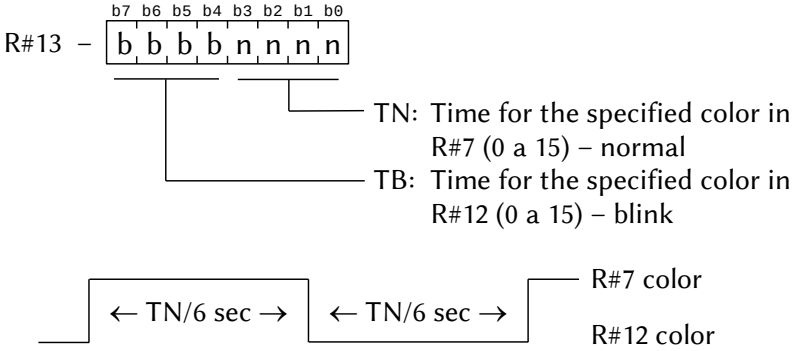
Blinking table with corresponding positions on the screen

Character colors in text mode 2 are specified in R#7 of R#12. The highest four bits of R#7 specify the character color (foreground color) and the lowest four bits specify the background and border color. When the Blink function is active, the character color will be specified by the highest four bits of R#12 and the character background color by the lowest four bits of R#12.





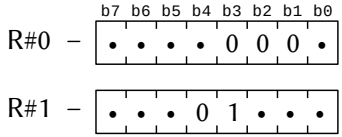
The blinking time (time the character assumes the blink colors and then returns to normal colors) is specified in R#13. The highest four bits of R#13 define the time the character is in the original color and the lowest four bits define the time the characters are in the blink color. The time period is specified in 1/6 second units. When the register is zeroed, the colors are permanently assumed. The illustration below shows how the time is divided for the blink function.



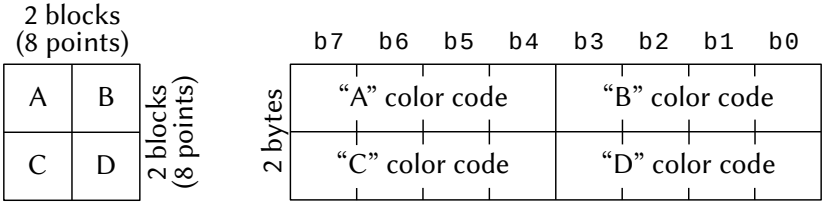
5.3.3 – Multicolor mode

- 64 (horizontal) x 48 (vertical) blocks;
- Up to 16 simultaneous colors;
- Each block has 4x4 points and a color;
- Requires 2048 bytes for the color table and 768 for specifying the location of blocks on the screen;
- Mode 1 sprites;
- Screen 3 compatible.

Multicolor mode is selected by R#0 and R#1 as illustrated below:

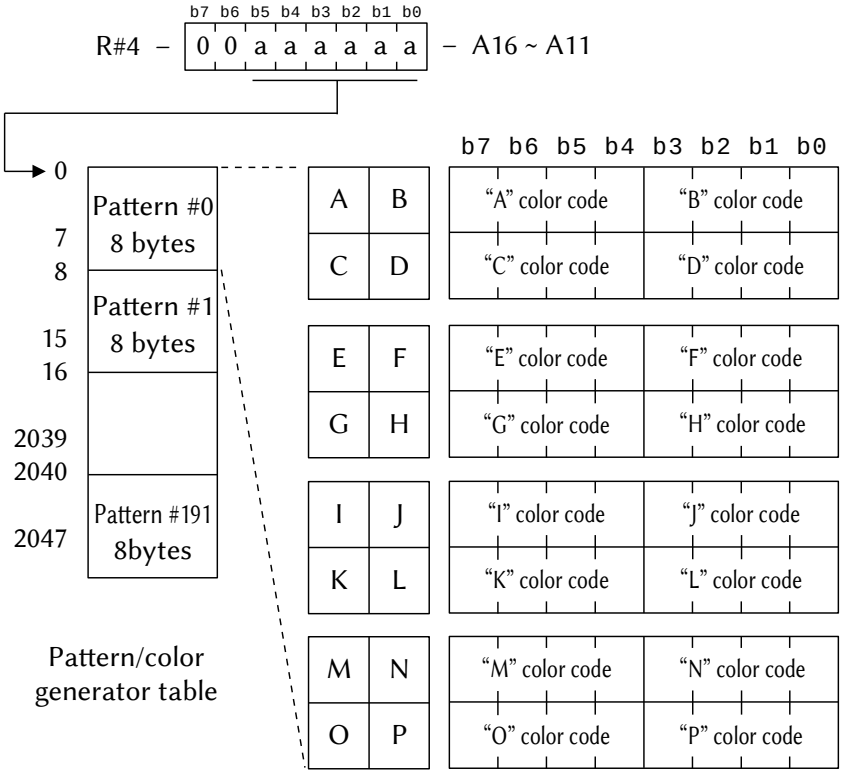


Organizing this way is a bit complex. Each pattern (character) corresponds to 4 blocks, in a 2x2 construction. Two bytes in the pattern table represent the color of each block. The organization of each pattern is illustrated below.



The color table address is specified in R#4. Only the highest 6 bits are specified; so the initial address of the table will always be a multiple of 2 Kbytes from 00 000H.

Each 8 bytes of the pattern table corresponds to 1 character wide (2 horizontal blocks) and scans the screen vertically in an interleaved form, as shown in the illustration below.



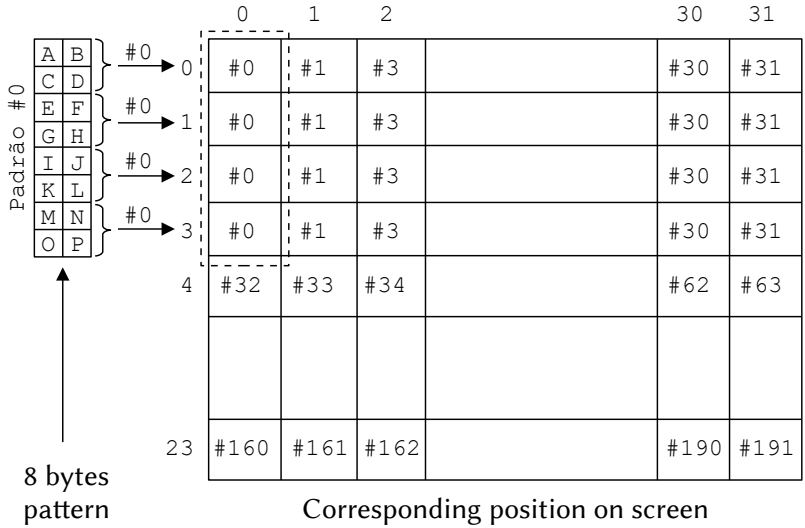
ABCD table: valid when Y is 0, 4, 8, 12, 16 or 20

EFGH table: valid when Y is 1, 5, 9, 13, 17 or 21

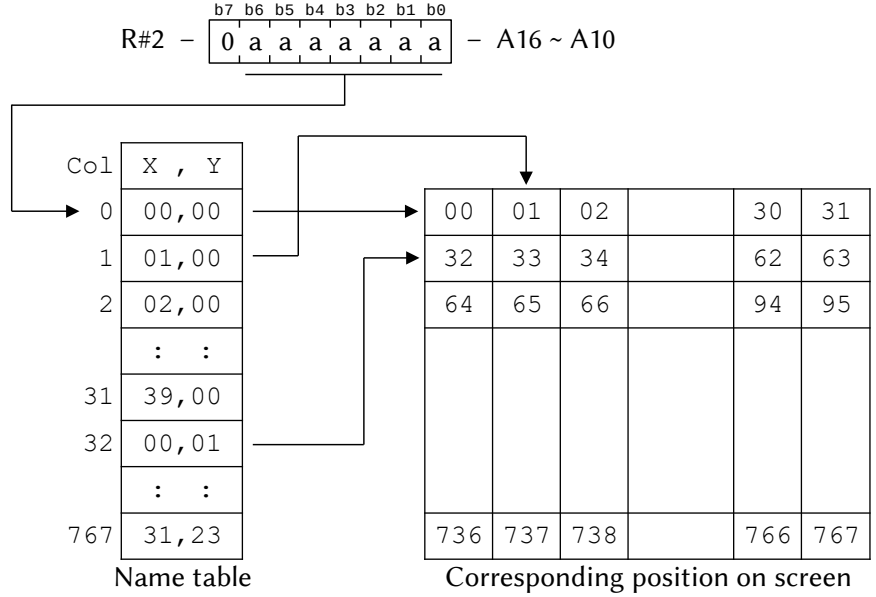
I J K L table: valid when Y is 2, 6, 10, 14, 18 or 22

MNOP table: valid when Y is 3, 7, 11, 15, 19 or 23

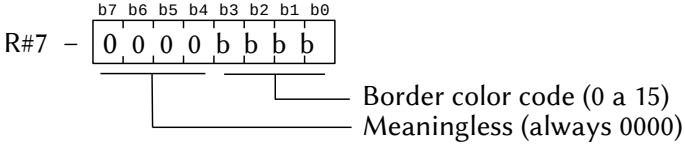
The name table specifies the screen coordinates where characters (2 x 2 blocks) will be displayed. It is also arranged 4 by 4 in an interleaved form, as illustrated below.



The initial address of the pattern name table is specified in R#2. Only the highest 7 bits are specified; therefore, the name table always starts at a multiple of 1 Kbyte from 00 000H.



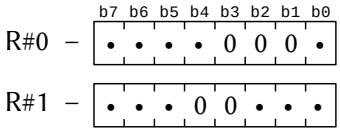
The border color in multicolor mode must be specified in the lowest 4 bits of R#7.



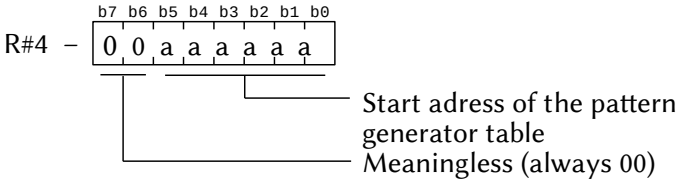
5.3.4 – Graphic mode 1

- 32 (horizontal) x 24 (vertical) patterns;
- Up to 16 colors can be displayed simultaneously (chosen from 512 for MSX2 or higher);
- Each pattern is 8 x 8 points and can be set freely;
- Different colors for each 8 patterns can be set;
- Requires 2048 bytes for the patterns source, 768 bytes for the table of names and 32 bytes for the color table;
- Mode 1 sprites;
- Screen 1 compatible.

Graphics mode 1 is selected by R#0 and R#1 as illustrated below.

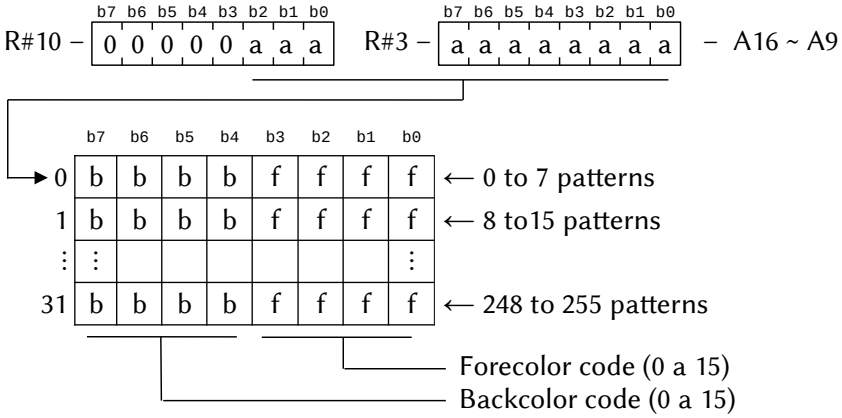


In this mode, 256 types of patterns, or characters, can be displayed on the screen. The source of each pattern is defined by the pattern generator table. The starting address of the pattern table is specified in R#4. Only the highest 6 bits are specified; so this table always starts at a multiple of 2 Kbytes from 00 000H.

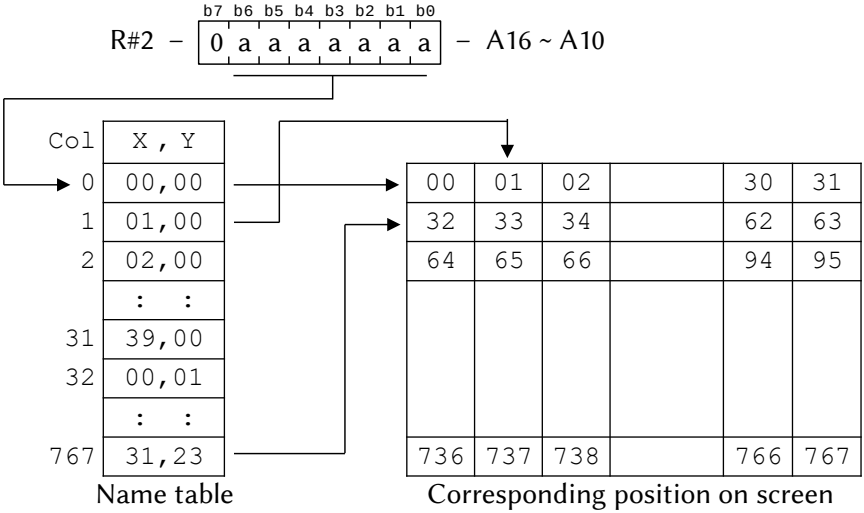


The color table specifies one color for every 8 consecutive patterns in the pattern table. The starting address of the color table is speci-

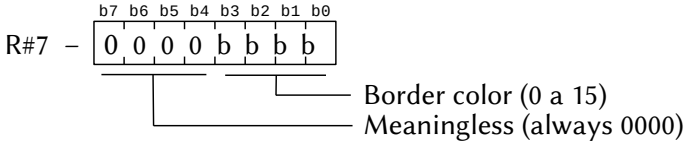
fied in R#3 and R#10. Only the highest 11 bits are specified (A16 to A6); so this table always starts at a multiple of 64 bytes from 0000H.



The pattern name table has 768 bytes and is responsible for locating them on the screen. The starting adress of this table is specified in R#2. Only the highest 7 bits (A16 to A10) are specified; bits A9 to A0 are always 0. Therefore, the pattern name table always starts at a multiple of 1 Kbyte from 0000H. The organization of this table is illustrated below.



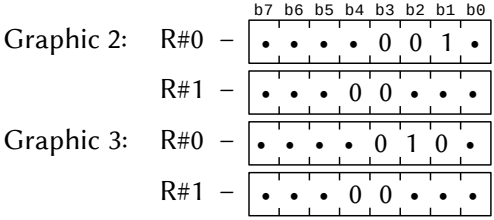
The border color in graphics mode 1 must be specified in the lowest 4 bits of R#7.



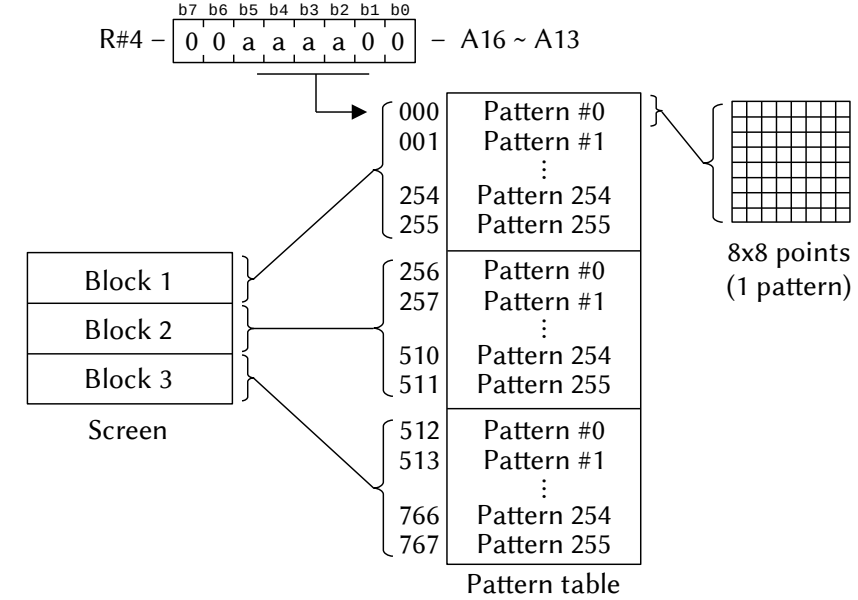
5.3.5 – Graphic Modes 2 and 3

- 32 (horizontal) by 24 (vertical) patterns;
- Up to 16 colors can be displayed simultaneously;
- 768 different patterns are available;
- Each pattern has 8 x 8 pointes;
- Any figure can be defined for each pattern;
- Only 2 colors can be defined for every 8 horizontal points;
- Requires 6144 bytes for the patterns source and another 6144 bytes for the color table;
- Sprites mode 1 for graph 2 and mode 2 for graph 3;
- Graphic 2 compatible with Screen 2 and graphic 3 with Screen 4.

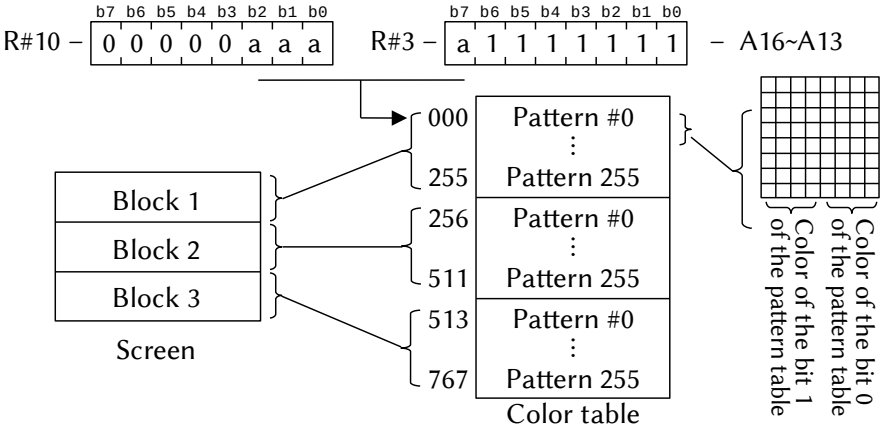
Graphic modes 2 and 3 are selected by R#0 and R#1 as illustrated below. They are identical except that Graphics Mode 2 uses Mode 1 sprites and Graphics 3 uses Mode 2 sprites.



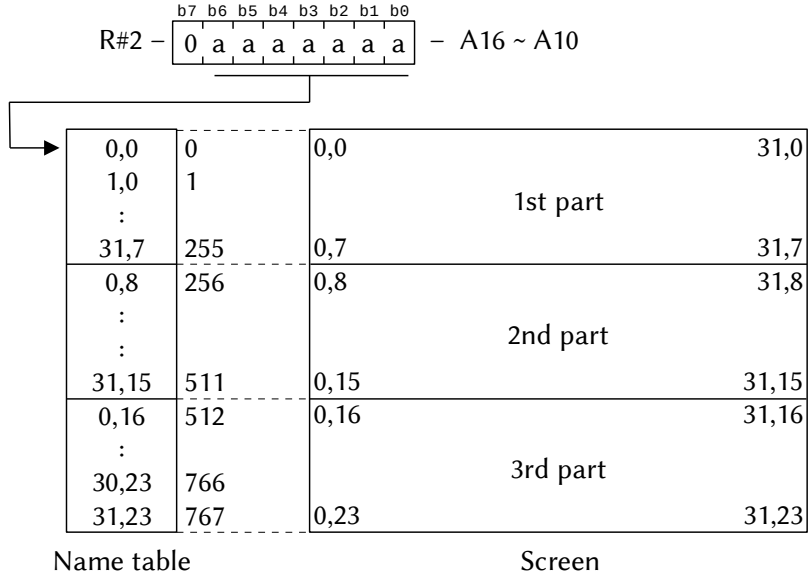
In these two modes, the pattern generator table is compatible with graphic mode 1, where 768 different patterns can be displayed. As each pattern is 8 x 8 points and can have a different design, there is a 256 x 192 point display simulation on the screen. The starting address of the pattern generator table is specified in R#4. Only the highest 4 bits of adress are valid (A16 to A13); so the initial adress will always be a multiple of 8 Kbytes from 00 000H. In this mode, the screen is divided into three blocks of 256 patterns each, making a total of 768 patterns.



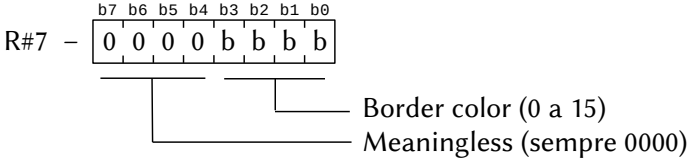
The size of the color table is the same as that of the pattern generator table, and colors can be specified for each bit 0 or 1 of each horizontal row of each pattern. The starting address of the color table is specified in R#3 and R#10, but only the highest four bits are specified; therefore, the color table always starts at a multiple of 8 Kbytes from 00 000H.



The pattern name table is divided into three parts, one for each screen block. Each part is 256 bytes long and is responsible for displaying 256 patterns on the screen. The starting address of the name table is specified in R#2.



The border color in graphics modes 2 and 3 is specified in the lowest 4 bits of R#7.

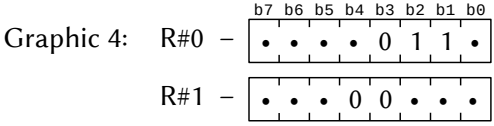


5.3.6 – Graphic mode 4

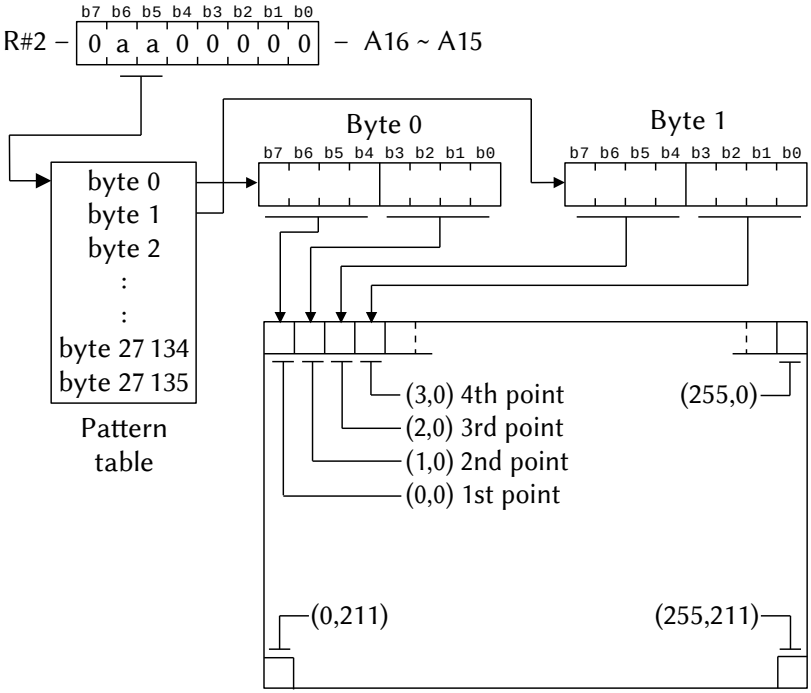
- 256 (norizontal) by 212 (vertical) points;
- Displays up to 16 colors chosen from 512 for each point;
- High speed hardware commands are available;
- Mode 2 sprites;
- Requires 26.5 Kbytes (4 bits x 256 points x 212 points) of memory;

- Easy-to-handle bit-mapped graphics;
- Screen 5 compatible.

Graphics mode 4 is selected by R#0 and R#1, as illustrated below:



In graphic mode 4, one byte in the pattern generator table corresponds to two points on the screen. Each point is represented by 4 bits; therefore up to 16 colors can be specified for each point. The starting address of the pattern generator table is specified in R#2. Only the highest two bits are specified; so the pattern generator table always starts at a multiple of 32 Kbytes from 00 000H.



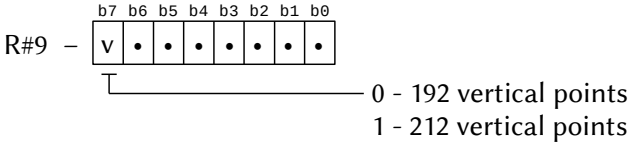
Location of points on the screen

In this mode, each byte corresponds to two horizontal points in sequence on the screen and as the horizontal resolution is 256 points, 128 bytes are required for each screen line. In this mode there is no need for the name table. The adress in the VRAM of each point can be calculated by the following expression:

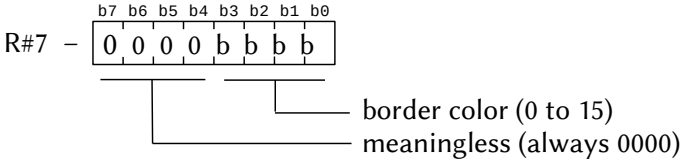
```
Address = X/2 + Y*128 + Initial address
```

Where X is the horizontal coordinate and Y is the vertical coordinate of the point. The point is specified by the highest 4 bits of adress if X is even and the lowest 4 bits of X is odd.

In this mode, the number of vertical points can be increased to 212. To do so, just set bit 7 of R#9, as illustrated below.



The border color in graphics mode 4 must be specified in the lowest 4 bits of R#7, as illustrated below.

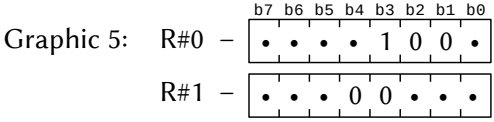


5.3.7 – Graphic mode 5

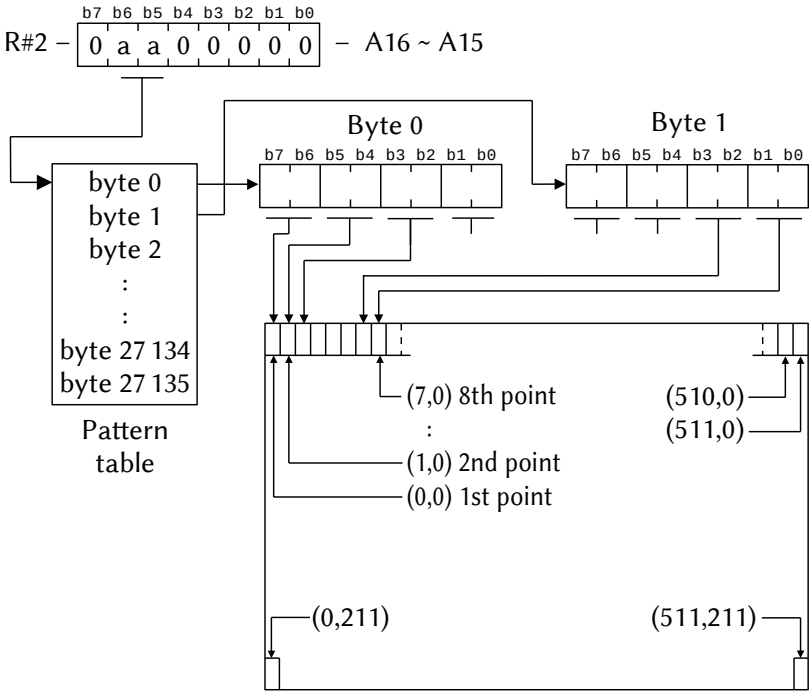
- 512 (norizontal) by 212 (vertical) points;
- Displays up to 4 colors chosen from 512 for each point;
- High speed hardware commands are available;
- Mode 2 sprites;
- Requires 26.5 Kbytes (2 bits x 512 points x 212 points) of memory;
- Easy-to-handle bit-mapped graphics;
- Screen 6 compatible.

Graphic mode 5 is selected by R#0 and R#1, as illustrated below.

Graphic 5:



In graphic mode 5 one byte in the pattern table corresponds to four points on the screen. Each point is represented by 2 bits and can have up to 4 colors. The starting address of the pattern generator table is specified in R#2, with only the highest two bits being valid. Therefore the table always starts at a multiple of 32 Kbytes from 00 000H.



Location of points on the screen

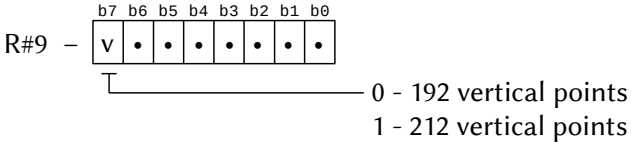
Each byte represents four points in a horizontal sequence on the screen and as there are 512 points of horizontal resolution, 128 bytes are needed to represent each line on the screen. The adress in the VRAM of each point can be calculated by the following expression:

```
Address = X/4 + Y*128 + Initial address
```

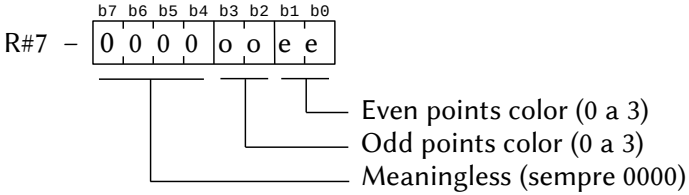
Where X is the horizontal coordinate and Y the vertical coordinate. Since each byte represents four points, one more operation is needed to determine which pair of bits represents each point:

If X mod 4 = 0, the point is represented by bits 7 and 6;
If X mod 4 = 1, the point is represented by bits 5 and 4;
If X mod 4 = 2, the point is represented by bits 3 and 2;
If X mod 4 = 3, the point is represented by bits 1 and 0.

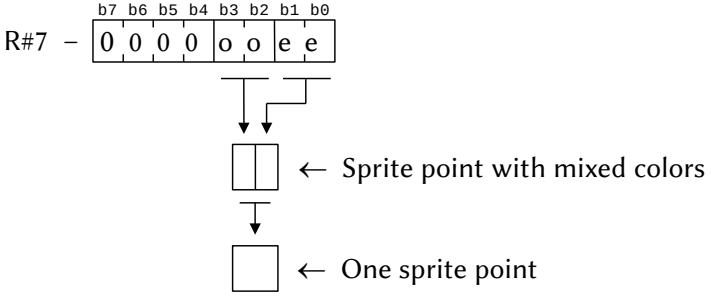
In this mode, the number of vertical points can be increased to 212. To do so, just set bit 7 of R#9, as illustrated below.



In graphics mode 5, there is a special treatment for the color of the border and sprites. Color is specified by 4 bits in R#7, two for even points and two for odd points.



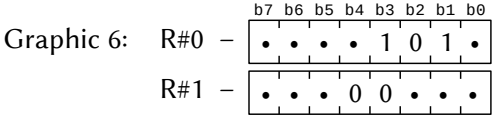
For the sprites, the colors are specified as illustrated below, where the codes for the even and odd points are inverted with respect to the border color:



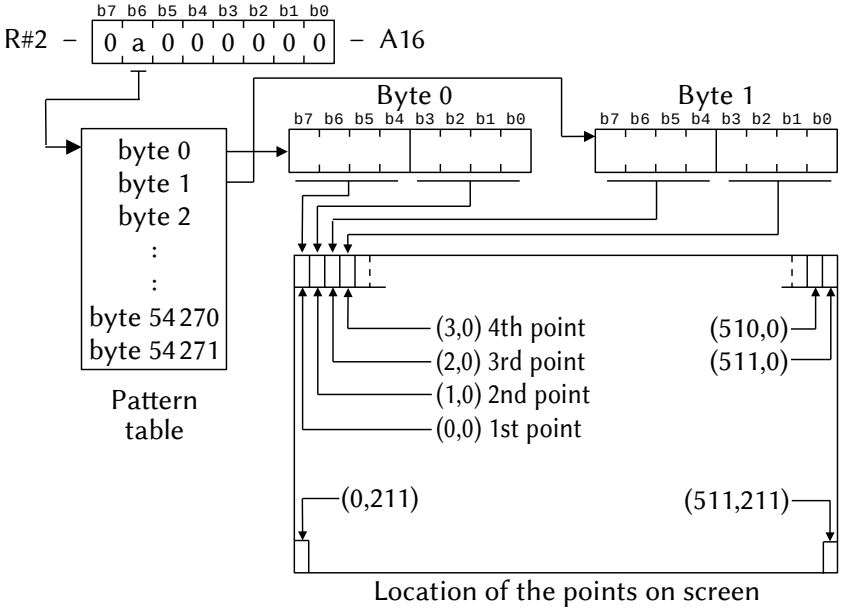
5.3.8 – Graphic mode 6

- 512 (horizontal) by 212 (vertical) points;
- Displays up to 16 colors chosen from 512 for each point;
- High speed hardware commands are available;
- Mode 2 sprites;
- Requires 53 Kbytes (4 bits x 512 points x 212 points) of memory;
- Easy-to-handle bit-mapped graphics;
- Screen 7 compatible.

Graphics mode 6 is selected by R#0 and R#1 as illustrated below.



In graphics mode 6, one byte in the pattern table corresponds to two points on the screen. Each point is represented by 4 bits; therefore 16 colors can be specified. The starting adress of the pattern table is specified by a single bit in R#2; therefore, the pattern generator table always starts at 00 000H or 10 000H.

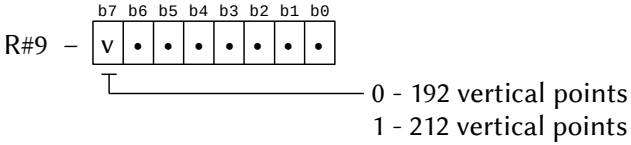


Since there are 512 points in each horizontal line and each byte represents two points, 256 bytes are required for each screen line. The adress of each point in the table can be calculated by the following expression:

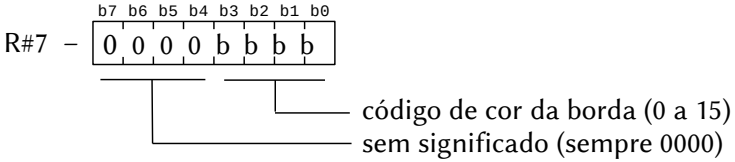
```
Address = X/2 + Y*256 + Initial address
```

Where X is the horizontal coordinate of the point and Y is the vertical coordinate. If the point is even, it will be represented by the highest 4 bits of the adress and if it is odd, it will be represented by the lowest 4 bits.

In this mode, the number of vertical points can be increased to 212. To do so, just set bit 7 of R#9, as illustrated below.



The border color in graphics mode 4 must be specified in the lowest 4 bits of R#7, as illustrated below.



5.3.9 – Graphic mode 7

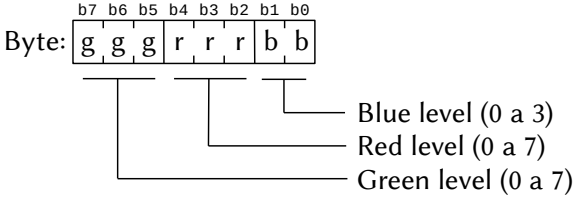
- 256 (norizontal) by 212 (vertical) points;
- Displays up to 256 simultaneous colors for each point;
- High speed hardware commands are available;
- Mode 2 sprites;
- Requires 53 Kbytes (8 bits x 256 points x 212 points) of memory;
- Easy-to-handle bit-mapped graphics;
- Screen 8 compatible.

Graphic mode 7 is selected by R#0 and R#1, as shown below:

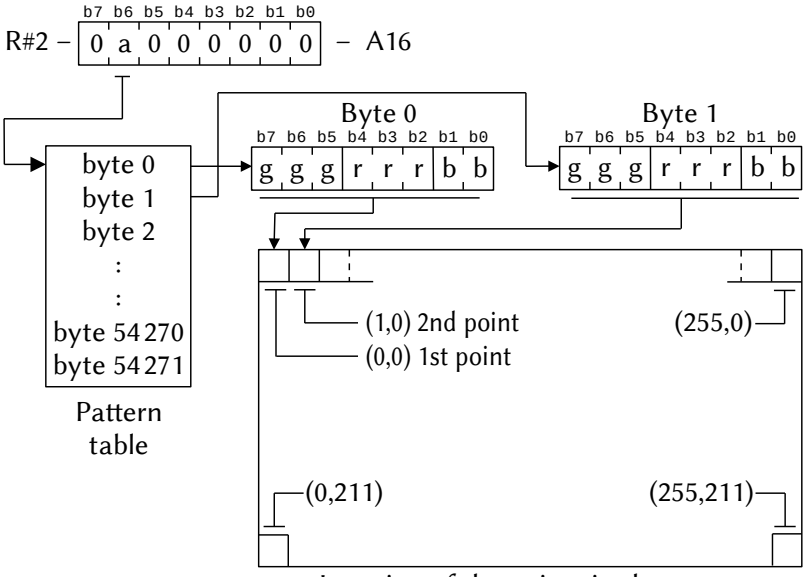
Graphic 7:

	b7	b6	b5	b4	b3	b2	b1	b0
R#0 –	•	•	•	•	1	1	1	•
R#1 –	•	•	•	0	0	•	•	•

The graphics mode 7 setting is the simplest of all; a point on the screen corresponds to a byte in the pattern table, and can display up to 256 simultaneous colors. In this mode, the color palette is not used, and each data byte reserves 3 intensity bits for green, 3 bits for red and 2 bits for blue.



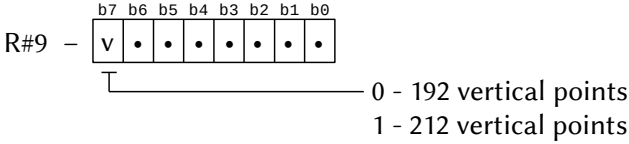
The starting address of the pattern table is specified in R#2 by a single bit; therefore, the pattern table always starts at 00 000H or 10 000H.



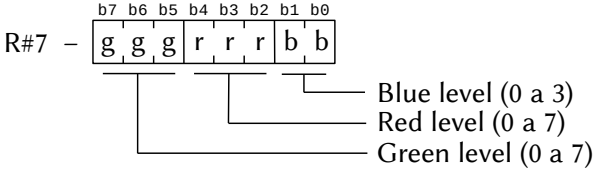
The address of each point on the screen can be calculated by the following expression:

```
Address = X + Y*256 + Initial address
```

Where X is the horizontal coordinate and Y is the vertical coordinate. In this mode, the number of vertical points can be increased to 212. To do so, just set bit 7 of R#9, as illustrated below.



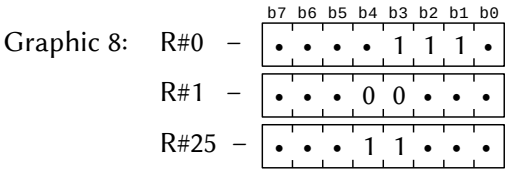
The border color must be specified in R#7 in the same format as the screen data bytes. All bits of R#7 are valid.



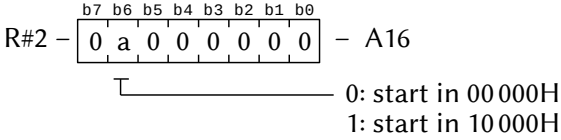
5.3.10 – Graphic mode 8

- 256 (norizontal) by 212 (vertical) points;
- displays up to 12 499 simultaneous colors on the screen;
- colors are specified for every four horizontal points;
- high speed hardware commands are available;
- mode 2 sprites;
- requires 53 Kbytes (256 points x 212 points) of memory;
- mixed YJK and bit-mapped graphic mapping;
- compatible with Screens 10 and 11.

Graphic mode 8 is selected by R#0, R#1 and R#25, as illustrated below.



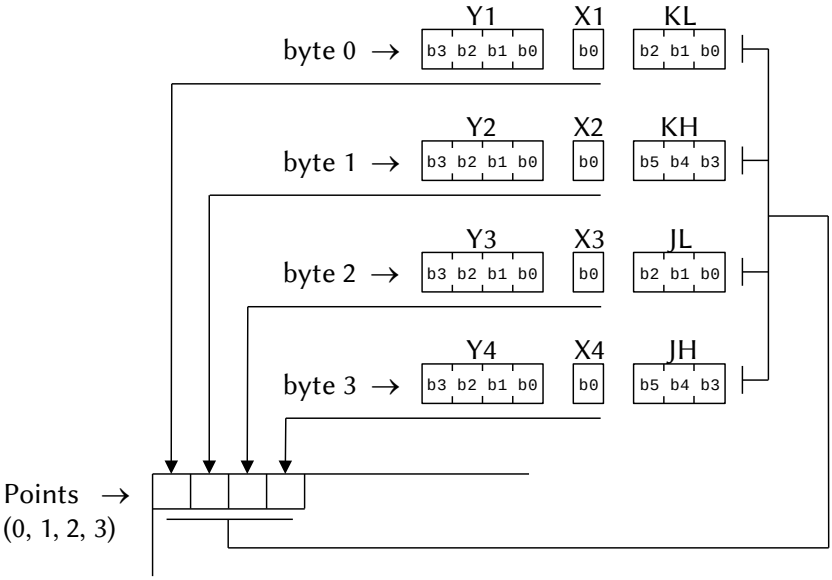
The start address of the pattern generator table is specified by a single bit of R#2; so it can start at 00 000H or 10 000H.



Configuring graphics mode 8 points and colors is a bit complex. In the modes already seen, the colors are dosed by the RGB system. In this new graphics mode, the color system used is YJK mixed with RGB.

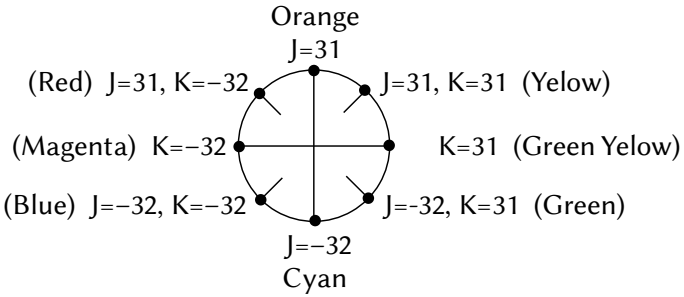
In this mode, the points are arranged in fours horizontally. Each group of 4 points can have a single color, chosen from 4096, with up to 16 levels of brightness for each individual point. Or, each point can have up to 16 colors chosen from a palette of 512, just like graphic mode 4.

It should be noted that the colors of each group of 4 horizontal points are not completely independent when using the YJK system. This can cause small screen blurs; however the method is excellent for representing photographed images. The structure of graphical mode 8 is illustrated below.



When the Xn bits are 1, the color for each point will be chosen from the 512 palette, with the respective 4 Yn bits ranging from 0 to 15, just as the colors are chosen for graphics mode 4. In this case, the J and K are ignored. It is not mandatory that all X bits of a group of 4 points are the same, and there may be mixing in the 4 bytes that make up the group.

When the Xn bits are 0, the YJK system will be used. In this system, the colors are chosen by the vectors J and K, with J being represented by 6 bits and K by another 6, according to the above scheme. The graph below illustrates how the colors are represented by these vectors.



The binaries corresponding to the values shown in the illustration are as follows:

	b5	b4	b3	b2	b1	b0
0 →	0	0	0	0	0	0
31 →	0	1	1	1	1	1
−32 →	1	0	0	0	0	0
−1 →	1	1	1	1	1	1

Since there are 12 bits to represent the color (6 bits for J and 6 bits for K), we make 2^12 = 4096 colors, which is the maximum number of colors that can be defined. Each group of 4 horizontal points can only have one color chosen from these 4096. However, each individual point of this group can have a brightness variation of 16 levels, represented by the Yn bits, where 0000B represents minimum brightness and 1111B maximum brightness.

The J and K vectors can vary from –32 to 31, as illustrated. By combining the extreme values, the four main colors of the YJK system can be formed: green, red, blue and yellow. Using the intermediate values, 4096 colors can be generated. Conversion from the YJK system to RGB and vice versa can be done using the following formulas:

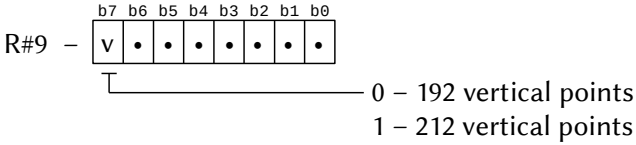
```
Y = R/4 + G/8 + B/2 R = Y + J
J = R - Y G = Y + K
K = G - Y B = 5/4 Y - 1/2 J - 1/4 K
```

An important detail is the number of colors. Since there are 4096 colors and 16 brightness levels for each one, there are actually 16 * 4096 = 65 536 possible colors. It turns out that in this mode the colors are not fully independent, which causes a reduction in the number of colors displayed simultaneously to 12 499.

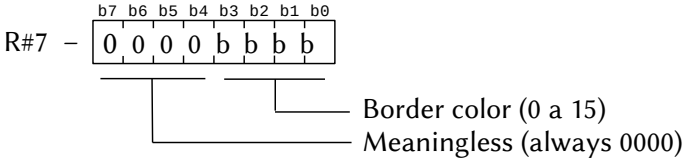
The adress of each point on the screen can be calculated by the following expression:

```
Address = X + Y*256 + Initial address
```

Where X is the horizontal coordinate and Y is the vertical coordinate. In this mode, the number of vertical points can be increased to 212. To do so, just set bit 7 of R#9, as illustrated below.

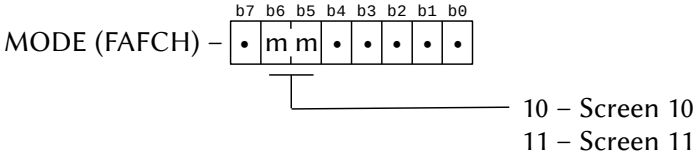


The border color in graphics mode 4 must be specified in the lowest 4 bits of R#7, as shown below, following the color palette, as in graphics mode 4.



In BASIC, this mode can be selected by Screen 10 or Screen 11. The difference between them is in the treatment given to them by the

BASIC interpreter. Screen 10 is treated as Screen 5 (or by the RGB system) and Screen 11 is treated as Screen 8 (or YJK system). To differentiate one from the other, a flag is used in the MODE (FAFCH) system variable, as illustrated below:

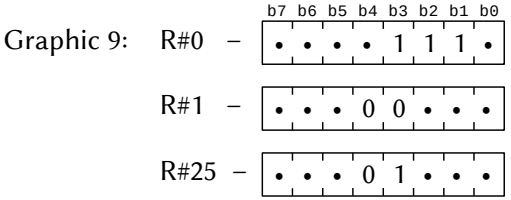


Note that BASIC does not have any special support for drawing on the YJK system and that switching between Screen 10 and Screen 11 does not clear the screen.

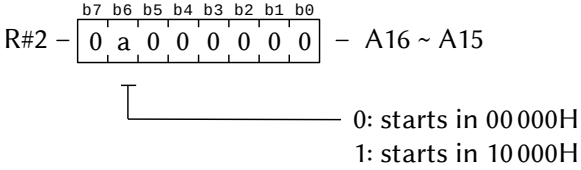
5.3.11 – Graphic mode 9

- 256 (horizontal) by 212 (vertical) points;
- Displays up to 19 268 simultaneous colors on the screen;
- Colors are specified for every four horizontal points;
- High speed hardware commands are available;
- Mode 2 sprites;
- Requires 53 Kbytes (256 points x 212 points) of memory;
- Map YJK chart;
- Screen 12 compatible.

Graphics mode 9 is selected by R#0, R#1 and R#25 as illustrated below:

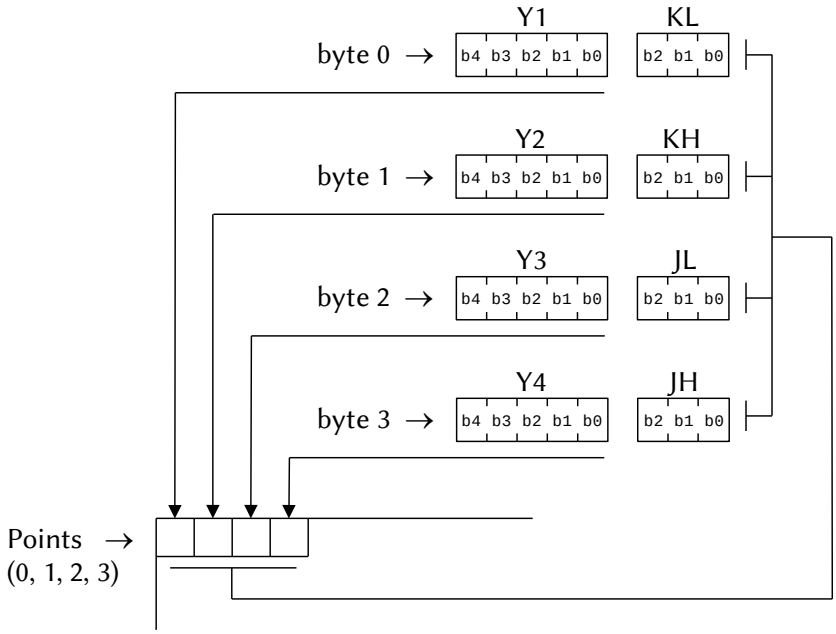


The organization of graphics mode 9 is similar but simpler than that of graphics mode 8. The address of the pattern generator table is specified in a single bit of R#2; therefore, the table always starts at 00 000H or 10 000H.

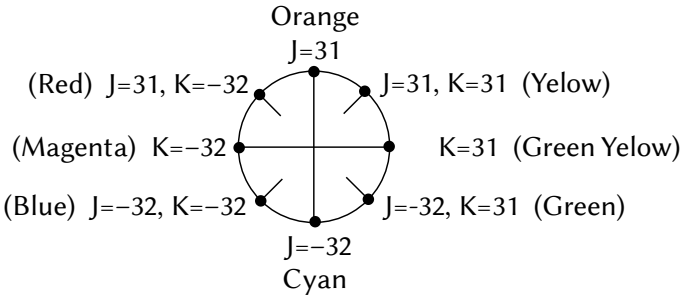


In graphic mode 9, the pure YJK system is used. The points are arranged four by four horizontally, with each group of 4 points having a single color, chosen from 4096, with up to 32 brightness levels for each individual point.

The color is chosen by the J and K vectors in exactly the same way as in graphic mode 8. The Y value, which is the brightness value, can vary from 11 111B (higher brightness) to 00 000B (lower brightness). Since there are 4096 colors and 32 brightness levels for each point, there are actually 131 072 possible colors (32 * 4096). However, because the points are not fully independent, there is a reduction in the number of colors that can be displayed simultaneously to 19 268. The arrangement of graphics mode 8 is illustrated below.



The colors are chosen by the J and K vectors with the same values as in graphic mode 8, whose illustration is reproduced below.



The binaries corresponding to the values shown in the illustration are as follows:

	b5	b4	b3	b2	b1	b0
0 →	0	0	0	0	0	0
31 →	0	1	1	1	1	1
−32 →	1	0	0	0	0	0
−1 →	1	1	1	1	1	1

Since there are 12 bits to represent the color (6 bits for J and 6 bits for K), we make 2^12 = 4096 colors, which is the maximum number of colors that can be defined. Each group of 4 horizontal points can only have one color chosen from these 4096. However, in mode 9, each individual point of this group can have a brightness range of 32 levels, represented by the Yn bits, where 00000B represents minimum brightness and 11111B maximum brightness .

The adress of each point on the screen for graphical mode 9 can be calculated by the following expression:

```
Address = X + Y*256 + Initial address
```

Where X is the horizontal coordinate and Y is the vertical coordinate. The number of vertical points must be specified in R#9, as already described. The border color must be specified in R#7, obeying the color palette, just like in graphics mode 4.

5.3.12 – System variables of the screen modes

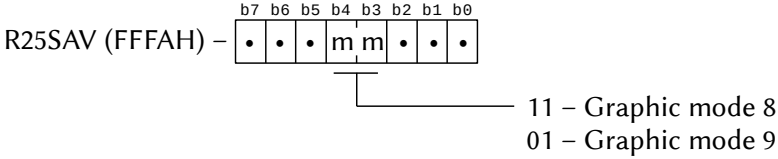
The following system variables are used by BIOS and BASIC to differentiate between screen modes:

```
LINL40  (F3AEH,1)
MODE    (FAFCH,1)
SCRMOD  (FCAFH,1)
R25SAV  (FFFAH,1)
```

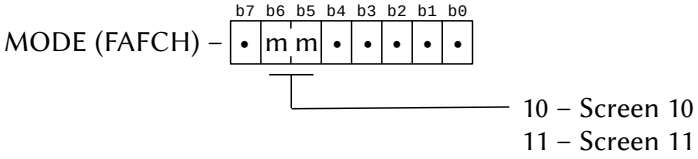
In multicolor and graphics modes 1 to 8, there is no secret; the value of the Screen in use is saved in the SCRMOD variable with the same value used by the interpreter in the SCREEN instruction.

In text modes, the value 0 is saved in the SCRMOD variable and the screen width in the LINL40 variable (1 to 40 for text mode 1 and 41 to 80 for text mode 2).

For graphic modes 8 and 9, the SCRMOD variable will always contain the value 8 (as in Screen 8). To differentiate one from the other, the variable R25SAV is used, as shown below.



To differentiate Screen 10 from Screen 11, you need to read the MODE system variable, as illustrated below.



5.4 – SPRITES (V9918/38/58)

Sprites are moving patterns or designs with 8x8 or 16x16 points. They are mainly used in games. There are two sprite modes for MSX2 onwards. Mode 1 is compatible with the TMS9918A of the MSX1. Mode 2 includes some new functions that have been implemented in the V9938

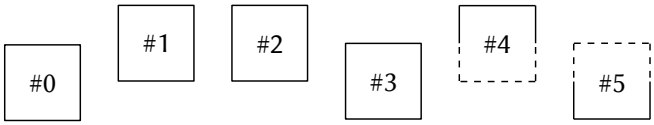
and V9958 VDPs. V9990 sprites are not compatible with V9918/38/58 sprites and will be described later.

Up to 32 sprites can be displayed simultaneously on the screen. They can have 2 sizes: 8x8 or 16x16 points. Only one size can be displayed on the screen at a time. The size of a sprite point is normally the size of one screen point, but in graphics modes 5 and 6 (which has a resolution of 512x212), the horizontal size is two screen points, so the absolute size of the sprite is always the same in any screen mode.

The sprite mode is automatically selected according to the screen in use. For graphics 1 and 2 and multicolor, mode 1 is selected and for graphics modes 3 to 9, mode 2 is selected. Text modes do not support sprites.

5.4.1 – Mode 1 sprites

Mode 1 sprites are exactly the same as MSX1 sprites. There can be up to 32 sprites numbered from 0 to 31 on the screen. Lower numbered sprites have higher display priority. When sprites are placed on the same horizontal line, up to 4 higher priority sprites are displayed in full. The part of sprites with priority greater than 3 (5th sprite onwards) coexisting on the same horizontal line is not shown, as shown in the illustration below.



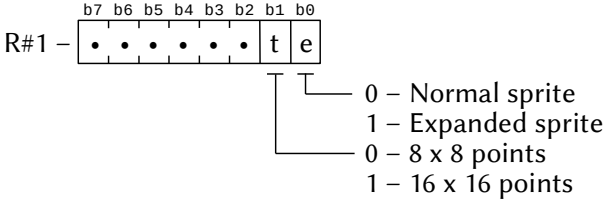
Mode 1 sprites priority

In the illustration, sprites #0, #1, #2, and #3 are shown whole, but only the top half of sprite #4 appears, as well as the bottom half of sprite #5. The dashed part is not displayed.

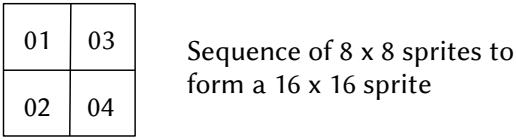
The size of the sprites, 8 x 8 or 16 x 16, is selected by bit 1 of R#1. The default size is 8 x 8 points.

Sprites can also be expanded to double the size, both vertically and horizontally. In this case, each point of the sprite corresponds to four points on the screen (except in graphics modes 5 and 6, where it

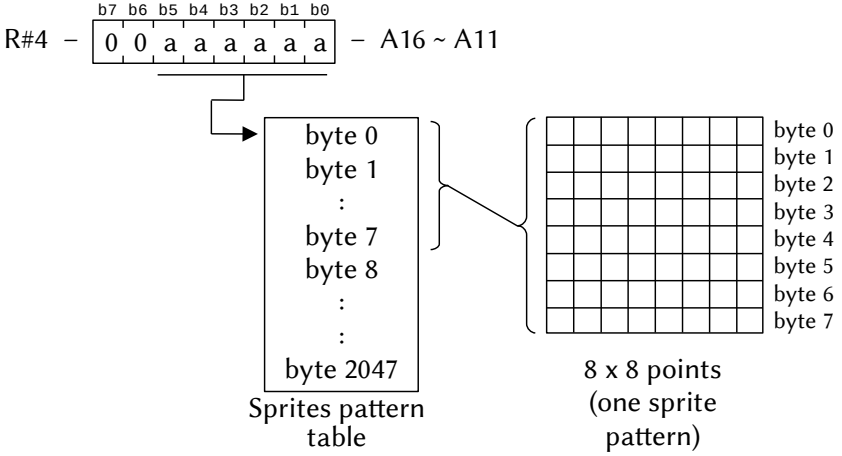
will correspond to 8 points, being 4 horizontally and 2 vertically). This function is controlled by bit 0 of R#1.



Sprite defaults are defined in VRAM. Up to 256 sprites can be defined if the size is 8 x 8, and up to 64 if the size is 16 x 16. Patterns are numbered from 0 to 255. To form a 16 x 16 sprite, 4 sprites 8 x 8 are used, in the sequence shown below:



The adress of the sprites pattern table is specified in R#16. Only the highest 6 bits are specified; therefore, the table always starts at a multiple of 2 Kbytes from 00 000H. The sprite pattern table is 2048 bytes long and reserves 8 bytes for each 8 x 8 sprite, as illustrated below.



When the bit corresponding to the sprite's point is 0, the respective point will be transparent; when the bit is 1, the point will have the

color specified in the Sprite Attributes Table. This table has 128 bytes and reserves 4 bytes for each sprite to be displayed on the screen, with a maximum of 32 simultaneous sprites. It starts at the adress pointed to by R#11 and R#5. As only the highest 7 bits are specified, the table always starts at a multiple of 1 Kbyte from 00000H. The four bytes reserved by the table for each sprite contain the following data:

Y Coordinate: Specifies the vertical coordinate of the top left corner of the sprite. The top line of the screen is not 0, but 255. By setting this value to 208 (D0H), all lower priority sprites are not shown.

X Coordinate: Specifies the horizontal coordinate of the upper left corner of the sprite.

Pattern #: Specifies which pattern from the sprites pattern generator table will be displayed.

Color code: Specifies the color, according to the palette, of the bits set to 1 in the pattern table.

EC: by setting this bit to 1, the respective sprite will be shifted 32 points to the left of the specified coordinate.

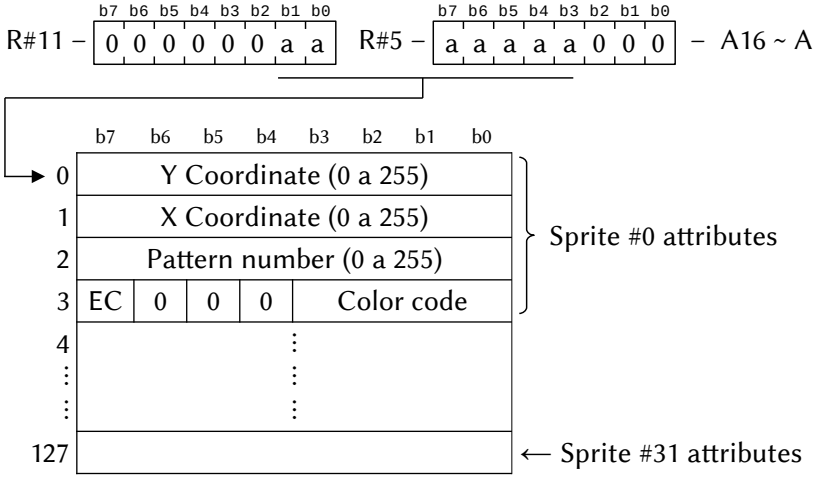
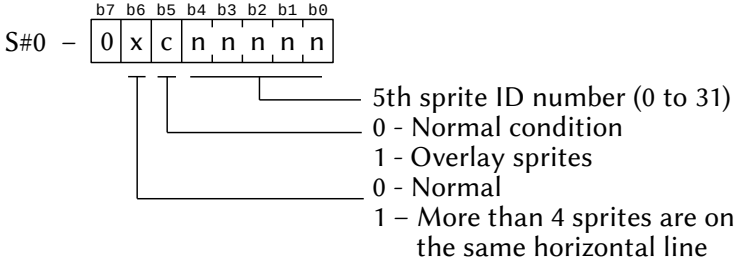


Tabela de atributos dos sprites

When two sprites overlap on the screen, bit 5 of S#0 is set, informing the situation. The overlap or conflict information only happens

when the 1 bits meet, that is, when the "drawn" part of the sprites overlap. When more than four sprites are placed on the same horizontal line, bit 6 of S#0 is set and the 5th sprite number is placed on the lowest 5 bits of S#0.

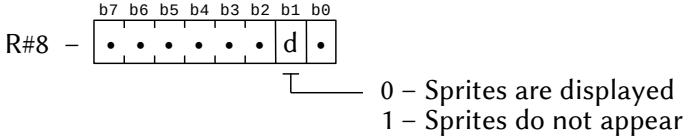


5.4.2 – Mode 2 sprites

Mode 2 sprites have been added to V9938/9958 VDPs bringing new features and greater flexibility than mode 1 sprites.

The number of sprites that can be displayed simultaneously is 32, and up to 8 sprites can be placed on the same horizontal line. Smaller-numbered sprites have higher display priority, as in mode 1. Sprite size (8x8 or 16x16) and expansion to double the size are selected in the same way as for mode 1 sprites.

Mode 2 sprites have an on-screen display on-off function, controlled by bit 1 of R#8. When this bit is 0, sprites will appear normally on the screen, but when it is 1, no sprites will appear.



The pattern generator table is set in the same way as for mode 1, but the attribute table has changed. In mode 2 sprites, a different color can be specified for each line of the sprite. This information is stored in the Sprite Color Table, which is independent of the attribute table. The attribute table stores the following information:

Y coordinate: Vertical coordinate of the sprite. The top line of the screen is not 0, but 255. By setting this value to 208 (D0H), all sprites

with higher priority are not shown. Putting it at 216, it is the lowest priority sprites that are not shown.

X Coordinate: Specifies the horizontal coordinate of the sprite.

Pattern No.: Specifies which pattern from the pattern generator table will be displayed.

The sprite color table is automatically set 512 bytes before the initial address of the attribute table. It allocates 16 bytes for each pattern, and each line of each sprite contains the following data:

Color: Color of the respective line, which can vary from 0 to 15, according to the color palette.

EC: When this bit is 1, the respective line will be shifted 32 points to the left of the specified coordinate.

CC: When this bit is 1, the respective sprite will have the same priority as the higher priority sprites. When sprites of the same priority overlap, a logical OR operation is performed between the sprite colors to determine the new color. In this case, the overlap does not conflict and is not detected as a sprite collision.

IC: When this bit is 1, the respective line will not cause conflict when overlapping with other sprites.

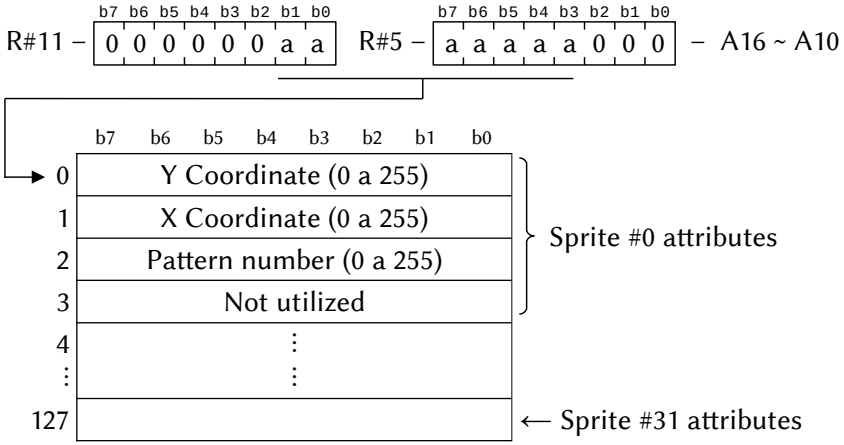
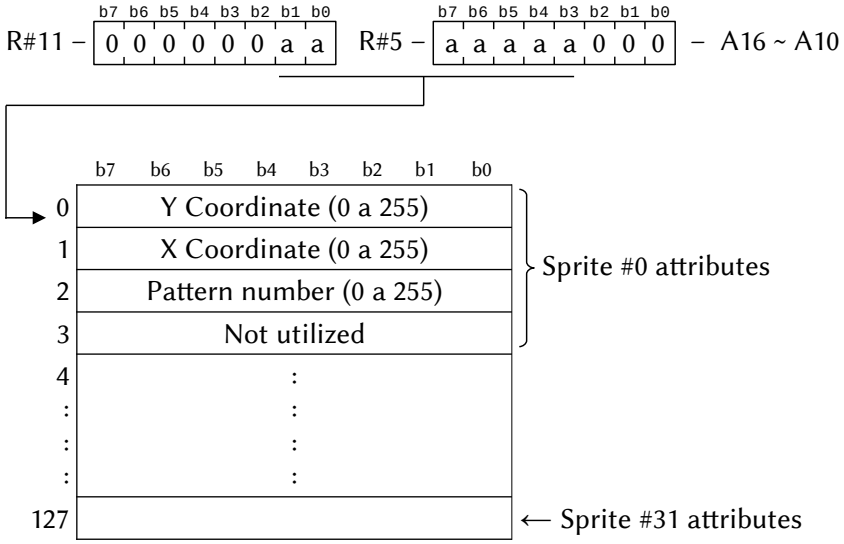
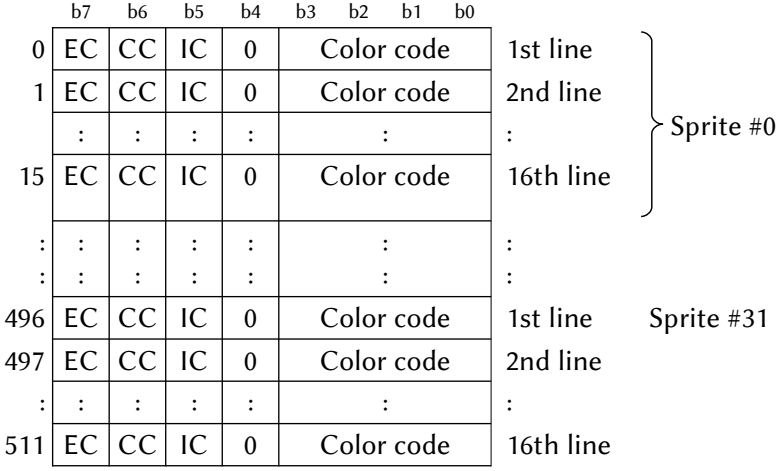


Tabela de atributos dos sprites



Sprites attributes table

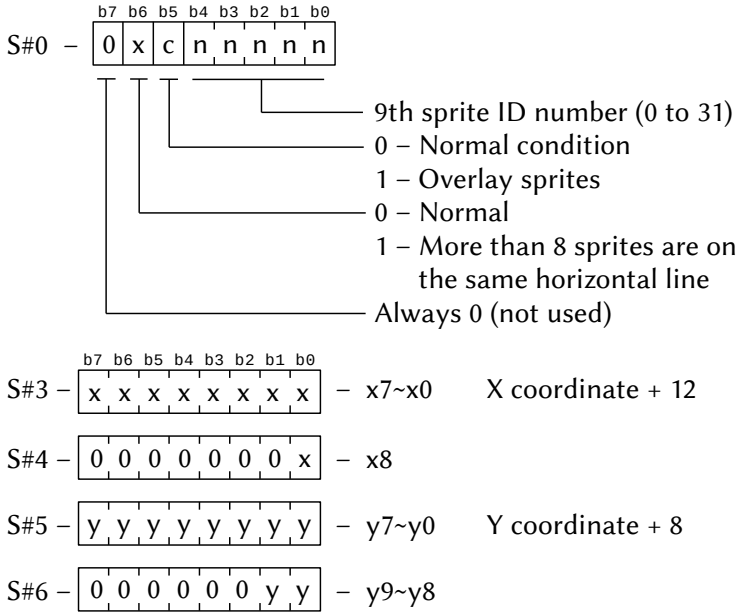


Sprites color table

Overlapping or conflicting mode 2 sprites is detected when the sprite's spot color is not transparent and the CC bits of the sprite's color table are 0. When overlap is detected, bit 5 of S#0 is set to 1 and the overlap coordinate is placed in S#3 to S#6. The coordinate returned by

these records is not the one where the conflict occurred. To get the exact coordinates, the following expression must be used:

```
X coordinate = (S#3 and S#4) - 12
Y coordinate = (S#5 and S#6) - 8
```



When more than 8 sprites are placed on the same horizontal line, bit 6 of S#0 is set to 1 and the plane number of the lowest priority sprite (9th sprite) is placed in the lowest 5 bits of S#0, as above illustration.

5.5 – VDP COMMANDS (V9938/58)

MSX-VIDEO can perform some basic graphical operations called VDP Commands. They are performed by hardware and are available for graphics modes 4 to 9. When VDP commands are executed, the location of the start and destination points are represented by coordinates (X, Y) and there is no page division; the 128 Kbytes of VRAM are treated as a single block, as illustrated below.

Graphic 4 (Screen 5)

(0,0)	(255,0)
Page 0	
(0,255)	(255,255)
(0,256)	(255,256)
Page 1	
(0,511)	(255,511)
(0,512)	(255,512)
Page 2	
(0,767)	(255,767)
(0,768)	(255,768)
Page 3	
(0,1023)	(255,1023)

Adress

00000H

07FFFH

0FFFFH

17FFFH

1FFFFH

Graphic 5 (Screen 6)

(0,0)	(511,0)
Page 0	
(0,255)	(511,255)
(0,256)	(511,256)
Page 1	
(0,511)	(511,511)
(0,512)	(511,512)
Page 2	
(0,767)	(511,767)
(0,768)	(511,768)
Page 3	
(0,1023)	(511,1023)

Graphics 7 to 9
(Screens 8 to 12)

(0,0)	(255,0)
Page 0	
(0,255)	(255,255)
(0,256)	(255,256)
Page 1	
(0,511)	(255,511)

Endereço

00000H

0FFFFH

1FFFFH

Graphic 6
(Screen 7)

(0,0)	(511,0)
Page 0	
(0,255)	(511,255)
(0,256)	(511,256)
Page 1	
(0,511)	(511,511)

5.5.1 – VDP commands description

There are 12 commands available in MSX-VIDEO. These commands are summarized in the table below.

Command type	Mnemonic	Source	Dest	Unit	Code R#46-4msb
Fast movements	HMMC	CPU	VRAM	bytes	1111
	YMMM	VRAM	VRAM	bytes	1110
	HMMM	VRAM	VRAM	bytes	1101
	HMMV	VDP	VRAM	bytes	1100
Logical movements	LMMC	CPU	VRAM	points	1011
	LMCM	VRAM	CPU	points	1010
	LMMM	VRAM	VRAM	points	1001
	LMMV	VDP	VRAM	points	1000

Line	LINE	VDP	VRAM	points	0111
Search	SRCH	VDP	VRAM	points	0110
Pset	PSET	VDP	VRAM	points	0101
Point	POINT	VRAM	VDP	points	0100
Reserved	-	-	-	-	0011
Reserved	-	-	-	-	0010
Reserved	-	-	-	-	0001
Stop command	STOP	-	VDP	-	0000

When data is written to R#46 (command register), the VDP starts executing the command and sets bit 0 (CE / command execute) of the S#2 status register. The necessary parameters must be placed in R#32 to R#45 before the command is executed. When command execution ends, bit 0 if S#2 is reset (0). To stop the execution of a command, the stop command (0000B) can be used. The VDP commands only work in graphics modes 4 to 9, but in modes 8 and 9 they should be used with caution as the screen may blur, as in these modes the points are arranged in blocks of four horizontally.

5.5.2 – Logical operations

When the commands are executed, various logical operations can be done between the VRAM and a specified data. These operations are described in the table below.

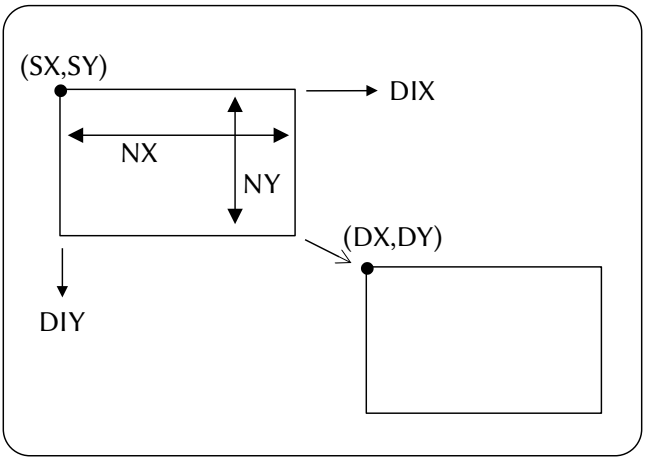
NAME	OPERATION	R#46-4lsb
IMP	DC = SC	0000
AND	DC = SC and DC	0001
OR	DC = SC or DC	0010
XOR	DC = SC xor DC	0011
NOT	DC = not (SC)	0100
TIMP	If SC=0, DC = DC else DC = SC	1000
TAND	If SC=0, DC = DC else DC = SC and DC	1001
TOR	If SC=0, DC = DC else DC = SC or DC	1010
TXOR	If SC=0, DC = DC else DC = SC xor	1011
TNOT	If SC=0, DC = DC else DC = not (SC)	1100

In the table, SC represents the source color code and DC the destination color code. IMP, AND, OR, XOR and NOT are the possible logical operations. In operations with names preceded by "T", the origin points that have the color 0 will not be object of logical operation in the destination. Using this feature, only the colored portions are superimposed. This feature is especially effective for animations.

5.5.3 – Area specification

Area move commands transfer data within an area specified by a rectangle. The area to be transferred is specified by a vertex, from which the sizes of the sides of the rectangle are informed, along with the direction in which the data will be transferred and the destination coordinates.

SX and SY are the origin coordinates; NX and NY are the length of each side of the rectangle in points and DIX and DIY specify the direction in which the data will be transferred and depend on the type of command. DX and DY specify the destination coordinates.



Specifying Areas for VDP Commands

5.5.4 – Using the VDP commands

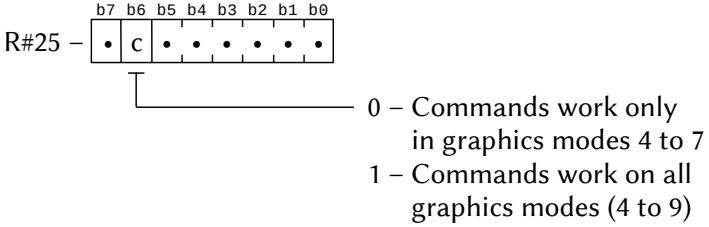
VDP commands are classified into three types: high speed transfer commands, logical transfer commands, and drawing commands.

They must be accessed directly; therefore, some care must be taken with the synchronization, waiting for the VDP to be ready. There must be an 8 μs pause between consecutive accesses (for this, a loop with the OUTI instruction, not the OTIR instruction, can be used for a standard MSX machine at 3.58 MHz). On the MSX turbo R, the MSX Engine S1990 generates 8 μs pauses for every byte written to one of the VDP ports, so there is no need to worry about synchronization on these machines. It is important to note, however, that once the data is written, it is stored in the S1990, freeing up the CPU. But if a data is written immediately afterwards, a HALT will be generated until reaching the necessary 8 μs. This feature can be used to have the R800 process data between consecutive accesses to the VDP.

It must also wait for the VDP to finish executing the command to send another command or data, which must be done by reading the CE bit of the S#2 status register. For this, the following routine can be used:

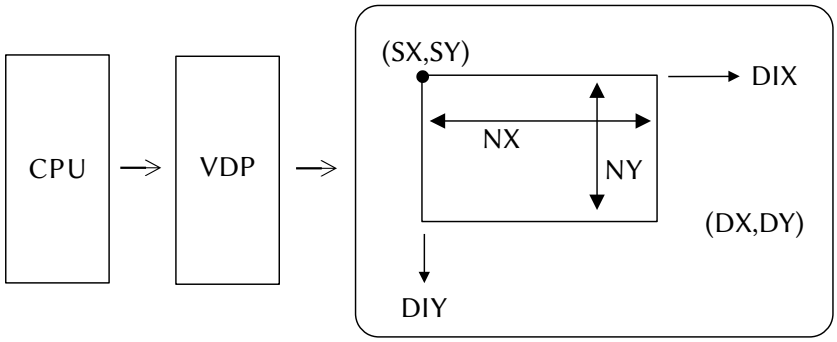
```
WAIT:   LD A,2
        CALL STATUS
        AND 1
        JR NZ,WAIT
        XOR A
        CALL STATUS
        RET
        ;
STATUS: OUT 099H,A
        LD A.08FH
        OCT 099H,A
        IN A,099H
        RET
```

In the case of using commands in V9958, there is a quirk. Note bit 6 of R#25 (CMD). If this bit is 0, commands will only work in graphics modes 4 to 7. It is the default value. In order for the commands to work in graphics modes 8 and 9, it is necessary to set this bit to 1. In this case, the commands will work in the same way as for graphics mode 7.

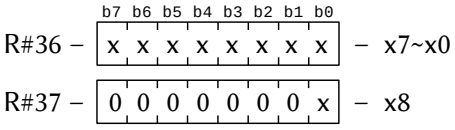


5.5.4.1 – HMMC (Byte transfer – CPU → VRAM)

In this command, data is transferred from the CPU to a specified area in the VRAM. Logical operations are not possible; data is transferred in bytes.

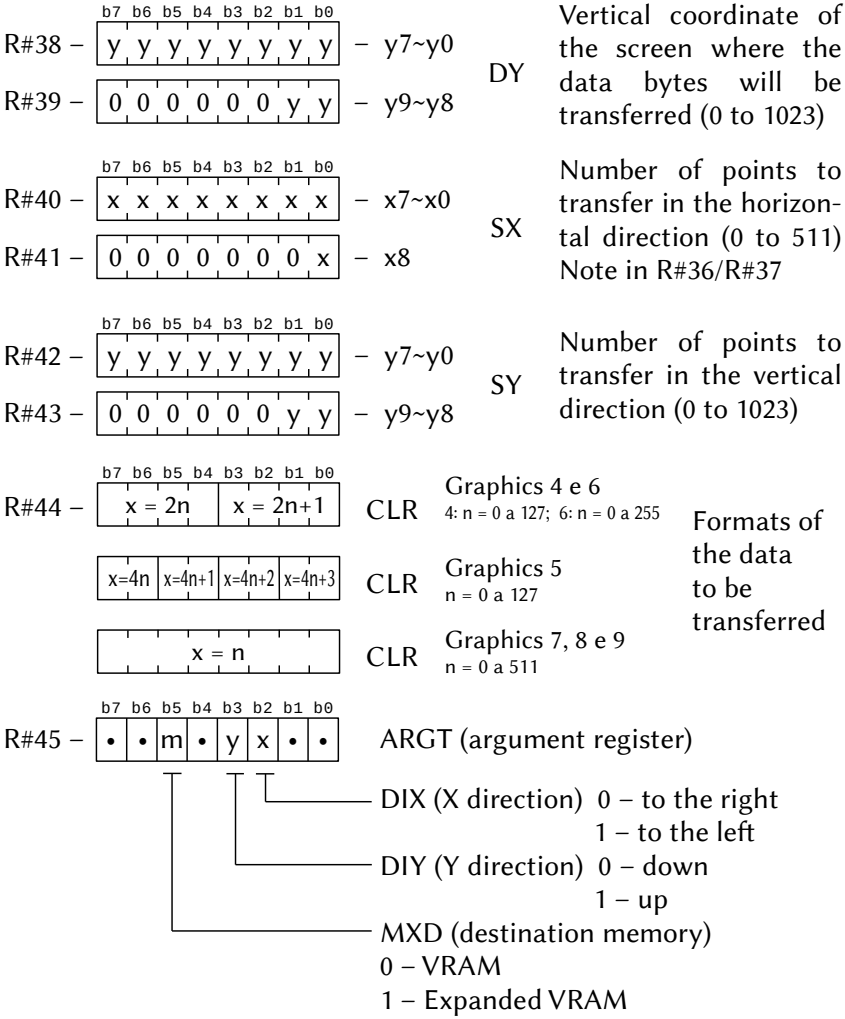


The parameters shown in the table below must be placed in the appropriate registers. At this point, the first byte of data to be transferred from the CPU to the VRAM must be written in R#44. To execute the command, it is necessary to write the command code F0H in R#46 and the byte contained in R#44 will be written in VRAM. Then, the VDP will wait for the second byte of data, which must also be written in R#44, and so on. The byte will only be transferred after the VDP receives it (if the TR bit of S#2 is 1). When the CE bit of S#2 is 0, it means that all data bytes have been transferred.

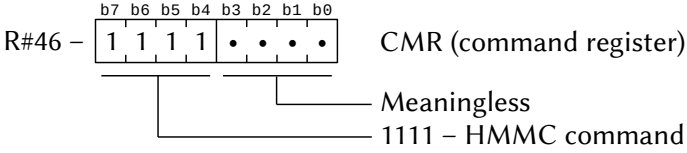


DX — Horizontal coordinate of the screen where the data bytes will be transferred (0 to 511)

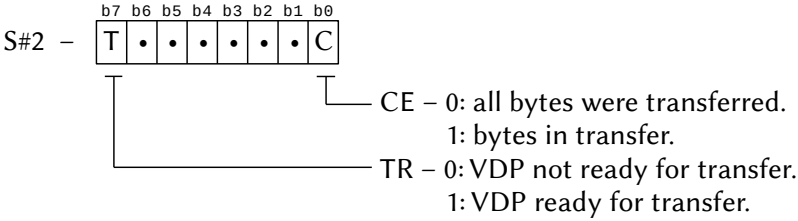
For graphics modes 4 and 6 the lowest adress bit (b0) is not valid as each byte transfers two points. For graphic mode 5, the lower two bits (b1 and b0) are not valid because each byte transfers 4 points. For graphics modes 7, 8 and 9 all bits are valid.



To execute the HMMC command, just write the F0H value in R#46:

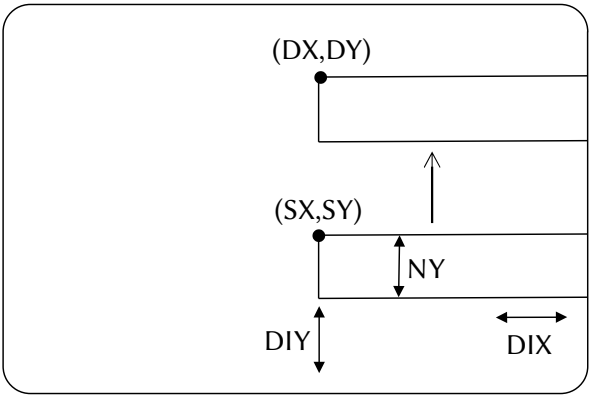


The transfer status can be read from S#2:

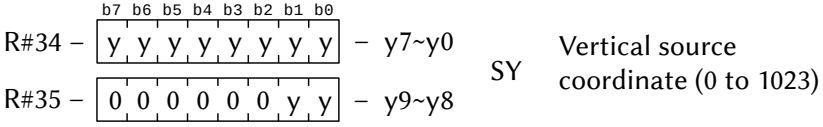


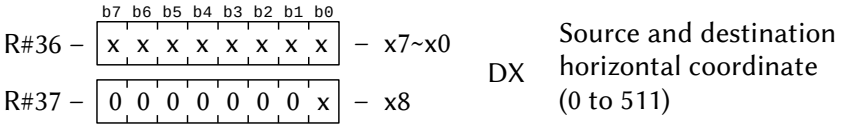
5.5.4.2 – YMMM (Byte transfer – VRAM direction Y)

In this command, data from a specified area of VRAM is transferred to another area of VRAM. Data is transferred only in the Y (vertical) direction, as illustrated below:

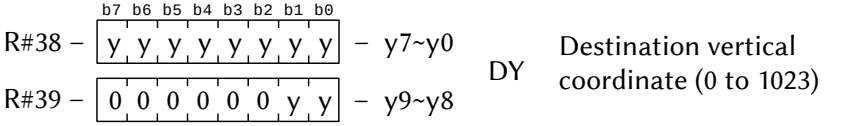


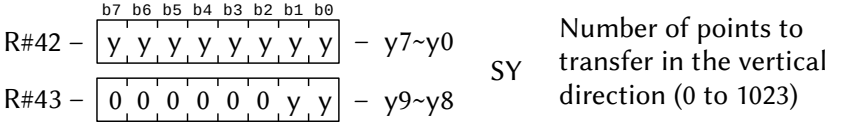
The registers must be loaded according to the illustration below.

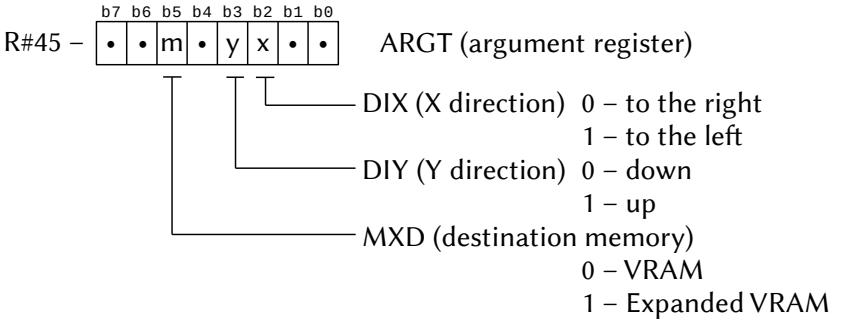




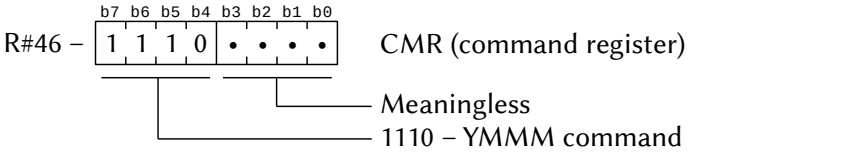
For graphics modes 4 and 6 the lowest adress bit (b0) is not valid as each byte transfers two points. For graphics mode 5, the lower two bits (b1 and b0) are not valid because each byte transfers 4 points. For graphics modes 7, 8 and 9 all bits are valid.



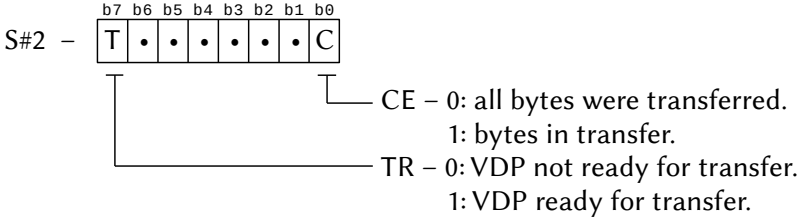




To execute the YMMM command, just write the value E0H in R#46:

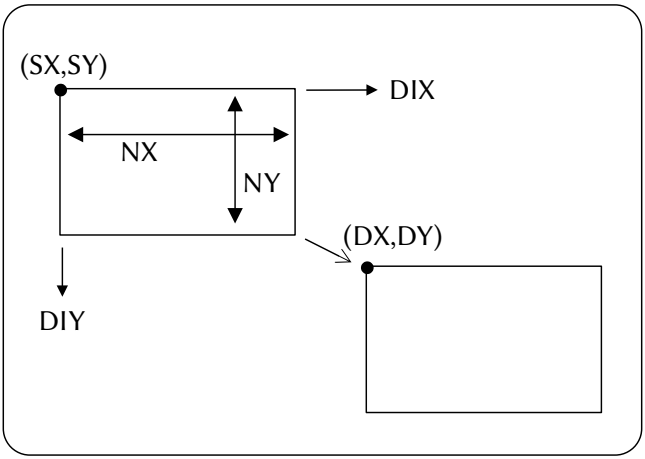


The transfer status can be read from S#2:

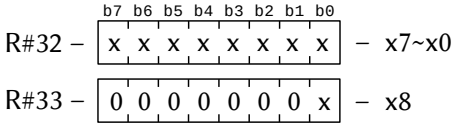


5.5.4.3 – HMMM (Byte transfer – VRAM → VRAM)

In this command, data is transferred from one area of the VRAM to another. Data is transferred in rectangular areas, as illustrated below:

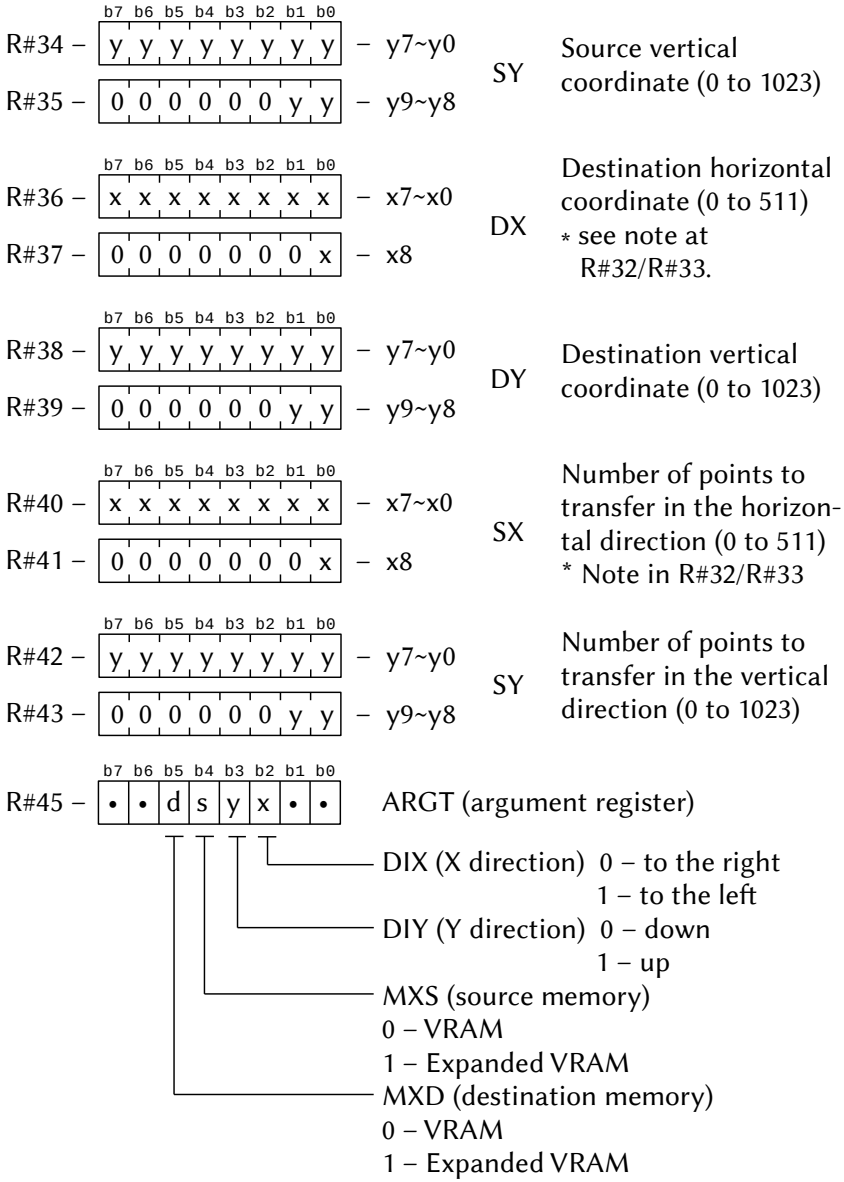


The following registers must be loaded:

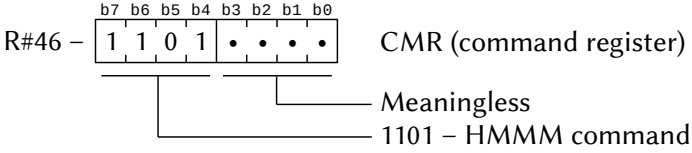


SX — Source horizontal coordinate (0 to 511)

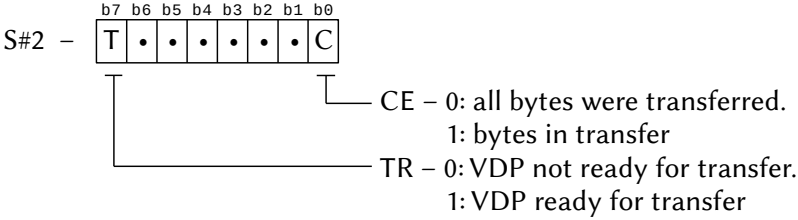
For graphics modes 4 and 6 the lowest adress bit (b0) is not valid as each byte transfers two points. For graphic mode 5, the lower two bits (b1 and b0) are not valid because each byte transfers 4 points. For graphics modes 7, 8 and 9 all bits are valid.



To execute the HMMM command, just write the D0H value in R#46:

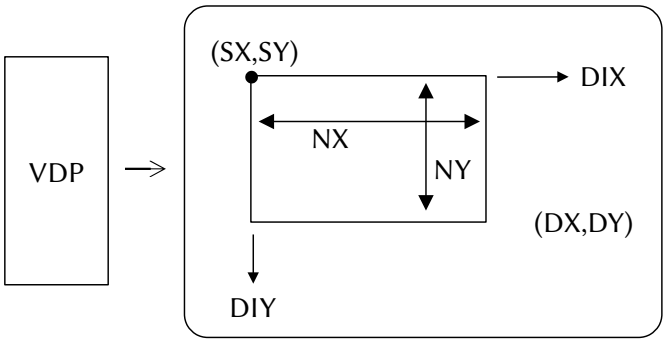


The transfer status can be read from S#2:

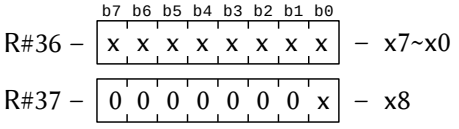


5.5.4.4 – HMMV (Draw rectangle in bytes)

In this command, each byte of data specified is drawn in the VRAM with the respective color code, as shown in the illustration.



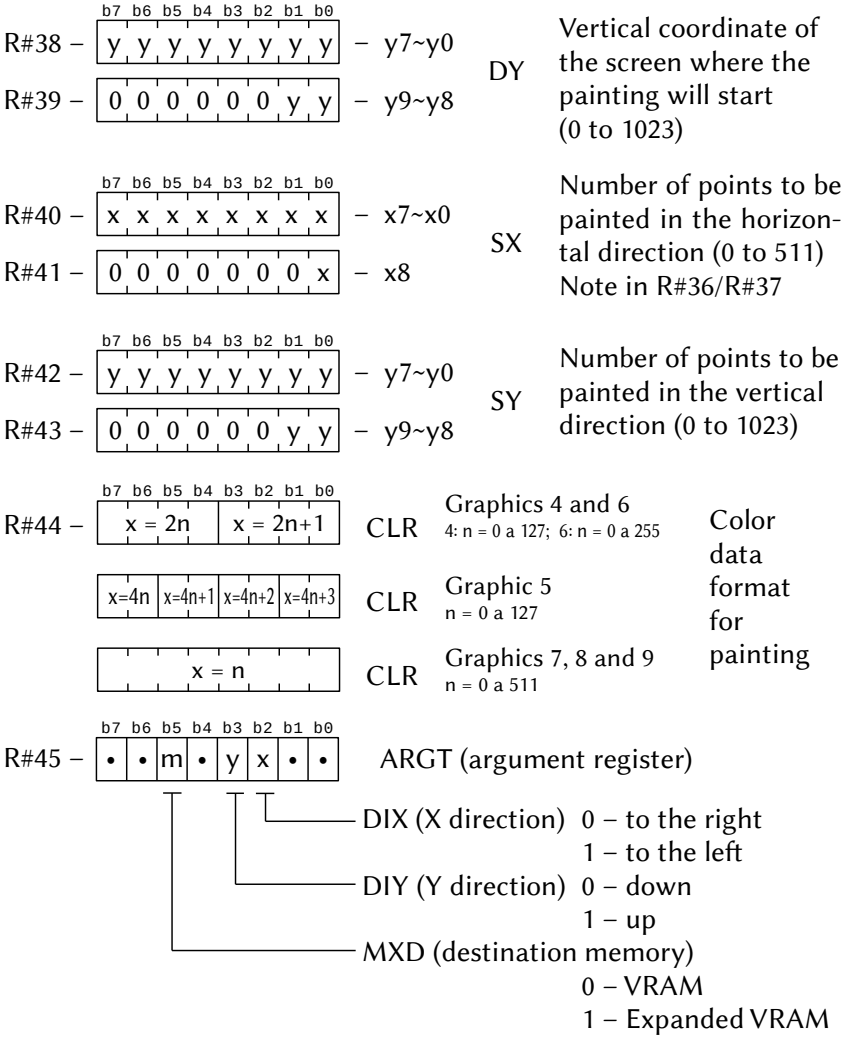
The following registers must be loaded:



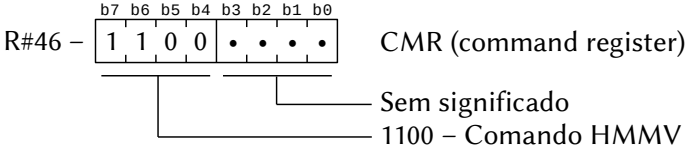
DX — Horizontal coordinate of the screen where the painting will start (0 to 511)

For graphics modes 4 and 6 the lowest Address bit (b0) is not valid as each byte transfers two points. For graphics mode 5, the lower two

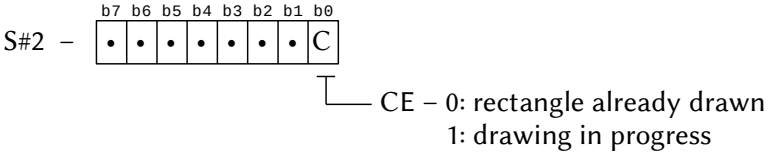
bits (b1 and b0) are not valid because each byte transfers 4 points. For graphics modes 7, 8 and 9 all bits are valid.



To execute the HMMV command, just write the C0H value in R#46:

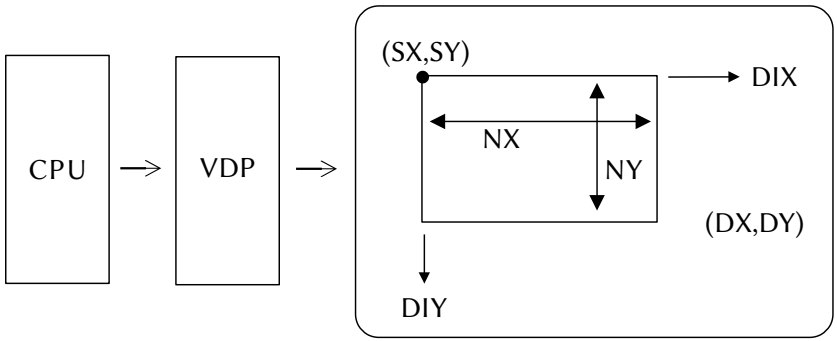


The paint status can be read from S#2:

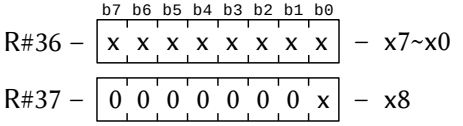


5.5.4.5 – LMMC (Logical transfer – CPU → VRAM)

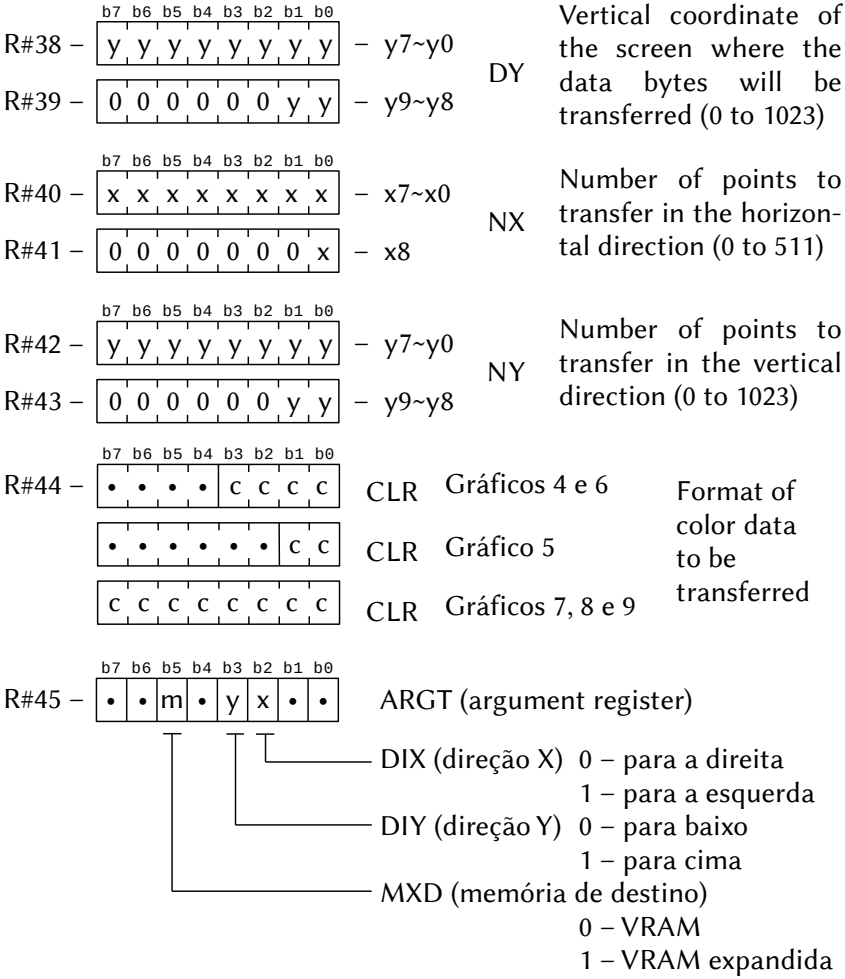
In this command, data bytes are transferred from the CPU to a specific area of the VRAM in points. Logical operations during transfer are possible. In logical transfer commands, data is transferred in points and one byte is required for each point in all screen modes. Therefore, all horizontal coordinate bits are valid. The logic operation code must be specified in the lower 4 bits of register R#46.



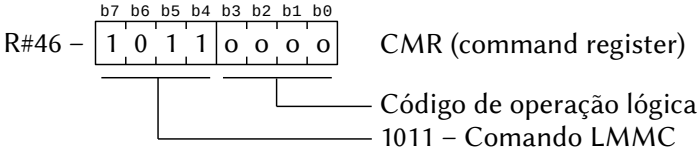
The following registers must be loaded:



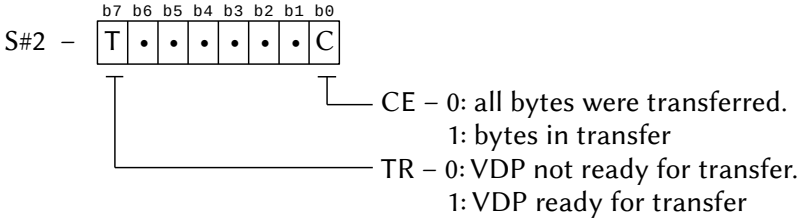
DX — Horizontal coordinate of the screen where the data bytes will be transferred (0 to 511)



To execute the LMMC command, the value 1011B must be written in the highest four bits of R#46 and the logic opcode in the lowest four bits.



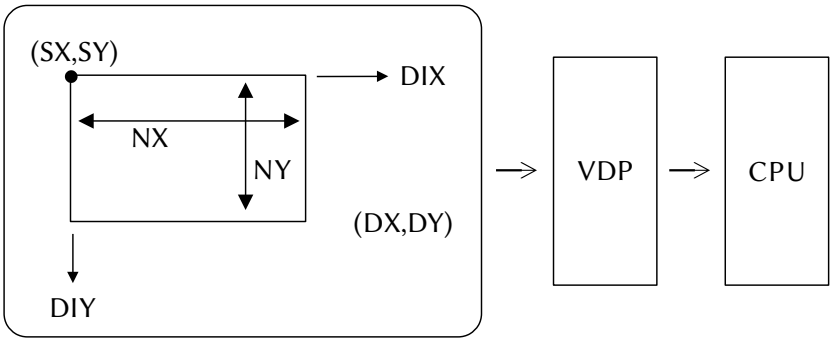
The transfer status can be read from S#2:



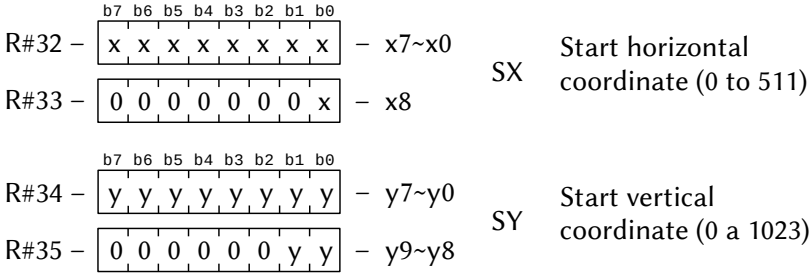
5.5.4.6 – LMCM (Logical transfer – VRAM → CPU)

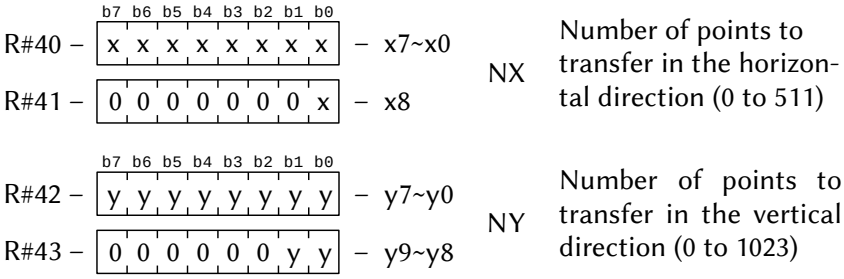
In this command, data is transferred from a specified area in VRAM to the CPU in points. One byte is required for each point.

When starting the execution of this command, the CPU must check the TR bit of S#2. If this bit is 1, the data byte is available to be read from S#7. When the CE bit of S#2 is 0, the data bytes to be transferred have ended.

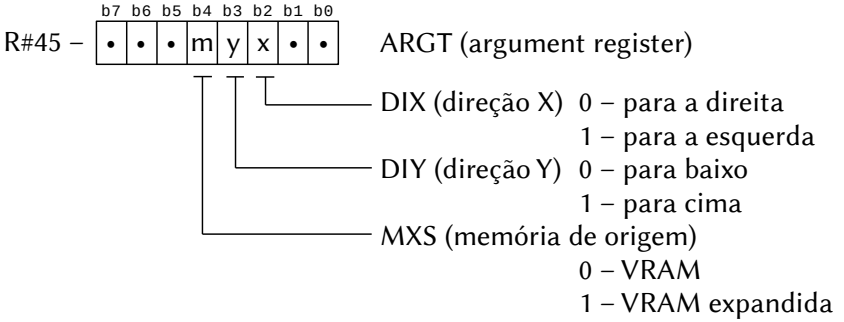


The following registers must be loaded:

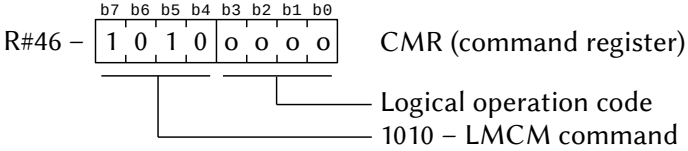




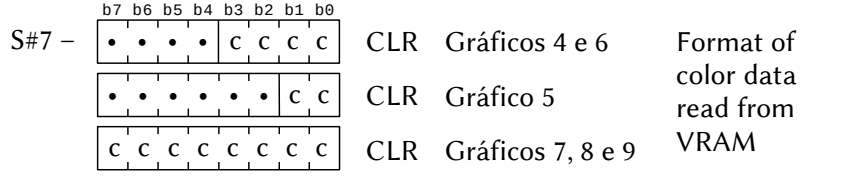
R#44 – Not used by this command.



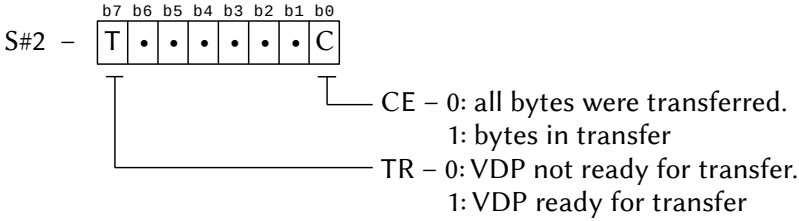
To execute the LMCM command, the value 1010B must be written to the highest four bits of R#46 and the logic opcode to the lowest four bits.



The value of the bytes read is available in the S#7 register, in the format illustrated below. When the last data is written to S#7 and the TR bit of S#2 is 1, the command will be terminated by the VDP and the CE bit of S#2 will be 0.

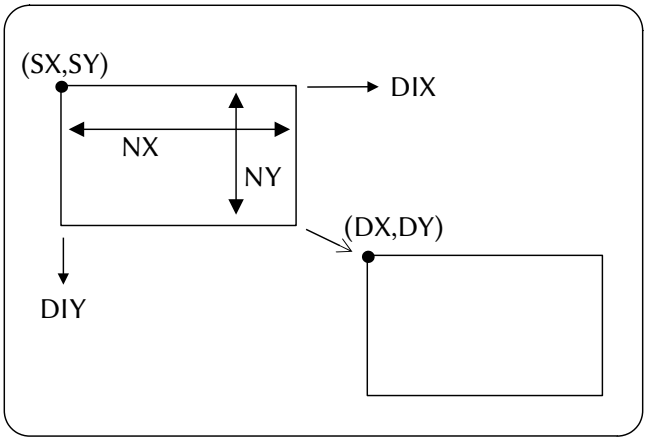


The transfer status can be read from S#2:

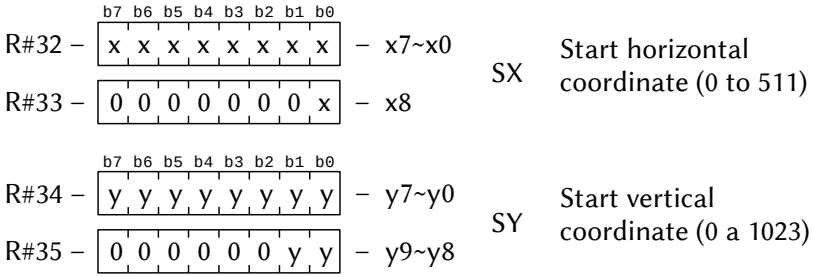


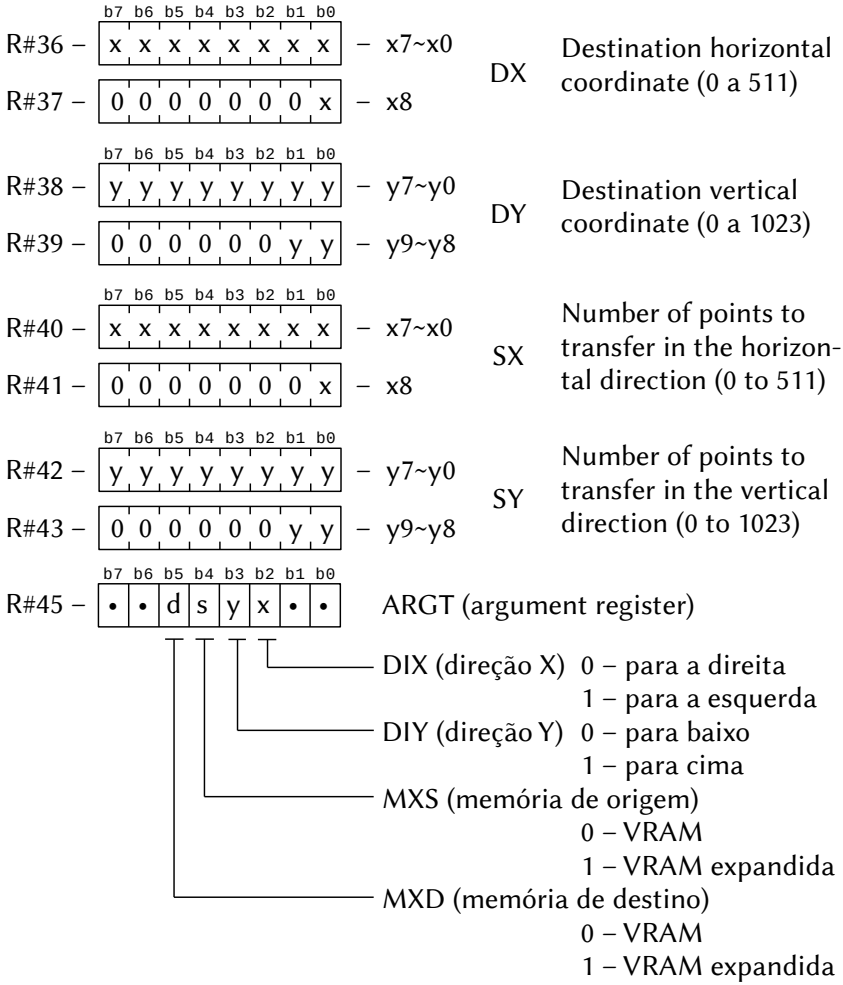
5.5.4.7 – LMMM (Logical transfer – VRAM → VRAM)

In this command, data from a specified area of VRAM is transferred to another area of VRAM in points. Logical operations on the destination are possible. As long as the CE bit of S#2 is 1, the command is being executed.

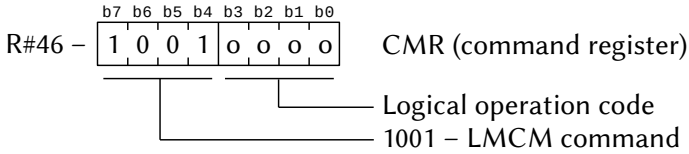


The following registers must be loaded:

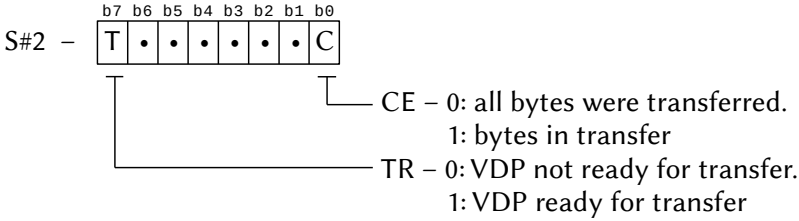




To execute the LMMM command, you must write the value 1001B in the highest four bits of R#46, with the lowest four bits containing the logic opcode.

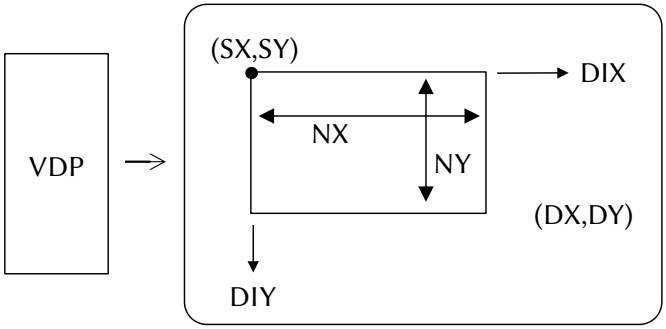


The transfer status can be read from S#2:

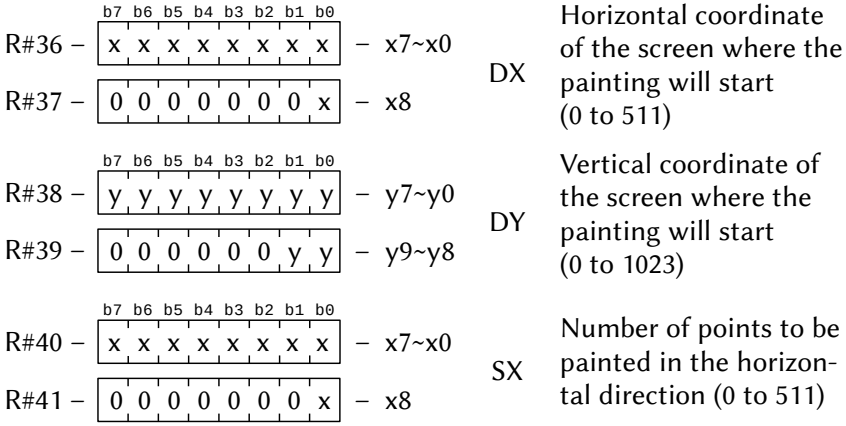


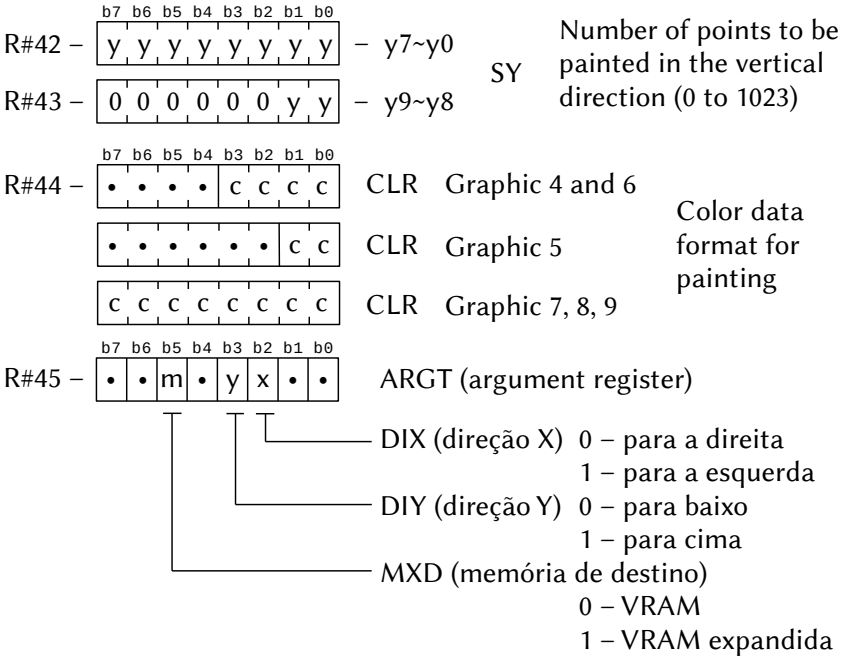
5.5.4.8 – LMMV (Logical VRAM painting)

Any rectangular area of the VRAM can be painted with points of a certain color and logical operations on the destination with the specified color are possible.

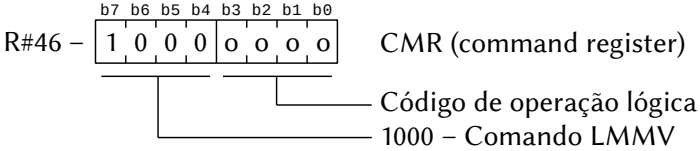


The following registers must be loaded:

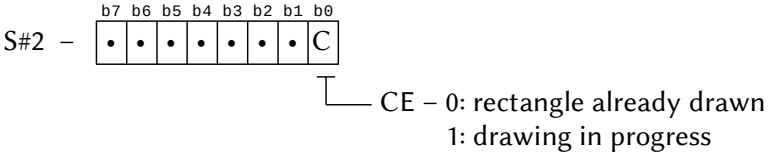




To execute the LMMV command, the value 1000B must be written in the four highest bits of R#46 and the four lowest bits must contain the logic operation code.



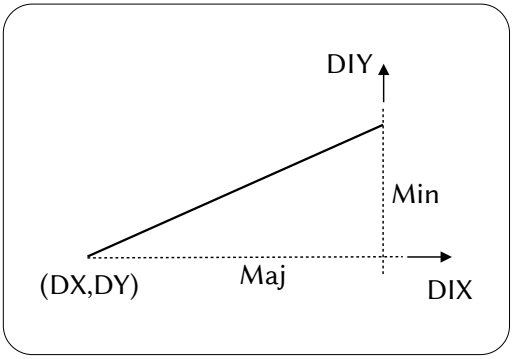
The paint status can be read from S#2:



5.5.4.9 – LINE (Draw a line)

This command draws a line between screen coordinates. The parameters are specified including the coordinate (X,Y) of the start of the

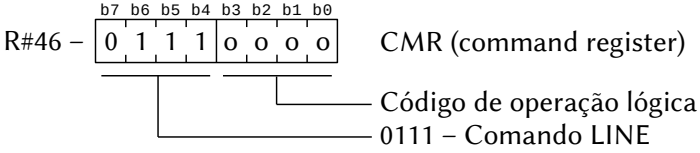
line and the horizontal and vertical length to the end point, as shown in the illustration below:



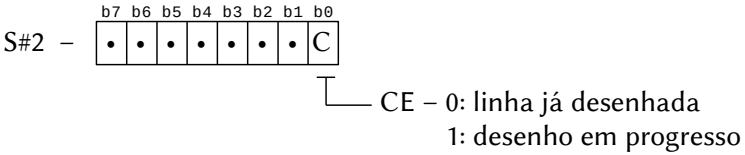
Before executing the LINE command, the following registers must be loaded:

R#36 – b7 b6 b5 b4 b3 b2 b1 b0: x x x x x x x x – x7~x0	DX	Horizontal coordinate of the screen from which the line will be drawn (0 to 511)
R#37 – 0 0 0 0 0 0 0 x – x8		
R#38 – b7 b6 b5 b4 b3 b2 b1 b0: y y y y y y y y – y7~y0	DY	Vertical coordinate of the screen from which the line will be drawn (0 to 1023)
R#39 – 0 0 0 0 0 0 y y – y9~y8		
R#40 – b7 b6 b5 b4 b3 b2 b1 b0: x x x x x x x x – x7~x0	Maj	Number of points on the largest leg of the reference right triangle for drawing
R#41 – 0 0 0 0 0 0 0 x – x8		
R#42 – b7 b6 b5 b4 b3 b2 b1 b0: y y y y y y y y – y7~y0	Min	Number of points on the smaller side of the reference right triangle for drawing
R#43 – 0 0 0 0 0 0 y y – y9~y8		

To execute the LINE command, the value 0111B must be written in the four highest bits of R#46, and the four lowest bits must contain the logic operation code.



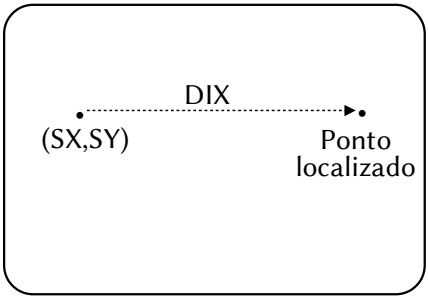
The drawing status can be read in S#2:



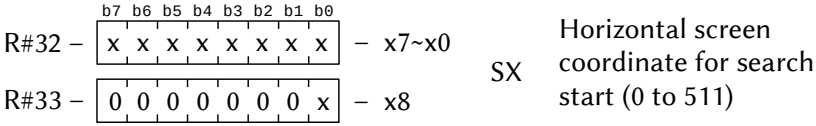
5.5.4.10 – SRCH (Search color code)

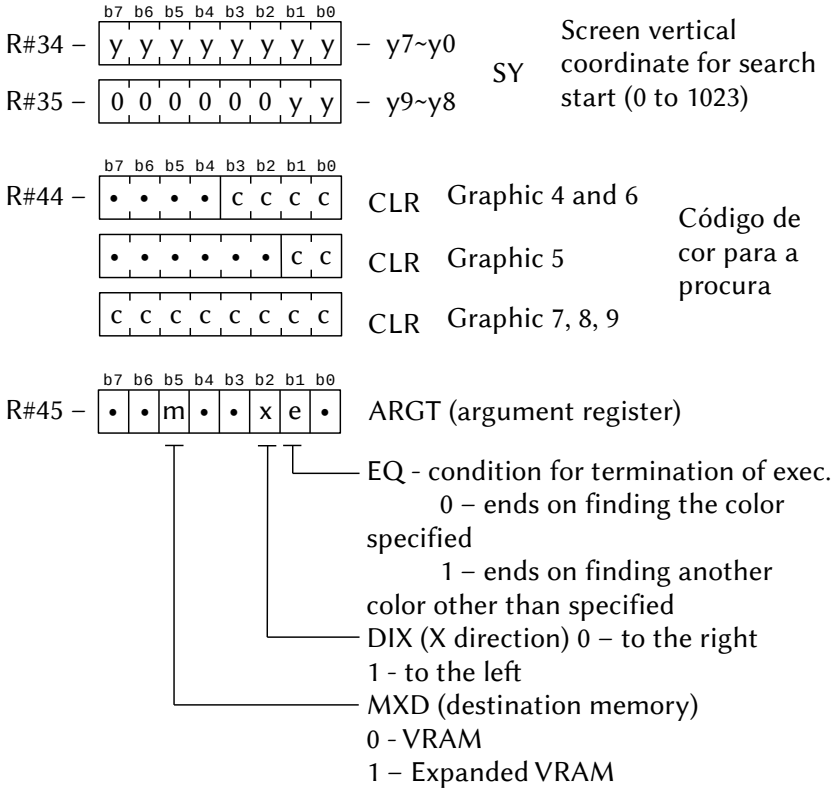
This command looks for the existence of a point with a specific color from a coordinate in the VRAM, always horizontally, to the left or to the right. It is a useful command for painting or filling routines.

The command ends when the point with the specified color is found or when the edge of the screen is reached. As long as the CE bit of S#2 is 1, the command is being executed. At the end of the command, if the point with the color was found, the BD bit if S#2 will be 1 and the horizontal coordinate of the point will be available in S#8 and S#9.

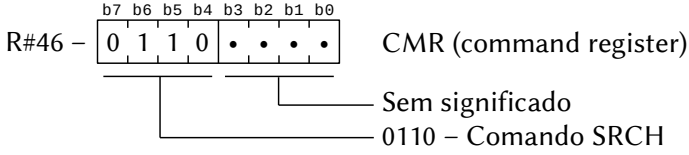


Before executing the SRCH command, the following registers must be loaded:

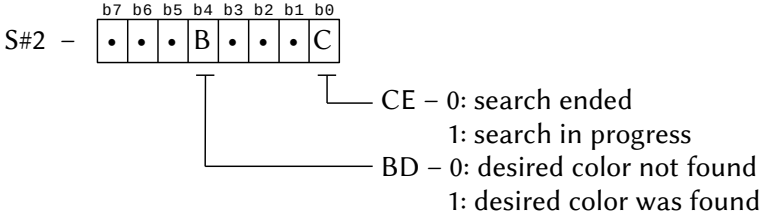




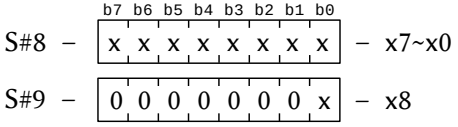
To execute the SRCH command, just write the value 60H (0110B) in register R#46. At the end of execution, if the BD bit if S#2 is 1, the point with the specified color has been found and its horizontal coordinate is stored in S#8 and S#9. If the point was not found, the BD bit of S#2 will be 0.



The search status can be read from S#2:



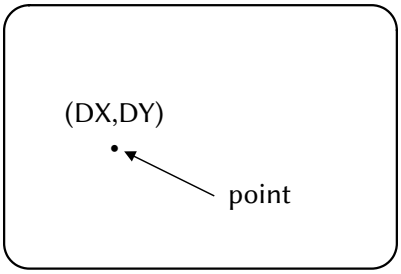
If the point with the selected features is found, its horizontal coordinate can be read in S#8 and S#9:



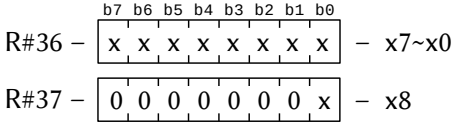
Horizontal coordinate of the point with the specified color, when found

5.5.4.11 – PSET (Draw a point)

Using this command, you can draw a point at any coordinate in the VRAM. Logical operations on the destination are possible.

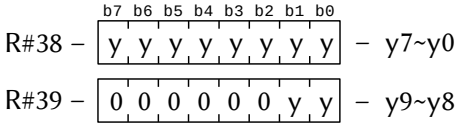


The following registers must be loaded:



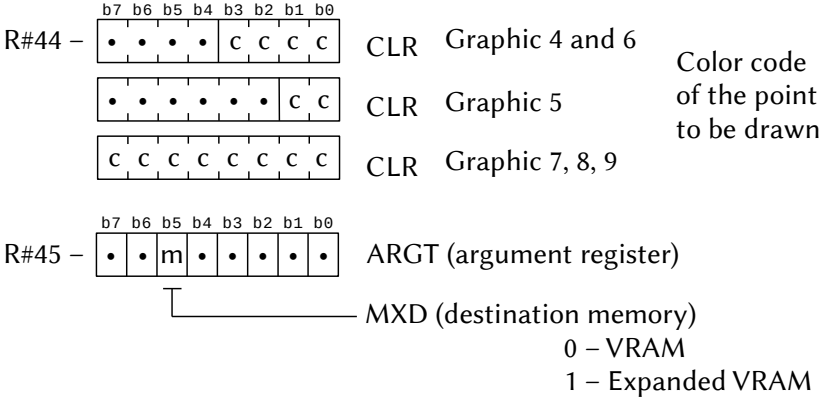
DX

Horizontal coordinate of the point to be drawn (0 to 511)

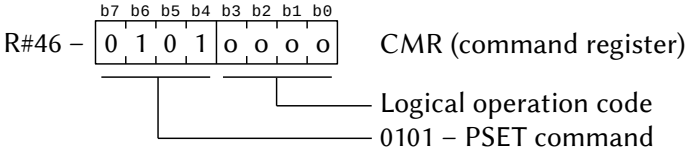


DY

Vertical coordinate of the point to be drawn (0 to 1023)

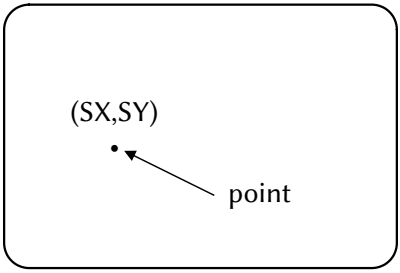


To execute the PSET command, simply write the value 0101B in the highest four bits of R#46 and the logic opcode in the lowest four bits.

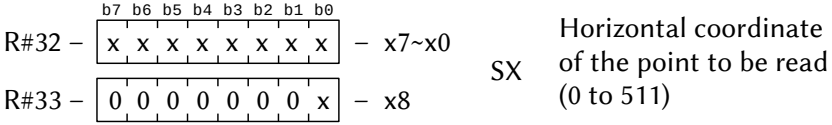


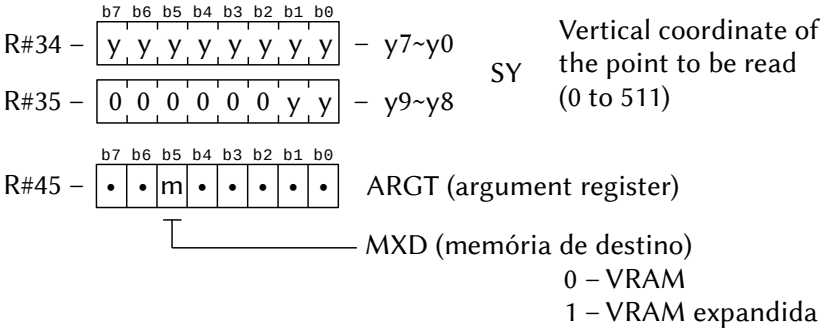
5.5.4.12 – POINT (Get color code of a point)

The POINT command reads a color code of a point at any VRAM coordinate.

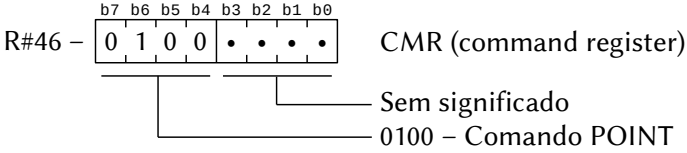


The following registers must be loaded:

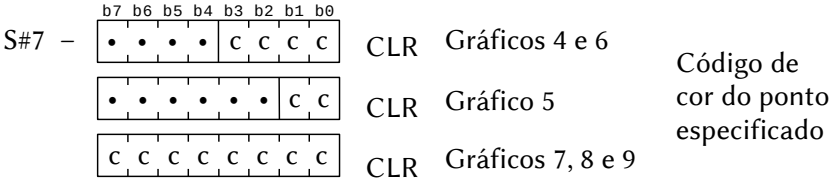




To execute the POINT command, just write 40H (0100B) in R#46:



The specified point color can be read in S#7:

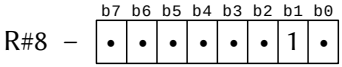


5.5 – MAKING COMMANDS FASTER

The VDP framework allows many other tasks to be performed during the execution of a command. Sometimes execution of some of these commands is slow due to this. If these functions are disabled, command execution will be faster.

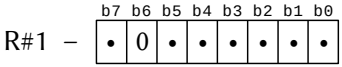
• Display inhibition of sprites

This means is very practical and allows a noticeable increase in speed when the sprites are removed from the screen. To do this, just set bit 1 of R#8 to 1.



• Screen presentation inhibition

This medium has the drawback that, when inhibited, the entire screen takes on the border color. To inhibit the display of the screen, just reset bit 6 of R#1, setting it to zero.

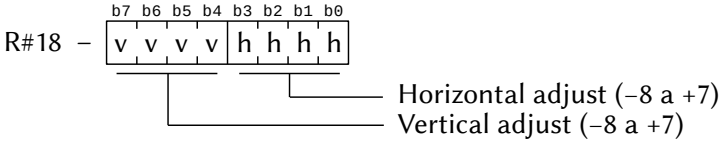


5.6 – MISCELLANEOUS VDP FUNCTIONS (V9938/58)

This section describes several additional VDP functions.

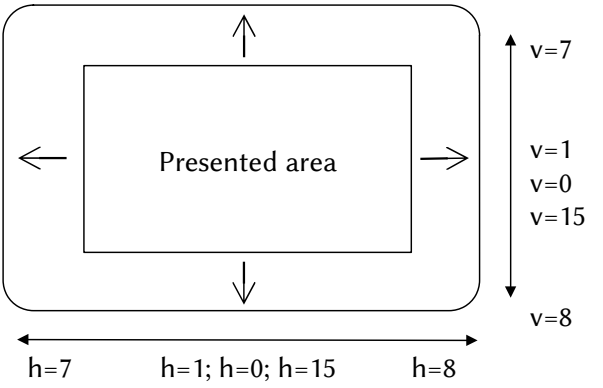
5.6.1 – Screen location adjustment

Register R#18 is used to adjust the screen location. Corresponds to BASIC's SET ADJUST statement.



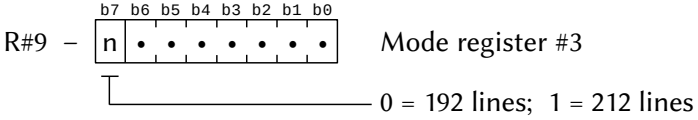
The value is specified in 2's complement according the following values:

0111	→ 7	→ +7	1111	→ 15	→ –1
0110	→ 6	→ +6	1110	→ 14	→ –2
⋮	⋮	⋮	⋮	⋮	⋮
0001	→ 1	→ +1	1001	→ 9	→ –7
0000	→ 0	→ 0	1000	→ 8	→ –8



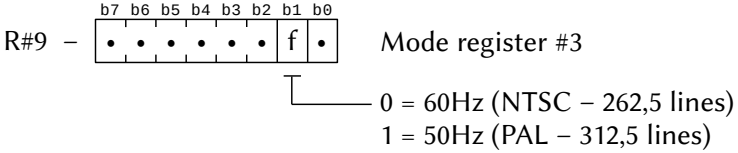
5.6.2 – Number of points in the vertical direction

The number of points in the vertical direction can be chosen between 192 or 212, through bit 7 of R#9. This function is only valid for text mode 2 and graphics modes 4 to 9. The 212 line mode for text mode 2 is not supported by BASIC.



5.6.3 – Interruption frequency (PAL/NTSC)

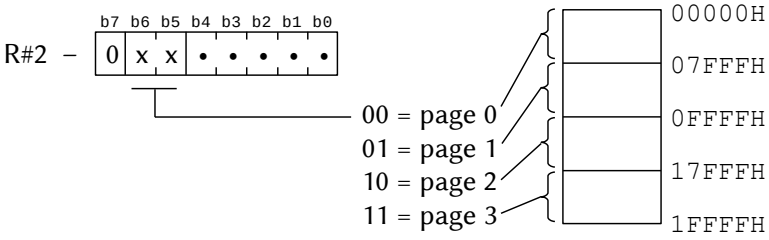
The interrupt frequency in MSX is controlled by the VDP and can be 50Hz or 60Hz. The 60Hz frequency is used in the Japanese NTSC system and the Brazilian PAL-M system. The 50Hz is used for the European PAL-N system.



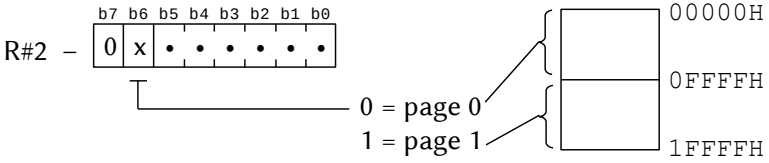
5.6.4 – Changing video pages

In graphics modes 4 to 9, the pages being displayed can be switched by modifying the start address of the pattern table.

Graphics modes 4 and 5:

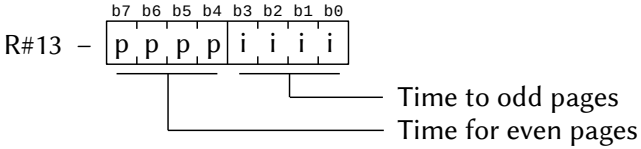


Graphic modes 6, 7, 8 and 9:



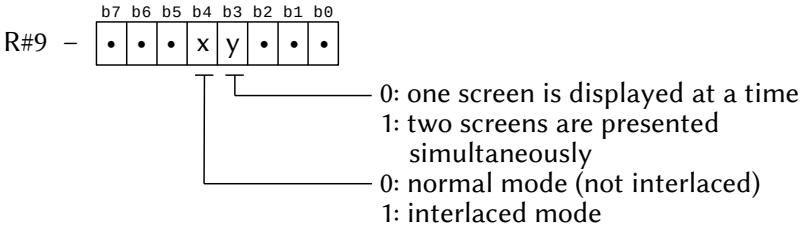
5.6.5 – Automatic screen switching

In graphics modes 4 to 9, two video pages can be displayed alternately. Pages 0 and 1 or 2 and 3 can use this feature. To start automatic screen switching, the odd page must be selected (1 or 3); then just adjust the change time in R#13. The highest four bits define the time for the even page and the lowest four bits for the odd page. The time period is counted in units of 1/6 of a second. If the period value is 0, blinking is disabled; in this case, any page can be selected for display. If it is "0001B" it will be 166 ms for 60 Hz (NTSC) or 200 ms for 50 Hz (PAL); if it is "1111B", it will be 2503 ms (2.5 seconds) for 60 Hz and 3000 ms (3 seconds) for 50 Hz.

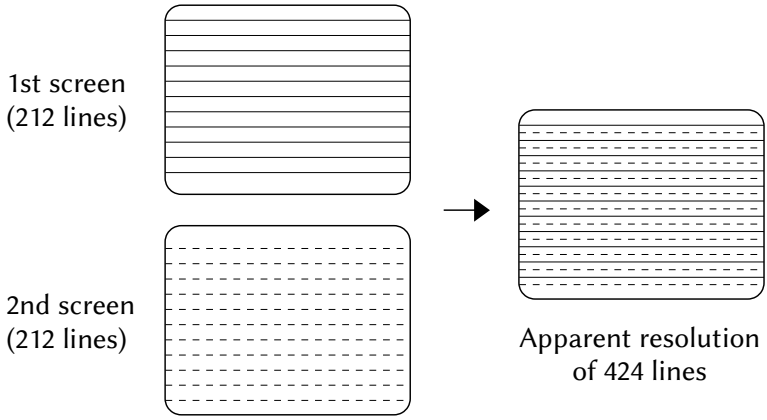


5.6.6 – Interlaced mode

Interlaced mode can be used to have an apparent vertical resolution of 424 lines. This is done by switching two pages of video at high speed and showing only half the height of each row on those pages. The two pages are switched 60 times per second, which can cause flickering. This mode is selected by R#9.

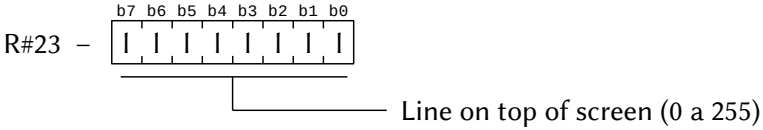


Interlaced mode works as illustrated below.



5.6.7 – Vertical scroll

Register R#23 is used to indicate the start line to the screen. By changing the value of this register, you can make a very smooth vertical scroll. As the scroll is made for 256 lines, the sprite table can appear and be moved to another page.

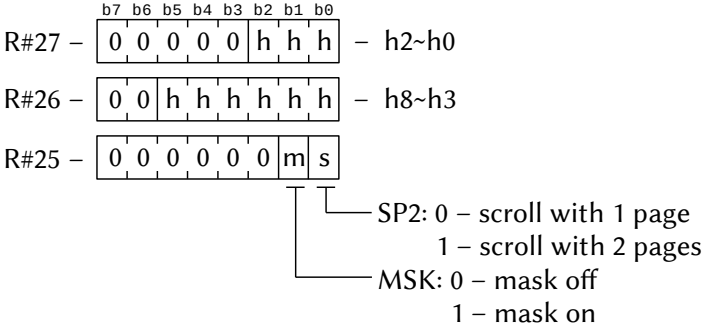


5.6.8 – Horizontal scroll (V9958 only)

Horizontal scroll is supported by MSX2+ or higher. It is done through registers R#26 and R#27, always considering that the screen has 256 horizontal points, even in graphics modes 5 and 6.

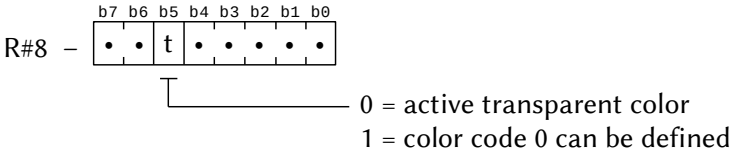
Register R#26 can range from 0 to 31 (scroll with one page of video) or from 0 to 63 (scroll with two pages of video). Each increment corresponds to the displacement of 8 points on the screen (16 in graphics modes 5 and 6). R#27, on the other hand, can vary from 7 to 0, with each decrement corresponding to the displacement of a point on the screen (two for graphic modes 5 and 6). It is important to note that when one is incremented, the other must be decremented.

Bit 0 of R#25 determines whether to scroll with two pages of video. If it is 0, the scroll will be done with only one page; if it is 1, it will be done with two consecutive pages, and the page being displayed must be odd. Bit 1 of R#25 determines the binding of a mask covering the left 8 columns of the screen. If it is 0, the mask is off; if it is 1, it will be on. The mask color is the same as the border color.



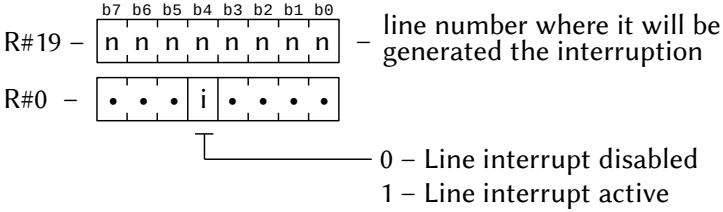
5.6.9 – Color code 0

Of the 16 colors in the palette, color 0 is transparent, that is, a color cannot be defined for it and any object drawn with it will not be seen. However, by setting bit 5 of R#8, the transparent function will be disabled and the color 0 can be set by P#0.



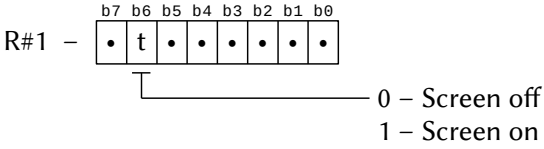
5.6.10 – Line scan interruption

In MSX-VIDEO, an interrupt can be generated when it finishes scanning a specific line of the screen. To do this, just put in R#19 the line number that should generate the interrupt and set bit 4 of R#0 to 1.



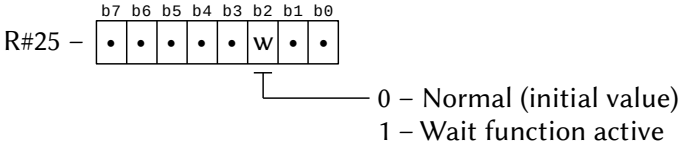
5.6.11 – Turn screen on/off

The on/off screen display function is controlled by bit 6 of R#1. When off, the entire screen is the color specified by the lowest four bits of R#7 (8 bits in graphics mode 7). VDP hardware commands are faster when the screen is off.



5.6.12 – Wait command (V9958 only)

This function can be used to speed up data transfer from CPU to VRAM. When active, all VDP access ports are placed in standby mode until VRAM access is complete. This function is controlled by bit 2 of R#25.

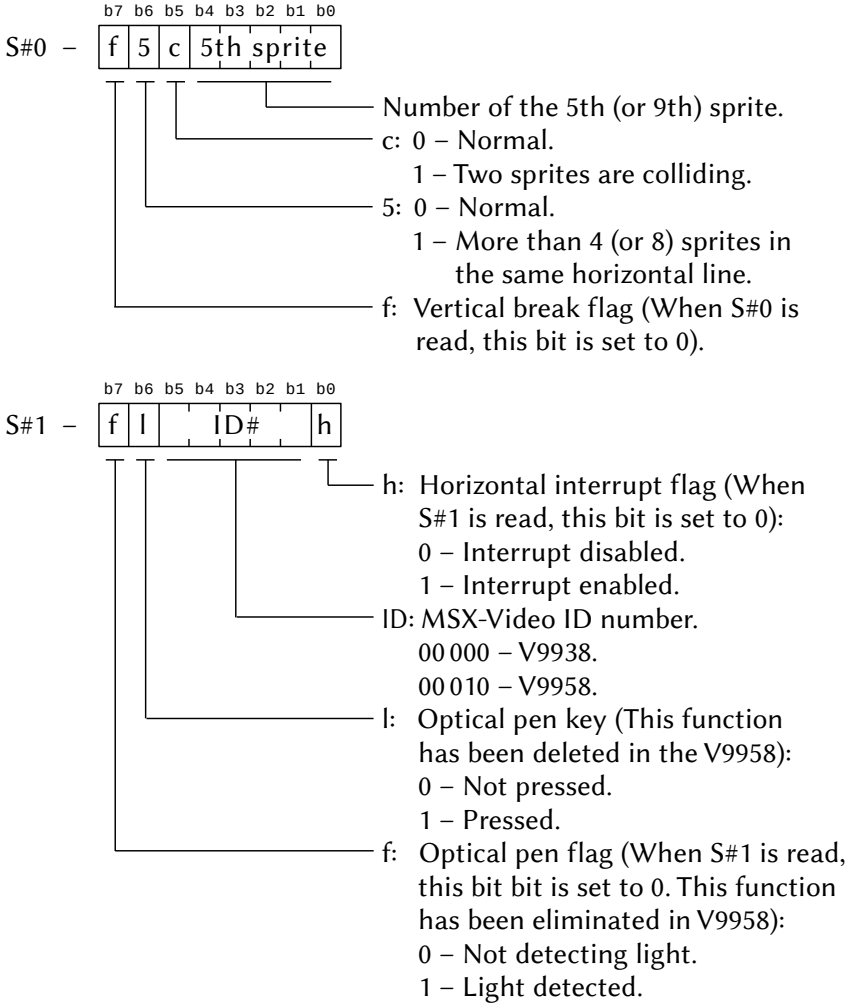


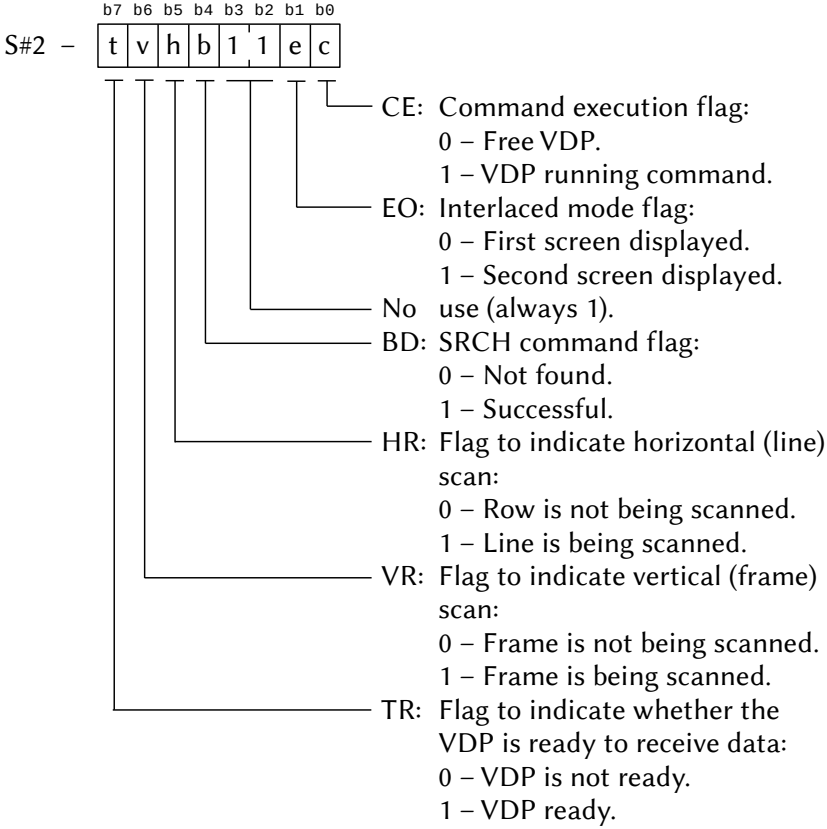
5.7 – REGISTERS DESCRIPTION (V9938/58)

This section briefly describes all registers of VDP's TMS9918, V9938 and V9958.

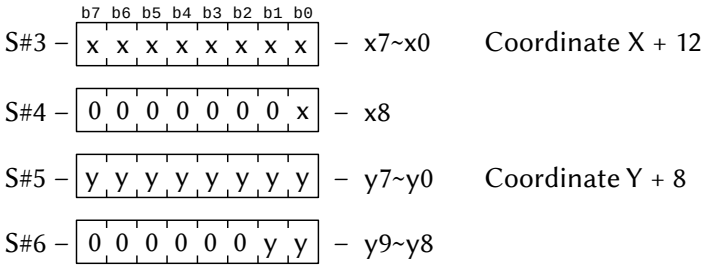
5.7.1 – Status registers

Status registers are numbered S#0 through S#9. They are read-only registers and their functions are illustrated below. Only the S#0 register is present in the TMS9918.

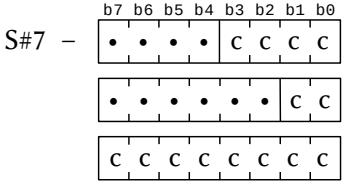




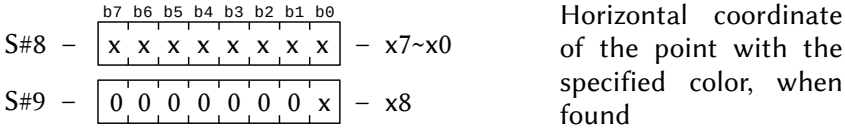
Registers S#3 to S#6 contain the location of the collision between sprites. For the V9938 they will also indicate the location of the stylus and the relative movement of the mouse (these functions were eliminated in the V9958).



Register S#7 contains the color code for the POINT and LMCM commands.

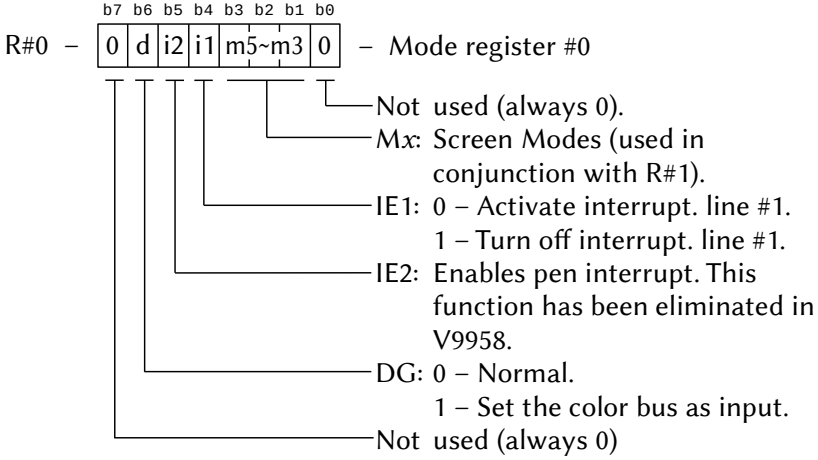


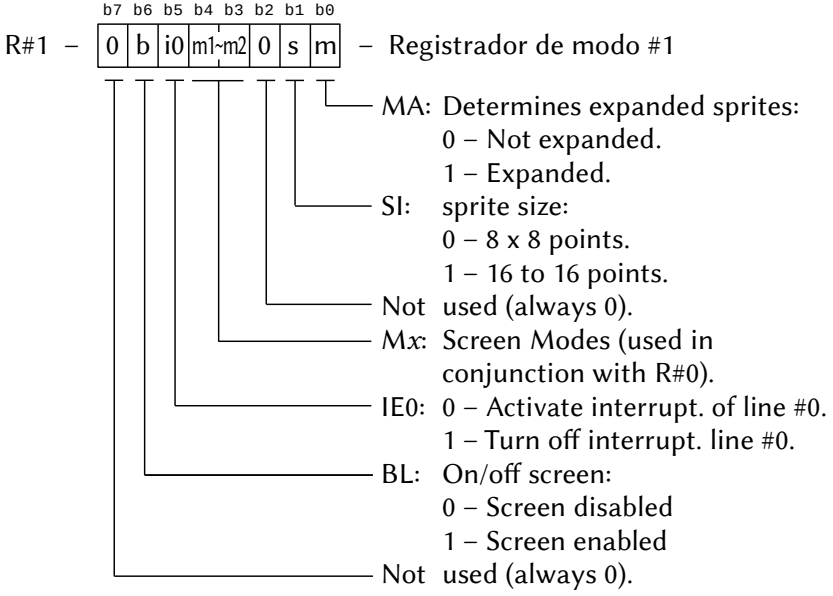
Registers S#8 and S#9 contain the horizontal position for the SRCH command.



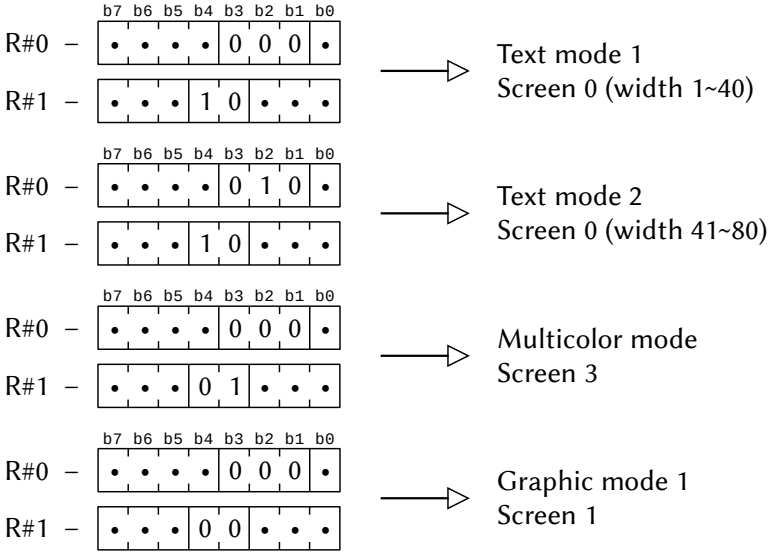
5.7.2 – Mode registers

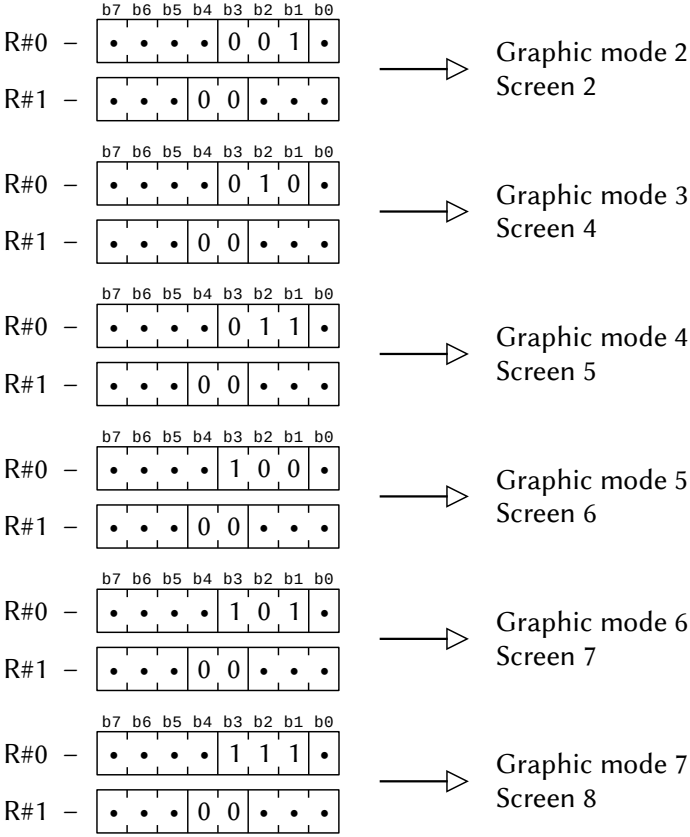
The mode registers are designated by R#0, R#1, R#8, R#9 and R#25, where R#8 and R#9 are unique to V9938/58 and R#25 is unique to V9958. They are write-only registers and control the operation modes of the VDP.



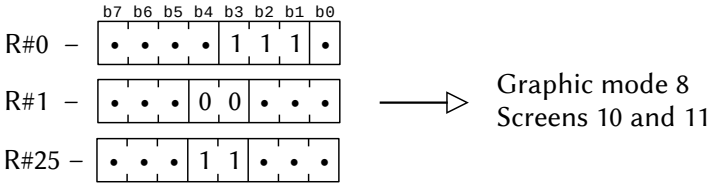


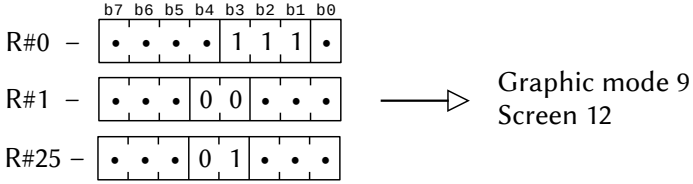
Screens are selected by bits m1~m5 of R#0 and R#1 according to the following illustrations:

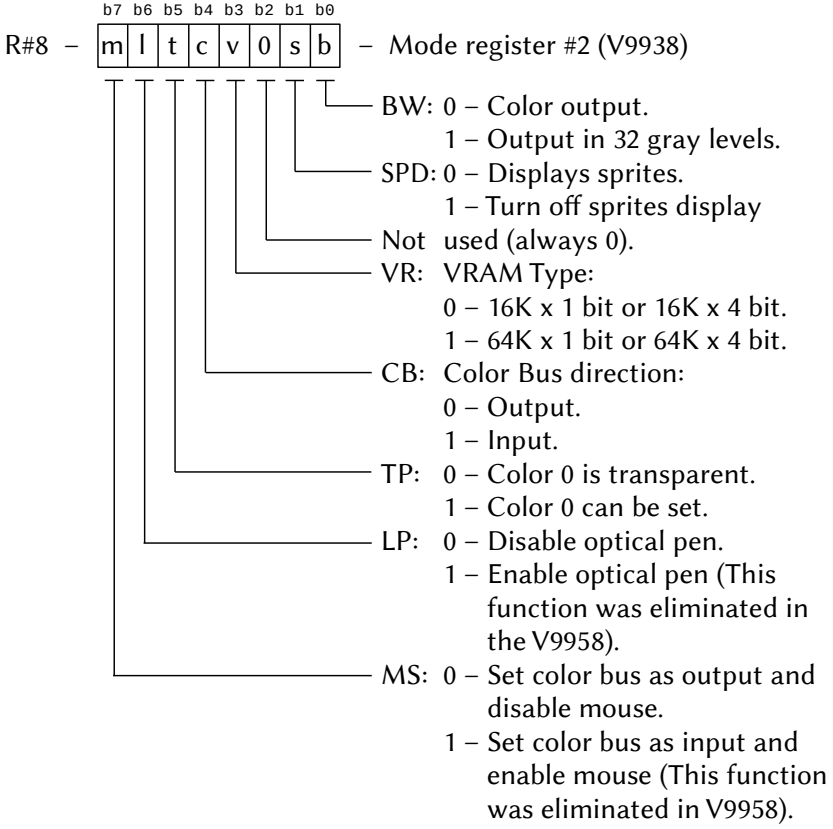


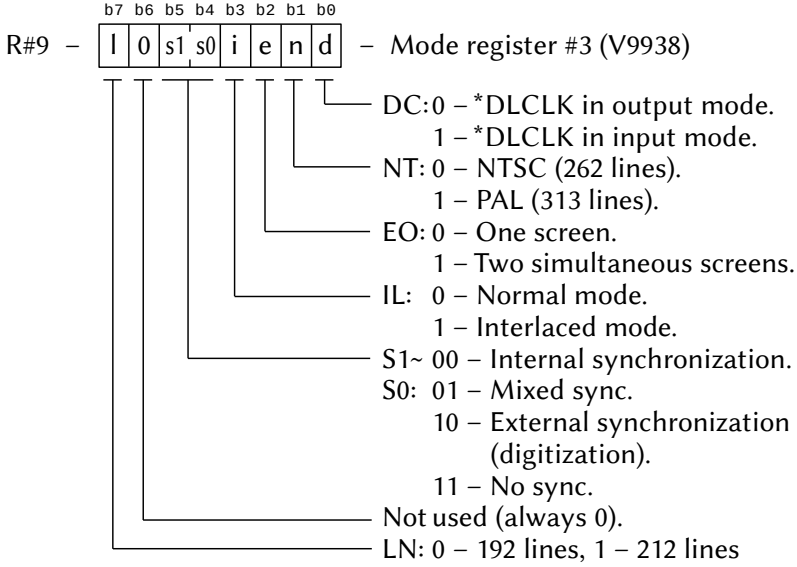


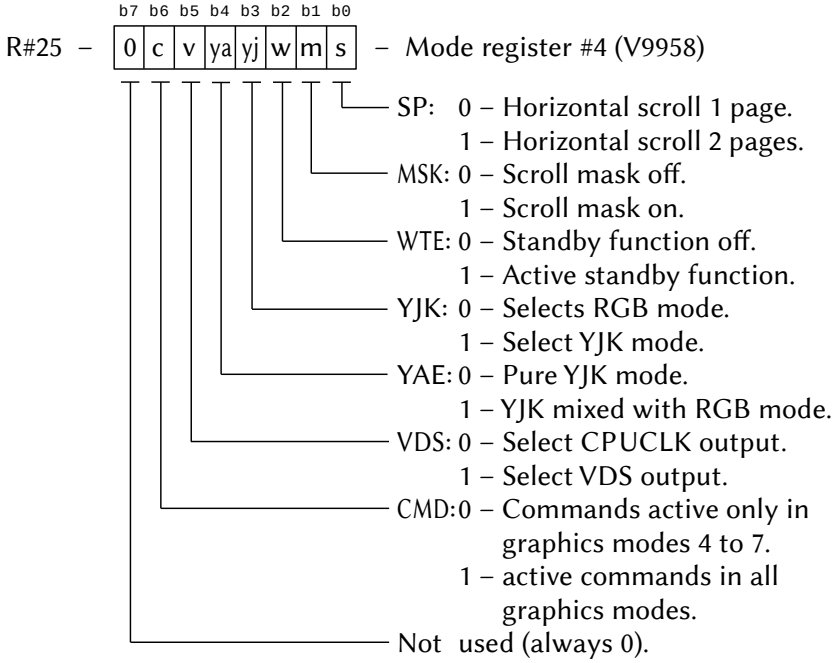
V9958's unique modes are selected together with R#25, which is mode register #4. For modes 1 to 7 the two bits of R#25 must be set to zero.




b7 b6 b5 b4 b3 b2 b1 b0
R#0 – • • • • 1 1 1 •
R#1 – • • • 0 0 • • •
R#25 – • • • 0 1 • • •
Graphic mode 9
Screen 12

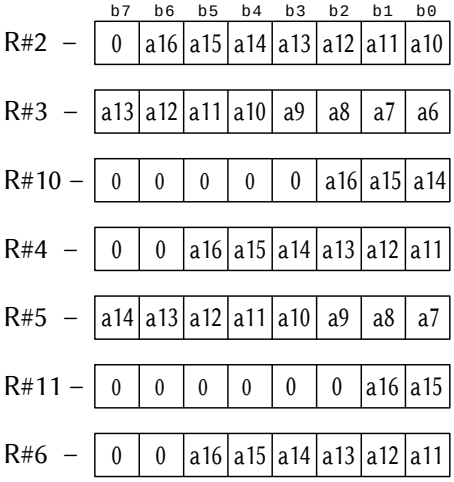

b7 b6 b5 b4 b3 b2 b1 b0
R#8 – m l t c v 0 s b – Mode register #2 (V9938)
BW: 0 – Color output.
1 – Output in 32 gray levels.
SPD: 0 – Displays sprites.
1 – Turn off sprites display
Not used (always 0).
VR: VRAM Type:
0 – 16K x 1 bit or 16K x 4 bit.
1 – 64K x 1 bit or 64K x 4 bit.
CB: Color Bus direction:
0 – Output.
1 – Input.
TP: 0 – Color 0 is transparent.
1 – Color 0 can be set.
LP: 0 – Disable optical pen.
1 – Enable optical pen (This function was eliminated in the V9958).
MS: 0 – Set color bus as output and disable mouse.
1 – Set color bus as input and enable mouse (This function was eliminated in V9958).


b7 b6 b5 b4 b3 b2 b1 b0
R#9 – l 0 s1 s0 i e n d – Mode register #3 (V9938)
DC: 0 – *DLCLK in output mode.
1 – *DLCLK in input mode.
NT: 0 – NTSC (262 lines).
1 – PAL (313 lines).
EO: 0 – One screen.
1 – Two simultaneous screens.
IL: 0 – Normal mode.
1 – Interlaced mode.
S1~ 00 – Internal synchronization.
S0: 01 – Mixed sync.
10 – External synchronization (digitization).
11 – No sync.
Not used (always 0).
LN: 0 – 192 lines, 1 – 212 lines


b7 b6 b5 b4 b3 b2 b1 b0
R#25 – 0 c v ya yj w m s – Mode register #4 (V9958)
SP: 0 – Horizontal scroll 1 page.
1 – Horizontal scroll 2 pages.
MSK: 0 – Scroll mask off.
1 – Scroll mask on.
WTE: 0 – Standby function off.
1 – Active standby function.
YJK: 0 – Selects RGB mode.
1 – Select YJK mode.
YAE: 0 – Pure YJK mode.
1 – YJK mixed with RGB mode.
VDS: 0 – Select CPUCLK output.
1 – Select VDS output.
CMD: 0 – Commands active only in graphics modes 4 to 7.
1 – active commands in all graphics modes.
Not used (always 0).

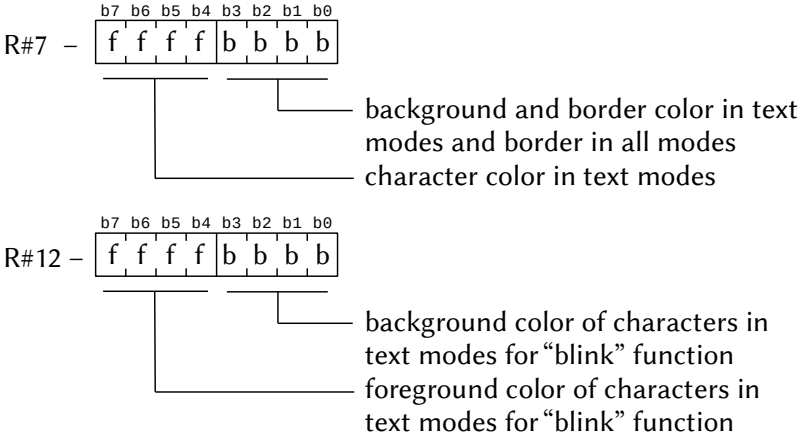
5.7.3 – Tables adresses registers

They are registers that store the tables adresses in the VRAM. The values of these registers depend on the screen in use.

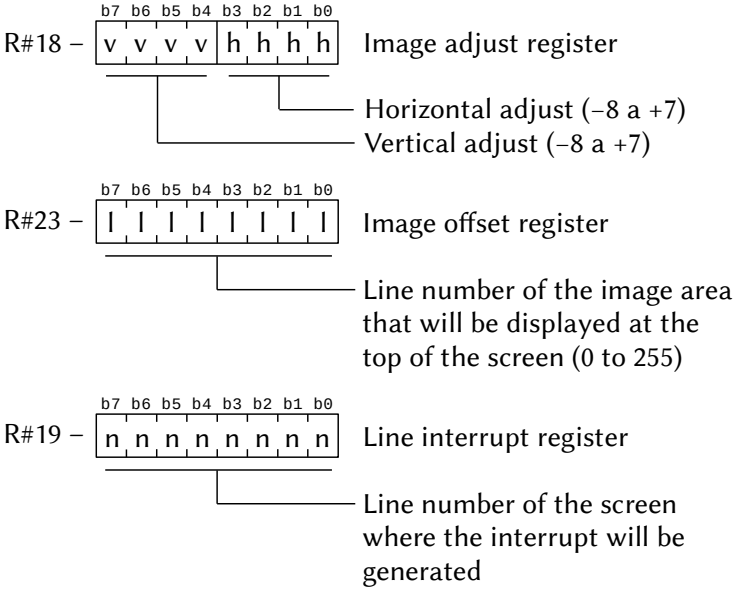


5.7.4 – Color and display registers

They are used to adjust the colors of the screen and other characteristics.

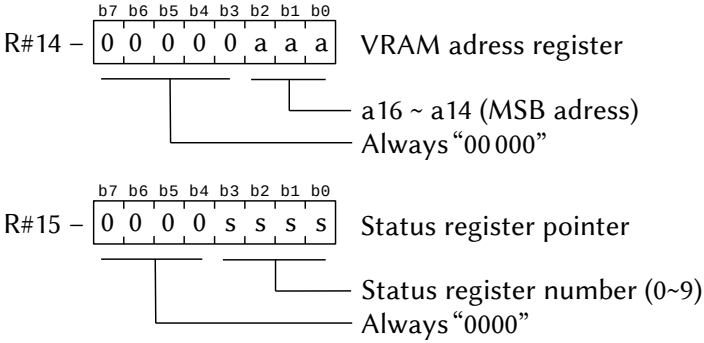


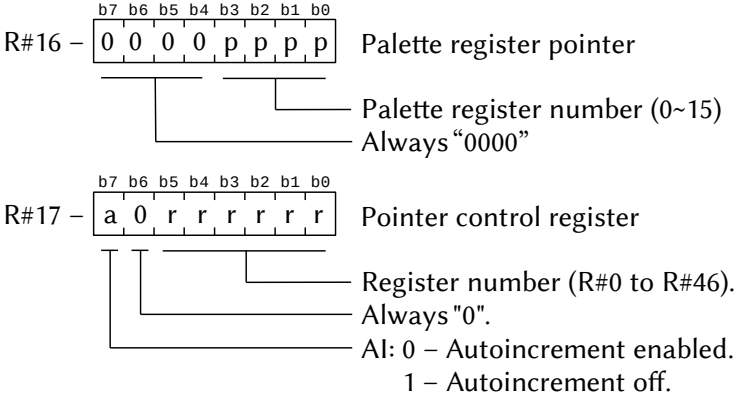
The above values define the color subcarrier signal for the composite video output of the V9938. This output is not present on the V9958. Also, in most MSX2's, the composite video signal is generated by external circuits, so these recorders have become useless.



5.7.5 – Access registers

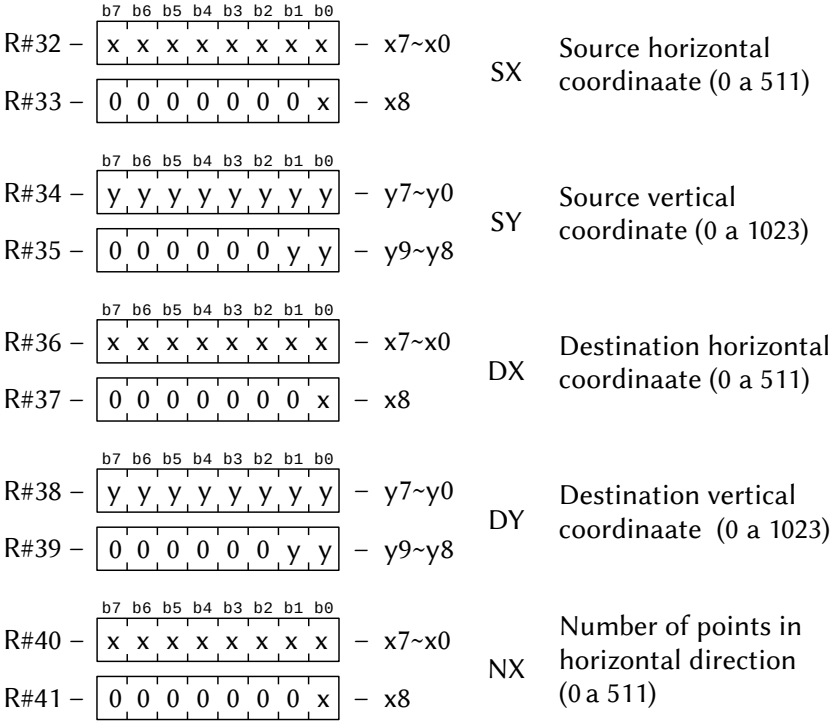
They are used to access VRAM or other VDP registers.

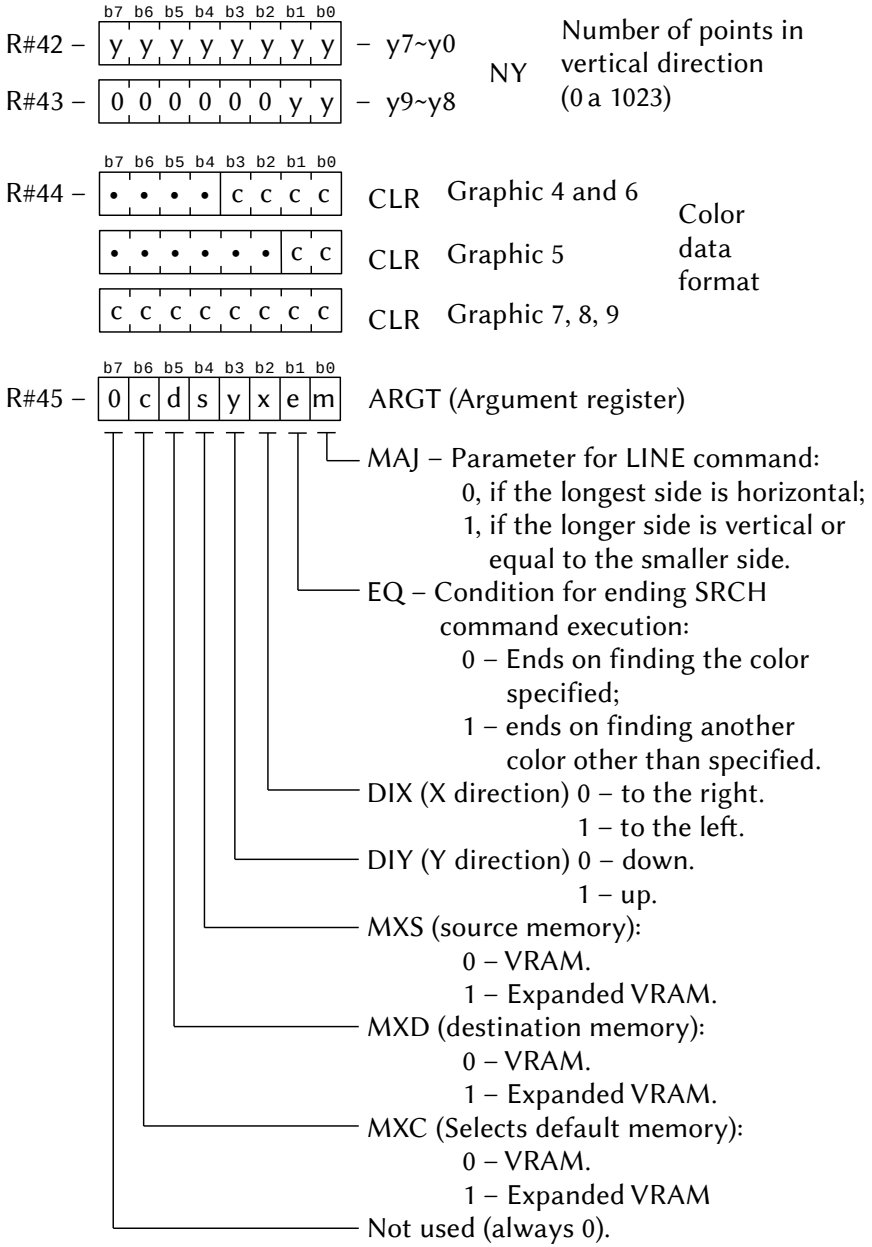


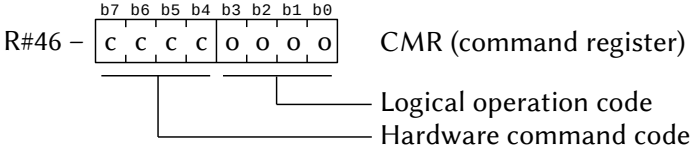


5.7.6 – Command registers

They are used to control VDP hardware commands.




b7 b6 b5 b4 b3 b2 b1 b0
R#42 – y y y y y y y y – y7~y0
R#43 – 0 0 0 0 0 0 y y – y9~y8
NY
Number of points in vertical direction (0 a 1023)
R#44 – • • • • c c c c CLR Graphic 4 and 6
• • • • • • c c CLR Graphic 5
c c c c c c c c CLR Graphic 7, 8, 9
Color data format
R#45 – 0 c d s y x e m ARGT (Argument register)
MAJ – Parameter for LINE command:
0, if the longest side is horizontal;
1, if the longer side is vertical or equal to the smaller side.
EQ – Condition for ending SRCH command execution:
0 – Ends on finding the color specified;
1 – ends on finding another color other than specified.
DIX (X direction) 0 – to the right.
1 – to the left.
DIY (Y direction) 0 – down.
1 – up.
MXS (source memory):
0 – VRAM.
1 – Expanded VRAM.
MXD (destination memory):
0 – VRAM.
1 – Expanded VRAM.
MXC (Selects default memory):
0 – VRAM.
1 – Expanded VRAM
Not used (always 0).



5.8 – THE V9990 (E-VDP-III)

The VDP V9990 was released in 1992 but has not been used on any MSX models. It has some screens similar to V9958, but it cannot be considered compatible. Some time later, a cartridge was released that used it (the GFX9000) but also did not have all the screen modes of the V9958. More recently, the PowerGraph cartridge was launched along the same lines. The table below illustrates the "semi-compatibility" of the V9990's display modes with the V9958's modes.

Screen modes		V9958		V9990	
		Modes	Sprites	Modes	Sprites
Screen 0 (40)	Text 1	Yes	Não	No	No
Screen 0 (80)	Text 2	Yes	Não	No	No
Screen 3	Multicolor	Yes	256	No	No
Screen 1	Graphic 1	Yes	256	No	No
Screen 2	Graphic 2	Yes	256	No	No
Screen 4	Graphic 3	Yes	256	No	No
Screen 5	Graphic 4	Yes	256	Yes	2
Screen 6	Graphic 5	Yes	256	Yes	2
Screen 7	Graphic 6	Yes	256	Yes	2
Screen 8	Graphic 7	Yes	256	Yes	2
Screen 10/11	Graphic 8	Yes	256	Yes	2
Screen 12	Graphic 9	Yes	256	Yes	2

Although it doesn't have all the screen modes of the V9958, the V9990 has a few more, in addition to being much faster.

5.8.1 – The V9990 registers

The V9990 has 52 8-bit registers to control its operations, numbered R#0 to R#28 and R#32 to R#54. The first subgroup controls screen operations and the second subgroup controls the VDP hardware commands. A brief description of all registers follows.

R#0	(W)	VRAM write address (A7 ~ A0)
R#1	(W)	VRAM write address (A15 ~ A8)
R#2	(W)	VRAM write address (A18 ~ A16)
R#3	(W)	VRAM read address (A7 ~ A0)
R#4	(W)	VRAM read address (A15 ~ A8)
R#5	(W)	VRAM read address (A18 ~ A16)
R#6	(R/W)	Screen mode #0
R#7	(R/W)	Screen mode #1
R#8	(R/W)	Control Register
R#9	(R/W)	Interrupt register #0
R#10	(R/W)	Interrupt register #1
R#11	(R/W)	Interrupt register #2
R#12	(R/W)	Interrupt register #3
R#13	(W)	Palette Control
R#14	(W)	Palette Pointer
R#15	(R/W)	Back Drop Color (background color)
R#16	(R/W)	Screen Adjustment
R#17	(R/W)	First screen vertical scroll (Y7 ~ Y0)
R#18	(R/W)	First screen vertical scroll (Y12 ~ Y8)
R#19	(R/W)	First screen horizontal scroll (X2 ~ X0)
R#20	(R/W)	First screen horizontal scroll (X10 – X3)
R#21	(R/W)	Second screen vertical scroll (Y7 ~ Y0)
R#22	(R/W)	Second screen vertical scroll (Y8)
R#23	(R/W)	Second screen horizontal scroll (X2 ~ X0)
R#24	(R/W)	Second screen horizontal scroll (X8 ~ X3)
R#25	(R/W)	Address from sprite pattern table
R#26	(R/W)	Control for LCD
R#27	(R/W)	Priority Control
R#28	(W)	Sprite palette control

R#29 to R#31 does not exist

R#32	(W)	Horizontal coordinate / Initial address (7 ~ 0)
R#33	(W)	Horizontal coordinate / Initial address (10 ~ 8)
R#34	(W)	Vertical coordinate / Initial address (7 ~ 0)

R#35	(W)	Vertical Coordinate / Initial Address (11 ~ 8)
R#36	(W)	Horizontal coordinate / Final address (7 ~ 0)
R#37	(W)	Horizontal coordinate / Final address (10 ~ 8)
R#38	(W)	Vertical Coordinate / Final Address (7 ~ 0)
R#39	(W)	Vertical Coordinate / Final Address (11 ~ 8)
R#40	(W)	Horizontal Transfer Counter (7 ~ 0)
R#41	(W)	Horizontal Transfer Counter (11 ~ 8)
R#42	(W)	Vertical Transfer Counter (7 ~ 0)
R#43	(W)	Vertical Transfer Counter (11~8)
R#44	(W)	Argument Register
R#45	(W)	Logic Operation Register
R#46	(W)	Writing Mask (7 ~ 0)
R#47	(W)	Writing Mask (15~8)
R#48	(W)	Front color (7 ~ 0)
R#49	(W)	Front color (15~8)
R#50	(W)	Background color (7 ~ 0)
R#51	(W)	Background color (15~8)
R#52	(W)	Command register
R#53	(R)	Horizontal coordinate of the edge (7 ~ 0)
R#54	(R)	Horizontal coordinate of the edge (10 ~ 8)

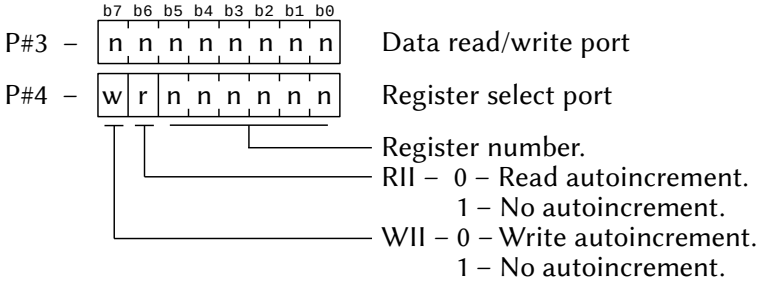
5.8.2 – Access to V9990

Access to the V9990 is directly via the Z80's 12 I/O ports, labeled P#0 to P#B. The function of each is described below.

P#0	60H	(R/W)	Access to VRAM
P#1	61H	(R/W)	Accessing the color palette
P#2	62H	(R/W)	Access to hardware commands
P#3	63H	(R/W)	Access to registers
P#4	64H	(W)	Registrar selection
P#5	65H	(R)	Status port
P#6	66H	(W)	Interrupt flag
P#7	67H	(W)	System control
P#8	68H	(W)	Kanji-ROM address (low) – 1
P#9	69H	(R/W)	Kanji-ROM address (high) and data – 1
P#A	6AH	(W)	Kanji-ROM address (low) – 2
P#B	6BH	(R/W)	Kanji-ROM address (high) and data – 2

5.8.2.1 – Access to registers

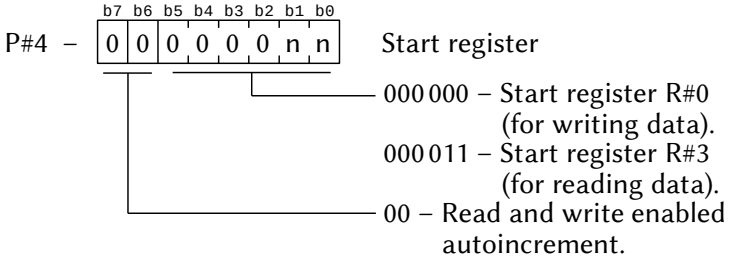
To write a value to a register, it is first necessary to write the register number to port P#4 (64H) and then the data byte to port P#3 (63H). To read the value of a register, just write the number of the register in port P#4 (64H) and then read the respective value in port P#3 (63H).

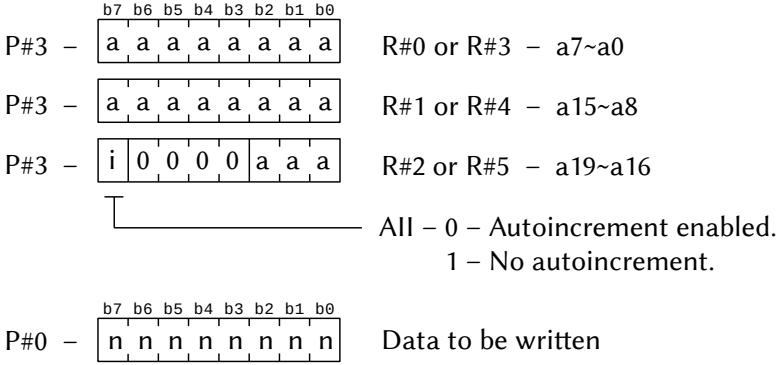


The RII and WII bits enable or disable auto-increment of registers during reading or writing, respectively. If they are 0, the auto-increment function is on and consecutive bytes written or read through port P#3 will cause access to subsequent registers. If they are 1, the bytes will always be sent to the same register, indefinitely. The register number to be accessed must be specified in the lowest 6 bits of port P#4.

5.8.2.2 – Access to VRAM

The V9990 can be connected to 128, 256 or 512 Kbytes of VRAM; so the address bus has 19 bits. To write a byte to VRAM, it is necessary to load registers R#0 to R#2 with the address to be written and write the byte through port P#0 (60H). To read a byte, registers R#3 to R#5 must be loaded with the address to be read and the byte can be obtained by reading port P#0 (60H). The sequence to be followed is as follows:





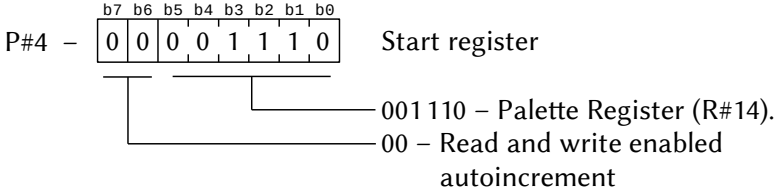
Bit 7 of R#2 or R#5 enables or disables autoincrement when writing or reading from VRAM. If it is 0, when a byte is read or written by port P#0 (60H), the Address will be automatically incremented by 1 and the next access will be in the next address. If this bit is 1, the auto-increment function will be disabled.

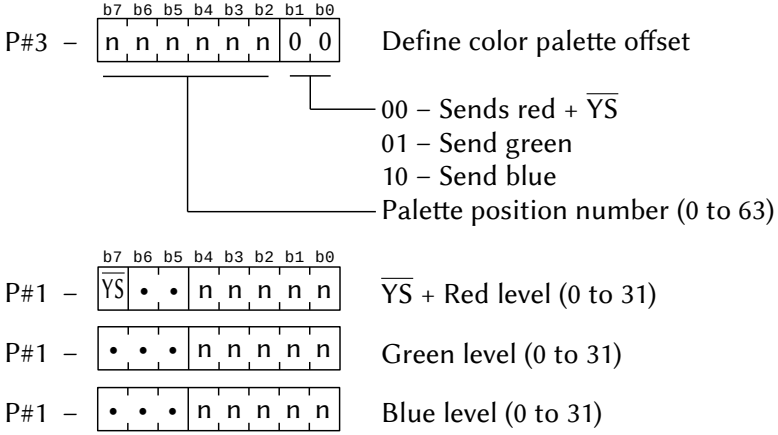
5.8.2.3 – Access to the color palette

To write data to the palette registers, it is necessary to write the palette number in R#14 and the respective values of red, green and blue in port P#1 (61H). Through the palette, up to 64 colors chosen from 32 768 can be defined.

The palette position number must be specified in the highest 6 bits of R#14 (0 to 63). The lower two bits should define which primary color will be sent (0=red, 1=green, 2=blue). These two bits are automatically incremented by 1 each time a byte of data is written to port P#1; therefore, setting them to 0, it is enough to send consecutively the values of red, green and blue, respectively.

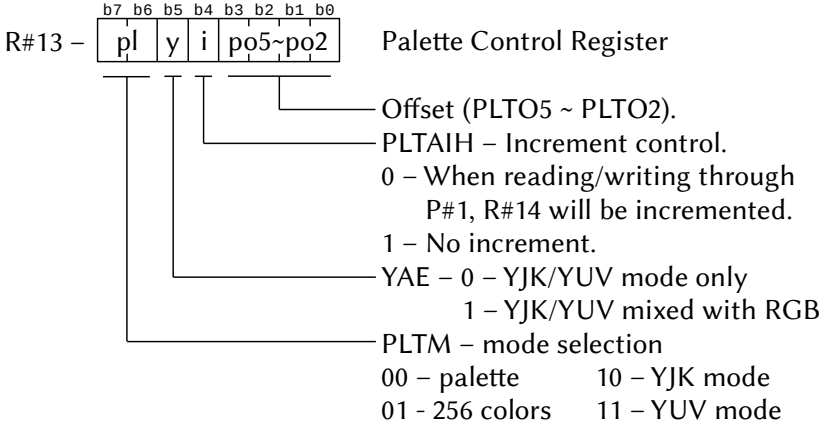
The sequence for entering data in the palette is as follows:





The $\overline{YS}$ bit determines transparent color, which is a feature used for superimpose. If this bit is 1, the transparency function is active; if it is 0, the color defined in the palette will be displayed.

Additional palette settings must be specified in the Palette Control Register (R#13).

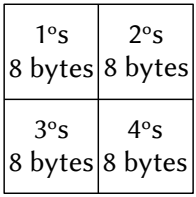


5.8.2.4 – Access to Kanji ROM

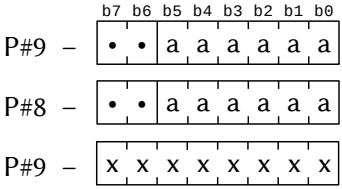
Kanji-ROM contents can be read through ports P#8 (68H) and P#9 (69H) for the primary set (JIS1) and P#A (6AH) and P#B (6BH) for

the secondary set (JIS2). Each set can contain up to 4096 Kanji or any other pattern defined in a 16x16 cell.

To perform the reading, the highest six bits of the JIS code of the respective Kanji must be sent through the P#9/P#B port, followed by the lowest six bits through the P#8/P#A port. Then the P#9/P#B port must be read 32 times to get the 32 bytes that make up the Kanji pattern.



The 32 bytes that make up the pattern are obtained in 4 sequences of 8 bytes, as shown on the side, in order to compose a cell of 16 x 16 points.



5.8.3 – V9990 screen modes

Strictly speaking, the V9990 has 8 screen modes, but each of them can have different palettes and RGB, YJK or YUV modes, so the number of screens ends up being much higher. With 512K of VRAM, there are 20 screens available (2 of which are "pattern" and 18 bit-map) plus two undocumented modes (which unfold into 6 screens, depending on the color configuration). There are two screen systems: Pattern Mode (P1 and P2) and Bit Map Mode (B1 to B6).

5.8.3.1 – Modes by pattern presentation

Mode name	P1	P2
Master frequency	21,5 MHz	21,5 MHz
Point frequency	5,4 MHz	10,7 MHz
Horizontal frequency	15,7 KHz (NTSC)	15,7 KHz (NTSC)
Resolution	256 x 212 points	512 x 212 points

Number of patterns		32 x 26,5 patterns	64 x 26,5 patterns
Pattern size		8 x 8 points	8 x 8 points
Número of screens		2 screens	1 screen
Simultaneous colors		15 + one transparent	15 + one transparent
Palettes		4 palettes of 16 colors chosen from 32 768	4 palettes of 16 colors chosen from 32 768
Image area		64 x 64 patterns	128 x 64 patterns
Selectable patterns		16 384 (maximum)	16 384 (maximum)
Pattern generator	128 Kbytes	1535 patterns / screen	3071 patterns
	256 Kbytes	3583 patterns / screen	7167 patterns
	512 Kbytes	7679 patterns / screen	15 359 patterns

5.8.3.2 – Bit-map modes

Mode	Hor Freq	Resolution	Master Freq	Point freq	BPP	Image size (128 K)	Image size (256 K)	Image size (512 K)
B1	15,75 Khz (NTSC)	256 x 212	21,5 MHz	5,4 MHz	16	256x256	256x512 512x256	256x1024 512x512 1024x256
					8	256x512 512x256	256x1024 512x512 1024x256	256x2048 512x1024 1024x512 2048x256
					4	256x1024 512x512 1024x256	256x2048 512x1024 1024x512 2048x256	256x4096 512x2048 1024x1024 2048x512
					2	256x2048 512x1024 1024x512 2048x256	256x4096 512x2048 1024x1024 2048x512	256x8192 512x4096 1024x2048 2048x1024

B2	15,75 Khz (NTSC)	384 x 240/290 (Overscan)	14,3 MHz	7,2 MHz	16	NC	512x256	512x512 1024x256
					8	512x256	512x512 1024x256	512x1024 1024x512 2048x256
					4	512x512 1024x256	512x1024 1024x512 2048x256	512x2048 1024x1024 2048x256
					2	512x1024 1024x512 2048x256	512x2048 1024x1024 2048x512	512x4096 1024x2048 2048x1024
B3	15,75 Khz (NTSC)	512 x 212	21,5 MHz	10,7 MHz	16	NC	512x256	512x512 1024x256
					8	512x256	512x512 1024x256	512x1024 1024x512 2048x256
					4	512x512 1024x256	512x1024 1024x512 2048x256	512x2048 1024x1024 2048x512
					2	512x1024 1024x512 2048x256	512x2048 1024x1024 2048x512	512x4096 1024x2048 2048x1024
B4	15,75 Khz	768 x 240/290	14,3 MHz	14,3 MHz	4	1024x256	1024x512 2018x256	1024x1024 2048x512
					2	1024x512 2018x256	1024x1024 2048x512	1024x2048 2048x1024
B5	25,3 MHz	640 x 400	21,5 MHz	21,5 MHz	4	NC	1024x512	1024x1024 2048x512
					2	1024x512	1024x1024 2048x512	1024x2048 2048x1024
B6	31,5 KHz	640 x 480	25,2 MHz	25,2 MHz	4	NC	1024x512	1024x1024 2048x512
					2	1024x512	1024x1024 2048x512	1024x2048 2048x1024

In B1 through B4 modes, the vertical resolution can be doubled using the interlace feature, as in the V9938/58. Modes B2 and B4 are overscan modes, that is, the displayed image goes beyond the screen area; therefore, these modes are borderless. In addition, there are two possible vertical resolutions, which can also be doubled by the interlace feature: 240 for NTSC (default 525 lines) and 290 for PAL (default 625 lines). Overscan modes are useful for displaying animations or videos.

5.8.3.3 – Undocumented bit-map modes

Mode	Hor Freq	Resolution	Master Freq	Point freq	BPP	Image size (128 K)	Image size (256 K)	Image size (512 K)
*B0	15,75 Khz (NTSC)	192 x 240/290 (Overscan)	14,3 MHz	3,6 MHz	16	256x256	256x512 512x256	256x1024 512x512 1024x256
					8	256x512 512x256	256x1024 512x512 1024x256	256x2048 512x1024 1024x512 2048x256
					4	256x1024 512x512 1024x256	256x2048 512x1024 1024x512 2048x256	256x4096 512x2048 1024x1024 2048x512
					2	256x2048 512x1024 1024x512 2048x256	256x4096 512x2048 1024x1024 2048x512	256x8192 512x4096 1024x2048 2048x1024
*B7	15,75 KHz	1024 x 212	21,5 MHz	21,5 MHz	4	NC	1024x512	1024x1024 2048x512
					2	1024x512	1024x1024 2048x512	1024x2048 2048x1024

The above modes were not documented in the "V9990 Application Manual". The two also allow you to double the vertical resolution using the interlace feature.

5.8.3.4 – Color systems

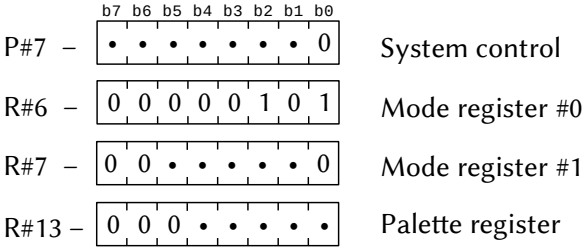
Bits per point	b7/b6 de R#13 (PLTM)	Codification	Number of colors
16 bits / point	0 0	RGB YS=1-bit; G=5-bit; R=5-bit; B=5-bit	32.768 simultaneous
8 bits / point	0 0	Palette	64 colors from 32.768
	01	RGB G=3-bit; R=3-bit; B=2-bit	256 simultaneous
	1 0	YJK	19.268 simultaneous
	1 1	YUV	19.268 simultaneous
4 bits / point	0 0	Palette	16 colors from 32.768
2 bits / point	0 0	Palette	4 colors from 32.768

5.8.3.5 – Values of the registers for each mode

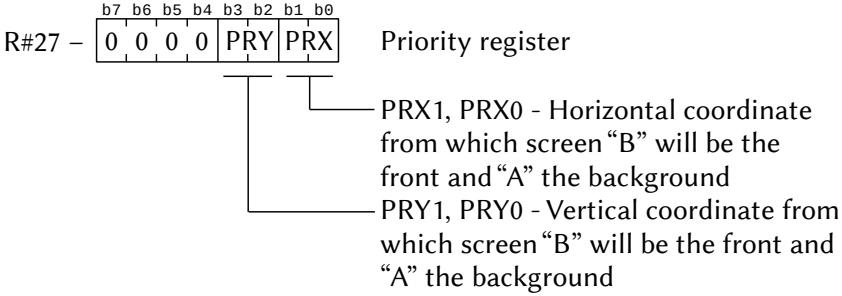
Modo	P#7	R#6				R#7						
	MCS	DSPM	DCKM	XIMM	CLRM	C25M	SM1	SM	PAL	EO	IL	HSCN
P1	0	0	0	1	1	0	0/1	0/1	0/1	0/1	0/1	0
P2	0	1	1	2	1	0	0/1	0/1	0/1	0/1	0/1	0
*B0	1	2	0	0~3	0~3	0	0/1	0/1	0/1	0/1	0/1	0
B1	0	2	0	0~3	0~3	0	0/1	0/1	0/1	0/1	0/1	0
B2	1	2	1	1~3	0~3	0	0/1	0/1	0/1	0/1	0/1	0
B3	0	2	1	1~3	0~3	0	0/1	0/1	0/1	0/1	0/1	0
B4	1	2	2	2~3	0~1	0	0/1	0/1	0/1	0/1	0/1	0
B5	0	2	2	2~3	0~1	0	0	0	0	0	0	1
B6	0	2	2	2~3	0~1	1	0	0	0	0	0	1
*B7	0	2	2	2~3	0~1	0	0/1	0/1	0/1	0/1	0/1	0

*B0 and *B7 modes are not documented in the "V9990 Application Manual".

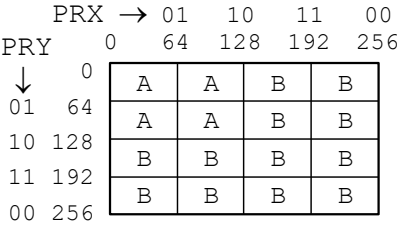
5.8.3.6 – P1 Mode



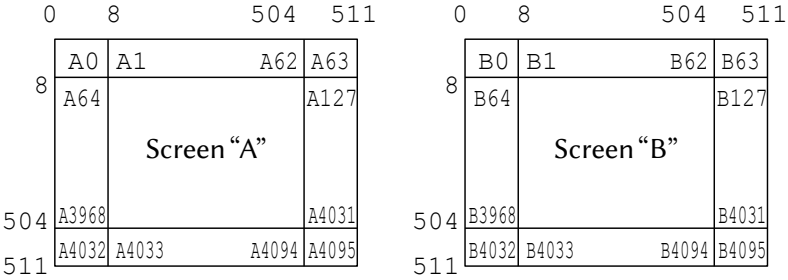
This mode has a resolution of 256 horizontal by 212 vertical points and is pattern-mapped. An interesting feature is that it has 2 independent screens that can be superimposed (A and B). The total image area is actually 512 x 512 points for each screen, but only 256 x 212 are displayed. The upper left point of the screen can be located in the image area by the scroll registers (R#17 to R#20 for the first screen, or "A", and R#21 to R#24 for the second screen, or "B "). There is also a priority register (R#27) for these screens.



When PRX and PRY are 0, screen "A" will be the front in the entire displayed area. By changing these values (00B to 11B), the foreground and background screen areas will be moved in 64 point increments. The example below shows the position of the screens when PRX=10B and PRY=10B.



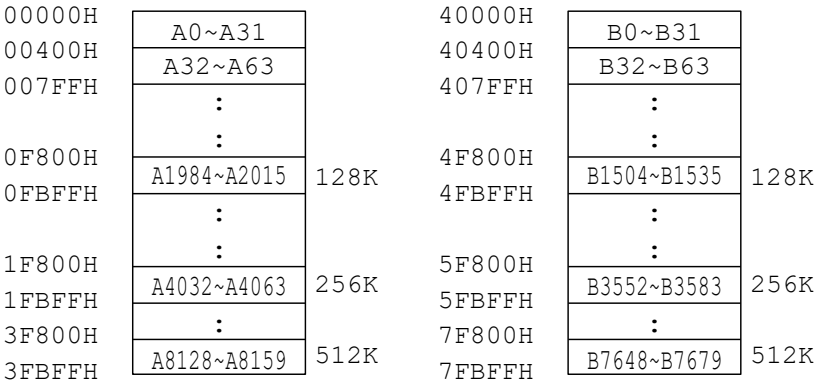
Each image space on screens "A" and "B" consists of 4096 (64 x 64) 8 x 8 point patterns, with 16 colors for each point. For each of these patterns, 2 bytes are reserved in the Name Table to locate them in the image area.



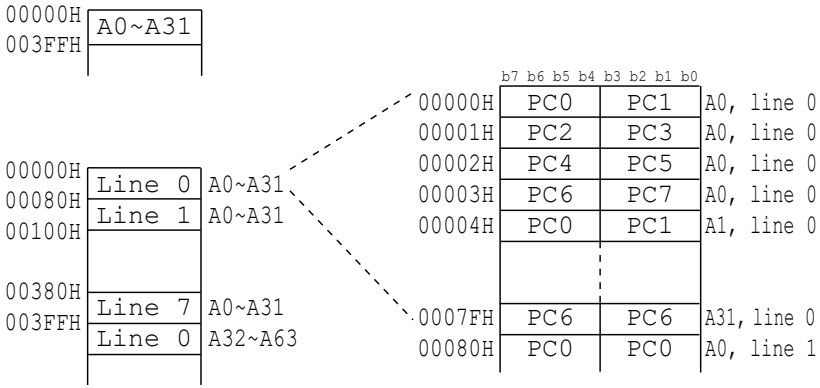
The name table starts at address 7C000H and goes up to address 7DFFFH for screen "A" and starts at 7E000H going up to 7FFFFH for screen "B". Thus, the addresses 7C000H/7C001H correspond to the pattern A0, 7C000H/7C001H to the pattern A1 and so on, in the low/high form. The maximum number of definable patterns depends on the VRAM size, as shown in the table below:

VRAM	"A" screen	"B" screen
128K	2015	1535
256K	4063	3583
512K	8159	7679

The pattern generator table starts at 00 000H and goes up to 3BFFFH for screen "A" and from 40 000H to 7BFFFH for screen "B", as illustrated below.

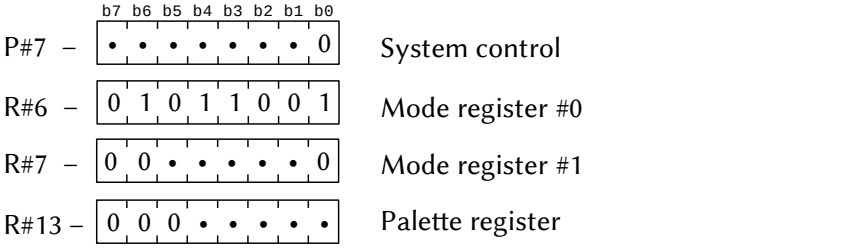


The P1 mode pattern generator table is organized as a bit map, based on 256 horizontal points, reserving 4 bits per point.

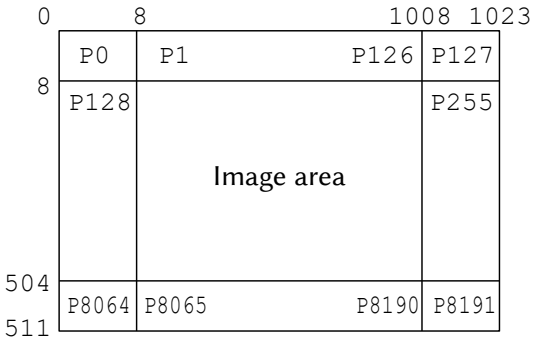


5.8.3.7 – P2 mode

P2 mode is selected by the following registers:



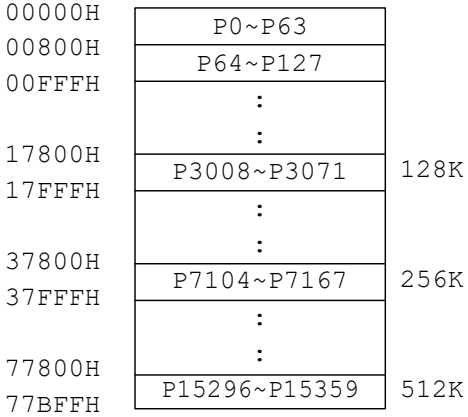
This mode has a resolution of 512 horizontal by 212 vertical points and is pattern-mapped. This mode has only one screen, not two as in P1 mode. The total image area is 1024 x 512 points although only 512 x 212 is displayed. The upper left point of the screen can be located in the image area by the scroll registers (R#17 to R#20). The image space consists of 8192 (128 x 64) 8 x 8 point patterns, with 16 colors for each point. For each of these patterns, 2 bytes are reserved in the name table to locate them in the image area.



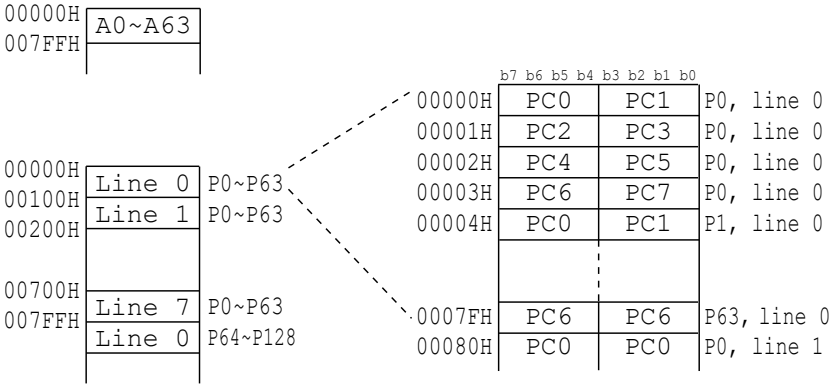
The name table starts at address 7C000H and goes up to address 7FFFFH. Thus, the addresses 7C000H/7C001H correspond to the P0 pattern, 7C000H/7C001H to the P1 pattern and so on, in the low/high form. The maximum number of definable patterns depends on the VRAM size, as shown in the table below:

VRAM	Number of patterns
128K	3071
256K	7167
512K	15 359

The pattern generator table starts at 00 000H and goes up to 77FFFH, as illustrated below.

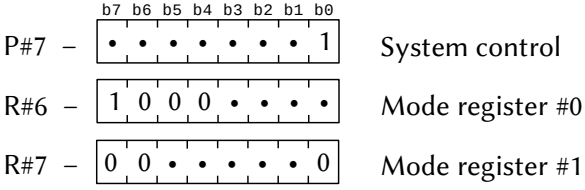


The P2 mode pattern generator table is organized as a bit map, based on 512 horizontal points, reserving 4 bits per point.

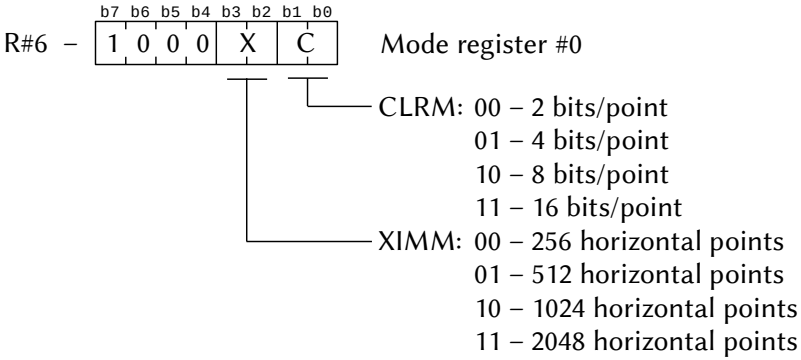


5.8.3.8 – B0 Mode

B0 mode is selected by the following registers:



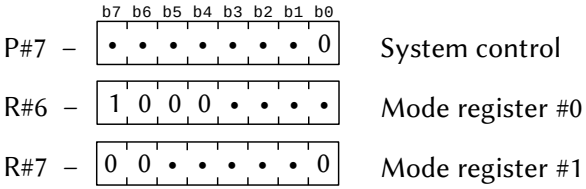
The B0 mode is not officially documented, has a resolution of 192 x 240 points (60Hz) or 192 x 290 points (50Hz) and is an overscan mode. The image area can vary greatly, ranging from 256 to 2048 points horizontally, and from 256 to 8192 points vertically, depending on the size of the VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per point displayed and the horizontal size of the image area are specified in bits b3~b0 of R#6, as illustrated below:



The number of vertical points in the image area is automatically set according to the XIMM value and the amount of available video memory.

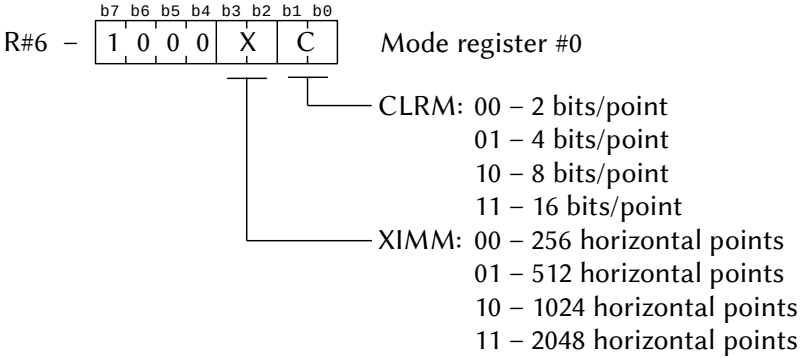
5.8.3.9 – B1 Mode

B1 mode is selected by the following registers:



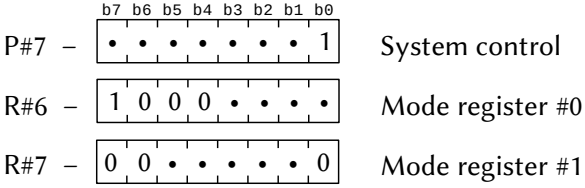
B1 mode has a resolution of 255 x 212 points and is a pure bit map mode. The image area can vary greatly, ranging from 256 to 2048 points horizontally, and from 256 to 8192 points vertically, depending on

the size of the VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points in the image area is automatically set according to the XIMM value and the amount of available video memory.

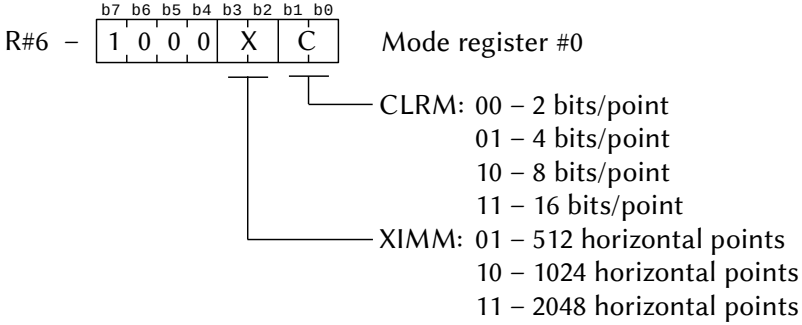


5.8.3.10 – B2 Mode

B2 mode is selected by the following registers:

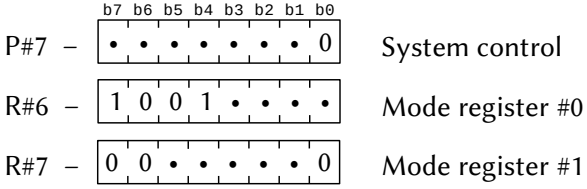


B2 mode has a resolution of 384 x 240 (60 Hz) or 384 x 290 (50 Hz) points (overscan mode) and its image area can vary from 512 to 2048 horizontal points and from 256 to 2048 vertical points depending on the size. of VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.

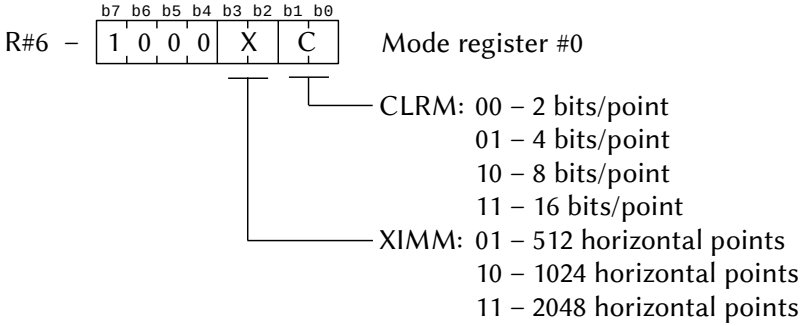


5.8.3.11 – B3 Mode

B3 mode is selected by the following registers:

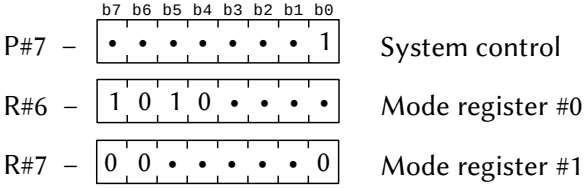


B3 mode has a resolution of 512 x 212 points. Its image area ranges from 512 to 2048 points horizontally, and from 256 to 4096 points vertically, depending on the VRAM size and palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.

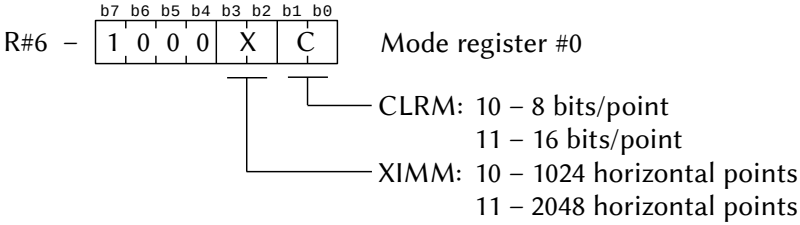


5.8.3.12 – B4 Mode

B4 mode is selected by the following registers:

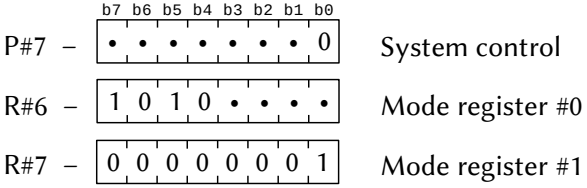


B4 mode has a resolution of 768 x 240 (60 Hz) or 768 x 290 (50 Hz) points (overscan mode) and its image area can vary from 1024 to 2048 horizontal points and from 256 to 2048 vertical points depending on the size. of VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.



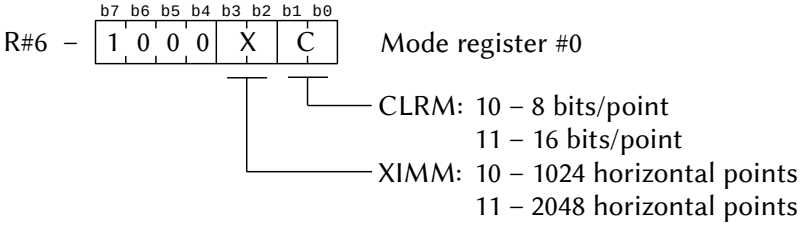
5.8.3.13 – B5 Mode

B5 mode is selected by the following registers:



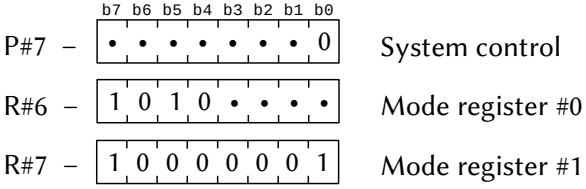
B5 mode is 640 x 400 points and is a high resolution mode, only working on VGA type monitors (fH=24.8 KHz). The image area can vary between 1024 and 2048 horizontal points and 512 and 2048 vertical points. depending on the size of the VRAM and the type of palette used. The-

se sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.

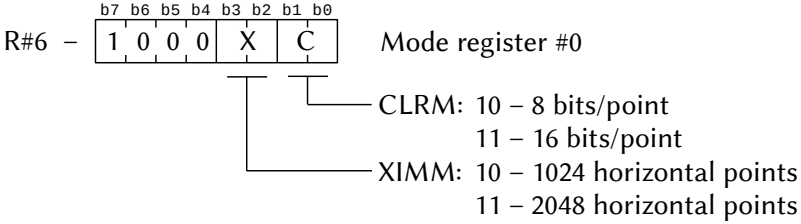


5.8.3.14 – B6 Mode

B6 mode is selected by the following registers:

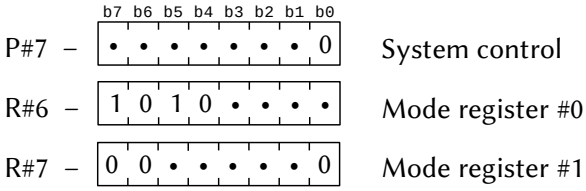


B6 mode is 640 x 480 points and is a high resolution mode, only working on VGA type monitors (fH=31.5 KHz). The image area can vary between 1024 and 2048 horizontal points and 512 and 2048 vertical points, depending on the size of the VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.

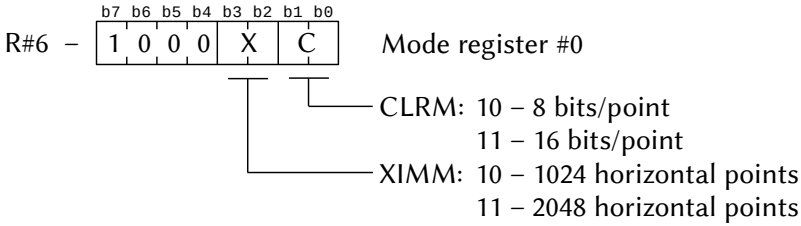


5.8.3.15 – B7 Mode

B7 mode is selected by the following registers:



B7 mode is 1024 x 212 points and is an undocumented mode officially. Despite the high horizontal resolution, it works on NTSC and PAL monitors, having fH = 15.75 Khz. The image area can vary between 1024 and 2048 points horizontally and 512 and 2048 points vertical, depending on the size of the VRAM and the type of palette used. These sizes were described at the beginning of this section. The number of bits used per displayed point and the horizontal size of the image area are specified in R#6. The number of vertical points is automatically set.



5.8.3.16 – Memory maps of B0~B7 modes

The memory area occupied by the B0~B7 modes is linear for the entire image area, reserving 512 bytes for the cursor function, at the end of the available memory, although even this area can be displayed. Points are assigned linearly, from left to right and then from top to bottom, referring to the image area and not the area displayed on the screen. The arrangement of points in memory also depends on the number of bits used for each point. The example below refers to an image area of 511 x 511 points, reserving 8 bits for each point.

	0	1		510	511
0	00000H	00001H	.....	001FEH	001FFH
1	00200H				003FFH
	⋮				⋮
511	3FE00H	3FE01H	.....	3FFFEH	3FFFFH

The points are distributed in memory according to the palette used, as illustrated below.

2 bits per point

	b7 b6	b5 b4	b3 b2	b1 b0
00 000H	0,0	1,0	2,0	3,0
00 001H	4,0	5,0	6,0	7,0
00 002H	8,0	9,0	10,0	11,0

4 bits per point

	b7 b6 b5 b4	b3 b2 b1 b0
00 000H	0,0	1,0
00 001H	2,0	3,0
00 002H	4,0	5,0

8 bits per point

	b7 b6 b5 b4 b3 b2 b1 b0
00 000H	0,0
00 001H	1,0
00 002H	2,0

16 bits per point

	b7 b6 b5 b4 b3 b2 b1 b0
00 000H	0,0 low
00 001H	0,0 high
00 002H	1,0 low

5.8.4 – Color Specifications

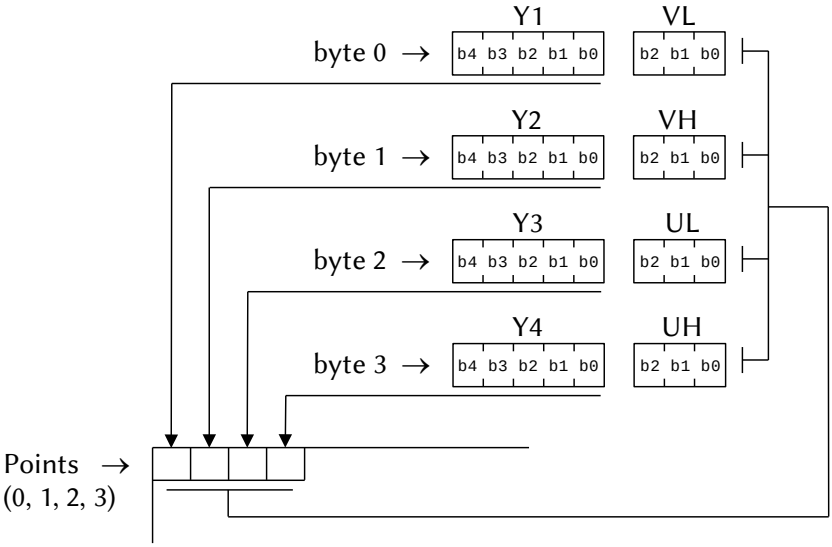
As already described, each point can occupy 2, 4, 8 or 16 bits in modes B1 to B6. However, there are nine different representation types that can be selected. These types are as follows:

1. BYUV – Pure YUV mode in B1 ~ B6 modes.
2. BYUVP – YUV mode plus palette in B1 ~ B6 modes.
3. BYJK – Pure YJK mode in B1 ~ B6 modes.
4. BYJKP – YJK mode plus palette in B1 ~ B6 modes.
5. BD16 – Displays 32 768 simultaneous colors without palette.
6. BD8 – Displays 256 simultaneous colors without palette.
7. BP6 – Presents 64 simultaneous colors / 32 768 palette.

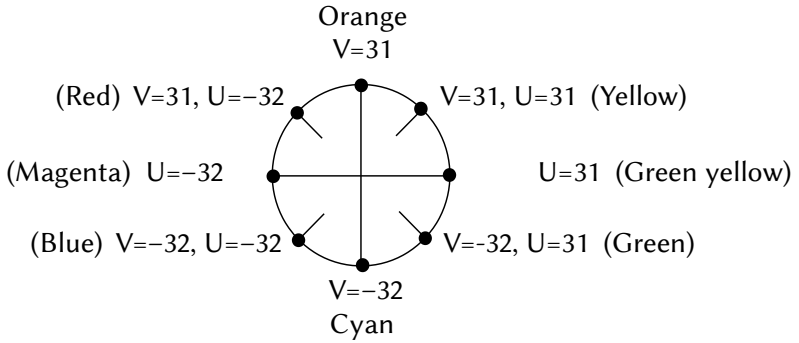
8. BP4 – Presents 16 simultaneous colors / 32768 palette.
9. BP2 – Presents 4 simultaneous colors / 32768 palette.

5.8.4.1 – BYUV mode

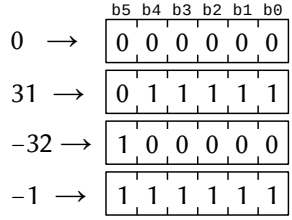
BYUV mode features up to 19268 simultaneous colors using only 8 bits per point. For this, the points are distributed in groups of 4 horizontally, as shown in the illustration below.



The colors are chosen by the U and V vectors, as illustrated below.



The binaries corresponding to the values shown in the illustration are as follows:



Since there are 12 bits to represent the color, we make 2^12 = 4096 colors, which is the maximum number of colors that can be defined. Each group of 4 horizontal points can only have one color chosen from these 4096. However, each individual point in this group can have a brightness range of 32 levels, represented by the Yn bits, where 00 000B represents minimum brightness and 11 111B represents maximum brightness.

The U and V vectors can vary from –32 to 31, as illustrated above. By combining the extreme values, the four primary colors of the YUV system can be formed: green, red, blue and yellow. Using four vectors of primary colors does not change the color mixing system used by the RGB system; it is only necessary to take into account the use of one more color. Using the intermediate values, 4096 colors can be generated. The conversion from the YUV system to RGB and vice versa can be done using the following formulas:

```
Y = R/4 + G/2 + B/8 R = Y + U
U = R - Y G = 5/4 Y - 1/2 U - 1/4 V
V = G - Y B = Y + V
```

An important detail is the number of colors. Since there are 4096 colors and 32 brightness levels for each one, there are actually 32 * 4096 = 131 072 possible colors. It turns out that in this mode the colors are not completely independent for each point (apart from technical characteristics of the V9990 that are irrelevant), which causes a reduction in the number of colors presented simultaneously to 19 268.

The Address in the VRAM of each point in the image area can be calculated by the following expression:

```
Address = X + Y*(Horizontal area size in points)
```

Where X is the horizontal coordinate and Y is the vertical coordinate.

BYUV mode is selected by R#6 and R#13, as illustrated below.

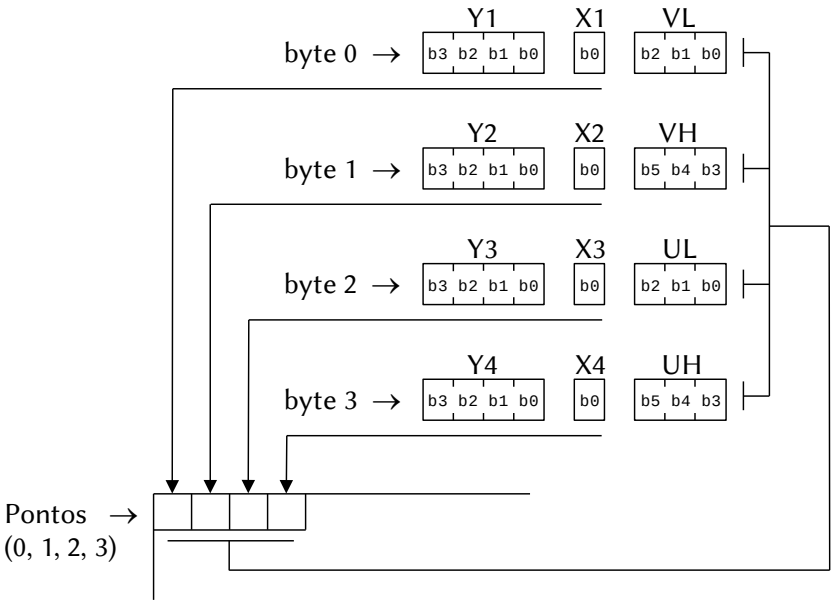
R#6 – | 1 | 0 | • | • | • | • | 1 | 0 | Mode register #0

R#13 – | 1 | 1 | 0 | • | • | • | • | • | Palette register

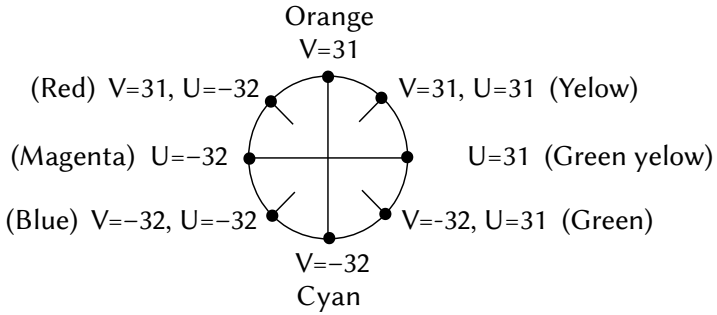
5.8.4.2 – BYUVP mode

The BYUVP mode is a mixed mode, being able to present up to 12 499 simultaneous colors, through the YUV system, or using the palette.

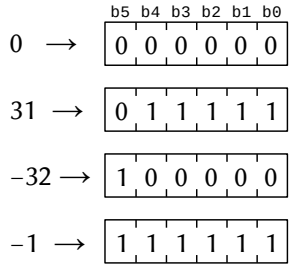
As in the YUV system, here the points are also arranged in fours horizontally. Each group of 4 points can have a single color, chosen from 4096, with up to 16 brightness levels for each individual point. Or, each point can have up to 16 colors chosen from a palette of 32 768. Arrangement in this way is illustrated below.



When the Xn bits are 0, the system used will be YUV, with the only difference that the brightness variation has only 16 levels, and not 32, as in the BYUV mode, since Yn can only vary from 0 to 15. the Xn bits are 1, the color will be chosen from the palette. Up to 16 colors out of 32768 can be chosen. It is not mandatory that all Xn bits are the same, and there may be mixing in the 4 points that make up the group. When the YUV system is selected, the point group colors are chosen by the U and V vectors, as shown in the illustration below.



The binaries corresponding to the values shown in the illustration are as follows:



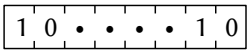
Since there are 12 bits to represent the color (6 bits for U and 6 bits for V), we make 2^12 = 4096 colors, which is the maximum number of colors that can be defined. Each group of 4 horizontal points can only have one color chosen from these 4096. However, each individual point of this group can have a brightness variation of 16 levels, represented by the Yn bits, where 0000B represents minimum brightness and 1111B maximum brightness.

An important detail is the number of colors. Since there are 4096 colors and 16 saturation levels for each one, there are actually 16 * 4096 = 65 536 possible colors. But as in this mode the colors are also not completely independent for each point, (apart from technical characteristics of the V9990 that are not the case), there is a reduction in the number of colors presented simultaneously to 12 499.

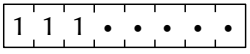
The Address in the VRAM of each point in the image area can be calculated by the following expression:

```
Address = X + Y*(Horizontal area size in points)
```

BYUV mode is selected by R#6 and R#13, as illustrated below.

R#6 – 

Mode register #0

R#13 – 

Palette register

5.8.4.3 – BYJK mode

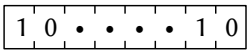
The BYJK mode is exactly the same as the BYUV mode, except for the ratio of the color vectors (JK or UV), where the green and blue color channels are swapped. In the case of the YJK system, the conversion to the RGB system can be done by the following formulas:

```
Y = R/4 + G/8 + B/2 R = Y + J
J = R - Y G = Y + K
K = G - Y B = 5/4 Y - J/2 - K/4
```

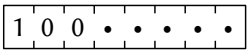
Conversion between YJK and YUV systems can be done using the following formulas:

```
Y = Y                    Y = Y
J = U                    U = J
K = Y/4 - U/2 - V/4      V = Y/4 - J/2 - K/4
```

BYJK mode is selected by R#6 and R#13, as illustrated below.

R#6 – 

Mode register #0

R#13 – 

Palette register

5.8.4.4 – BYJKP mode

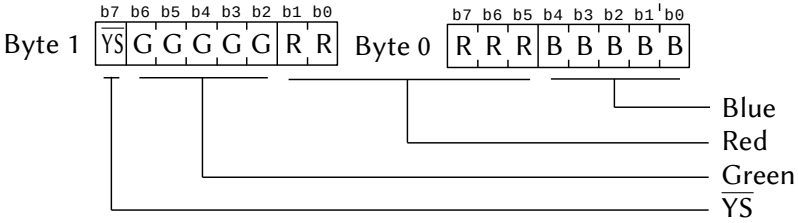
The BYJKP mode is exactly the same as the BYUVP mode, except for the proportion of the color vectors (JK or UV), where the green and blue color channels are swapped and can be calculated by the formulas presented above. This mode is selected by R#6 and R#13, as illustrated below.

R#6 – | 1 | 0 | • | • | • | • | 1 | 0 | Mode register #0

R#13 – | 1 | 0 | 1 | • | • | • | • | • | Palette register

5.8.4.5 – BD16 mode

In this mode, up to 32768 colors can be displayed simultaneously, without using a palette. Two bytes are reserved for each point, in the form LSB-MSB. 5 bits are used for each primary color, plus one bit for the $\overline{YS}$ (superimpose) function. These bits are arranged in the VRAM as shown in the illustration below.



Where G is the intensity of green (00000B to 11111B), R is red and B is blue. $\overline{YS}$ is a flag to indicate superimpose for each individual point. When $\overline{YS}$ is 0, the superimpose function for the point is disabled; when it is 1 it will be activated. The address of each point in the image area can be calculated by the following expression:

```
Address = X*2 + Y*(horizontal area size in points)*2
```

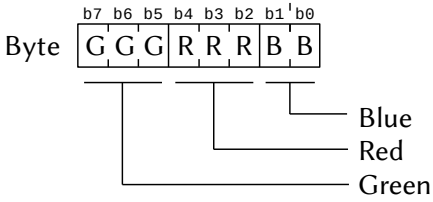
BD16 mode is selected by the following registers:

R#6 – | 1 | 0 | • | • | • | • | 1 | 1 | Mode register #0

R#13 – | 0 | 0 | 0 | • | • | • | • | • | Palette register

5.8.4.6 – BD8 mode

This mode is similar to the previous one, but only reserves 8 bits for each point; therefore, only 256 colors can be displayed simultaneously, also without using the palette. Each byte is organized as illustrated below.



Where G is the intensity of green (0 to 7), R a of red (0 to 7) and B a of blue (0 to 3). The Address of each point in the image area can be calculated by the following expression:

```
Address = X + Y*(Horizontal area size in points)
```

This mode is selected by the following registers:

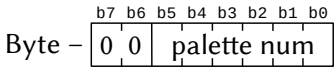
R#6 – 1 0 • • • • 1 0 Mode register #0

R#13 – 0 1 0 • • • • • Palette register

In this mode, there is no way to superimpose.

5.8.4.7 – BP6 Mode

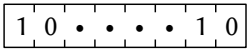
In this mode, each point can have up to 64 colors chosen from a palette of 32 768. Each point is represented by a byte, with only the lowest six bits being valid, as shown in the illustration below.



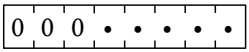
Where "palette no" can range from 0 to 63. This mode allows selective superimpose, as long as this function is selected for the respective color in the palette (bit $\overline{YS}$ set to 1). In this case, all points with the same color will be selected for superimpose. The address of each point in the image area can be calculated by the following expression:

```
Address = X + Y*(Horizontal area size in points)
```

This mode is selected by the following registers:

R#6 – 

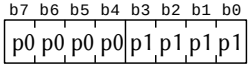
Mode register #0

R#13 – 

Palette register

5.8.4.8 – BP4 mode

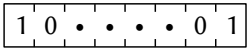
In this mode, each point can have up to 16 colors chosen from a palette of 32768. The advantage over BP6 mode is that each byte is used to represent two points, occupying half the memory of BP6 mode, as illustrated below.



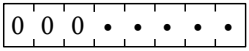
Where "p0" represents the palette of the first point (0 to 15) and "p1" represents the palette of the next point in the horizontal direction, each one can vary from 0 to 15. This mode allows selective superimpose, as long as this function is selected for the respective color in the palette (bit $\overline{YS}$ set to 1). In this case, all points with the same color will be selected for superimpose. The address of each point in the image area can be calculated by the following expression:

```
Address = X/2 + Y*(Horizontal area size in points)/2
     Even point: 4 bits higher
     Odd point: 4 bits lower
```

BP4 mode is selected by the following registers:

R#6 – 

Mode register #0

R#13 – 

Palette register

5.8.4.9 – BP2 Mode

In this mode, each point can have up to 4 colors chosen from a palette of 32768. It occupies only half of the memory used by BP4 mode, since each byte represents 4 points in the horizontal direction, reserving only 2 bits for each one, as shown below.

	b7	b6	b5	b4	b3	b2	b1	b0
Byte –	p0	p0	p1	p1	p2	p2	p3	p3

Where p0 represents the palette of the first point (0 to 3), p1 represents the palette of the second point (0 to 3), p2 the palette of the third point (0 to 3) and p3 the palette of the fourth point (0 to 3), always horizontally, from left to right. This mode allows selective superimpose, as long as this function is selected for the respective color in the palette (bit $\overline{YS}$ set to 1). In this case, all points with the same color will be selected for superimpose. The address of each point in the image area can be calculated by the following expression:

```
Address = X/4 + Y*(Horizontal area size in points)/4
     First point:  bits b7 and b6
     Second point: bits b5 and b4
     Third point:  bits b3 and b2
     Fourth point: bits b1 and b0
```

BP2 mode is selected by the following registers:

R#6 –	1	0	•	•	•	•	0	0	Mode register #0
R#13 –	0	0	0	•	•	•	•	•	Palette register

5.8.4.10 – Modes P1 e P2 colors

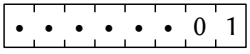
The modes seen so far are valid only for screens B1 to B6. For screens P1 and P2, there is a special mode, PP. There are 4 palettes that can each display 16 colors of 32768. Two of them can be used simultaneously, according to the table below.

P1 Mode, “A” plane	bits b1 and b0 of R#13
P1 Mode, “B” plane	bits b3 and b2 of R#13
P2 Mode, odd points	bits b1 and b0 of R#13
P2 Mode, even points	bits b3 and b2 of R#13

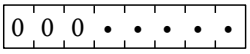
In fact, the palette is just one and has 64 positions. What the 2 bits of R#13 reserved for the selection of palettes select are the 4 segments of 16 positions within these 64, as shown in the table below.

b1/b3	b0/b2	
0	0	0 to 15 positions
0	1	16 to 31 positions
1	0	32 to 47 positions
1	1	48 to 63 positions

These modes also allow selective superimpose, as long as this function is selected for the respective color in the palette (bit $\overline{YS}$ set to 1). In this case, all points with the same color will be selected for superimpose. PP mode is selected by the following registers:

R#6 – 

Mode register #0

R#13 – 

Palette register

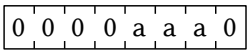
5.8.5 – Sprites and cursors

There are two sprite modes that can be used on the V9990. A more powerful mode is used for screens P1 and P2. The other is called the cursor function, and is used for modes B1 to B6.

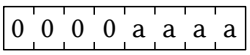
5.8.5.1 – Sprites for P1 and P2 modes

For these modes, up to 125 16 x 16 sprites can be defined with 16 independent colors for each point chosen from 32 768 (including the "transparent" color needed to shape the sprite). Up to 16 sprites can be placed on each horizontal line, and all 125 can be displayed simultaneously on the screen. In the case of P1 mode, the priority of the sprites can be defined taking into account the two image planes.

The sprites' format is defined through the Sprite Patterns Generator Table, and its initial Address is indicated by register R#25, as illustrated below.

R#25 – 

A17~A15 – (P1 mode)

R#25 – 

A18~A15 – (P2 mode)

The sprite generator table can only start in multiples of 16 Kbytes starting at 00 000H. For P1 mode, it has the following structure:

Offset		
00000H	Line 0	S0~S15
00080H	Line 1	S0~S15
00100H	⋮	
00380H	Line 15	S0~S15
003FFH	Line 0	S16~S31

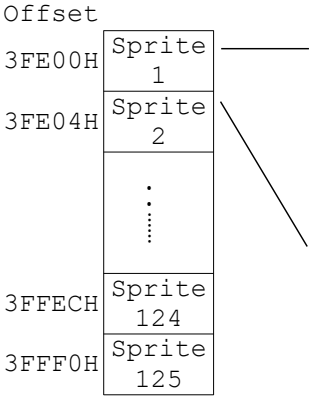
Offset	b7 b6 b5 b4 b3 b2 b1 b0		
00000H	SC0	SC1	S0 line 0
	⋮		
00007H	SC14	SC15	S0 line 0
00008H	SC0	SC1	S1 line 0
	⋮		
0007FH	SC14	SC15	S15 line 0
00080H	SC0	SC1	S0 line 1

For P2 mode, there is a slight change in Address.

Offset		
00000H	Line 0	S0~S31
00100H	Line 1	S0~S31
00200H	⋮	
00F00H	Line 15	S0~S31
01000H	Line 0	S32~S63

Offset	b7 b6 b5 b4 b3 b2 b1 b0		
00000H	SC0	SC1	S0 line 0
	⋮		
00007H	SC14	SC15	S0 line 0
00008H	SC0	SC1	S1 line 0
	⋮		
000FFH	SC14	SC15	S31 line 0
00100H	SC0	SC1	S0 line 1

Sprite Attribute Table always starts at 3FE00H and ends at 3FFFFH for P1 mode and 7BE00H/7BFFFH for P2 mode. It is organized as shown in the illustration below.



Offset	b7	b6	b5	b4	b3	b2	b1	b0	
3FE00H	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	Vert co-ord
3FE00H	P7	P6	P5	P4	P3	P2	P1	P0	Patt number
3FE00H	X7	X6	X5	X4	X3	X2	X1	X0	Hor co-ord
3FE00H	C5	C4	C3	C2			X9	X8	Addit info

The sprites can be located by the X and Y values only in the screen area (256 x 212 for P1 mode and 512 x 212 for P2 mode). The vertical position of the sprite is the position specified by bits Y plus 1. For each sprite, one of the 4 segments of 16 positions on the palette can be selected, using bits P1 and P0, as shown in the table below:

P1	P0	
0	0	Position 0 to 15
0	1	Position 16 to 31
1	0	Position 32 to 47
1	1	Position 48 to 63

As illustrated in the table above, the entire palette can be selected for the 125 sprites, which can have up to 64 colors out of 32768. However, only 15 colors are possible per individual sprite plus the transparent 0 color needed to define the sprite design.

The priority of sprites presentation in P1 mode takes into account the two screen planes (A and B), as shown in the table below.

P1	P0	Priority order
0	0	SP > A > B > BD
1	0	A > SP > B > BD
-	1	A > B > BD (4)

SP – Sprites plane
A – Front image plane
B – Back image plane
BD – Background plane

The presentation priority in P2 mode follows the table below.

P1	P0	Priority order
0	0	SP > IP > BD
1	0	IP > SP > BD
-	1	IP > BD (4)

SP – Sprites plane
IP – Image plane
BD – Background plane

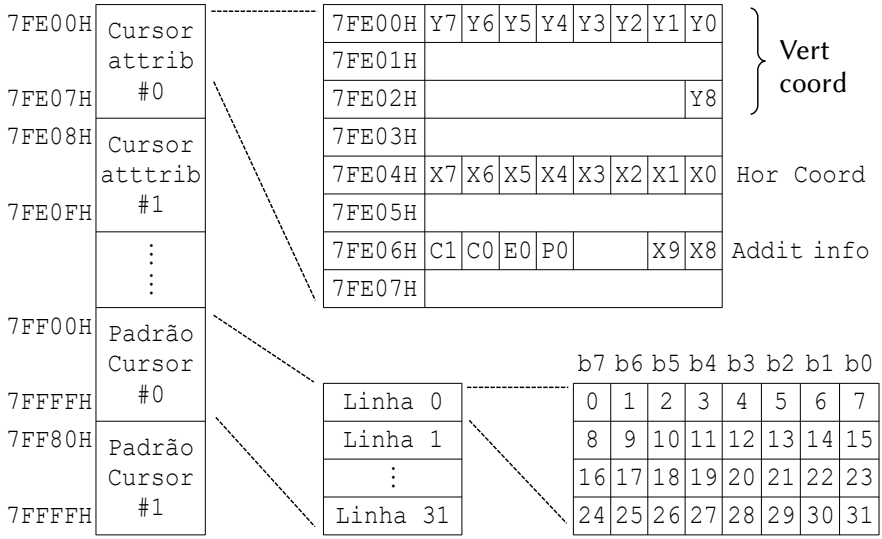
In either case, the lower-numbered sprite has higher display priority than the higher-numbered sprite, as for the VDP V9958.

5.8.5.2 – Cursors for modes B0 to B7

For modes B1 to B6, there is a function called “cursor”. In fact, they are sprites with far fewer resources. Only 2 cursors 32 x 32 points can be defined, with only 1 color each. Optionally, an XOR operation can

4 The sprites are not displayed when P0=1

be performed between the cursor and image points. Cursors are always defined at the end of memory, from 7FE00H to 7FFFFH, occupying 512 bytes. The structure of this table is illustrated below.



The area where the cursor can be displayed is the screen area. The vertical display position of the cursor is equal to the defined Y coordinate plus 1 (or plus 2 in the case of interlaced mode). In fact, the choice of colors is quite limited: only one of the 4 initial colors of the palette can be chosen, with color 0 being transparent. Color is selected by bits C1 and C0 (0 to 3). If the EO bit is set, a logical XOR operation will be performed between the cursor points and the image points. P0 is a flag that indicates the cursor display: if it is 0, the cursor will be displayed; if it is 1, it will not be.

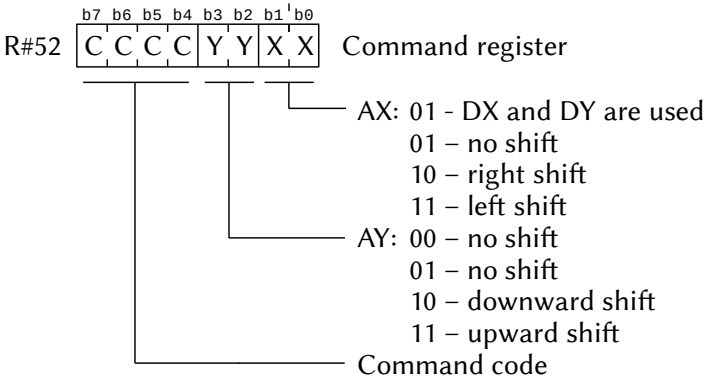
5.8.6 – VDP V9990 commands

The V9990 also has hardware commands, which work similarly to the V9938 and V9958 VDPs. These commands can be executed in both P1 and P2 and B1 to B6 modes and are based on the image area; therefore its parameters vary according to the selection of this area.

The V9990 has 15 possible hardware commands plus a stop command. When the respective value is written to register R#52, the command starts to be executed. Registers R#32 to R#50 plus R#53 and R#54 must be specified before executing the command. The table below briefly describes all commands.

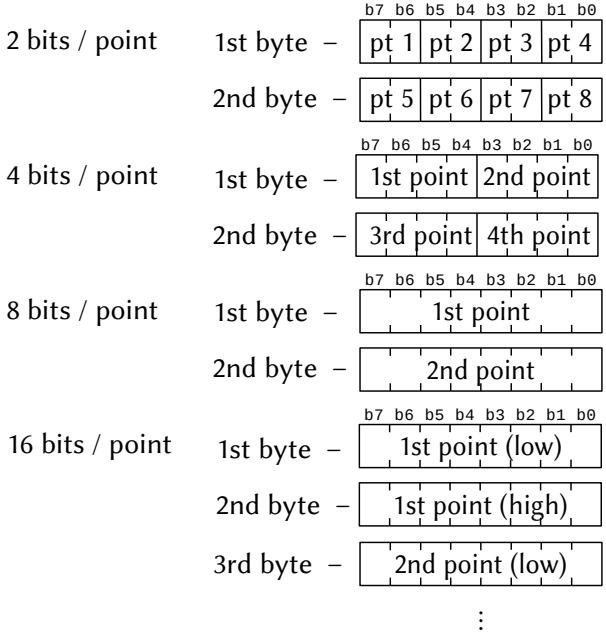
0 0 0 0	STOP	Stop command
0 0 0 1	LMMC	CPU → VRAM transfer (coordinates)
0 0 1 0	LMMV	Paint rectangle in the VRAM
0 0 1 1	LMCM	VRAM → CPU transfer (coordinates)
0 1 0 0	LMMM	VRAM ↔ VRAM transfer (coordinates)
0 1 0 1	CMMC	CPU → VRAM character transfer
0 1 1 0	CMMK	KanjiROM → VRAM data transfer
0 1 1 1	CMMM	VRAM → VRAM character transfer
1 0 0 0	BMXL	VRAM ↔ VRAM transfer (linear → coord)
1 0 0 1	BMLX	VRAM ↔ VRAM transfer (coord → linear)
1 0 1 0	BMLL	VRAM ↔ VRAM transfer (linear → linear)
1 0 1 1	LINE	Draw a line
1 1 0 0	SRCH	Search point color code
1 1 0 1	POINT	Read point color code
1 1 1 0	PSET	Draw a point and advance coordinates
1 1 1 1	ADVN	Advance coordinates without drawing

The command code must be written in R#52 in the following format (AX and AY values are only valid for PSET and ADVN commands):

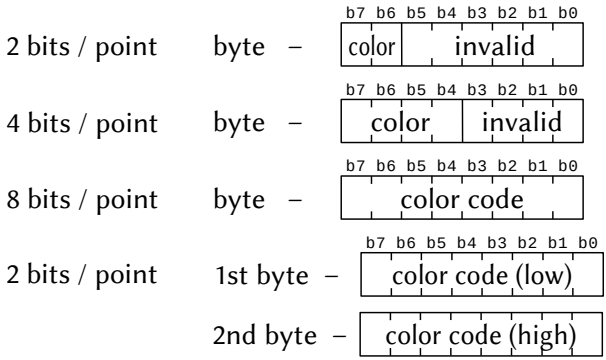


5.8.6.1 – Data formats for commands

For some commands, the format of the data to be sent to the VDP varies depending on the palette used. For the LMMC and LMCM commands, the data written by the P#2 port (command port) must be specified as illustrated below.



For the POINT command, the format for the data is as follows:



5.8.6.2 – Parameters for commands

The parameters for executing each of the commands must be loaded into registers R#32 to R#51 before sending the respective command to register R#52. These parameters are described below.

	b7	b6	b5	b4	b3	b2	b1	b0
R#32 –	SX7	SX6	SX5	SX4	SX3	SX2	SX1	SX0
	SA7	SA6	SA5	SA4	SA3	SA2	SA1	SA0
	KA7	KA6	KA5	KA4	KA3	KA2	KA1	KA0
R#33 –	0	0	0	0	0	SX10	SX9	SX8
R#34 –	SY7	SY6	SY5	SY4	SY3	SY2	SY1	SY0
	SA15	SA14	SA13	SA12	SA11	SA10	SA9	SA8
	KA15	KA14	KA13	KA12	KA11	KA10	KA9	KA8
R#35 –	0	0	0	0	SY11	SY10	SY9	SY8
	0	0	0	0	0	SA18	SA17	SA16
	0	0	0	0	0	0	KA17	KA16

These registers specify the starting coordinates and/or Addresses for the execution of commands, and can be set in three different ways, depending on the command to be executed.

LMCM, LMMM, BMLX, SRCH and POINT commands:

SX0~10: Specifies the starting horizontal coordinate. It will be set to 0 when it is greater than the width of the image area. In P1 mode, plane "A" will be selected when SX9=0 and plane "B" will be selected when SX9=1.

SY0~10: Specifies the starting vertical coordinate. It will be set to 0 when it is greater than the height of the image area.

CMMM, BMXL and BMLL commands:

SA0~18: Specifies the VRAM starting address.

CMMK command:

SA0~18: Specifies the Kanji-ROM Address.

	b7	b6	b5	b4	b3	b2	b1	b0
R#36 –	DX7	DX6	DX5	DX4	DX3	DX2	DX1	DX0
	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
R#37 –	0	0	0	0	0	DX10	DX9	DX8
R#38 –	DY7	DY6	DY5	DY4	DY3	DY2	DY1	DY0
	DA15	DA14	DA13	DA12	DA11	DA10	DA9	DA8
R#39 –	0	0	0	0	DY11	DY10	DY9	DY8
	0	0	0	0	0	DA18	DA17	DA16

The registers above specify the final coordinates and/or Addresses for executing the commands, and can be set in two different ways, depending on the command to be executed.

Commands LMMC, LMMV, LMMM, CMMC, CMMK, CMMM, BMXL, LINE, PSET and ADVN.

DX0~10: Specifies the final horizontal coordinate. It will be set to 0 when the specified coordinate is greater than the width of the image area. In P1 mode, plane "A" will be selected when DX9=0 and plane "B" will be selected when DX9=1.

DY0~11: Specifies the final vertical coordinate. It will be set to 0 when the specified coordinate is greater than the height of the image area.

BMLX and BMLL commands

DA0~18: Specifies the final address of the VRAM.

	b7	b6	b5	b4	b3	b2	b1	b0
R#40 –	NX7	NX6	NX5	NX4	NX3	NX2	NX1	NX0
	NA7	NA6	NA5	NA4	NA3	NA2	NA1	NA0
	MJ7	MJ6	MJ5	MJ4	MJ3	MJ2	MJ1	MJ0

R#41 –	0	0	0	0	0	NX10	NX9	NX8
	0	0	0	0	MJ11	MJ10	MJ9	MJ8
R#42 –	NY7	NY6	NY5	NY4	NY3	NY2	NY1	NY0
	NA15	NA14	NA13	NA12	NA11	NA10	NA9	NA8
	MI7	MI6	MI5	MI4	MI3	MI2	MI1	MI0
R#43 –	0	0	0	0	NY11	NY10	NY9	NY8
	0	0	0	0	0	NA18	NA17	NA16
	0	0	0	0	MI11	MI10	MI9	MI8

The registers above specify the number of points or bytes for the execution of commands, and can be set in three different ways, depending on the command to be executed.

Commands LMMC, LMMV, LMCM, LMMM, CMMC, CMMK, CMMM, BMXL and BMLX

NX0~10: Specifies the number of points in the horizontal direction. It will be set to 0 when the specified coordinate is larger than the image area size. Its maximum value is 2048 (all bits equal to 0).

NY0~11: Specifies the number of points in the vertical direction. It will be set to 0 when the specified coordinate is greater than the height of the image area. Its maximum value is 4096 (all bits equal to 0).

BMLL command

NA0~18: Specifies the number of bytes to transfer. It will be set to 0 when its value exceeds the capacity of the VRAM. Its maximum value is 512K (all bits equal to 0).

LINE command

MJ0~11: Size of the longer side of the reference right triangle in points. It will be set to 0 when its value exceeds the size of the image area.

MI0~11: Size of the short side of the reference right triangle in points. It will be set to 0 when its value exceeds the size of the image area.

	b7	b6	b5	b4	b3	b2	b1	b0
R#44 –	•	•	•	•	DIY	DIX	NEQ	MAJ

This is the argument register.

DIY: Vertical direction of transfer. Indicates increment when it is 0 (offset down) and decrement when it is 1 (offset up). With the BMXL and BMLX commands, the linear Address is always incremented.

DIX: Horizontal transfer direction. Indicates increment when it is 0 (right shift) and decrement when it is 1 (left shift). With the BMXL and BMLX commands, the linear address is always incremented and with BMLL, DIX and DIY are specified equally.

NEQ: In the border color specification for SRCH, 0 indicates color specified for detection and 1 color not specified.

MAJ: Indicates the direction of the longer side of the reference right triangle for the LINE command. If it is 0, the long side will be parallel to the X axis (horizontal) and if it is 1, it will be parallel to the Y axis (vertical).

	b7	b6	b5	b4	b3	b2	b1	b0
R#45 –	•	•	•	TP	L11	L10	L01	L00

This register specifies the logical opcode that can be performed between the source and destination color code bits. When the TP bit is 1, origin points that have color code 0 (transparent) will not be transferred. Possible codes are listed below.

L11-L10-L01-L00	Logical operation
0 0 0 0	
0 0 0 1	WC = not (SC or DC)
0 0 1 0	
0 0 1 1	WC = not (SC)
0 1 0 0	
0 1 0 1	
0 1 1 0	WC = SC xor DC
0 1 1 1	WC = not (SC and DC)
1 0 0 0	WC = SC and DC

1 0 0 1	WC = not (SC xor DC)
1 0 1 0	
1 0 1 1	
1 1 0 0	WC = SC
1 1 0 1	
1 1 1 0	WC = SC or DC
1 1 1 1	

	b7	b6	b5	b4	b3	b2	b1	b0
R#46 –	WM7	WM6	WM5	WM4	WM3	WM2	WM1	WM0
R#47 –	WM15	WM14	WM13	WM12	WM11	WM10	WM9	WM8

These registers specify a bitwise write mask. R#46 is the mask for VRAM0 and R#47 for VRAM1. For P1 mode, R#46 is the mask for the "A" plane and R#47 for the "B" plane. When the bit of these registers is 1, writing is enabled for the respective bit of the data to be written. When the register bit is 0, writing is prohibited.

	b7	b6	b5	b4	b3	b2	b1	b0
R#48 –	FC7	FC6	FC5	FC4	FC3	FC2	FC1	FC0
R#49 –	FC15	FC14	FC13	FC12	FC11	FC10	FC9	FC8
R#50 –	BC7	BC6	BC5	BC4	BC3	BC2	BC1	BC0
R#51 –	BC15	BC14	BC13	BC12	BC11	BC10	BC9	BC8

These registers specify the font color for the CMMC, CMMK, and CMMM commands and also the drawing color for LMMV, LINE, and PSET. For the SRCH command, specify the border color via FC0~FC15. The correspondence in the VRAM is the same as in the write mask. FC0~FC15 is the color code for data source #1 and BC0~BC15 for data source #0. The color codes format must be set according to the value contained in R#6, as shown below:

16 bits per point: All bits are valid.
8 bits per point: Same data for 0~7 and 8~15.
4 bits per point: Same data for 0~3, 4~7, 8~11 and 12~15.
2 bits per point: Fill 0~15 eight times with the 2 bits.

5.8.6.3 – Executing the commands

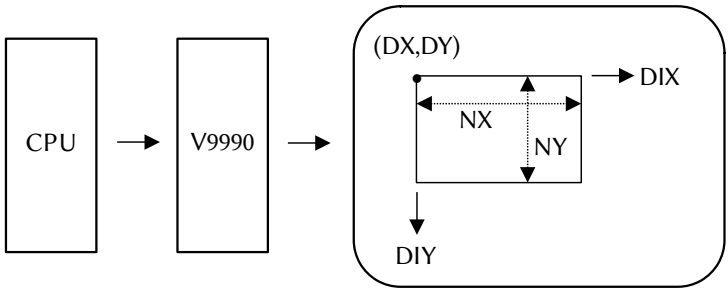
To execute the V9990 hardware commands, it is first necessary to set all values in the appropriate registers. Then, just write the command code in R#52, along with the data for shifting points for the PSET and ADVN commands. To stop the command, just write the stop command in R#52 (00H).

While the command is running, the CE bit of port P#5 will be set to 1.

P#5 –	•	•	•	•	•	•	•	CE

5.8.6.4 – LMMC (Logical transfer CPU → VRAM)

In this command, data is transferred from the CPU to a rectangular area in the VRAM.



The registers must be loaded according to the illustration below.

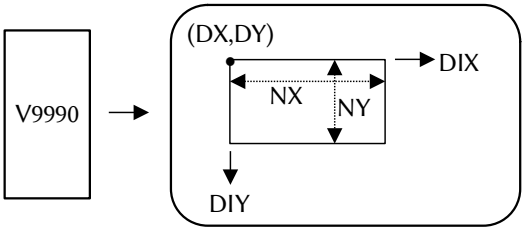
	b7	b6	b5	b4	b3	b2	b1	b0			
R#36 –	x	x	x	x	x	x	x	x	– x7~x0	DX	Destination horizontal coordinate (0 a 2047)
R#37 –	0	0	0	0	0	x	x	x	– x10~x8		
R#38 –	y	y	y	y	y	y	y	y	– y7~y0	DY	Destination vertical coordinate (0 a 4095)
R#39 –	0	0	0	0	y	y	y	y	– y11~y8		
R#40 –	x	x	x	x	x	x	x	x	– x7~x0	NX	Number of points to transfer in the horizontal direction
R#41 –	0	0	0	0	0	x	x	x	– x8		

	b7 b6 b5 b4 b3 b2 b1 b0			
R#42 –	y y y y y y y y	– y7~y0	NY	Number of points to transfer in the vertical direction
R#43 –	0 0 0 0 y y y y	– y9~y8		
R#44 –	0 0 0 0 y x 0 0	– DIY,DIX		Transfer direction
R#45 –	0 0 0 t l l l l	– LOP		Logical operation code
R#46 –	w w w w w w w w	– wm7~wm0		Write mask
R#47 –	w w w w w w w w	– wm15~wm8		
R#52 –	0 0 0 1 0 0 0 0	– OP-CODE		LMMC command code

When executing the command, the required number of bytes to transfer must be sent through the command port (P#2).

5.8.6.5 – LMMV (Draw rectangle)

This command draws a rectangle in the image area.



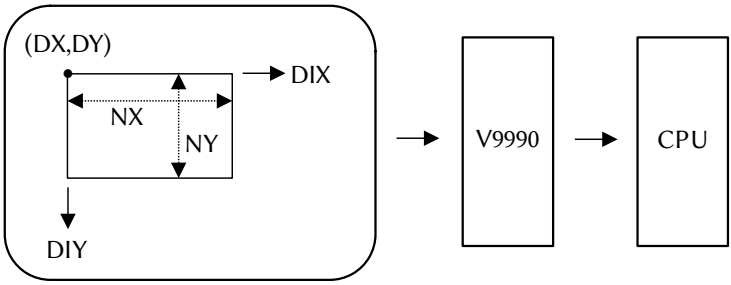
The registers must be loaded according to the illustration below.

	b7 b6 b5 b4 b3 b2 b1 b0			
R#36 –	x x x x x x x x	– x7~x0	DX	Destination horizontal coordinate (0 a 2047)
R#37 –	0 0 0 0 0 x x x	– x10~x8		
R#38 –	y y y y y y y y	– y7~y0	DY	Destination vertical coordinate (0 a 4095)
R#39 –	0 0 0 0 y y y y	– y11~y8		

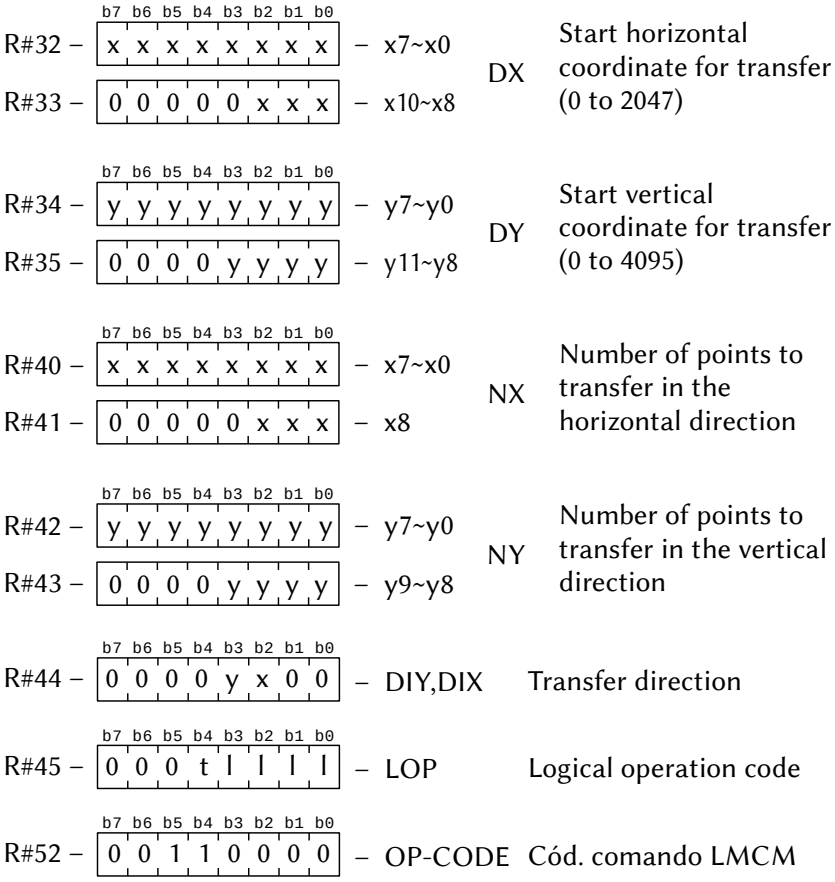
	b7	b6	b5	b4	b3	b2	b1	b0			
R#40 –	x	x	x	x	x	x	x	x	– x7~x0	NX	Number of points to be painted in the horizontal direction
R#41 –	0	0	0	0	0	x	x	x	– x8		
R#42 –	y	y	y	y	y	y	y	y	– y7~y0	NY	Number of points to be painted in the vertical direction
R#43 –	0	0	0	0	y	y	y	y	– y9~y8		
R#44 –	0	0	0	0	y	x	0	0	– DIY,DIX		Transfer direction
R#45 –	0	0	0	t	l	l	l	l	– LOP		Logical operation code
R#46 –	w	w	w	w	w	w	w	w	– wm7~wm0		Write mask
R#47 –	w	w	w	w	w	w	w	w	– wm15~wm8		
R#48 –	f	f	f	f	f	f	f	f	– fc7~fc0		Paint color code
R#49 –	f	f	f	f	f	f	f	f	– fc15~fc8		
R#52 –	0	0	1	0	0	0	0	0	– OP-CODE		LMMV command code

5.8.6.6 – LMCM (Logical Transfer VRAM → CPU)

In this command, data from a rectangular area in VRAM is transferred to the CPU. The LMCM command is illustrated below.



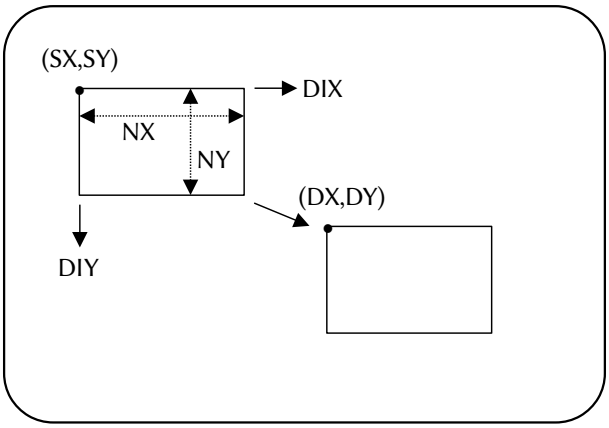
The following registers must be loaded:



The data bytes to be transferred must be sent through the command port (P#2).

5.8.6.7 – LMMM (Logical Transfer VRAM → VRAM)

In this command, a rectangular area of the VRAM is transferred to another position in the VRAM. Logical operations on the destination are possible.



The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0			
R#32 –	x	x	x	x	x	x	x	x	– x7~x0	DX	Start horizontal coordinate for transfer (0 to 2047)
R#33 –	0	0	0	0	0	x	x	x	– x10~x8		
R#34 –	y	y	y	y	y	y	y	y	– y7~y0	DY	Start vertical coordinate for transfer (0 to 4095)
R#35 –	0	0	0	0	y	y	y	y	– y11~y8		
R#36 –	x	x	x	x	x	x	x	x	– x7~x0	DX	Destination horizontal coordinate (0 a 2047)
R#37 –	0	0	0	0	0	x	x	x	– x10~x8		
R#38 –	y	y	y	y	y	y	y	y	– y7~y0	DY	Destination vertical coordinate (0 a 4095)
R#39 –	0	0	0	0	y	y	y	y	– y11~y8		
R#40 –	x	x	x	x	x	x	x	x	– x7~x0	NX	Number of points to transfer in the horizontal direction
R#41 –	0	0	0	0	0	x	x	x	– x8		

	b7 b6 b5 b4 b3 b2 b1 b0			
R#42 –	y y y y y y y y	– y7~y0	NY	Number of points to transfer in the vertical direction
R#43 –	0 0 0 0 y y y y	– y9~y8		
R#44 –	0 0 0 0 y x 0 0	– DIY,DIX		Transfer direction
R#45 –	0 0 0 t l l l l	– LOP		Logical operation code
R#46 –	w w w w w w w w	– wm7~wm0		Write mask
R#47 –	w w w w w w w w	– wm15~wm8		
R#52 –	0 1 0 0 0 0 0 0	– OP-CODE		LMMM command code

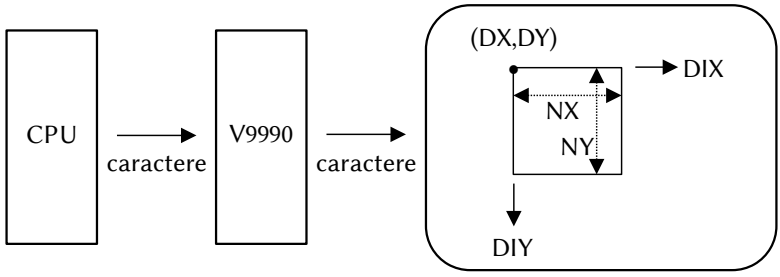
5.8.6.8 – CMMC (Transfer character → VRAM)

In this command, characters are transferred from the CPU to a rectangular area of VRAM. Logical operations on the destination are possible. The character format is 16x16 points and its arrangement is the same as for Kanji-ROM characters. See the illustration below.

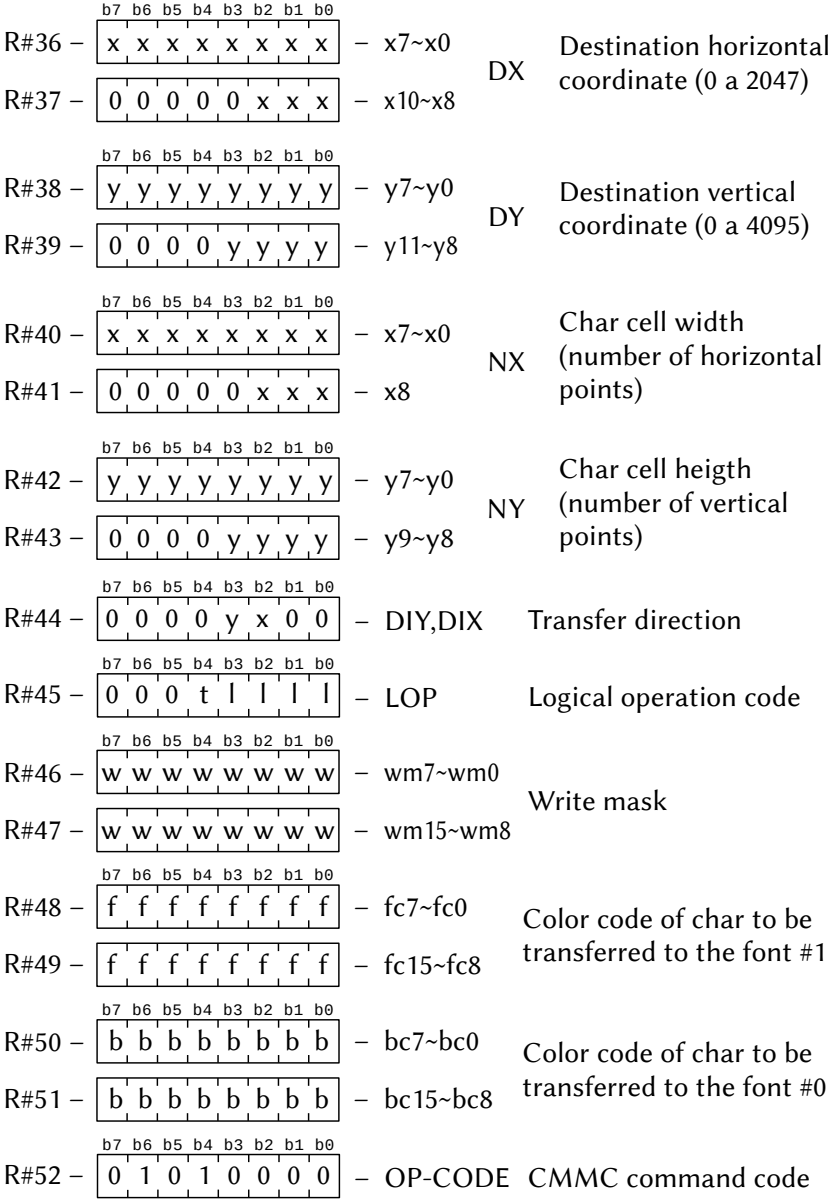
1ºs 8 bytes	2ºs 8 bytes
3ºs 8 bytes	4ºs 8 bytes

The 32 bytes that make up the pattern must be organized into 4 sequences of 8 bytes, as shown on the side, in order to compose a cell of 16 x 16 points.

Just below is the illustration of how the CMMC command is executed.



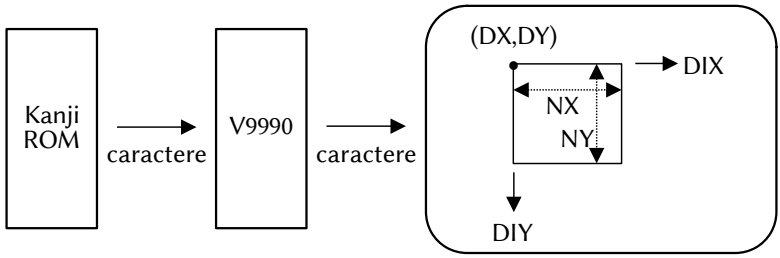
The registers to be loaded are illustrated below.



The data bytes to be transferred must be sent through the command port (P#2).

5.8.6.9 – CMMK (Transfer kanji character → VRAM)

Kanji characters are arranged as shown in the previous CMMC command.



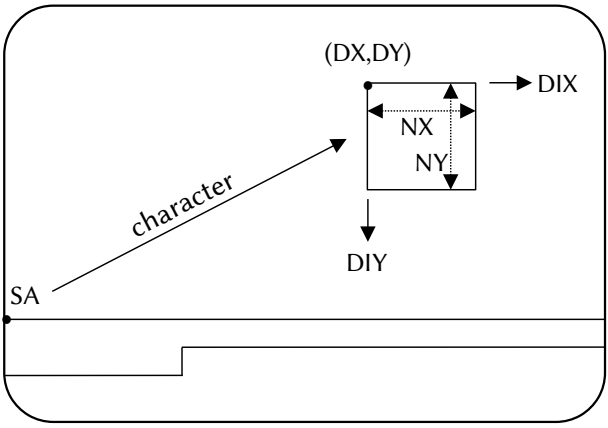
The registers to be loaded are illustrated below.

	b7 b6 b5 b4 b3 b2 b1 b0			
R#32 –	k k k k k k k k	– ka7~ka0		Adress of the Kanji char to be transferred to VRAM
R#34 –	k k k k k k k k	– ka15~ka8		
R#35 –	0 0 0 0 0 0 k k	– ka17~ka16		
R#36 –	x x x x x x x x	– x7~x0	DX	Destination horizontal coordinate (0 a 2047)
R#37 –	0 0 0 0 0 x x x	– x10~x8		
R#38 –	y y y y y y y y	– y7~y0	DY	Destination vertical coordinate (0 a 4095)
R#39 –	0 0 0 0 y y y y	– y11~y8		
R#40 –	x x x x x x x x	– x7~x0	NX	Char cell width (number of horizontal points)
R#41 –	0 0 0 0 0 x x x	– x8		
R#42 –	y y y y y y y y	– y7~y0	NY	Char cell heigth (number of vertical points)
R#43 –	0 0 0 0 y y y y	– y9~y8		

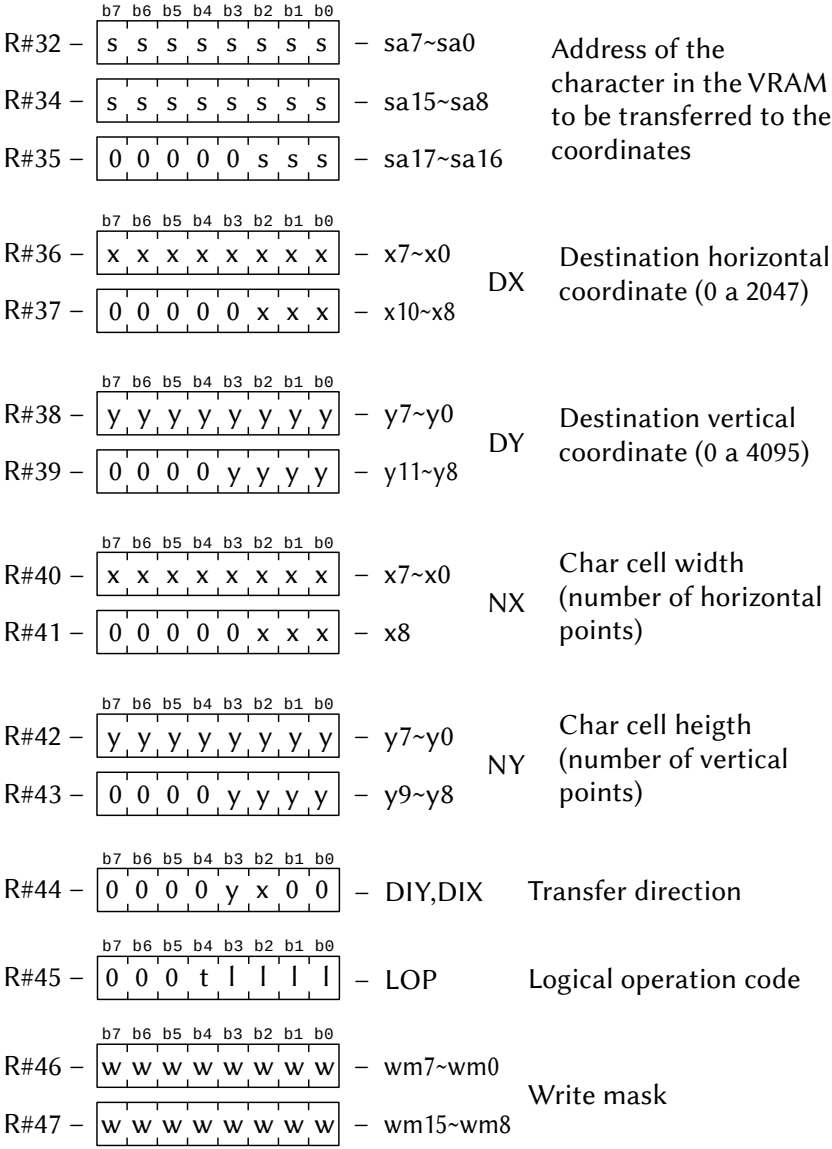
	b7 b6 b5 b4 b3 b2 b1 b0			
R#44 –	0 0 0 0 y x 0 0	–	DIY,DIX	Transfer direction
R#45 –	0 0 0 t l l l l	–	LOP	Logical operation code
R#46 –	w w w w w w w w	–	wm7~wm0	Write mask
R#47 –	w w w w w w w w	–	wm15~wm8	
R#48 –	f f f f f f f f	–	fc7~fc0	Color code of char to be transferred to the font #1
R#49 –	f f f f f f f f	–	fc15~fc8	
R#50 –	b b b b b b b b	–	bc7~bc0	Color code of char to be transferred to the font #0
R#51 –	b b b b b b b b	–	bc15~bc8	
R#52 –	0 1 1 0 0 0 0 0	–	OP-CODE	CMMK command code

5.8.6.10 – CMMM (Transfer character VRAM → VRAM)

In this command, a character is transferred from a linear area of VRAM to a rectangular area of VRAM. Logical operations on the destination are possible. The character has the same format as shown in the CMMC command.



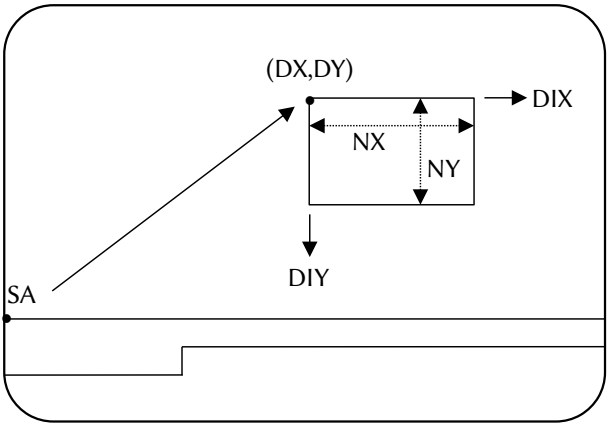
To execute the CMMM command, the following registers must be loaded:



	b7	b6	b5	b4	b3	b2	b1	b0		
R#48 –	f	f	f	f	f	f	f	f	– fc7~fc0	Color code of char to be transferred to the font #1
R#49 –	f	f	f	f	f	f	f	f	– fc15~fc8	
R#50 –	b	b	b	b	b	b	b	b	– bc7~bc0	Color code of char to be transferred to the font #0
R#51 –	b	b	b	b	b	b	b	b	– bc15~bc8	
R#52 –	0	1	1	1	0	0	0	0	– OP-CODE	CMMM command code

5.8.6.11 – BMXL (Byte transfer linear → coordinates)

In this command, bytes of data are transferred from a linear area of VRAM to a rectangular area of VRAM. Logical operations on the destination are possible.



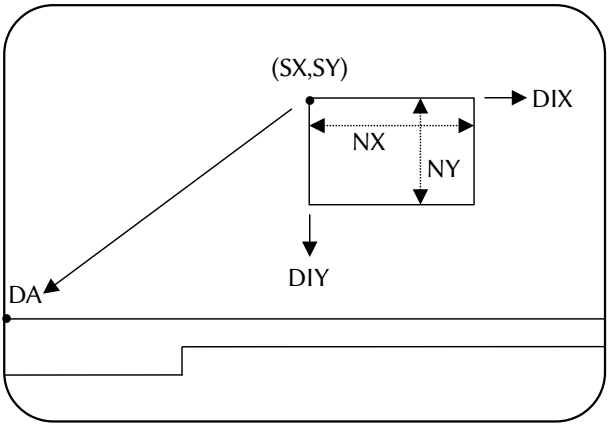
The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0		
R#32 –	s	s	s	s	s	s	s	s	– sa7~sa0	Linear adress on VRAM
R#34 –	s	s	s	s	s	s	s	s	– sa15~sa8	
R#35 –	0	0	0	0	0	s	s	s	– sa18~sa16	

	b7	b6	b5	b4	b3	b2	b1	b0				
R#36 –	x	x	x	x	x	x	x	x	–	x7~x0	DX	Destination horizontal coordinate for transfer (0 a 2047)
R#37 –	0	0	0	0	0	x	x	x	–	x10~x8		
R#38 –	y	y	y	y	y	y	y	y	–	y7~y0	DY	Destination vertical coordinate for transfer (0 a 4095)
R#39 –	0	0	0	0	y	y	y	y	–	y11~y8		
R#40 –	x	x	x	x	x	x	x	x	–	x7~x0	NX	Number of points to transfer in the horizontal direction
R#41 –	0	0	0	0	0	x	x	x	–	x8		
R#42 –	y	y	y	y	y	y	y	y	–	y7~y0	NY	Number of points to transfer in the vertical direction
R#43 –	0	0	0	0	y	y	y	y	–	y9~y8		
R#44 –	0	0	0	0	y	x	0	0	–	DIY,DIX		Transfer direction
R#45 –	0	0	0	t	l	l	l	l	–	LOP		Logical operation code
R#46 –	w	w	w	w	w	w	w	w	–	wm7~wm0		Writing mask
R#47 –	w	w	w	w	w	w	w	w	–	wm15~wm8		
R#52 –	1	0	0	0	0	0	0	0	–	OP-CODE		BMXL command code

5.8.6.12 – BMLX (Byte transfer coordinates → linear)

In the BMLX command, bytes of data are transferred from a rectangular area of VRAM to a linear area of VRAM. Logical operations on the destination are possible.



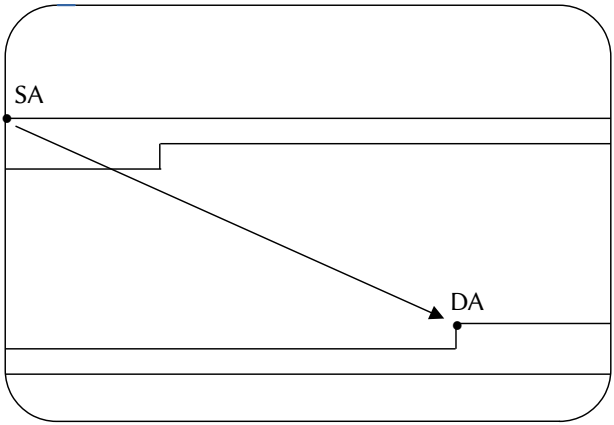
The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0				
R#32 –	x	x	x	x	x	x	x	x	–	x7~x0	DX	Start horizontal coordinate for transfer (0 to 2047)
R#33 –	0	0	0	0	0	x	x	x	–	x10~x8		
R#34 –	y	y	y	y	y	y	y	y	–	y7~y0	DY	Start vertical coordinate for transfer (0 to 4095)
R#35 –	0	0	0	0	y	y	y	y	–	y11~y8		
R#36 –	d	d	d	d	d	d	d	d	–	da7~da0		
R#38 –	d	d	d	d	d	d	d	d	–	da15~da8		VRAM linear adress
R#39 –	0	0	0	0	0	d	d	d	–	da18~da16		
R#40 –	x	x	x	x	x	x	x	x	–	x7~x0	NX	Number of points to transfer in the horizontal direction
R#41 –	0	0	0	0	0	x	x	x	–	x8		
R#42 –	y	y	y	y	y	y	y	y	–	y7~y0	NY	Number of points to transfer in the vertical direction
R#43 –	0	0	0	0	y	y	y	y	–	y9~y8		

	b7	b6	b5	b4	b3	b2	b1	b0			
R#44 –	0	0	0	0	y	x	0	0	–	DIY,DIX	Transfer direction
R#45 –	0	0	0	t	l	l	l	l	–	LOP	Logical operation code
R#46 –	w	w	w	w	w	w	w	w	–	wm7~wm0	Write mask
R#47 –	w	w	w	w	w	w	w	w	–	wm15~wm8	
R#52 –	1	0	0	1	0	0	0	0	–	OP-CODE	BMLX command code

5.8.6.13 – BMLL (Byte transfer linear → linear)

In this command, a block of data from a linear area of VRAM is transferred to another linear area. Logical operations on the destination are possible.



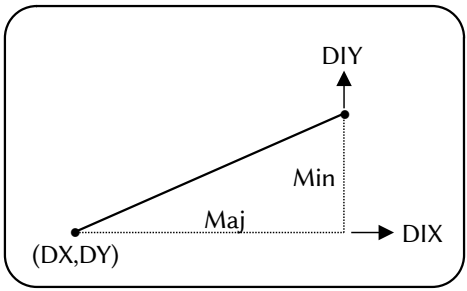
The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0			
R#32 –	s	s	s	s	s	s	s	s	–	sa7~sa0	Source linear adress on VRAM
R#34 –	s	s	s	s	s	s	s	s	–	sa15~sa8	
R#35 –	0	0	0	0	0	s	s	s	–	sa18~sa16	

	b7 b6 b5 b4 b3 b2 b1 b0			
R#36 –	d d d d d d d d	–	da7~da0	Destination linear adress on VRAM
R#38 –	d d d d d d d d	–	da15~da8	
R#39 –	0 0 0 0 0 d d d	–	da18~da16	
R#40 –	n n n n n n n n	–	na7~na0	Number of bytes to transfer
R#42 –	n n n n n n n n	–	na15~na8	
R#43 –	0 0 0 0 0 n n n	–	na18~na16	
R#44 –	0 0 0 0 y x 0 0	–	DIY,DIX	Transfer direction
R#45 –	0 0 0 t l l l l	–	LOP	Logical operation code
R#46 –	w w w w w w w w	–	wm7~wm0	Write mask
R#47 –	w w w w w w w w	–	wm15~wm8	
R#52 –	1 0 1 0 0 0 0 0	–	OP-CODE	BMLL command code

5.8.6.14 – LINE (Draw a line)

This command draws a line between image area coordinates. Logical operations on the destination are possible. The parameters are specified including the coordinate (X,Y) of the start of the line and the horizontal and vertical length to the end point, as shown in the illustration below:



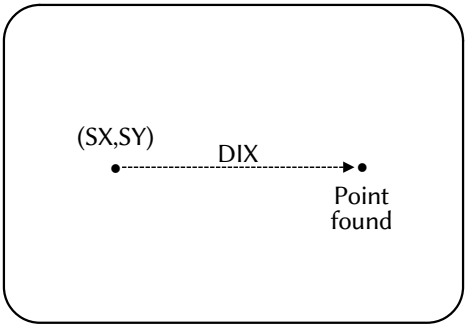
The following registers must be loaded:

R#36 –	b7 b6 b5 b4 b3 b2 b1 b0 x x x x x x x x	– x7~x0	DX	Start horizontal coordinate from which the line will be drawn (0 a 2047)
R#37 –	0 0 0 0 0 x x x	– x10~x8		
R#38 –	b7 b6 b5 b4 b3 b2 b1 b0 y y y y y y y y	– y7~y0	DY	Start vertical coordinate from which the line will be drawn (0 a 4095)
R#39 –	0 0 0 0 y y y y	– y11~y8		
R#40 –	b7 b6 b5 b4 b3 b2 b1 b0 j j j j j j j j	– x7~x0	Maj	Number of points on the largest side of the reference right triangle for drawing
R#41 –	0 0 0 0 0 j j j	– x8		
R#42 –	b7 b6 b5 b4 b3 b2 b1 b0 i i i i i i i i	– y7~y0	Min	Number of points on the smaller side of the reference right triangle for drawing
R#43 –	0 0 0 0 i i i i	– y9~y8		
R#44 –	b7 b6 b5 b4 b3 b2 b1 b0 0 0 0 0 y x 0 m	– DIY,DIX,Maj		Drawing direction*note
R#45 –	b7 b6 b5 b4 b3 b2 b1 b0 0 0 0 t l l l l	– LOP		Logical operation code
R#46 –	b7 b6 b5 b4 b3 b2 b1 b0 w w w w w w w w	– wm7~wm0		Write mask
R#47 –	w w w w w w w w	– wm15~wm8		
R#48 –	b7 b6 b5 b4 b3 b2 b1 b0 f f f f f f f f	– fc7~fc0		Color code of the line do be drawn
R#49 –	f f f f f f f f	– fc15~fc8		
R#52 –	b7 b6 b5 b4 b3 b2 b1 b0 1 0 1 1 0 0 0 0	– OP-CODE		BMXL command code

Note: for Maj=0, the longer side of the reference right triangle is parallel to the X axis (horizontal) and for Maj=1, the longer side is parallel to the Y axis (vertical).

5.8.6.15 – SRCH (Search color code of a point)

This command looks for the existence of a point with a specific color in the image area, always in the horizontal direction, to the left or right. The command ends when the point is found, when a point with the border color is found, or when the limit of the image area is reached.



The following registers must be loaded:

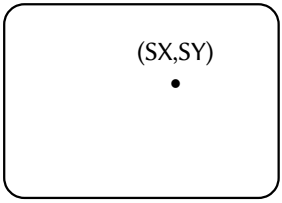
	b7 b6 b5 b4 b3 b2 b1 b0			
R#32 –	x x x x x x x x	– x7~x0	SX	Search start horizontal coordinate (0 a 2047)
R#33 –	0 0 0 0 0 x x x	– x10~x8		
R#34 –	y y y y y y y y	– y7~y0	SY	Search start vertical coordinate (0 a 4095)
R#35 –	0 0 0 0 y y y y	– y11~y8		
R#44 –	0 0 0 0 y x n 0	– DIY,DIX,NEQ*note		Color and search direction
R#45 –	0 0 0 t l l l l	– LOP		Logical operation code
R#48 –	f f f f f f f f	– fc7~fc0		Point color code to be found
R#49 –	f f f f f f f f	– fc15~fc8		

	b7	b6	b5	b4	b3	b2	b1	b0		
R#52 –	1	1	0	0	0	0	0	0	– OP-CODE	SRCH command code
R#53 –	x	x	x	x	x	x	x	x	– x7~x0	Horizontal coordinate of the point, if found
R#54 –	0	0	0	0	0	x	x	x	– x10~x8	

Note: for NEQ=0, the color for detection is specified; for NEQ=1 the color for detection is not specified.

5.8.6.16 – POINT (Read the color code of a point)

This command reads the color code of any point in the image area. The color code read is available on port P#2.

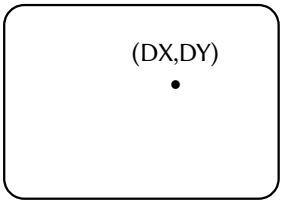


The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0			
R#32 –	x	x	x	x	x	x	x	x	– x7~x0	SX	Point horizontal coordinate (0 a 2047)
R#33 –	0	0	0	0	0	x	x	x	– x10~x8		
R#34 –	y	y	y	y	y	y	y	y	– y7~y0	SY	Point vertical coordinate (0 a 4095)
R#35 –	0	0	0	0	y	y	y	y	– y11~y8		
R#52 –	1	1	0	1	0	0	0	0	– OP-CODE		POINT command code

5.8.6.17 – PSET (Desenha um ponto e avança)

This command draws a point in the image area and then advances coordinates according to the value passed in R#52. Logical operations on the destination are possible.



The following registers must be loaded:

	b7	b6	b5	b4	b3	b2	b1	b0			
R#36 –	x	x	x	x	x	x	x	x	– x7~x0	DX	Horizontal coordinate of the point to be drawn (0 a 2047)
R#37 –	0	0	0	0	0	x	x	x	– x10~x8		
R#38 –	y	y	y	y	y	y	y	y	– y7~y0	DY	Vertical coordinate of the point to be drawn (0 a 4095)
R#39 –	0	0	0	0	y	y	y	y	– y11~y8		
R#45 –	0	0	0	t	l	l	l	l	– LOP		Logical operation code
R#46 –	w	w	w	w	w	w	w	w	– wm7~wm0		Write mask
R#47 –	w	w	w	w	w	w	w	w	– wm15~wm8		
R#48 –	f	f	f	f	f	f	f	f	– fc7~fc0		Color code of the point to be drawn
R#49 –	f	f	f	f	f	f	f	f	– fc15~fc8		
R#52 –	1	1	1	0	AY		AX		– OP-CODE		PSET command code

AX: 00 – DX and DY are used.
01 – No shift.
10 – Shift to the right.
11 – Shift to the left.

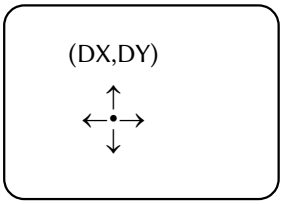
AY: 00 – No shift.
01 – No shitt.
10 – Shift down.
11 – Shift up.

Some precautions must be observed for the execution of this command. When the point is drawn at the current position, registers R#36 to R#39 must not be loaded. After executing the command, the pointer advances according to the values of AY and AX and the next point can be drawn at that position. Officially, the AY and AX values are named as shown in the illustration below:

	b7	b6	b5	b4	b3	b2	b1	b0
R#52 –	1	1	0	1	AYM	AYE	AXM	AXE

5.8.6.18 – ADVN (Advance coordinates)

This command simply advances coordinates in the image area without drawing.



The following registers must be loaded:

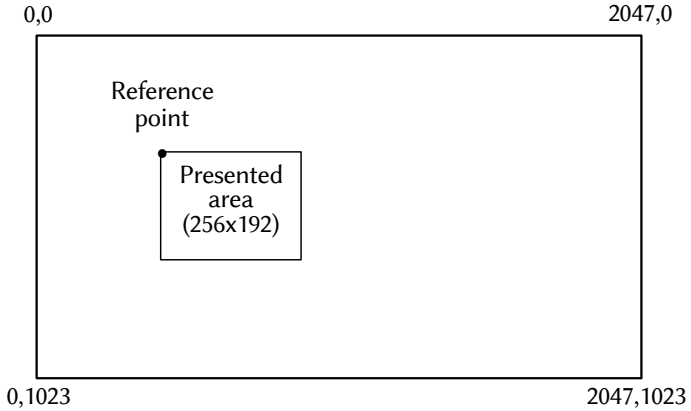
	b7	b6	b5	b4	b3	b2	b1	b0			
R#36 –	x	x	x	x	x	x	x	x	– x7~x0	DX	Horizontal coordinate from which the advance will occur (0 to 2047)
R#37 –	0	0	0	0	0	x	x	x	– x10~x8		
R#38 –	y	y	y	y	y	y	y	y	– y7~y0	DY	Horizontal coordinate from which the advance will occur (0 to 4095)
R#39 –	0	0	0	0	y	y	y	y	– y11~y8		

	b7	b6	b5	b4	b3	b2	b1	b0		
R#52 –	1	1	1	1	AY	AX			– OP-CODE	ADVN command code

The offset values (AY and AX) are the same as those used for the PSET command, described above. As with the PSET command, registers R#36 to R#39 must not be loaded when advancing from the current position.

5.8.7 – Scroll and image area

On the V9990, the image size is normally larger than the area displayed on the screen. For example, for B1 mode with 512 Kbytes of VRAM and 4 colors, we can have an image of up to 2048 x 1024 points. However, only 256 x 212 dots appear on the screen. The upper left point displayed can be defined by the scroll registers. Thus, you can "sweep" the entire image area, and the effect on the screen will be a smooth scroll in all directions. Below is an illustration of the above example.



The reference point can be moved freely around the image area through registers R#17 to R#24, as described below.

	b7	b6	b5	b4	b3	b2	b1	b0	
R#17 –	y	y	y	y	y	y	y	y	– SCAY7~SCAY0
R#18 –	R	R	0	y	y	y	y	y	– SCAY12~SCAY8
R#19 –	0	0	0	0	0	x	x	x	– SCAX2~SCAX0
R#20 –	x	x	x	x	x	x	x	x	– SCAX10~SCAX3
R#21 –	y	y	y	y	y	y	y	y	– SCBY7~SCBY0
R#22 –	A	B	0	0	0	0	0	y	– SCBY8

	b7	b6	b5	b4	b3	b2	b1	b0	
R#23 –	0	0	0	0	0	x	x	x	– SCBX2~SCBX0
R#24 –	0	0	x	x	x	x	x	x	– SCBX8~SCBX3

The SCAY and SCAX values correspond to the coordinates relative to the "A" plane of the P1 mode and all other screen modes. When using 16 bits per point in B2 and B3 modes, the least significant bit (SCAX0) is ignored and the horizontal coordinate is specified in 2-point increments. The SCBY and SCBX values correspond exclusively to the coordinates for the "B" plane of the P1 mode.

The number of vertical points that can be used for scrolling is specified in R#18, as illustrated below.

	b7	b6	b5	b4	b3	b2	b1	b0	
R#18 –	R	R	0	y	y	y	y	y	– SCAY12~SCAY8

00: Scroll across the entire image area.
01: Scroll in 256 lines.
10: Scroll in 512 lines.
11: Meaningless.

Bits b7 and b6 of R#22 can be used to enable or disable the P1 and P2 planes, as illustrated below:

	b7	b6	b5	b4	b3	b2	b1	b0	
R#22 –	A	B	0	0	0	0	0	y	– SCBY8

SDB: If "1" disables plan "B" and sprites.
SDA: If "1" disables plan "A" and sprites.

5.8.8 – V9990 Registers description

5.8.8.1 – VRAM write adress (write only)

R#0 –	A7	A6	A5	A4	A3	A2	A1	A0	VRAM write adress (low)
R#1 –	A15	A14	A13	A12	A11	A10	A9	A8	VRAM write adress (mid)

R#2 –

AI	0	0	0	0	A18	A17	A16

VRAM write adress (high)
AI = 0 ⟶ autoincrement

5.8.8.2 – VRAM read adress (write only)

R#3 –

A7	A6	A5	A4	A3	A2	A1	A0

VRAM read adress (low)

R#4 –

A15	A14	A13	A12	A11	A10	A9	A8

VRAM read adress (mid)

R#5 –

AI	0	0	0	0	A18	A17	A16

VRAM read adress (high)
AI = 0 ⟶ autoincrement

5.8.8.3 – Screen mode (read/write)

R#6 –

b7	b6	b5	b4	b3	b2	b1	b0
DSPM		DCKM		XIMM		CLRM	

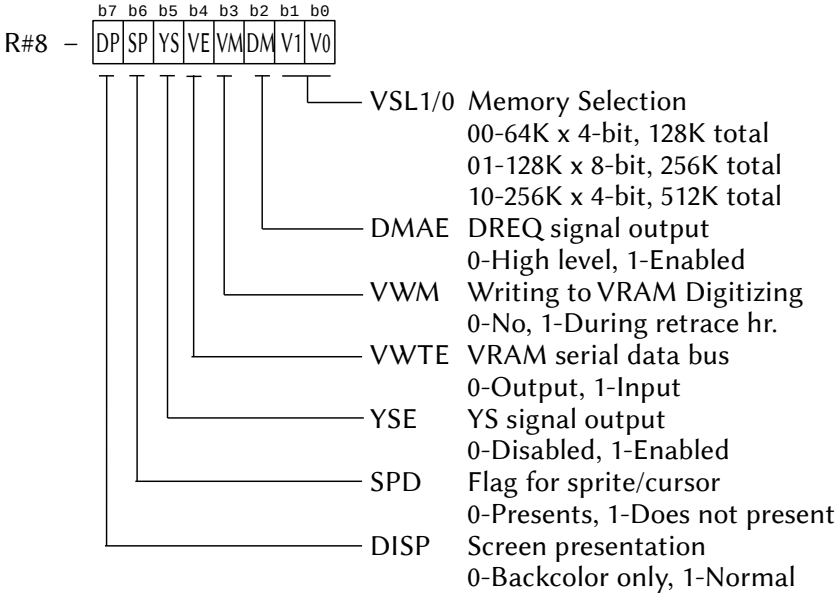
CLRM	00-2bits/pixel	10-8bits/pixel
	01-4bits/pixel	11-16bits/pixel
XIMM	00-Y=256pixels	10-Y=1024pixels
	01-Y=512pixels	11-Y=2048pixels
DCKM	00-XTAL=1/4	10-XTAL=1/1
	01-XTAL=1/2	11-N/A
DSPM	00-P1 mode	10-Bit Map
	01-P2 mode	11-Stand-by

R#7 –

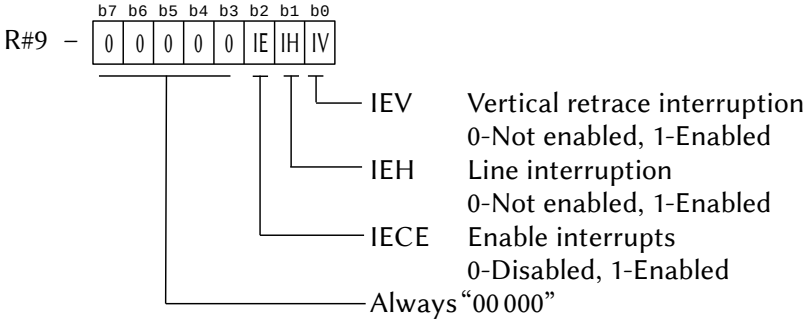
b7	b6	b5	b4	b3	b2	b1	b0
0	CM	S1	S2	PL	EO	IL	HS

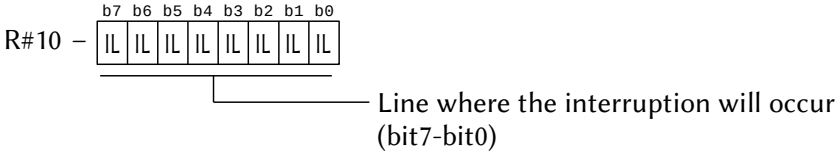
HSCN	0-other modes	1-modes B5-B6
IL	0-do not interl.	1-interlaced
EO	0-Y=normal	1-Y=doubled
PAL	0-NTSC	1-PAL
SM	0-1H=fsc/228	1-1H=fsc/227.5
SM1	0-262 lines	1-263 lines
C25M	0-other modes	1-mode B6
always 0		

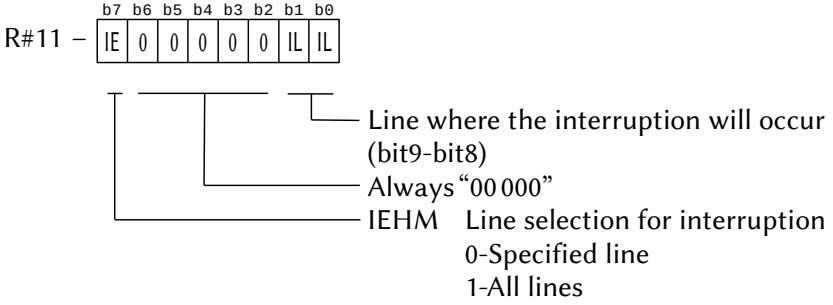
5.8.8.4 – System control (read/write)

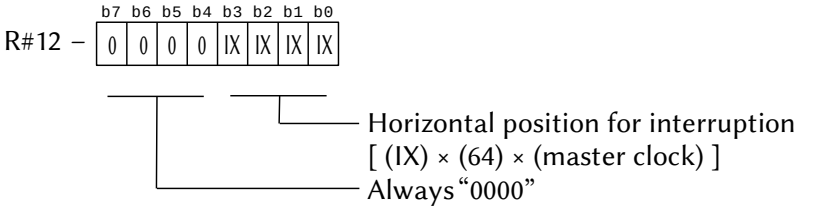


5.8.8.5 – Interruption control (read/write)

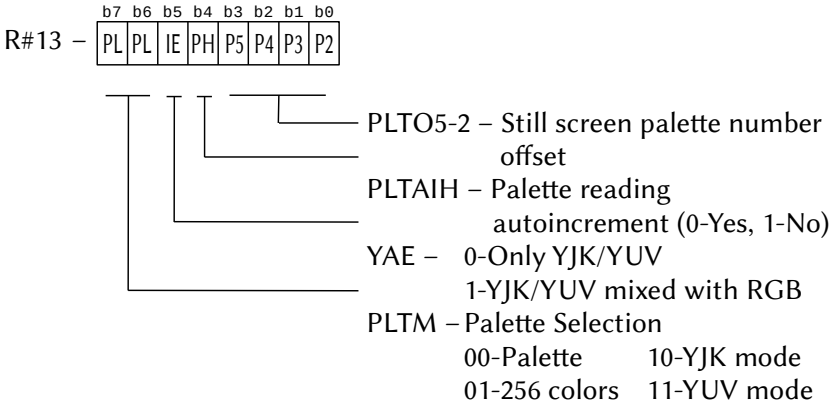


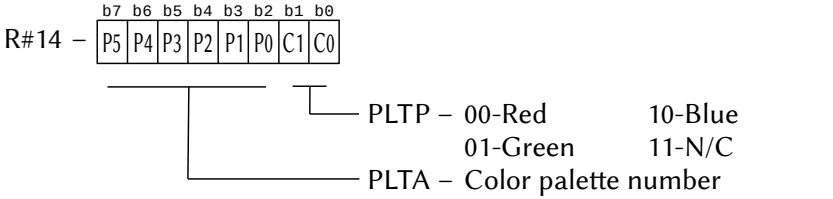




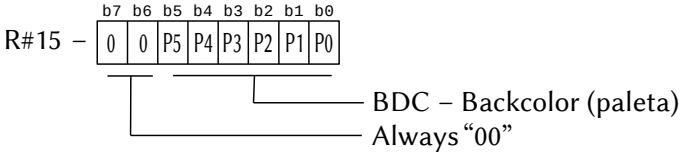


5.8.8.6 – Palette control (write only)

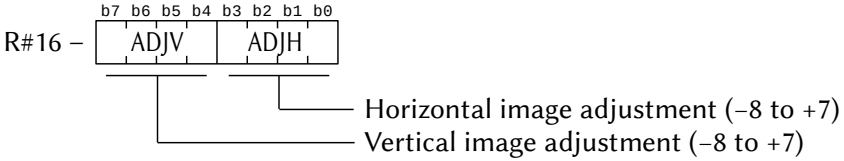




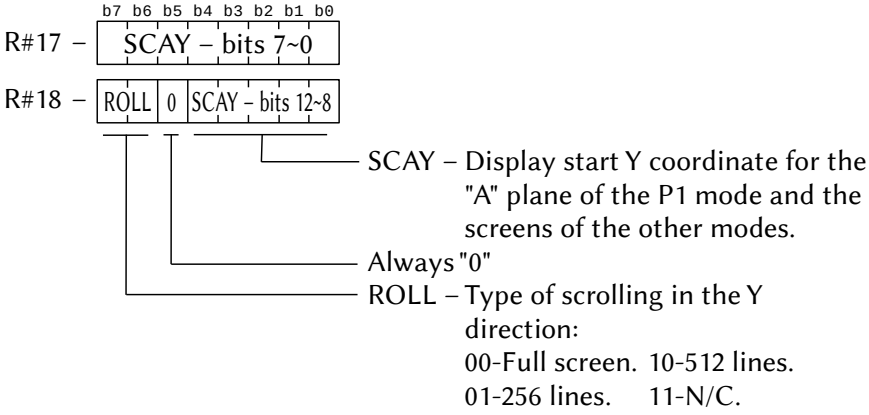
5.8.8.7 – Back color (read/write)

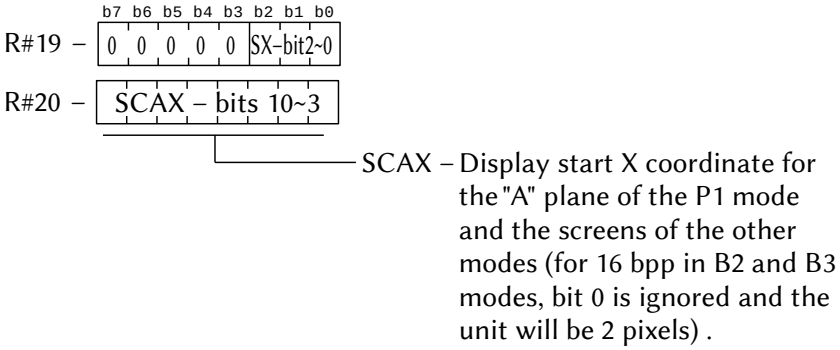


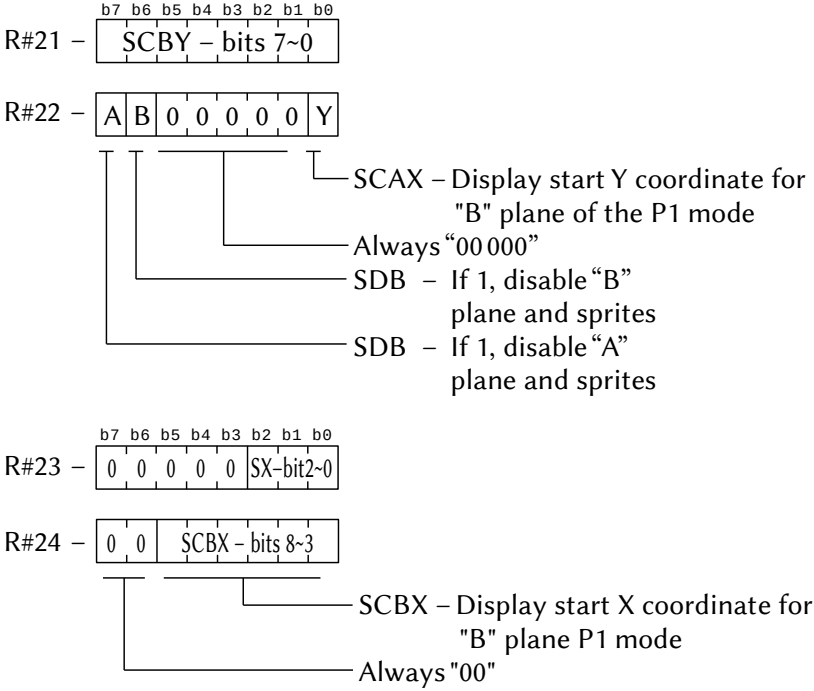
5.8.8.8 – Screen adjust (read/write)



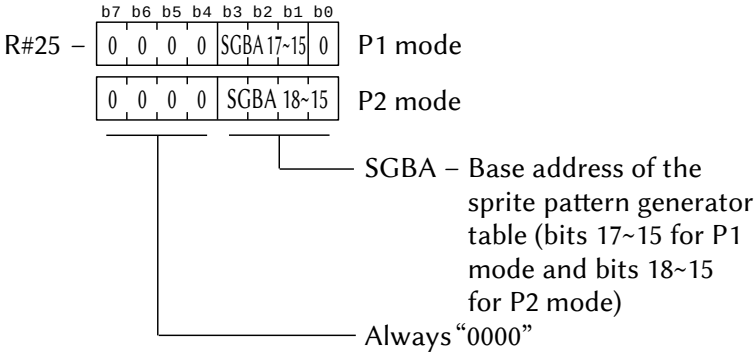
5.8.8.9 – Scroll control (read/write)



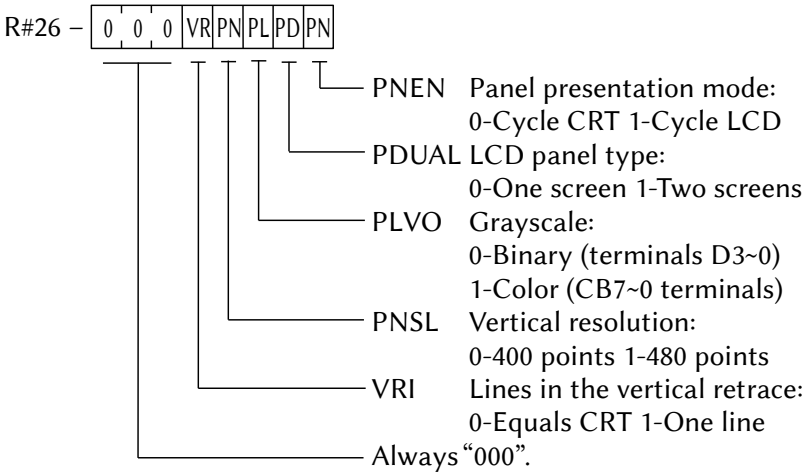




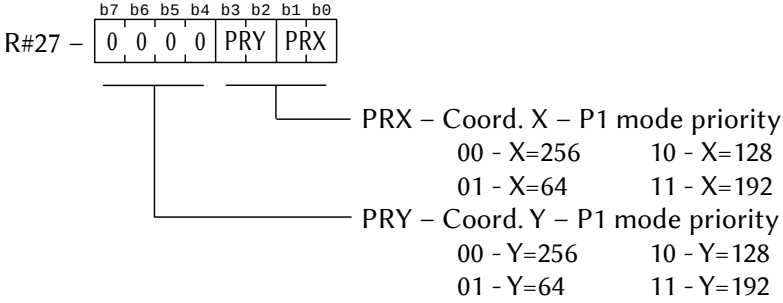
5.8.8.10 – Sprites table adress (read/write)



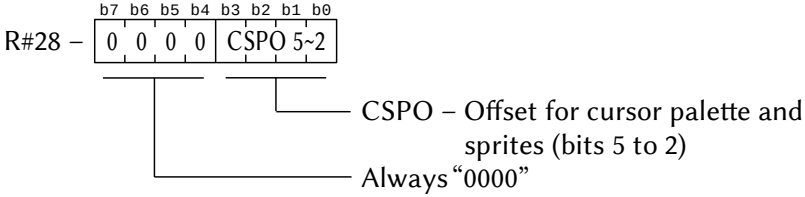
5.8.8.11 – LCD panel control (read/write)



5.8.8.12 – Priority control (read/write)



5.8.8.13 – Sprite/cursor palette offset (write only)



Registers R#19 to R#31 do not exist.

5.8.8.14 – VDP commands – Source: linear adress and Kanji-ROM (write only)

	b7	b6	b5	b4	b3	b2	b1	b0
R#32 –	SX7	SX6	SX5	SX4	SX3	SX2	SX1	SX0
	SA7	SA6	SA5	SA4	SA3	SA2	SA1	SA0
	KA7	KA6	KA5	KA4	KA3	KA2	KA1	KA0
R#33 –	0	0	0	0	0	SX10	SX9	SX8
R#34 –	SY7	SY6	SY5	SY4	SY3	SY2	SY1	SY0
	SA15	SA14	SA13	SA12	SA11	SA10	SA9	SA8
	KA15	KA14	KA13	KA12	KA11	KA10	KA9	KA8
R#35 –	0	0	0	0	SY11	SY10	SY9	SY8
	0	0	0	0	0	SA18	SA17	SA16
	0	0	0	0	0	0	KA17	KA16

SX10 ~ SX0 ⟶ Horizontal coordinate of the source

SY11 ~ SY0 ⟶ Vertical coordinate of the font

SA18 ~ SA0 ⟶ Linear address of the source

KA18 ~ KA0 ⟶ Kanji-ROM address

5.8.8.15 – VDP commands – Destination: XY Coordinate / Linear Address (write only)

	b7	b6	b5	b4	b3	b2	b1	b0
R#36 –	DX7	DX6	DX5	DX4	DX3	DX2	DX1	DX0
	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
R#37 –	0	0	0	0	0	DX10	DX9	DX8
R#38 –	DY7	DY6	DY5	DY4	DY3	DY2	DY1	DY0
	DA15	DA14	DA13	DA12	DA11	DA10	DA9	DA8

R#39 –	0	0	0	0	DY11	DY10	DY9	DY8
	0	0	0	0	0	DA18	DA17	DA16

DX10 ~ DX0 ⟶ Horizontal coordinate of the destination

DY11 ~ DA0 ⟶ Vertical coordinate of the destination

DA18 ~ DA0 ⟶ Linear address of destinatio

5.8.8.16 – VDP commands – Size: (Number of points to transfer XY / Linear / Major/minor line) (write only)

	b7	b6	b5	b4	b3	b2	b1	b0
R#40 –	NX7	NX6	NX5	NX4	NX3	NX2	NX1	NX0
	NA7	NA6	NA5	NA4	NA3	NA2	NA1	NA0
	MJ7	MJ6	MJ5	MJ4	MJ3	MJ2	MJ1	MJ0
R#41 –	0	0	0	0	0	NX10	NX9	NX8
	0	0	0	0	MJ11	MJ10	MJ9	MJ8
R#42 –	NY7	NY6	NY5	NY4	NY3	NY2	NY1	NY0
	NA15	NA14	NA13	NA12	NA11	NA10	NA9	NA8
	MI7	MI6	MI5	MI4	MI3	MI2	MI1	MI0
R#43 –	0	0	0	0	NY11	NY10	NY9	NY8
	0	0	0	0	0	NA18	NA17	NA16
	0	0	0	0	MI11	MI10	MI9	MI8

NX10 ~ NX0 ⟶ Number of points to transfer horizontally

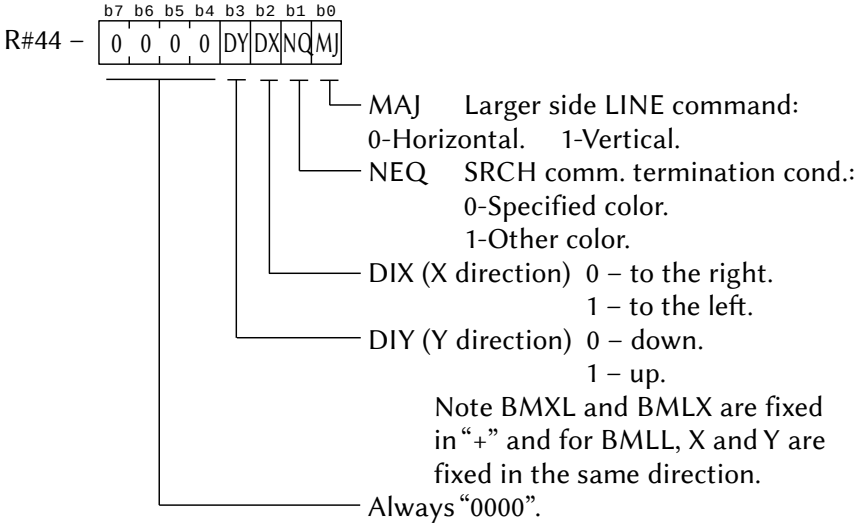
NY11 ~ NY0 ⟶ Number of points to transfer vertically

NA18 ~ NA0 ⟶ Number of bytes to transfer

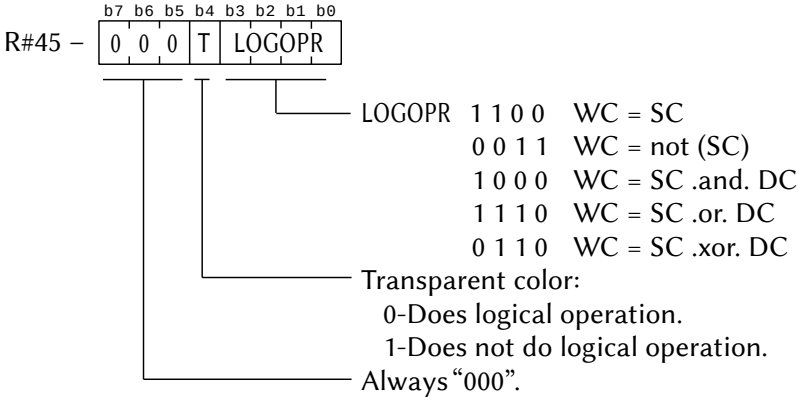
MJ11 ~ MJ0 ⟶ Longer side of the line (LINE command)

MI11 ~ MI0 ⟶ Short side of the line (LINE command)

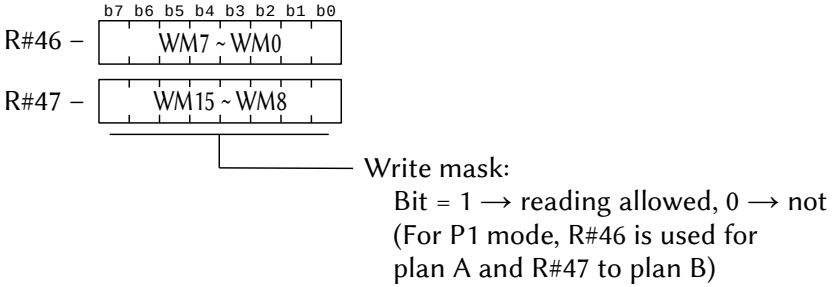
5.8.8.17 – VDP commands – Arguments (read only)



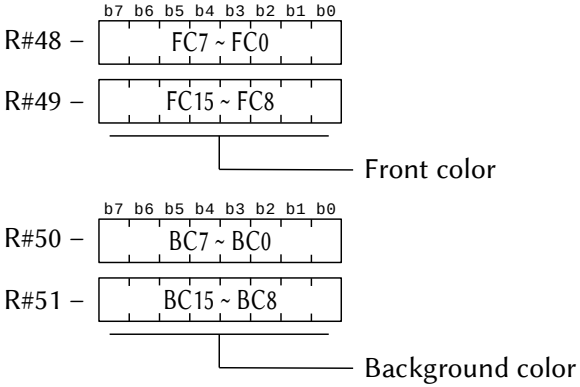
5.8.8.18 – VDP commands – Logical operation (write only)



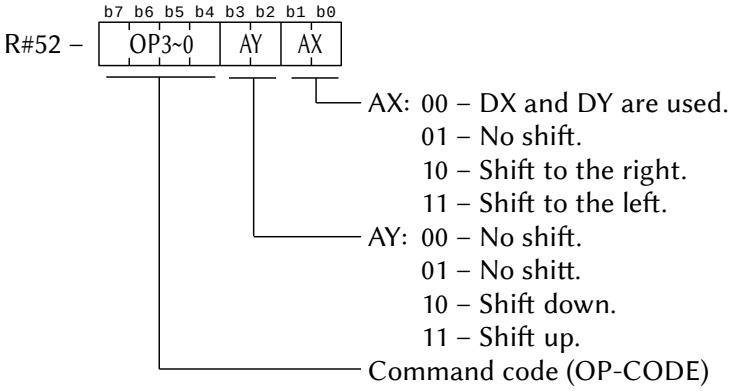
5.8.8.19 – VDP commands – Write mask (write only)



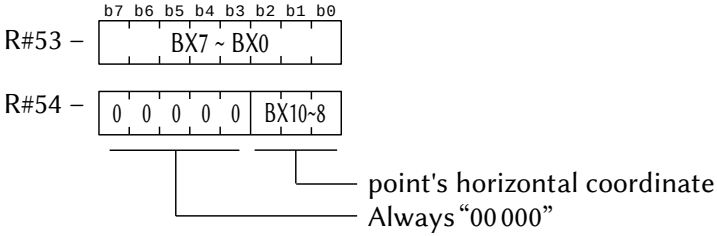
5.8.8.20 – VDP commands – Front/back color (write only)



5.8.8.21 – VDP commands – Operation control (write only)



5.8.8.22 – VDP commands – Horizontal coordinate (read only)



Chapter 6
AUDIO GENERATORS

The MSX computers have many options for generating sounds, ranging from simple AM generators to sophisticated digitizers. These options are listed below:

1 – PSG (MSX1 default)
2 – 1-bit I/O port (MSX1 default)
3 – MSX-Music - OPLL (optional MSX2, standard MSX2+)
4 – PCM (standard MSX turbo R)
5 – MSX-Audio (optional)
6 – SCC (for some Konami games)
7 – OPL4 (optional, only in expansion cartridge)
8 – Covox (optional)

6.1 – PSG

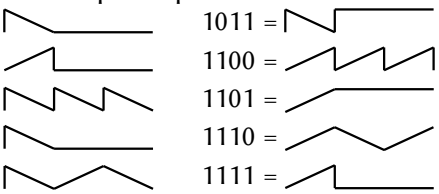
PSG stands for "Programmable Sound Generator". PSG can generate up to 3 voices in up to 4096 scales (equivalent to 8 octaves) and 16 independent volume levels for each voice. Additionally, it has a white noise generator (hiss) that must be present in one of the 3 voices. The chip responsible is the AY-3-8910A.

PSG has 16 8-bit registers for generating sounds. They are described in the table below. Registers 14 and 15 are used for I/O operations and not for specifying sounds.

6.1.1 – Description of the registers

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short description
R#0 R#1	7 •	6 •	5 •	4 •	3 11	2 10	1 9	0 8	Channel "A" frequency 111860,87 / F_num (b11~b0)
R#2 R#3	7 •	6 •	5 •	4 •	3 11	2 10	1 9	0 8	Channel "B" frequency 111860,87 / F_num (b11~b0)
R#4 R#5	7 •	6 •	5 •	4 •	3 11	2 10	1 9	0 8	Channel "C" frequency 111860,87 / F_num (b11~b0)
R#6	•	•	•	4	3	2	1	0	White noise frequency 111860,87 / F_num (b4~b0)

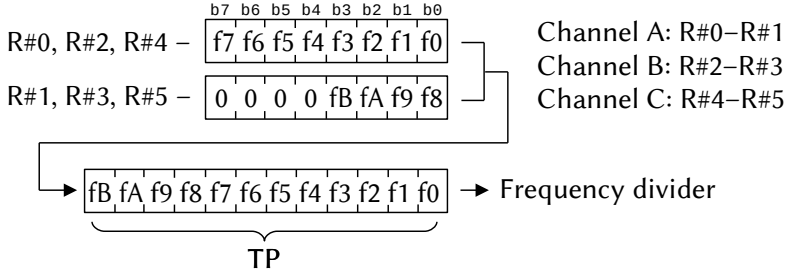
R#7	ioB	ioA	rC	rB	rA	tC	tB	tA	Enable/disable sounds
	b0~b2		Enable/disable tones (0=enable)						
	b5~b4		Enable/disable white noise (0=enable)						
	b6		Configures "A" I/O port (0=in, 1=out)						
	b7		Configures "B" I/O port (0=in, 1=out)						
R#8	•	•	•	m	v	v	v	v	Volume of channel A
R#9	•	•	•	m	v	v	v	v	Volume of channel B
R#10	•	•	•	m	v	v	v	v	Volume of channel C
	b7~b5		Not used (always "000")						
	b4		0=Not use envelope; 1=Use envelope						
	b3~b0		0000=Minimum volume; 1111=Maximum volume						
R#11	7	6	5	4	3	2	1	0	Envelope frequency
R#12	15	14	13	12	11	10	9	8	6983,3 / F_num (b15~b0)
R#13	•	•	•	•	e	e	e	e	Envelope shape
	b7~b4		Not used (always "0000")						
	b3~b0		Defines envelope shape:						
R#14	a7	a6	a5	a4	a3	a2	a1	a0	Read/write "A" I/O port
R#15	b7	b6	b5	b4	b3	b2	b1	b0	Read/write "B" I/O port



The operation of the PSG registers is very simple. Just write the appropriate values for the sound to be generated. The registers are described in detail below.

6.1.1.1 – Frequency specitication

The center frequency used by PSG to drive the frequency divider is 111860.78 Hz. Thus, to obtain the tone generator output frequency, simply divide 111860.78 by the TP value, represented by the register pairs R#0-R#1, R#2-R#3 and R#4-R#5.

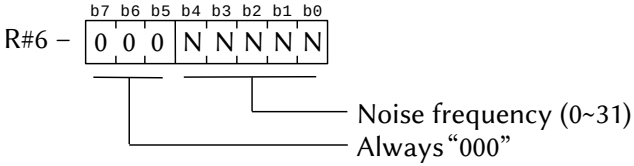


The values of each TP register for the 8 octaves of the three tone generators with a center A note of 440 Hz are listed below.

Octave ⟶		1ª	2ª	3ª	4ª	5ª	6ª	7ª	8ª
Dó	C	D5D	6AF	357	1AC	0D6	06B	035	01B
	C#	C9C	64E	327	194	0CA	085	032	019
Ré	D	BE7	5F4	2FA	17D	0BE	05F	030	018
	D#	B3C	59E	2CF	168	0B4	05A	02D	016
Mi	E	A9B	54E	2A7	153	0AA	055	02A	015
Fá	F	A02	501	281	140	0A0	050	028	014
	F#	973	4BA	25D	12E	097	04C	026	013
Sol	G	8EB	476	23B	11D	08F	047	024	012
	G#	86B	436	21B	10D	087	043	022	011
Lá	A	7F2	3F9	1FD	0FE	07F	040	020	010
	A#	780	3C0	1E0	0F0	078	03C	01E	00F
Si	B	714	38A	1C5	0E3	071	039	01C	00E

6.1.1.2 – White noise generator

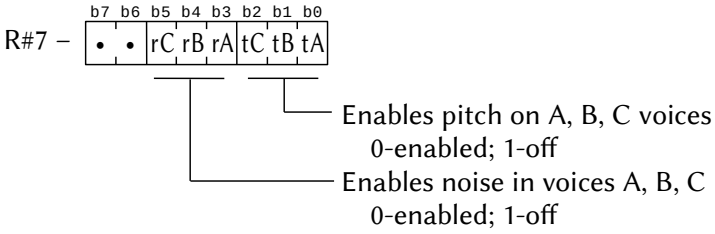
The white noise generator is useful for generating explosion and other sounds. The PSG generates the noise through one of three tone voices and its frequency is specified in register R#6.



The central frequency used by the noise generator is also 111860.78 Hz. As the value of R#6 can vary from 1 to 31, the frequency of the noise varies from 3.6 KHz to 111.8 KHz (dividing 111860.78 by the value contained in R#6).

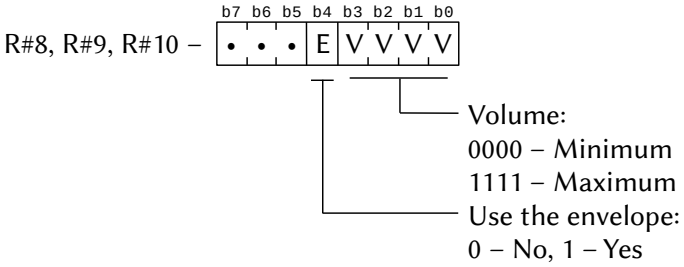
6.1.1.3 – Mixing the sounds

Register R#7 is used to enable or disable the pitch or noise of each of the three voices. Bits b7 and b6 control I/O operations and do not interfere with sound generation.



6.1.1.4 – Volume adjustement

Registers R#8 to R#10 are used to specify the volume of each of the three voices and can range from 0 (minimum volume) to 15 (maximum volume), or give volume control to the envelope generator (R# 8 – channel A; R#9 – channel B; R#10 – channel C).

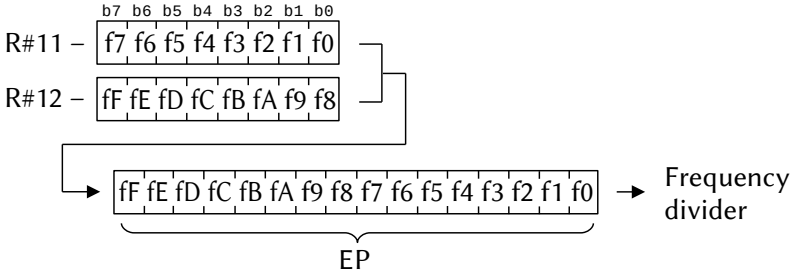


When the E bit is 0, the volume is specified by the V bits. When the E bit is 1, the volume is controlled by the envelope generator and the V bits are solemnly ignored.

6.1.1.5 – Envelope frequency

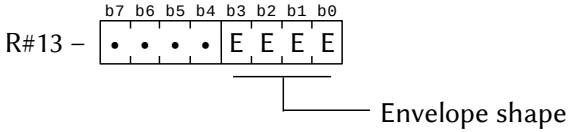
Registers R#11 and R#12 are used as a frequency divider for the envelope generator. All bits are valid. The center frequency used by the

envelope generator to drive the frequency divider is 6983.3 Hz; therefore, the envelope frequency can range from 6983.3 Hz to 0.107 Hz.

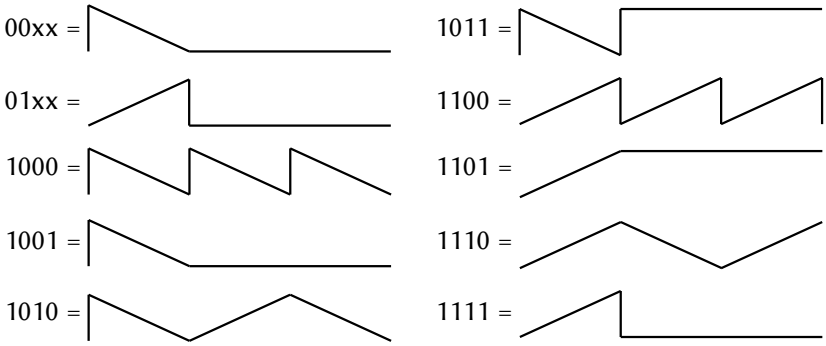


6.1.1.6 – Envelope shape

The shape of the envelope is specified in the first four bits of R#13, as illustrated below.



The possible shapes of the envelope are illustrated below.



6.1.2 – Access to PSG

Access to PSG is via I/O ports. However, the MSX standard dictates that all PSG accesses must be done through BIOS routines to avoid synchronization problems.

The BIOS routines for accessing the PSG are as follows:

GICINI (0090H / Main)
Function: Initializes the PSG and sets the initial values for the PLAY command.
Input: None.
Output: None.
Registers: All.

WRTPSG (0093H / Main)
Function: Writes a byte of data to a PSG register.
Input: A – PSG registrar number.
E – Byte of data to be written.
Output: None.
Registers: None.

RDPSG (0096H / Main)
Function: Reads the contents of a PSG register.
Input: A – PSG registrar number.
Output: A – Byte read.
Registers: None.

Direct access is also possible. There are three ports intended for access to PSG. These ports are:

Port A0H: address port (0 to 15)
Port A1H: data write port (0 to 255)
Port A2H: data read port (0 to 255)

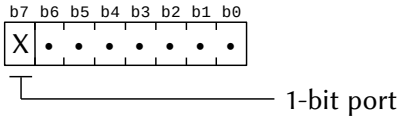
Accessing these ports is very simple: just send the number of the register to be accessed (0 to 15) through the address port (A0H). Then, there may be repeated accesses to the same register through ports A1H (write) or A2H (read).

6.2 – SOUND GENERATION THROUGH 1-bit PORT

The MSX has another standard method for generating sounds, albeit a very limited one. These are generated by repeatedly turning a 1-bit I/O port on and off. The "click" sound of the keys is generated in this way. Accessing this bit can be done through a BIOS routine:

CHGSND (0135H / Main)
Function: Changes the state of the sound-generating 1-bit port.
Input: A = 0 turns the bit off, another value turns the bit on.
Output: None.
Registers: AF.

Here, direct access is also possible. The 1-bit port is accessed by bit b7 of the PPI port C (I/O port AAH). In this case, care must be taken not to modify the other bits of that port.



By repeatedly turning this bit on and off, various types of sound effects can be generated, including rough reproduction of the human voice.

6.3 – THE OPLL (MSX-MUSIC)

MSX-Music (FM-OPLL) can generate 9 simultaneous voices or 6 voices plus 5 rhythm parts. Its sound quality is far superior to that of PSG. The FM generator is also known as OPLL (FM OPerator type LL). The chip is the YM2413 and appeared as a cheap alternative to MSX-Audio, and is standard from MSX2+ onwards.

The FM-OPLL has a 9-bit DA converter and instrument definition ROM built into the chip, incorporating 15 instruments and 5 rhythm tones, as well as all the tones used for CAPTAIN and TELETEXT. It also has a recorder that allows the creation of sound effects and original instruments.

6.3.1 – FM synthesis description

The FM-OPLL uses harmonics generated by frequency modulation to synthesize musical sounds, called “FM synthesis”. This type of synthesis is expressed by 3 parameters:

$$F = A \sin(\omega c t + I \sin \omega m t)$$

Where A is the output amplitude, I is the modulation index, ωc and ωm the angular frequencies of the carrier and modulator, respectively. Equation 1 can alternatively be expressed as below:

$$A\,[J0\,(I)\sin\omega ct + J1\,(I)\,(\sin(\omega c + \omega m)t - \sin(\omega c - \omega m)t\,) + \\ + J2\,(I)\,(\sin(\omega c + 2\omega m)t + \sin(\omega c - 2\omega m)t + \ldots]$$

Where Jn (I) is the nth order of the first-type Bessel function. The amplitude of each harmonic component is expressed as the Bessel function of the modulation index. FM synthesized sounds can be used to obtain specific musical sounds or various types of sound effects. Serial sounds, however, cannot be obtained since the distribution of harmonics is not uniform. The "feedback" or feedback method is used to solve the problem. It is characterized by the following equation:

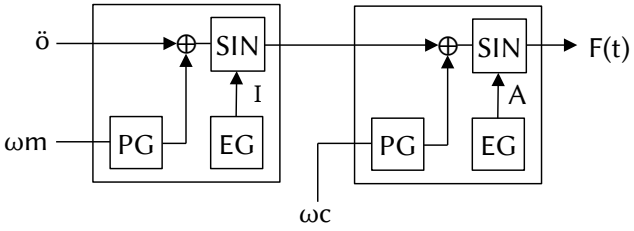
$$F = A \sin(\omega ct + \beta F)$$

Where β is the feedback rate. The harmonic spectrum produced has a sawtooth waveform.

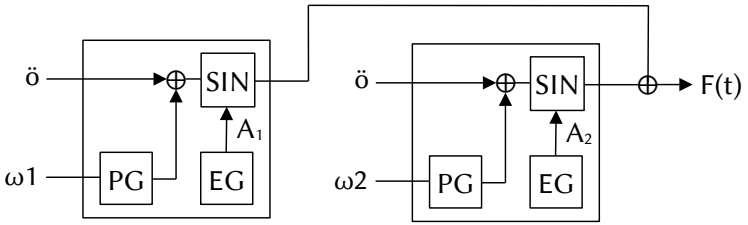
Three blocks are used to synthesize FM sounds:

1. Phase generator (PG) to generate ωt;
2. Envelopment generator (EG) to generate the amplitude A and the modulation index (I);
3. SIN table (sine).

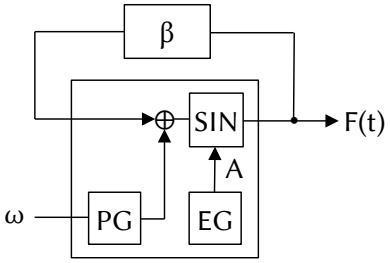
FM synthesis can be performed as shown in the figure below, in cells that combine the functions of the three blocks. It is only necessary to define the frequency and envelope parameters. The FM generation through individual cells is illustrated below.



$$F(t) = A \sin(\omega ct + I \sin \omega mt)$$



$$F(t) = A_1 \sin \omega_1 t + A_2 \sin \omega_2 t$$



$$F(t) = A \sin(\omega t + \beta.F(t))$$

6.3.2 –Map of OPLL registers

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H	AM	VIB	EGT	KSR	Multiple				→ (m) – Modulated wave
$01H	AM	VIB	EGT	KSR	Multiple				→ (c) – Carrier wave
	b7 b6 b5 b4 b0~b3	AM: 0=tremolo off; 1=tremolo on VIB: 0=vibrato off; 1=vibrato on EGT: 0=decaying sound; 1=sustained sound KSR: 0=same level; 1=frequency attenuation (KSL) Multiplication factor (0=1/2, 1=1, 2=2, ..., 15=15)							
$02H	KSL(m)		Total level modul. (m)						Instrument definition
	b6~b7 b6~b0	KSL (m): 00=0dB/oct, 01=1,5dB, 10=3dB, 11=6dB Total level: b0=0,75dB, b1=1,5dB, ..., b5=24dB							

$03H	KSL(c)	•	DC	DM	Feedback					Instrument definition
	b6~b7 b5 b4 b3 b2~b0	KSL (c): 00=0dB/oct, 01=1,5dB, 10=3dB, 11=6dB Not used (always 0) DC: 0=integer carrier wave, 1=half-wave DM: 0=integer modulated wavw, 1=half-wave Feedback: (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)							
$04H	Attack (m)				Decay (m)					Attack (0dB a 48dB ⟶ min. 0,14 mS; max 1730 mS)
$05H	Attack (c)				Decay (c)					Decay (0dB a 48dB ⟶ min. 1,27 mS; max 20 926 mS)
$06H	Sustain (m)				Release (m)					Sustain (b7=24dB, b6=12dB, b5=6dB, b4=3dB)
$07H	Sustain (c)				Release (c)					Release (0dB a 48dB ⟶ min. 1,27 mS; max 28 926 mS)
$0EH	•	•	R	BD	SD	TOM	TCY	HH		Rhythm control
	b7~b6 b5 b0~b4	Not used (always "00") 0=melody mode; 1=rhythm mode 0=rhythm instrument off; 1=on BD–Bass Drum SD–Snare Drum TOM–Tom tom TCY–Top cymbal HH–Hi-hat								
$0FH	Test register									OPLL test
$10H ⋮ $18H	Frequency LSB (8 bits)									Registers used to select the frequencies of the tone generator
$20H ⋮ $28H	•	•	Sustain	Key	Octave			Freq.		Frequency MSB 1 bit Octave Key / Sustain on/off
	b7~b6 b5 b4 b3~b1	Not used (always"00") 0=No sustain; 1=Release Rate will decay gradually 0=key off; 1=key on Octave setting. The fourth is 011.								

<table>
<tr><td></td><td>b0</td><td colspan="3">MSB Frequency 1 bit. The A 440 Hz central note is obtained with b0=1 and $10H~18H=00 100 000</td></tr>
<tr><td rowspan="2">$30H
⋮
$38H</td><td colspan="2">Instruments</td><td>Volume</td><td>Registers used to select the instruments and volume</td></tr>
<tr><td>b7~b4

b3~b0</td><td colspan="3">Instruments:
0000 – To be defined 1000 – Organ
0001 – Violin 1001 – Horn
0010 – Guitar 1010 – Sinthesizer
0011 – Piano 1011 – Harpsichord
0100 – Flute 1100 – Vibraphone
0101 – Clarinet 1101 – Sinthesizer Bass
0110 – Oboe 1110 – Acoustic Bass
0111 – Trumpet 1111 – Electric Guitar
Volume (0000=minimum; 1111=maximum)</td></tr>
<tr><td colspan="5">Register map for drum mode ($0EH, b5=1)</td></tr>
<tr><td>$36H</td><td>•</td><td>•</td><td>•</td><td>•</td><td>BD volume</td><td rowspan="3">Volume registers for the rhythm sounds</td></tr>
<tr><td>$37H</td><td colspan="4">HH volume</td><td>SD volume</td></tr>
<tr><td>$38H</td><td colspan="4">TOM volume</td><td>TCY volume</td></tr>
</table>

Register	Bit	Content
00: (m) 01: (c)	b7 b6 b5 b4 b0~b3	Amplitude modulation on/off (tremolo) Frequency modulation on/off (vibrato) 0=percussive tone; 1=non-percussive tone Key scale rate Multi-sample control and harmonics
02: (m) 03: (c)	b6~b7	“Key Scale” level
02:	b0~b5	Total level of modulation
03:	b4(c) b3(m) b0~b2(m)	Carrier waveform distortion Modulating waveform distortion FM feedback constant
04: (m)	b4~b7	Envelope attack level control

05: (c)	b0~b3	Envelope decay level control
06: (m) 07: (c)	b4~b7 b0~b3	Decay indication; sustain level Control of the release level of the wrap
0E:	b5 b0~b4	1-drum mode; 0-melody mode Turn on/off rhythm sounds
0F:	b5~b4 b3 b2 b1 b0	Not used (always "0000") 1=Updates the LFO every sample 1=Keep but set the waveform to "0". 1=Keep but set LFO phase to "0". 1=Sets the EG to full volume.
10~18:	b0~b7	Frequency (LSB 8 bits)
20~28:	b5 b4 b1~b3 b0	Turn sustain on/off Turn key on/off Select the octave Frequency (MSB 1 bit)
30~30:	b4~b7 b0~b3	Instrument selection Volume control

The OPLL has 15 pre-programmed instruments internally and one more that can be defined by the user, in addition to five rhythm parts. The instrument that can be programmed is number 0 (original). The instruments available are the following:

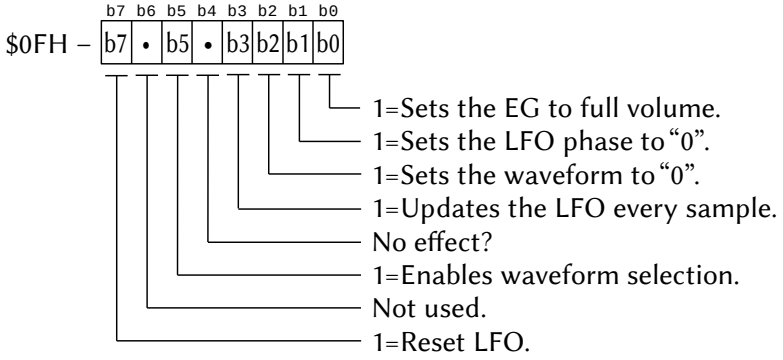
0: Original	8: Organ	Drums:
1: Violin	9: Piston	
2: Guitar	10: Synthesizer	BD: bass drum
3: Piano	11: Harpsichord	SD: snare drum
4: Flute	12: Vibraphone	TOM: tom-tom
5: Clarinet	13: Electric bass	TCY: top cymbal
6: Oboe	14: Acoustic bass	HH: high hat
7: Trumpet	15: Electric guitar	

6.3.3 – Description of registers

This section describes in detail the various registers of the YM2413 and how they work.

6.3.3.1 – Test register

The $0FH register is the test register. Normally its value is 0. When in test mode, an internal 4-bit DAC can be accessed. The way to access it is described later.



Bit 0: If "1", the envelope generators output is set to "0" (full volume) for both the modulator and the carrier. The envelopes continues to run.

Bit 1: If "1", keeps the LFO phase at zero. This stops, disables and resets the tremolo and vibrato LFO.

Bit 2: If "1", keep the waveform phase at zero. Envelopes continue to run, but the output is silenced.

Bit 3: If "1", updates the tremolo and vibrato LFOs every sample instead of once every multiple samples (tremolo is 64 times faster and vibrato is 1024 times).

Bit 4: No effect?

Bit 5: If "1", activates waveform selection.

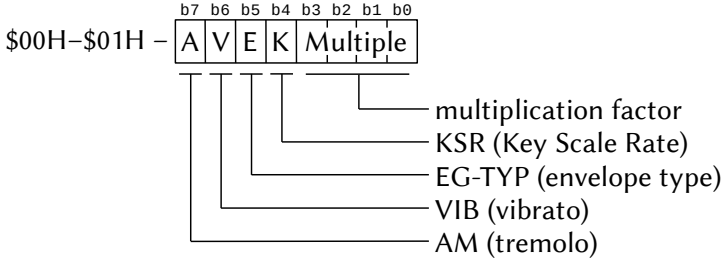
Bit 6: Not used.

Bit 7: If "1", sets the LFOs to their initial values (maximum amplitude, zero phase shift).

6.3.3.2 – Instruments definition registers

• AM/VIB/EG-TYP/KSR/MULTIPLE ($00H and $01H)

These registers specify the multiplication factor for the modulator ($00H) and carrier ($01H) frequencies with their respective components, such as the envelope and others.



MULTIPLE (b0~b3)

The frequencies of the carrier wave and the modulated wave, which generate the envelope, are controlled according to certain multiplication factors, which can be seen in the table below:

Register value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Multiplication value:	½	1	2	3	4	5	6	7	8	9	10	10	12	12	15	15

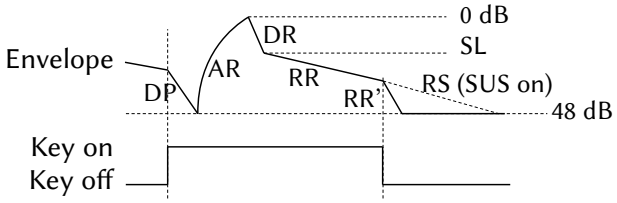
KSR (b4)

This bit is a flag that indicates whether or not the "Key Scale Rate", specified by the KSL bits, will be used. After setting the musical tones, they can have their levels changed. If KSR is equal to 0, the level will be the same for all frequencies. If KSR is equal to 1, there will be sound attenuation according to frequency; the higher the generated frequency, the greater the attenuation level. This level is specified in the KSL bits.

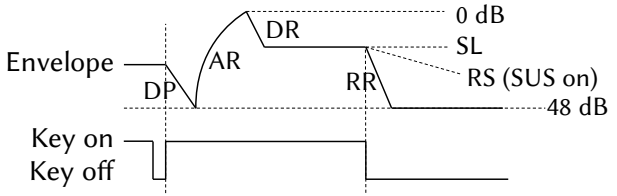
EG-TYP (b5)

This bit selects the envelope type, which can be constant tone or percussive tone. If the bit is 0, the tone will be percussive and if it is 1, the tone will be constant, as illustrated below.

Percussive Tone (b5=0)



Non-percussive Tone (b5=1)



VIB (b6)

Flag used to turn vibrato on or off. If it's 1, vibrato is on and if it's 0, it's off. The vibrato frequency is 6.4 Hz.

AM (b7)

Flag used to turn amplitude or tremolo modulation on or off. If it is 1 then amplitude modulation is active and if it is 0 it is off. The frequency for amplitude modulation is 3.7 Hz.

• KSL / TOTAL LEVEL / DISTORTION / FEEDBACK LEVEL ($02H,$03H)

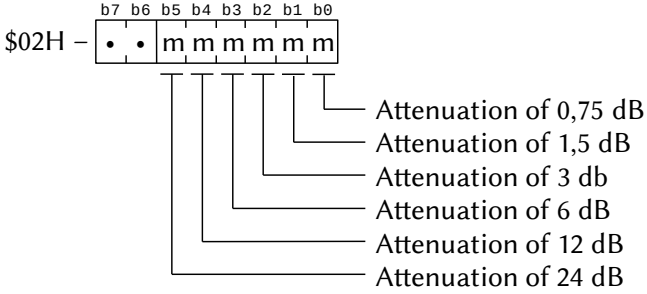
These registers are used to regulate the output so that the sound generated by the OPLL approximates that of real musical instruments.

	b7 b6	b5	b4	b3	b2 b1 b0
$02H	KSL(m)	Total level (c)			
$03H	KSL(c)	•	DC	DM	Feedback

TOTAL LEVEL (b0~b5)

This value allows controlling the modulation level through its attenuation (envelope). With the value 000 000, there will be no attenua-

tion and the modulation will be maximum. With the value 111111, the attenuation will be maximum, approximately 48 dB.



To get the correct attenuation value, just add the values when the respective bit is 1.

KSL (b6~b7)

These bits control the key scaling level. In the "key scale" mode (KSR = 1), the progressive sound attenuation level can vary from 0 dB per octave to 6 dB per octave, according to the table below:

b7	b6	Attenuation
0	0	0 dB / octave
0	1	1,5 dB / octave
1	0	3 dB / octave
1	1	6 dB / octave

DM (b3, $03H)

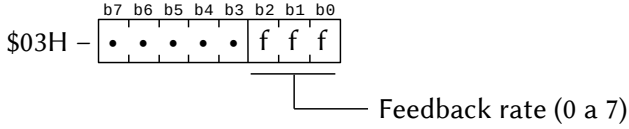
When this bit is equal to 1, the modulated wave is rectified to half wave.

DC (b4, $03H)

When this bit is equal to 1, the carrier wave is rectified to half wave.

FEEDBACK (b0~b2, $03H)

These bits define the feedback index (portion of the output signal that is re-injected into the input) for the modulated wave.



Register value:	000	001	010	011	100	101	110	111
Feedback value:	0	π/16	π/8	π/4	π/2	π	2π	4π

• ATTACK/DECAY RATES ($04H e $05H)

The "attack" and "decay" rates are defined by registers $04H and $05H, as illustrated below. The higher the value, the shorter the "attack" (AR) and/or "decay" (DR) time. The time variation follows, approximately, a geometric progression.

	b7	b6	b5	b4	b3	b2	b1	b0
$04H	Attack (modulated)				Decay (modulated)			
$05H	Attack (carrier)				Decay (carrier)			

The "attack" and "decay" values are described in the table below. Times are expressed in milliseconds (mS).

RATE		EG decay time (mS)		EG attack time (mS)	
RM	RL	0dB~48dB	10% ~ 90%	0dB~48dB	10% ~ 90%
15	3	1.27	0.52	0.00	0.00
15	2	1.27	0.52	0.00	0.00
15	1	1.27	0.52	0.00	0.00
15	0	1.27	0.52	0.00	0.00
14	3	1.47	0.60	0.14	0.10
14	2	1.71	0.60	0.18	0.12
14	1	2.05	0.82	0.22	0.14
14	0	2.55	1.03	0.28	0.18
13	3	2.94	1.21	0.30	0.22
13	2	3.42	1.37	0.34	0.22
13	1	4.10	1.65	0.42	0.26
13	0	5.11	2.05	0.50	0.32
12	3	5.87	2.41	0.54	0.36
12	2	6.84	2.74	0.60	0.38
12	1	8.21	3.30	0.70	0.44

12	0	10.22	4.10	0.84	0.54
11	3	11.75	4.03	0.97	0.64
11	2	13.60	5.47	1.13	0.72
11	1	16.41	6.60	1.37	0.84
11	0	20.44	8.21	1.69	1.09
10	3	23.49	9.65	1.93	1.29
10	2	27.36	10.94	2.25	1.45
10	1	32.03	13.19	2.74	1.69
10	0	40.07	16.41	3.30	2.17
9	3	46.99	19.31	3.86	2.57
9	2	54.71	12.80	4.51	2.98
9	1	65.65	26.39	5.47	3.30
9	0	81.74	32.03	6.76	4.34
8	3	93.97	38.62	7.72	5.15
8	2	109.42	43.77	9.01	5.79
8	1	131.31	52.78	10.94	6.76
8	0	163.49	65.65	13.52	8.69
7	3	187.95	77.24	15.45	10.30
7	2	218.84	87.54	18.02	11.59
7	1	262.61	185.56	21.00	13.52
7	0	326.98	131.31	27.03	17.30
6	3	375.98	154.40	30.90	20.60
6	2	437.69	175.07	36.04	23.17
6	1	525.22	211.12	43.77	27.03
6	0	653.95	262.61	54.87	34.76
5	3	751.79	308.96	61.79	41.19
5	2	875.37	350.15	72.89	46.34
5	1	1050.45	422.24	87.54	54.07
5	0	1307.91	525.22	108.13	69.51
4	3	1503.58	617.91	123.50	82.39
4	2	1750.75	700.30	144.48	92.69
4	1	2180.89	844.40	175.07	108.13
4	0	2615.82	1050.45	216.27	139.63
3	3	3007.16	1235.82	247.16	164.70
3	2	3501.49	1400.60	280.36	185.37
3	1	4201.79	1680.95	358.15	216.27
3	0	5231.64	1280.89	432.54	278.06

2	3	6014.32	2471.64	494.33	329.55
2	2	7002.98	2801.19	576.72	370.75
2	1	8403.58	3377.91	780.30	432.54
2	0	10 463.30	4201.79	865.88	556.12
1	3	12 078.66	4943.20	988.66	659.11
1	2	14 606.80	5602.39	1153.43	741.49
1	1	16 807.20	6755.82	1400.60	865.80
1	0	20 926.60	8403.58	1730.15	1112.24

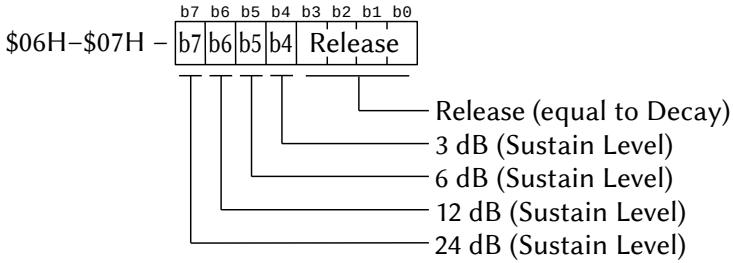
• SUSTAIN LEVEL / RELEASE RATE ($06H e $07H)

"Sustain level" is the level at which the envelope remains after being attenuated by the decay rate. For the percussive tone, it is the point of switching from "decay" mode to "release" mode. The higher the value of the register, the lower the "sustain" level.

"Release rate" is the rate at which the sound disappears after key off. For the percussive tone, it is expressed by the attenuation after the "sustain level". The higher the value of the register, the shorter the duration of the release rate.

	b7 b6 b5 b4	b3 b2 b1 b0
$06H	Sustain (modulated)	Release (modulated)
$07H	Sustain (carrier)	Release (carrier)

The values for "sustain level" are calculated as follows:

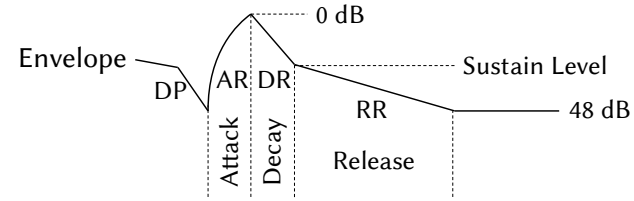


To obtain the correct value of "sustain", it is necessary to add the values when the respective bit is "1".

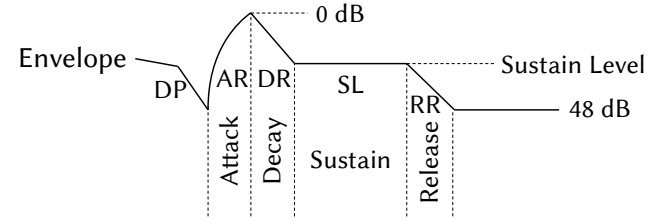
The times referring to the "release rate" are the same as the "decay rate" illustrated in the table above.

Below is an illustration of the “attack”, “decay”, “sustain level” and “release rate” values on the waveform.

Percussive Tone (b5=0)



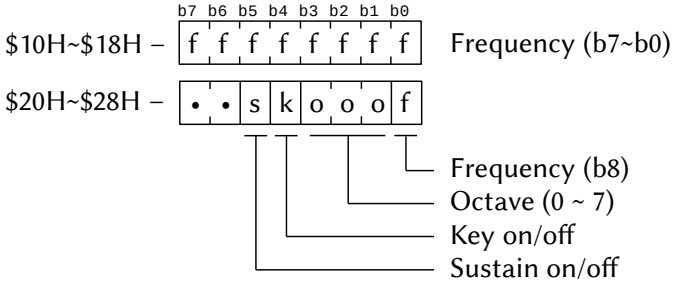
Non-percussive Tone (b5=0)



6.3.3.3 – Selection registers

• OCTAVE/FREQUENCY/KEY/SUSTAIN ($10H~$18H, $20H~$28H)

There are nine groups of two registers of 8 bits each, forming pairs, being numbered from $10H~$18H to $20H~$28H. Thus, registers $10H and $20H control the first voice, registers $11H and $21H control the second voice, and so on. These registers define the frequency of each of the 9 voices that can be generated by the OPL



FREQUENCY ($1xH and bit 0 of $2xH)

These 9 bits define a frequency scale for each octave. In the table below the values of the registers for the fourth octave (out of a total of 8 octaves) are specified, with the central note A of 440 Hz.

	Frequency	Decimal	$2xH,b0	$1xH
C#	277.2 Hz	181	0	1 0 1 1 0 1 0 1
D	293.7 Hz	192	0	1 1 0 0 0 0 0 0
D#	311.1 Hz	204	0	1 1 0 0 1 1 0 0
E	329.6 Hz	216	0	1 1 0 1 1 0 0 0
F	349.2 Hz	229	0	1 1 1 0 0 1 0 1
F#	370.0 Hz	242	0	1 1 1 1 0 0 1 0
G	392.0 Hz	257	1	0 0 0 0 0 0 0 1
G#	415.3 Hz	272	1	0 0 0 1 0 0 0 0
A	440.0 Hz	288	1	0 0 1 0 0 0 0 0
A#	466.2 Hz	305	1	0 0 1 1 0 0 0 1
B	493.9 Hz	323	1	0 1 0 0 0 0 0 1
C	523.3 Hz	343	1	0 1 0 1 0 1 1 1

The frequency values hold a geometric relationship equal to the 12th root of 2, which is 1.059463094359 2. You can use this number to change the register values to increase or decrease the generated frequency within the musical scale. The values of the registers also have the same relationship to each other.

OCTAVE ($2xH, b3~b1)

These three bits define the octave. Up to 8 octaves can be set, from 000 to 111, with the fourth octave being 011.

KEY($2xH, b4)

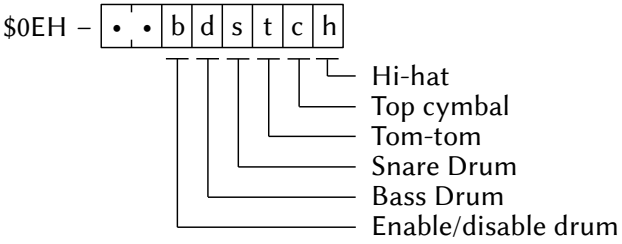
This bit must be set to 1 for the sound of each of the nine voices to be enabled. When it is 0, the sound of the respective voice will be off (key off).

SUSTAIN ($2xH, b5)

When this bit is set to 1, the value of "release rate – RR" will gradually decrease when the respective "key" bit is turned off; otherwise, the sound will be cut off abruptly.

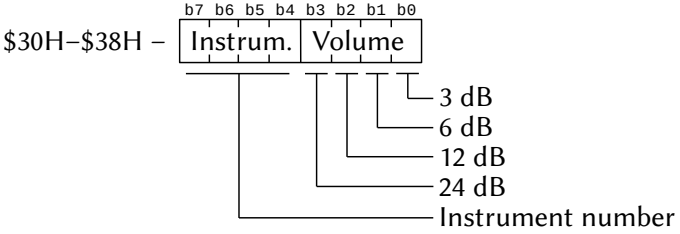
• RHYTHM CONTROL ($0EH)

Register $0EH controls the OPLL rhythm mode. To activate it, just set its bit 5 to 1. Bits 0 to 4 enable or disable each of the 5 pieces of rhythms available. When in rhythm mode, only the first six voices of the OPLL are available for generating sounds from other musical instruments.



• INSTRUMENTS AND VOLUME SELECTION ($30H~$38H)

These registers select the instrument and volume for each of the nine available voices. Thus, $30H is used for the first voice, $31H for the second, and so on.



Bits b3~b0 determine the volume. The lowest resolution is 3 dB and the highest 45 dB, obtained by adding the values when the respective bit is "1".

Bits b7~b4 select the instrument, the value 0000B selects the user-defined instrument. The 15 instruments available are as follows:

0001 – violin	0110 – oboe	1011 – harpsichord
0010 – guitar	0111 – trumpet	1100 – vibraphone
0011 – piano	1000 – organ	1101 – electric bass
0100 – flute	1001 – piston	1110 – acoustic bass
0101 – clarinet	1010 – synthesizer	1111 – electric guitar

In rhythm mode, registers $36H, $37H and $38H only determine the volume of each of the available rhythm parts, but registers $30H to $35H keep their functions unchanged. In this mode, the OPLL can generate six instruments plus five rhythm pieces.

$36H	•	•	•	•	Bass Drum	Volume register for the rhythm instruments
$37H	Hi Hat				Snare Drum	
$38H	Tom tom				Top Cymbal	

6.3.4 – The FM-BIOS

Normally, the OPLL comes with a ROM that allows accessing, through BASIC, all its registers. This extended BASIC is called MSX-MUSIC. Additionally, 48 more instruments are defined in this ROM. Thus, you have access to up to 63 instruments. However, only the OPLL's 15 internal instruments can be freely mixed with each other in any of the nine voices. Selected FM-BIOS instruments cannot be mixed with each other as the OPLL accepts the external definition of only one instrument. FM-BIOS reserves the number 63 (silence) for the external instrument definition by MSX-MUSIC. For direct access, the user-defined instrument is number 0. The table below shows the definition values of all FM-BIOS instruments.

	Instruments Regs →	$0	$1	$2	$3	$4	$5	$6	$7
00	Piano 1	OPLL data (3)							
01	Piano 2	30	10	0F	04	D9	B2	10	F4
02	Violin	OPLL data (1)							
03	Flute 1	OPLL data (4)							
04	Clarinet	OPLL data (5)							
05	Oboe	OPLL data (6)							
06	Trumpet	OPLL data (7)							
07	Pipeorgan	34	30	37	06	50	30	76	06
08	Xylophone	17	52	18	05	88	D9	66	24
09	Organ	OPLL data (8)							
10	Guitar	OPLL data (2)							
11	Santool 1	19	53	0C	06	C7	F5	11	03
12	Electric guitar	OPLL data (15)							

13	Clavicode 1	03	09	11	06	D2	B4	F5	F6
14	Harpsicode 2	OPLL data (11)							
15	Harpsicode 2	01	01	11	06	C0	B4	01	F7
16	Vibraphone	OPLL data (12)							
17	Koto 1	13	11	0C	06	FC	D2	33	84
18	Taiko	01	10	0E	07	CA	E6	44	24
19	Engine 1	E0	F4	1B	87	11	F0	04	08
20	UFO	FF	70	19	07	50	1F	05	01
21	Synthesizer bell	13	11	11	07	FA	F2	21	F5
22	Chime	A6	42	10	05	FB	B9	11	02
23	Synthesizer bass	OPLL data (13)							
24	Synthesizer	OPLL data (10)							
25	Synthesizer percussion	01	03	0B	07	BA	D9	25	06
26	Synthesizer rhythm	40	00	00	07	FA	D9	37	04
27	Harmdrum	02	03	09	07	CB	FF	39	06
28	Cowbell	18	11	09	05	F8	F5	26	26
29	Close hi-hat	0B	04	09	07	F0	F5	01	27
30	Snare drum	40	40	07	07	D0	D6	01	27
31	Bass drum	00	01	07	06	CB	E3	36	25
32	Piano 3	11	11	08	04	FA	B2	20	F5
33	Wood bass	OPLL data (14)							
34	Santool 2	19	53	15	07	E7	95	21	03
35	Brass	30	70	19	07	42	62	26	24
36	Flute 2	62	71	25	07	64	43	12	26
37	Clavicode 2	21	03	0B	05	90	D4	02	F6
38	Clavicode 3	01	03	0A	05	90	A4	03	F6
39	Koto 2	43	53	0E	85	B5	E9	85	04
40	Pipe organ	34	30	26	06	50	30	76	06
41	RhodsPLA	73	33	5A	06	99	F5	14	15
42	RhodsPRA	73	13	16	05	F9	F5	33	03
43	Orch L	61	21	15	07	76	54	23	06
44	Orch R	63	70	1B	07	75	4B	45	15
45	Synthesizer violin	61	A1	0A	05	76	54	12	07
46	Synthesizer organ	61	78	0D	05	85	F2	14	03
47	Synthesizer brass	31	71	15	07	B6	F9	03	26
48	Tube	OPLL data (9)							
49	Shamisen	03	0C	14	06	A7	FC	13	15

50	Magical	13	32	81	03	20	85	03	B0
51	Huwawa	F1	31	17	05	23	40	14	09
52	Wander flat	F0	74	17	47	5A	43	06	FD
53	Hardrock	20	71	0D	06	C1	D5	56	06
54	Machine	30	32	06	06	40	40	04	74
55	Machine V	30	32	03	03	40	40	04	74
56	Comic	01	08	0D	07	78	F8	7F	FA
57	SE-Comic	C8	C0	0B	05	76	F7	11	FA
58	SE-Laser	49	40	0B	07	B4	F9	00	05
59	SE-Noise	CD	42	0C	06	A2	F0	00	01
60	SE-Star 1	51	42	13	07	13	10	42	01
61	SE-Star 2	51	42	13	07	13	10	42	01
62	Engine 2	30	34	12	06	23	70	26	02
63	Silence	00	00	00	00	00	00	00	00

The table lists the 63 instruments in MSX-MUSIC. The eight related bytes must fill, respectively, the first eight registers of the OPLL ($00H to $07H), which are responsible for defining the instrument created by the programmer. The table also contains the OPLL's internal instruments and, in this case, instead of the bytes, it brings the expression “OPLL data”, followed by the instrument number.

6.3.5 – The stereo FM

Although not officially foreseen for the MSX standard, FM stereo ended up being standardized by the market due to a characteristic of the OPLL: the responsible chip, the YM2413, has two separate outputs for the sounds; one is called “melody output” and for this the OPLL generates the first six voices; the other is called the “rhythm output” and from this the OPLL generates the remaining three voices or five rhythm parts. It was then agreed that the “melody output” would be one of the stereo channels and the “rhythm output” plus the PSG sound would be the other stereo channel.

Thus, FM stereo can generate two channels of six voices each. Many programs, especially games, can make use of FM stereo even if they haven't been programmed to use it, which most of the time generates a beautifully nuanced, beautiful sound.

6.3.6 – Access to OPLL

Access to OPLL can be done directly or through FM-BIOS, which is a specific ROM installed in FM-OPLL cartridge and also in all MSX that come with OPLL. This ROM is located on physical page 1 (between 4000H and 7FFFH) and has some routines that can be accessed by the user.

To access the OPLL, the first step is to verify the presence of FM-BIOS. To do this, you need to do a scan looking for the following strings in all slots and subslots in the following order:

1st – "APRLOPLL" in 4018H ~ 401FH (Internal or clone)
2nd – "OPLL" in 401CH ~ 401FH (Panasoft FM-PAC)

The order of checks is important because if done in the wrong order, the Panasoft FM-PAC will ring simultaneously with the internal FM-PAC. This also means that an internal MSX-MUSIC (or clone cartridge) takes precedence over a Panasoft FM-PAC.

If the former is found, you can directly use OPLL on the I/O ports. If the first is not found, but the second is, then you need to first enable the I/O ports by setting the I/O enable flag at address 7FF6H (see below), or use memory-mapped I/O.

Direct access to internal OPLL

I/O port	R/W	Use	Wait (µsec)	Wait (clock cycles)
7CH	W	Select	3.4	12
7DH	W	Data	23.5	84

Direct access to external OPLL (in cartridge)

I/O port	Adress	R/W	Use	Wait (µsec)	Wait (clock cycles)
7CH	7FF4H	W	Select	3.4	12
7DH	7FF5H	W	Data	23.5	84

To activate the I/O ports for accessing the OPLL in cartridge, it is necessary to set the bit "0" of the address 7FF6H in the FM-PAC slot, otherwise the OPLL chip will be available for use only through memory addresses. To avoid simultaneous access to two devices on the same I/O

ports, it is not recommended to enable FM-PAC when there is internal MSX-MUSIC present.

If there is an "APRLOPLL" string, bit0 of #7FF6 should not be set, as this breaks compatibility with Panasonic MSX2+ computers. On Panasonic A1WX and A1WSX, setting bit 0 in any address between #7FF0 and #7FFF will disable FM-BIOS completely.

6.3.6.1 – Access via FM-BIOS

If you use MSX-BASIC or FM-BIOS, no extra attention is needed because the CALL MUSIC command and the OPLINI routine will automatically activate OPLL when needed. These routines will always look for the internal MSX-MUSIC first (APRLOPLL) and then the external one (PAC2OPLL) and activate the external one only if the internal one is not found.

The following routines are available in FM-BIOS:

WRTOPL (4110H) – Writes a byte of data to an OPLL register.
INIOPL (4113H) – Initializes the FM-BIOS/OPLL workarea.
MSTART (4116H) – Starts playing music.
MSTOP (4119H) – Stop the music.
RDDATA (411CH) – Returns instrument data from ROM.
OPLDRV (411FH) – Input for the OPLL driver. It is the routine that plays the music, and must be called by the interrupt handler through the HTIMI hook.
TSTBGM (4122H) – Checks if there is still data in the music queue.

These routines are described in detail in the Appendix, item "8.7 – MSX-MUSIC ROUTINES (FM/OPLL)".

6.3.6.2 – Direct access

Direct access is via two I/O ports, 7CH and 7DH, or via addresses 7FF4H and 7FF5H. Port 7CH (or 7FF4H) selects registers and port 7DH (or 7FF5H) writes data bytes to them. However, OPLL is slow. Between one access and another there must be a pause, as shown in the table below.

Register selection (7CH) 12 cycles (master clock – 3.58 MHz).
Data Write (7DH) 84 cycles (master clock – 3.58 MHz).

These values correspond to the following times:

	Time	Z80 T cycles	R800 T cycles
Select (7CH)	3,4 µs	12	24
Write (7DH)	23,5 µs	84	168

Note that the R800's internal clock is 7.16 MHz; therefore the number of cycles T must be doubled. Unlike VDP, no hardware pauses are generated for YM2413 access; therefore there will be desynchronization and data will be corrupted when the R800 is active on the MSX turbo R, in case the program has not foreseen this.

For the Z80, pauses like "EX (SP),HL" or "NOP" can be used until the OPLL is ready for new access.

First, you must select the register to be written through port 7CH. After writing, there must be a pause of at least 12 T cycles in the case of a standard MSX (Z80 at 3.58 MHz). The instruction "OUT (07CH),A" takes 11 T cycles to be processed; therefore, at least 1 more T cycle is required. A NOP instruction (which takes 4 T cycles to be processed) can be used for this, as shown below:

```
LD   A,REG    ; Register number in A
OUT  (07CH),A ; Select the register
NOP           ; Pause
```

Soon after, the data is written in the selected register through port 7DH. The pause should now be at least 84 T cycles on a standard MSX machine (Z80 at 3.58 MHz). For this, four "EX (SP),HL" instructions can be used (which take 19 T cycles each to be processed), resulting in a pause of 76 T cycles which, added to the 11 cycles of the OUT instruction, result in 87 T cycles. Then the OPLL will be ready to receive new data. It is essential that the "EX (SP),HL" instructions are in pairs; otherwise the contents of the stack will change, causing data corruption.

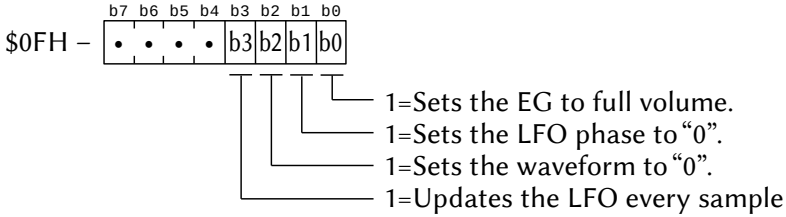
```
LD   A,REG     ; Register number in A
OUT  (07CH),A  ; Select the register
NOP            ; Pause
```

```
LD   A,DATA     ; A = data to be written
OUT (07DH),A    ; Write data to register
EX   (SP),HL    ; Pause
EX   (SP),HL    ; Pause
EX   (SP),HL    ; Pause
EX   (SP),HL    ; Pause
```

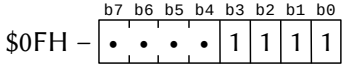
The pause time can also be used to process some task, as long as this processing takes at least 84 T cycles for the Z80 or 168 for the R800 and does not interfere with the values to be written to the OPLL.

6.3.7 – Access to internal DAC

The OPLL has a 4-bit DAC internally that can be accessed when the chip is put into test mode. It works in the same way as the PCM described later in item 4. To access it, it is necessary to use the test register ($0FH).



To enable the internal DAC, you need to set bits 0 to 3 of the test register to "1":



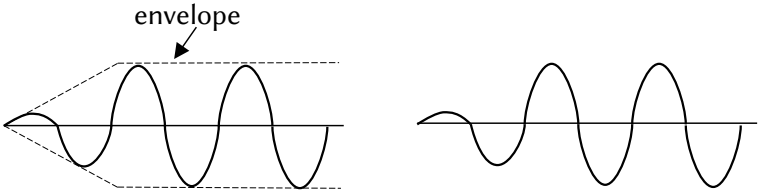
Then the data can be sent to the top 4 bits of register $10H. It is necessary to use a mask of type "AND 0F0H" so that the lowest four bits remain at "0". Before sending the data, it is necessary to set some registers:

$0FH ⟵ 15 or 255
$30H ⟵ 140 or 255
$20H ⟵ 31 or 255

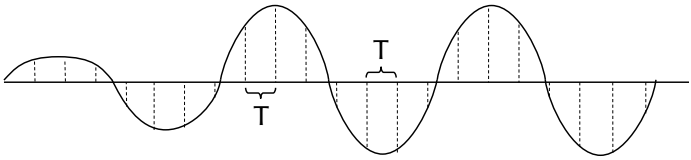
Then they can be sent from the PCM data to $10H, using the mask "AND 0F0H". Timing must be controlled by software.

6.4 – THE PCM

PCM means "pulse code modulation". PCM is not a sound generator itself; works as a digitizer or "sampler" of sounds. In this way, it is possible to reproduce sounds of any nature, including the human voice, practically perfectly. PCM does not have registers to specify sounds; these are obtained by sampling. A typical signal generated by the PCM, in this case a sine wave, is illustrated below:



The way the PCM works to reproduce this same wave is illustrated below:



At each period T, the PCM collects the sound level. The period T is fixed and the frequency with which the sampling takes place is called the "sampling rate". To reproduce the sound, simply repeat the data at the same speed at which it was collected. The higher the sampling frequency (sampling rate), the better the quality of the sound reproduced.

Another feature of PCM is resolution. In the case of MSX, the resolution is 8 bits, that is, each collection has 8 bits and therefore the amplitude of the collected sound wave has a variation of 256 levels. Each sample therefore occupies one byte of memory.

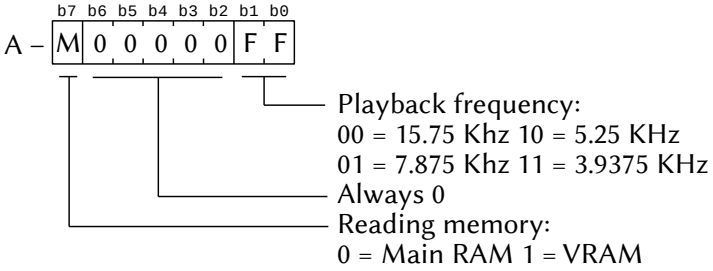
On MSX, there are 4 standard sampling rates: 3.9375 KHz, 5.25 KHz, 7.875 KHz and 15.75 KHz. This means that, for example, at the rate of 5.25 KHz, there are 5250 collections of 8 bits every second, and for the storage of 1 second of sound, more than 5 Kbytes of memory are spent.

Therefore, the data for the PCM is stored in the PC's own RAM and not in registers.

6.4.1 – Access to PCM

The PCM can be accessed either through the BIOS or directly. In case of BIOS access, there are two routines available:

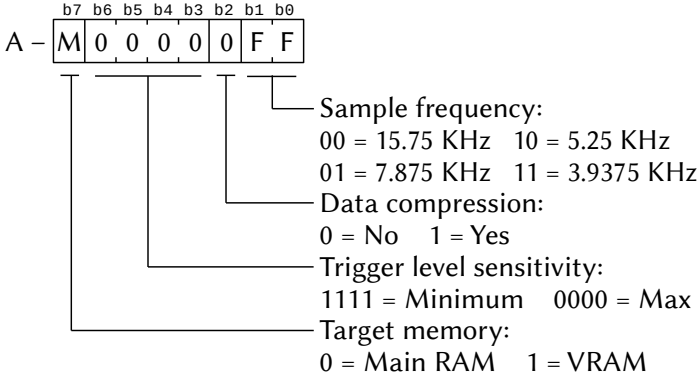
PCMPLY (0186H / Main)
Function: Play sounds through the PCM.
Input: EHL – Address to start reading.
DBC – Size of the block to be reproduced (length).



Note: The 15.75 Khz frequency can only be used in the R800 DRAM mode.

Output: CY – 0 ⟶ Playback OK.
1 ⟶ Playback error.
Cause of error:
A – 0 ⟶ error in specifying the frequency.
1 ⟶ interruption by CTRL+STOP.
EHL – Address as far as it actually reproduced.
Registers: All.

PCMREC (0189H / Main)
Function: Digitize sounds through the PCM.
Input: EHL – Address to start reading.
DBC – Size of the block to be digitized (length).

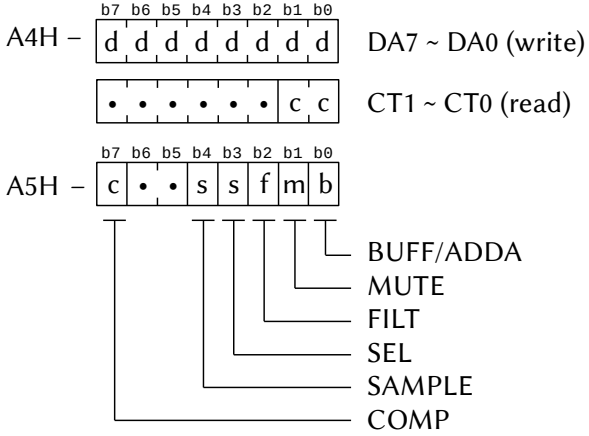


Note: The 15.75 Khz frequency can only be used in R800 DRAM mode.

Output: CY – 0 ⟶ Record OK.
1 ⟶ Record error.
Cause of error:
A – 0 ⟶ error in specifying the frequency.
1 ⟶ interruption by CTRL+STOP.
EHL – Address as far as it actually recorded.

Registers: All.

Direct access to the PCM is via two I/O ports. Port A4H is the data port and A5H is the command port.



BUFF/ADDA: sets the conversion direction. For sound generation (output), this bit must be 0 (D/A conversion). For sound digitization (input), this bit must be 1 (A/D conversion).

MUTE: Turns the sound output of the entire system on or off. If it is 0, the output will be off (mode selected on reset). If it is 1, it will be on.

FILT: defines the signal type for A/D conversion. If it is 0, it will use the normal signal (selected at reset). If it is 1, the signal will pass through the filter.

SEL: selects the filter input signal. If it is 0, the low pass filter (D/A conversion) will be used. If it is 1, the microphone signal will be used.

SAMPLE (sample hold): defines how the input signal will be treated. If 0, A/D conversion is active (mode selected on reset). If it is 1, it will be off.

COMP: this bit is only valid for port reading. It sets the output comparator signal level (D/A conversion). If this bit is 1, the D/A converter signal will be greater than the sample hold signal. If it is 0, it will be smaller.

DA7 ~ DA0: D/A converter output data. The data format is in absolute binary, where the value 127 corresponds to level 0.

CT1 ~ CT0 (counter data): is a reference counter. Every 63.5µs, the counter is incremented. As this period corresponds to the frequency of 15.75 KHz, the counter serves as a reference for the four sampling rates available, as illustrated below:

00 – 15.75 Khz	01 – 7.875 KHz
10 – 5.25 Khz	11 – 3.9375 KHz

When writing data in A4H, the counter is reset.

To digitize a sound using direct access, it is necessary to read the data bit by bit. Here's an assembler routine that digitizes sounds through PCM.

```
PMDAC   EQU  0A4H
PMCNT   EQU  0A4H
PMCNTL  EQU  0A5H
PMSTAT  EQU  0A5H
SYSTML  EQU  0E6H

REC:    LD   A,  00001100B
        OUT  (PMCNTL),A   ; A/D MODE
        DI
        XOR  A
        OUT  (SYSTML),A   ; COUNTER 0
REC1:   IN   A,(SYSTML)
        CP   E
        JR   C,REC1
        XOR  A
        OUT  (SYSTML),A
        PUSH BC
        LD   A,00011100B
        OUT  (PMCNTL),A    ; HOLD DATA
        LD   A,80H
        LD   C,PMSTAT
        OUT  (PMDAC),A     ; BIT CONVERSION
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD0
        AND  01111111B
RECAD0: OR   01000000B     ; BIT 7
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD1
        AND  10111111B
RECAD1 :OR   00100000B     ; BIT 6
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD2
        AND  11011111B
RECAD2: OR   00010000B     ; BIT 5
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP  M,RECAD3
        AND  11101111B
RECAD3: OR   00001000B     ; BIT 4
        OUT  (PMDAC),A
```

```
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD4
        AND  11110111B
RECAD4: OR   00000100B     ; BIT 3
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD5
        AND  11111011B
RECAD5: OR   00000010B     ; BIT 2
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP   M,RECAD6
        AND  11111101B
RECAD6: OR   00000001B     ; BIT 1
        OUT  (PMDAC),A
        DEFB 0EDH,70H      ; IN F,(C)
        JP  M,RECAD7
        AND  11111110B
RECAD7: OR   00000000B     ; BIT 0
        LD   (HL),A        ; SAVES READ DATA
        LD   A,00001100B
        OUT  (PMCNTL),A
        POP  BC
        INC  HL
        DEC  BC
        LD   A,  C
        OR   B
        JR   NZ,REC1
        LD  A,00000011B
        OUT  (PMCNTL),A     ; D/A MODE
        EI
        ;
        RET
```

To play the sounds, you need to set the PCM to play (A5H port) and then send the data bytes at the correct speed through the A4H port. The following routine plays the sounds through PCM.

```
PMDAC   EQU  0A4H
PMCNT   EQU  0A4H
PMCNTL  EQU  0A5H
PMSTAT  EQU  0A5H
SYSTML  EQU  0E6H
```

```
PLAY:   LD  A,00000011B
        OUT  (PMCNTL),A     ; D/A MODE
        DI
        XOR  A
        OUT  (SYSTML),A     ; COUNTER 0
PLAY1 : IN   A,(SYSTML)
        CP   E
        JR   C,PLAY1
        XOR  A
        OUT  (SYSTML),A
        LD   A,(HL)         ; READ DATA BYTE
        OUT  (PMDAC),A      ; PLAY DATA
        INC  HL
        DEC  BC
        LD  A,  C
        OR   B
        JR   NZ,PLAY1
        EI
        ;
        RET
```

6.5 – MSX-AUDIO

The MSX-Audio was released together with the MSX2 in 1985 as a standard optional peripheral, but, probably due to its high price, it didn't become standard, later giving way to the OPLL. It was used in only a few sound cartridges.

The responsible chip is the Y8950. Of all the sound generators created for MSX, MSX-Audio is the most complete. Like the OPLL, it has 9 FM sound voices, but all are resettable. It also has an ADPCM (Adaptive Differential Pulse Code Modulation) channel, which works similarly to PCM, but stores the data in a compressed form. In addition, it can also be connected to 256 Kbytes of external memory, freeing up Main RAM and CPU, and also to an external music keyboard directly. These and other features make MSX-Audio the best and most complete audio generator ever created for the MSX system.

6.5.1 – Description of ADPCM analysis and synthesis

ADPCM is a method of sound analysis or synthesis in which the relative difference between current PCM data and subsequent PCM data is obtained. This method prevents deterioration of the synthesized sound and reduces the amount of memory required. In fact, ADPCM converts an 8-bit resolution PCM data into a 4-bit ADPCM data. The encoding of data during digitization proceeds as follows:

1. The data is digitized at 8 bits of resolution (Xn);

2. The result is multiplied by 256 to convert it to 16 bits and is then compared with the subsequent data (Xn + 1) to obtain the difference (dn);

3. When the difference results in a positive value, the L4 bit of the ADPCM data will be 0; when negative, it will be 1. At the same time, the absolute value of the difference is calculated (|dn|).

4. Then, the remaining three bits are set according to the table below (Δn is the calculated relative variance).

L_4		L_3	L_2	L_1	Value analyzed
$dn \geq 0$	$dn < 0$				
0	1	0	0	0	$\lvert dn \rvert < \Delta n /$
		0	0	1	$\Delta n / 4 \leq \lvert dn \rvert < \Delta n /$
		0	1	0	$\Delta n / 2 \leq \lvert dn \rvert < \Delta n * 3/4$
		0	1	1	$\Delta n * 3/4 \leq \lvert dn \rvert < \Delta n$
		1	0	0	$\Delta n \leq \lvert dn \rvert < \Delta n * 5/4$
		1	0	1	$\Delta n * 5/4 \leq \lvert dn \rvert < \Delta n * 5/4$
		1	1	0	$\Delta n * 3/2 \leq \lvert dn \rvert < \Delta n * 7/4$
		1	1	1	$\Delta n * 7/4 \leq \lvert dn \rvert$

5. After the ADPCM data is obtained, new subsequent data (Xn+2) and new variation (Δn+1) are obtained.

$$Xn+2 = (1 - 2 * L4) * (L3 + L2 / 2 + L1 / 4 + 1/8) * \Delta n + Xn+1$$

$$\Delta n+1 = f(L3, L2, L1) * \Delta n$$

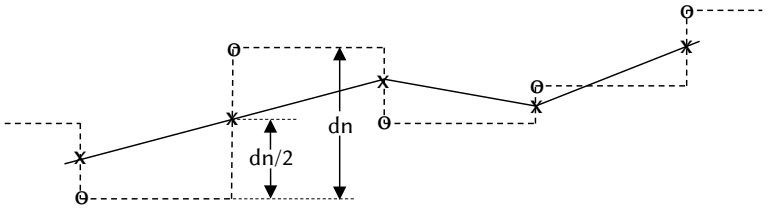
The table below presents the multiplication factors used to adjust the relative variance (Δn).

L_3	L_3	L_3	f
0	0	0	0,9
0	0	1	0,9
0	1	0	0,9
0	1	1	0,9
1	0	0	1,2
1	0	1	1,6
1	1	0	2,0
1	1	1	2,4

Complete ADPCM data is obtained by repeating steps 1 to 5 for each 8-bit PCM sample.

Sound synthesis using the ADPCM data is done by reading the data obtained in step 5 and calculating the relative variation between them. Direct reproduction of this data, however, causes great waveform distortion and noise. Therefore, MSX-Audio incorporates a procedure to smooth the waveform.

First, the data is processed through the smoothing circuit. it's like a low-pass filter to eliminate high-frequency noise. Then a linear interpolation is performed and the digitized data is repeated at a frequency of 50 KHz in the intervals between the original samples. The result is shown below.



o : ADPCM data sample
x : data after smoothing
------ : wave form with no smoothing
——— : waveform after smoothing and applying 50 KHz

6.5.2 – Map of MSX-Audio registers

MSX-Audio has 141 8-bit registers to specify all its functions, numbered $01H to $1AH, $20H to $35H, $40H to $55H, $60H to $75H, $80H to $95H, $ A0H to $A8H, $B0H to $B8H, $BDH and $C0H to $C8H. The functions of these registers are briefly described in the table below.

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$01H	Test								Test register
$02H $03H	1st Timer (80 µS) 2nd Timer (320 µS)								Time registers
$04H	IRQ	T1M	T2M	EOS	BR	•	ST2	ST1	Flags register
	b7 b6 b5 b4 b3 b2 b1 b0	IRQ – If write 1, reset all flags. T1M – If write 1, b0 will reset. T2M – If write 1, b1 will reset. EOS – B3 mask, indicating the end of current operation BR – ADPCM / Audio memory mask (1=enable) Not used (always0) ST2 – $03 Start/stop control (1=start counter) ST1 – $02 Start/stop control (1=start counter)							
$05H $06H	External keyboard (input) External keyboard (output)								Registers for access to external musical keyboard
$07H	STA	REC	MEM	REP	OFF	•	•	RST	Control register (1)
	b7 b6 b5 b4 b3 b1~b2 b0	STA – Must be 1 to start data read/write REC – Must be 1 to write data in the memory MEM – Must be 1 to access audio memory REP – When 1, enable ADPCM data repeat OFF – When 1, cut off audio output Not used (always“00”) RST – When 1, puts ADPCM in the initial state							
$08H	CSM	SEL	•	•	SAM	DAD	64K	ROM	Control register (2)
	b7 b6 b5~b4	CSM – 1=composite sinusoisal modulation mode SEL – External keyboard octaves separation point Not used (always “00”)							

	b3 b2 b1 b0	SAM – 0=start DA conversion; 1=start AD conversion DAD – 0=conv. AD / mus. output; 1=$15~$16 ⟶ output 64K – Memory size: 0=256K; 1=64K ROM – Memory type: 0=RAM; 1=ROM							
$09H $0AH $0BH $0CH	Start adress (b7~b0) Start adress (b15~b8) End adress (b7~b0) End adress (b15~b8)								Start and end adresses for CPU and ADPCM acesses
$0DH	f7	f6	f5	f4	f3	f2	f1	f0	ADPCM frequency 3580 / F_num (1,8~16 KHz)
$0EH	•	•	•	•	•	f10	f9	f8	
$0FH	ADPCM data								Data register
$10H	i7	i6	i5	i4	i3	i2	i1	i0	ADPCM interpolation factor (i15~i0) = 1310,72 *sampling rate
$11H	i15	i14	i13	i12	i11	i10	i9	i8	
$12H	ADPCM volume								ADPCM volume (0~255)
$15H	f9	f8	f7	f6	f5	f4	f3	f2	DA conversion data Out: Vcc/2 + Vcc/4*(−1+f9+ f8*2^{-1}+...+f1*2^{-8}+f0*2^{-9}+2-10)*2^{-E} E=S2*4+S1*2+S0*1 (S0+S1+S2>0)
&16H	f1	f0	•	•	•	•	•	•	
$17H	•	•	•	•	•	S2	S1	S0	
$18H	•	•	•	•	I/O control				I/O ports control ($18H ⟶ 0=input; 1=output)
$19H	•	•	•	•	I/O data				
$1AH	ADPCM data								Data register
$20H ⋮ $35H	AM	VIB	EGT	KSR	Multiple				Instruments definition
	b7 b6 b5 b4 b3~b0	AM (1=tremolo on – Frequency: 3,7Hz) VIB (1=vibrato on – Frequency: 6,4Hz) EG-TYP (0=decaying sound; 1=sustained sound) Se 0, KSR⟶0~3; Se 1, KSR⟶0~15 Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)								
$40H ⋮ $55H	KSL		Total level						KSL (00= 0dB/octave, 01=1,5dB, 10=3dB, 11=6dB) Total level (b0=0,75dB, b1=1,5dB ..... b5=24dB)

$60H ⋮ $75H	Attack Rate (AR)			Decay Rate (DR)					Attack (0dB a 96dB → min. 0,2 mS; max 2826 mS) Decay (0dB a 96dB → min. 2,4 mS; max 39 280 mS)
$80H ⋮ $95H	Sustain Level (SL)			Release Rate (RL)					Sustain (b7=24dB, b6=12dB, b5=6dB, b4=3dB) Release (0dB a 96dB → min. 2,4 mS; max 39 280 mS)
$A0H ⋮ $A8H	Frequency (LSB 8 bits)								FM frequency (b7~b0)
$B0H ⋮ $B8H	•	•	KEY	Octave		Freq. MSB 2 bits			FM Frequency (b9~b8) Octave (FM) Key on/off (FM)
	b7~b6 b5 b4~b2 b1~b0	Not used (always“00”) 0=key off; 1=key on (voice on) Octave definition. The fourth is 011. MSB Frequency 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 001							
	Operators (para$20H~$35H e$A0H~$A8H) Oper: 01 02 03 04 05 06 07 08 09 Voz: 1 2 3 1 2 3 4 5 6 Reg: $20$21$22$23$24$25$28$29$2A Freq:$A0$A1$A2$A0$A1$A2$A3$A4$A5 Oper: 10 11 12 13 14 15 16 17 18 Voz: 4 5 6 7 8 9 7 8 9 Reg: $2B$2C$2D$30$31$32$33$34$35 Freq:$A3$A4$A5$A6$A7$A8$A6$A7$A8								The operators are associated as below: $20/$40/$60/$80/$A0/$B0/$C0 ou $23/$43/$63/$83/$A0/$B0/$C0
$BDH	AM	VIB	BAT	BD	SD	TOM	TCY	HH	FM rhythm control
	b7 b6 b5 b4 b3	AM – Tremolo level (0=1dB. 1=4,8dB) VIB – Vibrato level (0=7%; 1=14%) BAT – 0=Melody mode; 1=Rhythm mode BD – 1=Bass Drum SD – 1=Snare Drum							

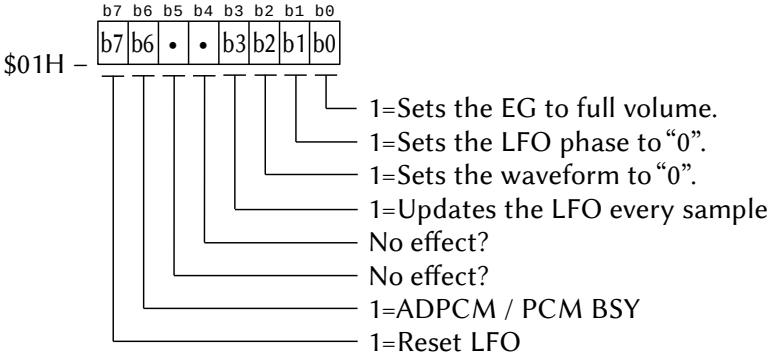
	b2 b1 b0	TOM – 1=Tom-tom TCY – 1=Top Cymbal HH – 1=High-Hat						
$C0H ⋮ $C8H	•	•	•	•	Feedback		CON	FM Feedback factor and connection type
	b7~b4 b3~b1 b0	Not used (always“0000”) Feedback (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π) Operators connection type (0=série; 1=paralelo)						
STAT	INT	T1	T2	EOS	BUF	•	•	PCM	Status register
	b7 b6 b5 b4 b3 b2-b1 b0	Will be 1 when one or more bits b3 to b6 contains 1 Will be 1 after timer 1 ends counting ($02) Will be 1 after timer 2 ends counting ($03) Will be 1 when ADPCM analysis/synthesis ends Will be 1 at the end of read/write/analysis/synthesis Not used (always“00”) Will be 1 during the ADPCM analysis/synthesis (if b7 of $07 are 1)						

6.5.3 – Description of registers

This section describes in detail the registers of the Y8950 and how they work.

6.5.3.1 – Test register

The $01H register is the test register. It is only used for testing MSX-Audio. Its value is normally 0.



Bit 0: If "1", the envelope generators output is set to "0" (full volume) for both the modulator and the carrier.

Bit 1: If "1", keeps the LFO phase at zero. This stops, disables and resets the tremolo and vibrato LFO.

Bit 2: If "1", keep the waveform phase at zero. The envelope continue to run, but the output is silenced.

Bit 3: If "1", updates the tremolo and vibrato LFOs every sample instead of once every multiple samples (tremolo is 64 times faster and vibrato is 1024 times).

Bit 6: ADPCM. Affects bit 0 (PCMBSY) of the status register.

Bit 7: If "1", sets the LFOs to their initial values (maximum amplitude, zero phase shift).

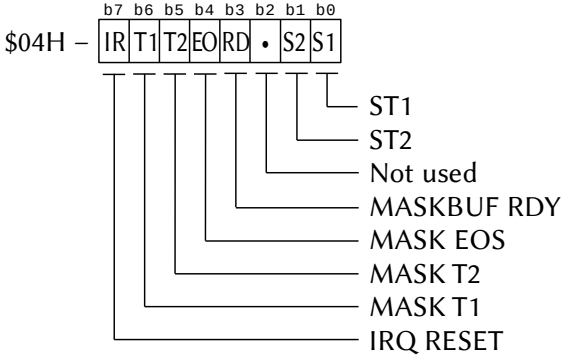
6.5.3.2 – Time registers

There are two time registers: $02H with 80 μs resolution and $03H with 320 μs resolution. They are 8-bit timers and can perform start, stop and signaling operations. If the flag is set, a hardware interrupt will be sent to the CPU. They can also control the composite sine modulation of the FM generator. These registers can be loaded with any value between 0 and 255. When the count exceeds the maximum value of the register, the flag will be set and the initial value will be reloaded. Then a hardware interrupt will be generated and/or all 9 FM voices will be Key-on and then off (Key-off). The count time in milliseconds of each register can be calculated by the following formulas:

T0 (ms) = (256 – ($02H)) * 0.08
T1 (ms) = (256 – ($03H)) * 0.32

6.5.3.3 – Flags control

The flag register ($04H) is used to control the start, stop and stop of registers $02H and $03H, the ADPCM and the external audio memory. Each bit of this register enables or disables a function, as described below.



b0 (ST1): This bit controls the start and stop operations of $02H. When it is 0, $02H is off. When it is 1, $02H will load and start counting.

b1 (ST2): This bit controls the start and stop operations of $03H, in the same way as b0 for $02H.

b2: Not used.

b3 (MASKBUF RDY): This bit controls the ADPCM and audio memory. When it is 0, the function is disabled. When set to 1, read and write data will be masked during data transfer between the processor and the ADPCM and audio memory.

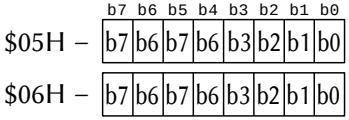
B4 (MASK EOS): This bit is used to mask bit b3, indicating the end of reading/writing from ADPCM or external storage, or the end of AD conversion.

B5 (MASK T2): When this bit is set to 1, b1 will be set to 0.

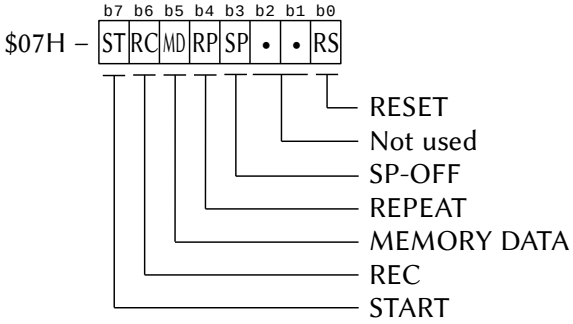
B6 (MASK T1): This bit is used to mask b0.

B7 (IRQ RESET): Each MSX-Audio flag is set to 1 when the respective event occurs and IRQ is at level 0 (interrupts are disabled). This bit is used to re-enable interrupts. When this bit is 1, all flags (flags) will be set to 0. If only some flags are to be reset, just set the corresponding MASK bit to 1. After all flags are reset, b7 is automatically reset to 0.

6.5.3.4 – Keyboard, memory and ADPCM control



These two registers are used to access the external music keyboard, where $05H is configured as an input and $06H as an output. Thus, the signal emitted by each bit of $06H can be read by each bit of $05H, forming an 8 x 8 matrix for the keyboard.



This register is used to control the start and stop of ADPCM and also to set access to audio memory. Each bit is a flag that performs a different operation, as described below.

b0: (RESET): When this bit is set to 1 during sound synthesis by ADPCM using audio memory as source, the ADPCM synthesis circuit and external audio memory are brought to their initial state. In this case, b4 (REPEAT) must be set to 0. This bit can be used when the ADPCM circuit or external memory loses control.

b1: Not used

b2: Not used

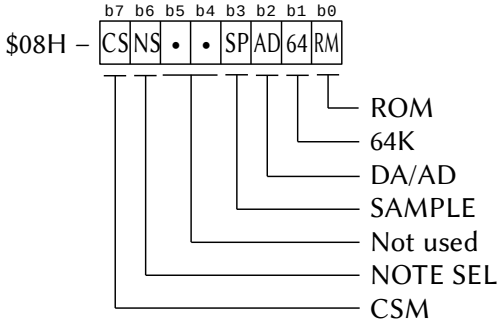
b3 (SP-OFF): When this bit is set to 1, the sound output terminal is off. This bit should be used to protect the speaker during ADPCM recording.

b4 (REPEAT): During sound synthesis by ADPCM using audio memory, this bit can be set to 1 to enable repetition of data from the same area (from start address to end address).

b5 (MEMORY DATA): This bit must be set to 1 when audio memory is to be accessed.

b6 (REC): This bit must be set to 1 for digitizing sounds by ADPCM or transferring data from CPU to audio memory.

b7 (START): This bit must be set to 1 for reading or writing data by ADPCM. The procedure differs depending on the location of the data (Main RAM or audio memory). If the data is in Main RAM, ADPCM will start reading or writing through register $0FH. If they are in audio memory, ADPCM will start accessing the specified start address. Consequently, all necessary registers must be loaded before setting this bit to 1.



This register is used together with $08H for ADPCM control and audio memory.

b0: (ROM): This bit is used to identify the type of audio memory: 0=RAM; 1=ROM.

b1: (64K): This bit is used to specify the amount of audio memory available (0 = 256K DRAM; 1 = 64K DRAM). When this bit is 1, the A8 address line is ignored. For ROM, this bit must be 0.

b2: (DA/AD): This bit is used in conjunction with b3 (SAMPLE) and controls the DA/AD conversion. When it is 1, the data specified in $15H and $16H is sent to the output. When set to 0, AD conversion (b3=1) or music output (b3=0) is enabled.

b3: (SAMPLE): This bit is used to enable the timer for AD/DA conversion. When it is 1, AD conversion starts; when it is 0, the D/A conversion is started.

b4: Not used.

b5: Not used.

b6 (NOTE SEL): This bit is used to specify an octave separation points for the external music keyboard. When 0, the separation point is specified by the two frequency MSB bits. When set to 1, the separation point is specified by the frequency MSB bit (registers $B0H to $B8H), as shown in the table below.

b6=0

0		1		2		3		4		5		6		7		Octave
0		1		2		3		4		5		6		7		Data block
1		1		1		1		1		1		1		1		F-num MSB
0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	2º F-num
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Keyboard split No.

b6=1

0		1		2		3		4		5		6		7		Octave
0		1		2		3		4		5		6		7		Data block
1		1		1		1		1		1		1		1		F-num MSB
*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	2º F-num
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Keyboard split No.

b7 (CSM): This bit must be set to 1 to activate composite sinusoidal modulation mode. For this, all voices must be set to Key-off.

6.5.3.5 – Access adresses

	b7	b6	b5	b4	b3	b2	b1	b0	
$09H –	b7	b6	b7	b6	b3	b2	b1	b0	Start adress (low)
$0AH –	b15	b14	b13	b12	b11	b10	b9	b8	Start adress (high)

These registers specify the start and end addresses of the audio memory to be accessed. The value of these registers differs slightly depending on the type of memory (ROM or DRAM). The way in which they specify addresses is illustrated below.

64K RAM (Start adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0AH -	- $09H -	
b7 b6 b5 0 b3 b2 b1 b0	b7 b6 b5 b4 0 b2 b1 b0	0 0 0 0 0

Note: b4 of $0AH and b3 of $09H must be "0".

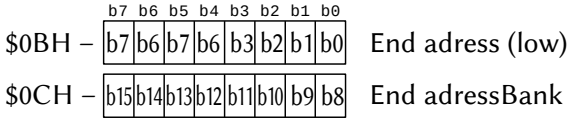
256K RAM (Start adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0AH -	- $09H -	
b7 b6 b5 b4 b3 b2 b1 b0	b7 b6 b5 b4 b3 b2 b1 b0	0 0 0 0 0

64K ROM (Start adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0AH -	- $09H -	
* * * b4 b3 b2 b1 b0	b7 b6 b5 b4 b3 b2 b1 b0	0 0 0 0 0

Note: The bits b7, b6 and b5 of $0AH must be equal to the $0CH.



64K RAM (End adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0CH -	- $0BH -	
b7 b6 b5 0 b3 b2 b1 b0	b7 b6 b5 b4 0 b2 b1 b0	1 1 1 1 1

Note: b4 of $0AH and b3 de $09H must be "0".

256K RAM (End adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0CH -	- $0BH -	
b7 b6 b5 b4 b3 b2 b1 b0	b7 b6 b5 b4 b3 b2 b1 b0	1 1 1 1 1

64K ROM (End adress)

Bank	CAS adress	RAS adress
b2 b1 b0	a8 a7 a6 a5 a4 a3 a2 a1 a0	a8 a7 a6 a5 a4 a3 a2 a1 a0
- $0CH -	- $0BH -	
* * * b4 b3 b2 b1 b0	b7 b6 b5 b4 b3 b2 b1 b0	1 1 1 1 1

Note: The bits b7, b6 and b5 of $0CH must be equal to the $0AH.

6.5.3.6 – Access to PCM and 4-bit I/O

Registers $0DH and $0EH specify the sampling rate for the ADPCM AD and DA conversion. The maximum rate is 16 KHz and the minimum rate is 1.8 KHz. The value to be loaded into the registers can be obtained using the formula below.

Rate (KHz) = 3580 / NPRE (@ ∅m = 3.58 MHz)

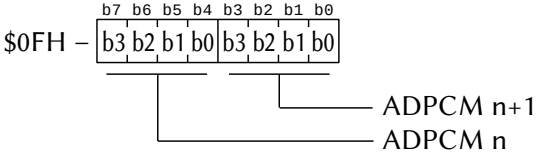
The NPRE value is the value contained in 11 bits of registers $0DH and $0EH, as illustrated below.

	b7	b6	b5	b4	b3	b2	b1	b0	
$0DH –	b7	b6	b7	b6	b3	b2	b1	b0	low
$0EH –	0	0	0	0	0	b10	b9	b8	high

16 KHz ⟶ $0DH = E0H and $0EH = 00H (freq max)

1,8 KHz ⟶ $0DH = C4H and $0EH = 07H (freq min)

Register $0FH is used to exchange data from ADPCM with the CPU. It is also used as a buffer when audio memory is used by the CPU. This register normally contains 2 data, as the ADPCM data is compressed into 4 bits each. The highest 4 bits contain data n and the lowest 4 bits contain data n+1.



Registers $10H and $11H specify the factor for linear interpolation with the 50 KHz frequency of the FM generator during the intervals of sound synthesis by the ADPCM. This factor is also used as the sampling rate for the synthesis, in which case the $0DH and $0EH registers are ignored. The formula for calculating the value of these registers is as follows:

$$\Delta n = k * 2^{16}, \quad k = \left(\frac{3.58\,MHz}{50\,KHz}\right) / \left(\frac{3.58\,MHz}{fsample}\right) \quad , \quad (@\ \varnothing m = 3{,}58\ MHz)$$

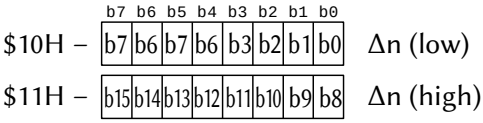
$$VOICE_{n.i} = VOICE_n + (Noff_n + i_n * k) * (VOICE_{n+1} - VOICE_n)$$

$$\begin{cases} 0 \le Noff_n + i_n * k < 1 \\ Noff_n < k, \quad Noff_n = Noff_{n-1} + i_{n-1} * k + k - 1 \end{cases}$$

(i'n-1 is the max value of the (n-1)th sampling)

But the following simplified formula can be used:

$$\Delta n = 1310{,}72 * \text{Sampling rate (Khz)}$$



16 Khz ⟶ $10H = ECH and $11H = 51H

1,8 KHz ⟶ $10H = 37H and $11H = 09H

The $12H register is the ADPCM output volume control. It has 256 levels, and the maximum volume is obtained when the register contains the value 255. The value 0 indicates null volume (no sound output). The value of this register does not affect the sound output of the FM generator.

$$AUDIO\ OUT = VOICE_n * EG$$

	b7	b6	b5	b4	b3	b2	b1	b0
$12H –	EG control							

Registers $15H to $17H are used to specify digital data for DA conversion. In the three registers, only 13 bits are used. Initial values must be written to these registers before setting bit 2 of register $08H. The values of these registers can be calculated by the following formulas:

$$\text{Output} = \frac{Vcc}{2} + \frac{Vcc}{4} * (-1 + f_9 + f_8 * 2^{-1} + \ldots + f_1 * 2^{-8} + f_0 * 2^{-9} + 2^{10}) * 2^{-E}$$

$$E = S_2 * 2^2 + S_1 * 2^1 + S_0 * 2^0 \qquad @\ S_0 + S_1 + S_2 > 0$$

	b7	b6	b5	b4	b3	b2	b1	b0
$15H –	f_9	f_8	f_7	f_6	f_5	f_4	f_3	f_2
$16H –	f_1	f_0	•	•	•	•	•	•
$17H –	•	•	•	•	•	S_2	S_1	S_0

Registers $18H and $19H are 4-bit registers used to control MSX-Audio's general-purpose I/O ports. $18H specifies whether it is input or output; must be set to 1 for output and 0 for input. $19H is used to transfer data over the 4-bit I/O port.

	b7	b6	b5	b4	b3	b2	b1	b0	
$18H –	•	•	•	•	b3	b2	b1	b0	I/O control
$19H –	•	•	•	•	b3	b2	b1	b0	I/O data

The $1AH register is used to store the data processed by the A/D conversion. The PCM code is expressed in two's complement, that is, the value 127 corresponds to level 0.

	b7	b6	b5	b4	b3	b2	b1	b0	
$1AH –	b7	b6	b7	b6	b3	b2	b1	b0	Dados PCM

6.5.3.7 – Access to the FM generator

MSX-Audio can generate up to 9 voices of FM sound or 6 voices plus 5 rhythm pieces like OPLL, but all 9 voices must be user defined.

Each voice uses two operators (referred to as modulated wave and carrier wave) to generate sounds, resulting in a total of 18 operators. The table below shows the relationship between operators, voices and registers.

01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	*1
1	2	3	1	2	3	4	5	6	4	5	6	7	8	9	7	8	9	*2
20	21	22	23	24	25	28	29	2A	2B	2C	2D	30	31	32	33	34	35	*3
A0	A1	A2	A0	A1	A2	A3	A4	A5	A3	A4	A5	A6	A7	A8	A6	A7	A8	*4

*1 – operator number used.

*2 – generated voice number.

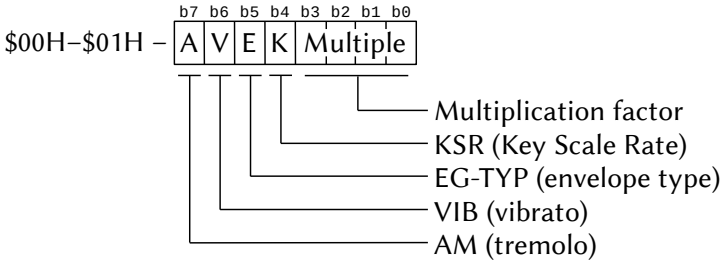
*3 – corresponding register (in example, $20H to $35H).

*4 – corresponding register (in the example, $A0H to $A8H).

The registers used for generating FM tones are described in detail below.

• AM/VIB/EG-TYP/KSR/MULTIPLE

These registers are used to specify the envelope shape and the multiplication factors for the carrier and modulator waves.



MULTIPLE (b0~b3)

These bits specify the multiplication factors used to convert the modulator and carrier waves. The multiplication factors can be seen in the table below.

Register value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Multiplication factor:	½	1	2	3	4	5	6	7	8	9	10	10	12	12	15	15

For example, if F-num is ωf, the factor for the carrier wave is 1 and for the modulated wave is 7, F(t) is calculated by the following formula:

$F(t) = E \sin(\omega ft + 1 \sin(7\omega ft))$

KSR (b4)

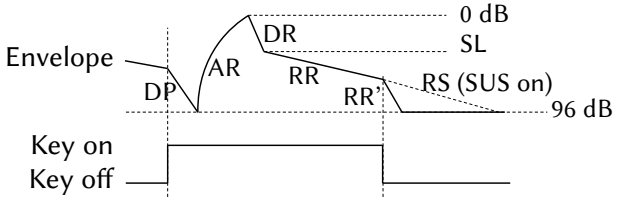
KSR stands for “key scale rate”. It specifies the "attack" and "decay" reasons. The “key scale” is used to make the sound generated by the FM approach that of acoustic musical instruments. The meaning of this bit is illustrated in the table below:

Key Scale		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Fatores	b4=0	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
	b4=1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

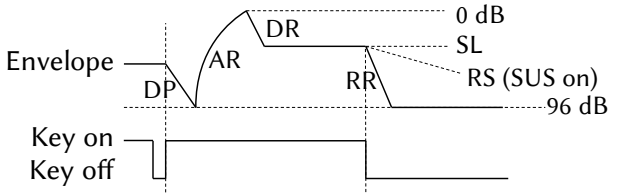
EG-TYP (b5)

This bit selects between constant tone or percussive tone. If it is 0, the pitch will be percussive and if it is 1 the pitch will be constant. The generated waveform varies as shown in the illustration below.

Percussive Tone (b5=0)



Non-percussive Tone (b5=1)



AR = Attack Rate DR = Decay Rate
SL = Sustain Level RR = Release Rate

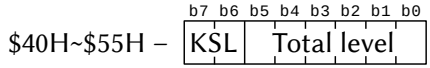
VIB (b6)

This bit turns vibrato on or off. If it is 1, vibrato is on and if it is 0 it is off. The vibrato frequency is 6.4 Hz. The depth of vibrato is set by the VIB-DEPTH bit of register $BDH.

AM (b7)

This bit turns amplitude modulation (tremolo) on or off. If it is 1 then amplitude modulation is on and if it is 0 it is off. The tremolo frequency is 3.7 Hz. The tremolo depth is set by the AM-DEPTH bit of register $BDH.

• KSL / TOTAL LEVEL



TOTAL LEVEL (b0~b5)

The six total level bits not used to control the depth of envelope modulation (envelope level). The table below shows all possible modulation depths.

b5	b4	b3	b2	b1	b0
24dB	12dB	6dB	3dB	1,5dB	0,75dB

To obtain the correct value, the depths must be added when the respective bit is 1. Thus, the maximum resolution of "decay" is 0.75 dB and the output level can be reduced up to 47.25 dB.

KSL (b6~b7)

These bits control the output level by progressively attenuating the sound (key scale level). The level of progressive sound attenuation by frequency can vary from 0 dB/octave to 6 dB/octave.

b6	b7	Attenuation
0	0	0 dB/octave
0	1	1,5 dB/octave
1	0	3 dB/octave
1	1	6 dB/octave

The table below describes decay factors in dB/octave according to F-number.

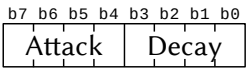
F-num ↓	Octave							
	0	1	2	3	4	5	6	7
0	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
1	0.000	0.000	0.000	0.000	0.000	3.000	6.000	9.000
2	0.000	0.000	0.000	0.000	3.000	6.000	9.000	12.000
3	0.000	0.000	0.000	1.875	4.875	7.875	10.875	13.875
4	0.000	0.000	0.000	3.000	6.000	9.000	12.000	15.000
5	0.000	0.000	1.125	4.125	7.125	10.125	13.125	16.125
6	0.000	0.000	1.875	4.875	7.875	10.875	13.875	16.875
7	0.000	0.000	2.625	5.625	8.625	11.625	14.625	17.625
8	0.000	0.000	3.000	6.000	9.000	12.000	15.000	18.000
9	0.000	0.750	3.750	6.750	9.750	12.750	15.750	18.750
10	0.000	1.125	4.125	7.125	10.125	13.125	16.125	19.125
11	0.000	1.500	4.500	7.500	10.500	13.500	16.500	19.500
12	0.000	1.875	4.875	7.875	10.875	13.875	16.875	19.875
13	0.000	2.250	5.250	8.250	11.250	14.250	17.250	20.250
14	0.000	2.625	5.625	8.625	11.625	14.625	17.625	20.625
15	0.000	3.000	6.000	9.000	12.000	15.000	18.000	21.000

Note: F-num indicates the highest 4 bits.

Multiply each value by 1/2 for 1.5 dB/octave decay.

Multiply each value by 2 for 6 dB/octave decay.

• ATTACK/DECAY RELATIONSHIP



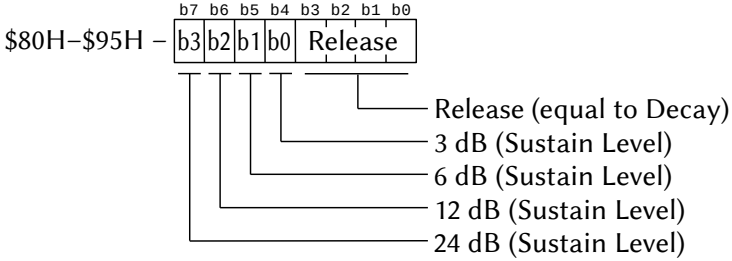
The “attack” and “decay” ratios are defined by registers $60H to $75H, with bits b0~b3 defining the level of “decay” and bits b4~b7 the level of “attack”. The higher the value, the shorter the “attack” and “decay” time. The time variation follows, approximately, a geometric progression, and its values are listed in the table below.

RATE		EG decay time (mS)		EG attack time (mS)	
RM	RL	0dB~96dB	10% ~ 90%	0dB~96dB	10% ~ 90%
15	3	2.40	0.51	0.00	0.00
15	2	2.40	0.51	0.00	0.00
15	1	2.40	0.51	0.00	0.00
15	0	2.40	0.51	0.00	0.00
14	3	2.74	0.58	0.20	0.11
14	2	3.20	0.63	0.24	0.11
14	1	3.84	0.81	0.30	0.14
14	0	4.80	1.01	0.38	0.19
13	3	5.48	1.15	0.42	0.22
13	2	6.40	1.55	0.46	0.26
13	1	7.68	1.62	0.56	0.31
13	0	9.60	2.02	0.70	0.37
12	3	10.96	2.32	0.80	0.43
12	2	12.80	2.68	0.92	0.49
12	1	15.36	3.22	1.12	0.61
12	0	19.20	4.02	1.40	0.73
11	3	21.92	4.62	1.56	0.85
11	2	25.56	5.38	1.84	0.97
11	1	30.68	6.42	2.20	1.13
11	0	38.36	8.02	2.76	1.45
10	3	43.84	9.24	3.12	1.70
10	2	51.12	10.76	3.68	1.94
10	1	61.36	12.84	4.40	2.26
10	0	76.72	16.04	5.52	2.90
9	3	87.68	18.48	6024	3.39
9	2	102.24	21.52	7.36	5.87
9	1	122.72	25.68	8.80	4.51
9	0	153.44	32.08	11.04	5.79
8	3	175.36	36.96	12.48	6.78
8	2	204.48	43.04	14.72	7.74
8	1	245.44	51.36	17.60	9.02
8	0	306.88	64.16	22.08	11.58
7	3	350.72	73.92	24.96	13.57

7	2	408.96	86.08	29.44	15.49
7	1	490.88	102.72	35.20	18.05
7	0	613.76	128.32	44.16	22.17
6	3	701.44	147.84	49.92	27.14
6	2	817.92	172.16	58.88	30.98
6	1	981.76	205.44	70.40	36.10
6	0	1227.52	256.64	88.32	45.74
5	3	1402.88	295.68	99.84	554.27
5	2	1635.84	344.32	117.76	61.95
5	1	1963.52	410.88	140.80	72.19
5	0	2455.04	513.28	176.64	92.67
4	3	2805.76	591.36	199.68	108.54
4	2	3271.68	688.64	235.52	123.90
4	1	3927.08	821.76	281.60	144.38
4	0	4910.08	1026.56	353.28	185.34
3	3	5611.52	1182.72	399.36	217.03
3	2	6543.36	1377.28	471.04	247.81
3	1	7854.08	1643.52	563.20	288.77
3	0	9820.16	2053.12	706.56	370.69
2	3	11 223.04	2365.44	798.72	434.18
2	2	13 086.72	2754.56	942.08	495.62
2	1	18 708.16	3287.24	1126.40	577.54
2	0	19 640.32	4106.24	1413.12	741.38
1	3	22 446.08	4730.88	1597.44	868.35
1	2	26 173.44	5509.12	1884.16	991.23
1	1	31 416.32	6574.08	2252.80	1155.07
1	0	39 280.64	8212.48	2826.24	1482.75

• SUSTAIN LEVEL / RELEASE RATE

"Sustain level" specifies the level at which the sound will be after the "decay rate". For the percussive tone, specifies the point to switch from "decay" mode to "release" mode. The higher the value of this register, the lower the "sustain" level.

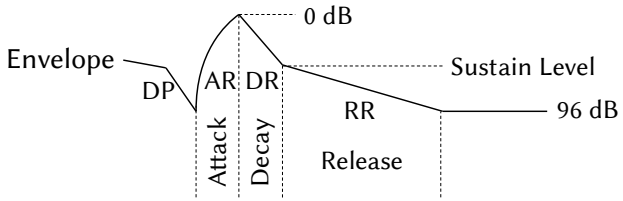


To obtain the correct value of "sustain", it is necessary to sum the values when the respective bit is "1".

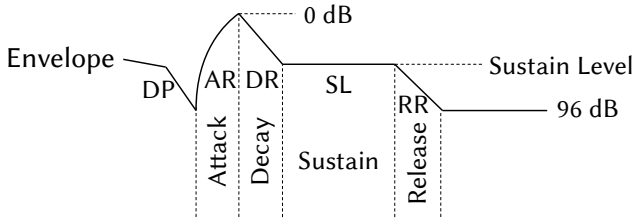
"Release rate", for constant pitch, specifies the "decay" rate after the "key off". For percussive tone, specifies the "decay" ratio after the "sustain level". The higher the value of the register, the shorter the duration of the release rate. The "release" value is specified with the same values used for "decay", described in the table above.

Below is an illustration of the "attack", "decay", "sustain level" and "release rate" values in the envelope waveform.

Percussive Tone (b5=0)

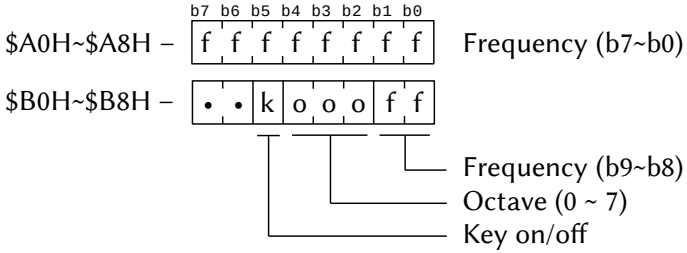


Non-percussive Tone (b5=1)



• OCTAVE/FREQUENCY

There are nine groups of two registers each, with registers $A0H and $B0H controlling the first voice, $A1H and $B1H controlling the second, and so on.



FREQUENCY ($AxH and bits 1 and 0 of $BxH)

These 10 bits define a frequency scale for each octave. In the table below, the register values for the fourth octave are specified, out of a total of 8 octaves, with the central note A of 440 Hz.

	Frequency	Decimal	$2xH,b1~b0	$1xH
C#	277.2 Hz	363	0 1	0 1 1 0 1 0 1 1
D	293.7 Hz	385	0 1	1 0 0 0 0 0 0 1
D#	311.1 Hz	408	0 1	1 0 0 1 1 0 0 0
E	329.6 Hz	432	0 1	1 0 1 1 0 0 0 0
F	349.2 Hz	458	0 1	1 1 0 0 1 0 1 0
F#	370.0 Hz	485	0 1	1 1 1 0 0 1 0 1
G	392.0 Hz	514	1 0	0 0 0 0 0 0 1 0
G#	415.3 Hz	544	1 0	0 0 1 0 0 0 0 0
A	440.0 Hz	577	1 0	0 1 0 0 0 0 0 1
A#	466.2 Hz	611	1 0	0 1 1 0 0 0 1 1
B	493.9 Hz	647	1 0	1 0 0 0 0 1 1 1
C	523.3 Hz	686	1 0	1 0 1 0 1 1 1 0

The frequency values hold a geometric relationship equal to the 12th root of 2, which is 1.059463094 3592. You can use this number to change the register values to increase or decrease the generated frequency within the musical scale. The values of the registers also have the same relationship to each other.

The values specified by the f0 ~ f9 bits are called "F-number" and their values can be calculated by the following formulas:

F-num = (fmus * 219 / fsam) / 2^{b-1}

Where:

F-num = given F-number
fmus = Desired frequency
fsam = Sampling frequency (50 KHz)
b = Block (octave)

The sampling frequency corresponds to the clock frequency (3.58 MHz on a standard MSX) divided by 72, which more accurately results in 49.722 KHz.

OCTAVE ($BxH, b4~b2)

These three bits define the octave. Up to 8 octaves can be set, from 000 to 111, with the fourth octave being 011.

KEY($BxH, b5)

This bit must be set to 1 for the sound of each of the nine voices to be enabled. When it is 0, the sound of the respective voice will be off (key off).

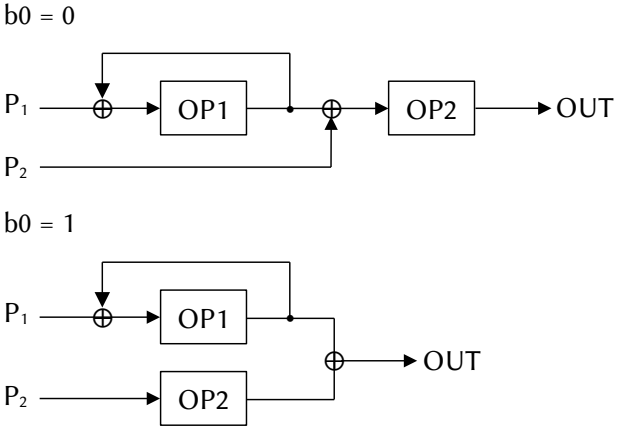
• FEEDBACK/CONNECTION

$C0H~$C8H –

b7	b6	b5	b4	b3	b2	b1	b0
•	•	•	•	R	R	R	C

CONNECTION ($CxH, b0)

This bit is used to specify the type of connection between the two FM operators (modulator wave and carrier wave). If 0, operators are in FM mode. If it is 1, they will be in composite sinusoidal modulation mode (in parallel). These modes are illustrated below.

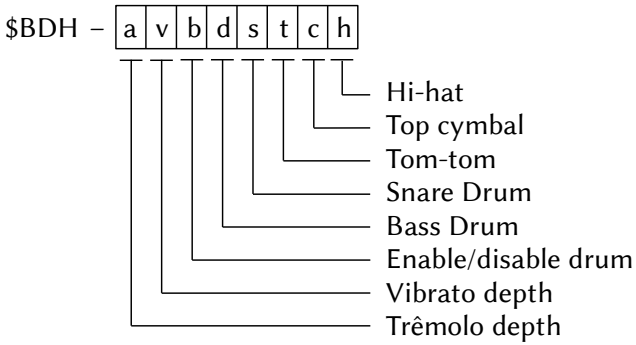


FEEDBACK ($CxH, b1~b3)

These bits specify the feedback index (portion of the output signal that is re-injected into the input) for the modulated wave. The larger the register value, the larger the feedback factor. The factors are shown in the table below.

Register value:	000	001	010	011	100	101	110	111
Feedback value:	0	π/16	π/8	π/4	π/2	π	2π	4π

• RHYTHM / AM.VIB–DEPTH



RHYTHM MODE (b0~b5)

To activate MSX-Audio's rhythm mode, just set bit 5 of $BDH to 1. Bits b0 to b4 enable (1) or disable (0) each of the five available rhythm parts. In rhythm mode, only the first six MSX-Audio voices are available to the programmer.

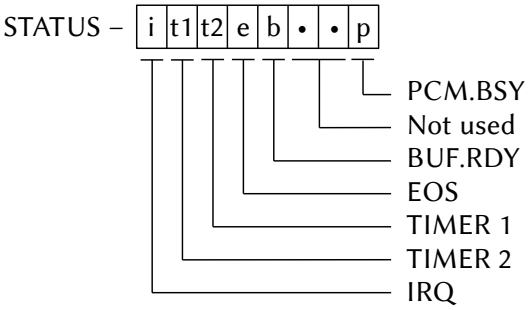
VIB-DEPTH (b6)

This bit is used to select the depth of vibrato. If it is 0, the depth will be 7% and if it is 1, 14%.

AM-DEPTH (b7)

This bit is used to select the depth of amplitude modulation (tremolo). If it is 0, the depth will be 1 dB and if it is 1, 4.8 dB.

6.5.3.8 – The status register



MSX-Audio has a state register with flags to control two timers and the audio memory, used during the synthesis or analysis of sounds by ADPCM. It is a read-only register.

b0 (PCM.BSY): During sound analysis or synthesis by ADPCM, this bit will be set to 1 if bit b7 of $07H. No interrupt signal is generated.

b1: Not used.

b2: Not used.

b3 (BUF.RDY): This bit will be set to 1 in the following cases:
→ End of sound analysis by ADPCM ($07H, b5=0).
→ End of sound synthesis by ADPCM ($07H, b5=0).

→ End of writing to the audio memory.
→ End of audio memory reading.

b4 (EOS): This bit will be set to 1 when sound analysis or synthesis by ADPCM is completed or when there is a time lapse during AD/DA conversion.

b5 (Timer 2): This bit is set to 1 after the time lapse generated by timer 2.

b6 (Timer 1): Same as b5, but set by timer 1.

b7 (IRQ): This bit will be set to 1 when one or more of the bits b3 to b6 is or is 1.

6.5.4 – Access to audio memory and ADPCM

Recommended protocols for accessing audio memory and ADPCM are described below.

6.5.4.1 – Sound analysis (MSX-Audio → CPU)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	00H	W	All flags are enabled
$04H	80H	W	All flags are reset
$07H	C8H	W	ADPCM is enabled and sound output is turned
$08H	00H	W	off
$0DH	C2H	W	"Sampling rate" = 8 KHz (NPRE = 450)
$0EH	01H	W	• Start analysis
$0FH	80H	R	Starts with reading the "dummy"
$0FH	48H	R	• Analysis
($04H	00H	W)	When BUF.RDY is 1, $0FH is read, data is stored
$07H		W	nado and the flag is reset. When BUF.RDY is 0,
$07H		W	wait.
			• End of analysis
			The analysis by ADPCM was completed
			Register $07H is reset

6.5.4.2 – Sound synthesis (CPU → MSX-Audio)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	00H	W	All flags are enabled
$04H	80H	W	All flags are reset
$07H	80H	W	Sound synthesis by ADPCM is enabled
$08H	00H	W	
$10H	F6H	W	"Sampling rate" = 8 KHz (Δn = 10 486)
$11H	28H	W	
$12H	xxH	W	Specify output volume
			• Start of synthesis
$0FH	yyH	W	Write data to ADPCM in $0FH
			• Synthesis
$0FH	zzH	W	When BUF.RDY is 1, the synthesis data is written
($04H	80H	W)	to $0FH and the flag is reset. When flag is 0, wait.
			• End of synthesis
$07H	80H		Synthesis by ADPCM was completed.

6.5.4.3 – Sound analysis (MSX-Audio → Audio memory)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	08H	W	Only the BUF.RDY flag is masked
$04H	80H	W	All flags are reset
$07H	68H	W	Analysis by ADPCM is enabled
$08H	02H/00H	W	Specifying the audio memory type
$09H	xxH	W	Audio memory start address
$0AH	xxH	W	End address of audio memory
$0BH	yyH	W	"Sampling rate" = 16 KHz (NPRE = 225)
$0CH	yyH	W	• Start of analysis
$0DH	E1H	W	Start when bit b0 of $07H is 1.
$0EH	00H	W	• Analysis
$07H	E8H	W	The EOS flag is set to 1 until the end of the analy-
$07H	68H	W	sis
$07H	00H	W	• End of analysis
			The analysis by ADPCM was completed
			Address $07H is reset

6.5.4.4 – Sound synthesis (Audio memory → MSX-Audio)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	08H	W	Only the BUF.RDY flag is masked
$04H	80H	W	All flags are reset
$07H	20H/30H	W	Synthesis by ADPCM is enabled
$08H	00H-02H	W	Specifying the audio memory type
$09H	xxH	W	Audio memory start address
$0AH	xxH	W	
$0BH	yyH	W	End address of audio memory
$0CH	yyH	W	
$10H	ECH	W	“Sampling rate” = 16 KHz (Δn = 20 992)
$11H	51H	W	
$12H	xxH	W	Specify output volume
			• Start of synthesis
$07H	A0H/B0H	W	Start when b7 of $07H is 1
			• Synthesis
			The EOS flag is set to 1 until the end of the syn-
($07H	A0H	W)	thesis
($07H	A1H	W)	(Repeat mode is set)
			(Force interruption of synthesis)
$07H	20H	W	• End of synthesis
$07H	00H	W	Synthesis by ADPCM was completed
			Register $07H is reset

6.5.4.5 – Write in the audio RAM (CPU → Audio memory)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	00H	W	All flags are enabled
$04H	80H	W	All flags are reset
$07H	60H	W	Memory write mode is set
$08H	00H/02H	W	Specifies the type of RAM
$09H	xxH	W	Audio memory start address
$0AH	xxH	W	
$0BH	yyH	W	End address of audio memory
$0CH	yyH	W	

			• Memory writing
$0FH	zzH	W	Data byte to be written
($04H	80H	W)	(When BUF.RDY is 1, the data is written; when is 0, wait. When the end of memory is reached ged, the EOS flag will be 1)
			• Reset
$07H	00H	W	Register $07H is reset

6.5.4.6 – Read from RAM/ROM (Audio memory → CPU)

Reg.	Dado	R/W	Comentários
			• Initialization
$04H	00H	W	All flags are enabled
$04H	80H	W	All flags are reset
$07H	20H	W	Memory read mode is established
$08H	00H-02H	W	Specifies the type of memory
$09H	xxH	W	Audio memory start address
$0AH	xxH	W	
$0BH	yyH	W	End address of audio memory
$0CH	yyH	W	
			• Memory reading
$0FH		R	Start after reading “dummy” twice
$0FH		R	(Necessary to check the flag)
$0FH	zzH	R	Data byte reading
$04H	80H	W	When BUF.RDY is 1, the data is read; when 0, wait. When the end of memory is reached, the EOS flag will be 1)
			• Reset
$07H	00H	W	Register $07H is reset

6.5.5 – Access to MSX-Audio

Access to MSX-Audio can be done directly or through MBIOS (Music BIOS).

6.5.5.1 – Direct Access

Direct access to MSX-Audio is via two CPU I/O ports, C0H and C1H. Port C0H selects registers or reads the state register and port C1H reads or writes data in other registers. However, like OPLL, MSX-Audio

is slow. There must be a pause between one access and another. The time of each pause is described in the table below.

Register selection	(C0H)	3.4 µs	12 T cycles (3.58 MHz)
Access to $00H to $1AH	(C1H)	3.4 µs	12 T cycles (3.58 MHz)
Access to regs $20H to $C8H	(C1H)	23.5 µs	84 T cycles (3.58 MHz)

It is recommended to use pauses like "EX (SP),HL" or "NOP" until the MSX-Audio is ready for new access.

First, you must select the register to be written through port 7CH. After writing, there must be a pause of at least 12 T cycles in the case of a standard MSX (Z80 at 3.58 MHz). The instruction "OUT (07CH),A" takes 11 T cycles to be processed; therefore, at least 1 more T cycle is required. A NOP instruction (which takes 4 T cycles to be processed) can be used for this, as shown below:

```
LD   A,REG      ; A ← register number
OUT  (07CH),A   ; Select register
NOP             ; Pause
```

Soon after, the data is written or read in the selected register through the C1H port, observing the pauses. To read the status register, there is no need to specify addresses. To write a byte of data in registers $00H to $1AH, proceed as illustrated below.

```
LD   A,REG    ; Register number (00H a 1AH)
OUT  (0C0H),A ; Select register
NOP           ; Pause
LD   A,DATA   ; Byte of data to be written
OUT  (0C1H),A ; Write data in the register
NOP           ; Pause
```

To write to registers $20H to $C8H, which are much slower, a longer pause is needed, 84 T cycles for a clock of 3.58 MHz. For this, 4 "EX (SP),HL" instructions can be used (which take 19 T cycles each to be processed), resulting in a pause of 76 T cycles which, added to the 11 cycles of the OUT instruction, result in 87 T cycles. Then MSX-Audio will be ready to receive new data.

```
LD   A,REG     ; Register number (00H a 1AH)
OUT  (0C0H),A  ; Select register
NOP            ; Pause
LD   A,DATA    ; Byte of data to be written
OUT  (0C1H),A  ; Write data in the register
EX   (SP),HL   ; Pause
EX   (SP),HL   ; Pause
EX   (SP),HL   ; Pause
EX   (SP),HL   ; Pause
```

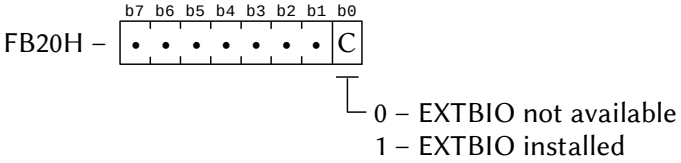
When used in pauses, the EX (SP),HL instruction must always come in pairs, to prevent the stack contents from being altered.

Registers $00H to $1AH and the state register are readable. In this case, proceed as illustrated below. Here there should be a pause of 12 T cycles as illustrated below.

```
LD    A,REG     ; register number (00H a 1AH)
OUT   (0C0H),A  ; select o register
NOP             ; pause
IN    A,(0C1H)  ; read register value
NOP             ; pause
IN    A,(0C0H)  ; read satus register
NOP             ; pause
```

6.5.5.2 – Access through Music BIOS

Access through the MBIOS, or Music BIOS, is done by the EXTBIO (FFCAH) routine on the work area, used to expand the BIOS residing in ROM. Initially it is necessary to consult the system variable HOKVLD (FB20H, 1) to confirm that the extended BIOS is installed. Bit 0 of this byte indicates the presence of an extended BIOS. If it is 0, there is no extended BIOS. If it is 1 there is at least one BIOS that can be called at address 0FFCAH (EXTBIO).



The next step is to check if there are MSX-Audio cartridges connected. This is done through the EXTBIO routine itself:

EXTBIO (FFCAH/Work Area)
  Function: Access extended BIOS functions.
  Input: A = 00H.
  D = 00H – Internal command.
  E = 00H – Examines the devices present in the system.
  B – ID of the slot where the table will be placed.
  HL – Table address.
  Output: B – Table slot ID.
  HL – Address of the next byte after the table.
  CY = 1 if there are no devices.
  Registers: All.

Then the table pointed to by the B and HL registers must be examined to verify the presence of the MSX Audio device identifier, which is 0AH. The table starts at the address pointed to by HL in the routine's entry and ends one byte before the address pointed to by HL in the return. If there is the ID, we must get how many MSX Audio cartridges are available, as follows:

EXTBIO (FFCAH/Work Area)
  Function: Access extended BIOS functions
  Input: A = 00H.
  D = 0AH – MSX-Audio manipulation device.
  E = 01H – Returns how many MSX-Audio cartridges are connected to the MSX (maximum 2).
  Output: A – 0 ⟶ No MSX-Audio connected.
  1 ⟶ There is an MSX-Audio cartridge connected.
  2 ⟶ There are two MSX-Audio cartridges connected.
  Registers: BC, DE, HL.

Then we must obtain the list of entry points of the available MBIOS routines, which is also done through EXTBIO as follows:

EXTBIO (FFCAH/Work Area)
  Function: Access extended BIOS functions
  Input: A = 00H.
  D = 0AH – MSX-Audio manipulation device.

E = 00H – Returns the pointer to the MSX-Audio information table.
B – Address table slot ID.
HL – Address of a 64-byte buffer for the table (should be on page 3).

Output: B – Information table slot ID.
HL – HL is incremented by 4 and will point to the end of a table that reserves 4 bytes for MSX-Audio. The original value of HL points to the beginning of the table, which has the following structure:

```
+00H - Slot ID
+01H - Lower adress
+02H - Higher adress
+03H - Reserved for expansion
```

The slot ID (+00H) and address (+01H,+02H) will point to a table with the following structure:

```
+00H VERSION Software version
+03H MBIOS   Music BIOS
+06H AUDIO   MSX-Audio startup
+09H SYNTHE  Calls SYNTHE app
+0CH PLAYF   PLAY statement status
+0FH BGM     Enable/disabel BGM mode
+12H MKTEMP  Set record/play time of the
             musical keyboard
+15H PLAYMK  Play by musical keyboard
+18H RECMK   Record from musical keyboard
+1BH STOPM   Play/rec keyboard/ADPCM;
             stops PLAY statement
+1EH CONTMK  Continue record from
             musical keyboard
+21H RECMOD  Configure rec mode
             of the musical keyboard
+24H STPPLY  Stop PLAY statement
+27H SETPCM  ADPCM/PCM protected area
+2AH RECPCM  ADPCM/PCM recording
+2DH PLAYPCM ADPCM/PCM playing
+30H PCMFREQ Change ADPCM/PCM sampling
             rate play frequency
+33H MKPCM   Configure/cancel ADPCM data
             for musical keyboard
```

```
+36H PCMVOL  ADPCM/PCM volume play
+39H SAVEPCM Save ADPCM/PCM data
+3CH LOADPCM Load ADPCM/PCM data
+3FH COPYPCM Transfer ADPCM/PCM data
+42H CONVP   Converts ADPCM -> PCM data
+45H CONVA   Converts PCM -> ADPCM data
+48H VOICE   Configure FM data
+4BH VOICECOPY Move FM data
```

Registers: F.

In the Appendix, item "8.5.5 – MSX-Audio", all available MBIOS routines are described with details.

6.6 – THE SCC

The SCC (Sound Creative Chip) is an audio generator created by the Japanese softhouse Konami to equip their megarom game cartridges. There are two types of SCC, the "simple" SCC and the SCC+.

The SCC generates sounds by recording the waveform in its internal memory and its reproduction takes place just like the PCM. The difference is that the memory reserved for sounds is very limited, only 32 bytes per voice, which must be repeated continuously during synthesis.

6.6.1 – The "simple" SCC

This SCC has internally 256 bytes of memory to store the data referring to each voice, which are five in number. Below is the memory reserved for each voice.

a – 32 bytes ⟶ waveform
b – 12 bits ⟶ playback frequency
c – 4 bits ⟶ output volume
d – 1 bit ⟶ turn voice on/off

The memory distribution of data for each of the five voices is illustrated in the table below.

Adresses	Short Description (SCC)
9800H~981FH	Waveform of the voice #1
9820H~983FH	Waveform of the voice #2
9840H~985FH	Waveform of the voice #3
9860H~987FH	SCC : Write/read: Waveform of the voices #4 e #5 SCC+: Read: Waveform of the voice #4
9880H~9881H	Frequency of the voice #1
9882H~9883H	Frequency of the voice #2
9884H~9885H	Frequency of the voice #3
9886H~9887H	Frequency of the voice #4
9888H~9889H	Frequency of the voice #5
	Example: 9880=|f7|f6|f5|f4|f3|f2|f1|f0| 9881=|•|•|•|•|f11|f10|f9|f8| $Ftone = \frac{F\_clock}{32 * ((f11\sim f0)+1)}$ (F_clock=3,579545 MHz)
988AH	Volume of the voice #1 (0 to 15)
988BH	Volume of the voice #2 (0 to 15)
988CH	Volume of the voice #3 (0 to 15)
988DH	Volume of the voice #4 (0 to 15)
988EH	Volume of the voice #5 (0 to 15)
998FH	|•|•|•|v5|v4|v3|v2|v1| v5=1 ⟶ turn on voice #5 v4=1 ⟶ turn on voice #4, etc
9890H~989FH	Mirror of 9880H~988FH
98A0H	SCC: no function SCC+: waveform read of the voice #5 (no write alowed)
98A1H~98BFH	Mirrors of 98A0H
98C0H	SCC: Mirror of 98A0H SCC+: Deformation register
98C1H~98DFH	SCC: Mirror of 98A0H SCC+: Mirror of 98C0H

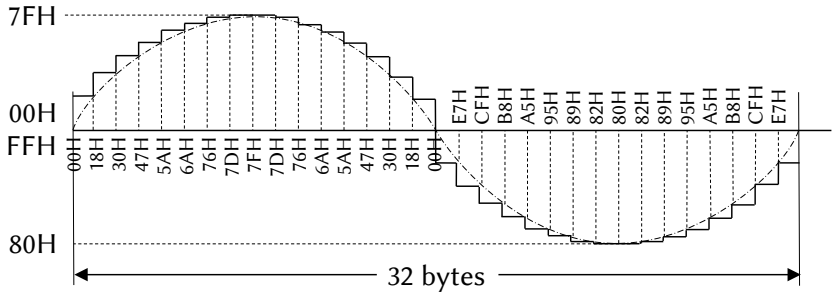
98E0H	SCC: Deformation register R R • • • • P P PP: 11/10→Ftone*16; 01→Ftone*256; 00→ Ftone*1 RR: 11 → white noise voices 4 and 5 cfe waveform 01 → continuous white noise 00 → no white noise SCC+: No function

6.6.1.1 – Waveform

The 32 bytes reserved for the waveform store it in two's complement: from 0 to 127 (00H to 7FH) the amplitude is increased; from –1 to –128 (FFH to 80H) the amplitude is decreased. The sequence of bytes to be stored for a sine waveform is illustrated in the table below.

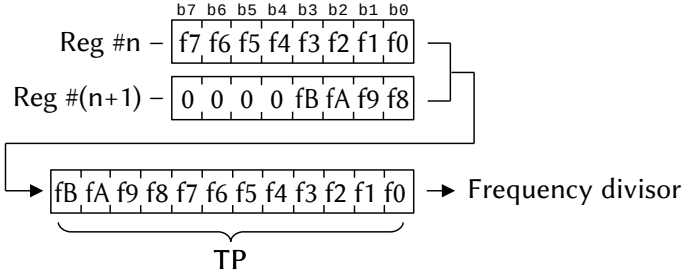
byte	value	byte	value	byte	value	byte	value
0	00H	8	7FH	16	00H	24	80H
1	18H	9	7DH	17	E7H	25	82H
2	30H	10	76H	18	CFH	26	89H
3	47H	11	6AH	19	B8H	27	95H
4	5AH	12	5AH	20	A5H	28	A5H
5	6AH	13	47H	21	95H	29	B8H
6	76H	14	30H	22	89H	30	CFH
7	7DH	15	18H	23	82H	31	E7H

The resulting waveform will be as follows:



6.6.1.2 – Frequency adjustement

The waveform frequency is set in the same format as for PSG, as illustrated below.



The value stored in TP is the period. Thus, the higher the TP value, the lower the frequency. The formula used to calculate the frequency is as follows:

$$\text{F-tone} = \frac{\text{F-clock}}{32 * (\text{TP} + 1)}$$

Where: F-clock is the clock present on the internal bus (usually 3.579545 MHz);

F-tone is the frequency generated in the respective voice;

TP is the value stored in the frequency registers.

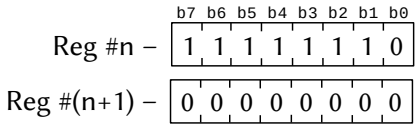
The formula derived to calculate the TP value from the frequency is:

$$\text{TP} = \frac{\text{F-clock}}{\text{F-tone} * 32} - 1$$

Thus, to obtain the central A note of 440 Hz, we have:

$$\text{TP} = \frac{\text{F-clock}}{\text{F-tone} * 32} - 1 = \frac{3\,579\,545}{440 * 32} = \frac{3\,579\,545}{14\,080} = 254$$

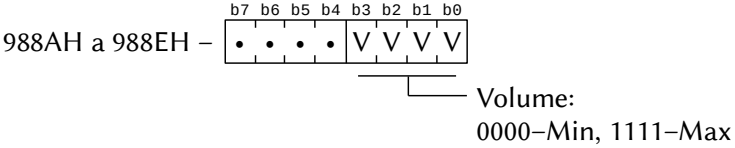
or 0FEH in hexadecimal. The registers will be:



It should be noted here that the frequency cycle is a complete passage through the 32 bytes that define the waveform. Thus, if F-tone equals 10, 10 passes per second will be made in all 32 bytes of the waveform register.

6.6.1.3 – Volume adjustement

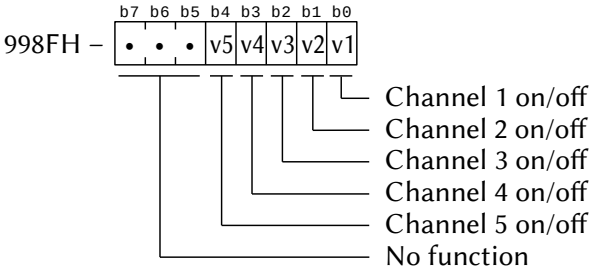
The volume is also stored in the same format as for PSG, as illustrated below.



When this register is 0, sound will be absent; when it is 15, the volume will be maximum.

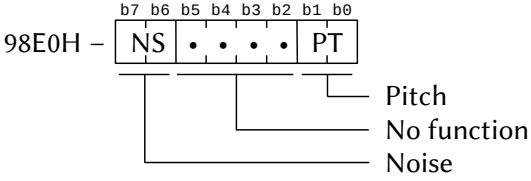
6.6.1.4 – Key register

The key register is mapped as illustrated below (if the respective bit is 0, turn off the voice; if it is 1, turn on the voice).



6.6.1.5 – Deformation register

All addresses from 98E0H to 98FFH refer to the same register. Its structure is as follows:

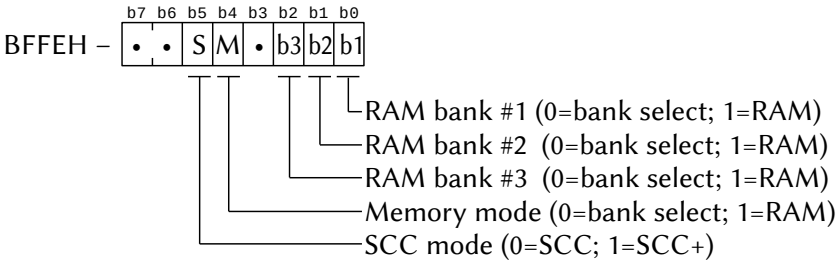


Pitch bits change the frequency of all voices. If it is 11B or 10B, the frequencies will be multiplied by 16. If it is 01B, the frequencies will be multiplied by 256. If it is 00B, the frequencies will not be affected.

Noise bits cause voices 4 and 5 to produce white noise. If they are 01B, they will produce continuous noise. If they are 11B, they will produce noise according to the defined waveform (with envelope). If they are 00B, there will be no noise (voices 4 and 5 will only generate the waveform).

6.6.2 – The SCC+

SCC+ is a little more elaborate than the "simple" SCC. In addition to generating sounds, it also maps up to 128 Kbytes of RAM. In this section, only the sound generation part will be dealt with. The SCC+ has a mode register that can be accessed at addresses BFFEH and BFFFH. Its structure is as follows:



To use SCC+ in "simple" SCC mode, SCC must be selected in the mode register, as well as bank 3 mode in the RAM bank selector. Simply write 00H to the mode register (selected value of the reset) and write the value 00 111 111B to the bank 3 selector register (either address 9000H or 97FFH). If bank 3 is in RAM mode, the SCC can be read but not written. Even so, there are some differences in the memory map, as described below.

Adresses	Short Description (SCC)
9800H~981FH	Waveform of the voice #1
9820H~983FH	Waveform of the voice #2
9840H~985FH	Waveform of the voice #3

9860H~987FH	SCC : Write/read: Waveform of the voices #4 e #5 SCC+: Read: Waveform of the voice #4
9880H~9881H	Frequency of the voice #1
9882H~9883H	Frequency of the voice #2
9884H~9885H	Frequency of the voice #3
9886H~9887H	Frequency of the voice #4
9888H~9889H	Frequency of the voice #5
	Example: 9880=|f7|f6|f5|f4|f3|f2|f1|f0| 9881=|•|•|•|•|f11|f10|f9|f8| $Ftone = \frac{F\_clock}{32 * ((f11\sim f0)+1)}$ (F_clock=3,579545 MHz)
988AH	Volume of the voice #1 (0 to 15)
988BH	Volume of the voice #2 (0 to 15)
988CH	Volume of the voice #3 (0 to 15)
988DH	Volume of the voice #4 (0 to 15)
988EH	Volume of the voice #5 (0 to 15)
998FH	|•|•|•|v5|v4|v3|v2|v1| v5=1 ⟶ turn on voice #5 v4=1 ⟶ turn on voice #4, etc
9890H~989FH	Mirror of 9880H~988FH
98A0H	SCC: no function SCC+: waveform read of the voice #5 (no write alowed)
98A1H~98BFH	Mirrors of 98A0H

In SCC mode, "simple" SCC is fully supported, except for the deformation register address.

To use SCC+ mode, it is necessary to set the respective bit in the mode register and write the value 10 000 000B (80H) in the bank 4 selector register (either address B000H or B7FFH). If bank 4 is in RAM mode, SCC+ can be read but not written. The SCC+ will appear in the memory area B800H to B8FFH as shown in the table below.

Adresses	Short Description (SCC+)
B800H~B81FH	Voice #1 waveform
B820H~B83FH	Voice #2 waveform
B840H~B85FH	Voice #3 waveform
B860H~B87FH	Voice #4 waveform
B880H~B89FH	Voice #5 waveform
B8A0H~B8A1H	Voice #1 frequency
B8A2H~B8A3H	Voice #2 frequency
B8A4H~B8A5H	Voice #3 frequency
B8A6H~B8A7H	Voice #4 frequency
B8AAH	Voice #1 volume (0 to 15)
B8ABH	Voice #2 volume (0 to 15)
B8ACH	Voice #3 volume (0 to 15)
B8ADH	Voice #4 volume (0 to 15)
B8AEH	Voice #5 volume (0 to 15)
B8AFH	Key register
B8B0H~B8BFH	Mirror of B8A0H~B8AFH
B8C0H	Deformation register
B8C1H~B8DFH	Mirrors of B8C0H
B8E0H~B8FFH	No function

The contents of the registers are exactly the same as in the "simple" SCC. The difference between them is the access addresses and the fact that voices 4 and 5 have separate registers for waveform definition, in the case of SCC+.

6.6.3 – Access to SCC

To access the SCC, simply select the slot where it is installed and write or read the data directly to the memory addresses. The SCC is not accessed through I/O ports. However, the memory area between 9880H and 98FFH is write-only; if it is read, it will always return FFH. The memory area between 9900H and 99FFH is a mirror of the area between 9800H and 98FFH. The same is true between areas 9A00H to 9AFFH and 9F00H to 9FFFH. This is because SCC does not use address lines A8~A10; therefore, it cannot distinguish address 9900H from 9800H or 9F00H from 9800H. The memory area from 8000H to 97FFH is usually a part of the ROM that normally comes with cartridges.

6.6.4 – Detecting the SCC

To check if there is any SCC cartridge connected, follow these steps:

1. Select the slot on page 2;
2. Select ROM page 63;
3. Check if you can read/write to a RAM location of wave shape;
4. select a ROM page other than 63 (2 for example);
5. Verify that the wave RAM location is read-only.

If all is successful, this is a slot with an SCC. Repeat this test for all possible slots.

The following routine for detecting SCC was written by Nyyrikki and published on msx.org:

```
;-----------------------------------------
; SCCDETECT (Made by : NYYRIKKI)
; Input: (None)
; Output:
;   Success:
;     CF = NC
;     SCC SLOT #8000-#BFFF
;     A = SlotID
;
;   Fail:
;     CF = C
;     (Random slot on #8000-#BFFF)
; Changes: All registers
;-----------------------------------------
;
SCCDETECT:
         LD D,#FF
         LD HL,#FCC1

.MAINL:
         INC D
         LD A,4
         CP D
         SCF
         RET Z
```

```
         LD A,(HL)
         INC HL
         AND #80
         JR Z,.MAINTST
         CALL .SUBTST
         JR C,.MAINL
         RET

.MAINTST:
         LD A,D
         CALL .TESTSLOT
         JR C,.MAINL
         LD A,D
         RET
;-----------------------
.SUBTST
         LD E,#FC
.SUBLOOP:
         LD A,4
         ADD A,E
         CP 16
         SCF
         RET Z
         LD E,A
         OR D
         OR #80
         LD C,A
         CALL .TESTSLOT
         JR C,.SUBLOOP
         LD A,C
         RET
.TESTSLOT:
         PUSH BC
         PUSH DE
         PUSH HL

         LD H,#80
         CALL #24

         LD A,(#9000)
         LD D,A
```

```
          LD A,#3F
          LD (#9000),A
          LD HL,#9800
          LD E,(HL)
          XOR A
          LD (HL),A
          LD A,(HL)
          OR A
          JR NZ,.NOSCC
          DEC A
          LD (HL),A
          LD A,(HL)
          INC A
          JR NZ,.NOSCC
          LD A,(#9000)
          CP #3F
          JR Z,.NOSCC
          XOR A
          JR .EXIT
.NOSCC
          LD (HL),E
          LD A,D
          LD (#9000),A
          SCF
.EXIT
          POP HL
          POP DE
          POP BC
          RET
```

6.7 – THE OPL4

OPL4 is not standard on MSX, but in terms of sound reproduction it is the most perfect, having audio CD quality. Currently, it can be found in several cartridges, but the pioneer was Moonsound, developed by Sunrise. To date (2022), the following Moonsound clones have been developed:

- Wozblaster, Argentina, by Gustavo Iriarte
- Shockwave/Shockwave 2, Brazil, by Tecnobytes
- Repro Factory Monster Sound FM Blaster, France
- JunSoft DalSoRi / JunSoft DalSoRi R2.0, Korea

- 8bits4ever MSX-Blaster, Spain
- MSX Calamar Wozblaster Enhanced, Spain

The responsible chip is the YMF278B. The OPL4 has 18 FM voices, all resettable, or 15 voices plus 5 rhythm parts, and 24 CD-quality PCM voices (16-bit resolution at 44.1 KHz sampling rate). It also has full stereo output. The OPL4 does not have an internal sound digitizer.

The OPL4 has 250 8-bit registers, numbered from $00H to $F9H, but its configuration varies depending on the type of playback (FM or Wave).

6.7.1 – Description of registers for wave synthesis

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H $01H	Test								Test registers
$02H	ID disp			Wave header			MT	MM	Special functions
	b7~b5 b4~b2 b1 b0	OPL4-ID (b7=0; b6=0; b7=1) Wave table header: 000=0 to 511 (000 000H) 100=384 to 511 (200 000H) 001=384 to 511 (080 000H) 101=384 to 511 (280 000H) 010=384 to 511 (100 000H) 110=384 to 511 (300 000H) 011=384 to 511 (180 000H) 111=384 to 511 (380 000H) Audio memory type (0=ROM; 1=RAM) Audio memnry access (0=OPL4; 1=CPU)							
$03H	•	•	a21	a20	a19	a18	a17	a16	Audio memory adress
$04H	a15	a14	a13	a12	a11	a10	a9	a8	
$05H	a7	a6	a5	a4	a3	a2	a1	a0	
$06H	Memory data								Data register
$08H ⋮ $1FH	Wave table number LSB (n7~n0)								24 registers with the LSB number (n7~n0) of the wave table
$20H ⋮ $37H	F_number (f6~f0)							Tab Wave (n8)	24 registers with the 7 bits LSB frequency and MSB wave table number (n8)

$38H ⋮ $4FH	Octave (o3~o0)		Pseudo-rev	F_number (f9~f7)	Octave (–7 a +7) Pseudo-reverberation Frequency (3 bits MSB)
	b7~b0 b7~b6 b3 b7~b4	With “b0” (n8) select up to 512 samples (0~511) With “b2-b1-b0” (f9~f7) defines the frequency If “1” turn pn the pseudo-reverberation; if “0”, turn off Octave. Range from –7 to +7 (–8 can´t used). With F_number defines the frequency. For octave = 1 and F_number = 0, the frequency is 44,1 Khz. $f(¢) = 1200 * (\text{octave} - 1) + 1200 * \log_2 \frac{1024+\text{F\_number}}{1024}$			
$50H ⋮ $67H	Total level (l6~l0)			DL	Total level 7 bits (l6~l0) Direct level
	b7-b1 b0	Total level (b7=-24dB, b6=-12dB, ... b1=-0,375dB) Direct level (0=change envelope during interpolation; 1=change envelope immediately)			
$68H ⋮ $7FH	Key on	Damp	LFO RST	CH	Panpot miscellaneous functions and stereo balancing (Panpot)
	b7 b6 b5 b4 b3~b0	0=key on; 1=key off 0=Damp off; 1=Damp on LFO RST (0=turn on the LFO; 1=turn off the LFO) 0=Wave mixed with FM; 1=No mixing			

```
Panpot:     0   1   2  …  6   7   8   9  …  13  14  15
Left (dB)   0  -3  -6  … -18 -∞  -∞   0  …   0   0   0
Right(dB)   0   0   0  …  0   0  -∞  -18 …  -9  -6  -3
```

$80H ⋮ $97H	•	•	LFO (s2~s0)	VIB (v2~v0)	Tremolo and vibrato frequency (LFO) Vibrato level (VIB)
	b7~b6 b5~b3 b2~b0	Not used LFO (0=0,168Hz, 1=2,019Hz, ... 7=7,066Hz) Vibrato level (0=off, 1=3,378; 2=5,065, ... 7=79,31)			

$98H ⋮ $AFH	Attack Rate	Decay Rate (1)	Attack Rate 10-90% ⟶ (1=3715 mS; 14=0,23 mS) Decay 1 Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)
$B0H ⋮ $C7H	Decay Level	Decay Rate (2)	Decay Level (b7=-24; b6=-12; b5=-6; b4=-3 dB) Decay 2 Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)
$C8H ⋮ $DFH	Rate Correction	Release Rate	Rate Correction Release Rate

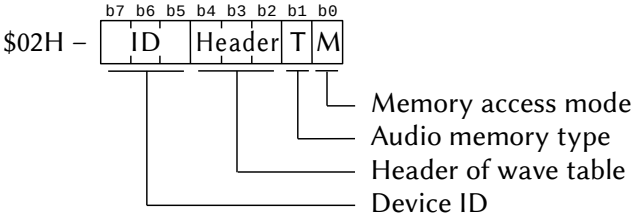
	b7~b4	Rate Correction: (RATE = (OCT + RC)*2 + f9 + RD) OCT = Octave (–7 a +7 in $38H~$4FH) RC = Rate correction (0~14 in $C8H~$DFH) f9 = bit "f9" of F_number ($38H~$4FH) RD = AR, D1R, D2R e RR values (0001=04; 0010=08; ...; 1111=63)
	b3~b0	Release Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)

$E0H ⋮ $F7H	•	•	•	•	•	AM(a2~a0)	Tremolo level
	b7~b3 b2~b0	Not used Tremolo level (0=off; 1=1,781; ...; 7=11,91)					
$F8H	•	•	Mix FM_R			Mix FM_L	Nível de saída FM
	b7~b6 b5~b3 b2~b0	Not used (always"00") Right FM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞) Left FM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞)					
$F9H	•	•	Mix PCM_R			Mix PCM_L	PCM output level
	b7~b6 b5~b3 b2~b0	Not used (always"00") Right PCM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞) Left PCM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞)					

The registers for wave synthesis (Wave Table Synthesis) are described shortly afterwards. At the end of this section, the wave synthesis model will be described. For FM synthesis, the registers differ slightly. They will be described later.

Registers $00H and $01H are used for testing the YMF278B only. They must always be set to 00H.

6.7.1.1 – Access to audio memory



Memory access mode

When this bit is 0, there is normal generation of sounds. When it is 1, the CPU can read or write data to audio memory and no sounds will be generated.

Audio memory type

When this bit is 0, only ROM can be connected. When it is 1, SRAM plus ROM can be connected.

Header of the wave table

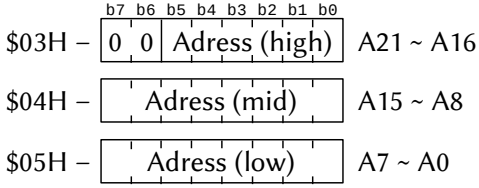
The header must be set from audio memory address 000 000H for wave numbers 0 to 511, and in 4 Mbit increments for wave numbers 384 to 511, as illustrated in the table below.

Header b4 b3 b2	Audio memory area
0 0 0	Waves numbers 0 to 511 in 000 000H
0 0 1	Waves numbers 84 to 511 in 080 000H
0 1 0	Waves numbers 384 to 511 in 100 000H
0 1 1	Waves numbers 384 to 511 in 180 000H
1 0 0	Waves numbers 384 to 511 in 200 000H
1 0 1	Waves numbers 384 to 511 in 280 000H
1 1 0	Waves numbers 384 to 511 in 300 000H
1 1 1	Waves numbers 384 to 511 in 380 000H

Device ID

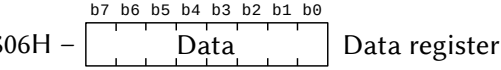
OPL4 ID register. Always returns the value 001B, even if it is written with other values.

Audio memory adresses



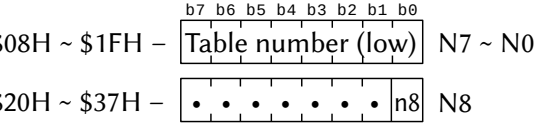
These registers specify the audio memory address to be written or read. The address is set by writing the value to $05H; therefore, it is always necessary to fill in the registers starting at $03H. These registers are incremented with each access to audio memory.

Memory data registers



This register is used for data transfer between CPU and audio memory. But it is a slow recorder. There must be a pause of 28 clock cycles before the next data is written and 38 clock cycles before the next data is read.

6.7.1.2 – Acess to wave mode



The OPL4 supports up to 512 defined wave tables and can play up to 24 of them simultaneously. Registers work in pairs; so $08H pairs with $20H and so on. You should always fill the lowest number register first (N7~N0 before N8). As the header is stored in the audio memory, while loading it, LFO, VIB, AR, D1R, DL, D2R, Rate Correction, RR or AM cannot be accessed or a problem may occur. The registers of other voices can be accessed normally. Header loading takes about 300µs after writing N8. Bit b1 of the status register indicates when a header is being loaded.

Frequency and octave

	b7	b6	b5	b4	b3	b2	b1	b0	
$20H~$37H –	f6	f5	f4	f3	f2	f1	f0	•	F_number (f6 ~ f0)
$38H~$4FH –	•	•	•	•	•	f9	f8	f7	F_number (f9 ~ f7)
$38H~$4FH –	o3	o2	o1	o0	•	•	•	•	Octave (o3 ~ o0)

These registers are used to control the pitch of the voice. They also work in pairs ($20H and $38H, $21H and $39H, etc). F_number is a positive number (0 to 1023) and the octave is 2's complement (–7 to +7). The value –8 must not be used. When F_number is 0 and octave is 1, the wave data is played back at a sampling rate of 44.1 KHz. This is the normal pitch (F(¢)=0, where ¢=1%).

The offset from the normal pitch can be calculated by the following expression:

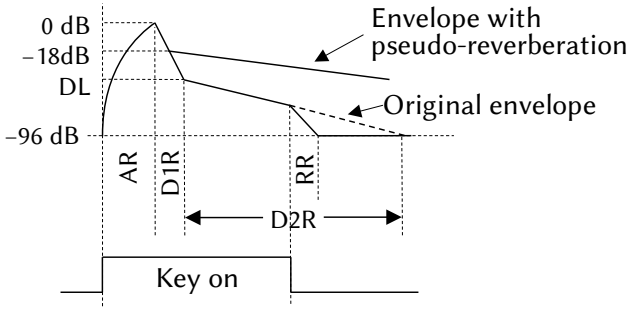
$$F(¢) = 1200 * (\text{octave} - 1) + 1200 * \log_2 \frac{1024 + \text{F-number}}{1024}$$

(1 octave = 1200¢)

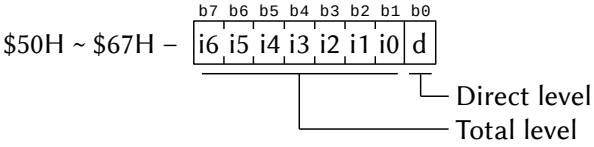
Pseudo reverberation

	b7	b6	b5	b4	b3	b2	b1	b0	
$38H~$4FH –	•	•	•	•	r	•	•	•	Pseudo-reverberation

When this bit is 0, pseudo-reverb is off; when it is 1, it will be on.



Total level / direct level

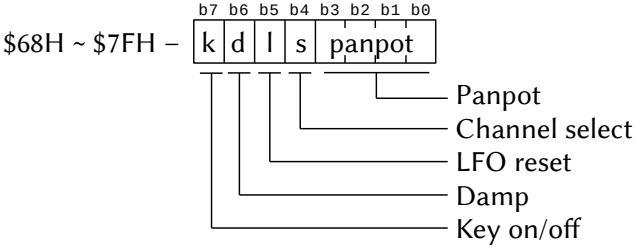


The total level defines the attenuation level of the sound played. The attenuation is the sum of the values described below when the respective bit in the register is 1.

i6 = –24 dB	i3 = –3 dB	i0 = –0,375 dB
i5 = –12 dB	i2 = –1,5 dB	
i4 = –6 dB	i1 = –0,75 dB	

The direct level selects the way the total level modifies the envelope. If 0, the total level changes the envelope during interpolation; if it is 1, the total level changes the envelope immediately. When the level is changed during interpolation, the rise time from minimum to maximum volume will be 78.2 ms and fall time from maximum to minimum will be 156.4 ms.

Key on, Damp, LFO reset, channel selection, panpot



The panpot function controls the stereo balance of each of the wave voices. The sound level of the right and left channels are set according to the table below.

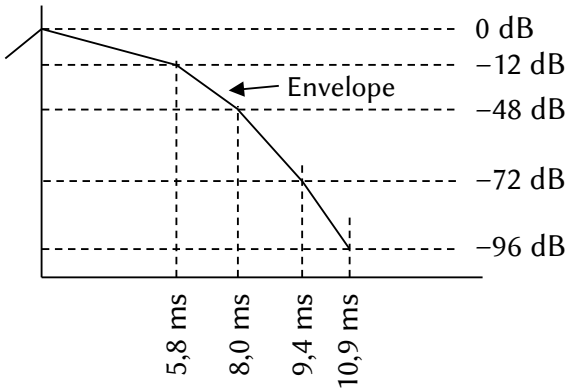
Panpot	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Left (dB)	0	–3	–6	–9	–12	–15	–18	–∞	–∞	0	0	0	0	0	0	0
Right (dB)	0	0	0	0	0	0	0	0	–∞	–∞	–18	–15	–12	–9	–6	–3

The output channel can be selected according to bit b4 (channel select) of these registers. If it is 0, the output will be mixed with the FM generator on the DO2 pin of the chip. If it is 1, there will be no mixing with the FM generator and the output will be routed to the DO1 pin of the chip.

The "LFO Reset" flag enables or disables the LFO (Low Frequency Oscillattor) which is used for vibrato and tremolo effects. If it is 0, the LFO is active; if it is 1, it will be off.

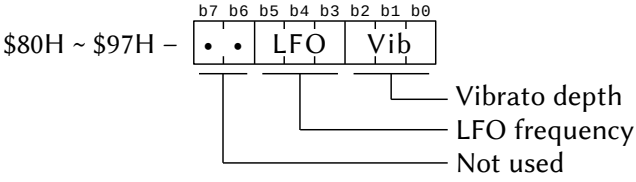
The "Damp" flag makes the "decay" and "release" times shorter when activated. If it is 0, the "damp" effect is off; if it is 1 it will be on. Pseudo-reverb is turned off during damping. The effect will be applied as illustrated below.

Time (ms)	5,8	8,0	9,4	10,9
Atennuation (dB)	–12	–48	–72	–96



The "key on" flag controls sound reproduction. If it is 0, "key off" will be selected, if it is 1, "key on" will be selected.

LFO, vibrato



Bits b2~b0 determine the vibrato depth, as shown in the table below.

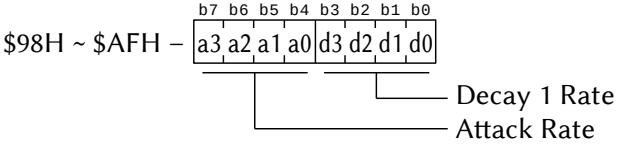
Bits b2~b0:	0	1	2	3	4	5	6	7
Depth (¢):	off	3,378	5,065	6,760	10,11	20,17	40,11	79,31

Bits b5~b3 determine the tremolo and vibrato frequency, as shown in the table below:

Bits b5~b3:	0	1	2	3	4	5	6	7
Frequency (Hz):	0,168	2,019	3,196	4,206	5,215	5,888	6,224	7,066

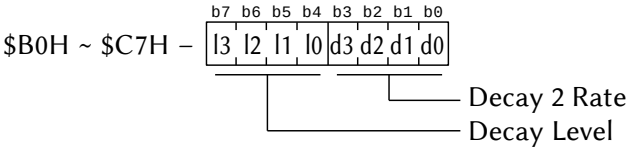
The b7~b6 bits are not used.

Attack rate, decay 1 rate



This register defines the “Decay 1 Rate” and the “Attack Rate”. More details can be seen in the “CALCULATING THE ‘RATES’” section.

Decay level, decay 2 rate

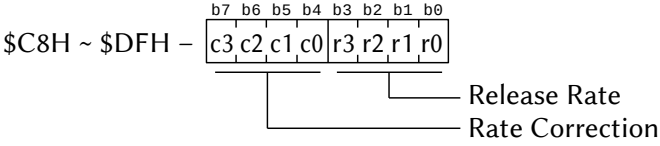


Bits b0~b3 of this register define the “Decay 2 Rate”. More details can be seen in the “CALCULATING THE ‘RATES’” section.

The bits b7~b4 define the "Decay Level". The decay level can be calculated according to the table below, adding the values when the respective bit is 1. However, when all bits are one, the decay level will be taken to –93 dB, and not at –45 dB which corresponds to the sum of the values.

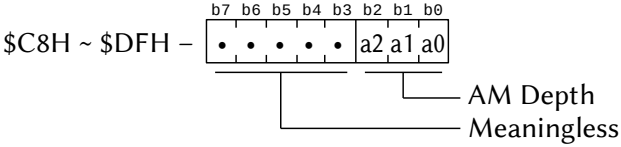
Bit	l3	l2	l1	l0
Level (dB)	–24	–12	–6	–3

Release rate, rate correction



Bits b3~b0 of this register define the "Release Rate" and bits b7~b4 define the "Rate Correction". More details can be seen in the "CALCULATING THE 'RATES'" section.

AM depth (tremolo)



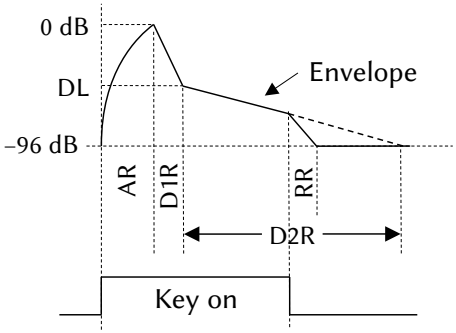
This recorder determines the tremolo grade according to the table below:

Register:	0	1	2	3	4	5	6	7
Depth (dB):	off	1,781	2,906	3,656	4,406	5,906	7,406	11,91

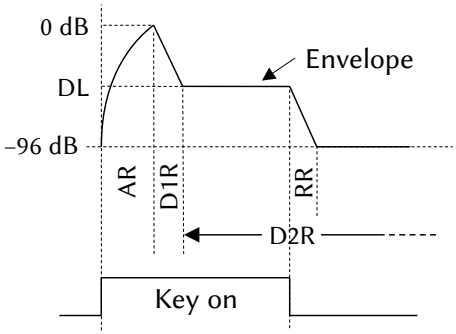
Envelope waveform

The envelope can have two distinct forms, called percussive tone and non-percussive tone. They are illustrated below.

Percussive tone (D2R > 0)



Non-percussive tone (D2R = 0)



Calculating the rates

The current rate can be calculated by the following formula:

RATE = (OCT + Rate Correction) * 2 + f9 + RD

Where: OCT: octave (–7 to +7) specified in $38H~$4FH
f9: Bit f9 of register F_number ($38H~$4FH)
Rate Correction: Value of $C8H~$DFH (0 to 14)

The RD value is determined by the values specified in AR, D1R, D2R, and RR. The relationship between these registers and RD is illustrated in the table below.

AR,D1R,D2R,RR	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
RD value	04	08	12	16	20	24	28	32	36	40	44	48	52	56	63

Whenever the value is greater than 63 in the equation, the value considered will always be 63. When AR=D1R=D2R=RR=0, RATE will be 0; When AR=D1R=D2R=RR=15, RATE will be 63.

The “attack”, “decay” and “release” ratios are defined in 4 bits. The higher the value, the shorter the “attack”, “decay” and “release” time. Their values are described in the table below.

Attack Rate (time in milliseconds – ms)

RATE	Time (0–100%)	Time (10–90%)
0	∞	∞
1	∞	∞
2	∞	∞
3	∞	∞
4	6222.95	3715.19
5	4978.37	2972.20
6	4148.66	2476.83
7	3556.01	2122.99
8	3111.47	1857.60
9	2489.21	1486.12
10	2074.33	1238.41
11	1778.00	1061.50
12	1555.74	928.80
13	1244.63	743.08
14	1037.19	619.23
15	889.02	530.75
16	777.87	464.40
17	622.31	371.56
18	518.59	309.61
19	444.54	265.40
20	388.93	232.20
21	311.16	185.80
22	259.32	154.83
23	222.27	132.70
24	194.47	116.10
25	155.60	92.93

RATE	Time (0–100%)	Time (10–90%)
32	48.62	29.02
33	38.91	23.27
34	32.43	19.37
35	27.80	16.60
36	24.31	14.51
37	19.46	11.66
38	16.24	9.70
39	13.92	8.30
40	12.15	7.26
41	9.75	5.85
42	8.12	4.85
43	6.98	4.17
44	6.08	3.63
45	4.90	2.95
46	4.08	2.45
47	3.49	2.09
48	3.04	1.81
49	2.49	1.45
50	2.13	1.22
51	1.90	1.09
52	1.72	0.95
53	1.41	0.77
54	1.18	0.63
55	1.04	0.54
56	0.91	0.50
57	0.73	0.36

26	129.66	77.41
27	111.16	66.35
28	97.23	58.05
29	77.82	46.49
30	64.85	38.73
31	55.60	33.20

58	0.59	0.27
59	0.50	0.27
60	0.45	0.23
61	0.45	0.23
62	0.45	0.23
63	0.00	0.00

Decay e Release Rate (time in milliseconds – ms)

RATE	Tempo (0–100%)	Tempo (10–90%)
0	∞	∞
1	∞	∞
2	∞	∞
3	∞	∞
4	89 164.63	19 040.36
5	71 331.75	15 278.73
6	59 443.13	12 724.54
7	50 951.25	10 890.16
8	44 582.31	9520.18
9	35 665.90	7639.37
10	29 721.59	6362.27
11	25 475.65	5445.08
12	22 291.16	4760.09
13	17 832.97	3819.68
14	14 860.82	3181.13
15	12 737.82	2722.54
16	11 145.58	2380.05
17	8916.51	1909.84
18	7430.43	1590.57
19	6368.93	1361.27
20	5572.79	1190.02
21	4458.28	954.92
22	3715.24	795.28
23	3184.49	680.63
24	2786.39	595.01
25	2229.16	477.46

RATE	Tempo (0–100%)	Tempo (10–90%)
32	696.60	148.75
33	557.32	119.37
34	464.44	99.41
35	398.10	85.08
36	348.30	74.38
37	278.68	59.68
38	232.24	49.71
39	199.05	42.54
40	174.15	37.19
41	139.37	29.84
42	116.15	24.85
43	99.55	21.32
44	87.07	18.59
45	69.71	14.92
46	58.10	12.43
47	49.80	10.66
48	43.54	9.23
49	34.83	7.44
50	29.02	6.08
51	24.90	5.31
52	21.77	4.67
53	17.41	3.72
54	14.51	3.13
55	12.43	2.68
56	10.08	2.36
57	8.71	1.95

26	1857.64	397.64
27	1592.24	340.32
28	1393.20	297.51
29	1114.60	238.73
30	928.84	198.82
31	796.15	170.16

58	7.23	1.59
59	6.21	1.36
60	5.44	1.18
61	5.44	1.18
62	5.44	1.18
63	5.44	1.18

6.7.1.3 – Wave table synthesis format

"Wave Table Synthesis" is a form of PCM playback that uses samples, usually short, that play the initial part and continuously repeat the following part. For this purpose, it has a header that is recorded in the external (audio) memory. The header structure is described below.

Reg	7	6	5	4	3	2	1	0	
00H	d1	d0	s21	s20	s19	s18	s17	s16	Data bit (d1 e d0) Start adress (s21~s0)
01H	s15	s14	s13	s12	s11	s10	s9	s8	
02H	s7	s6	s5	s4	s3	s2	s1	s0	
03H	l15	l14	l13	l12	l11	l10	l9	l8	Loop adress (l15~l0)
04H	l7	l6	l5	l4	l3	l2	l1	l0	
05H	e15	e14	e13	e12	e11	e10	e9	e8	End adress (e15~e0)
06H	e7	e6	e5	e4	e3	e2	e1	e0	
07H	•	•	f2	f1	f0	v2	v1	v0	LFO frequency / vibrato depth
08H	ar3	ar2	ar1	ar0	dr3	dr2	dr1	dr0	Attack Rate; Decay 1 Rate
09H	dl3	dl2	dl1	dl0	dr3	dr2	dr1	dr0	Decay Level; Decay 2 Rate
0AH	rc3	rc2	rc1	rc0	rr3	rr2	rr1	rr0	Rate Correction; Release Rate
0BH	•	•	•	•	•	am2	am1	am0	AM depth (tremolo)

All headers are in sequence in the audio memory, before the data area (samples), from 0 to 383 or from 384 to 511, even if not used.

Bits resolution

Bits d1 and d0 specify the bit resolution of the data to be reproduced, as shown in the table below.

d1	d0	Resolution	d1	d0	Resolution
0	0	8 bits	1	0	16 bits
0	1	12 bits	1	1	Prohibited

The respective formats in the data area are as follows:

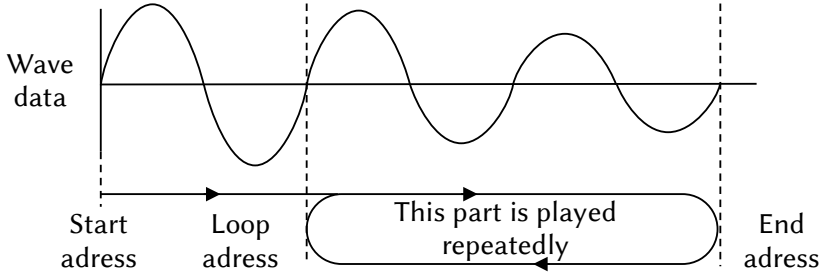
16 bits	d15	d14	d13	d12	d11	d10	d9	d8	+00H
	d7	d6	d5	d4	d3	d2	d1	d0	+01H
12 bits	d11	d10	d9	d8	d7	d6	d5	d4	+00H
	d3	d2	d1	d0	d3	d2	d1	d0	+01H
	d11	d10	d9	d8	d7	d6	d5	d4	+02H
8 bits	d7	d6	d5	d4	d3	d2	d1	d0	+00H

Start adress

The starting address of wave data is specified absolutely. For 12-bit resolution, the starting address must always be specified starting from bit 8 of the highest order byte.

Loop adress

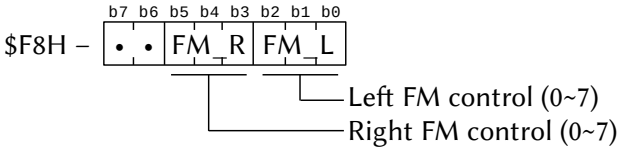
The loop address specifies the address from which the data will be repeated. It is relative to the starting address.

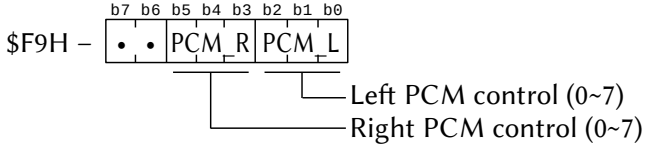


End adress

The playback end address is specified relative to the start address. As only 16 bits are used for the final address, each sample can be a maximum of 64 Kbytes.

6.7.1.4 – Wave/FM mix control





These registers specify the mix level of the FM generator and PCM output on pin DO2. When reset, the OPL4 sets the mix level of the FM ($F8H) to –9 dB by default and the PCM ($F9H) to 0 dB, balancing the volume of the FM and PCM outputs. The levels are described in the table below.

Register:	0	1	2	3	4	5	6	7
Mix level (dB):	0	–3	–6	–9	–12	–15	–18	-∞

6.7.2 – Registers description for the FM synthesis

FM generator – Register Array 0 (A1 = "1")									
Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H $01H	Test								Test registers
$02H $03H	1st Timer (80,8 µS) 2nd Timer (323,1 µS)								Time registers
$04H	RST	MT1	MT2	•	•	•	ST2	ST1	Flag register
	b7 b6 b5 b4-b2 b1 b0		RST – If write 1, resets b5, b6 and b7. MT1 – If write 1, b0 will be 0. MT2 – If write 1, b1 will be 0. Not used (always"000") ST2 – $03 Start/stop control (1=start counter) ST1 – $032Start/stop control (1=start counter)						
$08H	•	NTS	•	•	•	•	•	•	Keyboard configuration
	b7 b6 b5~b0		Not used (always"0") NTS – If 0, the separation point is determined by the higher 2 bits of F_number. If 1, the separation point is determined only by the MSB of F_number. Not used (always"000 000")						

<table>
<tr><td rowspan="2">$20H
⋮
$35H</td><td>AM</td><td>VIB</td><td>EGT</td><td>KSR</td><td>Multiple</td><td>Instruments definition</td></tr>
<tr><td colspan="2">b7
b6
b5
b4
b3~b0</td><td colspan="4">AM (1=tremolo on (Frequency: 3,7Hz)
VIB (1=vibrato on (Frequency: 6,4Hz)
EG-TYP (0=decaying sound; 1=sustained sound)
If 0, KSR⟶0~3; If 1, KSR⟶0~15
Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)</td></tr>
<tr><td>$40H
⋮
$55H</td><td colspan="2">KSL</td><td colspan="3">Total level</td><td>KSL (00= 0dB/octave,
01=1,5dB, 10=3dB, 11=6dB)
Total level (b0=0,75dB,
b1=1,5dB ..... b5=24dB)</td></tr>
<tr><td>$60H
⋮
$75H</td><td colspan="3">Attack Rate
(AR)</td><td colspan="2">Decay Rate
(DR)</td><td>Attack (0dB a 96dB ⟶
min. 0,2 mS; max 2826 mS)
Decay (0dB a 96dB ⟶
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$80H
⋮
$95H</td><td colspan="3">Sustain Level
(SL)</td><td colspan="2">Release Rate
(RL)</td><td>Sustain (b7=24dB, b6=12dB,
b5=6dB, b4=3dB)
Release (0dB a 96dB ⟶
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$A0H
⋮
$A8H</td><td colspan="5">Frequency (LSB 8 bits)</td><td>Frequency (b7~b0)</td></tr>
<tr><td rowspan="2">$B0H
⋮
$B8H</td><td>•</td><td>•</td><td>KEY</td><td colspan="2">Octave</td><td>Freq.
MSB
2 bits</td><td>Freq. MSB 2 bits (b9~b8)
Octave (FM)
Key on/off (FM)</td></tr>
<tr><td colspan="2">b7~b6
b5
b4~b2
b1~b0</td><td colspan="5">Not used (always“00”)
0=key off; 1=key on (voice active)
Define the octave. The fourth is 011.
Frequency MSB 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 110</td></tr>
<tr><td></td><td colspan="7">Operators (for $20H~$35H and $A0H~$A8H)

Oper: 01 02 03 04 05 06 07 08 09
Voz: 1 2 3 1 2 3 4 5 6
Reg: $20 $21 $22 $23 $24 $25 $28 $29 $2A
Freq: $A0 $A1 $A2 $A0 $A1 $A2 $A3 $A4 $A5

The operators are associated as below:</td></tr>
</table>

	Oper:	10	11	12	13	14	15	16	17	18
	Voz:	4	5	6	7	8	9	7	8	9
	Reg:	$2B	$2C	$2D	$30	$31	$32	$33	$34	$35
	Freq:	$A3	$A4	$A5	$A6	$A7	$A8	$A6	$A7	$A8

$20/$40/$60/$80/$A0/$B0/$C0
or
$23/$43/$63/$83/$A0/$B0/$C0

$BDH	AM	VIB	BAT	BD	SD	TOM	TCY	HH	Controle da bateria do FM

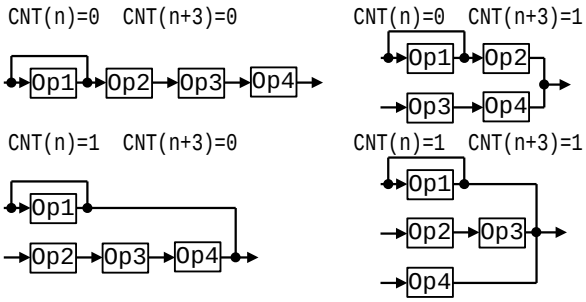
b7	Tremolo level (0=1dB. 1=4,8dB)
b6	Vibrato level (0=7%; 1=14%)
b5	0=Melody mode; 1=Rhythm mode
b4	1=Bass Drum
b3	1=Snare Drum
b2	1=Tom-tom
b1	1=Top Cymbal
b0	1=High-Hat

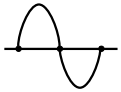
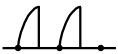
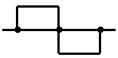
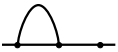
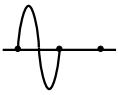
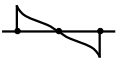
$C0H ⋮ $C8H	•	•	•	•	Feedback	CON	Feedback factor; connect type

b7~b4	Not used (always“0000”)
b3~b1	Feedback factor (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)
b0	Connection type (0=serie; 1=paralell)

For 4 operators:

A1	Channel	CNT(n)	CNT(n+3)
0	1	C0H	C3H
0	2	C1H	C4H
0	3	C2H	C5H
1	4	C0H	C3H
1	5	C1H	C4H
1	6	C2H	C5H



$E0H ⋮ $F5H	•	•	•	•	•	Wave Select		Waveform select
	b7~b3 b2~b1	Not used (always“00 000”) Waveform select: 000 –  011 –  110 –  001 –  100 –  111 –  010 –  101 – 						

The “Register Array 0” is OPL3 compatible; “Register Array 1” has been expanded to OPL4 mode. These modes are selected by the “NEW” and “NEW2” bits. The differences between “Register Array 0” and “Register Array 1” are from a few registers and are illustrated below.

FM generator – Register Array 1 (A1 = “H”)									
Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H $01H	Test								Test registers
$04H	•	•	Connection SEL						4-operators selection

	b7~b6
	b5~b0
		Not used (always“00”)
		Enable 4-operators mode for the respective slot:
		`Bit: b5 b4 b3 b2 b1 b0`
		`Slot: 6 5 4 3 2 1`							

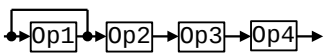
$05h	•	•	•	•	•	•	NEW2	NEW	Expansion register
	b7~b2 b1 b0	Not used (always“000 000”) If 1, enable OPL4 mode (Register Array 1) If 1, enable OPL3 mode (Register Array 0)							
$20H ⋮ $35H	AM	VIB	EGT	KSR	Multiple				
	b7 b6 b5 b4 b3~b0	AM (1=trêmolo on (frequency: 3,7Hz) VIB (1=vibrato on (frequency: 6,4Hz) EG-TYP (0=decaying sound; 1=sustained sound) If 0, KSR⟶0~3; If 1, KSR⟶0~15 Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)								

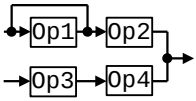
<table>
<tr><td>$40H
⋮
$55H</td><td>KSL</td><td colspan="5">Nível total</td><td>KSL (00= 0dB/octave,
01=1,5dB, 10=3dB, 11=6dB)
Total level (b0=0,75dB,
b1=1,5dB ..... b5=24dB)</td></tr>
<tr><td>$60H
⋮
$75H</td><td colspan="3">Attack Rate
(AR)</td><td colspan="2">Decay Rate
(DR)</td><td>Attack (0dB a 96dB →
min. 0,2 mS; max 2826 mS)
Decay (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$80H
⋮
$95H</td><td colspan="3">Sustain Level
(SL)</td><td colspan="2">Release Rate
(RL)</td><td>Sustain (b7=24dB, b6=12dB,
b5=6dB, b4=3dB)
Release (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$A0H
⋮
$A8H</td><td colspan="6">Frequency (LSB 8 bits)</td><td>Frequency (b7-b8)</td></tr>
<tr><td rowspan="2">$B0H
⋮
$B8H</td><td>•</td><td>•</td><td>KEY</td><td>Octave</td><td>Freq.
MSB
2 bits</td><td>Freq. MSB 2 bits (FM)
Octave (FM)
Key on/off (FM)</td></tr>
<tr><td colspan="2">b7~b6
b5
b4~b2
b1~b0</td><td colspan="4">Not used (always“00”)
0=key off; 1=key on (active voice)
Defines the octave. The fourth is 011.
Frequency MSB 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 110</td></tr>
<tr><td></td><td colspan="7">Operators (para$20H~$35H e$A0H~$A8H)

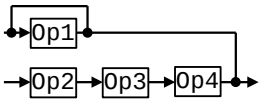
Oper: 19 20 21 22 23 24 25 26 27
Voz: 1 2 3 1 2 3 4 5 6
Reg: $20$21$22$23$24$25$28$29$2A
Freq:$A0$A1$A2$A0$A1$A2$A3$A4$A5

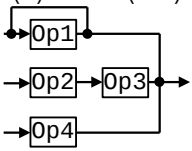
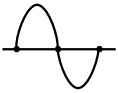
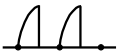
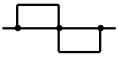
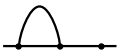
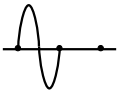
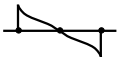
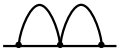
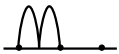
Oper: 28 29 30 31 32 33 34 35 36
Voz: 4 5 6 7 8 9 7 8 9
Reg: $2B$2C$2D$30$31$32$33$34$35
Freq: $A3$A4$A5$A6$A7$A8$A6$A7$A8

The operators are associated as below:
$20/$40/$60/$80/$A0/$B0/$C0
or
$23/$43/$63/$83/$A0/$B0/$C0</td></tr>
<tr><td rowspan="2">$C0H
⋮
$C8H</td><td>•</td><td>•</td><td>•</td><td>•</td><td>Feedback</td><td>CON</td><td>Feedback factor/connect type</td></tr>
<tr><td colspan="2">b7~b4
b3~b1
b0</td><td colspan="5">Not used (always“0000”)
Feedback factor (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)
Connection type (0=serie; 1=paralell)</td></tr>
</table>

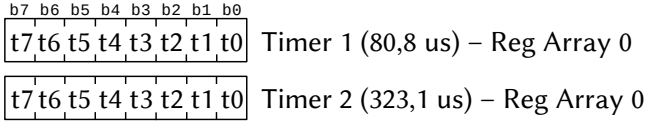
<table>
<tr><td></td><td></td><td>For 4 operators:
<table>
<tr><th>A1</th><th>Canal</th><th>CNT(n)</th><th>CNT(n+3)</th></tr>
<tr><td>0</td><td>1</td><td>C0H</td><td>C3H</td></tr>
<tr><td>0</td><td>2</td><td>C1H</td><td>C4H</td></tr>
<tr><td>0</td><td>3</td><td>C2H</td><td>C5H</td></tr>
<tr><td>1</td><td>4</td><td>C0H</td><td>C3H</td></tr>
<tr><td>1</td><td>5</td><td>C1H</td><td>C4H</td></tr>
<tr><td>1</td><td>6</td><td>C2H</td><td>C5H</td></tr>
</table>
CNT(n)=0 CNT(n+3)=0
 

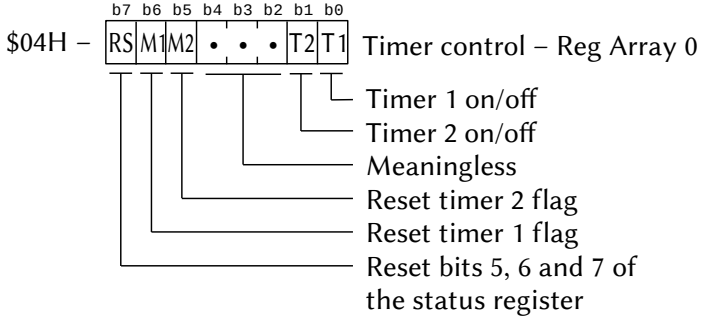
CNT(n)=0 CNT(n+3)=1
 

CNT(n)=1 CNT(n+3)=0
 

CNT(n)=1 CNT(n+3)=1
 
</td></tr>
<tr><td>$E0H
⋮
$F5H</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>Wave Select</td><td>Waveform select</td></tr>
<tr><td></td><td>b7~b3
b2~b1</td><td colspan="6">Not used (always“00 000”)
Waveform select
000 –  011 –  110 – 
001 –  100 –  111 – 
010 –  101 –  </td></tr>
</table>

6.7.2.1 – Timers





There are two timers in OPL4. The resolution of timer 1 is 80.8 µs and that of timer 2 is 323.1 µs. The formulas that allow calculating the time of each one, in milliseconds, are the following:

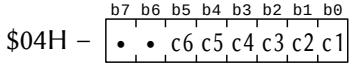
t1(ms) = (256 – n1) * 0.0808 (timer 1)
t2(ms) = (256 – n2) * 0.3231 (timer 2)

Where n1 and n2 represent the value of each counter (0 ~ 255). When each counter has timed out, an interrupt signal is sent to the CPU.

Bits T1 and T2 of $04H enable or disable timers 1 or 2, respectively. When the bit is 0, the timer is off; when it is 1, it will be active. When the M1 or M2 bits are set to 1, the respective timer flag will always be 0, regardless of the operation of the timers. In this case, no interruption will be generated. When the RS bit is set to 1, bits d5, d6 and d7 of the status register are cleared. Afterwards, RS will automatically return to 0.

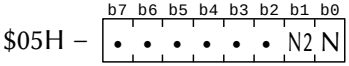
6.7.2.2 – Access to FM mode

4 operators mode



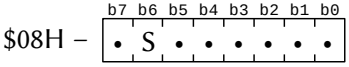
When any of the bits c1~c6 is set to 1, the corresponding channel can be used in 4-operator mode. Further details are described in the “CHANNELS AND SLOTS” section.

Expansion register



These registers allow expansion of OPL2 and OPL3 modes to OPL4. If both bits are 0, OPL2 mode is active. If bit N is 1, OPL3 mode (Register array 0) is active. If bit N2 is 1, OPL4 (Register Array 1) mode is active. As these two bits are reset on reset, they must be set to 1 to activate OPL4 mode before using array 1 or PCM.

Keyboard split selection



Up to 8 octaves can be selected from a total of 16 for all FM Voices. Bit b6 of $08H (NTS) determines which octaves will be active, as shown in the table below:

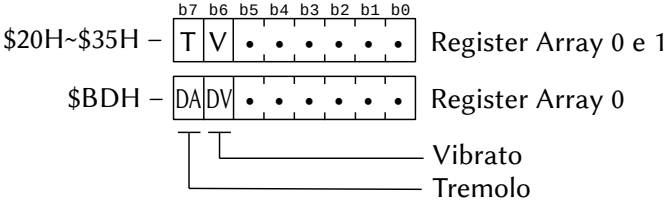
NTS = 0

Octave	0		1		2		3		4		5		6		7	
F_num msb	•		•		•		•		•		•		•		•	
F_num 2º	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
Key Scale No	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

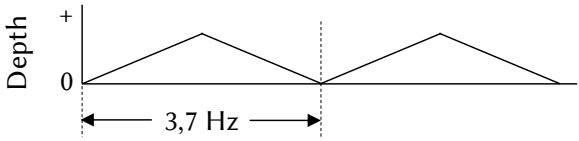
NTS = 1

Octave	0		1		2		3		4		5		6		7	
F_num msb	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
F_num 2º	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Key Scale No	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

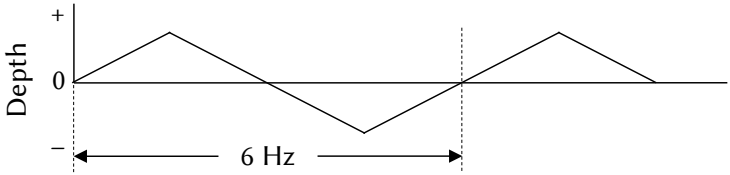
Tremolo and vibrato



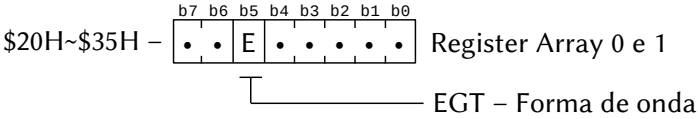
When bit b7 of $20H~$35H is 1, tremolo for the respective voice will be on. The tremolo frequency is 3.7 Hz and the depth is determined by the DA bit (DA=0.1db; DA=1, 4.8dB), as illustrated below.



Bit b6 of $20H~$35H turns vibrato on or off for the respective voice; if it is 0, it is off; if it is 1 it will be on. The vibrato frequency is 6 Hz and its depth is determined by the DV bit (DV=0.7%, DV=1.14%), as illustrated below.

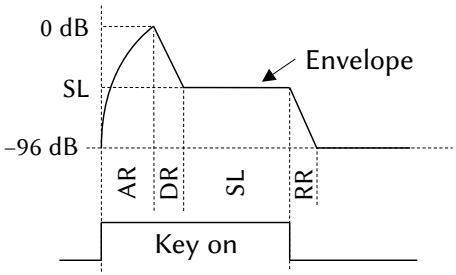


Envelope waveform

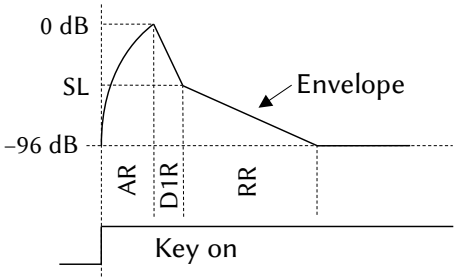


This bit determines the envelope waveform as below.

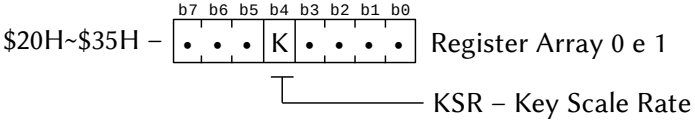
Non-percussive tone (EGT = 0)



Percussive tone (EGT = 1)



KSR (Key scale rate)



This bit is used to regulate the amount of silence in the pitch shift range, simulating real musical instruments. The values follow the table below.

Key Scale Value		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Rof	KSR=0	0	0	0	0	1	1	1	1	2	2	2	2	3	3	3	3
	KSR=1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

For the "key scale" numbers, see item "$08H – Keyboard split selection".

The RATE can be calculated by the following expression:

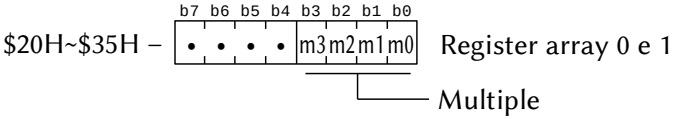
RATE = (Rate value) * 4 + Rof

When "rate value" is 0, RATE will be 0. When RATE exceeds 63, it will always be set to 63. The "attack", "decay" and "release" ratios are set to 4 bits. The higher the value, the shorter the "attack", "decay" and "release" time. In the table below, their extreme values in milliseconds are described. The time variation obeys, approximately, a geometric progression between these values. Consult the table in the item "Calculating the rates" in the section "7.1.2 – Accessing the wave mode" for exact values.

	Minimum	Maximum
Attack (0dB a 96dB)	0,20 ms	2826 ms
Attack (10% a 90%)	0,11 ms	1482 ms
Decay (0dB a 96dB)	2,40 ms	39 280 ms
Decay (10% a 90%)	0,51 ms	8212 ms
Release (0dB a 96dB)	2,40 ms	39 280 ms
Release (10% a 90%)	0,51 ms	8212 ms

Note: For RATE=60~63, the time will be null and for RATE=0~3, the time will be infinite.

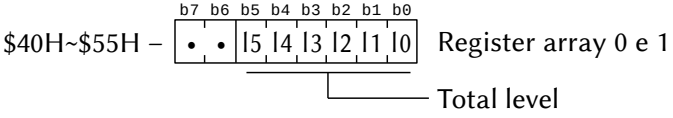
Multiple



This register specifies the multiplier for the frequencies specified by BLOCK and F_number. The multiplication factors are illustrated below.

Register value:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Multiplication factor:	½	1	2	3	4	5	6	7	8	9	10	10	12	12	15	15

Total level



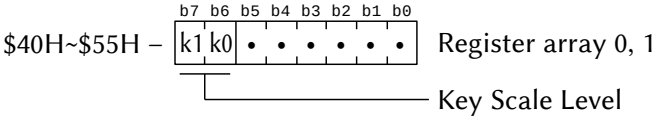
The total level defines the damping for the envelope. It controls the volume and modulation rate. Its value corresponds to the sum of the values listed in the table below when the respective bit is 1.

l5 = –24 dB	l2 = –3 dB
l4 = –12 dB	l1 = –1,5 dB
l3 = –6 dB	l0 = –0,75 dB

The actual level of damping is determined by dividing an octave into 16 parts by the value of the highest four bits of the F-number. The damping ratio shown below is for 3dB/oct. For 1.5dB/octave, damping is half and for 6.0dB/octave, damping is double the table.

Oct. / F_num	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	3.000	6.000	9.000
2	0	0	0	0	3.000	6.000	9.000	12.000
3	0	0	0	1.875	4.875	7.875	10.875	13.875
4	0	0	0	3.000	6.000	9.000	12.000	15.000
5	0	0	1.125	4.125	7.125	10.125	13.125	16.125
6	0	0	1.875	4.875	7.875	10.875	13.875	16.875
7	0	0	2.625	5.625	8.625	11.625	14.625	17.625
8	0	0	3.000	6.000	9.000	12.000	15.000	18.000
9	0	0.750	3.750	6.750	9.750	12.750	15.750	18.750
10	0	1.125	4.125	7.125	10.125	13.125	16.125	19.125
11	0	1.500	4.500	7.500	10.500	13.500	16.500	19.500
12	0	1.875	4.875	7.875	10.875	13.875	16.875	19.875
13	0	2.250	5.250	8.250	11.250	14.250	17.250	20.250
14	0	2.625	5.625	8.625	11.625	14.625	17.625	20.625
15	0	3.000	6.000	9.000	12.000	15.000	18.000	21.000

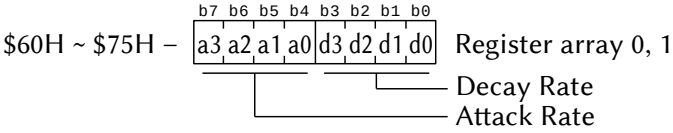
KSL (Key scale level)



This register specifies the progressive attenuation of the generated sound in order to approximate it to the sound of acoustic musical instruments. This attenuation occurs according to the table below.

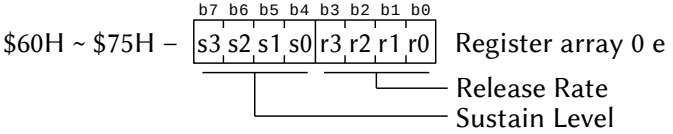
KSL	0	1	2	3
Attenuation	0 dB/octave	3 dB/octave	1,5 dB/octave	6 dB/octave

Attack rate (AR) / Decay rate (DR)



This recorder defines the "Attack Rate" and the "Decay Rate". More details can be seen in the item "$20H~$35H – KSR (KEY SCALE RATE)".

Release rate (RR) / sustain level (SL)

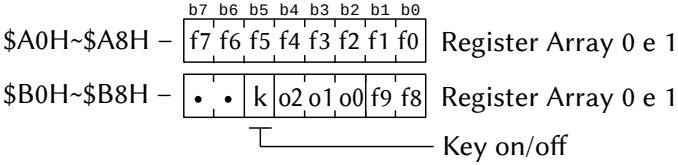


For the "release rate" see more details can be seen in the item "$20H~$35H – KSR (Key scale rate)".

The "Sustain Level" specifies the level at which the envelope remains after the "Decay Rate". For percussive tone, specifies the transition point from "Decay Rate" to "Release Rate". Its value corresponds to the sum of the values listed below, when the respective bit is 1. When all bits are 1, the value of SL is set to –93dB.

s3	s2	s1	s0
–24dB	–12dB	–6dB	–3 dB

Frequency adjustement / Key on/off



The "k" (Key on/off) bit controls the sound generation. See the item "ENVELOPE WAVEFORM" above.

Values f0~f9 are called F_number and specify the generated frequency and values o0~o2 specify the octave (also called "block"). F_number can range from 0 to 1023 and the octave from 0 to 7.

The relationship between frequency, F_number and octave is given by the following equation:

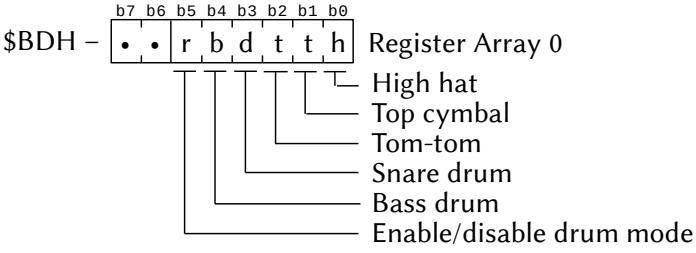
$$F\_number = \frac{(Frequency) \times 2^{19} \div (Sample\ frequency / 684)}{2^{(octave-1)}}$$

The values for the number 4 octave (in the 0 to 7 scale) are listed below.

	Frequency	Decimal	$BxH,b1-b0	$AxH
C	261,6 Hz	346	0 1	01 011 010
C#	277,2 Hz	367	0 1	01 101 111
D	293,7 Hz	389	0 1	10 000 101
D#	311,1 Hz	412	0 1	10 011 100
E	329,6 Hz	436	0 1	10 110 100
F	349,2 Hz	462	0 1	11 001 110
F#	370,0 Hz	490	0 1	11 101 010
G	392,0 Hz	519	1 0	00 000 111
G#	415,3 Hz	550	1 0	00 100 110
A	440,0 Hz	582	1 0	01 000 110
A#	466,2 Hz	617	1 0	01 101 001
B	493,9 Hz	654	1 0	10 001 110
C	523,3 Hz	693	1 0	10 110 101

The frequency values hold a geometric relationship equal to the 12th root of 2, which is 1.059 463 094 3592. You can use this number to change the register values to increase or decrease the generated frequency within the musical scale. The values of the registers also have the same relationship to each other.

Rhythm mode

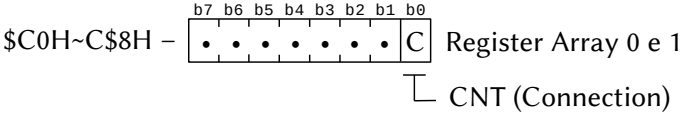


This register controls rhythm mode. When the "r" bit is 1, rhythm mode will be active and the last three FM generator voices will be unavailable. However, up to 5 rhythm pieces can be generated as described above. The slots used are as follows:

Instrument	Slot
Bass drum (BD)	13,16
Snare drum (SD)	17
Tom-tom (TOM)	15
Top cymbal (TC)	18
High hat (HH)	14

The "Rate" and other values can be set to manipulate the sound of the rhythm parts. When in rhythm mode, the "Key" bit must be set to 0 for slots 13 to 18.

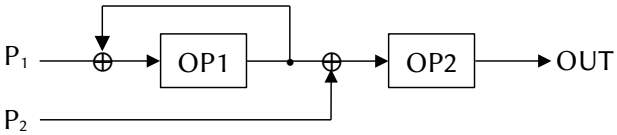
CNT (Connection)



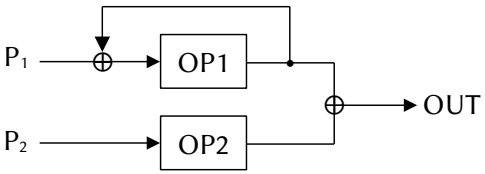
• 2-Operators mode

In 2 operators mode, when this bit is 0, algorithm 1 is selected. When it is 1, algorithm 2 is selected.

Algorithm 1 (CNT=0)

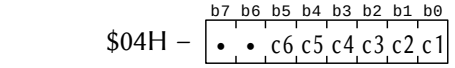


Algorithm 2 (CNT=1)



• 4-Operators mode

To select the 4-operator mode, it is necessary to set the respective bit in register $04H (Connection SEL) and then use the two available CNT bits to apply the algorithms.

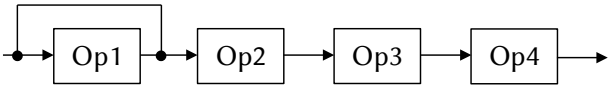


The value of the CNT bits required for algorithm selection are illustrated in the table below.

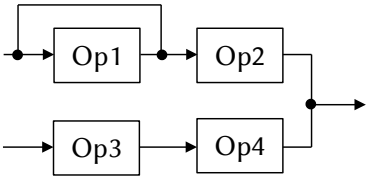
A1	Channel Number	Make bit CNT = 1	
		CNT_n	CNT_{n+3}
0	1	$C0H	$C3H
	2	$C1H	$C4H
	3	$C2H	$C5H
1	1	$C0H	$C3H
	2	$C1H	$C4H
	3	$C2H	$C5H

The 4 possible algorithms using the CNTn and CNTn+3 bits are illustrated below.

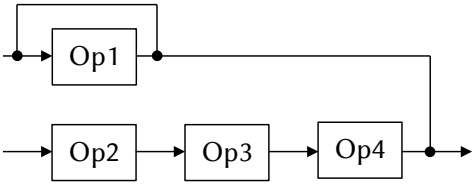
Algorithm 1 (CNT_n=0, CNT_{n+3}=0)



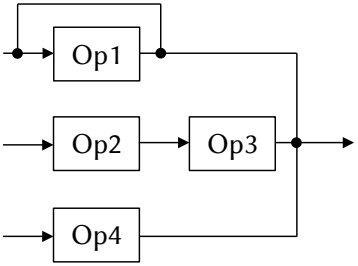
Algorithm 2 (CNT_n=0, CNT_{n+3}=1)



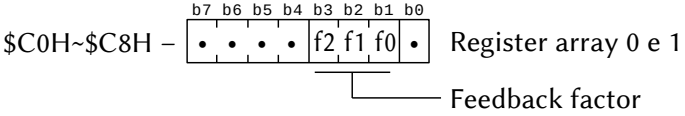
Algorithm 3 (CNT_n=1, CNT_{n+3}=0)



Algorithm 4 (CNT_n=1, CNT_{n+3}=1)



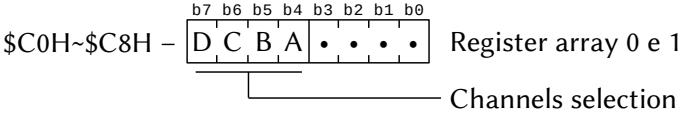
Feedback



This register defines the feedback factor (portion of the output signal that is re-injected into the input). Feedback values are described in the table below.

Register value:	0	1	2	3	4	5	6	7
Modulation rate:	0	π/16	π/8	π/4	π/2	π	2π	4π

Output channels selection

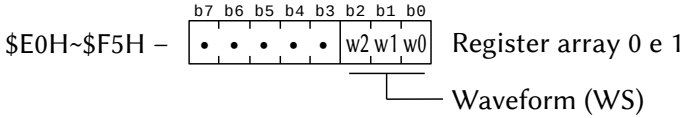


Up to 4 output channels are available for the FM generator. The output will be enabled when the respective bit is 1.

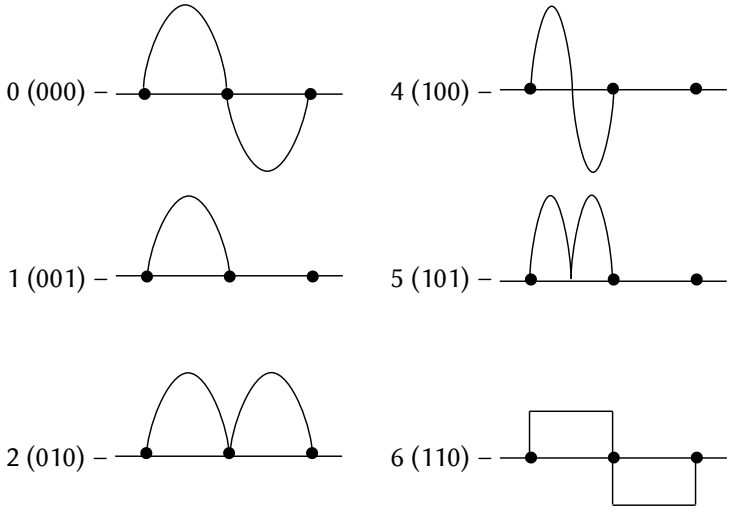
Channels A and B: The FM synthesizer output is digitally mixed with the wave table synthesizer of the channel in which the CH bit of registers $68H to $7FH of the wave table synthesizer is set to "0", and the signal will be sent to pin DO2. CHA is mixed with the left output of the wave table synthesizer and CHB is mixed with the right output.

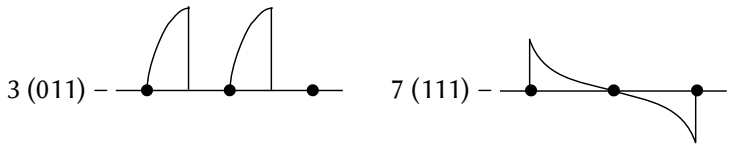
Channels C and D: The output defined by CHC and CHD is output from the DO0 pin. As FM outputs from DO0 pin and PCM outputs from DO1 pin, sound effects can be applied to a particular sound by connecting the YSS225 chip to DO0 and DO1 pins. Of course, a D/A converter can also be connected.

Waveform select



This register selects the waveform to be used in each slot. Possible waveforms are illustrated below. When in OPL2 mode, only WS0 (000) to WS3 (011) will be available.





6.7.3 – Access to OPL4

OPL4 is accessed directly through I/O ports, in the same way as OPLL and MSX-Audio. There are no standardized access routines. The ports used by the Moonsound cartridge are as follows:

0C4H – FM register array 0 (bank 1) and state register.
0C5H – FM (data).
0C6H – FM register array 1 (bank 2).
0C7H – Mirror of C5H (access via C5H is preferred).
07EH – PCM (wave) registers.
07FH – PCM data (wave).

The access method is very simple: just select the register through C4H, C6H or 7EH and then write the data through C5H or 7FH. No need to worry about pauses until the MSX turbo R; accesses can be made sequentially without any problems. Access to the FM registers requires a pause of 56 cycles and the PCM of 88 clock cycles of the OPL4, which is 33.8688 MHz. The times are as follows:

Time	OPL4 T cycles	Time (µs)	Z80 T cycles	R800 T cycles
FM	56	1,65 µs	6 cycles	12 cycles
PCM/MIX	88	2,60 µs	10 cycles	19 cycles

However, to enable access to the PCM (wave), it is necessary to set bits b0 and b1 in register 5 of bank 2 of the FM, as illustrated below:

```
OUT   0C6H,5
OUT   0C5H,00000011B
```

6.8 – COVOX

Covox is a sound generator that uses the printer port to reproduce PCM data at 8-bit resolution. More details on how it works can be seen in section "4 – The PCM".

Covox encoding is also in absolute binary (two's complement) as in PCM and SCC. By using an extremely simple circuit, however, there are no standardized sampling rates; the rate must be determined by software timing.

6.8.1 – Access to covox

To access Covox, simply send the data bytes sequentially through the printer's I/O port (91H). It is not necessary to use pauses between consecutive data bytes and it is not necessary to set any additional registers. The following sequence can be repeated as many times as necessary, with PCM data starting from (HL):

```
LD   A,(HL)
INC  HL
OUT  (#91),A
```

Timing, in the case of MSX2 onwards, can be obtained by the VDP, setting the line interruption, which makes the VDP send interruption signals at the frequency of 15.75 KHz, which allows to obtain a good sound quality. For MSX1 it is more complicated, and the timing must be obtained via software, counting the CPU cycles.

Chapter 7
MASS STORAGE SYSTEMS

Large external mass storage capacity combined with high access speed and high reliability are necessary requirements for a large number of applications. These requirements are fulfilled by mass storage devices (disk-drive, hard disks, ZIP drive, CD-ROM, DVD-ROM, Memory Cards, etc). These peripherals are normally drivered by BDOS (Basic Disk Operating System) routines, currently known as Kernel. In the case of MSX, direct access to these devices is not recommended, since each manufacturer is free to choose any type of controller for the disk system. Access must be done through BDOS or BIOS, through the PHYDIO and FORMAT routines.

Currently, there are several disk systems available for MSX: MSXDOS, MSXDOS2, Nextor, UZIX, SymbOS and WiOS, the latter two using graphical interfaces.

MSXDOS needs 64 Kbytes of RAM and can access up to six simultaneous drives, designated by A: to E:, but it's very simple. Although it can be connected to HD's, the control of the files is poor due to the fact that there are no subdirectories.

MSXDOS2 needs 128 Kbytes of mapped memory and accepts up to 8 simultaneous drives, from A: to H:, with the H: drive being configured as RAMDISK. This system has subdirectories, and can be easily configured for use with hard drives. Nextor is an evolution of MSXDOD2 and has native access to FAT16, allowing partitions up to 2 or 4 Gigabytes (FAT12 only allows partitions up to 32 Mbytes).

UZIX is a UNIX based system. Requires a minimum of 256 Kbytes to work well, has subdirectories, is multitasking and multi-user and was specially developed to be used with hard drives, but uses a different file system than MSXDOS and MSXDOS2, called "ext".

SymbOS is a graphical multitasking system and uses FAT as the file system, but is incompatible with MSXDOS programs. It needs a minimum of 128 Kbytes, directly accesses up to 1 Mbyte of memory and

works with FAT12 and FAT16. Optionally, it can also work with V9990 for optimized graphics.

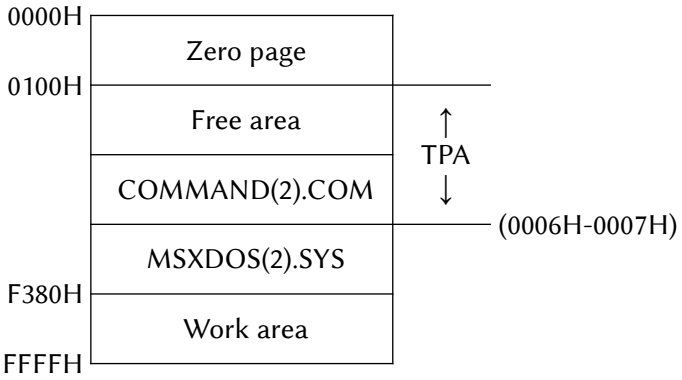
WiOS is a very specific system that only runs on the V9990. It needs 160 Kbytes of memory and is multitasking, but it works with the MSXDOS2 Kernel.

The standard for physical formatting of MSX disk devices is the same for all systems. They are illustrated below.

Media ID	1DD 3½ F8H	2DD 3½ F9H	1DD 5¼ FCH	2DD 5¼ FDH	HD F0H
Number of sides	1	2	1	2	-
Tracks per side	80	80	40	40	-
Sectors by track	9	9	9	9	63
Bytes per sector	512	512	512	512	512

7.1 – MSXDOS, MSXDOS2 and NEXTOR

MSXDOS (1 or 2) consists of the following modules: disk interface with BDOS (Kernel) in ROM and MSXDOS.SYS and COMMAND.COM files (for MSXDOS2 these are MSXDOS2.SYS and COMMAND2.COM). The MSX disk system differs from other systems in that DOS itself is not located on the system disk, but in the ROM of the disk interface, so Disk-BASIC does not need any disk in the drive to work. The MSXDOS.SYS and MSXDOS2.SYS files simply serve as a kind of boot to set the parameters necessary for COMMAND.COM or COMMAND2.COM to work, which are responsible for executing MSXDOS commands. The disk interface ROM includes drive routines, the DOS Kernel and the Disk-BASIC interpreter, and is located between addresses 4000H and 7FFFH (page 1) for MSXDOS and MSXDOS2, although the latter has 4 pages (64K) that are exchanged exclusively on physical page 1. After the system has been loaded into memory, the ROM is normally turned off and all RAM is enabled, as shown in the illustration below.



The area between 0000H and 00FFH is the zero page (system scratch area) and is extremely important for MSXDOS and application programs. This area will be described in detail later. The area that starts at 0100H and ends at the address indicated by the zero-page bytes 0006H/0007H is called TPA (Transient Program Area) and it is where programs that run under DOS are loaded. COMMAND.COM is located on top of the TPA and MSXDOS.SYS starts at the first address after the TPA.

7.1.1 – COMMAND.COM

The COMMAND.COM file is responsible for executing MSXDOS commands. These commands can be internal, external or batch.

Built-in commands are those that reside in COMMAND.COM itself. When called, they are executed immediately.

In the case of external commands, COMMAND.COM loads the routine from the disk (which must have the extension '.COM') and places it in the TPA from address 0100H, and the execution of the command starts at that same address. When the execution of the external command finishes (via a RET instruction), MSXDOS.SYS examines whether COMMAND.COM has been destroyed (in the case of very large external routines) and, if necessary, reloads COMMAND.COM and gives it the control.

Batch commands are a series of commands recorded in a file (with the extension .BAT) that COMMAND.COM executes one by one, sequentially (for COMMAND 2.41 there may be a conditional branch). The commands present in a batch file can be both internal and external, and even another batch command is possible. In this case, the called batch command destroys the calling batch command.

7.1.2 – MSXDOS.SYS

MSXDOS.SYS is the core of MSXDOS. It controls access and communication with disk devices. The MSXDOS.SYS functions are performed by the BDOS (Basic Disk Operating System), present in the ROM of the disk interface, which constitutes what is called by DOS Kernel. MSXDOS.SYS is just the intermediary between the I/O operations required by COMMAND.COM or external commands and the DOS Kernel.

7.1.3 – The DOS Kernel

The DOS Kernel contains the basic I/O routines for accessing disk devices. It resides in the disk interface ROM and performs the BDOS functions of MSXDOS.SYS. Any system that uses disk access can work perfectly using just the DOS Kernel. DISK-BASIC performs its operations by calling the DOS Kernel directly, not needing the system disk.

7.1.4 – Structure of files on disk

Information about the structure of data on disk and how it is controlled is important for developing programs that access the disk. This section contains all the information necessary for this.

7.1.4.1 – Sectors

Each disc type has a certain number of tracks; thus, 5¼" floppy disks have 40 or 80 tracks and 3½" floppy disks have 80 tracks. In the MSX system, each track is divided into 9 parts of 512 bytes each, called "sectors". The DOS Kernel considers each sector to be the basic data unit on the disk. Sectors are addressed by numbers, starting from 0, up to a maximum that depends on the disk capacity.

7.1.4.2 – Clusters

Although they are considered basic disk data units, it is not by sectors that the DOS Kernel controls the data on the disk, but by units called "clusters". A cluster can contain one or more sectors. In the case of floppy disks, each cluster occupies two sectors. In a HD formatted with FAT12, each cluster occupies 16 sectors (8 Kbytes) for partitions of 32 Mbytes. In the case of FAT16, each cluster occupies 64 sectors (32 Kbytes) for 2 Gbyte partitions. Nextor accesses 64 Kbyte clusters, allowing up to 4 Gbytes per partition.

7.1.4.3 – Data areas on disk

In MSXDOS, a disk is divided into 4 main areas, shown in the table below. The data itself is placed in the "data area". The boot sector is always sector 0, but the start sectors of the other areas (FAT, directory and data area) differ depending on the type of disk. This information is contained in the DPB.

Boot sector: MSXDOS startup program and information.
FAT: physical and logical control of the data area.
Directory: information about the files in the data area.
Data area: area for user data.

<table>
<tr><td rowspan="4">↑
Entire
disk
↓</td><td>Boot sector</td><td>Sector #0</td></tr>
<tr><td>FAT</td><td rowspan="2">The start sector and the size of these areas must be obtained from DPB</td></tr>
<tr><td>Directory</td></tr>
<tr><td>Data area</td><td>Last sector</td></tr>
</table>

7.1.4.4 – The boot sector and DPB

The acronym DPB comes from the "Drive Parameter Block". For each connected drive, MSXDOS allocates one DPB in RAM. The information contained in the DPB is originally copied from the boot sector of the disk during boot, although some data is different between the boot sector and the DPB.

The table below describes the contents of the boot sector and the DPB.

Boot sector offset

0BH/0CH	Size of a sector (in bytes).
0DH	Size of a cluster (in sectors).
0EH/0FH	Number of reserved sectors.
10H	Number of FAT's.
11H/12H	Number of root directory entries.
13H/14H	Number of disk sectors.
15H	Disc type identification.
16H/17H	FAT size (in sectors).
18H/19H	Number of sectors per track.
1AH/1BH	Number of disk faces.
1CH/1DH	Number of hidden sectors.

DPB offset

+0	Drive number (0=A:, 1=B, etc).
+1	Disc type identification.
+2/+3	Sector size in bytes.
+4	Directory mask.
+5	Directory size in sectors.
+6	Cluster Mask.
+7	Cluster size in sectors.
+8/+9	First sector of FAT.
+10	Number of FAT's.
+11	Number of root directory entries.
+12/+13	First sector of the data area.
+14/+15	Total disk clusters + 1.
+16	Number of sectors per FAT.
+17/+18	First sector of the directory area.
+19/+10	Address of FAT in RAM.

To access the DPB information, the 1BH function of BDOS can be used, which, among other data, brings the DPB address in RAM.

7.1.4.5 – The FIB (MSXDOS2)

The acronym FIB comes from the "File Info Block". It only exists for MSXDOS2 and is used for more complex operations, such as searching for unknown file directories or subdirectories. It is a 64-byte area in RAM that contains information about directory entries or specific fi-

les or subdirectories. To obtain the FIB information, use the MSXDOS2 functions 40H, 41H or 42H.

offset FIB information.

+0	Always FFH.
+1/+13	ASCII file name.
+14	Byte of file attributes.
+15/+16	Time of last modification of the file.
+17/+18	Date the file was last modified.
+19/+20	Initial cluster of the file.
+21/+24	File size.
+15	Logical drive number.
+26/+63	Internal information (do not modify).

The FFH byte at the beginning serves to distinguish the FIB from a path/filename string. FIB data is stored in the same format as directory data. They are detailed in the "DIRECTORY" section below.

7.1.4.6 – FAT (File allocation table)

The acronym FAT comes from "File Allocation Table". It is a kind of disk map. In MSXDOS, the cluster is the basic unit of data on disk. For large files, multiple clusters are used to store them. However, if multiple files are created and deleted, empty clusters are left between the undeleted files. When a larger file is created, it is divided into several parts and these are written to the available clusters. It is necessary, therefore, a way to know how many and which clusters are available and in how many and in which clusters the desired file is. This is the function of FAT.

When a bad cluster is found, FAT is also used to register it and prevent access to it. Information about clusters, including bad ones, is needed for handling files on disk. Without this information, the disk is unusable. that's why there are two FAT's, if there is a problem with one, there is the other.

Currently, there are two types of FAT for MSX: FAT12 and FAT16.

FAT12

The structure of FAT12 is illustrated below. The first byte is called the FAT ID and indicates the type of disk (the same value contained

in the boot sector and in the DPB). The next two bytes contain the "dummy". From the fourth byte (initial address + 3), information about the clusters (link) is written in an irregular format of 12 bits per cluster. Each group of 12 bits is called a "FAT entry". The FAT entry number is the corresponding cluster number on the disk.

	4 bits	4 bits	
Start adress →	F	9	FAT ID (80 tracks, 9 sectors)
	F	F	Dummy
	F	F	Dummy
	0	3	1st FAT entry – link = 003H
	4	0	
	0	0	2nd FAT entry – link = 004H
	F	F	3rd FAT entry – link = FFFH (end)
	6	F	
	0	0	4th FAT enty – link = 006H
	F	F	etc.

The "link" information indicates the next cluster of the corresponding file. The example above shows a file that occupies two clusters (003H and 004H). When the "link" value is FFFH, it means the file has ended. In practice, the "link" numbers are not necessarily in numerical order. The illustration below shows how the "link" numbers are organized in FAT.

2	1	link = 321H
4	3	
6	5	link = 654H

As we have 12 bits, theoretically FAT12 could address up to 4096 clusters (2^12). However, it can only address a maximum of 4079 clusters. This is because clusters numbered from FF0H to FFFH have special meaning, as shown in the table below.

Link	Description
000H	Available cluster (not used).
002H to FEFH	Used; point to the next cluster.
FF0H to FF6H	Reserved clusters.
FF7H	Damaged cluster.
FF8H to FFFH	Used; last file cluster.

FAT12 is quite efficient for mapping data onto floppy disks. However, it limits disk access to 32 Mbytes. Above that, it is necessary to create more partitions. This limitation occurs because in both the boot sector and the DPB the number of sectors on the disk is specified in two bytes, totaling a maximum of 65 536 sectors on the disk. So, since each sector is 512 bytes, you can do 65 536 * 512, which gives 32 Mbytes. For FAT to be able to address this total, each cluster must have 8 Kbytes, which leads to a certain waste of disk space. To be able to access larger partitions, FAT16 must be used.

FAT16

As the name implies, FAT16 uses 16 bits to address clusters. The organization of FAT16 is illustrated below.

	4 bits	4 bits	
Start adress →	F	0	FAT ID
	F	F	Dummy
	F	F	Dummy
	1	2	1st FAT entry – link = 1234H
	3	4	
	1	2	1st FAT entry – link = 1235H
	3	5	
	F	F	1st FAT entry – link = FFFFH (end)
	F	F	
	0	0	etc.

As with FAT12, there are also some "link" numbers with special meaning, as illustrated in the table below.

Link	Description
0000H	Available cluster (not used).
0001H to FFEFH	Used; point to the next cluster.
FFF0H to FFF6H	Reserved clusters.
FFF7H	Damaged cluster.
FFF8H to FFFFH	Used; last file cluster.

FAT16 on MSXDOS uses the same schema as FAT16 on PC; therefore, a FAT16 partition can be up to 2 Gbytes. For partitions of this size, however, the clusters are huge (32 Kbytes), which is a huge waste of disk space.

FAT16 only exists as a patch for MSXDOS2; is not available for MSXDOS1. To be able to use it, however, it is necessary to have a FAT12 partition to boot the system and load the patch, since the DOS Kernel only works natively with FAT12.

In turn, Nextor has native support for FAT16 partitions, being ideal for working with mass storage, as it can access up to 4 Gbytes per partition.

7.1.4.7 – The directory

FAT, described above, stores the location of a file's data on disk, but does not contain any information about the file's contents. Therefore, there is a section on the disk called directory, where the information about the file is. Each directory entry is made up of 32 bytes containing the name, attributes, time and date of file creation, in addition to the first cluster and its size, as illustrated below.

Offset	Description
+0/+7	File name (up to 8 characters).
+8/+10	Length (up to 3 characters).

+11 Attributes byte – b7 b6 b5 b4 b3 b2 b1 b0

- b0 – If this bit is 1, the file can be read but not deleted or modified (MSXDOS2 only)
- b1 – If this bit is 1, the file name will not appear in the DIR or FILES command, but can be accessed normally (MSXDOS1 and MSXDOS2)
- b2 – Same as b1, but BDOS functions cannot delete or modify the file and it cannot be accessed by COMMAND2.COM. It means it is a system file (MSXDOS2 only).
- b3 – If this bit is 1, the 11 bytes of the file name will contain the disk name (volume name) and the rest of the directory will be ignored (MSXDOS2 only).
- b4 – If this bit is 1, the file is a subdirectory and cannot be read or written normally. When listed with the DIR command, the expression "<DIR>" will appear in place of the file size (MSXDOS1 and MSXDOS2, but MSXDOS1 will not be able to access the subdirectory).
- b5 – If this bit is 1, the file cannot be closed before writing (MSXDOS2 only).
- b6 – Reserved (always 0).
- b7 – If this bit is 1, all others will be ignored and the FIB will point to a device character (eg ".CON" – console input). MSXDOS2 only.

+12/+21 Reserved (do not use).
+22/+23 Time of file creation.
+24/+25 Date of file creation.
+26/+27 First cluster in the file.
+28/+31 File size in bytes.

Time

[23th byte]								[22th byte]							
b7	b6	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	b3	b2	b1	b0
h4	h3	h2	h1	h0	m5	m4	m3	m2	m1	m0	s4	s3	s2	s1	s0
Hour(0~23)					Minute (0~59)						Second (0~29)				

Note: To get the correct value of seconds, multiply the value of the register by 2.

Date

[25th byte]								[24th byte]							
b7	b6	b5	b4	b3	b2	b1	b0	b7	b6	b5	b4	b3	b2	b1	b0
a6	a5	a4	a3	a2	a1	a0		m3	m2	m1	m0	d4	d3	d2	d1	d0
Year (0~99)								Month (1~12)				Day (1~31)			

Note: to get the correct year, add 1980 to the value of the register (1980 to 2079).

The first sector of the directory can be obtained from the respective DPB (or from the boot sector). When a file is created, the respective directory entry is placed in the free part closest to the beginning of the directory. Each directory entry is initially padded with 00H bytes. If a file is created and then deleted, only the first byte of the respective entry in the directory is changed to E5H. When all directory entries are filled, no more files can be created even if there is space available on the disk. The number of entries in the directory can also be obtained from the respective DPB.

Subdirectories (MSXDOS2)

Only MSXDOS2 and NEXTOR can manipulate subdirectories. The subdirectory is a special type of file whose structure is identical to the directory. As it is a file, however, there is no area reserved for it; is in the data area of the disk. Its operation is extremely simple: the initial cluster of the directory points to the file that is the subdirectory. Bit b4 of the attributes byte must be set.

A subdirectory does not have a fixed size and therefore has no limit on entries. As more entries are added, the subdirectory will increase in size as needed.

When a subdirectory is created, two special files, which are "inside" it, are created simultaneously: the "." and the "..". These files are used to exit the subdirectory and return to the previous root directory or subdirectory. They cannot be erased or manipulated.

7.1.5 – Access to disk files

When talking about file access, an acronym should be kept in mind: FCB. This acronym comes from the "File Control Block". All information recorded on the disk is given the file name. Each file is named, consisting of up to 8 characters plus an optional extension of three (eg MSXDOS.SYS). Direct file access using directory and FAT is very complex; that's why the FCB exists. It occupies 37 bytes and the programmer only needs to specify the file name and the drive to access it. The FCB can be located anywhere in memory, but MSXDOS normally uses address 005CH to store it. The structure of the FCB is described below.

Offset	Comments
+0	Drive number (0=default; 1=A:, 2=B:, etc).
+1/+11	File name and extension.
+12/+13	Current block.
+14/+15	Random record size in bytes.
+16/+19	File size in bytes.
+20/+21	Date (same format as directory).
+22/+23	Time (same format as directory).
+24	Device ID.
+25	Directory location.
+26/+27	First cluster in the file.
+28/+29	Last cluster accessed.
+30/+31	Relative cluster location.
+32	Current sequential record.
+33/+36	Random record number.

• Drive number (00H)

Indicates the disk-drive on which the disk containing the file is located.

• File name (01H to 08H)

The file name can contain up to 8 characters. When you have less, the remaining bytes will be padded with spaces (20H).

• Extension (09H to 0BH)

The file name extension can be up to 3 characters long. When you have less, the remaining bytes will be padded with spaces (20H). The extension is optional.

• Current block (0CH to 0DH)

Indicates the current block number for sequential access (BDOS functions 14H and 15H).

• Random record size (0EH to 0FH)

Specifies the size in bytes of the data unit (record) for random reading or writing (BDOS functions 14H, 15H, 21H, 27H, and 28H).

• File size (10H to 13H)

Contains the file size in bytes.

• Date (2pm to 3pm)

Date of last access to the file. The format is the same as the directory.

• Time (4pm to 5pm)

Time of last file access. The format is the same as the directory.

• Device ID (18H)

When a peripheral is opened as a file, the value listed below is specified in that byte. For normal files, the value of this field is 40H + drive number. For example, the byte ID of drive A: is 41H. For future expansions, application programs must not use the byte ID.

Byte	Device ID
FFH	CON (console or keyboard).
FEH	AUX (auxiliary).
FDH	NUL (null).
FCH	LST (list on printer).
FBH	PRN (printer).

• Directory location (19H)

Indicates the entry position in the file directory.

• First cluster in the file (1AH to 1BH)

Contains the number of the first cluster of the file on disk.

• Last cluster accessed (1CH to 1DH)

Contains the last cluster number accessed.

• Relative cluster location (1EH to 1FH)

Indicates the relative location of the last cluster accessed from the first cluster in the file.

• Current sequential record (20H)

Contains the current record number for sequential access (BDOS functions 14H and 15H).

• Random record number (21H to 24H)

Contains the random record number to be accessed. By specifying a value from 1 to 63 for the record size, all 4 bytes from 21H to 24H are used. When the record size is greater than 63 only bytes 21H to 23H have meaning (BDOS functions 14H, 15H, 21H, 22H, 27H and 18H).

7.1.5.1 – Opening a file

Before accessing a file, it is necessary to open it. "Open a file" means to transform incomplete information contained in the FCB (only file name and drive number) into all the information that the FCB may contain.

When opening a file, the drive number in the FCB is converted to the real drive (1 to 6 for MSXDOS1 or 1 to 8 for MSXDOS2) and the other FCB fields are filled in (BDOS function 0FH).

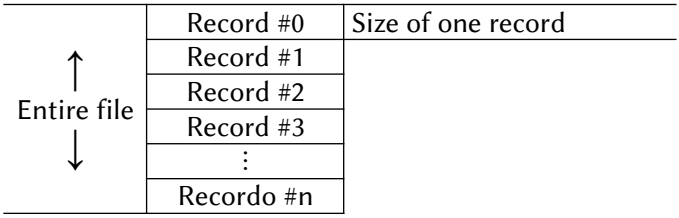
7.1.5.2 – Closing a file

When a file is opened, the information contained in the directory is transferred to the FCB. While handling the file, the contents of the FCB fields are being modified. Therefore, after completing the handling of the file, it is necessary to close it. The operation of closing a file causes the information contained in the FCB to return to the updated directory, in order to allow later accesses (function 10H of BDOS).

7.1.5.3 – Sequential and random access

In random access, the records that make up the file can be accessed freely, without any established pattern, unlike sequential access, where the records are accessed one after the other, without fail. The record size can be any size, as long as it is greater than or equal to one byte up to a limit of 64 Kbytes. The record can even be the size of the entire file (extreme sequential access) or just one byte (extreme random ac-

cess). The default value for record size is 128 bytes. A file with their respective records is illustrated below.



7.1.5.4 – Headers

In order for the disk system to correctly recognize, load and execute (if applicable) the files or programs stored on the disk, these usually contain a header. The header varies depending on the file type. The different types of headers are described below.

Binary files

Binary files contain a 7-byte header whose structure is as follows:

Offset	Content
0	File type (FEH = binary).
1–2	Starting address of data in RAM.
3–4	End address of data in RAM.
5–6	Run address (for executable files).

This file type is used by BASIC to manipulate blocks of data directly in RAM or VRAM and also to save, load, and run assembly programs.

BASIC text files

BASIC programs are saved to disk preceded by an FFH byte. The format of the data after this byte is identical to the tokenized text stored in RAM. BASIC text can also be saved in ASCII format.

ASCII and text files

These files have no header. The end of line is normally indicated by the combination of the bytes 0DH+0AH (carriage return and line feed). The end of the ASCII file must be marked with a 1AH byte (EOF – end of file). DOS '.BAT' (batch) files are text files.

'.COM' files

CP/M and MSXDOS executable files (extension '.COM') have no header and no specific format. They are always loaded and executed at address 0100H. Because of CP/M file system compatibility, the size of CP/M files must be a multiple of 80H.

Other files

For other file types, there is no particular format. They are recognized exclusively by the extension of their name.

7.1.5.5 – Handle files (MSXDOS2)

A file handle is nothing more than a number that the user associates with a common device or file. The value of a handle file can vary from 0 to 63. Using only the handle number as a reference, you can manipulate the file or device associated with it. This file type is only supported by MSXDOS2 through functions added to BDOS, such as 43H, 44H, 45H, 53H and others.

The internal memory area used by the handle files is allocated on a logical page (16K) outside the TPA area, therefore not reducing its size.

Handle files 0 to 4 are predefined, as described below.

0 – Standard input (CON).
1 – Standard output (CON).
2 – Standard error input/output (CON).
3 – Standard auxiliary input/output (AUX).
4 – Standard printer output (PRN).

7.1.6 – The functions of BDOS (Kernel)

BDOS consists of a set of routines that perform basic I/O operations for disk devices. These routines allow easy access to the disk system and reside in the disk interface ROM. Also known as DOS Kernel.

The BDOS functions are available for both MSXDOS and Disk-BASIC, just varying the call address:

MSXDOS: 0005H

Disk-BASIC: F37DH (&HF37D)

To run the BDOS functions, simply do the following:

1. Load the CPU C register with the desired function number;
2. Load registers A, B, DE and HL (if necessary) with the proper values;
3. Make a call (CALL) to the BDOS address (0005H for MSXDOS or F37DH for Disk-BASIC).

The following example illustrates a call to the BDOS function 1FH.

```
LD    A,000H  ;loads A with 00H
LD    C,01FH  ;loads C with function number 1FH
CALL  00005H  ;executes the function
```

It is important to note that calls to BDOS destroy the contents of the registers. Therefore, before calling any function, the contents of registers that should not be modified must be saved.

There are 44 calls for BDOS in the case of MSXDOS1 and 94 for MSXDOS2 (which includes all the functions of MSXDOS1). The functions are numbered from 00H to 70H, but there are some that are not implemented: 1CH to 20H, 25H, 29H and 32H to 3FH. A call to these functions just returns the value 0 in the A register.

Whenever accessing disk devices, it is advisable to use BDOS functions. Direct access is quite complicated, as each manufacturer can use the controller that suits them best and programs may not work on different interfaces.

The BDOS functions are described in detail in the appendix (Chapter 4 – MSXDOS, item 4.3 – Calls to BDOS)

7.1.6.1 – Absolute reading/writing of sectors

MSX accesses disk through 'logical sectors'. They are defined independently of the physical sectors of the disk and are numbered from 0 to a maximum that depends on the disk's capacity:

40 tracks, 1 side: 0 to 359
40 tracks, 2 sides: 0 to 719
80 tracks, 1 side: 0 to 719
80 tracks, 2 sides: 0 to 1439
32 Mb partition: 0 to 65 535 (2 bytes)
2Gb partition: 0 to 4,294,967,296 (4 bytes, FAT16 only)

The BDOS functions described below directly access the logical sectors of the disk.

RDABS (2FH)
Function: Read logical sectors from disk. The sectors read are placed from the DTA.
Input: DE – Number of the first logical sector to read;
H – Number of sectors to read;
L – Drive number (0=A:, 1=B:, etc.).
Output: A = 0 ⟶ The reading was successful;
A ≠ 0 ⟶ Error code.

WRABS (30H)
Function: Writing of logical sectors to disk. The data to be written to disk will be read into RAM from DTA.
Input: DE – Number of the first logical sector to be written;
H – Number of sectors to write;
L – Drive number (0=A:, 1=B:, etc.).
Output: A = 0 ⟶ The writing was successful;
A ≠ 0 ⟶ Error code.

To access partitions larger than 32 Mbytes it is necessary to have installed the FAT16 driver or Nextor, which has native support. The latter implements two functions for direct access:

RDDRV (73H)
Function: Read the absolute sectors of the unit. This function is able to read sectors regardless of the file system viewed on the drive (FAT12, FAT16 or an unknown file system), and even when there is no file system. The read sectors will be placed from the current disk's DTA.
Input: A – Unit number (0 = A:, 1=B:, etc.).
B – Number of sectors to read.
HL:DE – Sector number.
Output: A – Error code (if it is 0, there was no error).

WRDRV (74h)
Function: Write absolute sectors to disk. This function is able to record sectors regardless of the file system viewed on the drive (FAT12, FAT16 or an unknown file system), and even when there is no file system. The sectors will be written from the current disk's DTA.
Input: A – Unit number (0 = A:, 1=B:, etc.).
B – Number of sectors to read.
HL:DE – Sector number.
Output: A – Error code (if it is 0, there was no error).

7.1.6.2 – Access to files using FCB

Accessing disk files using direct access to logical sectors is a very complicated process. BDOS functions that access the disk using the FCB make these operations simpler.

There are three categories of file access using FCB: sequential access, random access, and block random access. This last type has the following facilities: records of any size can be specified; access can be done on multiple records and file size is controlled in bytes.

An important information is that some functions do not work correctly when the FCB is located between addresses 4000H and 7FFFH (MSXDOS1 and MSXDOS2): SFIRST function (11H), SNEXT function (12H) and the I/O functions for devices (CON, PRN, NUL, AUX).

The following functions allow access to files using FCB:

FOPEN (0FH)	Open file (FCB).
FCLOSE (10H)	Close file (FCB).
SFIRST (11H)	Search for the first file. This function acceptswildcard characters (* and ?).
SNEXT (12H)	Search for the next file. This function accepts wildcard characters (* and ?).
FDEL (13H)	Delete files. Wildcard characters (* and ?) can be used.
RDSEQ (14H)	Sequential reading.
WRSEQ (15H)	Sequential write.
FMAKE (16H)	Create files.
FREN (17H)	Rename files. The wildcard character "?" Can it be used to rename multiple files simultaneously.
RDRND (21H)	Random reading. The read record will be placed in the area indicated by the DTA and has the fixed size of 128 bytes.
WRRND (22H)	Random writing.
FSIZE (23H)	Read the file size.
SETRND (24H)	Set random record field.
WRBLK (26H)	Random block write. the registration number random is automatically incremented after of writing, and its size can vary from 1 up to 65 535 bytes.
RDBLK (27H)	Block random access.
WRZER (28H)	Random writing with 00H bytes. This function is the same to 22H (WRRND), except for filling in the remaining file records with 00H bytes, if the specified record is not the last in the file.

The functions below were added by MSXDOS2 and are not available for the first version of MSXDOS.

FFIRST (40H)	Looks for the first entry in the directory.
FNEXT (41H)	Finds next directory entry. This function should only be used after function 40H. She accepts the wildcard characters "?" and "*" set to 40H.
FNEW (42H)	Search for new entry
OPEN (43H)	Open file handle. If the "inheritable" bit of A is set, the handle file must be opened by another process (see function 60H).

CREATE (44H)	Create file handle. The file created by this function will automatically open (function 43H)
CLOSE (45H)	Close file handle.
ENSURE (46H)	Protect file handle (the file pointer current cannot be modified).
DUP (47H)	Duplicate file handle.
READ (48H)	Read from a file handle. The four sequences control (Ctrl+P, Ctrl+N, Ctrl+S, Ctrl+C) are checked. WRITE (49H) Write by a file handle. If the end of file is found, it will be extended up to the value required.
SEEK (4AH)	Move handle file pointer.
IOCTL (4BH)	Control for I/O devices.
HTEST (4CH)	Test handle file.
DELETE (4DH)	Delete file or subdirectory. one subdirectory only can be deleted if it does not contain any file. If a device name is specified, will not return an error, but the device will not be "wiped out".
RENAME (4EH)	Rename file or subdirectory.
MOVE (4FH)	Move file or subdirectory. A file does not can be moved if the respective handle file is open. The FIB of the moved file will not be updated.
ATTR (50H)	Set or read attributes from a file. the attributes of a file cannot be modified if the corresponding handle file is open.
FTIME (51H)	Read or set date and time from a file.
HDELET (52H)	Delete file handle. If there is another file open handle to the same file, this one is not can be erased.
HRENAM (53H)	Rename file handle. The file cannot be renamed if there is another handle file open to the same file. This function is identical to the 4EH function, except that the HL register not be able to point to a FIB.
HMOVE (54H)	Move file handle. The file cannot be moved if there is another handle file open for the same file. This function is identical to the function 4FH, except that the HL register cannot point to a FIB.
HATTR (55H)	Read or set attributes from file handle. the byte of attributes cannot be modified if there is another file handle open to the same file.

HFTIME (56H) Read or change time and date from file handle. If there is another handle file open for the same file, the date and time cannot be modified. This function is identical to function 51H, except there is no pointer; only the handle file.
GETDTA (57H) Read the address of the DTA (Disk Transfer Area).
GETVFY (58H) Read write verification flag.
GETCD (59H) Read current directory or subdirectory.
CHDIR (5AH) Change the current subdirectory.
PARSE (5BH) Parses pathname.
PFILE (5CH) Parse filename.
CHKCHR (5DH) Checks character. 16-bit characters are also checked.
WPATH (5EH) Read full path string, no drive specification and character "\". For greater reliability, call first function 40H or 41H and then call WPATH twice, as other functions may change the data.
FLUSH (5FH) Flush disk buffers.
FORK (60H) Branch files into tree.
JOIN (61H) Join files into tree. This function returns to the original handle file the handle file that was copied by the previous function.
TERM (62H) End with error code.

These routines (as well as all the others) are described in detail in the appendix (Chapter 4, item 4.3 – Calls to BDOS).

7.1.7 – System area for MSXDOS

The system disk area, for both MSXDOS1 and MSXDOS2, takes up a good amount of memory just below the system work area (which starts at F380H). MSXDOS1 takes up more memory in this area because it copies the FAT from the disk being used and also from the virtual drive B:. Therefore, when pressing the CTRL key during reset, disabling drive B:, there is an increase of 1.5 Kbytes in available memory. MSXDOS2, on the other hand, copies the FAT in another memory area, and the savings when deactivating drive B: is only 21 bytes, referring to the respective DPB.

So a standard MSX with no disk interface can use up to address F08FH. The space between F08FH and F380H is used.

An MSX loaded with MSXDOS and two drives can be used up to around DB80H, with some variation up or down depending on the interface. If CTRL is pressed at boot to free a drive unit, the value goes up to about E160H.

With MSXDOS2, the address is around E240H with two drives, decreasing just 21 bytes for each drive installed.

In any case, the system variable HIMEM (FC4AH, 2) which stores the highest available address can be consulted.

The MSXDOS and MSXDOS2 system variables are described in detail in the appendix, chapter 7, items 7.1 and 7.2 (System area for MSXDOS and MSXDOS2)

7.1.8 – Disk interface routines

There are some BDOS routines that are called directly from the disk interface. These routines have their entry on page 1, so it is not advisable to call them directly, because under MSXDOS page 1 contains RAM and under BASIC it contains the interpreter ROM. So the DOS Kernel ROM will never be active normally.

Thus, the interface routines must be called by the BIOS CALSLT routine, which is normally active under both MSXDOS and BASIC. The call sequence should be as follows:

```
CALSLT:   EQU  0001CH    ;CALSLT routine adress
HPHYD:    EQU  0FFA7H    ;PHYDIO hook
CALBAS:   EQU  04022H    ;CALBAS routine adress
          LD   IX,CALBAS ;IX <- CALBAS routine adress
          LD   IY,HPHYD  ;IY <- disk interface slot
          CALL CALSLT    ;execute CALBAS routine
```

To use the CALSLT routine it is necessary to know the slot where the interface is installed. A safe place to get this information is the disk command hooks. They contain the primary disk interface slot ID in their second byte. In this case, the BIOS PHYDIO routine hook was used.

The disk interface routines are described in detail in the appendix, chapter 8, item 8.6 (Disk interface routines). All registers are modifi-

ed by the routines; therefore it is necessary to save on the stack the registers that should not be modified.

7.1.9 – Page zero

Page zero is the memory area located between RAM addresses 0000H and 00FFH, occupying 256 bytes. This area is only active under MSXDOS and is extremely important for application programs. Some BIOS routines are available in this area. They must be called exactly as they are for the BIOS (CALL or RST instruction), except the WBOOT routine (0000H), which must be called with a JP 0000H. Page zero is mapped as described below.

WBOOT (0000H, 3)
: Warm boot. By calling this routine, a match is promoted hot MSXDOS (MSXDOS is reloaded without restarting the computer).

DRIVE (0004H, 1)
: This byte stores the default drive (00H=A:, 01H=B:, etc).

BDOS (0005H, 3)
: Entry point for BDOS routines.

RDLT (000CH, 8)
: This routine reads one byte in any slot. it's just like the routine BIOS RDLT.

WRSLT (0014H, 8)
: This routine writes a byte to any slot. it's exactly the same as BIOS WRSLT routine.

CALSLT (001CH, 8)
: Calls a routine in any slot. In the present case, it may be output to call other BIOS routines. it's exactly the same as BIOS CALSLT routine.

ENASLT (0024H, 8)
: Enables a page in any slot. it's just like the routine BIOS RDLT.

CALLF (0030H, 8)

Calls a routine in any slot, with inline parameters. At the In this case, it can be used to call other BIOS routines. It is exactly the same as the BIOS CALLF routine.

INTPRT (0038H, 3)
Calls the interrupt handler routine. This entry does not must be used by the programmer.

CHSLTS (003BH, 33)
Routine used by the system to switch secondary slots. This entry must not be used by the programmer.

FCBDOS (005CH, 24)
This area contains the FCB used by BDOS.

DTA (0080H ~ 00FFH)
DTA starting address.

The area between 0080H and 00FFH (DTA) is where a line collected by COMMAND.COM is placed. For example, when typing an external command like "PROG ABC", COMMAND.COM will search the disk for the program named PROG.COM. If found, it will load it from address 0100H. The argument "ABC" will be loaded from address 0080H, with the structure shown below.

0080H – 20H (space)
0081H – 0DH (carriage return)
0082H – "A"
0083H – "B"
0084H – "C"
0085H – 0DH (carriage return)
0086H – 00H (end of argument)

After loading the argument and the program, its execution starts at address 0100H. The area from 0100H to the highest available address is known as the Transient Program Area (TPA).

7.1.10 – The boot sector

On all floppy disks (and other disk devices) there is a "boot sector", which is always sector 0 of the disk. Every time the computer is restarted, the DOS Kernel residing in the ROM of the disk interface checks

if there is any disk connected to the system. If not, activate Disk BASIC; otherwise, it loads the boot sector at address C000H (beginning of page 3 of RAM) and executes the routine contained at address C01EH. Below is illustrated how the boot sector is in memory.

C000H – Byte ID (55H for floppy disks)

C001H~C002H – FEH, 90H (DOS boot instruction, used in "warm" boot – WBOOT)

C003H~C00AH – Manufacturer name or formatting identification in ASCII. It can be modified by the programmer.

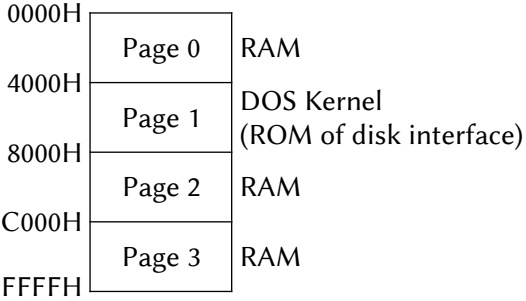
C00BH~C01DH – Boot sector data. These data are detailed described in section 1.4.4 (THE BOOT SECTOR and the DPB).

C01EH~C0FFH – Initialization routine. It is described in detail just ahead.

C100H~C1FFH – Reserved area. It must not be used.

Despite the sector having 512 bytes, the instructions contained in it can only have up to 256 bytes (C000H to C0FFH), because right after the boot sector is loaded, the DOS Kernel fills the area from C100H with specific routines. If the disk is not a system disk, page zero will also be filled, except for the WBOOT routine (0000H) and the BDOS entry (0005H). In this case, the BDOS entry in F37DH should be used.

The memory mapping at boot execution is illustrated below.



7.1.11 – The startup routine

After setting all the necessary data, the DOS Kernel passes control to the routine contained at address C01EH. The default routine used by the original MSXDOS1 is as follows:

```
      BDOS:   EQU  0F37DH
C01E          RET  NC
C01F          LD   (BOOT+1),DE
C023          LD   (0C04H,A)
C026          LD   (HL),056H
C028          INC  HL
C029          LD   (HL),0C0H
C02B  BOOT0:  LD   SP,0F51FH
C02E          LD   DE,FCBDOS
C031          LD   C,000H
C033          CALL BDOS
C036          INC  A
C037          JP   Z,BOOT2
C03A          LD   DE,00100H
C03D          LD   C,01AH
C03F          CALL BDOS
C042          LD   HL,00001H
C045          LD   (0C0ADH),HL
C048          LD   HL,03F00H
C04B          LD   DE,FCBDOS
C04E          LD   C,027H
C050          CALL BDOS
C053          JP   00100H
C056          LD   E,B
C057          RET  NZ
C058  BOOT1:  CALL 00000H
C05B          LD   A,C
C05C          AND  0FEH
C05E          CP   002H
C060          JP   NZ,BOOT3
C063  BOOT2:  LD   A,(0C0C4H)
C066          AND  A
C067          JP   Z,04022H
C06A  BOOT3:  LD   DE,ERROR
C06D          LD   C,009H
C06F          CALL BDOS
```

```
C072            LD    C,007H
C074            CALL  BDOS
C077            JR    BOOT0
C079  ERROR:    DEFB  'Boot error',00DH,00AH
      DEFB      'Press any key for retry'
      DEFB      00DH,00AH,024H
C09F  FCBDOS:   DEFB  000H,'MSXDOS  SYS'
C0AB  END
```

The rest of the boot sector is filled with 00H bytes.

Before executing the boot routine, the DOS Kernel fills the DE and HL registers with certain addresses and sets a value in the A register.

The initialization routine can be modified by the programmer to adapt it to the program he wants to put on the disk. The details to note are as follows: there must be a RET NC instruction at the beginning; the A, DE and HL registers contain a valid value; the mapping when executing the routine contains the DOS Kernel ROM on page 1 and RAM on pages 0, 2 and 3; the memory area reserved for the initialization routine is only 222 bytes (C01EH~C0FFH).

The initial three bytes of the boot sector (EBH, FEH, 90H) should not be modified by the programmer, despite the system changing the first byte (EBH on disk) to 55H in memory (720K floppy disks). Boot sector data must be set (C00BH~C01DH).

The default routine used by the original MSXDOS2 is as follows:

```
      BDOS:     EQU   0F37DH
C01E            JR    BOOT0
C020            DEFB  ´VOL_ID´
C026            DEFB  000H,015H,075H,005H,01BH
C02B            DEFB  000H,000H,000H,000H,000H
C030  BOOT0:    RET   NC
C031            LD    (BOOT2+1),DE
C035            LD    (BOOT3+1),A
C038            LD    (HL),067H
C03A            INC   HL
C03B            LD    (HL),0C0H
C03D  BOOT1:    LD    SP,0F51FH
```

```
C040           LD   DE,FCBDOS
C043           LD   C,00FH
C045           CALL BDOS
C048           INC  A
C049           JR   Z,BOOT3
C04B           LD   DE,00100H
C04E           LD   C,01AH
C050           CALL BDOS
C053           LD   HL,00001H
C056           LD   (0C0B9H),HL
C059           LD   HL,03F00H
C05C           LD   DE,FCBDOS
C05F           LD   C,027H
C061           CALL BDOS
C064           JP   00100H
C067           LD   L,C
C068           RET  NZ
C069  BOOT2:    CALL 00000H
C06C           LD   A,C
C06D           AND  0FEH
C06F           SUB  002H
C071  BOOT3:    OR   000H
C073           JP   Z,04022H
C076           LD   DE,ERROR
C079           LD   C,009H
C07B           CALL BDOS
C07E           LD   C,007H
C080           CALL BDOS
C083           JR   BOOT1
C085  ERROR:    DEFB 'Boot error',00DH,00AH
               DEFB 'Press any key for retry'
               DEFB 00DH,00AH,024H
C0AB  FCBDOS:   DEFB 000H,'MSXDOS  SYS'
C0B6           END
```

7.1.12 – The Nextor

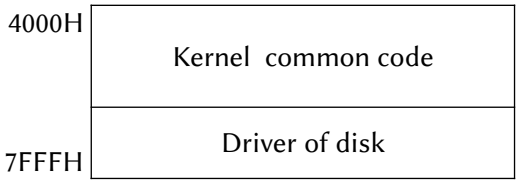
Nextor is an enhanced version of MSXDOS 2, based on version 2.31, with which it is 100% compatible.

7.1.12.1 – The MSXDOS 1 Kernel

The original MSXDOS Kernel (version 1) was written in a ROM built into external floppy controllers or in the internal ROM of MSX floppy drive. It has 16K and is located on page 1 (addresses 4000H to 7FFFH).

The MSXDOS 1 Kernel is divided into two main parts, the common kernel code and the disk driver. The common code part of the Kernel is not 100% driver independent.

The illustration below shows a diagram with the structure of an MSXDOS 1 Kernel.



There can be up to four MSXDOS Kernel ROMs active. If more than one is present, the one with the lowest slot number becomes the "master" (the one whose common Kernel code is actually executed), and the others are the "slaves" (only the driver code is executed).

MSXDOS sees storage devices as drive letters, while the disk driver has one or more driver drives. The mapping between the two entities is fixed and one by one, so for example drive A: is mapped to driver drive 0 of the first Kernel, drive B: is mapped to driver drive 1 and so on .

7.1.12.2 – The MSXDOS 2 Kernel

MSXDOS Kernel 2 uses the slot page 1 address space like MSXDOS Kernel 1. However, it has a size of 64K. This space is divided into four 16K banks that are swapped at runtime exclusively on physical page 1 (4000H ~ 7FFFH). The contents of the banks are as follows:

- Bank 0 contains common Kernel and disk driver code.
- Banks 1 and 2 contain the Kernel common code.

• Bank 3 contains a copy of the MSXDOS Kernel 1, with a copy of the disk driver code (MSX Turbo R machines only).

The illustration below shows a diagram with the structure of an MSXDOS 2 Kernel.

<table>
<tr><th></th><th>Bank 0</th><th>Banks 1 e 2</th><th>Bank 3</th></tr>
<tr><td>4000H</td><td>Page zero code</td><td>Page zero code</td><td rowspan="2">MSXDOS 1
Kernel</td></tr>
<tr><td>40FFH</td><td>Bank ID (0)</td><td>Bank ID
(1 or 2)</td></tr>
<tr><td>4100H</td><td>Kernel code
Bank 0</td><td rowspan="2">Kernel code
Banks 1 and 2</td><td rowspan="2">Driver of disk</td></tr>
<tr><td>7FD0H</td><td>Driver of disk</td></tr>
<tr><td>7FFFH</td><td>Bank switching
code</td><td>Bank switching
code</td><td>Bank switching
code</td></tr>
</table>

There are three parts that are common to all banks (bank 3 only contains the bank's exchange code):

1. The code on page 0 is 255 bytes long and contains an entry point for the interrupt routine, a routine for calling code in another bank, and other useful utility code.

2. The bank ID is just one byte with the bank number (needed to make inter-slot calls).

3. The bank change code is required to change the visible bank. The exact code placed here depends on the type of ROM mapper used (the original DOS 2 cartridge mapping is ASCII16).

When booting in DOS 2 mode, bank 0 is switched permanently and other banks are switched only temporarily when the code in bank 0 needs to call a routine or access data in one of these banks. When booting in DOS mode 1, bank 3 is activated at boot.

As was the case with the MSXDOS Kernel 1, up to four MSXDOS Kernel ROMs can be active at the same time, one being the "master" and the others the "slaves". However, this time the master will not be the Kernel with the lowest slot number, but the Kernel with the hig-

hest version number (the Kernel with the lowest slot number is still selected as the master in case two or more Kernels have the same version number).

7.1.12.3 – The Nextor Kernel

The Nextor Kernel has an architecture based on the MSXDOS 2 Kernel, with the following changes:

- There are more 16K banks.
- Disk driver code ("device driver" in Nextor terminology) is no longer embedded at the end of Kernel banks 0 and 3, but occupies an entire bank after the last bank of the Kernel common code. If necessary, the driver can occupy more than one bank.
- The device driver framework makes it easy to add BASIC extended commands ("CALL" commands), extended BIOS commands, and an interrupt service routine.
- Code page 0 has been modified to contain extra utility routines. These routines can be used by driver code.
- There is an information byte in the 4FFEH address of all banks, containing the size of the common Kernel code in 16K banks (alternatively, this value can be seen as the driver's bank number).
- The MSXDOS Kernel 1 in bank 3 has been modified so that it can make calls to the device driver.
- There is 1K unused space in banks 0 and 3 (visible at addresses 7BD0H to 7FCFH). This space does not contain any Kernel code and can be used to place any code or data that is required by the driver. See "The free space in the Kernel's main bank" section for more details.
- There are five entry points in Kernel banks 0 and 3 (starting at addresses 7850H) that will be redirected to another five entry points in the driver bank. In this way, the driver can provide code that will be accessible via direct cross-slot call to the Kernel slot. See DRV_DIRECT0/1/2/3/4 (4142H, 4145H, 4148H, 414BH, 414EH) sections for more details.

The illustration below shows a diagram with the structure of a Nextor Kernel ("K" is the Kernel common code bank counter).

<table>
<tr><th></th><th>Bank 0–(K–1)</th><th>Bank K</th><th>Banks (K+1) ... opt.</th></tr>
<tr><td>4000H</td><td>Page zero code</td><td>Page zero code</td><td>Page zero code</td></tr>
<tr><td>40FEH</td><td>K</td><td>K</td><td>K</td></tr>
<tr><td>40FFH</td><td>Bank ID</td><td>K (Bank ID)</td><td>Bank ID</td></tr>
<tr><td>4100H</td><td rowspan="2">Kernel code
of the bank</td><td rowspan="2">Driver
code</td><td rowspan="2">Driver
code
(additional)</td></tr>
<tr><td>7BD0H</td><td>Free space (1K)
(banks 0 and 3)</td></tr>
<tr><td>7FD0H</td><td>Bank switching
code</td><td>Bank switching
code</td><td>Bank switching
code</td></tr>
<tr><td>7FFFH</td><td></td><td></td><td></td></tr>
</table>

Nextor will use the same rule as MSXDOS 2 to decide which Kernel will be the master if more than one Kernel is found (the Kernel with the highest version number will win). However, this applies to other Nextor Kernels only; Nextor will always overwrite other MSXDOS 1 or 2 Kernels present on the system, regardless of their version number.

7.1.12.4 – Creating a Nextor Kernel with an embedded driver

To create a complete Nextor Kernel ROM, up to four components are required:

- The Kernel Nextor base file. This file contains the common kernel code, ie the "Bank 0–(K–1)" part shown in the illustration above. The bank switch code is for the ASCII16 mapper (the original mapper used by the MSXDOS 2 Kernel) .
- The device driver file. Its structure is detailed in section 4. The size must be exactly 16080 bytes (16K minus the size of the page 0 code and the bank's exchange code). If the driver is in more than one bank, this applies to all banks.
- The bank switching code file (only if the mapper to be used by the target hardware is not ASCII16). This code depends on the mapping

type supported by the ROM cartridge where the complete kernel will be burned. Compiled bank switching code files are provided for the ASCII8 and ASCII16 mappers; for other type of mappers, custom code files must be made, following the rules detailed in "Rules for the bank switching code" section. ROM mappers that work with 8K banks instead of 16K banks are supported only if it is possible to select the bank visible at the first half of page 1 (4000H-5FFFh) by writing a single byte in a memory mapped port with a LD(xxxx),A instruction. This is the case of ASCII8, for example.

- Optionally, the code that can be placed in the unused 1K space in banks 0 and 3 (See "The free space in the Kernel's main bank").

The procedure to create the complete Nextor Kernel ROM file basically consists of attaching the driver code to the base Kernel file and then patching the resulting file with the appropriate bank swap code. This can be done manually or using the MKNEXROM utility. Both options are explained below.

Manual creation

To manually create a complete Nextor ROM file, the following steps must be followed (file positions start at zero):

1. Create a copy of the Kernel base file (NEXTOR.BASE.DAT).
2. Append the code from page 0 to the end of the file. This code can be copied from the first 255 bytes of the Kernel base file itself.
3. Append a byte with the value K (the base Kernel file bank count) at the end of the resulting file (this will be the driver bank ID). The value of K can be read from position 254 of the Kernel's own base file.
4. Attach the driver file (which must be exactly 16080 bytes) to the end of the file obtained in step 3.
5. Append the bank switching code at the end of the resulting file. For an ASCII16 mapper the code can be copied from the last 48 bytes of the Kernel's own base file.

6. If the driver code does not fit in a single bank, repeat steps 2 through 5 to add extra banks, increasing the ID for each bank added.

7. If necessary, correct the resulting file to add custom code or data in the 1K free space in banks 0 and 3. Place the contents of the file (up to 1K in length) twice, at positions 3BD0H and FBD0H in the file.

8. If the mapper type is not ASCII16, patch the Kernel common code banks switching code (the last 48 bytes of the first 16K "K" blocks of the resulting file, where "K" is the value obtained in step 3) with custom bank switch code.

9. If the target hardware mapper type is not ASCII16, place the same custom bank switch code used in steps 5 and 8 in the 2012 file position.

10. Only if the ROM mapper uses 8K banks:

    a. Write an LD instruction (xxxxH),A at position F7H of the generated file, where xxxx is the mapped memory port that selects the 8K bank visible in the first half of page 1 (4000H~5FFFH).

    b. Repeat the previous step for all 16K parts of the file. That is, you must write the instruction LD (xxxxH),A at file positions (4000H * n) + F7H, where n goes from zero to the total of 16K banks minus one.

The result is a ready-to-use Nextor ROM file with the device driver already built in.

Using the MKNEXROM Utility

Instead of manually performing all the steps necessary to build a complete Nextor Kernel ROM, it is more convenient to use the MKNEXROM utility. This tool can be used to create a new Kernel Nextor ROM file, also allowing to modify an existing file by changing the mapper code and/or adding extra content in the free 1K areas present in banks 0 and 3.

MKNEXROM comes as a command-line executable file for Windows, but standard C source code is also provided.

The syntax for using the MKNEXROM tool is as follows:

```
MKNEXROM <basefile> <newfile> [/d:<driverfile>]
                      [/m:<mapperfile>] [/e:<extrafile>]
                   [/8:<8K bank selection port address>]
```

<basefile> can be:

- The Kernel Nextor base file, ie the file that contains only the common Kernel code.
- A complete Nextor Kernel ROM file with the driver bank(s) already attached, which must comply with the rules and structure described in section 4. The contents of this file must be as follows:

  1. 256 "dummy" bytes.
  2. The driver signature
  3. The driver jump table
  4. The driver code itself

  And optionally, if the driver generates in more than one 16K bank, for each additional 16K block:

  5. 256 dummy bytes.
  6. The driver code or additional data.
  7. Up to 16K dummy space (not needed for last bank).

  Specifying a driver file is mandatory if a driverless Kernel base file is specified in <basefile>, and prohibited if a full Kernel ROM file is specified.

<mapperfile> is the file that contains the bank's exchange code. If no mapper file is specified, the mapper code from the base file itself will be appended to the driver code.

<extrafile> is the file that contains the code or extra data for the resulting ROM file. This extra data can be up to 1K in size and will be placed at position 0x3BD0H of banks 0 and 3, being visible to applications through calls between standard slots (such as RDLST or CALSLT) and to the Kernel slot at address 0x7BD0H. See "Kernel main bank free space" section for more details.

/8 should only be used if the ROM mapper uses 8K banks. <8K bank select port address> is the address of the memory-mapped port that selects the 8K bank visible in the first half of page 1 (4000H~5FFFH).

As an alternative to using the /8 parameter when using ROM mappers with 8K banks, MKNEXROM can be instructed to properly patch the generated ROM by adding a header to the mapping file itself. This header consists of an FFH byte followed by the bank select port address in little-endian format.

Rules for bank exchange code

If the target hardware mapper type where the resulting ROM will be written is not ASCII16, a file containing the proper mapping code must be provided. This code must follow the following rules:

1. Must be a maximum of 48 bytes.
2. You must change on page 1 the 16K ROM bank whose number is passed in the A register (banks are numbered starting at zero). It is assumed that the ROM slot is already activated on page 1.
3. It can only corrupt the AF register pair. All other registers must be preserved.
4. Must be able to run at any address (cannot contain absolute jumps).

For illustration purposes, this is a valid bank-switch code source code for the ASCII8 mapper:

```
RLCA
LD   (6000H),A
INC  A
LD   (6800H),A
RET
```

If a file with the above code is passed to MKNEXROM as the mapper file to use, it is necessary to add a parameter /8:6000 to the command line so that the generated ROM file includes the appropriate patch for 8K bank ROM mappers. The file can have a header to make the /8 parameter unnecessary. See below:

```
DB    0FFH
DW    6000H
RLCA
LD    (6000h),A
INC   A
LD    (6800h),A
RET
```

Nextor driver structure

This section contains all the details needed to develop a device driver for Nextor. Source code for a "dummy" driver (a valid driver that does not manage devices) is provided and can be used as a reference for developing other drivers.

Drive-based and device-based drivers

When developing a Nextor device driver, the developer must choose between two driver types: drive-based and device-based.

Drive-based drivers have the same quirk as MSXDOS drivers: they locate a set of driver drives and Nextor assigns a fixed drive letter to each drive; any drive required for device or partition mapping must be performed by the driver itself. The set of storage device access routines found by these drivers is the same used by the MSXDOS drivers (DSKIO, DSKCHG, GETDPB, CHOICE, DSKFMT and MTOFF).

Device-based drivers take a different approach. They don't find driver drives, but storage devices directly. In addition, they expose routines for raw access to devices, and it is the Nextor Kernel itself that manages the drive-to-device and partition mapping.

In general, it is recommended to develop device-based drivers, as the implementation routines are easier and the driver code only needs to read and write absolute device sectors without having to worry about partitions; in addition, Nextor's device partitioning tool can be used to create partitions on devices controlled by device-based drivers only. Developing a driver-based driver can, however, be a good option to easily convert an existing MSXDOS driver to Nextor.

Nextor will perform automatic drive-to-device and partition mapping at boot for drives assigned to device-based drivers. This mapping can be modified later using the MAPDRV utility.

Page 0 – Routines and data

This section explains the routines and data that are available on page 0 (addresses 4000H-40FFh) of all Nextor banks, including the driver bank(s).

When creating a Kernel Nextor with embedded driver, the code on page 0 becomes part of all driver banks when the complete Kernel Nextor ROM is generated.

The driver header

The driver header contains some information that helps Nextor to identify it and determine its type. It is placed from 4100H, as described below:

DRV_SIGN (4100H)

Valid driver signature. It is used by the Kernel during startup to verify that the database actually contains a valid driver. It literally consists of the string "NEXTOR_DRIVER" (without the quotes), zero-terminated and capitalized.

DRV_FLAGS (410EH)

Flags byte containing driver information:
bit 0: 0 → drive-based driver;
1 → device-based driver.
bit 1: Reserved, must be zero.
bit 2: 1 if the driver implements the DRV_CONFIG routine.
bits 3~7: Reserved, must be zero.
DRV_CONFIG is used by Nextor from version 2.0.5 onwards.

RESERVED (410Fh)

Reserved byte, must be zero.

DRV_NAME (4110h)

String containing the driver name. It must consist of 32 printable ASCII characters (ASCII codes 32 to 126), left justified and right padded with spaces.

Routines for drivers

This section explains the driver routines that are required by drivers. Jump table entries for routines used by device-based drivers use the same addresses as routines for drive-based drivers, as a driver will implement only one of the two sets of routines.

Device-based drivers do not use the concept of letter-mapped drives; instead, they access storage devices directly, and it is the Nextor Kernel itself that manages the drive to device and partition assignment, both automatically and manually.

Device-based drivers can control up to seven storage devices, numbered 1 through 7. Each in turn has one to seven logical drives, also numbered 1 through 7. Devices cannot be physically divided into logical drives, and the driver must treat the device as having a single logical unit of index 1.

"Block devices" are all devices that can be read and written by accessing logical sectors. This includes floppy disks, hard disks, USB sticks, multimedia cards, etc. Block devices must be readable and optionally writable via the DEV_RW routine.

In the current version, Nextor only works with block devices and sectors with a size of 512 bytes. Support for direct file system access is planned for future releases.

Information about cylinders, heads, and sectors per track applies to the hard disk only; for other types of devices, or when this information is not available, these fields must be returned with a value of zero.

"Read-only" flags should only be set for devices that are physically read-only (eg a CD-ROM). A device that can be protected/write enabled should not be treated as a read-only device.

In the current version of Nextor, it is not possible to format devices controlled by device-based drivers. This is a planned feature for future releases.

The free space in the Kernel's main bank

The Nextor Kernel has 1K unused space in the two main banks (bank 0 when running in normal mode, bank 3 when running in MSX-

DOS 1 mode) that can be filled with any type of data or code useful to the driver.

The main bank is always available in the Kernel slot under normal circumstances; therefore, this area can be accessed through standard slot access mechanisms (such as inter-slot call via CALSLT, inter-slot read via RDLT, etc) even by software that does not know Nextor's mapping mechanism. This space occupies the memory area from 7BD0H to 7FCFH.

There are three cases where you may need to add custom content to this area:

- When a non-null formatting choice string (for devices that cannot be formatted) or the string "Single side / Double side" (in case of legacy MSX floppy disk driver) should be returned by DRV_CHOICE or DRV_FORMAT routines. These two strings are already provided by the Kernel, other strings must be placed in the 1K space.
- When data that is to be read by RDLST or an equivalent mechanism (eg a UNAPI implementation identifier).
- When a hook should be fixed. For this, the hook must be configured to make an interbank call to this area, at the address where the code is. BIOS extension interrupt hooks should not use this mechanism.

The code in this area must use the CALBNK routine if it needs to call routines in the driver bank, whose number can be read from the K_SIZE address.

Any code placed in this area must be identical in banks 0 and 3, for everything to work correctly. The MKNEXROM tool will properly patch both banks if a data file for this area is provided.

Nextor Kernel Routines

Below are all the routines present in the Nextor Kernel. They are described in detail in the appendix, chapter 8, item 8.6.4 – Routines added by Nextor.

GSLOT1 (402DH)	Returns the current driver slot.
RDBANK (403CH)	Reads a byte from any Kernel bank.

CALLB0 (403FH) Temporarily switch the main bank of the Kernel and call the routine whose address is in CODE_ADD (F1D0H).
CALBNK (4042H) Call routine in another Kernel bank.
GWORK (4045H) Get address of 8-byte SLTWRK entry to the past slot or to the current slot on page 1.
K_SIZE (40FEH) Number of Nextor Kernel Banks.
CUR_BANK (40FFH) Current Kernel bank number.
CHGBNK (7FD0H) Makes the specified bank visible on the page 1 of the Z80.
PROMPT (41E8H) Displays the message "Please insert the disk into the X: and press a key when ready", waiting for the requested action.
DRV_SIGN (4100H) Valid driver signature.
DRV_FLAGS (410EH) Flags with information about the driver.
RESERVED (410FH) Reserved, do not use.
DRV_NAME (4110H) String containing the name of the driver.
DRV_TIMI (4130H) Driver interrupt routine input
DRV_VERSION (4133H) Return the driver version.
DRV_INIT (4136H) Driver initialization routine.
DRV_BASSTAT (4139H) BASIC Extended Instruction Handler ("CALLs").
DRV_BASDEV (413CH) BASIC Extended Device Handler.
DRV_EXTBIO (413FH) Extended BIOS handle.
DRV_DIRECT0/1/2/3/4 (4142H, 4145H, 4148H, 414BH, 414EH)

Entries for direct driver calls:

DRV_CONFIG (4151H) Allow driver to provide information about configuration at startup.
RESERVED (4155H to 415FH) Reserved, do not use.
DRV_DSKIO (4160H) Read or write sectors from storage device mass storage associated with a drive unit.
DRV_DSKCHG (4163H) Obtain information about the change state of the media associated with a given drive unit.
DRV_GETDPB (4166H) Get a DPB (Drive Parameter Block) for the media associated with a given driver drive.
DRV_CHOICE (4169H) Returns a format choice string for a disk.

DRV_FORMAT (416CH)	Formats a disk and initializes its boot sector, FAT and root directory.
DRV_MTOFF (416FH)	Stop the motor of all drives.
DEV_RW (4160H)	Read or write absolute sectors to or from a device.
DEV_INFO (4163H)	Return information about a device.
DEV_STATUS (4166H)	Check availability and change the state of a device or logical drive.
LUN_INFO (4169H)	Get information for a logical drive.

7.2 – THE UZIX

Uzix is an operating system developed for MSX that allows multitasking. Uzix, in fact, is a less powerful Unix, being able to run, strictly speaking, any application for Unix as long as, after being compiled, it fits in 32 Kbytes (Uzix 1.0) or 48 Kbytes (Uzix 2.0). This system was created from UZI (Unix Zilog Implementation), a Unix system created for the Z80. Uzix stands for "Unix Zilog Implementation for MSX". Uzix does not need any specific ROM to run; loads the kernel from disk and uses the BDOS direct disk access routines. In fact, in its second version, it has direct access to most hardware devices, including hard drives. This greatly improves system performance.

7.2.1 – Uzix filesystem

Files are organized in Uzix quite differently from MSXDOS. To locate and reference files, they have three areas:

Name

It is the mandatory identification of the file. In Uzix, a filename can contain a maximum of 14 characters, which can be anything (letters, numbers, period, slash, space, equal sign, etc.).

Contents

It is what makes up the file itself (data, text, executable code, etc.)

Other identification data

These are data with the file assignments. These data are access permission bits, number of links, owner and group id, file size,

creation/modification date, etc. They are stored in a structure called 'inode', separately from the filename, and here lies the biggest difference between Uzix and MSXDOS.

7.2.1.1 – File types

Uzix uses 3 main types of files:

1 – Ordinary (common) files;
2 – Files directories;
3 – Special files.

Ordinary files

They make up the majority of system files. They are used to store information (program data, texts, executables, etc.) and are characterized by not having any particular internal format.

Directories files

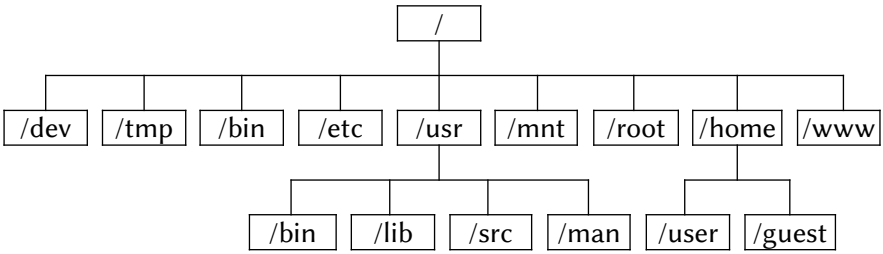
In order to organize files, there are files directories, or simply directories. In MSXDOS, the directory is stored quite differently from other files, but here the directory is just a file. These directories contain a list of filenames, which can be ordinary, special or other file directories.

Special files

They are read and written like regular files, but are used to reference logical system devices, which in turn can activate hardware devices such as printers, terminals, mass memory devices, etc.

7.2.1.2 – Hierarchical structure

In Uzix, we also have the concept of subdirectories as in MSXDOS 2. However, here there is a predefined structure of subdirectories. This structure can be modified by the user, but it is not advisable to do so because it is standard in the Unix world. See the illustration below.



Each of these subdirectories has a specific, but not mandatory, use. The description of each is below.

/ – Root directory.
/dev – Contains the names of special files associated with hardware or software devices.
/tmp – Used system-wide for creating temporary files.
/bin – Contains the most general system applications.
/etc – Files used to administer the system.
/usr – General system files. This subdirectory contains 4 more subdirectories:
  /bin – Generic applications.
  /lib – Libraries.
  /src – Source codes.
  /man – System manuals (text files).
/mnt – Used as a connection point for a file system of another device. Also used for mounting.
/root – System administrator's working directory.
/home – Used by regular users as a desktop.
/user – User "user".
/guest – User "guest".
/www – Internet files.

7.2.1.3 – File access permissions

Uzix uses the same file access permission system used by Unix. Thus, the user can define, for example, who can read their files or make changes to them. This protection can be applied to 3 classes of users:

(u) – Owner user or system administrator.
(g) – Group, or set of users that have some characteristic in common with the owner user.
(o) – Other users of the system.

Access permissions have three levels:

(r) – Read: allows you to list the contents of the file or directory.
(w) – Write: allows you to change the contents of the file or create and rename files and directories.
(x) – Execution: allows executing the file or entering/manipulating files and directories.

The "root" user has unlimited access to the system; an ordinary user cannot prevent the "root" user from accessing their files.

Access permissions are written to inodes and can be listed by the command "ls" or its default alias "dir".

7.2.1.4 – Structure of files on disk

Uzix files are structured on disk in 4 groups: boot sector, superblock, inodes and data blocks.

Boot sector

The boot sector is always sector 0 of the disk and its function is the same as MSXDOS: make the system startup, loading the necessary files. It is executed in the same way as for MSXDOS, loading sector 0 at address C000H and executing the routine in C01EH. Data recorded from C000H to C01DH are not valid for Uzix.

Superblock

It occupies only one sector (sector 1) and contains some information about the disk. This information is as follows:

+0/+1 Signature. It is the first five digits of the PI number stored as an integer (31415).
+2/+3 First logical block of inodes.
+4/+5 Total logic blocks reserved for inodes.
+6/+7 Total logical blocks reserved for files.
+8/+9 Total logical blocks on disk.
+10/+11 Total blocks reserved for the Kernel. The default value for Uzix is 50 blocks.
+12/+111 Pointers to Kernel blocks. 50 2-byte pointers.
+112/+113 Total fs inodes on disk. The default value is 50 inodes.
+114/+115 Total free fs (file system) inodes.
+116/+215 Pointers to inodes fs. 50 2-byte pointers.

+216/+217 Last modified time (MSXDOS format).
+218/+219 Last modified date (MSXDOS format).
+220 File system modification flag.
+221 Filesystem read-only flag.
+222/+223 Used to check for corruption (inode mounting).
+224 Flag modified (mounting the inode).
+225/+226 Device referring to the inode (inode mounting).
+227/+228 Inode number (inode mount).
+229/+230 In-core reference counter (inode mounting).
+231 Filesystem read-only flag.
+222/+295 Copy of disk inode (as per the structure below).
+296/+297 File system device.

The 50 reserved blocks for superblock are not part of fs (filesystem). They contain the Uzix Kernel and are accessed by the secondary bootstrap to boot Uzix from disk.

Inodes

The inodes come right after the superblock. Each inode occupies 64 bytes, and up to 8 inodes can be defined per logical block (each logical block is equivalent to one sector, or 512 bytes). The function of inodes is to store all information about the files (except the name) and to map the file on disk through direct and indirect pointers. They are organized as follows.

+0/+1 Mode flag. Contains all access permissions and the file type.
+2/+3 Total pointers to the file.
+4 File user number.
+5 File access group number.
+6/+9 File size.
+10/+11 Time of last file access.
+12/+13 Date the file was last accessed.
+14/+15 Time of last modification of the file.
+16/+17 Date the file was last modified.
+18/+19 Time of file creation.
+20/+21 Date the file was created.
+22/+57 Direct pointers (18 2-byte pointers).
+58/+59 First-level indirect pointer.
+60/+61 Second-level indirect pointer.
+62/+63 0000H.

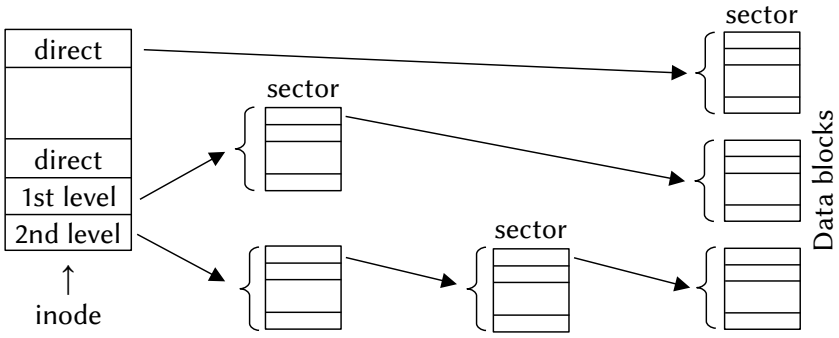
The date and time are stored in the same way as for MSXDOS. Both the user and the group and their passwords are stored in their own file. Passwords are encrypted. There can be up to 256 groups registered in the system with up to 256 users each.

Each pointer is two bytes long and points to a logical block (which is 512 bytes), so the maximum inode-addressable capacity is 32 Mbytes per device (512 * 65 536). As there are only 18 direct pointers in an inode, only files up to 9 Kbytes are directly addressed. Above that, it is necessary to use indirect pointers.

Indirect pointers work like this: the first level indirect pointer points to a logical block that contains pointers to the data blocks. As each pointer is 2 bytes, there can be up to 256 pointers in a logical block. They can map files up to 128 Kbytes (256 * 512). Using first level pointers, we can access files up to 135 Kbytes (128 + 9).

For files larger than 135 Kbytes, second-level indirect pointers are used. The second-level indirect pointer points to a logical block. Each pointer in this block points to another logical block with pointers, now to the file. In this case, files of up to 32 Mbytes (128 Kbytes * 256 pointers) can be mapped, which is the maximum capacity addressable by the inodes.

The pointing scheme is illustrated soon. In the illustration, only one inode (64 bytes) is represented. The blocks it points to correspond to a logical block (a sector of the disk, 512 bytes).



It should be noted that access is getting slower. So, for files up to 9 Kbytes, only one pointer operation is needed. For files larger than 9 Kbytes up to 135 Kbytes, two operations are required. Above that, up to the 32 Mbyte limit, three operations are required.

Directories files

File directories store the names of files or other directories. They are divided into 16-byte blocks. The first two bytes point to the respective inode and the other 14 contain the name of the file itself. There can be up to 32 names in a logical block. Directories files cannot be opened by normal commands.

Mounting

In the MSXDOS FAT system, when loading it into memory, by itself, the FAT already constitutes a map of the disk, with all the information about the free and occupied sectors. In the case of Uzix, this information is spread across the disk, in pointers, and not in a single block as in the case of FAT. In order for the system to know which logical blocks are free and which are occupied, a process called mounting is necessary. In this process, the disk is analyzed and all its space mapped. The resulting map is loaded into RAM.

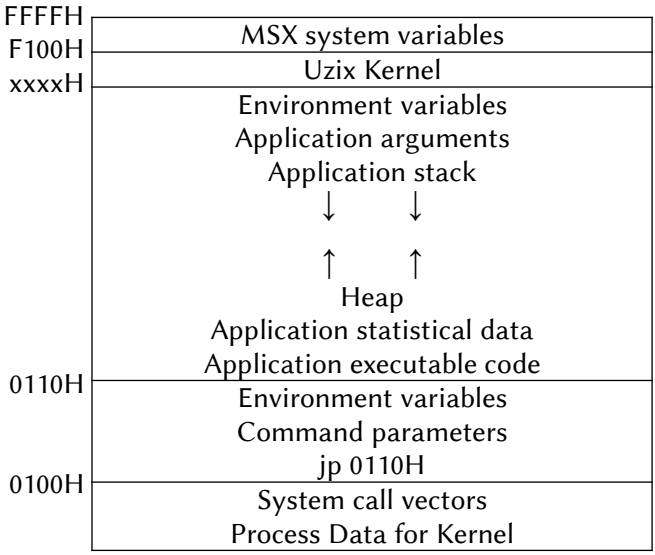
Every device (hardware or software) must undergo the assembly process so that the system can recognize it, although this is processed in very different ways according to the device to be assembled. The primary disk drive is automatically mounted during system startup. Listed below are some standard devices that can be mounted:

fd0~fd7	Floppy disk drives.
null	Null device.
lpr	Printer.
tty/tty0~tty2	Monitor.
con	Keyboard.
mem/kmem	Memory.
sga0~sga(n)	Hard disk partitions.
sge(n)	Hard disk partition where UZIX is.

7.2.2 – Memory mapping

Memory mapping is the biggest difference between Uzix 1.0 and 2.0. In fact, Uzix 2.0 far surpassed 1.0 by requiring the same hardware features.

The memory mapping for Uzix is illustrated below where xxxxH is 8000H for Uzix 1.0 and C000H for Uzix 2.0.



Uzix 1.0 is entirely resident in the high memory area, starting at address 8000H. Every process always occupies 32 Kbytes of memory. Therefore, there can be a maximum of 127 concurrent processes if there are 4 Mbytes of mapped memory.

Uzix 2.0 has more efficient memory management. Although its Kernel takes up more memory than 1.0, only a part of it is resident in high memory, on page 3, starting at address C000H. When necessary, Uzix switches pages to access the rest of the Kernel, executes the function, and returns to the application. Each process can occupy 16 Kbytes, 32 Kbytes or 48 Kbytes, depending on its length. So there can be a maximum of 252 concurrent processes on an MSX with 4 Mbytes of memory. The Uzix 2.0 Kernel occupies 64 Kbytes of memory in total.

7.2.3 – Developing software for Uzix

Applications for Uzix should preferably be developed in C, which is the language in which the system was written. Just a few precautions should be taken.

If the application is to run under Uzix 1.0, the total memory occupied by the application (code, data and stack) must be less than 32 Kbytes. As for Uzix 2.0, the limit is increased to 48 Kbytes. After compilation, it is recommended to look at 'Hbss' address in the map file. If it is very close to the highest address available to the application (7FFFH for Uzix 1.0 and BFFFH for 2.0) it is better to reduce the code size. It turns out that the stack, environment variables, and application arguments are placed on top of memory, and if 'Hbss' is too close to the stack, there can be overlap and system crash. Recommended maximum values for 'Hbss' are 7A00H for Uzix 1.0 and BA00H for Uzix 2.0.

It is also recommended to avoid very large local variables (how char buffer[512]), as the stack will drop too low. It is better to declare them as static. This ends up wasting space in the application, but prevents the stack from eventually overrunning the dynamic data, corrupting it.

If routines are used in machine code, the following should be observed, under penalty of corrupting and even paralyzing the system:

- The Z80 DI and EI instructions should NEVER be used;
- NEVER have direct access to the hardware and
- NEVER access data below 0100H or above the application.

A specific library for software development for Uzix was developed by its author and can be found on the official Uzix page (http://uzix.sf.net). This library is specific to the Hitech-C compiler.

7.2.4 – Shell commands

ADDUSER	Adds a user to the system.
ALIAS	Displays or sets an alias command.
BANNER	Prints a message in large characters.
BASENAME	Removes component orientation from a directory.
BOGOMIPS	Print processing speed in BogoMips.

CAL	Displays a calendar.
CAT	Concatenates files and prints to standard output.
CD	Swap directories.
CDIFF	Prints the difference between two files with context.
CGREP	Search for a string and print the lines wherever they are found.
CHGRP	Changes the group owning user for each file.
CHMOD	Changes file access permissions.
CHOWN	Swaps the common user and the group owning user for the specified file.
CHROOT	Changes the root directory.
CKSUM	Displays the checksum and file size.
CLEAR	Clears the screen.
CMP	Compare files.
CRC	Displays the file data checksum.
CP	Copies files.
CPDIR	Copies directories.
DATE	Displays the current system date and time.
DD	Copy file by converting it.
DF	Displays free disk space in units of 512 bytes.
DHRY	Displays the processing speed in dhrystones.
DIFF	Print the difference between two files
DIRNAME	Prints the suffix of a file name.
DOSDEL	Deletes a file on MSXDOS disks.
DOSDIR	Lists files from an MSXDOS disk.
DOSREAD	Reads a file from an MSXDOS disk.
DOSWRITE	Writes a file to an MSXDOS disk.
DU	Shows the space occupied by directories and subdirectories.
ECHO	Displays a line of text.
ED	Runs a standard text editor.
EXIT	Exits the current session.
FALSE	Does nothing; it simply returns with error state "1".
FGREP	Searches for a string and prints the lines where it is found.
FILE	Makes an assumption about what type the file is.
FLD	Read and concatenate fields from a file
FORTUNE	Randomly prints a proverb.
GREP	Searches for a string and prints the lines where it is found.

HEAD	Print the first lines of the file.
HELP	Prints some commands with their format.
INIT	Process initialization control.
KILL	Kills system processes.
LOGIN	Starts a session.
LN	Add links between files.
LOGOUT	Logs out a session.
LS	Lists the contents of directories.
MAN	Displays the online manual.
MKDIR	Create directories.
MKNOD	Create special files
MORE	Paging Utility.
MOUNT	Mounts the <device> in the specified <path>.
MV	Rename or move files.
PASSWD	Changes the user's password.
PROMPT	Changes the Uzix prompt.
PS	Prints a process status report.
PWD	Print the path of the current working directory.
QUIT	Ends the current session.
REBOOT	Resets the computer.
RM	Remove files.
RMDIR	Remove directories.
SASH	It is a type of shell with built-in commands.
SET	Displays or sets environment variables.
SLEEP	Makes the system "sleep" for <seconds> seconds.
SU	Temporarily connects as superuser or other user.
SOURCE	Displays the "source" of the file.
SUM	Analyzes the checksum and block counter of the file.
SYNC	Flushes the file system buffers.
TAIL	Prints the last lines of a file.
TAR	Concatenates/extracts files for storage.
TEE	Reads from standard input and writes to a file.
TIME	Executes the command and prints the real time, user time and system time (hours–minutes–seconds).
TOP	Lists the most active processes.
TOUCH	Changes the time and date of files.
TR	Swaps characters in a file (transliterates).
TRACE	Trace mode?

TRUE	Does nothing, just returns with error state 0.
UMOUNT	Unmounts the specified device's file system.
UMASK	Removes masks.
UNALIAS	Removes an aliased command.
UNAME	Prints information about the system.
UNIQ	Remove duplicate lines in sorted files.
WC	Prints the number of bytes, words and lines in a file.
WHOAMI	Prints the username associated with the current user ID.
YES	Print "y" or <string> repeatedly to standard output.

All these commands are described in detail in the appendix, chapter 6, section 6.1.2 – Description of Commands.

7.2.5 – System calls

Just as MSXDOS allows direct calls to BDOS/Kernel, UZIX also has its own direct calls, compatible with AT&T Unix Version 7 functionality.

To make a system call it is necessary to stack the parameters in the reverse order of the declaration, then the call number and then executing a CALL 08H. It is the responsibility of the application to pop the parameters after the CALL. The 16-bit return value is placed in the DE register. The only exception is the 'lseek' call, whose return value is 32 bits long and is placed in HL:DE (HL is the most significant word).

These calls are divided into direct, indirect and via GETSET. They are described in detail in the appendix, including the necessary library, chapter 6, section 6.4 – System calls.

7.2.5.1 – Direct calls

ACCESS (#00) – Determines the access level of a file.
ALARM (#01) – Schedules a signal after a specified time.
BRK (#02) – Changes core allocation.
CHDIR (#03) – Change default directory.
CHMOD (#04) – Change attributes of a file
CHOWN (#05) – Switch user and group of a file.
CLOSE (#06) – Closes a file.
GETSET (#07) – Implements calls that read or change values of system variables.

DUP (#08) – Duplicates an open file descriptor.
DUP2 (#09) – Duplicates an open file descriptor.
EXECVE (#10) – Executes a file.
EXIT (#11) – Terminates a process.
FORK (#12) – Spawns a new process.
FSTAT (#13) – Get information about the file.
GETFSYS (#14) – Gets system information.
IOCTL (#15) – Device Control.
KILL (#16) – Sends the 'sig' signal to the process specified in 'r0'.
LINK (#17) – Link to a file.
MKNOD (#18) – Creates a directory or a special file.
MOUNT (#19) – Mounts the filesystem.
OPEN (#20) – Opens a file for reading or writing.
PAUSE (#21) – Pauses the system.
PIPE (#22) – Creates an interprocess pipe.
READ (#23) – Reads from a file.
SBRK (#24) – Changes core allocation (see BRK – #02).
LSEEK (#25) – Moves the read/write pointer.
SIGNAL (#26) – Takes or ignores signals.
STAT (#27) – Get the state of the file
STIME (#28) – Sets system date and time.
SYNC (#29) – Updates the superblock.
TIME (#30) – Gets the date and time.
TIMES (#31) – Returns time information.
UMOUNT (#32) – Unmounts filesystems.
UNLINK (#33) – Removes directory entry.
UTIME (#34) – Sets the time of a file.
WAITPID (#35) – Waits for the process to change state.
WRITE (#36) – Writes to a file
REBOOT (#37) – Reboots the system
SYMLINK (#38) – Creates a new name for a file.
CHROOT (#39) – Changes the root directory.
MOD_REG (#40)
MOD_DEREG (#41)
MOD_CALL (#42)
MOD_SENDREPLY (#43)
MOD_REPLY (#42)

7.2.5.2 – Indirect call

CREAT – Creates a new file

7.2.5.3 – Calls via GETSET

GETPID – Gets the process ID.
GETPPID – Gets the ID of the calling process.
GETUID – Returns the real user identity of the current process.
SETUID – Defines the identity of the user and group.
GETEUID – Returns the effective user ID of the calling process.
GETGID – Returns the actual user ID of the current process.
SETGID – Sets the group ID.
GETEGID – Returns the effective group ID of the calling process.
GETPRIO – Returns the priority of the current process.
SETPRIO – Sets the priority of a process.
UMASK – Defines a file creation mask.
SYSTRACE – Generate and apply protocols for system calls.

7.2.6 – TCP/IP Module

The TCP/IP module implements a subset of IPv4 and allows Uzix to communicate with other systems that support this protocol. The TCP/IP module's signature is 04950H, and it provides 14 functions that are detaliled described in appendix.

7.2.7 – Error codes

Uzix system calls return a value greater than 0 on success and less than 0 on error. The error code is placed in the global variable (defined in the 'stub' of Uzix programs) 'errno'. Below are the existent error codes.

1	EPERM	Operation not permitted
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
7	E2BIG	Arg list too long

8	ENOEXEC	Exec format error
9	EBADF	Bad file number
10	ECHILD	No child processes
11	EAGAIN	Try again
12	ENAMEM	Out of memory
13	EACCES	Permission denied
14	EFAULT	Bad address
15	ENOTBLK	Block device required
16	EBUSY	Device or resource busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device
20	ENOTDIR	Not a directory
21	EISDIR	Is a directory
22	EINVAL	Invalid argument
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
26	ETXTBSY	Text file busy
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek
30	EROFS	Read–only file system
31	EMLINK	Too many links
32	EPIPE	Broken pipe
33	EDOM	Math argument out of domain of func
34	ERANGE	Math result not representable
35	EDEADLK	Resource deadlock would occur
36	ENAMETOOLONG	File name too long
37	ENOLCK	No record locks available
38	EINVFNC	Function not implemented
39	ENOTEMPTY	Directory not empty
40	ELOOP	Too many symbolic links encountered
42	ESHELL	it’s a shell script
	ENOSYS	EINVFNC

7.3 – SYMBOS

The name SymbOS comes from SYmbiosis Multitasking Based Operating System and work in Z80 based systems. It is based on a microkernel, which provides preemptive, priority-driven multitasking and manages up to 1024 KB of RAM. It has a graphical interface (GUI) similar to Microsoft Windows and is available for Amstrad CPC, Amstrad PCW, CPC-TREX, C-ONE, Enterprise 64/128 machines and, of course, for all MSX2 or higher models supporting V9990.

Memory management divides RAM into small blocks of 256 bytes, which can be dynamically assigned. Applications are always running on a 64K secondary bank, where no memory space is taken up by the operating system or video memory, and up to 63K can be reserved simultaneously. Memory management allows SymbOS to access up to 1 Mbyte of RAM.

Communication between the different tasks and the operating system is usually done through messages. This is necessary within a multitasking environment to avoid organization issues with the stack, global variables, and shared system resources. The SymbOS kernel supports synchronous and asynchronous IPC.

7.3.1 – SymStudio System Library

Due to the multitasking environment, most system functions can only be accessed through process messages. To facilitate access to system functions, there are special libraries included in SymStudio. In this documentation for each function you will find a reference to the library, if the function can be accessed by it.

They can be used without SymStudio, but SymStudio offers the advantage of assembling only the parts, making the code shorter.

7.3.2 – System configuration

All user settings in SymbOS are stored in the “SYMBOS.INI” file. This chapter describes the contents of the system configuration and its functions, allowing you to access and manipulate the settings. The user

will usually do this with the control panel. The following information will only be useful for writing your own code or if you need some variables for your application.

The SYMBOS.INI file is divided into 5 parts:

- Header, which contains the identifier and length of the following three parts;
- Part of the core area, which contains data loaded into the first bank of RAM;
- Part of the data area, which contains additional data usually loaded into a secondary database;
- Part of the clipboard (currently empty);
- Source.

There is a System Manager function, which allows you to change the configuration and find out your memory addresses. This function is SYSINF (8103H) and is described in detail in the appendix, chapter 5, section 5.5.4 – System Manager Functions.

7.3.3 – Kernel

The kernel is the heart of SymbOS and controls the main resources of the system. It is divided into "multitasking", "memory", "bank" and "message" modules. It is also the interface between applications and all parts of the operating system.

7.3.3.1 – Kernel Access

The SymbOS Kernel accesses the same RST entries as the BIOS and MSXDOS Kernel, but with different functions. They are as follows:

RST 08H (MSGSLP)	Checks for a new message from another process.
RST 10H (MSGSND)	Sends a message to another process.
RST 18H (MSGGET)	Checks for a new message from another process.
RST 20H (BNKSCL)	Calls a routine, which is placed in the first bank of RAM.
RST 28H (BNKFCL)	Calls a routine, which is placed in the first RAM bank. It is faster than RST 20H (BNKSCL).

RST 30H (MTSOFT)	Frees up CPU time for the system.
RST 38H (MTHARD)	Z80 interrupt handler input, which is called 50 or 60 times per second. It must not be called by the user.

These routines are detailed in the appendix, chapter 5, section 5.1.1 – Kernel Restarts.

Kernel Commands and Responses

(Multitasking Management)

Kernel commands are triggered via a message, which must be sent with RST #10 (MSGSND) to the kernel process. The kernel process always has ID 1. The complete list of commands is detailed in the appendix, chapter 5, item 5.1.2 – Kernel Commands.

Kernel responses come in the form of messages, which must be received with RST 18H (MSGSND) or RST 08H (MSGSLP) from the Kernel process, which always has ID 1. The complete list of messages is detailed in the Appendix, Chapter 5, item 5.1.3 – Kernel Responses.

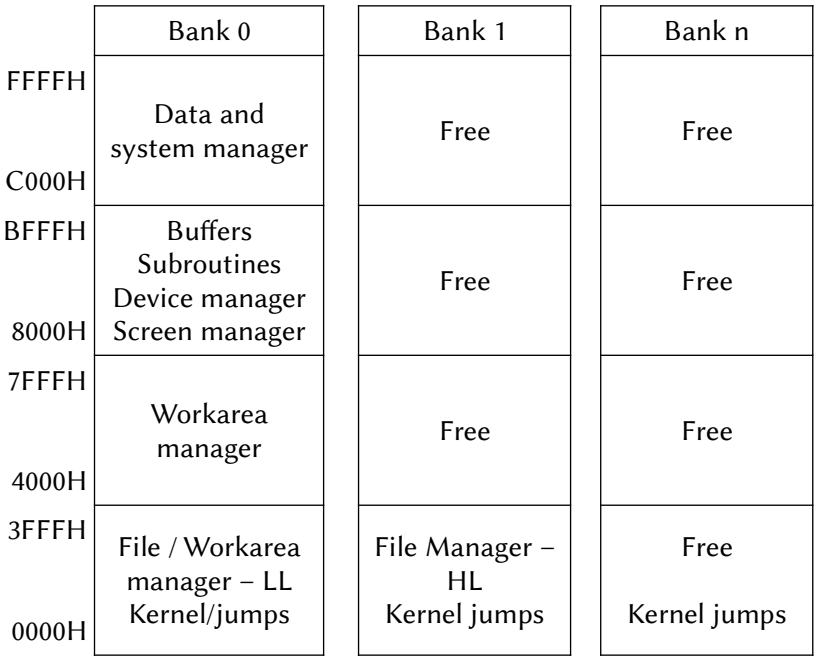
Memory Management / Memory Banks

Most kernel memory and bank functions must be called with RST #20 (BNKSCL) or RST #28 (BNKFCL), depending on the function. Many functions can only be called once at a time, so they are protected with a flag mechanism. The calling process will be switched to idle mode while the function is working for another process. SymbOS can handle up to 16 64K banks, which allows for a total of 1 Mbyte of RAM.

The complete list of memory-related functions is available in the appendix, chapter 5, section 5.1.4 – Kernel Functions (Memory Management)

Memory Map

The following diagram shows how different banks and memory blocks are used in SymbOS.



Application memory map

Memory within an application RAM bank (1 – n) is used as follows:

1 – 0000–03FF Kernel jumps, Kernel multitasking, and bank management routines.
2 – 0400–FFFF Internal application code and data.
3 – 0400–3FFF Application data used by manager of screen.
4000–7FFF An object must be within a 16K block.
8000-BFFF
C000-FFFF
4 – C000-FFFF Application "transfer" data, used by the desktop, message buffer and stack manager.

Memory settings

The following diagram shows how memory is configured during activity of one of the SymbOS modules.

	WorkAreaManager (C1)	ScreenManager (C4-7)	FileManager-HL (C4)
FFFFH	Bank n Block 3 Transfer RAM	Bank 0 Block 3	Bank 0 Block 3
C000H			
BFFFH	Bank 0 Block 2	Bank 0 Block 2 ScreenManager	Bank 0 Block 2
8000H			
7FFFH	Bank 0 Block 1 WorkAreaManager	Bank n Block m Data RAM	Bank 1 Block 0 FileManager-HL
4000H			
3FFFH	Bank 0 Block 0	Bank 0 Block 0	Bank 0 Block 0
0000H			

	FileManager-LL (**)	Application (C2)
FFFFH	Bank 0 Block 3	Bank n Block 3 Trnf, Code, Data
C000H		
BFFFH	Slot x,y Disk-ROM	Bank n Block 2 Code, Data
8000H		
7FFFH	Bank n Block m Data RAM	Banck n Blok 1 Code, Data
4000H		
3FFFH	Bank 0 Block 0 FileManager-LL	Bank n Block 0 Code, Data
0000H		

7.3.4 – Network Daemon

The SymbOS network daemon provides all services for full network access, starting with version 3.0.

The Network Daemon application enables SymbOS for full TCP/IP based networking and internet access, including multiple connections to multiple applications at the same time. Both TCP and UDP protocols within the transport layer are supported, as well as services such as DHCP and DNS. Network Daemon runs as a background service and can be used by any application via its network API.

Supported hardware includes DenYoNet (MSX), GR8NET (MSX) and M4Board (CPC) for access via ethernet or wifi.

A special localhost version of the Network Daemon makes it possible to run all network applications even on machines without additional network hardware.

The description of the network access messages are described in detail in the appendix, chapter 5, section 5.12 – Network Daemon.

7.3.5 – Screen Manager

Currently, the screen manager only contains one function for direct access to the video. This function can also be used by applications. It must be called via RST #20 (BNKSCL).

TXTLEN (815DH) – Screen_TextLength

Description:	Returns the width and height of a textline in pixels, if it would be printed to the screen. You can define the text length (number of chars) in IY. If the text is terminated by 0 or 13 you should use –1 for the maximal text length. Please note, that this function always uses the system font for calculating he width and height.
How to call:	rst 20H : dw 815DH.
Input:	HL – Text address. A – Text RAM bank (1~15). IY – Maximal number of chars (text length).
Output:	DE – Text width in pixels. A – Text height in pixels.
Registers:	F, BC, HL, IX.

7.3.6 – Screensaver

SymbOS desktop manager supports handling screen saver applications. These are special programs, which are automatically loaded during startup or when entering sleep mode. If the system is idle for a period of time (no mouse and keyboard activity), the desktop manager sends a message to the screen saver application to run it.

This chapter describes the implementation of a screen saver application and the messages used to communicate between the system and the screen saver.

A screen saver application can run in three different ways:

- Operating mode: The desktop manager loads and initializes the screen saver. The system remains suspended until the desktop manager sends a "start" message.
- Configuration Mode: The control panel temporarily loads the screen saver and sends "start" messages for testing or "config" messages for configuration purposes.
- Demo mode: The screensaver is manually loaded by the user. In this case it starts immediately. When a key is pressed or the mouse is moved, the screen saver ends.

To fulfill these requests, the screen saver must perform the following sequence of operations:

0 – The screen saver application starts;

1 – Wait for the initial message;

2 – If the message cannot be received:
- Start the animation phase;
- Wait for the key or mouse;
- Go to 5.

3 – Check the received message:

a – If the "start" command message (see MSC_SAV_INIT) was received
- Receive configuration data;

- If the data is not valid, use the default configuration data;
- Launch the screensaver.

b – If the "config" command message is received (see MSC_SAV_CONFIG):

- Open the configuration window;
- Let the user make changes to the configuration;
- Send "config" response message (see MSR_SAV_CONFIG) with updated configuration data to the process that sent the MSC_SAV_INIT.

c – If the "start" command message (see MSC_SAV_START) is received:

- Start the animation phase;
- Wait for the keyboard or mouse to respond.

d – If the message "quit" (0) was received:

- Go to 5.

4 – Wait for the next message while in sleep mode, then go to 3.

5 – Exit the application.

The commands and responses for accessing the screen saver are described in detail in the appendix, chapter 5, item 5.9 – Screen Saver Applications.

7.3.7 – Text Terminal (SymShell)

The SymShell system application provides a text-based user interface program environment. Applications, which are working within the SymShell environment, can use terminal input and output routines to send and receive text data to and from the standard console. It can also be redirected to other origins and destinations.

SymShell commands are triggered via messages, which must be sent with RST#10 (MSGSND) to the SymShell process. SymShell will pass its process ID and text screen resolution to the application via the command line.

A command line has the following format:

```
[drive:/path/filename.com] (parâmetros) %spPPXXYYVV
```

The 256-byte string is always placed right after the last byte of an application's code area. The user parameter list must be enclosed in parentheses. Soon after, it is necessary to put an additional parameter that must be read by the application. It starts with "%sp" and has the following structure (all numbers must be in ASCII format and represent 2 decimal digits for each value):

PP – SymShell process number. The application must know the process number, to which it has to send the text input and output commands.
XX – Width of the text terminal window in characters.
YY – Height of the text terminal window in characters. These two pieces of information can be used for reformatting text output.
VV – SymShell version; the first digit is the major version, the second is the minor version

Example:

```
c:\symbos\symshell\quizgame %level:9 %sp07602021
```

The "quizgame.com" application will run in the path "c:\symbos\symshell\". The user passed the parameter "%level:9" to it. SymShell version is 2.1, it runs with process ID 7 and the text terminal window has a size of 60x20 characters.

It is important to note that if there is no '%sp' parameter, the application is likely not started inside SymShell and must exit by itself as it cannot use the text terminal features.

The commands supported by SymShell are as follows:

ATTRIB Displays current file attributes.
CD Displays or changes the current subdirectory.
CLS Clears the screen.
COPY Copies files. Accepts the wildcard character '*'.
DATE Displays or changes the system date.
DEL Deletes one or more files. Accepts the wildcard character '*'.
DIR Displays the names of files on the disk.

HELP	Displays the help file.
MKDIR	Creates a subdirectory.
MOVE	Moves files to another part of the disk.
RMDIR	Removes an empty subdirectory.
REN	Renames file <old name> to <new name>.
TIME	Displays or changes the system time.

Default apps:

DIMON.COM	Command line disk monitor.
NETSTAT.COM	Displays active network connections with status.
TELNET.COM	Telnet client.
UNZIP.COM	Extract .ZIP or .GZ files.
WGET.COM	Application for downloading files via HTTP.

All commands are described in detail in the appendix, chapter 5, section 5.7.1 – Text terminal commands.

7.3.8 – System Manager

The system manager is responsible for starting and stopping applications and performing general system work. It provides various dialog services and owns the file manager, which can only be accessed through the system manager process.

System manager commands are triggered via a message, which must be sent with RST #10 (MSGSND) to the system manager process which always has ID 3.

All its commands are described in detail in chapter 5, section 5.5 – System Manager.

7.3.9 – Applications

In general, applications are executable programs that can be launched in the SymbOS multitasking environment, which can be of two types.

The first type are GUI-based applications with the extension "*.EXE" and use the desktop manager as the user interface. The second types are text terminal-based applications (*.COM), which require SymShell as a user interface.

The structure of both types is identical, the only difference being that the application shell additionally can use the SymShell terminal for text input and output of the result.

7.3.9.1 – Types of memory areas

Before starting with the description of the application structure, it is necessary to know the three types of memory that exist in SymbOS.

Since the Z80 CPU's address bus is 16 bits long, the CPU cannot address more than 64K (65 536 bytes) at the same time. To be able to work with more than 64K it is necessary to use 'banks' which means that parts of the memory will be visible within the 64K address space of the Z80s, while other parts will be hidden. This is also known as memory mapping.

Initially SymbOS was developed for Amstrad CPC computers. The CPC-6128 already had very powerful bank switching possibilities compared to other 80's 8-bit computers. However, the CPC bank switching methods still have some limitations. While the possibilities of other machines like MSX2, Amstrad PCW or Enterprise 128 allow very flexible 4x16 Kbyte banking services, CPC is limited to some specialized banking configurations. SymbOS is based on these limitations. This makes it possible to run the SymbOS microkernel on many platforms like CPC, MSX2/+/TurboR, PCW, Enterprise and even Sam Coupe.

RAM is divided into 64K chunks, which are called “banks”. Each bank is divided into four 16K pieces called “blocks”. CPC is limited to four different bank switching modes. Three of them are used by SymbOS, which can also be configured on all other supported platforms.

SymbOS consists of several modules called “managers”. Most parts of the operating system are placed in the first 64K bank. Applications are always placed inside a secondary 64K RAM bank, which will be activated while the application is running.

In order to access application memory, the system must change parts of its RAM to the visible 64K. As the possibilities for switching from the original bank are limited, not all managers can access all parts of the application's memory:

a.) Code area: The code area can be placed anywhere in memory. Because of this flexibility, it is easier for the operating system to allocate memory of this type. Everything that doesn't need to be in other areas should be placed here.

The kernel is able to access all parts of memory. The same goes for the system manager, which only uses the kernel to access application memory. This means that all the functions and data of an application, which are in contact with the kernel or the system manager, can be placed anywhere in memory.

This type of memory is called a “code area”, but it can contain any type of data. In general, an application's code area contains all of its routines and additional data, which do not need to be placed in one of the other two memory areas.

b.) Data Area: The data area needs to be placed within a 16K block. All text and graphics, which are plotted on the screen, must be placed here.

For performance reasons, a single line of text or a single graphic must be placed within a 16K block, so that the screen manager can access it in one piece. This area contains all application data that is accessed by the screen manager.

The screen manager is also able to access all parts of memory, but only one 16K block at a time. Its function is to draw texts, graphics, lines and boxes on the screen. It does all the low-level work for the desktop manager.

c.) Clipboard: The clipboard needs to be placed in the last 16K block (#C000–#FFFF), since, because of its location in memory and the limitations of the original bank, the “Clipboard manager work” can only access the last 16K of each RAM bank. All application data that needs to be accessed directly by the manager must be placed in this area. These are the window data and control data registers. Also the stack and message buffer must be placed in this area, as the RST #28 (BNKFCL) function can only be used with such stack position. This type of memory is called a “clipboard” as it is the only part whose data can be transferred to the desktop manager and kernel message module with quick use of bank switching.

Note that the application itself doesn't have to worry about the locations of memory areas. This is done by the system manager, when it allocates memory during application loading, and by the kernel, when the application wants to allocate additional memory of one of three types.

7.3.9.2 – Applications structure

Each application is subdivided into four parts. The first three are the main ones according to the three different memory types. The fourth contains the reallocation table and will not be kept in memory after the boot process.

The code area part of the application always starts with a 256-byte header, which contains all the necessary information that the system manager needs to know to load and initialize the application. The header data structure is described below.

After the application starts, the system manager writes some useful information in the header like the RAM bank number and the process ID. The clipboard part of an application must always start with the stack, the length of which needs to be written in the application header. This allows the manager to tell kernel where to start the main application process.

The three parts of an application must be placed immediately after each other. The system manager will break them down and place each one in appropriate areas of memory.

The fourth part contains the relocation table, which is a list of pointers (words) to the reference addresses to allow application code to be relocated. The table is temporary, being removed after the relocation process.

Application header

The application header structure is illustrated below.

000 02 Length of code area.
002 02 Length of the data area.
004 02 Length of clipboard.
006 02 Original source of the assembler code.
008 02 Number of entries in the reallocator table.

010	02	Stack length in bytes.
012	02	*Reserved* (must be 0).
014	01	*Reserved* (must be 0)
015	24	Application name. The string must end with 0.
049	01	'0' (terminator).
040	01	Flags (+1=16 color icon included).
041	02	16 icon file offset colors.
043	05	*Reserved* (must be 0).
048	08	"SymExe10" (SymbOS executable file identification)
056	02	Size in bytes of the reserved (additional) code area.
058	02	Size in bytes of the reserved (additional) data area.
060	02	Size in bytes of the reserved (additional) transfer area.
062	26	*Reserved* (must be 0).
088	02	Minimum operating system version (minor/major).
090	19	Application icon (small version), 8x8 pixels, SymbOS graphic format
109	147	Application icon (large version), 24x24 pixel, SymbOS graphic format

Words 0, 2 and 4 contain the length of the static memory parts. If more memory is needed in the RAM bank where the application is running, it must be allocated during the load process. Instead of adding buffer to the program code itself, which would waste memory on disk, additional code, data, and transfer areas can be allocated with words 56, 58, and 60.

Pointers should not be used to access the additional (extended) area as the reallocator would think you want to access memory in another area as the pointer is outside the static part. It is necessary to first calculate the address within the code.

After the application is launched, some parts of the header are replaced with new data as below.

006	02	Address of the data area.
008	02	Address of the clipboard.
010	04	IDs of additional subprocesses or timers; 4 process/timer Ids can be registered here.
014	01	RAM bank number (1~15) where the application is located

048 40 Additional memory areas; up to 8 areas of 5 bytes can be registered here, with the structure described below:
00 01 RAM Bank number (1~15; if 0, the entry will be ignored).
01 02 Address.
03 02 Length.
088 01 Application ID.
089 01 Main process ID.

The application process ID is needed for sending messages to other processes and for when the application needs to terminate itself. The RAM bank number is required for all types of system calls where a memory pointer is required. Bytes 10 and 48 should be used to record additional allocated memory areas, subprocess IDs, and timers.

Path and attached parameters

When the system manager loads an application, it appends 256 bytes just after the end of the code area, containing the full command line, with which the application was started. The appended string looks similar to the following:

```
B:\symbos\apps\symsee.exe A:\pics\mycat.scr -full
```

So the application can know where it is and load its configuration file from the same location. It can also read any parameters added to the command.

CPU register usage

Every process can use only the primary register set (AF,BC,DE,HL,IX,IY). The secondary register set is used by the multitasking manager, and the I register is used to store the current bank configuration. Such registers must not be used by the application.

7.3.10 – Pulldown menus

Up to 8 submenu levels can be set. The “Window Data Record” points to the highest menu level. These are the entries you see in a window's menu bar. These entries usually point to their submenus, which also contain clickable entries or that point to an additional new submenu.

7.3.11 – Graphics

7.3.11.1 – Standard graphics

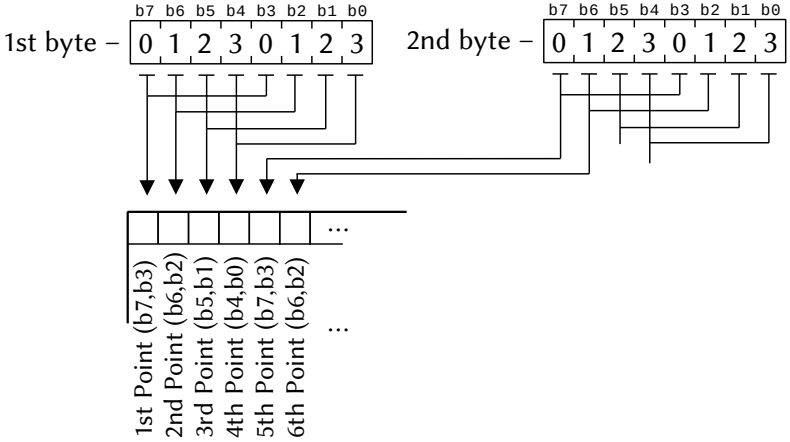
A standard SymbOS graphic has 4 colors with a maximum size of 255x255 dots. Each graphic object starts with a 3-byte header:

00 1B [bit0~6] Width of the graph in bytes.
[bit7] Encoding type (0=CPC, 1=MSX).
01 1B Width of the graph in points.
02 1B Height of the graph in points.

Right after the header, the amount of [width_in_bytes] * [height_in_points] bytes contains the graphic data. Each graphic is stored line by line as a sprite.

Points must be encoded in CPC (Mode 1) format. Graphics on an MSX system will automatically be converted to MSX format when they are first displayed. Bit 7 of byte 0 of the header contains the current encoding format. Note that it is not allowed to store an original graphic in MSX format, as the CPC system is not able to handle these graphics.

The following is a description of the CPC encoding format. Each byte contains 4 dots:



Only applications, which need to modify a graphic after it has been displayed for the first time, should be concerned with the encoding type and the MSX format.

7.3.11.2 – Extended header graphics

Since the width of a graph is limited to 255 points, it would not be possible to store a full screen (such as 320 x 200 in CPC 1 mode) in a single tile. The screen needs to be split into two parts (eg 2 x 160 x 200), which makes it very difficult to write graphical modification routines.

Extended graphics do not have this limitation and also allow for more than 4 colors. They can only be used for control ID 10, "graphic_extended". A graphic can be stored in a single piece with a width of up to 1020 points. The "graphic_extended" control is then able to display a portion of a large graphic linear storage.

The extended header structure is as follows:

00	1B	Width of the complete graph in bytes (must be an even value).
01	1B	Width of the graphic area, which must be displayed, in pixels.
02	1B	Height of the graphic area, which should be displayed, in pixels.
03	1W	Graphic data address, including area offset.
05	1W	Address of the encoding information byte (this byte must ALWAYS be placed immediately before the graphics data).
07	1W	Full chart size.
??	1B	Encoding information: bit0–1: Color encoding (0 ⟶ CPC, 1 ⟶ MSX). bit2–3: Color depth (0 ⟶ 4 colors, 1 ⟶ 16 colors). Only the following two initial values are allowed: 0 ⟶ 4 colors, CPC format; an MSX system will convert the graphic to MSX format when it is first displayed. 5 ⟶ 16 colors, MSX format; a CPC and PCW system will render the full graphic to 4 colors (CPC format) when it is first displayed.
??+1	x	Graphics data.

The chart header does not need to be stored immediately before the chart data, it just needs to be located in the same 16K area as the chart. This chart type can be used:

• If the graphic is larger than 255 pixels;
• If you are going to display only a part of the graph;
• If the header cannot be stored immediately before the chart data;
• If 16-color graphics are used.

In any other case, standard graphics should be used, as they are faster. An example for a 320x200 chart is illustrated below:

```
encoding_type: db 0 ;= 4 color CPC format
            ;** must be placed directly **
            ;** in front of graphic data **
graphic_data: db x,x,x,x,x,x,...x
            ; ↑ line 1, includes 80 bytes = 320 pixels
      db x,x,x,x,x,x,...x
            ; ↑ line 2
                [...]
      db x,x,x,x,x,x,...x ;line 200

graphic_header_for_area_1:
            db 80
            ;80 * 4 = total of 320 pixels
            db 160
            ;this area has only 160 pixels wide
      db 200 ;heigth is 200 pixels
      dw graphic_data
      dw encoding_type
      dw 80*200

graphic_header_for_area_2:
      db 80 ;the same above...
      db 160
      db 200
      dw graphic_data+40      ;...but this area begin
                              ;in offset byte 40
      dw encoding_type
      dw 80*200
```

The graphic itself ("graphic_data") is stored in a single block in memory (no header). So we have two headers (“graphic_header_ for_area_1” and “graphic_header_for_area_2”) that are pointing to two different areas of the complete graph. As can be seen, two controls are still needed to display the chart, but the data itself does not need to be split.

7.3.12 – Sources

A font defines the appearance of characters used to print text in SymbOS. A font starts with a simple 2-byte header, followed by the characters' bitmask. Its format is as follows:

```
00   1B  Height of each character in points (1~15, default: 8).
00   1B  First character of the font (0~255).
00   1B  Width of the first character in points.
01   1B  1st line bitmask of first character.
02   1B  2nd line bitmask of the first character.
 ⋮
15   1B  Bitmask of the 15th line of the first character.
16   1B  Width of second character in points.
17   1B  1st line bitmask of second character.
 ⋮
```

7.3.13 – Device manager

The device manager controls all parts of the hardware. In general, applications do not communicate directly with the device manager, as other parts of the operating system are between the device manager and the application, such as the system manager and the desktop manager.

So there are only a few device manager functions, which can be called and used directly. All device manager functions are described in detail in the appendix, chapter 5, section 5.6.6 – Device Manager Functions.

7.3.14 – File manager

The file manager is controlled by the system manager, which is the only one allowed to call the file manager functions. If an application wants to use the file manager, it needs to send a message to the system manager process, which will call the file manager function and return a message with the result to the calling application.

The system manager process always has ID 3.

Note that in SymbOS all texts must end with byte 0. Thus, the paths and filenames used in the file manager must also end in the same way.

7.3.15 – Directory manager

Directory management functions allow you to display and edit the contents of a directory. Many functions require a path with or without a filename. These paths must have a maximum size of 255 bytes. They are built like this:

```
[Drive:][\][dir1\][dir2\][...\][filename or filemask]
   A     B   C               D
```

[A] If the unit is specified, the active device is not used, but a specified one. If the device is not included in the path, the system searches the current device.

[B] If the path itself starts with a "\", the system starts working from the root directory. If there is no "\" at the beginning, the system starts searching the current directory.

[C] You can enter as many subdirectories as you like. You can also change to the parent directory ("..") or the current one ("."). In the following example sequence, you will remain in your current directory:

```
new_path = "subdir\.\..\subdir\..\"
```

[D] If you want to specify a directory, the path must end with a \. If you want to specify a filename or a file mask, you append it behind the last \.

A file mask can contain ? (any character at this location) and * (any character up to the end of the current filename part).

Instead of the backslash ("\", Microsoft), the forward slash ("/", Unix, Linux) can also be used.

Note that the drive and path must always be specified as the application is running in a multitasking environment, and other applications may change the default drive and path. Due to limited memory re-

sources, SymbOS does not support additional drive/path instances for each process.

7.4 – WiOS

WiOS is a graphical multitasking operating system for MSX computers that use the V9990 VDP. WiOS is a collection of routines that support the programming of applications in a graphical environment. It allows a shared number of up to 252 drivers or tasks (which is a synonym for "application"), manages up to 3072 memory segments and up to 256 windows.

In order to work, WiOS needs:

- Mouse to move the cursor.
- V9990-based graphics card on port 60H.
- At least 160k of free memory.
- MSX-DOS 2 or higher.

However, for comfortable performance, an MSX turbo R with 512 Kbytes of memory and a hard disk should be used.

7.4.1 – Applications for WiOS

The ASCII-C v1.2 language is advisable, due to the fact that the function calls must have the same structure. See the READ.ME file in the "Application Creation Toolkit" for a complete packing list

There is no need to have ASCII-C in the same directory as WiOS. To find the compiler files, you can specify the directory in the PATH environment or put the full drive and path before the '.COM' files in the build batch file.

To find standard C libraries outside the current directory, include them with "#include <headfile.h>" and set the INCLUDE environment variable to the drive and directory of the header files.

If you've never worked with WiOS (and certainly never worked with WiOS) it's important to read this section thoroughly to better understand the basic structure of how WiOS works.

7.4.1.1 – Program Structure

Application programming for WiOS is different from the usual way of structuring programs. The main difference is the multitasking environment. There are two main types:

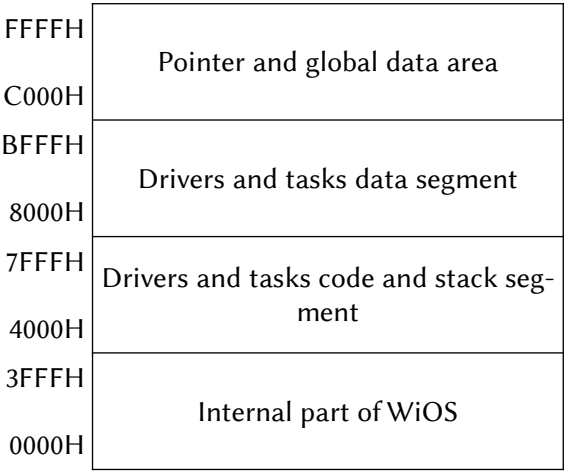
Preemptive multitasking – Each application is given a certain amount of time. Control is given and taken away from the system. The big problem is preserving the state of what an application did when control was taken.

Cooperative Multitasking – With this type, the speed of the system depends on the cooperation of the tasks. Each task has full control of the execution time. To provide the possibility for other tasks to be able to do something in parallel, each task must give up control at some point in the execution.

WiOS uses cooperative multitasking as it is easier, faster, and more memory-saving.

7.4.2 – Memory Structure

To run multiple programs at the same time, more than 64K of memory must be accessible. Also, there should be a fixed area that is always present to ensure that some functions, like polling, can be called from anywhere. The WiOS memory structure is as follows:



7.4.2.1 – Global Data Area (GDA)

Here is the interrupt handler and the fixed table of entry points. Whenever the inside of the WiOS is changed, the correct addresses will be placed in the GDA. This indirect addressing allows other programs to access the functions even if their position changes. The list of pointers can be expanded in the future, but existing entries cannot be changed. WiOS also uses this area for its own internal stack.

7.4.2.2 – Data Segment

This is where the task can freely switch memory segments to store data. It is not advisable to put code here as the system behavior will be unpredictable

7.4.2.3 – Code and Stack Segment

Each task has 16K for code. For speed, the stack pointer is also kept in this segment. Therefore, whenever the polling routine has to switch to another thread to return the next task, the respective stack is already there; only the stack position should be corrected.

7.4.2.4 – Internal part of WiOS

Here are the functions that should be accessible from any point and that handle the switching in the code and data segment. The addresses of functions and variables may vary in different versions of WiOS.

7.4.3 – Drivers

A 16K page is small, so only the necessary parts are placed on page 0, such as the lookup routine, global variables, memory management, and tasks. Thus, WiOS is modular, dividing functions into internal, which are accessed directly, and external, which are accessed indirectly through internal functions.

External functions are mapped on page 1. As a result, WiOS switches 16K segments to access them. This makes possible an almost unlimited number of external functions, which are called drivers.

Difference between tasks and drivers

Drivers have two main differences from tasks. They cannot be polled, so routines in tasks block any other activity as soon as they are executed and cannot be present on the windowed screen (which follows from the fact that they cannot be polled).

But drivers can call other drivers (like subroutines). External functions are called using numbers, which must be defined in the driver itself. Each driver has a fixed entry point where the internal function caller sends the task or driver arguments. As each driver can use the same function numbers as other drivers, they are separated by handles.

7.4.3.1 – Handles

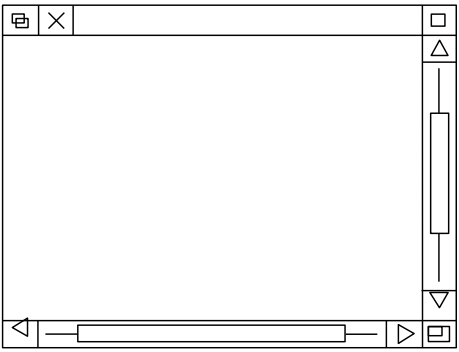
The WiOS memory manager needs to save the user of a thread to prevent tasks/drivers from freeing up threads from other tasks/drivers. To ensure this, each task must have its own identification code. This identifier is called a handle like file identifiers in DOS 2. Whenever a task is loaded, WiOS looks for the next free handle and applies it to the task.

Task handles are searched from start to finish, driver handles vice versa. The maximum number of handles is 252 and the total amount of possible tasks and drivers depends on how many tasks or drivers are already installed. WiOS has some default drivers whose handles are stored in the GDA.

Each task/driver has its own identifier, independent of the identifier assigned by WiOS. This ID is a 0-terminated string, whose length and content can be freely chosen by the programmer. A task or driver identifier can be searched for by its name. A driver that has no entry in the GDA can be searched for by name, and WiOS returns the identifier or a code. The handle can then be stored in a variable and the new driver can be accessed like all the others. The windows, default icons and fonts are also accessed through these handles.

7.4.4 – Window Interface

Windows are almost entirely controlled by WiOS. So applications don't have to worry about user activities outside the desktop. A WiOS window looks like the illustration below.



The symbols that are part of the window are called window icons. There are 7 icons and each one can be set separately for each window using 'icon-flags'. Starting in the upper left corner clockwise are as follows:

- Back icon
- Close icon
- Title bar
- Switch window size
- Vertical scroll bar
- Resize
- Horizontal scroll bar

7.4.4.1 – Parameters

To create a window, you need to configure a parameter heap with icon flags and the following data:

- Absolute coordinate of the upper left corner of the screen;
- Size of the visible work area (from now on called 'work-area size');
- Total desktop size (virtual size);
- Scroll offsets;
- Minimum desktop size (ie how small the window can be made by the user);
- Maximum size of the work area;
- Window area flags;
- Icon flags;
- Desktop flags.

The workspace is where the task displays its own information. The desktop size is never changed by WiOS. For example, adding a title bar will not take up your desktop space, but will increase the overall height of the window. WiOS does not check that icons fit in the window. If the window is too small, some icons overlap others.

WiOS checks the window icons but the task has full control over whether or not to accept what the user does. WiOS sends the parameter heap to the task before doing any action on the screen, and it is up to the task to execute the command to redraw the new window.

7.4.4.2 – Window Management

WiOS only contains a list of parameters for each window in memory. Their identification is similar to that of tasks and drivers, but they only have handles, without names. Windows and their content are not stored as a bitmap, but only as screen graphics.

Since parts of a window can be hidden by other windows, it is not easy to draw only the visible parts. When WiOS sends a list of fragments on the screen, you can redraw (this feature has not been tested) the entire window at the coordinates that WiOS offers (on a hidden page), allowing WiOS to copy parts of this window to the screen.

It should be noted that WiOS is optimized for functionality, not speed or efficiency.

7.4.4.3 – Types of Windows

Parent windows

They are conventional windows. If the task opens a single window, it will receive the "parent" status.

Pane windows

These are windows that are linked to a parent window. They are always drawn directly over the parent window. If the parent window goes to the back, all the pane windows will too. They will be closed if the parent window is closed.

7.4.5 – Events (Communication between WiOS and Tasks)

Imagine the following situation: The task does a very complex calculation that takes a few minutes. Now the user closes a window that is in front of the calculation task window. Although the task is busy, the system should be able to redraw its window on demand.

Thus, WiOS has an event handler that sends information to the task right after polling. A number called 'Event' is returned to the task. It has to deal with that, so each task must call its own event handler and act on the event before proceeding with its calculation.

To ensure an immediate screen refresh after a window move operation by the user, the events have a different priority. This means that the task that needs to redraw a window must be called before its turn in the normal sequence and other tasks that are just waiting to be returned without the need to redraw.

This requires differentiating between tasks that must be called to refresh the screen and other tasks. In WiOS there are three event priorities. They are (highest priority first):

1 – Open;
Redraw;
Scroll;
Close;
User message.

2 – Mouseclick;
Pointer has left window;
Pointer enters window;
Pointer is over work-area;
End of a drag-operation.

3 – Null event.

The event with the lowest priority (null event) is only sent if there is nothing else to do and signals that the task can continue its work. Tasks that receive a high-priority event must poll again immediately after this event has been handled, and wait for the null event to continue their work.

Not every task needs to do complex calculations all the time. If a task just waits for some user action, it doesn't need to be called all the time if nothing happens. That way, there is the feature of masking events.

The task can mask events so it doesn't get called all the time. The simplest case is the 'null-event'. If a task waits for something, it can tell the polling routine to activate it only when a mouse click is made or a window needs to be redrawn.

7.4.6 – User Input

The system checks the keys and the mouse all the time through a routine connected to the interrupt. Therefore, the keys pressed while the interrupt is disabled are not placed in the buffer, as well as the movement and clicks of the mouse, since the movement of the mouse is done via the interrupt.

Thus, the mouse click is read during a direct mouse scan. To know if the mouse was clicked, WiOS stores a timer in the buffer when a click is made. The task should use this timer to know if the mouse was released and pressed again during polling. If the mouse is still pressed and the timer in the buffer is the same as before polling, the mouse is on hold. If the timer is different, the mouse was pressed again.

For the keyboard, WiOS uses a buffer with multiple pointers for each task. Whenever a task calls the function to empty or read the keyboard buffer, only its pointer is reset or incremented, so one task can clear its buffer while the next task will get the last 100 key presses. The buffer is 256 bytes long and is updated polling-time and whenever the WiOS 'getkey' function is called. If a task starts reading the buffer after about 300 keystrokes, it only receives the last 300 – 256 = 44 keys.

If a task reads the keys directly via the 'getkey' call, only that task will get their values. If a task waits for a keystroke, empty the buffer and loop with the 'getkey' routine.

7.4.7 – Drag and Drop

The WiOS multitasking framework allows tasks to call routines from other tasks. This makes it unnecessary for programmers to code common routines over and over again. A good example is the disk menu.

There is only one task that does all the file handling like deleting, moving, copying and even opening files. This menu is a standard WiOS basic task.

Normally, all interactions between tasks are done by events. There is an event called 'End-Of-Drag-Operation'. If the user drags files from the file manager to a task window or its icon in the toolbar, the task will receive this event with the id that there are filenames arriving and where they are stored. Thus, WiOS tasks can be written without the disk menu.

7.4.8 – File Management

A weak point of WiOS is that no checks are done when they are opened in multiple instances of a task or in different tasks. Thus, if changes are made to a file, it must be saved by the application. However, disk errors are designed to be handled by WiOS itself.

7.4.9 – Programming Language

It is strongly recommended that apps for WiOS be written specifically in ASCII-C 1.2, not BDS, Hi-Tech or any other C.

C is strictly limited to 16-bit address space, with no built-in memory mapping. The last action to prevent programs from getting too large is the linking process. There is L80-2 version, but use version 1 to link C programs: your program may be bigger than version 2.

Every COM file is placed from 0100H and executed from that same address. However, to create WiOS executables, the linker must be informed that the task must start at 4000H. This can be done with the '/P:4000' option as the first argument on the L80 command line. You will get a file with about 16K of zeros at its beginning. This file must be left AS IS. With the 'Application-Creation-Toolkit' comes the MAKDRV.COM utility, with which it is possible to make real large applications. It eliminates the leading 16K of zeros and fills in some important information in the file header.

It is essential, however, that large applications must be divided into blocks of up to 16K each, which can make their development difficult.

As for the linker, all external functions you want to call from a task must be linked in the executable. It is not possible to link full WiOS to your task as WiOS itself is larger than 64KB. What needs to be linked is the GDA file that contains the global addresses that allow tasks to access WiOS functions and variables. This file only contains labels that point to a specific address table on page 3 (above C000H).

Always include the 'GDA.H' file if you want access to WiOS.

7.4.10 – Calling WiOS functions

In C, functions are normally called using a syntax like function(); where the code generated for this function always calls the 'function' address directly.

For calling at an address where a variable points, the syntax is a little different. As long as there is indirect addressing, the label will not point to the function itself, but to an address in the GDA that contains the function's address. This label is named with a leading underscore: '_function'.

In C, the contents of a pointer are accessed with an asterisk before the label name. To call the address where a pointer points, the label and asterisk must be enclosed in square brackets.

To call the address where the tag points, write:

```
(*_função)();
```

Few functions are called directly via GDA. Many parts of WiOS are on page 1, from 4000H to 7FFFH, where the calling task is. In this case, the tasks must call a function that maps to the driver, calls, restores the thread, and returns to the task. This function is called 'caldrv' (call driver).

The reference GDA input is named '_caldrv'. WiOS functions should always be called with:

```
(*_caldrv)(...);
```

where '(...)' represents the arguments to be sent to the 'caldrv' function.

WiOS needs to know which driver you want to call. You can usually search for the driver by name and get the identifier. But there are some default drivers that are always loaded when WiOS starts. Their respective handles can also be found in the GDA.

The default drivers are as follows:

_hextdrv	External driver (internal WiOS functions that didn't fit inside).
_hfsdrv	File system driver (used for all file operations).
_hgiodrv	Graphic I/O driver (loads fonts and icons).
_hgrpdrv	Graphics driver (handles the graphics output to the screen).
_hmemdrv	Memory driver (memory mapping).
_hstddrv	Default driver (default functions).
_htaskdrv	Task driver (do anything with tasks).
_hwindrv	Window driver.

The first argument to the 'caldrv' function is the driver identifier. So just write

```
(*_caldrv)(*_hwindrv);
```

to access the window driver.

Each driver can have more than one function, so we need to tell the task which function we want to call. These functions are numbered from 1 to 32 767. The default drivers (that is, all 8 drivers mentioned above) are numbered linearly, starting with 1.

As function 4 in the window driver does something completely different from function 4 in the graphics driver, care must be taken not to access the wrong driver.

To make things easier, the function numbers of all drivers can be accessed via text. Each driver has its own header file where all functions are defined. To get the current data of a window at address 5000H, the source code looks like this:

```
(*_caldrv)(*_hwindrv,GET_WIN_STATE,0x5000);
```

where ‘hwindrv’ stands for ‘handle of window driver’.

To access driver functions by name, header files must be included, eg 'WINFNC.H' if you need window driver functions.

Each driver function returns a 16-bit unsigned integer. But not all functions return a valid value (because they don't need to return anything). Unique values must be returned directly from the function. For more than one value, a pointer to the address where the return values are stored must be returned.

7.4.10.1 – Direct Calls

WiOS has calls which can not be accesses via the drivers. They must be called directly. The variables are declared in 'GDA.H'. The list shows the usage of the commands in C-code.

(*_poll)((unsigned)mask)

Calls the poll routine with an event mask (events whose bits are set in the mask are not sent to the task). It returns the address of the 'event block'. This block is defined in 'DEFSTR.H' and has the structure described in chapter about "Event Block Structure".

(*_caldrv)((char)d_handle, (D_FNC)func, (uint)arg1, (uint)arg2, ,...)

Calls the driver-caller, a routine which calls the driver with the handle 'd_handle', and sends the function number as the first parameter, then the arguments.

(*_cal_seg)((uint)addr , (T_SEG)seg , (char *)block)

Only for tasks. Switch to the multi part segment 'seg', call the address which is pointed to by 'addr' and send 'block' as the address of 'adrblk'. The stack pointer of the old segment is saved and the one of the new segment is restored. The address of the routine to save the stack, restore the caller's segment and restore the old stack is put on the stack of the called function, so the return from a multi part function is simply done with a 'RET' instruction or with normal end-of-function in C. This function always returns a 16-bit value (in HL) to the caller function. Multi part functions can be nested as long as the stack does not underflow the 16K of the task or (which is more likely) the task's code is overwritten. No checking is done for that.

(*_dcal_seg)((uint)address, (T_SEG)segment, (char *)block)

Only for drivers. Switch to the multi part segment 'seg', call the address which is pointed to by 'addr' and send 'block' as the address of 'adrblk'. The stack pointer is not changed, since drivers use the WiOS-stack. The address of the routine to restore the caller's segment is put on the stack of the called function, so the return from a multi part function is simply done with a 'RET' instruction or with normal end-of-function in C. This function always returns a 16-bit value (in HL) to the caller function. Multi part functions can be nested as long as the stack does not underflow the upper barrier of the GDA – else strange things may happen and the system will be instable. Since there's more than 4K of free stack size, this should never happen.

7.4.11 – Event Reference

The following events are defined as the 'standard' events and event handlers should be able to deal with them.

The following list gives their name, number and the needed arguments if you send the data – and this is also the structure for the task receiving the data. In contrary to subroutine calls, these events do never return a value since the program receiving the events is not called immediately. An event is just a block of data which is put on a stack, telling the destination task after the polling that it should do something. Due to this fact, you should care about sending the correct data. Keep in mind that it is normally not your program receiving the data you send, and tasks from other people might not have such a powerful error-handling routine like your tasks have. Also, the events are sent one by one after each polling – so if you put 50 events on the stack, it will take a while to 'clear' it.

Event names with an asterisk (*) are sent by WiOS and should NOT be sent by any task. They should be handled as 'receive-only'.

7.4.11.1 – Event Block Structure

Every event contains a block of 16 bytes for data storage:

Type	Name	Description
T_TASK	task	Handle of destination task.
D_FNC	event	Event number.
uint	array[6]	For data storage.
char	sender	Handle of source task which sent this event.

This structure is defined as 'EBSTR' in 'DEFSTR.H'.

7.4.12 – Data Exchange Specification

These events are the so-called 'user-events'. They are sent by the E_USERMSG event and, although the data has no effect on WiOS itself, you should use the following definition for standard data transfer to other tasks so every programmer who read this documentation can write applications which can handle this kind of data.

7.4.12.1 – Execution of a data exchange

To send data to other tasks, a special procedure is needed. First, an end-of-drag-event ('E_EDRAG') with the basic data-type must be sent to the destination task. Although it could be possible that the destination task masked out this event, it is strongly recommended not to do that. Normally, the destination task receives this event and has to answer. As soon as the source task (sender) receives the message 'SEND', it can send the destination task the 'DATA' message with detailed information about where the data is and what type and size this data has.

If a task shall not receive any data, it should also answer the end-of-drag event with the message 'NOSEND' so any source task which wants to send data will know that this task will not receive data.

Between these steps, it is required that these tasks poll. And, as if it wouldn't be hard enough already, each task should also be prepared that the next event after polling is not the answer but could also be a open or redraw request.

The program flow looks like this:

Source Task	Destination Task
• send 'E_EDRAG' event with data-type • poll	
	• answer with 'SEND' or 'NOSEND' message • poll
• check for 'standard' events (and if yes, poll again) • check for 'SEND' or 'NOSEND' message	
These steps are only necessary if the 'SEND' message was sent	
• if 'SEND', send 'DATA' message with details • poll	
	• check for 'standard' events (and if yes, poll again) • handle reception of data

From this flowchart you can see that this process is not too hard to handle. When viewing each task as separate, the sender has to send the 'E_EDRAG' event and poll as long as the 'SEND' or 'NOSEND' message comes, then send the data (if destination task wants to) and poll again.

For the destination task, it's even simpler: it evaluates the 'E_EDRAG' event and sends either 'SEND' or 'NOSEND' message, then polls. If it wants to receive the data, it polls until the 'DATA' message comes, else it's finished. Nothing else than sending the 'NOSEND' event must be done if the task doesn't need this kind of data.

From this, it is obviously that answering the 'E_EDRAG' event is essentially, or else the source task will 'hang' in an endless loop until an 'E_SHUTDOWN' event comes.

7.4.12.2 – Data Type Definition

This is a weak point. Since I have never been programming on existing multitasking systems, the data type definition is in no way perfect nor were the data types created from their need in real applications – there has not been any application for WiOS yet.

7.4.12.3 – Data sent with the 'E_EDRAG' event

Data type names in brackets are defined in ⊠DTDEF.H⊠ and must be sent in array[0], the specifier in array[1].

Data type: 0 ('TEXT').
Specifier: 0 ('STRING').
Description: One or more text strings with no special function.

Data type: 0 ('TEXT').
Specifier: 1 ('FILENAME').
Description: One or more text strings with filenames to be handled / edited / opened.

Data type: 1 ('GRAPHIC').
Specifier: 0..255 bits per pixel.
Description: One rectangular bitmapped image (resolution is defined in the data itself).

7.4.12.4 – Data sent with the 'E_USERMSG' event

Whenever sending user messages, array[0] holds the user-message code. That is, in this case, code 'DATA', which is defined in 'DTDEF.H'.

Data: STRING
Arguments: array[1] Address of data block.
array[2] Segment of data block.

Data Block

Offset	Len	Valid Entries	Description
+0	2	0~65 535	Number of strings.
+2	2	1~3071	Number of segments used for data.
+4,6,...	2	0~3071	Segment numbers with strings.
After segment numbers:			
	2	0	Null-terminated strings.
		1~255	Number of fields.

- If strings are null-terminated, the strings come directly after the '0'.

- If strings are in fields of fixed length, there follows the length of the field.
- 2 bytes for each field length. After the length data comes the field-formatted string data.

Note: If the strings exceed 16K, it's up to the program to switch the segments.

Data: FILENAME

Arguments: array[1] Address of data block.
array[2] Segment of data block.

Data Block

Offset	Len	Valid Entries	Description
+0	2	0~65 535	Number of filenames.
After number of filenames:			
			Null-terminated strings.

Note: The filename data structure is exactly like the one of null-terminated strings – with the difference that the destination task does not have to insert this data somewhere but to open these filenames. In practice, that means to load these files – either in new windows for each file or for a slide show, one after each other, depending on the program. Also it's up to the destination task to parse the filename extensions or the files whether they are valid files for this task.

Data: GRAPHIC

Arguments: array[1] Address of data block.
array[2] Segment of data block.

Data Block

Offset	Len	Valid Entries	Description
+0	2	Unused.	
+2	2	1~3071	Number of segments used for image.
+4,6,...	2	0~3071	Segment numbers with image-data.
After segment numbers:			
	2	0~65 535	Width of image.
	2	0~65 535	Height of image.
		15-bit-mapped image data (see VDP-spec).	

Note: If an image exceeds 16K, it's up to the program to switch the segments

7.4.13 – GDA Reference

This section describes the list of needed GDA variables.

The GDA area starts at C030h. it's end is open since it can be expanded in future. Do not change reserved variables. They are not protected, but changing them may result in system instability or a crash. All GDA variables are pointers to the address of the variable or field. Type definitions are in 'DEF.H'.

Type	Name	Offset	Len	Description
T_SEG	*_tot_seg	+0	2	Total number of segments in computer.
T_SEG	*_free_seg	+2	2	Current number of free segments.
T_SEG	*_tmp_seg	+4	2	Segment of temporary segment.
<unused>		+6	2	
<internal use>		+8	2	
T_TASK	*_p1_task	+10	2	Current handle of task in page 1.
T_SEG	*_p1_seg	+12	2	Current segment in page 1.
char	*_p1rseg	+14	2	Mapper segment number of current segment in page 1.
char	*_p1_slt	+16	2	Slot of current segment in page 1
<unused>		+18	2	
T_SEG	*_p2_seg	+20	2	Current segment in page 2.
char	*_p2rseg	+22	2	Mapper segment number of current segment in page 2.
char	*_p2_slt	+24	2	Slot of current segment in page 2.
<internal use>		+26	2	
uint	(*_caldrv)(.)	+28	2	WiOS-function call entry address.
T_TASK	*_hmemdrv	+30	2	Handle of memory driver.
T_TASK	*_hstddrv	+32	2	Handle of standard driver.
T_TASK	*_htaskdrv	+34	2	Handle of task driver.
T_TASK	*_hfsdrv	+36	2	Handle of file system driver.
T_TASK	*_hgiodrv	+38	2	Handle of graphic I/O driver.
T_TASK	*_hgrpdrv	+40	2	Handle of graphic driver.
T_TASK	*_hwindrv	+42	2	Handle of window driver.

T_TASK	*_hextdrv	+44	2	Handle of external driver.
uint	*_wiosver	+46	2	Current version of internal WiOS part.
VOID	(*_wiosend)()	+48	2	WiOS shutdown call (only in Alpha-version).
uint	(*_poll)()	+50	2	Poll entry address.
<internal use>		+52 to +106		
VOID	(*_dump)()	+108	2	Entry call to make a memory dump on V9958 (argument is address).
struct EBSTR	*_eventblk	+110	2	Address of event-block holding the data after polling
struct WIBSTR	*_wibblk	+112	2	Address of window information Block struct
WINSTR	*_winblk	+114	2	Address of window data block
uint	*_act_font	+116	2	Handle of current font
uint	*_std_font	+118	2	Handle of standard (⊠system⊠) font
<internal use>		+120	2	
char	*_busyflag	+122	2	Status of direct-write VDP status
uint	*_adrblk	+124	2	Address of global, segment-switch independent address block
char	*_mb	+126	2	Current state of mouse buttons (bitmapped). bit description 0 ⟶ left trigger state (1=pressed). 1 ⟶ right trigger state (1=pressed).
char	*_cltype	+128	2	Type of last mouse click (bitmapped). bit description 0 ⟶ left single click (1=clicked). 1 ⟶ left double click (1=clicked). 2 ⟶ right single click (1=clicked). 3 ⟶ right double click (1=clicked).
uint	*_cltime	+130	2	Time of mouse click (array of 2). offset description +0 time of last left-click. +2 time of last right-click.
int	*_coord	+132	2	Current mouse coordinates (array of 2).

uint	(*_cal_seg)()	+134	2	Entry call for tasks to call multiple-part functions.
<internal use>		+136	2	
uint	(*_dcal_seg)()	+138	2	Entry call for drivers to call multiple-part functions
<internal use>		+140	2	
<internal use>		+142	2	

... to be expanded in future

7.4.14 – Menu Reference

WiOS' menus and icons have to be handled by each task itself. Yet, there's no automatic handling of mouse-clicks in defined areas, but WiOS offers helpful functions to get the most of menus with an ease of usability.

There are two main functions you need to handle menus: 'DRW_MENU' to put the draw the menu on the screen and 'CHK_MENU' to check whether specific coordinates are in one of the defined points or not.

When talking about items in this section, a single menu point with a possible graphic icon is meant.

The function CHK_MENU returns the number of the item on which the coordinates (normally the pointer) are, which are counted linear from the beginning of the menu-list. Each textentry, button, and box is one item. If the menu consists of a list with 5 entries and two options (in this order), the latter entries will return the item numbers 5 and 6, if the coordinates of the pointer are over them, since the first 5 entries have the numbers 0 to 4. A special type is the scrollbar, because it contains of 5 items in the following order:

1. Arrow up/left;
2. Page up/left;
3. Slider;
4. Page down/right;
5. Arrow down/right.

Depending on whether the scrollbar is vertical or horizontal. To get to know more exactly how the items are evaluated, create a menu with different types in one list and print the returnvalues of the 'CHK_MENU-function' (with the mouse-coordinates the coordinate-agrument) on the screen.

7.4.14.1 – Menu types

The menu functions offer 5 different types of menus:

Option: text strings with an option box on the left. Options are a block of one or more items where each item can be enabled or disabled separately.

Radio Button: text strings with a radio button on the left. Radio buttons are a block of one or more items where only one item can be enabled. If another item in this block is selected, all other items will be disabled.

List: plain text strings. Normally, lists are used for pull-down or pop-up menus

Scrollbar: graphical display of a given scale using a user defined scale with the possibility to select a certain area in the given scale.

Box: rectangular area. Boxes are not visible on the screen. They can be used for checking any rectangular area, e.g. for menus that cannot be handled with the above menu types.

The complete menu data of a window is normally placed in one big data block. Such a block uses the basic idea of a macro language, but at a very low level, i.e. word-, byte- or bitmapped. The idea behind this is to have one structure where each item has its own number. The coordinate checking has to be called only once to get the item number of the desired coordinates, no matter if you have only a pull-down menu or a complex structure with many options, radio buttons and scrollbars.

7.4.14.2 – Menu Features

Option / Radio Button: Items can be printed in both directions, from top to bottom, and from left to right. Independent from the direction will the coordinate checker validate coordinates beginning from the left – at the icon – to the right end of the text of each separate item,

i.e. if you check an area right to the text, you will receive no item number.

When writing items in the destination from left to right, WiOS will calculate the width of the longest string in the option / radio button block and use this as the standard distance between each item. Although the distance is equal, the items begin with the icon and end with the right end of the text of each item.

Disabled items will be printed in grey (instead of black). Whenever a item is selected, it is up to the task to set (and for radio buttons: reset all other) icon-enabled-bits.

Lists: Items can only be printed from top to bottom. The coordinate checker calculates the width of the longest string and will – in contrary to the option / radio button – use this width for each item in the list block.

Disabled items: see option / radio button.

Items can have an arrow icon on the right border to indicate the existence of a sub-menu – which must be opened as a new window by the task.

Scrollbar: They can be adjusted in horizontal and vertical direction, and they are always separate (no combination in one scrollbar block possible, but there can be more than one scrollbar in a menu block!!!)

Disabled scrollbars will not have any visible difference in graphic – the task has to indicate that separately on the screen. The scale of the 'virtual size' (similar to windows), the size on the screen in pixels and the size of the area represented by the slider (see 'work-area') can be chosen freely. Yet, there are some problems drawing very small sliders.

Slider-handling, scrolling and page up/down must be completely processed by the task – the step-size is not pre-defined. If the user clicks on the slider and drags it around, it is up to the task to trace the mouse and show the slider on the screen. This might be fixed, if wanted.

Box: It can be freely defined with a coordinate pair (upper-left / lower right)

7.4.14.3 – General conventions about menus

If items overlap, the checker will return the item number of the first valid area of the coordinates. If they overlap, it will return, not as drawn on the screen, the number of the 'lowest' area (e.g. if two boxes intersect, the number of the box drawn first will be returned although only the second box is visible since it has 'overwritten' the first box).

Every item can be disabled separately. Disabled items are still displayed on the screen but are not checked by 'CHK_MENU'. The item numbers of the following items stay the same, so for example if you disable the first item of a list box (which has number '0'), you will not be'able to check it any more, but the second item still has the number '1'.

Border styles can be chosen for each item block except for the scrollbars. At the moment, only two border styles are valid: no border and filled box border with standard background color (which can not yet be altered). The border distance in pixels is doubled for x-direction because in 512x212 resolution, a pixel is twice as high as it is wide.

7.4.14.4 – Menu-Block Structure

The menu block is one big field of data where one or more menu types can be defined. The following symbols are used in the structure definition:

prefix

b	Byte (1 byte).
w	Word (2 bytes).
t	Text (variable byte length).
#	Definition of valid values for 'D' (see suffix).

suffix

N	Any number.
D	Number from a pre-defined list.
S	And character string.
0	Must be this value.

The menu types are described in the appendix, chapter 7 – WiOS, section 7.11 – Menu block structure.

7.4.15 – Programming Tips

It is not only of importance to know which functions do exist, but also in which order and, that's even more important, when they should be used. Although there are sample programs in this documentation, you can look up the way how special things are managed in this section.

For programming applications in a graphic-based multitasking system, you often need more than just calling one function to do common procedures. This can not be derived from the functions itself. You need to know how these procedures which can only be done with the cooperation of two or more functions are handled.

During your first programs, it is advisable to use the structure in the sample-programs as-is, no matter if you understand why it is done like this or not, since there are other things you should focus on first – like working with the ASCII-C compiler or testing the graphic functions. If you think you understand the basics of WiOS and want to do real applications (or if you just want to know anything), go on reading.

7.4.15.1 – Windows

To put windows on screen, there are two major steps necessary. One step does the 'virtual' stuff for a window, that is the calculation and storing of the window in memory so WiOS is able to handle all user-entries concerning the modification of the window using the mouse.

The second step is changing the window 'physically', i.e. draw and move the window on the visible desktop.

Opening Windows

First of all, you can only draw windows in the event-handling routine – after polling. There is no difference whether the user, your task or another task wants to redraw a window of a task.

User

After polling, the task receives an open-window-event whenever a window was dragged, rescaled,... by the user. The task has to call the 'OPEN_WIN' function which adds the redraw-event. After polling, the task receives this event and must redraw the window on the screen.

Your Task

Creating a new window can be done with the 'CREATE_WIN' function of the window driver. This will add your window to the window list and fills in some variables. To draw the window on the screen, you have to arrange that you will receive a redraw-event after polling. This event is sent by the 'OPEN_WIN' function. To get the complete data of the new window, call 'GET_WIN_STATE'. This data can be sent to the 'OPEN_WIN' function, which adds the redraw-event. Moving an existing window is done as described in the steps before without the 'CREATE_WIN' function.

Other Tasks

Forcing other tasks to move their windows can be done exactly by sending them the openwindow-event – which is exactly the same way WiOS informs the tasks after the user has moved a window. The corresponding task will then call the 'OPEN_WIN' function which sends the redraw-event to itself, and after polling, it redraws its window.

Why sending the open-event, let the task call 'OPEN_WIN' before the redraw-event is sent? Although there can already be given some restrictions for the user not to move or rescale the window as he wants, it is of desirable to have more control over the window. The basic idea behind this is to 'ask' the task before giving the command to redraw the window. The task may then determine whether or not to accept the changes. These changes are sent by the 'OPEN_WIN' function. it's up to the task to send the changes to the window driver and update the window entry in the list or to reject these changes by ignoring this event.

Why cannot the task redraw the window directly after the open-event has been sent? There is more than only one window on the screen. Changes on one window can affect the others as well. These changes can not be predicted before the open-event sends the final win-

dow-data to the 'OPEN_WIN' function. If a window is changed (i.e. moved, rescaled,...) on the screen, other windows may be visible now. These windows also have to be redrawn. The window-driver does not send extra open-events to the tasks of these windows, but only redraw-events – since there is nothing changed with these windows. This requires a separate redraw routine which must be independent of the open-routine.

Regarding these facts, a task using windows has to have the following routines:

- Window-creation-routine for the 'standard' windows (called only at the beginning).
- Window-redraw-routine (handles the redraw of the windows on screen).
- Window-open-routine (checks window-parameters and calls 'OPEN_WIN').
- Event-handling-routine (calls window-open- or window-redraw-routine).

Redrawing Windows

The redraw-routine has to know a bit more than just the data in the window structure. This data can be ordered by the window driver and is called the window-information-block (WIB). It has the following structure:

T_SEG	stackseg	segment of the redraw-stack.
uint	elements	number of rectangular areas in redraw-stack.
struct WINSTR	*windat	pointer to window-data (outside page 1 or 2). Needed window-members are: x,y nx,ny realx,realy
uint	sx,sy	start coordinates of window-copy (always 0,212).
uint	offx,offy	offset for work-area.
int	moffx,moffy	move-offset relative to last position.
uint	toffx,toffy	title offset.
uint	t_nx,t_ny	title width.

The redraw stack has the following structure:

uint	x1,y1	Upper left coordinate (absolute).
uint	x2,y2	Lower right coordinate (absolute).

The task's redraw-routine has to look like this:

- Call 'GETWIB' and store return-address
- If elements is equal 0, nothing is to do
- Draw your own data at coordinates (sx+offx , sy+offy) that come with the 'GETWIB' function
- Call 'COPYWIN'

For examples, look in section about Sample Programs.

Scroll Requests

Whenever an 'E_SCROLL' event is sent to another task, must be taken care to send an 'E_WOPEN' event immediately afterwards. No polling should be done between the sending of the two events.

If there are protests, I might change WiOS to send an open window request automatically if a scroll event is sent.

7.4.15.2 – User Input

One weak point in WiOS is that there is no built-in routine which allows the user to edit one or more characters. That has to be simulated by your program (i.e. you have to write your own routine) until the routine will be present in a future version of WiOS.

Getting the next pressed key:

- Call the external driver with function 'GETKEY'.

Wait for a current keypress:

- Call 'RESKEY'.
- Loop until 'GETKEY' returns not equal 0.

Only ASCII codes will be returned. Yet, you cannot check whether any control key has been pressed if there is no separate ASCII-code for the key with and without the control key (e.g. SHIFT+ 'A' returns a

different code that 'A', but SHIFT+CURSOR returns the same code as CURSOR only).

This might be fixed in a future version if there's need for that.

7.4.15.3 – Using the Temporary Segment

Whenever you need to store 16K or less data temporarily, for example for calculation purpose or for data transfers, you may use the system's temporary segment. Its number is in the GDA ('_tmp_seg').

This segment is also used by the system as a temporary page (e.g. whenever data from disk is loaded) and it might be used from other tasks. So do not store your data there for too long, or it might be altered. If you want to use this page, be sure not to load from disk or poll – in these cases, your data is lost for sure. Instead, you can allocate a segment temporarily and free it later, if you need it to hold the data for several pollings (which is strongly recommended, since frequent polling is an essential part of co-operative multitasking).

The temporary segment is always in the main-slot.

7.4.15.4 – Mouse Pointer

The mouse pointer is moved in interrupt, so every 1/60 second, a function which reads the new mouse coordinates and puts the pointer to the screen is called. The sprite data must be written to VRAM, and therefore it is needed to set up the write address and send the coordinates to the VRAM. From this fact arises the problem that there can't be another program which writes data linear to VRAM through P#0 or sets up a VDP command. Linear data would be written to the correct place as long as the interrupt function is called, then any other data would be written behind the sprite coordinates. If a VDP command is set up, the register pointer would be changed to R#0 from the interrupt routine and all other register data would be written to R#3 and up. One possibility is to disable the interrupts, but if there are other tasks running in interrupt like a music replayer, they would halt as well. This is handled with the variable 'busyflag', which can be found in the GDA as '_busyflag'.

Whenever this flag is set to a value not equal 0, the pointer update on the screen will not be processed. If you want to perform a direct write to VRAM via P#0 or set up a graphic command, set the busyflag, and if you finished, reset it. When writing a lot of linear data, you could as well set up a graphic box and send the data through P#2 to prevent long phases where she pointer does not move.

7.4.15.5 – Sending data to other program parts using the address block

Sometimes, it is necessary to send other programs more than just one or two arguments. This works transparently with the ‘caldrv’ function, bus there are cases where the data must be stored in a place which is not affected by segment switching by the programmer. One of these cases is the calling of multi-part tasks. Since you know that it is taboo to write data in the GDA or somewhere outside of a task’s own segments, there is a place where a task is allowed to store data: this area is pointed to by the GDA variable ‘_adrblk’. This area has place for 16 bytes which can be defined freely. This is not too much, but it is enough for most cases.

Another use for the address block is to eliminate a weak point of C: the fact that only one parameter can be returned. Sometimes, it is necessary to return more than one value. These values could be stored in this area.

Remember that this is not a place to store data permanently since other tasks may use it as well. So, similar to the temporary segment, store only very immediate data at this address, for example if you call a function in another segment of a multi part application.

7.4.15.6 – Stacks

Every task has its own stack-pointer inside the actual segment. Whenever a task is loaded, this stack is set to the top of the 16K page (i.e. to 8000) Whenever calling WiOS-functions, the is stored data on the stack – also whenever a task is called from WiOS. Since the code size of every part in an application may vary, there’s no limitation or internal checking whether the stack overwrites areas of the code. If you experience problems with applications which code size is 15k or more, try to move some functions to a new part. Normally, this should not happen,

but if you nest 100 inter segment calls to multi part functions, your stack might come dangerously near your program code.

7.4.15.7 – Program Termination

For WiOS applications, it is important never to end a task by returning from the main function which is called after loading. Also, you may never use the 'exit()' routine, CTRL+STOP or any other DOS abort routine, since the memory pages are altered and WiOS interrupt driver is still active. The task must be closed with the KILLTASK function of the task driver. Since this function does not work in this Alpha-Release, you should use the 'ESC' key which is scanned at each polling. This routine shuts down WiOS, restores the original memory segments, hooks out the interrupt driver and returns to DOS.

7.4.16 – Compilation

The creation of a WiOS task requires several steps, which differ slightly for single and multi part applications.

First, a compilation batch file is needed (let's call it 'COMP.-BAT'). If should basically contain the following commands:

`[driver\path\]cf %1`	Parse and create .TCO file
`[driver\path\]fpc -u %1`	Check function parameters with functions in this file
`[driver\path\]cg %2 %1`	Generate .MAC file from .TCO file
`[driver\path\]m80 =%1/m`	Generate .REL file from .MAC file
`del %1.tco+%1.mac`	Delete unnecessary files

The option '%2' at 'cg' should normally be '-r2000' to have enough space for functions. This can be set permanently to this value so you don't have to type it for each compile, but if changes are needed, you have to edit it separately. For C files which require a special pool:symbol:hash proportion for the parser and memory set-up (-rnnnn) for the code generator it is advisable to create separate batch files for them (for example TASK1.BAT to compile the file TASK1.C) where the '%n' arguments can be replaced by concrete names and options, so you have only to type 'TASK1' to compile TASK1.C.

For advanced C users, it is preferable to have two compilation batch files. If the code generator wants a different '-rnnnn' value, it is not needed to create the .TCO file again.

Therefore, here are my personal batch files:

COMP.BAT

```
cf -r7:5:2 %1
fpc -u %1
comp2 %1 %2
```

COMP2.BAT

```
cg -r2000 %2 %1
m80 =%1/m
del %1.tco+%1.mac
```

The first batch file is called with 'COMP filename' (without extension) and creates only the .TCO file and checks the function parameters. If no errors occur, it calls COMP2 with the arguments from the command line.

In the second batch file, the code generator and assembler is called, and the non used files are deleted.

The advantage of using two files is to be able to 'continue' the compilation process at the code generation, since the parsing is not needed is the code generator has a memory error.

Also, you can override the '-r2000' option by writing a second argument in the command line like:

```
comp2 task1 -r1500
```

The code generator receives the line 'comp2 -r2000 -r1500 task1' and uses the later options rather than preceding options.

7.4.16.1 – Single Part Applications (up to 16K code)

They consist (normally) of several C files. To link the .REL files, a line similar to the following is needed:

```
c:\c\180 /p:3ff6,taskhead,task1a,task1b,task1c,task1d,
                ...,gda,c:clibs,c:crun/s,c:cend,task/n/e
```

This creates a single '.COM' file with about 16K of zeroes and the code beginning at 3FF6H – so that the real application (without the file-header) starts at 4000H.

Then MAKDRV must be called to cut the preceding 16K and to fill the correct values in the header like:

```
makdrv task.com task.tsk
```

This creates the WiOS-executable file TASK.TSK. Its file size should be smaller than 16K. If the '.COM' file has more than >16K of code (i.e. is larger than 32K), MAKDRV will not proceed.

After the task is created, the '.COM' file is no longer needed and may be deleted.

To make things easier, this process should also be done in a batch job. For that purpose, let's suppose the task file shall be called TASK.TSK, the main source code TASK.C and the standard header file TASKHEAD.MAC. Then a file called LT.BAT (which means Link Task) has to be written:

LT.BAT

```
c:\c\180 /p:3ff6,%1head,%1,%1,%3,%4,%5,%6,%7,%8,%9,
                    gda,c:clib/s,c:crun/s,c:cend, %1/n/e
makdrv %1.com %1.tsk
del %1.com
```

If TASK.C is the only file needed for the task, just write

```
LT TASK
```

to link TASKHEAD and TASK, create the .TSK file and remove the '.COM' file. If you have another part called 'TASKB.C' which has to be linked, just add 'TASKB' at the end, as well as up to 9 other parts. If you have more parts, it's advisable to write an extra batch file for the linking process.

The creation of your task is then simply done with two commands:

```
COMP TASK
LT TASK
```

If no errors occurred, start WiOS (W.COM) and watch what your program does.

7.4.16.2 – Multi Part Applications (more than 16K of code)

The first part (with the init routine) is created in exactly the same way as a single part application until the '.COM' file is done. Every part (normally) also consists of several C files.

All the following code parts are compiled with the same steps mentioned above – completely independent of the other parts – but they differ in the way they are linked. They do not have the standard header file for tasks, since it is only needed at the beginning of the .TSK file.

Their link line looks like this:

```
c:\c\180 p:3ffe,etskhead,task2a,task2b.task2c,task2d,
          ..,gda, c:clib/s, c:crun/s,c:cend, task2/n/e
```

For a multi part application, you should have two or more '.COM' files.

The '.COM' files of multi part applications should never be deleted, although they use a lot of disk space, because they are always needed to create the .TSK file. This is done by:

```
makdrv task.com task2.com task3.com ... task.tsk
```

Write as many parts as you want into this line and the name of the .TSK file at the end.

One problem is, of course, the limited size of the command line, but if there's really somebody writing such a huge application that suitable filenames (1.com, 2.com, 3.com,...) do not fit into the line.

For this kind of task, it's also easier to have some batch files for compilation. For that purpose, let1s suppose the task file shall be called TASK.TSK, the main source code TASK.C and TASKB.C (for part 1) and TASK2.C and TASK2B.C (for part 2), the standard header file TASKHEAD.MAC and the extended header file ETSKHEAD.MAC. Then a file called LT1.BAT to link the first part has to be written:

LT1.BAT

```
c:\c\180 p:3ff6,%1head,%1,%2,%3,%4.%5,%6,%7,%8,%9,
                        gda,c:clib/s, c:crun/s,c:end, %1/n/e
```

This batch file is the same as LT.BAT without the calling of MAKDRV and without deleting the '.COM' file, which is needed for later creation of the task.

```
LT TASK TASKB
```

links TASKHEAD, TASK and TASKB, assuming you already compiled TASK and TASKB with the COMP batch file.

To create the other parts, you need another batch file because the header file is different. So create a file called LT2.BAT which is used to link all other parts.

LT2.BAT

```
c:\c\180 p:3ffe,e%1head,%2,%3,%4,%5,%6,%7,%8,%9,gda,
                             c:clib/s,c:crun/s,c:cend, %2/n/e
```

To link the sample files TASK2, TASK2B and ETSKHEAD, type

```
LT2 TSK TASK2 TASK2B
```

Now if you have the '.COM' files, if could be handy to write a new batch file to create the task. This file, called MAKETASK.BAT could look like:

MAKETASK.BAT

```
makdrv task.com task2.com task.tsk
```

7.4.17 – Task Specifications

Multipart tasks are compiled and linked separately, and the 'interface' between them is an address table at the top of the page (around

4000H) and the WiOS functions which offer the possibility to carry data and jump and return to and from the threads multipart.

The first part or a multipart application does not have a standard address table. If other parts want to call a function in the first part, an address table must be configured in this part and the respective address must be placed in the function 'cal_seg'. The table can be anywhere between 4008H to 7FFFH and can be included in the standard header file as is done in the extended header.

7.4.17.1 – Headers

A task's 'standard' header must have a list of information about the task. Every task must have this header. For multi part applications, the first part contains the header.

An 'extended' header is only for extended parts, i.e. part 2 and higher of a multi part task. The headers are '.MAC' files to warrant the correct order of variable and pointer positions. All names with a '@' are declared public and can be accessed as a normal variable in the C code. Labels which are not pointed to in a 'ds' or 'dw' instruction in another line can be left out. They are just here to make clear what this line defines.

Header files must be assembled with

```
M80 =filename/m
```

to create a linkable .REL file.

You can set the number of needed segments in the header file, so it is guaranteed that if your task runs, it will have enough segments, or else WiOS does not load it. The number of needed segments must be at least as big as the number of program parts because each part needs one segment. But this number can also be larger if a task needs more segments to store data permanently. You can add the number of segments to the number written in 'PPARTS' and write the result at the constant 'NEEDSEGS'.

Remember that this is only for the number of segments which are needed permanently to run the task. Theoretically, you can tempo-

rary allocate all available memory later if needed to speed some things up or to store huge picture data.

Header Files

The following header files are essential to write tasks in C:

```
#include <stdio>
#include "def.h"
#include "gda.h"
```

and one or more of the following are optional:

```
#include "eventfnc"      Events.
#include "defstr.h"      Structures.
#include "winfnc"        See section about Function Reference
                         (WiOS Drivers).
#include "taskfnc.h"
#include "stdfnc.h"
#include "memfnc.h"
#include "grpfnc.h"
#include "giofnc.h"
#include "fsfnc.h"
#include "extfnc.h"
```

Standard Header Structure

Normally, you only need to change these two lines (and the task name).

```
NEEDSEGS   equ 1    minimum number of segments needed for
                    this task (incl. data segments).
PPARTS     equ 1    program parts (for multiple part tasks).
```

External header

```
WiOS_ver:  dw 0              WiOS version needed to run driver.
segs:      dw NEEDSEGS       Needed segments to load driver.
p_parts:   dw PPARTS         Number of program-parts in file.
t_main:    dw taskinit@      Task's start-address.
lastadr:   dw @endx@-4000h   Len-info for loader, filled by L80.
```

Internal header (4000H)

```
tsknam@:   dw taskname      Pointer to task's name.
p_stable:  dw segtable@     Pointer to segment-numbers.
stack:     dw taskmain@     Internal Stack Pointer of task (must
                            point to function taskmain in file).
usedsegs:  ds 2             Total number of used segments
                            (filled by WiOS).
```

Variable part (4008h)

```
segtable:  ds NEEDSEGS*2    Space for segment numbering
                            during allocation (filled by WiOS).
taskname:  dw 0             Task's version.
           db "Test-Task"   Task's name.
           db 0             Taskname termination.

           public tsknam@
           public segtable@
           extrn  taskmain@
           extrn  taskinit@
           extrn  @endx@

           end
```

External header for extended header structure

```
lastadr:    dw  @endx@-4000     Len-info for loader, filled by L80.
```

Internal header for extended header structure (4000H):

```
mp@seg@:   dw 0     Segment of 1st part with all task-infos
                    (filled by WiOS).
mp@seg@:   dw 0     Segment of this part (filled by WiOS).
stack:     dw 0     Internal stack pointer of task (filled by WiOS).
```

Variable part for extended header structure (4006H)

```
func_1:    dw test1@   Address of routine 'test1' (4006H)
func_2:    dw test2@   Address of routine 'test2' (4008H)
func_3:    dw test3@   Address of routine 'test3' (400AH)
```

```
⋮
        extrn test@
        public mp@seg@,my@seg@
        extrn  @endx@
        end
```

7.4.17.2 – Pre-processor directives

To get the optimum performance with the C compiler, the following lines should be on the top of each C file:

`#pragma optimize time`	It seems to have no effect on the code.
`#pragma regalo`	To enable register allocation, if possible.
`#pragma nonrec`	Not to store all variables for every function.

If a function is recursive, you may write the same line and replace 'nonrec' by 'ecursive' immediately before the function and add the line with 'nonrec' again after it.

Although this is the theoretical optimum, it's often better to disable register allocation so the code generator does not try to optimize the code. The program runs slightly slower and uses a few bytes more, but the compilation process will be up to twice as fast without optimization.

These lines are more likely for the real programmer:

```
#pragma optimize time
#pragma noregalo
#pragma nonrec
```

7.4.17.3 – External Variables

There are variables defined by you (task's name) and WiOS (segment numbers) in the header file of each task. If you want to have access to them, they should be declared in the C file:

```
extern char *tsknam;
extern uint segtable(0);
```

The number in brackets can be set freely, since C does no boundary checking either. This just means that the variable 'segtable' is an in-

dexed 16-bit variable. With the variable 'segtable' you know where your task, all the other parts and the data segments are. To get the number of the first non-code segment, just use the number of parts of your task as the index. For example if you have a multi part task with 2 parts, the index 0 and 1 is used for the code segments and the index 2 and higher contains the numbers of the data segments.

When writing the task's name, remember to add 2 to the pointer, since the first two bytes of the task name contain the version number of the task.

7.4.17.4 – Task Initialization

Whenever a task is loaded, WiOS calls the address pointed to by 'stack' in the first part of the task (see section bout Standard Header Structure). This can be used for task initialization, for example to check whether another instance of this task is already running or to check the presence of required drivers and, if necessary, to load them.

Important: This routine is not the real start of the task. It may not call the poll routine nor does any event have to be checked nor should windows be opened. Normally the init sequence is only for checking if the task finds everything which is required to run. This routine may return immediately without doing anything. The task 'officially'starts at the address pointed to by t_main. The handle of the task is passed as an argument of type char.

This value may be either ignored or saved (to prevent a 'search for name'-call later to determine the handle of the task). This function is necessary, but it's not necessary for the function to do something.

7.4.17.5 – Main Routine

This is the function where the task begins. The function should never return. If you want to stop the task, use the 'KILLTASK' function of the task driver. For the Alpha-Release, see section about Program Termination. This address is pointed to by 't_main' in the standard header file.

```
VOID taskmain()
{
    printf("task kas started!!!\n);

    ...create_and_open_windows...
    ...poll...
    ...do_something...
    ...format_harddisk...
    ...print_virus_message...
}
```

it's up to your imagination to create a usable program. This is the function which has to set up anything which can be seen on the screen, handle user events, doing anything – the 'main()' function from 'normal' C.

7.4.17.6 – Window Creation

Before you can see anything on the screen and let WiOS do all the work with user, you have to set up proper window data. For this, and also for some functions later in the redraw and open section, the 'windat' variable should point to the task's own window block which is used to create and change windows, so somewhere in the C file, there should be a line like:

```
struct WINSTR windat;
```

Here comes an example of a perfect set-up of a window for WiOS:

```
VOID creatwin()
{
int whandle,i;
    windat.x=50;
    windat.y=60;
    windat.nx=100;      /* work-area size */
    windat.ny=75;
    windat.vx=100*2;    /* virtual size */
    windat.vy=75*2;
    windat.scrx=100/2;  /* scroll offset (middle) */
    windat.scry=75/2;
    windat.minx=0;      /* no minimum limit */
```

```
    windat.miny=0;
    windat.maxx=0;      /*max size is virtual or scr*/
    windat.maxy=0;      /* size */
    windat.behind=-1;   /* open window on top */
    windat.winflag=0;   /* see referring section */
    windat.iconflag=127 /* allow all icons */
    windat.workflag=32+8+4+2; /* allow all clicks */
    windat.parent=-1;   /* window has no parent */
    windat.statflag=0;  /* . . . */

    for (i=0; i<25; i++) windat.dummy[i]=0
                                    /* reset dummies */
    whandle=windat.handle=(_caldrv)
                        (*_hwindrv,CREATE_WIN,&windat);
    printf("created window has handle %d\n",
                                          windat.handle;
(*_caldrv)(*_hwindrv,GET_WIN_STATE,&windat);
                                     /* get win data */
(*_caldrv)(*_hwindrv,OPEN_WIN,&windat;
                /* send data for opening the window */
}
```

There should not be any problems in understanding the code. One important thing is that 'whandle' should be stores in a global variable because it is the only point where the task gets to know the handle of the created window. This is important to redraw the correct window if a redraw request is sent. The 'GET_WIN_STATE' function before opening the window is necessary because WiOS filled some variables to the window structure like the window's task and the real window size including the icon frame.

7.4.17.7 – Polling and Event Handling

Here's an example of what should an event handling routine be capable to deal with. We use the simplest type of main loop:

```
VOID taskmain()
{
struct EBSTR *block;
.
.
.
```

```
    while (1)
    {       /* poll and return sdress of block */
            /* allow all events */
        block=(struct EBSTR *)(*_poll)(0);
            /* process the event */
        procevent(block);
    }
}
```

or the last two lines of code (for C cracks) so 'block' is not needed at all:

```
        procevent(struct EBSTR *)(*_poll)(0);
```

For the description of the event block, see section about Event Reference. The function 'procevent' should then handle all the necessary events and could look like:

```
VOID procevent(block)
struct EBSTR *block;
{
char c; printf("processing event %d\n",block->event);
    switch(block->event)
    {
        case E_NULL:
            printf("receiving the NULL-event...");
            break;

        case E_REDRAW:
            printf("REDRAW event for window %d
                        received\n", block->array[0]);
            redraw(block);
            break;

        case E_SCROLL:
            printf("SCROLL event for window %d
                        received\n", block->array[0]);
            wscroll(block);
            break;
```

```
case E WOPEN:
    printf("OPEN event for window %d
                received\n", block->array[0]);
    wopen(block);
    break;

case E WCLOSE:
    printf("CLOSE event for window %d
                received\n", block->array[0]);
    (* caldrv)
    (* hwindrv,CLOSE WIN,block->array[0]);
break;

case E PNTOUT:
    printf("pointer has left window %d\n",
                              block->array[0]);
    break;

case E _ PNTIN:
    printf("pointer has entered window %d\n",
                              block->array[0]);
    break;

case E MCLICK:
    printf("CLICK whandle=%d coord=%d,%d
    click-type:%d time=%d\n",
    block->array[0],block->array[1],
                               block->array[2],
    block->array[3],block->array[4]); break;

case E WKAREA:
printf
   ("pointer is over work-area of window %d\n",
                               block→array[0]);
break;

case E EDRAG:
    nosend();
    break;

default:
printf ("Event Handler was called with
```

```
                                        an unknown event\n");
        break;
    }
}
```

The example requires the existence of the functions 'redraw', 'wscroll' and 'wopen'. The 'redraw' function redraws the window physically on the screen. The following example simply would draw a white box in the work-area, assuming we have a box-draw routine to draw a box with the arguments (x,y,nx,ny,colour).

```
VOID redraw(block)
struct EBSTR *block;
{
struct WIBSTR *wib;
    wib=(struct WIBSTR *)(*_caldrv)(*_hwindrv,GETWIB,
                                     block->array[0]);
    _box( wib->sx + wib->offx , wib->sy + wib->offy,
            wib->windat→nx , wib->windat->ny , 32767);
    if (wib->elements) (*_caldrv)
                              (*_hwindrv,COPYWIN,wib);
}
```

After GETWIB was called, the window frame exists at the coordinates returned with the members 'sx' and 'sy' and the window data is copied to the global area (i.e. outside of page 1 and 2) pointed to by the member 'windat'. If the window is visible (it normally should, but the window driver is not fully optimized for that), it is copied from the hidden area to the screen.

The 'wscroll' routine simply has to determine whether the scrolling performed by the user (or requested by another task) is valid and confirm it by updating the scroll-offset of the window, poll again to receive the 'E_WOPEN' event and return to the normal task to poll (so the 'E_REDRAW' event can be sent). Since this routine also has to handle the open event, I use one common function for the scroll and for the open event. This function is called 'do_wopen':

```
VOID do_open(block)
struct EBSTR *block
{
    windat.behind=block->array[1];   /*setup window*/
    windat.x=block->array[2];        /*according*/
    windat.y=block->array[3];        /*to the block*/
    windat.nx=block->array[4];
    windat.ny=block->array[4];
                  /* send data for opening the window
                     and creation of REDRAW-events */
    (*_caldrv)(*_hwindrv,OPEN_WIN,&windat);
}
```

The 'OPEN_WIN' function changes the window in the memory list and sends the redraw event to the task (besides, it does more, like rearranging pane windows and all other windows which might be visible now, and sends a lot of open window requests, etc.)

Now for the function:

```
VOID wscroll(block)
struct EBSTR *block;
{
    windat.handle=block->array[0];
                              /* get window data */
    (*_caldrv)(*_hwindrv,GET_WIN_STATE,&windat);
    windat.scrx=block→array[1];
    windat.scry=block->array[2];
                  /* do one more possing to receive
                    the window-open request and send
                    the data to do_wopen          */
    do_wopen((struct EBSTR *)(*_poll)(0));
}
```

Although it looks a bit dirty because there is no event-checking done after the poll routine, it is correct. Every window scroll request from WiOS is followed by a 'E_WOPEN' event. If a task sends a scroll request to another window, is must also send an open window request immediately afterwards.

The event-handling is almost complete now:

```
VOID wopen(block)
struct EBSTR *block;
{
    windat.handle=block→array[0];
    (*_caldrv)(*_hwindrv,GET_WIN_STATE,&windat);
                                        /* get win data */
    do_wopen(block);
}
```

This function gets a copy of the window data in the task's own window block since to be able to change the size without setting up the data of the whole window itself.

7.4.18 – Security

Once a task has been created with 'MAKDRV' it should not be changed via a disk-editor. To protect tasks against changes, they contain a checksum which is checked by WiOS before executing the task. If the checksum is incorrect, the task will not be started. Yet there is no info-window for that. This checksumming is not a protection against hard-core crackers, nor is it a guarantee for the task being unaltered. It is some kind of 'guarantee' that if the task starts, it will be the same version the programmer compiled, and there will not be any bugs due to disk errors or bad modem lines.

7.4.19 – Starting Tasks with the Alpha-Release

Whenever WiOS is loaded, two tasks are loaded and started. They must be named

TASK.TSK and
TSK2.TSK

If these files are not present, WiOS will not start correctly. For Alpha-Testing, one file contains the normal code, and one file contains the code to send data or files, just as WiOS file manager will do in future. Of course, if you don't need to send any data, you can also create a dummy task which does nothing or even start the same task twice by just copying TASK.TSK to TSK2.TSK.

7.4.20 – Keyword Description

Address block: 16 byte area which stays untouched from segment switching. It can be used to store immediate data which has to be sent to other program parts. The GDA variable '_adrblk' points to its address.

Application: Task

GDA (Global Data Area): the area where global variables and function addresses which should be able to be accessed by other programs are pointed to.

Task: Application

WIB (Window Information Block): a data structure defined as WIBSTR window block The complete structure of a window which is defined as WINSTR. This block must be set up to use the window functions.

7.5 – DIRECT ACCESS TO THE DISK CONTROLLER

Although not recommended, direct disk access is possible, overriding the operating system and BDOS/Kernel. In this case, it is necessary to know which type of interface is installed, as different interfaces use different means for access. In this section, only the access to the FDC (floppy disk control) is described, in a very summarized way.

The FDC is accessed by writing and reading data from its internal registers. These registers are as follows:

1 – Status register.
2 – Command register.
3 – Track recorder.
4 – Sector register.
5 – Data logger.
6 – Drive register, floppy side and drive motor.
7 – IRQ register, busy and data request.

Some FDC's have differences between these registers. This will be described in detail later.

7.5.1 – FDC commands

There are 4 categories of commands that disk controllers can execute, as described below:

Type	Command	b7	b6	b5	b4	b3	b2	b1	b0
I	Restore	0	0	0	0	h	V	r1	r0
I	Seek	0	0	0	1	h	V	r1	r0
I	Step	0	0	1	T	h	V	r1	r0
I	Step-In	0	1	0	T	h	V	r1	r0
I	Step-Out	0	1	1	T	h	V	r1	r0
II	Read Sector	1	0	0	m	S	E	C	0
II	Write Sector	1	0	1	m	S	E	C	a0
III	Read Adress	1	1	0	0	0	E	0	0
III	Read Track	1	1	1	0	0	E	0	0
III	Write Track	1	1	1	1	0	E	0	0
IV	Force Interrupt	1	1	0	1	i3	i2	i1	i0

The flags shown in the table are described below.

r1,r0 Motor step rate (0=6ms, 1=12ms, 2=20ms, 3=30ms).
V Track number check flag (0=no, 1=check dest).
h Head flag (1=position head on track 0).
T Track update flag (1=update track logger).
a0 Data address tag (0=FB, 1=F8 [DAM deleted]).
C Side comparison flag (1=enable side comparison).
E 15ms delay (1=enables 15ms delay).
S Side comparison flag (0=compare to side 0, 1=compare to side 1).
m Multiple records flag (0=single record, 1=multiple records).
i3-i0 Terminates without interruption (INTRQ).
i3 = 1 ⟶ stop immediately, requires restart.
i2 = 1 ⟶ index pulse.
i1 = 1 ⟶ transition ready to not ready.
i0 = 1 ⟶ transition not ready to ready.

7.5.1.1 – Type I commands

Type I commands are used to move the drive head. The motor step is normally set to 6 ms (r1 and r0 = 0) for 3½" floppy drives. An op-

tional check of the head position can be done by setting bit 2 (V=1) of the command word.

When V=1, upon completion of the search, the track number of the ID field of the first sector found is read and compared with the content of the track register. If the two are the same and the CRC of the ID field is correct, an INTRQ without errors will be generated. Otherwise, the Seek Error bit of the status register will be set.

When V=0, when completing a search, the track number will not be verified. This mode must be enabled for unformatted floppy disks. The command ends when the last pulse is sent to the stepper motor. A pause is required before reading or writing for the head to stabilize on the track.

When the fetch is completed, an interrupt request is generated and the "busy" bit of the status register is reset to 0. When the CPU reads the status register, the interrupt signal is reset.

'Restore' command (search for track 0)

This command places the drive head on track 0. The track logger will be set to 0 and an interrupt will be generated when track 0 is reached.

'Seek' command

This command positions the head on the track indicated by the data logger. The FDC will update the track recorder and send pulses to the stepper motor until the head reaches the desired position. An interrupt will be generated at the end of the command.

'Step-In', 'Step-Out' and 'Step' commands

These commands send a pulse to the stepper motor. The 'step-in' command moves the head towards track 0, 'step-out' towards the last track and 'step' moves in the same direction as the previous command. The track register will only be updated if the "T" bit is set in the command word. An interrupt will be generated at the end of the command.

7.5.1.2 – Type II commands

Type II commands are used to read and write sectors on the disk. Before executing a type II command, the sector register must be lo-

aded with the desired sector. Upon receiving a type II command, the "busy" bit of the status register is set. If the sector ID field with the correct track and sector is not found, the status register's "sector not found" flag will be set and an interrupt will be generated.

The 'm' flag indicates multiple sectors. If 0, a single sector will be accessed; if 1, multiple sectors are accessed. In this case, the sector register is being updated and an address check can occur with each sector read. The FDC accesses the sectors in ascending order until the sector register exceeds the number of sectors in the track or until a forced interrupt is requested (Force Interrupt command).

The C flag is used to enable disk side comparison. If it is 0, there will be no comparison. If it is 1, the LSB bit of the disk ID field is read and compared to the content of the S flag.

'Read Sector' command

Upon receiving this command, the head is positioned, the "busy" bit of the status register is set, and when the ID field is found and contains the correct track, sector, side and CRC, the data field is made available to the CPU. A DRQ is generated whenever the data register contains valid data. In this case, the CPU must read the data immediately. The "lost data" bit of the status register will be set if the CPU has not read the data in time, but reading will continue until the end of the sector is reached. At the end of the read operation, the type "data address mark" found in the data field will be recorded in the status register (bit 5).

'Write Sector' command

Upon receiving this command, the head is positioned, the "busy" bit of the status register is set, and when the ID field is found and contains the correct track, sector, side and CRC, a DRQ is generated. The FDC counts 22 bytes (at dual density) from the CRC and the "write gate" output is activated if the DRQ is answered. If the DRQ is not answered, the command is terminated and the "lost data" bit of the status register is set. If the DRQ is answered, 12 bytes 00H (double density) will be written to disk. Then the DAM (data address mark) is determined by the 'a0' field of the command. After that, the FDC will write the data field and generate DRQ's for the CPU. If the DRQ is not answered in time for continuous writing, the "lost data" bit of the status register will be set

and a 00H byte will be written to disk. The command will continue until the last byte of the sector is reached. After the last byte of data is written, a two-byte CRC is computed internally and written to disk, followed by an FFH byte.

7.5.1.3 – Type III commands

Type III commands are used to access the headers of tracks and sectors on the disk.

'Read Address' command

Upon receiving this command, the head is positioned, the "busy" bit of the status register is set. The next ID field found is read and the six bytes of data from the ID field are assembled and transferred to the data logger. A DRQ is generated for each byte read. The six ID bytes are:

1 – Track address	4 – Sector size
2 – Side number	5 – CRC1
3 – Sector address	6 – CRC2

Although the CRC bytes are transferred to the CPU, the FDC checks their validity and the "CRC error" bit of the status register will be set if there is a CRC error. The track address of the ID field is written to the sector register to enable user comparison. At the end of the command, an interrupt is generated and the "busy" bit of the status register is reset.

'Read Track' command

Upon receiving this command, the head is positioned and the "busy" bit of the status register is set. Reading starts immediately at the first indexing pulse encountered and continues until the next indexing pulse. All gaps, headers and data bytes are assembled and transferred to the data register. A DRQ is generated for each byte transferred. Byte accumulation is synchronized for each address tag encountered. An interrupt is generated when the command completes. Address tag ID, ID field, CRC byte ID, DAM, data and CRC data bytes for each sector must be correct. Gap bytes may be read incorrectly during the write pause because of synchronization.

'Write Track' command (track formatting)

Upon receiving this command, the head is positioned, the "busy" bit of the status register is set. Writing starts immediately at the first indexing pulse encountered and continues until the next indexing pulse, at which point the interrupt is activated. The data request is activated immediately upon receiving the command, but writing will not start until the first byte of data is written to the data register. If this is not loaded in the timing of the index pulse, the operation is terminated with the device not occupied, the "lost data" bit of the status register is set and the interrupt is activated. If a byte is not present in the data register when needed, a 00H byte is assumed. This sequence is repeated from one index mark to another.

Normally, any data pattern that is loaded into the data logger is written to disk with a normal pattern cycle. However, if the FDC detects a data pattern from F5H to FEH in the data register, it will be interpreted as an address mark, without cycle generation or CRC. The CRC generator is started when an F5H byte is about to be transferred (in MFM). One F7H byte will generate two CRC bytes. As a consequence, bytes F5H to FEH cannot be part of gaps, data fields or ID fields. When formatting tracks, sectors can contain 128, 256, 512 or 1024 bytes.

7.5.1.4 – Type IV command

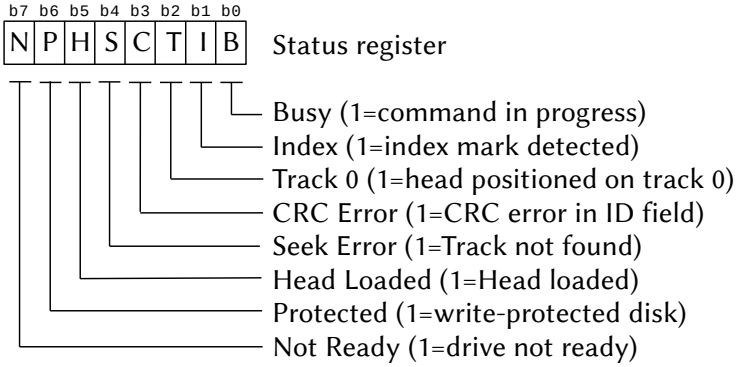
Command type IV (force interrupt) is generally used to terminate access to multiple sectors. This command can be loaded into the command register at any time. If there is a command running (bit "busy" = 1), the command will be terminated and the "busy" bit of the status register will be cleared.

7.5.2 – The status register

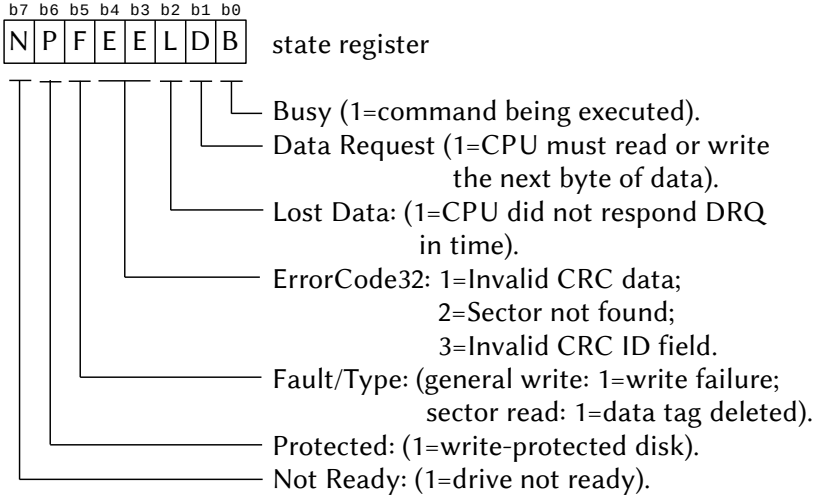
Upon receiving any command, with the exception of the Force Interrupt command, the "busy" bit is set and the other bits of the status register are updated or cleared for the new command. The user has the option of reading the status register through the program or using the DRQ line in conjunction with the DMA or interrupt. When the data register is read, the DRQ bit in the status register and the DRQ line are automatically cleared. A write to the data logger also has the same ef-

fect. The “busy” bit should always be monitored so that the user knows when a command is being executed. When using INTRQ, checking the “busy” bit is not recommended because reading this bit will reset the INTRQ rung.

Status register for type I commands



Status register for type II and III commands



Bits 1~6 are reset when updated. Error codes (ErrorCode32) are not valid for writing or reading tracks.

Status register for type IV commands

If a "Force Interrupt" command is received while executing another command, bit 0 (busy) of the status register is cleared and the remaining bits remain unchanged. However, if a "Force Interrupt" command is received when no other commands are running, bit 0 (busy) of the status register is cleared and the other bits are updated or cleared. In this case, the state register behaves as in type I commands.

7.5.3 – Additional functions and precautions

The FDC does not have internal selection for drive number, side, disk density and motor on/off control of the drives. These functions must be added by external circuits, which must be controlled separately.

As the FDC has only one track register which must be used for all drives, the track position must be saved in memory and the track register must be updated with each drive change.

7.5.4 – Formatting

In order for the disk to be usable, a process called formatting is required. In formatting, the disk is logically divided into tracks and sectors. The table below shows the data pattern and its interpretation by the FDC in the MFM system.

00~F4	Writes 00 to F4
F5	Write A1, CRC preset
F6	Write C2
F7	Generates 2 bytes CRC
F8~FF	Writes F8 to FF

Formatting example

The following example shows the data sequence that must be sent to the "Write Track" command to format a disk with 256 bytes per sector (MSX uses 512 bytes sectors by default). The values on the left are write repeat counters (in decimal) for the values on the right.

First, the Track Header must be written, followed by the sector ID and sector data fields (for each sector). Finally, 4EH bytes must be written until the command is completed.

```
Track Header (track header)
80 * 4EH
12 * 00H
03 * F6H    (writes C2)
01 * FCH    (index mark)
50 * 4EH    Sector ID field
12 * 00H
03 * F5H    (writes A1, preset CRC)
01 * FEH    (address ID mark)
01 * track number
01 * side number
01 * sector number
01 * 01     (sector size – 256 bytes)
01 * F7H    (writes 2 bytes CRC)
22 * 4EH
Sector field data
12 * 00H
03 * F5H    (writes A1, preset CRC)
01 * FBH    (address data mark)
256 * sector data
1 * F7H     (writes 2 bytes CRC)
54 * 4EH
End of track (fill unused bytes)
... * 4EH
```

7.5.5 – FDC access adresses

This section describes several access addresses for drive interfaces, based on both memory and I/O. For interfaces accessed by memory, the slot where it is installed must be enabled. For interfaces accessed by I/O, this care is not required. This type of access was used only in some Brazilian interfaces. The default access for MSX is through memory.

Addresses for memory access (Default)

7FF8H	R	Status register
7FF8H	W	Command Register
7FF9H	R/W	Track Register
7FFAH	R/W	Sector Register
7FFBH	R/W	Data Logger
7FFCH	R?/W	Side (bit 0) [Motor here?]
7FFDH	R?/W	Drive (bit 0) [Motor here?]
7FFEH – Not used		
7FFFH	R	Data request (bit 7) and busy (bit 6)

Note: MSXDOS/BarbarianLoader selects memory at addresses 8000H~BFFFH; in this case, BFFxH addresses should be used instead of 7FFxH.

Addresses for memory access (Alternative)

This mapping is used only by SpectraVideo's SV738 (X'Press) model, by Technoahead's BDOS and by Arabic BDOS. In the latter case, the addresses used are 7F80H~7F87H, and in the first two cases they are 7FB8H~7FBFH.

7FB8H/7F80H	R	Status register
7FB8H/7F80H	W	Command register
7FB9H/7F81H	R/W	Track Register
7FBAH/7F82H	R/W	Sector Register
7FBBH/7F83H	R/W	Data Logger
7FBCH/7F84H	R	bit 7 = IRQ/Not busy
		bit 6 = Data request
7FBCH/7F84H	W	bits 0/1 = Select drive
		bit 2 = side
		bit 3 = side

Addresses 7FBDH~7FBFH and 7F85H~7F87H are not used.

Addresses for access by I/O ports

D0H	R	Status register
D0H	W	Command register
D1H	R/W	Track Register

D2H	R/W	Sector Register
D3H	R/W	Data Logger
D4H	W	Drive (bit 1), Side (bit 4), Motor (bit ??)
D4H	R	IRQ/Not busy (bit 7), Data request (bit 6)

Addresses D5H to D7H are not used. This type of access is used by all Brazilian interfaces, except for ACVS/CIEL which uses standard memory access.

Reading through the D4H port is only supported by version 3.0 or higher. For earlier versions, bits 0 and 1 of the status register are used, which have the same meaning. Version 2.7 and above use mixed access, by I/O ports and by default memory.

Chapter 8
ADDITIONAL DEVICES

This chapter describes some devices that were not described in previous chapters.

8.1 – THE CLOCK AND SRAM

On MSX2 and higher micros, a specific chip is used for system clock functions, the RP-5C01. It is called CLOCK-IC. As it is powered by batteries, it is always active, even with the computer turned off. The watch has a small SRAM that is used to store some functions that the MSX performs automatically when turned on.

8.1.1 – Clock-IC functions

Clock

- Read and update data for the year, month, day of the month, day of the week, hours, minutes and seconds;
- Time display in 12 or 24 hours;
- Months of 30 and 31 days are recognized. The month of February (28 days) and leap years are also recognized.

Alarm

- When active, the clock generates a signal at the chosen time;
- The alarm is set to "XXday, XXhours, XXminutes".

Memory

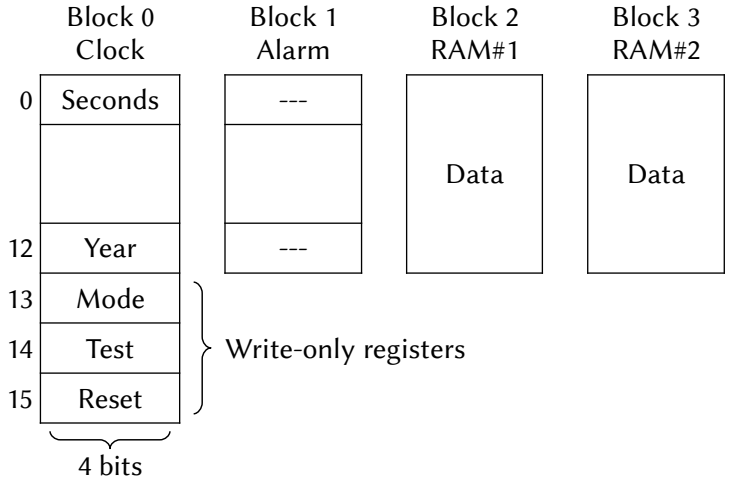
- Screen adjustment (set adjust);
- Initial SCREEN, WIDTH and COLOR values;
- Volume and tone of the beep;
- Home screen color;
- Country code;
- Password, BASIC prompt or splash screen title.

8.1.2 – Clock-IC structure and registers

The CLOCK-IC has four memory blocks, each one consisting of 13 registers of 4 bits each, addressed from 0 to 12. It also has three more

registers of 4 bits, for the selection of blocks and control of the functions, being accessed by the addresses 13 to 15.

Block registers (#0 to #12) and mode register (#13) can be read or written. The test (#14) and reset (#15) registers can only be written. They are illustrated below.



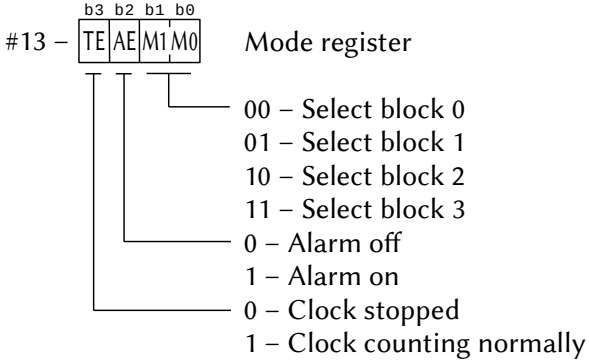
8.1.2.1 – The mode register (#13)

The mode register has three functions.

The first is block selection. The four blocks of 13 registers of 4 bits each (addressed #0 to #12) are selected by the lower two bits of the mode register. Registers #13 to #15 are accessed regardless of the selected block.

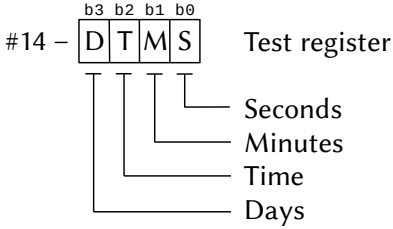
The second function is to turn the alarm output on or off. Bit 2 of the mode register is used for this. But MSX2 standard does not support the alarm function, and changing this bit has no effect.

The third function is the clock stop. By writing 0 to bit 3 of the mode register, the counting of seconds is interrupted and the clock function is stopped. By setting bit 3 to 1, counting resumes.



8.1.2.2 – The test register (#14)

The test register (#14) is used to quickly increment and confirm the clock's date and time. By setting each bit of this register to 1, pulses of 16 384 Hz are entered directly into the day, hour, minute, and second registers.



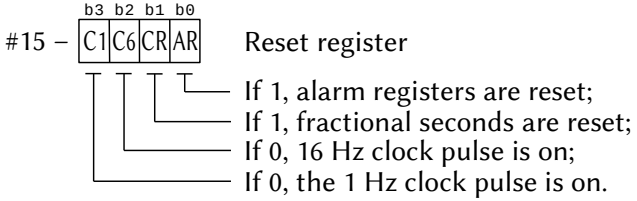
8.1.2.3 – The reset register (#15)

This register has three functions.

To reset the alarm, simply set bit 0 to 1 and all alarm registers will be reset.

Bit 1, when set to 1, resets the fractional seconds counter. This function is useful for setting the seconds correctly.

Setting bit 2 to 0 activates the 16 Hz clock pulse and setting bit 3 to 0 activates the 1 Hz clock.



8.1.2.4 – Setting the clock and alarm

Block 0 of memory is used for the clock. To set the date and time, this block must be selected and the data written to the correct registers.

Block 1 is used for the alarm. In this case, only days, hours and minutes can be set.

In the clock, the year is represented by 2 digits only (registers #11 and #12. To obtain the correct year, add 80 or 1980 to this value. For example, if these registers are 0, the correct year will be 1980.

The day of the week is represented by a value ranging from 0 to 6 in register #6. See the illustration below. The bits indicated with "•" must always be 0.

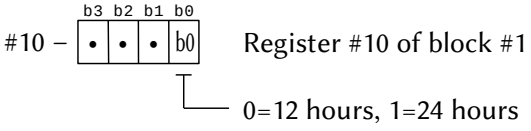
Block #0 – Clock

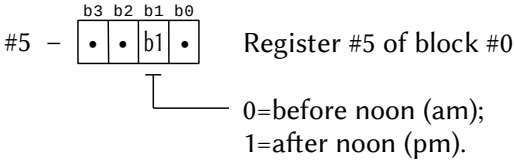
	Bits →	3	2	1	0
0	Sec. 1st digit	x	x	x	x
1	Sec. 2nd digit	•	x	x	x
2	Min. 1st digit	x	x	x	x
3	Min. 2nd digit	•	x	x	x
4	Hour 1st digit	x	x	x	x
5	Hour 2nd digit	•	•	x	x
6	Week day	•	x	x	x
7	Day 1st digit	x	x	x	x
8	Day 2nd digit	•	•	x	x
9	Month 1st digit	x	x	x	x
10	Month 2nd digit	•	•	x	x
11	Year 1º digit	x	x	x	x
12	Year 2º digit	•	v	x	x

Block #1 – Alarm

	Bits →	3	2	1	0
0	- - -	•	•	•	•
1	- - -	•	•	•	•
2	Min. 1st digit	x	x	x	x
3	Min. 2nd digit	•	x	x	x
4	Hour 1st digit	x	x	x	x
5	Hour 2nd digit	•	•	x	x
6	Week day	•	x	x	x
7	Day 1st digit	x	x	x	x
8	Day 2nd digit	•	•	x	x
9	- - -	•	•	•	•
10	12/24 hours	•	•	•	x
11	Leap year	•	•	x	x
12	- - -	•	•	•	•

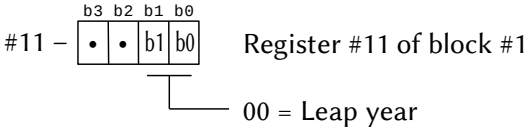
Two modes can be selected for the hour count: 12 hours or 24 hours. In 24-hour mode, when it is 1 pm, the watch will indicate 1:00 pm and in 12-hour mode, it will indicate 1:00 pm. Register #10 of block 1 is used for this function.





The am/pm flag in register #5 of block 0 can only be used in case of selection of 12 hours by register #10 of block 1.

Register #11 of block 1 is a counter of 4 (0 to 3) incremented each year. When the lower two bits of this register are 0, the year is considered a leap year and 29 days are counted for the month of february. The reference for this counter is the year 1980, which was a leap year.



8.1.2.5 –Content of the additional SRAM

CLOCK-IC SRAM blocks 2 and 3 have no clock function. In MSX, they are used to store some data that the computer recognizes when turned on to automatically execute some functions based on this data.

Block #2 content

	bit 3	bit 2	bit 1	bit 0
0	ID			
1	Horizontal adjust (–8 ~ +7)			
2	Vertical adjust (–8 ~ +7)			
3	•	•	Interlace	Screen
4	Initial screen width – low			
5	Initial screen width – high			
6	Initial front color code			
7	Initial back color code			
8	Initial border color code			
9	CAS veloc.	Printer	Key-click	Key on/off
10	Beep tone		Beep volume	
11	•	•	Splash screen color	
12	Native code			

Block 3 can have three different functions, depending on the contents of the ID position (register #0 of block 3). If the ID is equal to 0, the micro will present a title of up to 6 characters on the home screen. If it is equal to 1, block 3 will store a password of up to 6 characters that must be entered when turning on the computer so that it can be accessed. If the ID is equal to 2, a new prompt will be stored for BASIC, in place of "Ok", which can also be up to 6 characters long.

The organization of block 3 for these functions is illustrated below.

ID = 0 ⟶ Displays a title on the home screen:

Block #3

	bit 3	bit 2	bit 1	bit 0
0	0 ⟶ Title			
1	1st char (low)			
2	1st char (high)			
⋮	⋮			
11	6th char (low)			
12	6th char (high)			

ID = 1 ⟶ Set the password

Block #3

	bit 3	bit 2	bit 1	bit 0
0	1 ⟶ Password			
1	Usage ID=1			
2	Usage ID=1			
3	Usage ID=1			
4	Password	Password data is stored compressed in 4 x 4 bits		
5	Password			
6	Password			
7	Password			
8	Key cartridge flag			
9	Key cartridge value			
10	Key cartridge value			
11	Key cartridge value			
12	Key cartridge value			

ID = 2 ⟶ Sets the prompt on BASIC

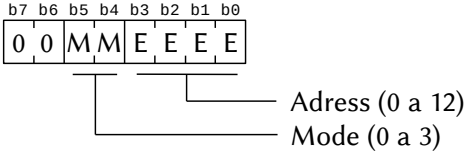
Block #3

	bit 3	bit 2	bit 1	bit 0
0	2 ⟶ BASIC prompt			
1	1st char (low)			
2	1st char (high)			
⋮	⋮			
11	6th char (low)			
12	6th char (high)			

8.1.3 – Access to clock-IC

The access to the clock and the battery-powered memory is done through two routines of the Sub-ROM BIOS, requiring the use of an inter-slot call to access them. Two routines are available:

REDCLK (01F5H / SubROM)
Function: Reads a nibble of data from the clock memory (Clock-IC).
Input: C – SRAM address of the clock, as shown below:



Output: A – Nibble read (4 bits lower).
Registers: AF.

WRTCLK (01F9H / SubROM)
Function: Writes a data nibble to the clock's memory
Input: C – SRAM address of the clock (equal to REDCLK).
A – Nibble to be written (4 bits lower).
Output: None.
Registers: F.

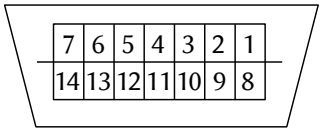
The routine below came as an example in the MSX2 Technhical Handbook.

```
; ----------------
; Set BASIC prompt
; ----------------
;
WRTCLK: EQU 01F9H
EXTROM: EQU 015FH

       ORG   0B000H

; Program start          ;Note: Set prompt message
                         ;      for BASIC.

START:
    LD   C,00110000B     ; Adress data
    LD   A,2             ; ID := prompt mode
    CALL WRTRAM          ; Write do back-up RAM
    LD   B,6             ; Loop counter
    LD   HL,STRING       ; Prompt data
L01:
    LD   A,(HL)          ; Read string data
```

```
    AND  0FH            ; A := high 4 bits
    INC  C              ; Increment adress
    CALL WRTRAM         ; Write data to back-up RAM
    LD   A,(HL)
    RRCA
    RRCA
    RRCA
    RRCA
    AND  0FH
    INC  C              ; Increment adress
    CALL WRTRAM         ; Write low 4 bits
    INC  HL
    DJNZ L01
    RET

;--- Write data to back-up RAM ---

WRTRAM:
    PUSH HL
    PUSH BC
    LD   IX,WRTCLK
    CALL EXTROM         ; Inter-slot call
    POP  BC
    POP  HL
    RET

;--- Data string ---

STRING:
    DB   'Ready?'

    END
```

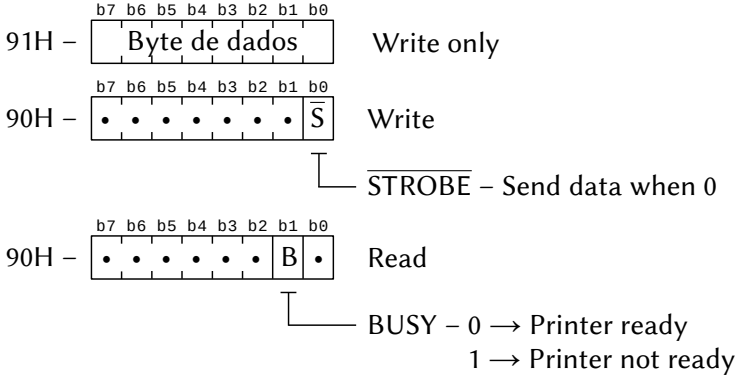
8.2 – PRINTER INTERFACE

This section describes how to access the printer through the BIOS and directly through the I/O ports. The printer interface is supported by BIOS, BASIC and DOS. MSX uses parallel port for printer access. The adopted standard is Centronics. The standard connector is also defined (Amphenol 14 contacts with female connector on the micro).



1 – $\overline{\text{STROBE}}$
2 ~ 9 – DATA (b0~b7)
11 – BUSY
14 – GND

The folowing I/O ports are used:



The data to be sent to the printer depends on whether it was specially developed for the MSX standard or not.

On a standard MSX printer, all characters that are output on the video can be printed. Special graphic characters of code 01H to 1FH can also be printed by sending the graphic header 01H followed by the character code + 40H.

Line wrapping on a standard MSX printer is done by sending the control characters 0DH and 0AH.

MSX has a function to transform the TAB code (09H) to the proper number of spaces on printers that do not have the TAB function. This is done via a flag in the system variables area:

RAWPRT (F41FH,1) – Replaces TAB with spaces when it is 0; otherwise, do not replace.

8.2.1 – Printer access

The printer can be accessed either directly or through BIOS routines. Access should preferably be done through BIOS routines to pre-

vent incompatibility and synchronization problems. The BIOS routines dedicated to the printer are as follows:

LPTOUT (00A5H / Main)
Function: Send a character to the printer.
Input: A – ASCII code of the character to be sent.
Output: If it fails, CY returns set.
Registers: F.

LPTSTT (00A8H / Main)
Function: Tests the printer status.
Input: None.
Output: A = 255 (and Z flag = 0) ⟶ printer ready.
A = 0 (and Z flag = 1) ⟶ printer is not ready.
Registers: AF.

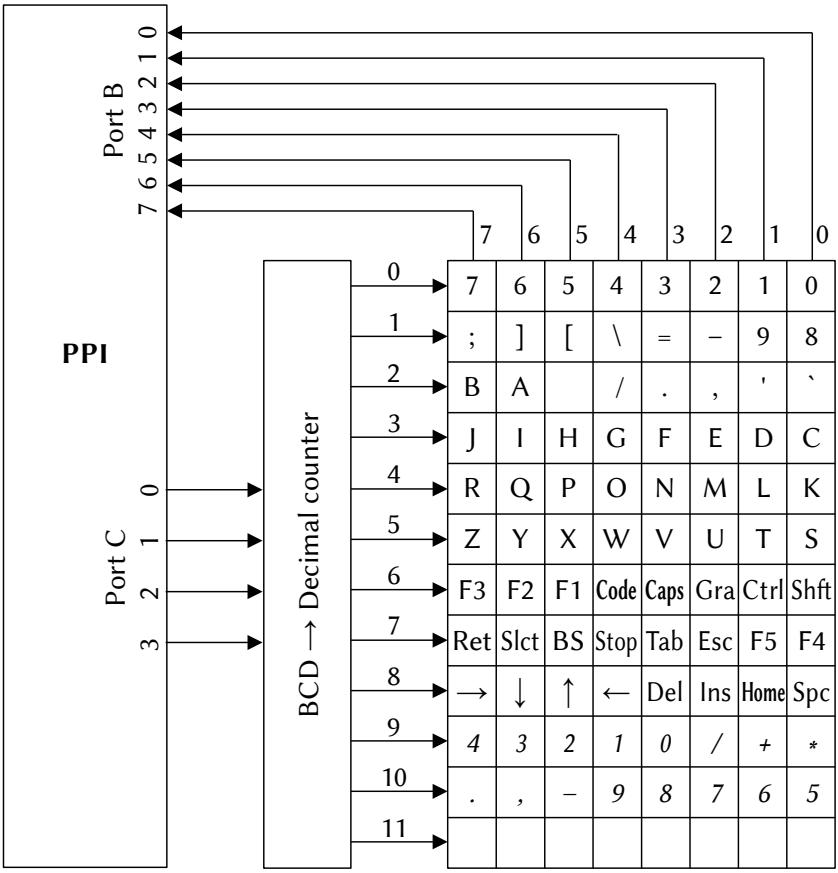
OUTDLP (014DH / Main)
Function: Formatted output for the printer. It differs from LPTOUT in the following points:
- If the character sent is a TAB (09H) spaces will be sent until reaching a multiple of 8;
- For non-MSX printers, hiraganas are converted to katakanas and graphic characters are converted to 1-byte characters;
- If there is a failure, an I/O error will occur.

Input: A – Character to be sent.
Output: None.
Registers: F.

8.3 – KEYBOARD INTERFACE

The keyboard interface is controlled by PPI ports B and C. The lower 4 bits of port C send a value from 0 to 10 corresponding to the row of the keyboard matrix to be read, and port B of the PPI reads the state of the keys. A read 0 bit indicates pressed key. As there are 11 lines and 8 bits in each line, the keyboard can have a maximum of 88 keys (11 * 8).

The international keyboard matrix is illustrated below.



Lines '9' and '10' correspond to the independent numeric keypad, when the computer has one. The '11' line is used in only some Japanese models to access the programs that come in the internal ROM.

The keyboard matrix is different for each country to suit each country's local language characters. Several of them are detailed in Appendix, Chapter 1, Section 1.2 – Keyboard Matrices.

8.3.1 – Keyboard access

Keyboard access can be done either directly, by accessing PPI ports B and C (A9H for port B and AAH for port C) or through the BIOS routine SNSMAT (0141H/Main). Because it is a slow peripheral, BIOS ac-

cess is preferable to direct access. The SNSMAT routine is described below.

SNSMAT (0141H / Main)
Function: Reads the value of a line from the keyboard matrix.
Input: A – Line to be read.
Output: A – Value read (the bit corresponding to a key pressed is 0).
Registers: AF, C.

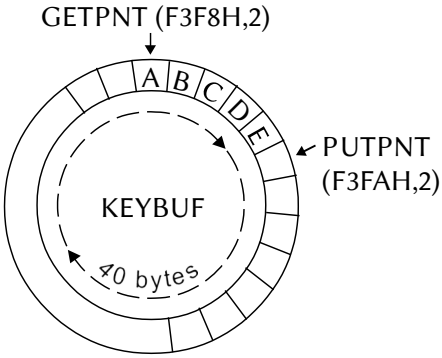
Other keyboard interface related routines are as follows:

CHSNS (009CH/Main) - Checks the status of the keyboard buffer.
CHGET (009FH/Main) - Input a character from the keyboard.
KILBUF (0156H/Main) - Clears the keyboard buffer.
CNVCHR (00ABH/Main) - Converts graphic character.
PINLIN (00AEH/Main) - Keypad line input.
INLIN (00B1H/Main) - Prompt keyboard line input.
INIFNK (003EH/Main) - Initializes contents of function keys.
BREAKX (00B7H/Main) - Detects CTRL+STOP keys.

The detailed description of these routines can be seen in the appendix, chapter 8, section 8.1 – MainROM routines.

8.3.2 – Keyboard scan

MSX automatically scans the entire keyboard array every three interrupt cycles, ie 20 times per second on a PAL machine (60 Hz), as long as interrupts are enabled. When it finds a key pressed, it is stored in a 40-byte circular buffer. This buffer is designated KEYBUF (FBF0H~FC17H) and works as illustrated below.

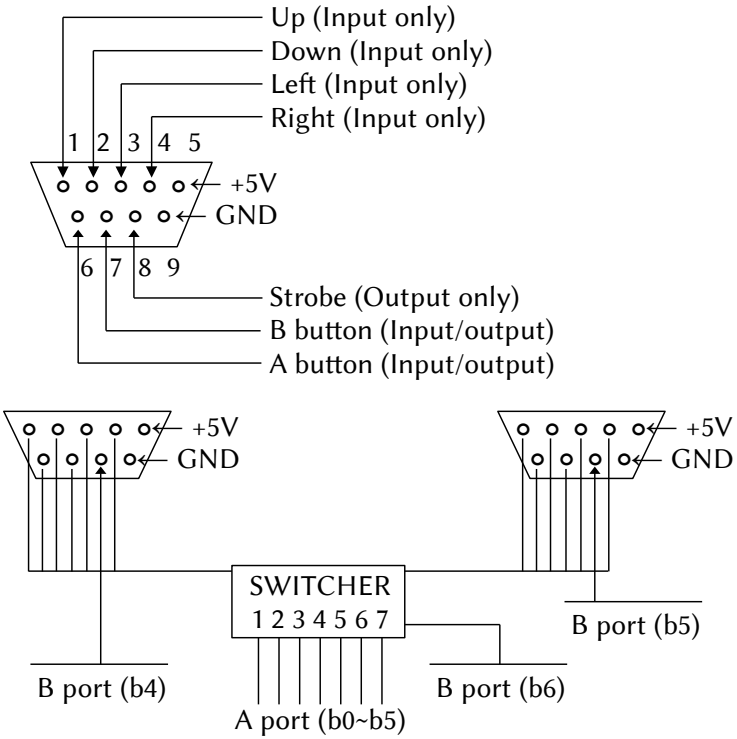


GETPNT points to the next character to be retrieved by the CHGET routine, and PUTPNT points to the next free position in the buffer, to be filled with the value of the next key pressed.

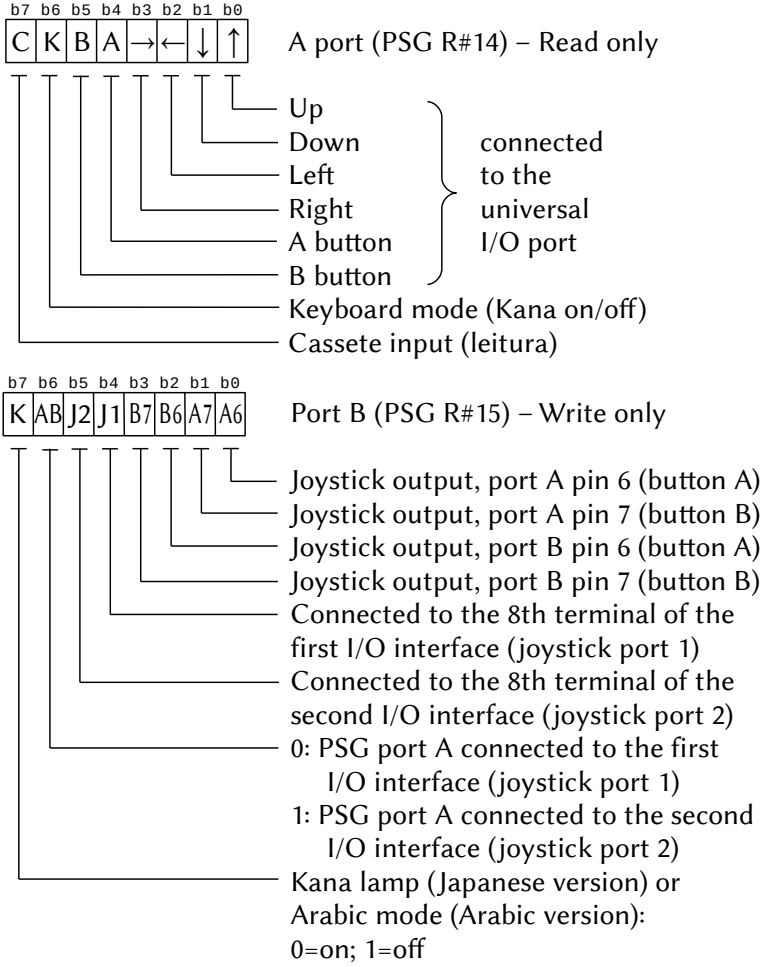
8.4 – UNIVERSAL I/O INTERFACE

As described in Chapter 5, the PSG has two general-purpose I/O ports. These ports are connected to the universal I/O interface (joystick ports). Various devices, in addition to the joystick, can be connected to this port, such as a mouse or paddles. For ease of access, there are some routines in the BIOS that support these ports.

These interfaces are wired as illustrated below.



The two PSG ports are used as described below:



Bits b0~b3 of port B are wired to pins 6 and 7 of each joystick connector via open collector TTL buffers in order to be able to send signals to the port. The pulse bits (bit 4 – J1 and bit 5 – J2) send a short pulse through pin 8. If bit 4 is set to 0, nothing is sent. If it is at 1, the pulse is sent. Bits 0 and 1 indicate which pin should be signaled high. Bit 0 corresponds to pin 6 and bit 1 to pin 7. Both can send data at the same time. According to the table below, the different possibilities of sending signal through the joystick port can be observed.

R#15 (Port B)								Pin 6	Pin 7	Pin 8
b7	b6	b5	b4	b3	b2	b1	b0	A button	B button	Strobe
x	x	0	0	1	1	1	1	0	0	Vcc
x	x	0	1	1	1	1	1	0	0	Pulse
x	x	0	0	1	1	1	0	Vcc	0	Vcc
x	x	0	1	1	1	1	0	Vcc	0	Pulse
x	x	0	0	1	1	0	1	0	Vcc	Vcc
x	x	0	1	1	1	0	1	0	Vcc	Pulse
x	x	0	0	1	1	0	0	Vcc	Vcc	Vcc
x	x	0	1	1	1	0	0	Vcc	Vcc	Pulse

The value xx001111 is always returned to register 15 after each operation. Therefore, to keep any pin connected (6 or 7), it is necessary to create a "loop" that constantly adjusts register 15 to the desired value, according to the table above. Pin 8, unlike pins 6 and 7, keeps the voltage level high.

Access to the universal I/O interface should preferably be done through the BIOS routines described below:

GTSTCK (005DH/Main) - Reads joystick status.
GTTRIG (00D8H/Main) - Reads fire button status.
GTPDL (00DEH/Main) - Reads information from the paddle.
GTPAD (00DBH/Main) - Accesses various I/O devices.

The detailed description of these routines can be seen in the Appendix, chapter 9 – BIOS routines, section 9.1.6 – I/O access routines for games.

Chapter 9
THE MSX TURBO R

In MSX turbo R models, a 16-bit CPU was introduced, fully compatible with the Z80 at the instruction level. The R800 CPU is built on an LSI chip with a 100-terminal QFP package. The R800's internal clock is 7.16 MHz. It also has 2 DMA channels and output for multitasking, but those options were not used.

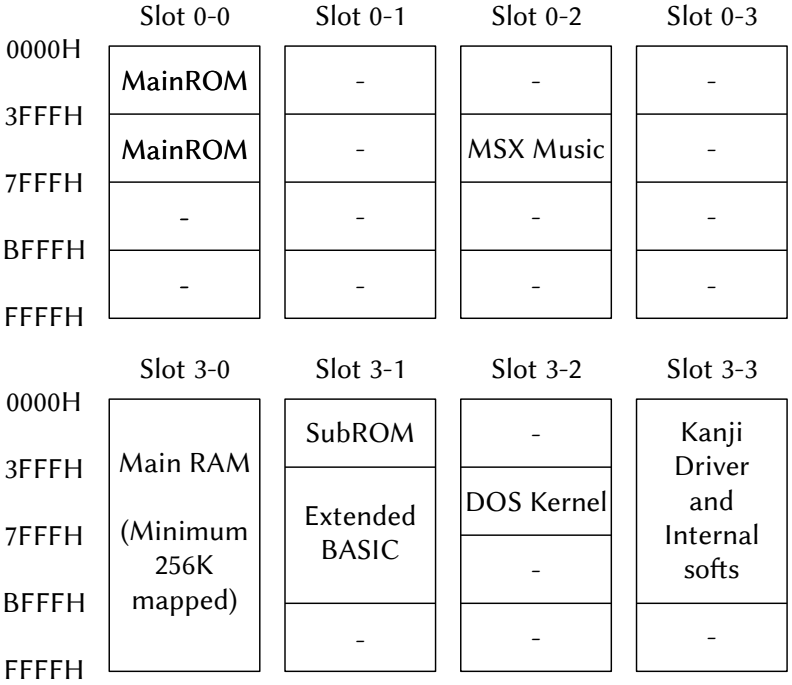
The R800 instruction set encompasses all the Z80 instructions and adds a few more, such as 8-bit and 16-bit forward multiplication and makes official some "secret" Z80 instructions: treating the IX and IY index registers as two 8-bit registers each, called .ixl, .ixh, .iyl, and .iyh.

As the R800 is fully compatible with the Z80 at the instruction level, it is possible to make a program that works from the MSX2 and put a small routine that detects if the program is resident in an MSX turbo R, and, in this case, activate the R800 to accelerate, on average, 7 times the execution of the program. Some care, however, must be taken. In the case of direct access via I/O ports, even to some internal components, such as the OPLL, the R800 mode must be turned off, as there will be desynchronization due to the higher speed of the R800.

In the case of accessing the VDP there is no problem, as another chip specific to the MSX turbo R (the S1990) adjusts the timing by hardware when necessary. In the case of memory mapped there is also no problem. Any other direct access, however, must be done in Z80 mode, to prevent synchronization problems. In case of access via BIOS, BDOS or BASIC there is no problem, as the BIOS compensates for timing differences when necessary.

9.1 – ORGANIZATION OF SLOTS AND PAGES

In MSX turbo R, the organization of slots and pages has been standardized, and the RAM is connected directly to the R800, allowing for a shorter access time. This also simplifies specific software development. Primary slots 0 and 3 are reserved for the system and slots 1 and 2 are external user slots. Slots 0 and 3 are expanded and their arrangement is as follows:



Note that the segment corresponding to the DOS Kernel has 4 segments of 16 Kbytes that are swapped exclusively on page 1. The first three segments are for MSXDOS2 and the last one for MSXDOS1.

9.2 – WAIT STATES

The wait states (wait cycles) in MSX turbo R are generated under some special conditions.

When an external slot is accessed, 3 wait states are generated. This is necessary to allow all hardware developed before the turbo R to work properly, as the higher speed of the R800 mode could make such peripherals unfeasible.

When the internal ROM is accessed, 2 wait states are generated, due to the relative slowness of the ROM chips.

When the internal DRAM is accessed, 1 wait state is generated. Therefore, access is faster in DRAM than in ROM.

9.3 – MODES OF OPERATION

As the MSX turbo R has 2 CPUs, there are some operating modes involving these CPUs. They can be freely switched during processing, but cannot be activated simultaneously. Two specific combinations between DOS and CPUs are recommended: Z80/ROM and R800/DRAM, but nothing prevents the R800 from working in ROM mode and the Z80 in DRAM mode. When the system boots, it checks the boot disk to enter the correct mode. If there is no disk, the system will automatically enter R800 DRAM mode with MSXDOS2 Kernel, unless the "1" key is pressed during reset, which forces the system to enter Z80 mode.

Note that there are two modes of operation for the R800: ROM and DRAM. In ROM mode, all mapped memory is free for use. In DRAM mode, the system transfers the contents of Main ROM (32K), Sub ROM (16K) and the first part of the Kanji Driver to the last four pages of the mapped memory. The advantage of this is that BIOS routines are processed faster, since ROM is much slower than DRAM. In view of this, there is a loss of 64 Kbytes of available RAM. However, if the program being executed makes many accesses to the BIOS, the loss of memory in exchange for the gain in speed can be advantageous. This must be decided during software development. The 64 Kbytes reserved in DRAM mode are always on the highest logical pages of mapped memory and cannot be written to, despite being RAM. A routine template that can be included in programs to make them use the R800 speed is illustrated below.

```
RDSLT:  EQU 0000CH
CALSLT: EQU 0001CH
CHGCPU: EQU 00180H
SLTROM: EQU 0FCC1H
;
;--- CHECKS VERSION ---
;
LD A,(SLTROM)
LD HL,0002DH
CALL RDSLT
CP A,2
JR C,NAOTUR
;
```

```
;--- PREPARE MODE CHANGE ---
;--- (CHOICE -ONLY- ONE OPTION) ---
;
;Z80 MODE
LD A,11001110B
AND 002H
XOR 082H
;
;R800 ROM MODE
LD A,01000100B
AND 002H
XOR 081H
;
;R800 DRAM MODE
LD A,11001101B
AND 002H
XOR 082H
;
;--- SWAP MODE ---
;
LD IY,(SLTROM-1)
LD IX,CHGCPU
CALL CALSLT
;
NAOTUR:
END
```

The routine presented makes a test to verify if it is running on an MSX turbo R or not. If not, it jumps to the label NAOTUR (terminate), but if it is, it calls the BIOS routine CHGCPU, which switches the processors according to the value passed in the A register. This routine works both under DOS and under BASIC, in any address.

9.3.1 – Speed comparison

The table below shows the speed gain when using the R800 in place of the Z80.

Instructions		Z80(µs)	R800(µs)	Gain
LD	r,s	1.40	0.14	x 10.0
LD	r,(HL)	2.23	0.42	x 5.3
LD	r,(IX+n)	5.87	0.70	x 8.4
PUSH	qq	3.35	0.56	x 6.0
LDIR	(BC<>0)	6.43	0.98	x 6.6
ADD	A,r	1.40	0.14	x 10.0
INC	r	1.40	0.14	x 10.0
ADD	HL,ss	3.35	0.14	x 24.0
INC	ss	1.96	0.14	x 14.0
JP		3.07	0.42	x 7.3
JR		3.63	0.42	x 8.7
DJNZ	(B<>0)	3.91	0.42	x 9.3
CALL		5.03	0.84	x 6.0
RET		3.07	0.56	x 5.5
MULUB	A,r	160	1.96	x 81.6
MULUW	HL,rr	361	5.03	x 71.7

The speed gain over the Z80 is very large, reaching an average of 7 times. The MULUB and MULUW instructions (multiplication of 8-bit and 16-bit operands, respectively) are exclusive to the R800 and do not exist in the Z80. To obtain the time in microseconds, routines optimized for the Z80 were used.

9.3.2 – R800 specific intructions

The R800 made some secret Z80 instructions official and added two multiplication instructions. Are the following:

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
LD u,u'	u ← u'	• • • • • •	11 011 101 01 u u'	DD --	08	10	02	02
LD v,v'	v ← v'	• • • • • •	11 111 101 01 v v'	FD --	08	10	02	02
LD u,n	u ← n	• • • • • •	11 011 101 00 u 110 ← n →	DD -- --	11	13	03	03

LD v,n	u ← n	• • • • • •	11 111 101 00 v 110 ← n →	FD -- --	11	13	03	03
ADD p	A ← A + p	↕ ↕ V ↕ 0 ↕	11 011 101 10 000 r	DD --	08	10	02	02
ADD q	A ← A + q	↕ ↕ V ↕ 0 ↕	11 111 101 10 000 r	FD --	08	10	02	02
ADC p	A ← A+p+CY	↕ ↕ V ↕ 0 ↕	11 011 101 10 001 r	DD --	08	10	02	02
ADC q	A ← A+q+CY	↕ ↕ V ↕ 0 ↕	11 111 101 10 001 r	FD --	08	10	02	02
SUB p	A ← A − p	↕ ↕ V ↕ 1 ↕	11 011 101 10 010 r	DD --	08	10	02	02
SUB p	A ← A - q	↕ ↕ V ↕ 1 ↕	11 111 101 10 010 r	FD --	08	10	02	02
SBC p	A ← A-p-CY	↕ ↕ V ↕ 1 ↕	11 011 101 10 011 r	DD --	08	10	02	02
SBC p	A ← A-q-CY	↕ ↕ V ↕ 1 ↕	11 111 101 10 011 r	FD --	08	10	02	02
AND p	A ← A ∧ p	0 ↕ P ↕ 0 1	11 011 101 10 100 p	DD --	08	10	02	02
AND q	A ← A ∧ q	0 ↕ P ↕ 0 1	11 111 101 10 100 p	FD --	08	10	02	02
OR p	A ← A ∨ p	0 ↕ P ↕ 0 1	11 011 101 10 110 p	DD --	08	10	02	02
OR q	A ← A ∨ q	0 ↕ P ↕ 0 1	11 111 101 10 110 p	FD --	08	10	02	02
XOR p	A ← A ⊕ p	0 ↕ P ↕ 0 1	11 011 101 10 110 p	DD --	08	10	02	02
XOR q	A ← A ⊕ q	0 ↕ P ↕ 0 1	11 111 101 10 110 p	FD --	08	10	02	02
CP A,p	A − p	↕ ↕ V ↕ 1 ↕	11 011 101 10 111 p	DD --	08	10	02	02
CP A,q	A − q	↕ ↕ V ↕ 1 ↕	11 111 101 10 111 p	FD --	08	10	02	02
MULUB r	HL ← A * r	↕ ↕ 0 0 • •	11 101 101 11 r 001	ED --	--	--	14	14
MULUW HL,tt	DE:HL ← HL * tt	↕ ↕ 0 0 • •	11 101 101 11 tt0 011	ED --	--	--	36	36

Registers conventions:

	00	01	10	11	100	101
p	•	•	•	•	IXH	IXL
q	•	•	•	•	IYH	IYL
rr	BC	DE	IY	SP	•	•
tt	BC	•	•	SP	•	•

```
TZ - Z80 T cycles
Z1 - Z80 + M1
TR - R800 T cycles
RW - R800 + Wait
```

The last two instructions ('mulub' and 'muluw') do not exist on the Z80, being unique to the R800.

9.4 – MSX-MIDI

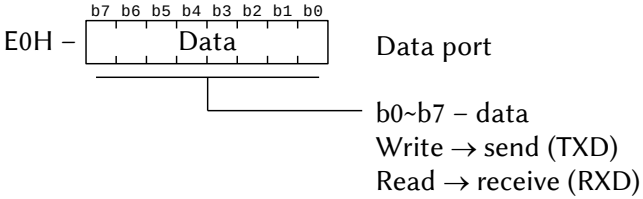
From the second MSX turbo R model, MSX-MIDI was standardized. MIDI stands for "Musical Instruments Digital Interface". With it you can control musical instruments that have MIDI input.

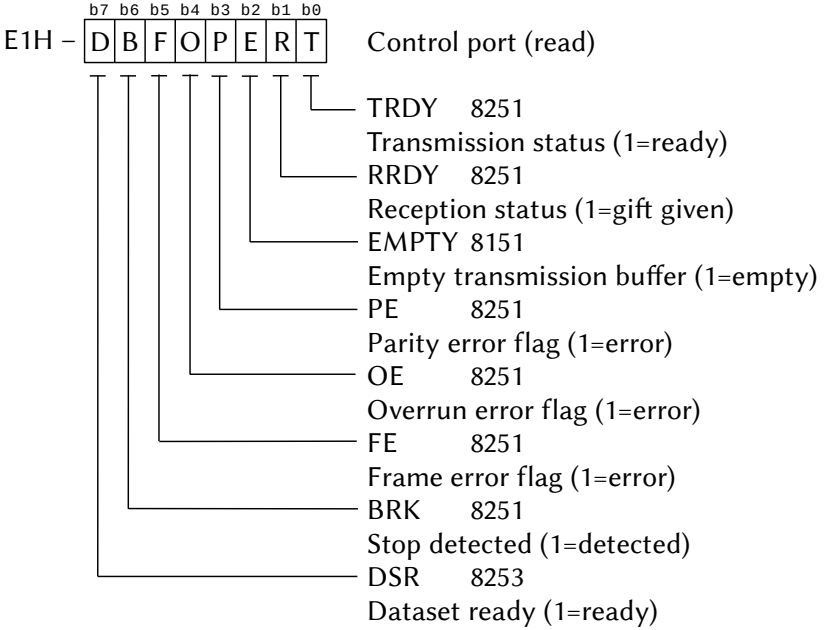
9.4.1 – Access to MSX-MIDI

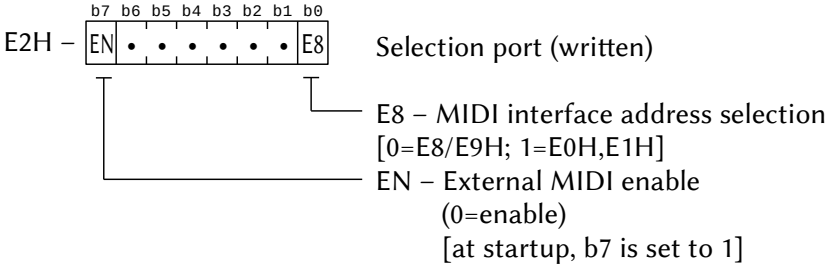
MSX-MIDI is directly controlled by I/O ports. The reserved ports are E8H to EFH when MIDI is internal and three more if MIDI is external: E0H to E2H. They are described below:

E0H - Data transmission / reception (external interface).
E1H - Control port (external interface).
E2H - Selection port.
E8H - Data transmission/reception
E9H - Control port.
EAH - Signals Latch (write only)
EBH - EAH Mirror.
ECH - Counter 0.
EDH - Counter 1.
EEH - Counter 2.
EFH - Control of counters (write only).

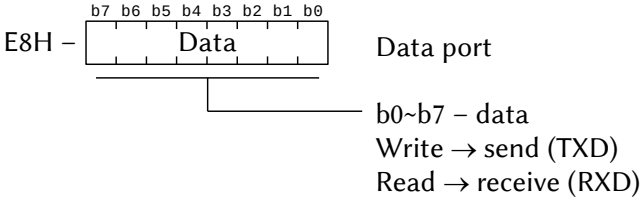
9.4.2 – Description of external MIDI ports







9.4.3 – Description of intenal MIDI ports



E9H –

b7	b6	b5	b4	b3	b2	b1	b0
D	B	F	O	P	E	R	T

Control port (read)

* Organization identical to the E1H port for reading. For writing, the organization is as follows:

E9H –

b7	b6	b5	b4	b3	b2	b1	b0
E	I	R	E	S	R	T	T

Control port (write)

Mode: b7=S2; b6=S1; b5=EP; b4=PEN;
b3=L2; b2=L1; b1=B2; b0=B1.

TEN Enable MIDI OUT transmission
(1 = on, 0 = off)

TIE timer 8253 (counter #2)
(1 = enable, 0 = disable)

RE Enable MIDI IN reception
(1 = on, 0 = off)

SBRK Normally '0'

ER Error Reset
(1 = reset error flag, 0 = no operation)

RIE Enable MIDI IN interrupt
(1 = on, 0 = off)

IR Normally '0'

EH Normally '0'

When writing data in command mode, a wait of 16 T cycles (3.58 MHz) is required for the result and for writing a sequence of commands.

EAH –

b7	b6	b5	b4	b3	b2	b1	b0
Data							

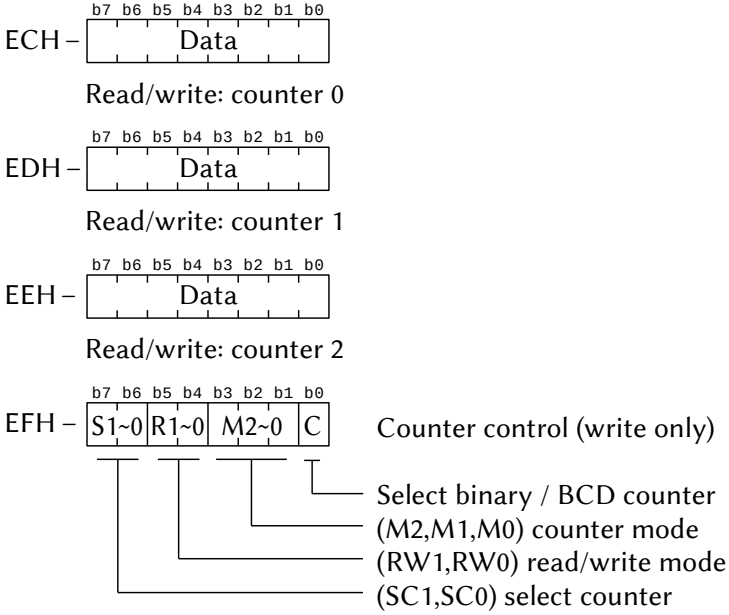
8253 data port

Write: 8253 OUT2 – terminal signals latch.
Read: no effect.

EBH –

b7	b6	b5	b4	b3	b2	b1	b0
Data							

This port is an image of EAH.



9.4.4 – Internal MIDI and external MIDI

MSX-MIDI can already come internal to MSX turbo R as well as can be implemented through cartridge, but only for MSX turbo R onwards. If MSX-MIDI is internal, bit 0 of ROM address 002EH will be set to 1.

The difference between internal or external MIDI can be obtained at address 4018H, as shown below:

Adress	Internal	External
4018H	41H(A)	??H(?)
4019H	50H(P)	??H(?)
401AH	52H(R)	??H(?)
401BH	4CH(L)	??H(?)
401CH	4FH(O)	4DH(M)
401DH	50H(P)	49H(I)
401EH	4CH(L)	44H(D)
401FH	4CH(L)	49H(I)

The MIDI interface also changes some hooks, and these are different depending on whether the MIDI is internal or external.

If the MIDI is internal, the redirected hooks will be:

Address	Old name	New name	New function
FF75H	HOKNO	HMDIN	MIDI IN
FF93H	HFRQI	HMDTM	8253 Timer

In the case of external MIDI, the above hooks cannot be used; in this case, use the HKEYI hook (FD9AH).

9.5 – TIMING FOR V9958

Although the V9958 is too slow for the R800, there is no timing issue in direct access to it because the MSX-Engine S1990 generates 8µs hardware pauses between consecutive accesses to the VDP. However, this is a relatively long time for the R800, corresponding to 57 T cycles.

It is possible to prevent the S1990 from generating pauses for the R800 when it accesses the VDP. It happens that the pause is only generated from the second access, if this is done before the counter returns to 0. It is then enough to time it by software, making the R800 execute some operations between consecutive accesses to the VDP. The operations performed must take a minimum of 57 T cycles, which causes the counter to return to 0 before the second access and avoids the generation of pauses.

9.6 – THE INTERNAL SRAM

The MSX turbo R has internally a small battery-powered SRAM, in addition to the clock. The FS-A1ST model has 16 Kbytes of SRAM and the FS-A1GT model has 32 Kbytes.

This SRAM is divided into segments of 8 Kbytes, which can be accessed exclusively in slot 3-3, the same where the Kanji-Driver and ROM software are. In fact, this same ROM is mapped in 192 segments of 8 Kbytes, in a total of 1.5 Mbytes. SRAM is mapped with segment numbers 128 through 131.

The procedure to disable the ROM and enable the SRAM in that slot is very simple: just write the SRAM segment number in one of the switch addresses, which are the following:

6000H – enable segment at 0000H~1FFFH.
6400H – enable segment at 2000H~3FFFH.
6800H – enable segment at 4000H~5FFFH.
6C00H – enable segment at 6000H~7FFFH.
7000H – enable segment at 8000H~9FFFH.
7400H – enable segment at A000H~BFFFH.
7800H – enable segment at C000H~DFFFH.
7C00H – enable segment at E000H~FFFFH.

Internal SRAM is used by ROM software to save their settings, but it can be used for many other purposes. However, some care is needed when handling data in the 6000H~7FFFH segment because it contains the switching addresses and the SRAM could be disabled or change the segment or addresses.

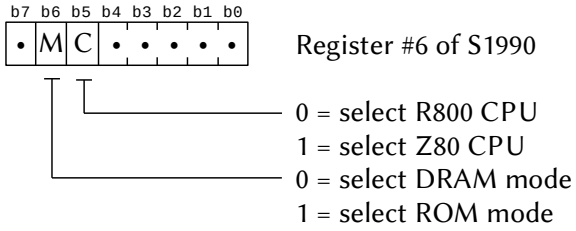
Internal SRAM is not compatible with the SRAM of PAC or FM-PAC cartridges.

9.7 – THE MSX ENGINE S1990

In MSX turbo R models, a chip was introduced to control its various functions, the S1990. This chip contains many internal registers, which are accessed by I/O ports E4H (register selection) and E5H (data). However, only registers 6, 14 and 15 have known functions so far.

Registers 14 and 15 are read-only. The S1990 stores the last byte written to memory in register 15 and the preceding byte in register 14.

Register 6 is used to control active CPU and ROM or DRAM mode. It is organized as illustrated below:



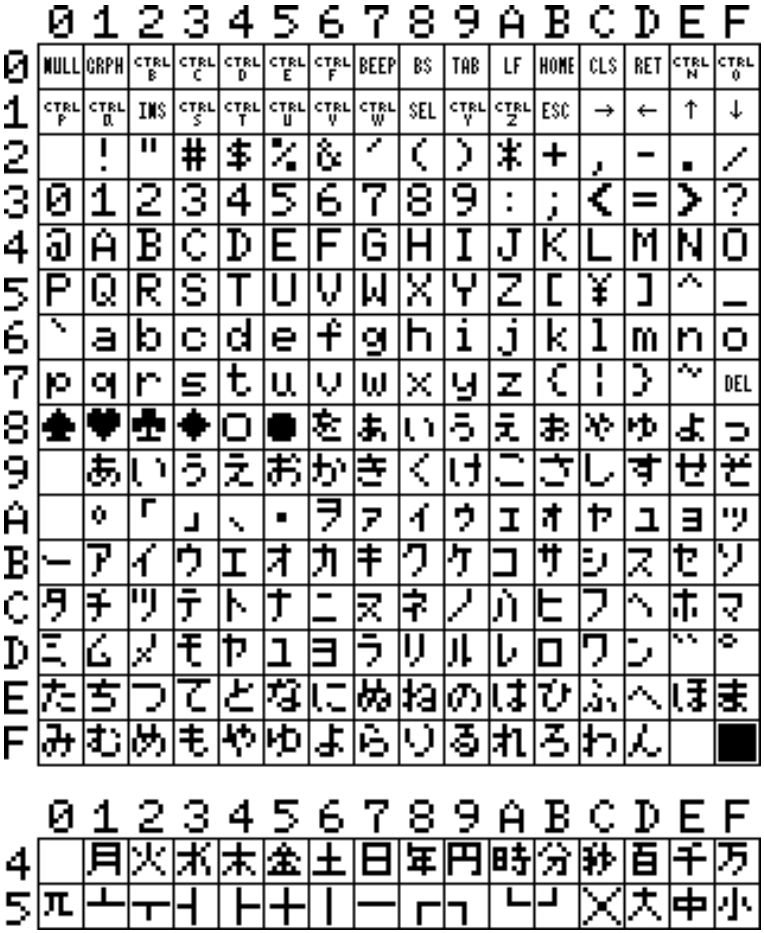
With this register, you can select Z80 DRAM mode. For this, first, the Z80 ROM mode must be selected through routine CHGCPU (0180H) and then the value 40H (64) must be written in register 6.

Appendix

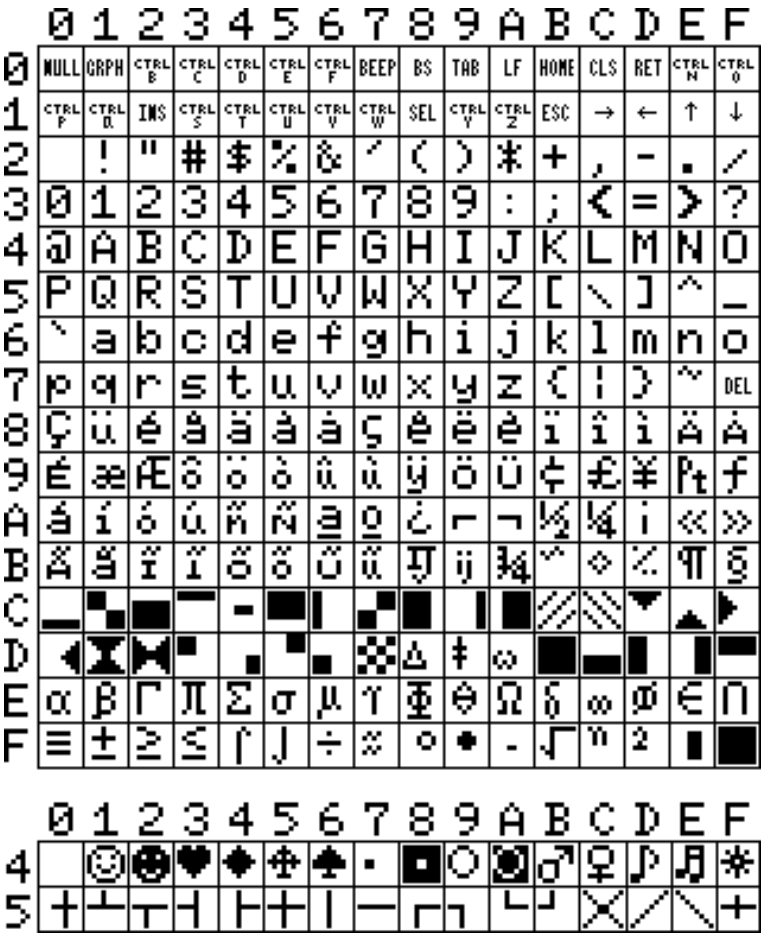
1 – CHARACTERS AND KEYBOARD

1.1 – CHARACTER SETS

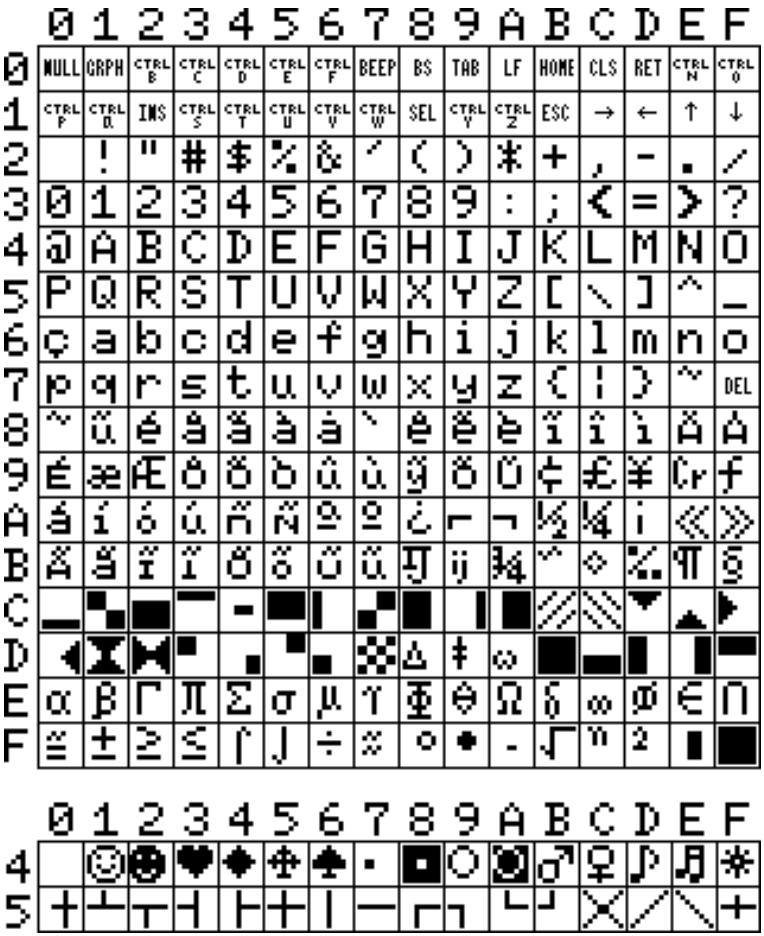
1.1.1 – Japanese Set



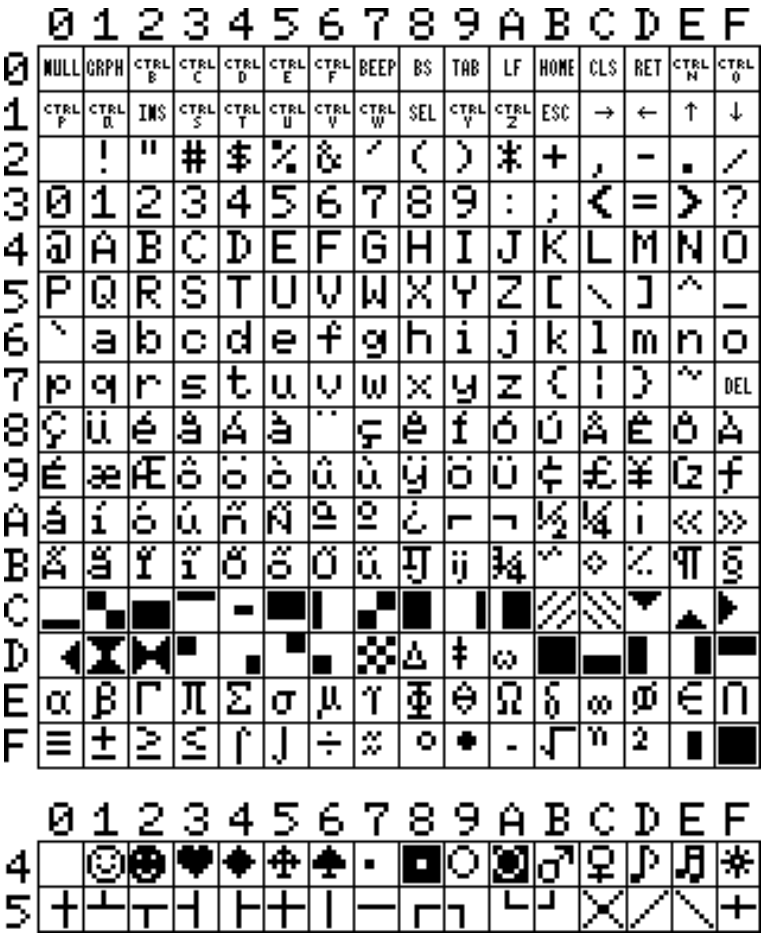
1.1.2 – Internacional Set



1.1.3 – Brazilian Set 1.0 (Expert 1.0)

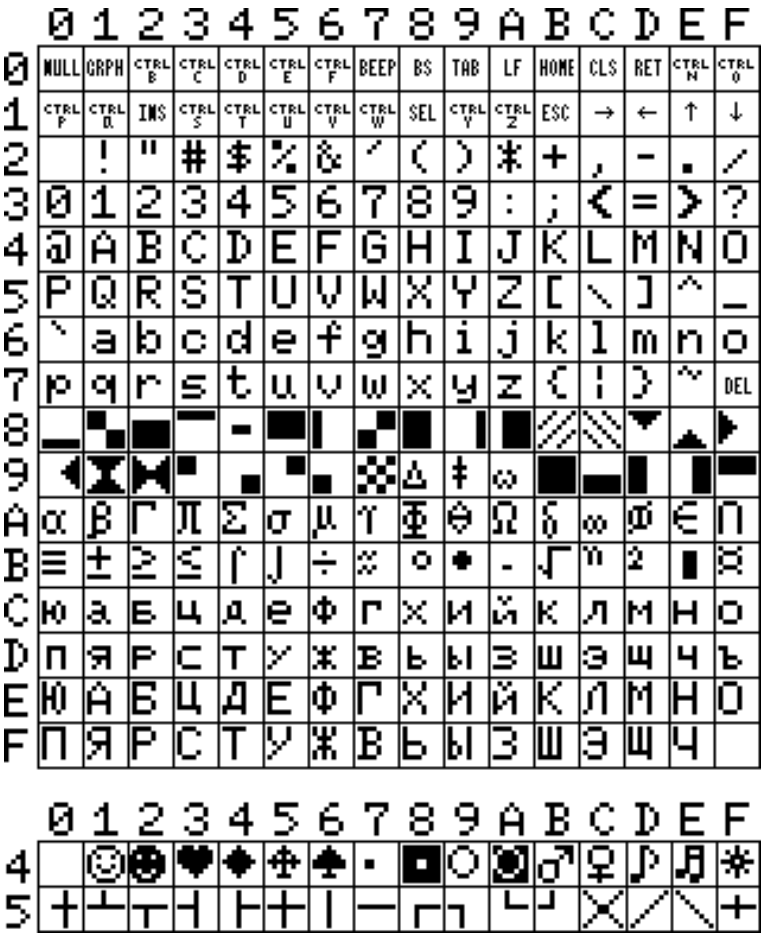


1.1.4 – Brazilian Set 1.1 (Expert 1.1 and Hotbit 1.2)



Obs.: The character in the 9EH position (Cz) was "Pt" in the first HOTBIT version.

1.1.5 – Russian Set



1.1.6 – Korean Set

1.1.7 – Arabic Set (AX-170)

1.1.8 – Arabic Set (AX-500)

1.2 – KEYBOARD MATRICES

1.2.1 – Japanese Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	; :	] }	[{	\ |	= +	- _	9 (	8 *
Col. 2	b B	a A	_	/ ?	. >	, <	' ~	` "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	かな	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5
Col. 11					実 行		取消	

Obs.1: Column 11 is used only by Panasonic models FS-A1WX, FS-A1WSX and turbo R, for access to the internal software in ROM. "実行" means "select" and "取消" means "cancel".

Obs.2: The "かな" position is the "KANA" key and corresponds to the CODE key in the international version.

1.2.1.1 – Japanese Matrix with locked かな/KANA key

JIS	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	や ゃ	お ぉ	え ぇ	う ぅ	あ ぁ	ふ	め	わ を
Col. 1	れ	゜「	゛	ー	へ	ほ	よ ょ	ゆ ゅ
Col. 2	こ	ち	ろ	め ･	る 。	ね 、	む 」	け
Col. 3	ま	に	く	き	は	い ぃ	し	そ
Col. 4	す	た	せ	ら	み	も	り	の
Col. 5	つ っ	ん	さ	て	ひ	な	か	と

ANSI	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	に	な	お ぉ	え ぇ	う ぅ	い ぃ	あ ぁ	の
Col. 1	も	ろ「	れ	る	り	ら	ね	ぬ
Col. 2	と	さ	ん ･	を 。	わ 、	よ ょ	゜」	゛ ー
Col. 3	み	ふ	ま	そ	せ	く	す	つ っ
Col. 4	け	か	ほ	へ	や ゃ	ゆ ゅ	め	む
Col. 5	た	は	ち	き	て	ひ	こ	し

GRAPH	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	土	金	木	水	火	月	日	万
Col. 1	♣	○		円		一	千	百
Col. 2	┘		♦	♠	大	小	●	♥
Col. 3		│	時	┤	┼	┌	├	└
Col. 4	┬		π			分	中	
Col. 5		年	╳		┴		┐	秒

1.2.2 – PX-7 Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	; :	] }	[{	\ |	= +	- _	9 (	8 *
Col. 2	b B	a A	゛ ¨ ヽ ^	/ ?	. >	, <	' ~	` "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	かな	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9						SupI	Video	Comp

SupI → **Superimpose**
Video → **Video**
Comp → **Computer**

Obs.: The PX-7 does not have a separate numeric keypad.

1.2.3 – Internacional Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	; :	] }	[{	\ |	= +	- _	9 (	8 *
Col. 2	b B	a A		/ ?	. >	, <	' ~	` "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

1.2.4 – Brazilian Matrix (Expert 1.1 and Hotbit)

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 "	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	ç Ç	¨ '	' `	\ ^	= +	- _	9 (	8 *
Col. 2	b B	a A	< >	/ ?	. :	, ;	[ ]	~ ^
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	4	3	2	1	0	/	+	*
Col. 10	.	,	-	9	8	7	6	5

Obs: The Expert 1.0 uses the international matrix.

1.2.5 – Argentine / Spanish Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	ñ Ñ	] }	[{	\ ¦	= +	- _	9 (	8 *
Col. 2	b B	a A		/ ?	. >	, <	; :	' "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

Obs.: Only columns 1 and 2 differ from the international.

1.2.6 – United Kingdom Matrix (England)

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	; :	] }	[{	\ |	= +	- _	9 (	8 *
Col. 2	b B	a A	£	/ ?	. >	, <	` ~	' "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

Obs.: Only column 2 differ from the international.

1.2.7 – Russian Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	& 6	% 5	¤ 4	# 3	" 2	! 1	+ ;	) 9
Col. 1	v V	* :	h H	- ^	= _	$ 0	(8	' 7
Col. 2	i I	f F	? /	< ,	@	b B	> .	\
Col. 3	o O	[}	r R	p P	a A	u U	w W	s S
Col. 4	k K	j J	z Z	] }	t T	x X	d D	l L
Col. 5	q Q	n N	| ~	c C	m M	g G	e E	y Y
Col. 6	F3	F2	F1	РУС	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

1.2.7.1 – Russian Matrix with locked РУС/CODE key

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	& 6	% 5	¤ 4	# 3	" 2	! 1	+ ;	) 9
Col. 1	ж Ж	* :	х Х	ъ Ъ	= _	$ 0	(8	' 7
Col. 2	и И	ф Ф	? /	< ,	ю Ю	б Б	> .	э Э
Col. 3	о О	ш Ш	р Р	п П	а А	у У	в В	с С
Col. 4	к К	й Й	з З	щ Щ	т Т	ь Ь	д Д	л Л
Col. 5	я Я	н Н	ч Ч	ц Ц	м М	г Г	е Е	ы Ы

1.2.8 – Korean Matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 '	6 &	5 %	4 $	3 #	2 "	1 !	0 0
Col. 1	; +	[{	@ `	₩ ¦	^ ~	- =	9)	8 (
Col. 2	b B	a A	_	/ ?	. >	, <	] }	: *
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	한글	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

1.2.8.1 – Korean Matrix with locked 한글CODE key

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0								
Col. 1								
Col. 2	ㅠ	ㅁ						
Col. 3	ㅓ	ㅑ	ㅗ	ㅎ	ㄹ	ㄷ ㄸ	ㅇ	ㅊ
Col. 4	ㄱ ㄲ	ㅂ ㅃ	ㅔ ㅉ	ㅐ ㅒ	ㅜ	ㅡ	ㅣ	ㅏ
Col. 5	ㅋ	ㅛ	ㅌ	ㅈ	ㅍ	ㅕ	ㅅ ㅆ	ㄴ

1.2.9 – Arabic Matrix

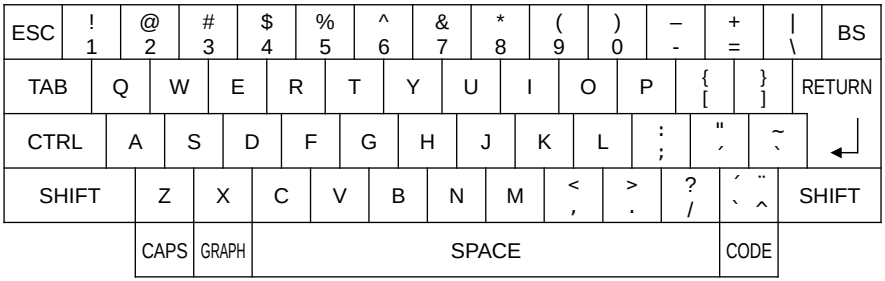
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	7 &	6 ^	5 %	4 $	3 #	2 @	1 !	0)
Col. 1	; :	] }	[{	\ |	= +	- _	9 (	8 *
Col. 2	b B	a A		/ ?	. >	, <	' ~	` "
Col. 3	j J	i I	h H	g G	f F	e E	d D	c C
Col. 4	r R	q Q	p P	o O	n N	m M	l L	k K
Col. 5	z Z	y Y	x X	w W	v V	u U	t T	s S
Col. 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
Col. 7	RET	SLCT	BS	STOP	TAB	ESC	F5	F4
Col. 8	→	↓	↑	←	DEL	INS	HOME	SPACE
Col. 9	Num4	Num3	Num2	Num1	Num0	Num/	Num+	Num*
Col. 10	Num.	Num,	Num-	Num9	Num8	Num7	Num6	Num5

1.2.9.1 – Arabic matrix with Arabic mode activated

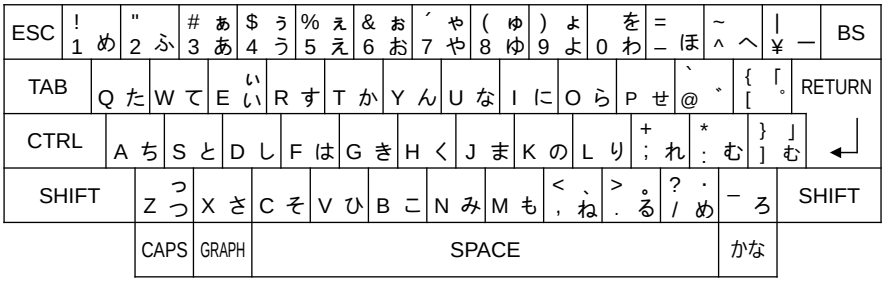
	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
Col. 0	٧	٦	٥	٤	٣	٢	١	٠
Col. 1	ك		ج				٩	٨
Col. 2	لأ لا	آ ش		؟		،	؛	ّ
Col. 3	ت	ه ُ	أ ا	لإ ل	ب	ث ٌ	ـَ ي	ذ د
Col. 4	ق ً	ض ْ	ح [	خ َ	لآ ة	ؤ و	م	ن
Col. 5	ظ ط	غ ٌ	ء ى	ص ِ	ز ر	ع ِ	ف َ	إ ش

1.3 – KEYBOARD LAYOUTS

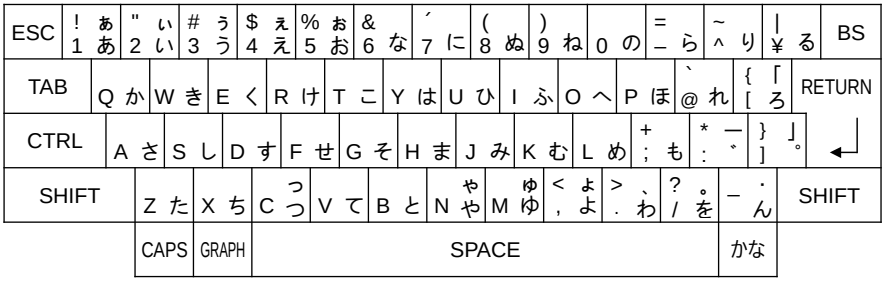
1.3.1 – Internacional Layout



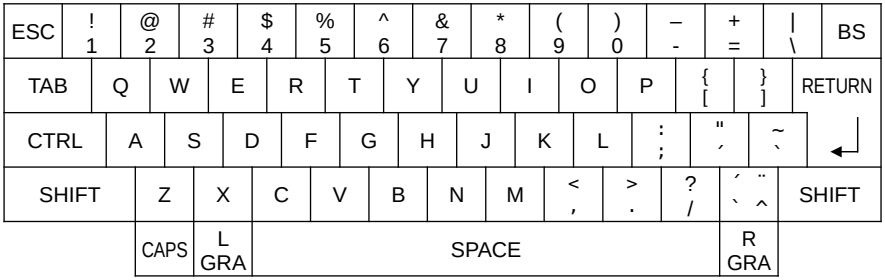
1.3.2 – Japanese Layout (JIS)



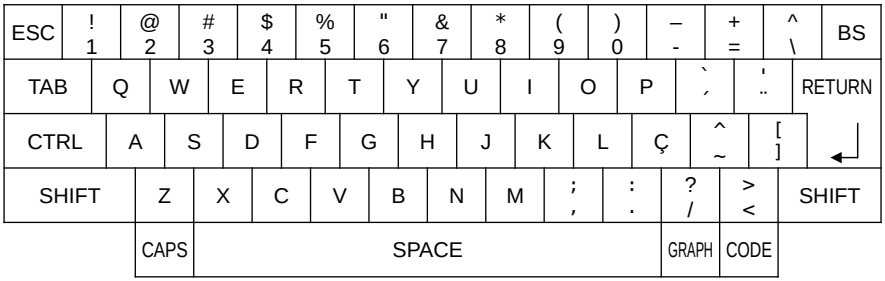
1.3.3 – Japanese Layout (ANSI)



1.3.4 – Brazilian Layout 1.0 (Expert 1.0)

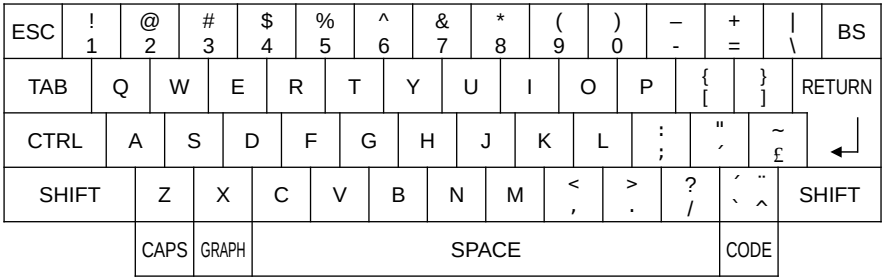


1.3.5 – Brazilian Layout 1.1 (Hotbit / Expert 1.1)

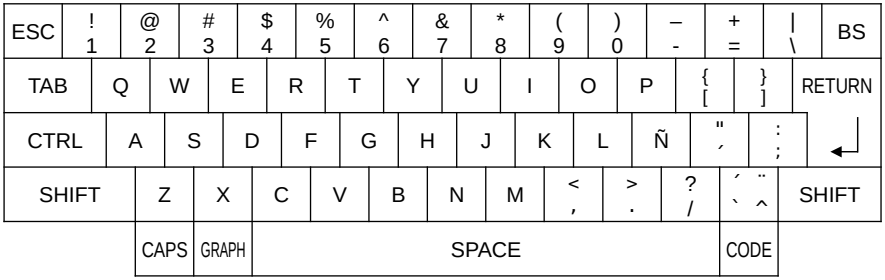


Note: For Expert, the GRAPH and CODE keys are renamed to L GRA and R GRA, in the same position as version 1.0.

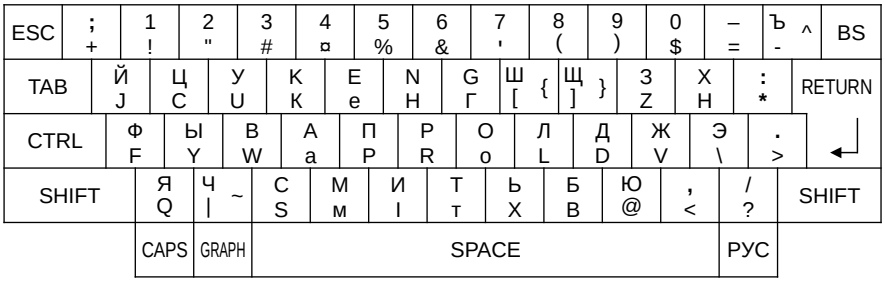
1.3.6 – United Kingdom Layout



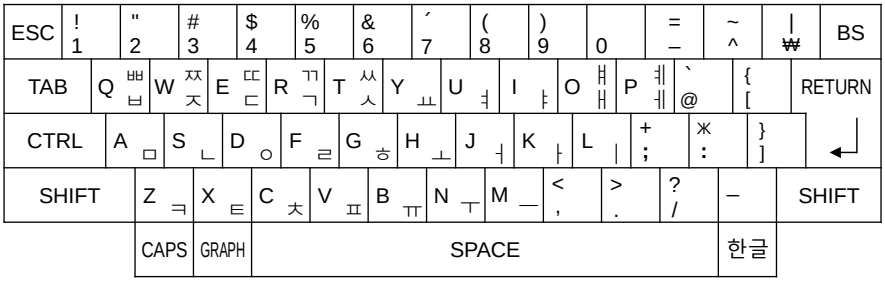
1.3.7 – Argentine / Spanish Layout



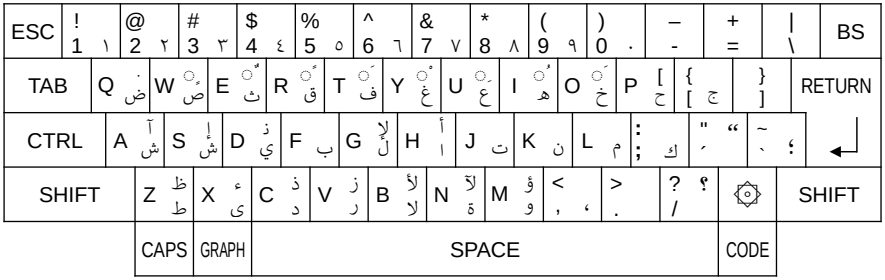
1.3.8 – Russian Layout (Cyrillic)



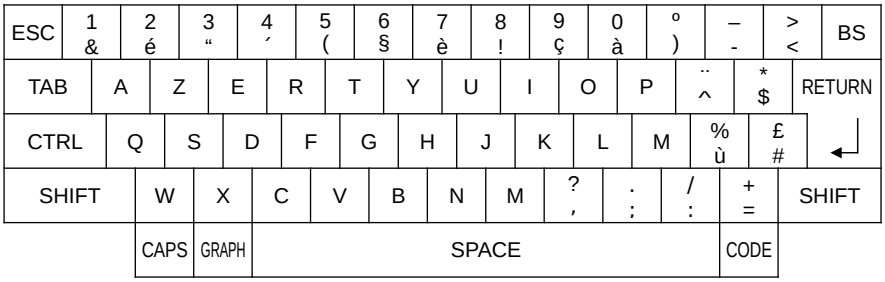
1.3.9 – Korean Layout (CPC-400)



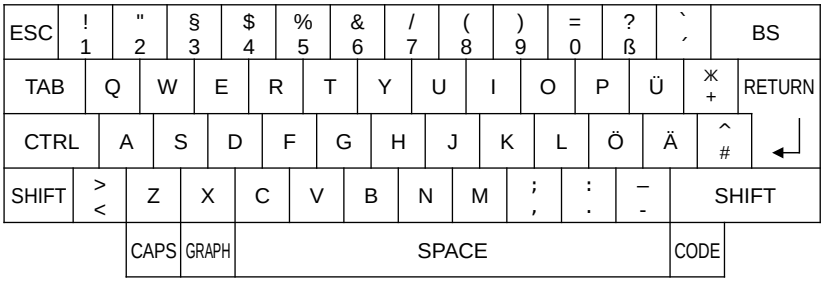
1.3.10 – Arabic Layout (AX-170)



1.3.11 – French Layout (ML-F80)



1.3.12 – German Layout (HB-F700D)



1.4 – CONTROL CODES

Shortcut	DEC	HEX	Function
Ctrl+A	001	01H	Determines graphic character.
Ctrl+B	002	02H	Deflects cursor to start of the previous word.
Ctrl+C	003	03H	Closes the entry condition.
Ctrl+D	004	04H	
Ctrl+E	005	05H	Cancel character from cursor to end of line.
Ctrl+F	006	06H	Deflect cursor to start the next word.
Ctrl+G	007	07H	Generates a beep.
Ctrl+H	008	08H	Deletes the letter before the cursor (BS).
Ctrl+I	009	09H	Move cursor to the next TAB position (TAB).
Ctrl+J	010	0AH	Line change (Linefeed).
Ctrl+K	011	0BH	Returns cursor to position 1.1 (HOME).
Ctrl+L	012	0CH	Clears the screen and puts the cursor in 1.1 pos.
Ctrl+M	013	0DH	Carriage Return (RETURN).
Ctrl+N	014	0EH	Moves the cursor to the end of the line.
Ctrl+O	015	0FH	
Ctrl+P	016	10H	
Ctrl+Q	017	11H	
Ctrl+R	018	12H	Turn on/off insertion mode (INS).
Ctrl+S	019	13H	
Ctrl+T	020	14H	
Ctrl+U	021	15H	Delete the entire line on which the cursor is.
Ctrl+V	022	16H	
Ctrl+W	023	17H	
Ctrl+X	024	18H	(SELECT).
Ctrl+Y	025	19H	
Ctrl+Z	026	1AH	(EOF) – End of File.
Ctrl+[	027	1BH	(ESC) – Escape.
Ctrl+\	028	1CH	Moves the cursor to the right.
Ctrl+]	029	1DH	Moves the cursor to the left.
Ctrl+^	030	1EH	Move the cursor up.
Ctrl+_	031	1FH	Moves the cursor down.
Delete	127	7FH	Deletes the character under the cursor (DEL).

2 – I/O PORTS MAP

00H~01H	Music Module MIDI (output) port (do not use at the same time with Sony Sensor Kid Cartridge).
00H~01H	Sony Sensor Kid Cartridge (do not use with Music Module).
02H~03H	FAC MIDI Interface (mirrored at 00H~07H).
04H~05H	Music Module MIDI (input).
00H~07H	MD Telcom modem.
08H~09H	No known use.
0AH	DAC of the Music Module.
08H~0EH	No known use.
0FH	MegaRAM Zemina.
10H~11H	PSG emulation for MegaflashROM in FPGA.
12H~13H	No known use.
14H~17H	YM2608 OPNA Cartridge.
18H~19H	Philips NMS 1170/20 barcode reader.
1AH~1FH	No known use.
20H~28H	Philips Modem NMS1251 (config. 30H~38H via jumper). Miniware M4000 modem (config. 30H~38H via jumper).
21H~27H	Sunrise MP3 player.
27H~2FH	Philips NMS serial interface 1210/1211/1212. (configurable in 37H~3FH via jumper).
28H~29H	DenYoNet ethernet interface.
2AH~2BH	PlaySoniq cartridge (setting registers).
30H~38H	Philips Modem NMS1251 (config. 20H~28H via jumper). Miniware M4000 modem (config. 20H~28H via jumper). Green-Mak SCSI interface. Philips NMS 0210 CD-ROM interface.
37H~3FH	Philips NMS serial interface 1210/1211/1212 (configurable in 27H~2FH via jumper).
3CH	Musical Memory Mapper control register.
3FH	Register of the SN76489 of the Musical Memory Mapper.
40H~4FH	Access to switchable I/O ports.
40H	(R/W) Device ID.
41H~4FH	(R/W) access to the device.
48H~49H	Franky Cartridge (SN76489 and VDP).
50H~5DH	No known use.
5EH~5FH	GR8NET interface (Ethernet).

60H~6FH VDP V9990:
- 60H (R/W) Access to VRAM.
- 61H (R/W) Access to the color palette.
- 62H (R/W) Access to hardware commands.
- 63H (R/W) Access to registers.
- 64H (W) Selection of registers.
- 65H (R) Status port.
- 66H (W) Interruption flag.
- 67H (W) System control.
- 68H (W) Address of Kanji-ROM (low) – 1.
- 69H (R/W) Kanji-ROM address (high) and data – 1.
- 6AH (W) Address of the Kanji-ROM (low) – 2.
- 6BH (R/W) Kanji-ROM (high) address and data – 2.
- 6CH~6FH Not used.

70H~73H Saurus MIDI Cartridge.
74H~76H No known use.
77H Super Game 90.
78H~7BH No known use.
7CH~7DH MSX-MUSIC (YM2413):
- 7CH (W) Selects registers.
- 7DH (W) Data port.

7EH~7FH Moonsound Cartridge (OPL4) – PCM synthesis:
7EH PCM registers (wave).
7FH PCM data (wave).
80H~87H Standard RS232C serial interface:
- 80H (R/W) USART 8251 – Data logger.
- 81H (R/W) USART 8251 – Status and command log.
- 82H (R/W) USART 8251 – Status / communication.
- 83H (R/W) Interrupt mask.
- 84H (R/W) 8253 – Counter 1.
- 85H (R/W) 8253 – Counter 2.
- 86H (R/W) 8253 – Counter 3.
- 87H (W) Meter control.

88H~8BH Access to external V9938.
8CH~8DH MSX Modem.
8EH~8FH Megaram:
- 8EH Page selection.
- 8FH Megaram-Disk.

90H~91H Printer:
90H (R) Status.
91H (W) Data.
92H~93H No known use.
94H Direction to printer port (non-standard).
95H~97H No known use.
98H~9BH VDP TMS9918 / V9938 / V9958:
98H (R/W) Read /Write data in VRAM.
99H (R/W) Read status register;
Write to the control register.
9AH (W) Writes to the palette registers.
9BH (W) Write in indirectly specified register.
9CH~9FH No known use.
A0H~A2H PSG AY-3-8910:
A0H (W) Address port.
A1H (W) Data writing port.
A2H (R) Data readout port.
A3H No known use.
A4H~A5H PCM (Turbo R):
A4H (R/W) Data port.
A5H (R/W) Control port.
A6H No known use.
A7H Controls panel lights on the MSX turbo R:
bit 1 = LED Pause.
bit 7 = turbo LED.
A8H~ABH PPI 8255:
A8H (R/W) PPI port A (slot selection).
A9H (R/W) PPI port B (keyboard reading).
AAH (R/W) PPI port C (keyboard line / click keys).
ABH (W) PPI control port.
ACH~AFH MSX-Engine (1chipMSX control).
B0H~B3H Memory expansion (SONY 8255 specification):
B0H Address lines A0~A7.
B1H Address lines A8~A10, A13~A15, control, R/W.
B2H Address lines A11~A12 and data D0~D7.
B4H~B5H Clock IC (RP-5C01):
B4H Address of the registers.
B5H Read/write data.

B6H~B7H Card reader?
B8H~BBH Lightpen control (SANYO specification).
BCH~BFH VHD Control (JVC 8255 specification).
C0H~C1H MSX-Audio Y8950:
C0H (R/W) Selects regs and reads reg. status.
C1H (R/W) Write or read reg. specified.
C0H~C3H Alternative ports for Moonsound / OPL4.
C4H~C7H Moonsound Cartridge (OPL4) – FM synthesis:
C4H FM register array 0 (bank 1) and reg. status.
C5H FM (data).
C6H FM register array 1 (bank 2).
C7H Mirror of (access via C5H is preferred).
C8H~CCH Asynchronous serial interface.
CDH~CFH No known use.
D0H~D7H Reserved for disk interface.
D8H~D9H Kanji-ROM Jis 1:
D8H (W) Address lines A0~A5.
D9H (R/W) Address lines A6~A11 and data D0~D7.
DAH~DBH Kanji-ROM Jis 2:
DAH (W) Address lines A0~A5.
DBH (R/W) Address lines A6~A11 and data D0~D7.
DCH~DDH Playsoniq Cartridge (Sega Gamepad support).
DEH~DFH No known use.
E0H~E2H MSX-MIDI external:
E0H Data transmission / reception.
E1H Control port.
E2H Selection port.
E3H No known use.
E4H~E7H Access to the S1990 (MSX turbo R):
E4H Registers adresses.
E5H Data.
E6H 16-bit counter (LSB) and counter reset.
E7H 16-bit counter (MSB).
E8H~EFH MSX-MIDI:
E8H Data transmission / reception.
E9H Control port.
EAH Latch of signals (written only).
EBH Mirror from EAH.

ECH Counter 0.
EDH Counter 1.
EEH Counter 2.
EFH Control of counters (write only).

F0H~F2H No known use.

F3H Current screen mode (MSX2+ only):
bit 0 = M3 bit 4 = M1
bit 1 = M4 bit 5 = TP
bit 2 = M5 bit 6 = YUV
bit 3 = M2 bit 7 = YAE

F4H RESET status for MSX2+ and MSX turbo R:
bit 5 – Flag to indicate that the system is already initialized.
bit 7 – 0 = hard reset; 1 = Soft reset.
Note: in some MSX2+, the read data must be inverted to obtain the correct value.

F5H System control (setting the bit to 1 enables):
b0 – Kanji-ROM b4 – MSX-Interface
b1 – Reserved Kanji b5 – Serial RS232C
b2 – MSX-Audio b6 – Lightpen
b3 – Superimpose b7 – Clock IC

F6H Color I/O bus (Color Bus).

F7H AV Control (setting the bit to 1 enables):
b0 – Simultaneous right and left audio.
b1 – Audio L (left) only.
b2 – Select video input (RGB21).
b3 – Flag indicating whether there is a video input or not.
b4 – AV Control (RGB21).
b5 – Ym control (RGB21).
b6 – Inverse of bit 4 of reg. # 9 of the VDP.
b7 – Inverse of bit 5 of the VDP reg. # 9.

F8H Access to PAL A/V control register.

F8H~FBH Access to the 8-bit MSB of the 16-bit register of the Playsoniq cartridge (they are the same ports used in some MSX that are disabled by default).

FCH~FFH Memory Mapper:
FCH (R/W) Physical page 0 (0000H~3FFFH).
FDH (R/W) Physical page 1 (4000H~7FFFH).
FEH (R/W) Physical page 2 (8000H~BFFFH).
FFH (R/W) Physical page 3 (C000H~FFFFH).

3 – MSX-BASIC

3.1 – FORMAT

INSTRUCTION NAME (instruction type, BASIC version)
  Format: Valid formats for the instruction.
  Function: Form of operation of the instruction.

There are five types of instructions, namely: declarations, commands, functions, system variables and logical operators.

The BASIC version indicates the version for which the instruction is implemented. Values separated by "–" indicate that there are differences in syntax or behavior for different versions.

1~4	MSX-BASIC version
M	MSX-MUSIC BASIC
K	Kanji-ROM required
D	Disk-BASIC 1.0
D2	Disk-BASIC 2.0

3.1.1 – Instructions Abbreviations

REM	‘
PRINT	?
CALL	_

3.1.2 – Logical Operation Codes

PSET	TPSET	Uses the specified color (default)
PRESET	TPRESET	Makes "NOT (color specified)"
XOR	TXOR	Makes "(target color) XOR (specified color)"
OR	TOR	Makes "(target color) OR (specified color)"
AND	TAND	Makes "(target color) AND (specified color)"

Note: when the operation is preceded by "T", no operation will be performed when the color is transparent.

3.1.3 – Code notations

&B Precedes a constant in binary form
&O Precedes a constant in octal form
&H Precedes a constant in hexadecimal form
% Marks variable as integer
! Marks variable as simple precision
Marks variable as double precision
$ Marks variable as alphanumeric
– Mathematical operator for subtraction
+ Mathematical operator for addition
/ Mathematical operator for division
* Mathematical operator for multiplication
^ Mathematical operator for potentiation
= Denotes equality and assigns values
<> Denotes difference

3.1.4 – Format Notations

<exprA> variable, constant, or string or numeric expression.
<exprN> variable, constant or numeric expression.
<expr$> variable, constant, or string expression.
<n> is a defined number. When in parentheses it can be an expression or numeric variable.
[ ] delimits optional parameter.
| it means that only one of the items can be used.
{ } delimits option.
X any variable.
X% any integer variable.
X! any single precision variable.
X# any double precision variable.
X$ any alphanumeric variable.

Characters in parentheses after multiple formats for an instruction indicate the version of BASIC in which that instruction format is available.

3.2 – INSTRUCTIONS DESCRIPTION

ABS (function, 1)
Format: X = ABS (<exprN>)
Function: Returns in X the absolute value (module) of <exprN>.

AND (logical operator, 1)
Format: <exprA1> AND <exprA2>
Function: Performs logical AND operation between <exprA1> and <exprA2>.

```
0 and 0 → 0    1 and 0 → 0
0 and 1 → 0    1 and 1 → 1
```

ASC (function, 1)
Format: X = ASC (<expr$>)
Function: Returns the ASCII code of the first character of expr$ in X.

ATN (function, 1)
Format: X = ATN (<exprN>)
Function: Returns in X the arcotangent of exprN (exprN must be expressed in radians).

AUTO (command, 1)
Format: AUTO [numlline, [increment]]
Function: Automatically generates line numbers, starting with [numline] and incremented with the value of [increment].

BASE (system variable, 1-2-3)
Format: X = BASE (<n>)
BASE (<n>) = <exprN>
Function: Returns in X or sets the starting addresses of the tables in VRAM for each screen mode. <n> is an integer that follows the following table:

	SCREEN MODES												
	0	1	2	3	4	5	6	7	8	10	11	12	Table of ...
BASE VALUE	0	5	10	15	20	25	30	35	40	50	55	60	pattern names
	-	6	11	16	21	26	31	36	41	51	56	61	colors
	2	7	12	17	22	27	32	37	42	52	57	62	pattern generator
	-	8	13	18	23	28	33	38	43	53	58	63	sprites attributes
	-	9	14	19	24	29	34	39	44	54	59	64	sprites generator

BEEP (declaration, 1)
Format: BEEP
Function: Generates a beep.

BIN$ (function, 1)
Format: X$ = BIN$ (<exprN>)
Function: Converts the value of <exprN> to a string of binary codes and returns the value obtained in X$.

BLOAD (command, 1-D)
Format: BLOAD "<filename>" [,R [,<offset>]]
BLOAD "<filename>" [{,R | ,S}] [,<offset>]] (D)
Function: Load a binary block into RAM or, if specified [,S], into VRAM (Disk Basic only). If specified [,R], executes a program in machine code.

BSAVE (command, 1-D)
Format: BSAVE "<filename>",<endini>, <endfim> [,<endexec>]
BSAVE "<filename>",<endini>, <endfim> [,<endexec> [, S]]
Function: Saves a binary block to disk or tape. [,S] saves a VRAM block (option available only under Disk Basic).

CALL (declaration, 1-2-3-4-D-M-Nextor)
Format: CALL <extended command> [(<argument> [, argument>...])]
Function: Executes extended commands through ROM cartridges or routines loaded in RAM. See the section "DESCRIPTION OF EXTENDED COMMANDS".

CDBL (function, 1)
Format: X# = CDBL (<exprN>)
Function: Converts the value of <exprN> to a double precision value and returns the value obtained in the variable X#.

CHR$ (function, 1)
Format: X$ = CHR$ (<exprN>)
Function: Returns in X$ the character whose ASCII code is expressed in <exprN>.

CINT (function, 1)
Format: X% = CINT (<exprN>)
Function: Converts the value of <exprN> to an integer number and loads it in the variable X%.

CIRCLE (statement, 1-2)
Format: CIRCLE {(X, Y) | STEP (X, Y)},<radio> [,<color>
[,<start angle> [,<end angle> [,<aspect ratio]]]]
Function: Draws a circle with a central point at (X, Y). If STEP is specified, the coordinates will be calculated from the current one. <start angle> and <end angle> must be specified in radians. <proportion> is the relation for ellipse; <1> being perfect circumference.

CLEAR (statement, 1)
Format: CLEAR [<string area size> [, upper memory limit>]]
Function: Initializes the BASIC variables and sets the size of the string area and the upper limit of memory used by BASIC.

CLOAD (command, 1)
Format: CLOAD [<filename>]
Function: Loads a BASIC program from the tape.

CLOAD? (command, 1)
Format: CLOAD? [<filename>]
Function: Compares a BASIC program on the cassette with the one that is in the memory.

CLOSE (command, 1-D)
Format: CLOSE [[#]<file number> [, [#]<file number> ...]]
Function: Closes the specified files. If no file is specified, close all opened files.

CLS (declaration, 1)
Format: CLS
Function: Clears the screen.

CMD (command, 1)
Format: No format defined.
Function: Reserved for implementing new commands.

COLOR (statement, 1-2)
Format: COLOR [<front color> [,<background color>
[,<border color>]]] (1-2)
Function: Specifies the colors of the screen.

COLOR = (declaration, 2)
Format: COLOR = (<palette number>, <red level>, <green level>,
<blue level>)
Function: Specifies the colors of the palette. The level can vary from 0 to 7 for each color.

COLOR = NEW (declaration, 2)
Format: COLOR [= NEW]
Function: Initializes the color palette.

COLOR = RESTORE (declaration, 2)
Format: COLOR = RESTORE
Function: Copies the contents of the color palette stored in VRAM to the VDP palette registers.

COLOR SPRITE (statement, 1-2)
Format: COLOR SPRITE (<sprite plane number>) = <color>
Function: Specifies the color of the sprites.

COLOR SPRITE$ (declaration, 2)
Format: COLOR SPRITE$ (<sprite plan number>) = <expr$>
Function: Specifies the color of each line of the sprites.
<expr$> = CHR$ (1st line color) + CHR$ (2nd line color) ...

CONT (command, 1)
Format: CONT
Function: Continues the execution of a program that was interrupted.

COPY (declaration, 1-2-D)
Format: COPY <filename1> [TO <filename2>] (1-D)
Function: Copy the contents of <filename1> to <filename2>.
Format: COPY (X1, X2) – (Y1, Y2) [,<source page>] TO (X3, Y3) [,
<target page> [,<logical operation>]] (2)
Function: Copies a rectangular area of the screen to another.

Format: COPY (X1, X2) – (Y1, Y2) [,<source page>] TO
<matrix variable | <filename>} (2-D)
Function: Copy the contents of a rectangular area of the screen to a matrix variable or to a file on disk.
Format: COPY {<matrix variable> | <filename>} [,<direction>] TO
(X3, Y3) [,<dest page> [,<logical operation>]] (2-D)
Function: Copies the contents of a matrix variable or a disk file to a rectangular area on the screen.
Format: COPY <filename> TO <matrix variable> (2-D)
Function: Copies the contents of a file to an array variable.
Format: COPY <matrix variable> TO <filename> (2-D)
Function: Copies the contents of an array variable to a file.

COPY SCREEN (declaration, 2, optional)
Format: COPY SCREEN [<mode>]
Function: Writes the Color Bus data on VRAM.
<mode> can be:
0 – Scans the current video page.
1 – Scans two pages, the first on the page before the active one and the second on the active page (interlaced mode).
Note: requires a digitizer or superimposer.

COS (function, 1)
Format: X = COS (<exprN>)
Function: Returns in X the cosine value of <exprN> (exprN must be expressed in radians).

CSAVE (command, 1)
Format: CSAVE <filename> [,<baud rate>]
Function: Saves a BASIC program to the cassette.

CSNG (function, 1)
Format: X! = CSNG (<exprN>)
Function: Converts the value of <exprN> to a simple precision value and store it in X!.

CSRLIN (system variable, 1)
Format: X = CSRLIN
Function: Contains the vertical position of the cursor.

CVD (function, D)
Format: X# = CVD (<8-byte string>)
Function: Converts the string to a double precision value and store it in X#.

CVI (function, D)
Format: X% = CVI (<2-byte string>)
Function: Convert the string to an integer value and store it in X%.

CVS (function, D)
Format: X! = CVS (<4-byte string>)
Function: Converts the string to a simple precision value and store it in X!.

DATA (declaration, 1)
Format: DATA <constant> [,<constant> ...]
Function: Stores a list of data for the READ command.

DEF FN (statement, 1)
Format: DEF FN<name> [(<argument> [,<argument> ...])] = <user function defining expression>
Function: Defines a user function.

DEFDBL (statement, 1)
Format: DEFDBL <character range> [,<character range> ...]
Function: Declares the specified variables as double precision.

DEFINT (statement, 1)
Format: DEFINT <character range> [,<character range> ...]
Function: Declares the specified variables as integers.

DEFSNG (statement, 1)
Format: DEFSNG <character range> [,<character range> ...]
Function: Declares the specified variables as simple precision.

DEFSTR (statement, 1)
Format: DEFSTR <character range> [,<character range> ...]
Function: Declares the specified variables as strings.

DEFUSR (statement, 1)
Format: DEFUSR [<number>] = <address>
Function: Defines an initial address for executing an assembly program to be called by the USR function. <number> can vary from 0 to 9.

DELETE (command, 1)
Format: DELETE {<start line> – <end line> | <line> | – <end line>}
Function: Deletes the specified lines from the BASIC text.

DIM (declaration, 1)
Format: DIM <variable> (<max index> [,<max index> ...])
Function: Defines an variable array and allocates space in memory.

DRAW (macro declaration, 1)
Format: DRAW <expr$>
Function: Draw a line according to <expr$>. The valid commands for <expr$> are as follows:

Un	to up	Dn	to down
Ln	to left	Rn	to right
En	up and right	Fn	down and right
Gn	low and left	Hn	up and left
B	move no drawing	N	back to origin
Mx,y	goes to X, Y	An	rotates n*90 degrees
Sn	scale n / 4	Cn	color n

Xseries runs macro in series.
Ex. `A$ = "C15U10" → DRAW "XA$"`
= <variable> – Place a parameter as an integer after the command. Ex. `A$="C15U10", S=50, → DRAW "XA$;R=S;D=S"`

DSKF (function, D)
Format: X = DSKF (<drive number>)
Function: Returns the free space on the specified drive in clusters. If Nextor is installed, it will return the space in Kbytes.

EOF (function, 1-D)
Format: X = EOF (<file number>)
Function: Returns –1 if the end of the file is detected.

ERASE (declaration, 1)
Format: ERASE <matrix variable> [,<matrix variable>...]
Function: Delete the specified matrix variables.

EQV (logical operator, 1)
Format: <exprA1> EQV <exprA2>
Function: Performs EQV logical operation between <exprA1> and <exprA2>. The result will be 1 if the two bits are equal and zero if they are different.

```
0 eqv 0 → 1    1 eqv 0 → 0
0 eqv 1 → 0    1 eqv 1 → 1
```

ERL (system variable, 1)
Format: X = ERL
Function: Contains the line number where the last error occurred.

ERR (system variable, 1)
Format: X = ERR
Function: Contains the error code of the last error occurred.

ERROR (statement, 1)
Format: ERROR <error code>
Function: Puts the program in error condition.

EXP (function, 1)
Format: X = EXP (<exprN>)
Function: Returns in X the value of the natural potentiation of <exprN>.

FIELD (statement, D)
Format: FIELD [#]<file number>, <field size> AS <string variabe> [,<field size> AS <string variable> ...]
Function: Assigns <string variable> for random disk access.

FILES (command, D)
Format: FILES [<filename>]
Function: Displays the directory of the disc according to <filename>. If <filename> is omitted, it displays the names of all files on the disk.

FIX (function, 1)
Format: X = FIX (<exprN>)
Function: Returns in X the entire part of <exprN>, without rounding.

FOR (statement, 1)
Format: FOR <variable name> = <initial value> TO <final value> [STEP <increment>]
Function: Repeat the execution of the section between FOR and NEXT.

FRE (function, 1)
Format: FRE (0 | "")
Function: Returns the size of the remaining memory for the BASIC text (0) or for the string variables ("").

GET (declaration, D)
Format: GET [#]<file number> [,<record number>]
Function: Reads a record from a random access file.

GET DATE (declaration, 2)
Format: GET DATE <string variable> [,A]
Function: Returns a string with the current date in the <string variable>. If ",A" is specified, returns the alarm date.

GET TIME (statement, 2)
Format: GET TIME <string variable> [,A]
Function: Returns a string with the current time in the <string variable>. If ",A" is specified, returns the alarm time.

GOSUB (statement, 1)
Format: GOSUB <line number>
Function: Calls a subroutine that starts at line <line no.>.

GOTO (statement, 1)
Format: GOTO <line number>
Function: Jump to line <line number>.

HEX$ (function, 1)
Format: X$ = HEX$ (<exprN>)
Function: Converts the value of <exprN> to a hexadecimal string and returns it in X$.

IF (statement, 1)
Format: IF <condition> THEN {<command> | <line number>}
[ELSE {<command> | <line number>}]
IF <condition> GOTO <line number> [ELSE <line number>]
Function: Executes commands according to <condition>.

IMP (logical operator, 1)
Format: <exprA1> IMP <exprA2>
Function: Performs IMP logical operation between <exprA1> and <exprA2>. The result will be 0 when the first bit is true and implies that the second is false. Otherwise, it will be 1.

```
0 imp 0 = 1          1 imp 0 = 0
0 imp 1 = 1          1 imp 1 = 1
```

INKEY$ (function, 1)
Format: X$ = INKEY$
Function: Returns the character in X$ whose key is being pressed; otherwise, returns a null string.

INP (function, 1)
Format: X = INP (<port number>)
Function: Reads an I/O port from the Z80 and returns its value in X.

INPUT (statement, 1)
Format: INPUT ["<prompt>";]<variable name> [,<variable name> ...]
Function: Reads a data entry using the keyboard and stores the obtained value(s) in the respective variable(s).

INPUT# (declaration, 1)
Format: INPUT# <file number>, <variable name> [,<variable name> ...]
Function: Read data from the specified file and store the obtained value(s) in the respective variable(s).

INPUT$ (function, 1)
Format: X$ = INPUT$ (<number of characters> [, [#]<file number>])
Function: Reads the specified number of characters from the keyboard or a file and stores the obtained value in X$.

INSTR (function, 1)
Format: X = INSTR ([<exprN>,]<expr$1>, <expr$2>)
Function: Searches for the occurrence of <expr$2> in <expr$1> from the position <exprN> and returns the value obtained in X. If <expr$1> is not found, returns 0.

INT (function, 1)
Format: X = INT (<exprN>)
Function: Returns in X the entire part of <exprN>, rounding off.

INTERVAL (statement, 1)
Format: INTERVAL {ON | OFF | STOP}
Function: Activate, deactivate or suspend interruption for a period of time.

IPL (command, 1)
Format: No defined format.
Function: Reserved for implementing new commands.

KEY (command / declaration, 1)
Format: KEY <key number>, <expr$>
Function: Assign the content of the specified function key.
Format: KEY (<key number>) {ON | OFF | STOP}
Function: Enables, disables or suspends function key interruption.
Format: KEY {ON | OFF}
Function: Turns on or off the display of the content of the function keys on the bottom screen line.

KEY LIST (command, 1)
Format: KEY LIST
Function: Lists the content of the function keys.

KILL (command, D)
Format: KILL <expr$>
Function: Delete files on the disk. <expr$> must contain a valid filename.

LEFT$ (function, 1)
Format: X$ = LEFT$ (<expr$>, <exprN>)
Function: Returns the <exprN> left <expr$> characters in X$.

LEN (function, 1)
Format: X = LEN (<expr$>)
Function: Returns in X the number of characters in <expr$>.

LET (declaration, 1)
Format: [LET]<variable name> = <exprA>
Function: Stores the value of <exprA> in the variable.

LFILES (command, 1)
Format: LFILES [<filename>]
Function: Lists the filenames of the disk in the printer according to <filename>. If <filename> is omitted, it lists the names of all files on the disk.

LINE (statement, 1-2)
Format: LINE [{(X1, Y1) | STEP (X1, Y1)}] – {(X2, Y2) | STEP (X2, Y2)} [,<color> [, {B | BF} [,<logical operation>]]]
Function: Draws a line, an empty rectangle (,B) or a painted rectangle (,BF). The STEP subcommand, when specified, defines the offset.

LINE INPUT (statement, 1)
Format: LINE INPUT ["<prompt>";]<string variable>
Function: Reads a sequence of characters from the keyboard and stores the value read in the <string variable>.

LINE INPUT# (declaration, 1-D)
Format: LINE INPUT#<file number>, <string variable>
Function: Reads a sequence of characters from a file and stores the value read in the <string variable>.

LIST (command, 1)
Format: LIST [[<start line>] – [<end line>]]
Function: List on the screen the BASIC program that is in memory.

LLIST (command, 1)
Format: LLIST [[<start line>] – [<end line>]]
Function: Lists the BASIC program in the printer.

LOAD (command, 1-D)
Format: LOAD "<filename>" [,R]
Function: Load a program into memory. The [,R] parameter makes the program to be executed after loading.

LOC (function, D)
Format: X = LOC (<file number>)
Function: Returns in X the number of the last accessed record of the file.

LOCATE (declaration, 1-2)
Format: LOCATE [<X coord.> [,<Y coord.> [,<cursor type>]]]
Function: Position the cursor on the text screens. If <cursor type> is 0 (default value) the cursor will not be displayed when the computer is busy; if any other value, the cursor will always be displayed.

LOF (function, D)
Format: X = LOF (<file number>)
Function: Returns the specified file size in X.

LOG (function, 1)
Format: X = LOG (<exprN>)
Function: Returns in X the natural logarithm of <exprN>.

LPOS (system variable, 1)
Format: X = LPOS
Function: Stores the horizontal location of the printer head.

LPRINT (declaration, 1)
Format: LPRINT [<exprA> [{; | ,} <exprA> ...]]
Function: Send the characters in the the expressions <exprA> to the printer. ";" prints the next character immediately after, "," prints the character in the next tab stop.

LPRINT USING (statement, 1)
Format: LPRINT USING <"format">; <exprA> [{; | ,} <exprA>...]
LPRINT USING <"expr$ formatted">

Function: Send to the printer the characters corresponding to the expressions <exprN> or <expr$>, formatting. ";" prints the next character immediately after, "," prints the character at the next tab stop. The characters used to format the output are as follows:

➔ Numeric formatting:

#	Space for one digit
.	Includes decimal point
+	Indicates + or -; used before or after the number
-	Indicates -; used after the number
$$	Put$ to the left of the number
**	Replaces leading spaces with asterisks
**$	Places a$ to the left preceded by asterisks
^^^^	Displays the number in scientific notation

➔ Alphanumeric formatting:

\ \	Character space
!	Space for a character
&	Variable spacing
_	Next character is printed normally
other	Print character

LSET (declaration, D)

Format: LSET <string variable> = <expr$>

Function: Stores the contents of <expr$> on the left of the variable string defined by the FIELD statement.

MAXFILES (declaration, 1-D)

Format: MAXFILES = <number of files>

Function: Defines the maximum number of files that can be opened at the same time.

MERGE (command, 1-D)

Format: MERGE <filename>

Function: Merges the program in memory with a program saved in ASCII format on disk or tape.

MID$ (function / declaration, 1)

Format: X$ = MID$ (<expr$>, <exprN1> [, exprN2])

Function: Returns, in X$,<exprN2> characters from the character <exprN1> of <expr$>.

Format: MID$ (<string variable>, <exprN1> [,<exprN2>]) = <expr$>
Function: Defines <expr$> using <exprN2> characters from the <exprN1> position of the <string variable>.

MKD$ (function, D)
Format: X$ = MKD$ (<double precision value>)
Function: Convert a double precision value to an 8-byte string and store it in X$.

MKI$ (function, D)
Format: X$ = MKI$ (<integer value>)
Function: Convert an integer value to a 2-byte string and store it in X$.

MKS$ (function, D)
Format: X$ = MKS$ (<simple precision value>)
Function: Converts a simple precision value to a 4-byte string and store it in X$.

MOTOR (control, 1)
Format: MOTOR [{ON | OFF}]
Function: Turns the cassette motor on or off.

NAME (command, D)
Format: NAME <filename1> AS <filename2>
Function: Rename the file <filename1> to <filename2>.

NEW (command, 1)
Format: NEW
Function: Deletes the program from memory and clears the variables.

NEXT (declaration, 1)
Format: NEXT [<variable name> [,<variable name>...]]
Function: Indicates the end of the FOR loop.

NOT (logical operator, 1)
Format: NOT (<exprA>)
Function: Performs the negation of <exprA>.

```
not 0 → 1
not 1 → 0
```

OCT$ (function, 1)
Format: X$ = OCT$ (<exprN>)
Function: Converts the value of <exprN> to an octal string and returns it in X$.

ON ERROR GOTO (statement, 1)
Format: ON ERROR GOTO <line number>
Function: Defines the starting line of the error handling routine.

ON GOSUB (statement, 1)
Format: ON <exprN> GOSUB <line number> [,<line number> ...]
Function: Executes the subroutine that starts in the <line number> according to <exprN>.

ON GOTO (statement, 1)
Format: ON <exprN> GOTO <line number> [,<line number> ...]
Function: Jump to the line <line number> according to <exprN>.

ON INTERVAL GOSUB (statement, 1)
Format: ON INTERVAL = <time> GOSUB <line number>
Function: Defines the interval and the line number for time interruption. <time> is defined in 1/60 second units on a NTSC machines and in 1/50 seconds in PAL machines.

ON KEY GOSUB (statement, 1)
Format: ON KEY GOSUB <line number> [,<line number> ...]
Function: Defines the line numbers for interrupting function keys.

ON SPRITE GOSUB (statement, 1)
Format: ON SPRITE GOSUB <line number>
Function: Defines the line number for sprite collision interruption.

ON STOP GOSUB (statement, 1)
Format: ON STOP GOSUB <line number>
Function: Defines the line number for interruption by pressing the CTRL+STOP keys.

ON STRIG GOSUB (statement, 1)
Format: ON STRIG GOSUB <line number> [,<line number> ...]
Function: Defines the line numbers for interruption by pressing the joystick trigger buttons.

OPEN (declaration, 1-D)
Format: OPEN <filename> [FOR {INPUT | OUTPUT}]
AS#<file number> [LEN = <record size>]
Function: Open a file on tape or disk.

OR (logical operator, 1)
Format: <exprA1> OR <exprA2>
Function: Performs logical OR operation between <exprA1> and <exprA2>.

```
0 or 0 → 1      1 or 0 → 0
0 or 1 → 0      1 or 1 → 1
```

OUT (statement, 1)
Format OUT <port number>, <exprN>
Function: Writes the value of <exprN> to an I/O port of the Z80.

PAD (function, 1-2)
Format: X = PAD (<exprN>)
Function: Examines the state of the mouse, trackball, lightpen or digitizer tablet and returns the value obtained in X.
<exprN> can be:
0 – Check touch pad on port 1 (255 if connected)
1 – Returns the X coordinate (horizontal).
2 – Returns the Y (vertical) coordinate.
3 – Returns the key state (255 if pressed).
4 – Check touch pad on port 2 (255 if connected).
5 – Returns the X (horizontal) coordinate.
6 – Returns the Y (vertical) coordinate.
7 – Returns the key state (255 if pressed).
8 – Check lightpen (255 if connected or touching the screen).
9 – Returns the X (horizontal) coordinate.
10 – Returns the Y (vertical) coordinate.
11 – Returns the key state (255 if pressed).
12 – Check mouse on port 1 (255 if connected).
13 – Returns X coordinate offset (horizontal).
14 – Returns Y coordinate offset (vertical).
15 – Always 0.
16 – Check mouse on port 2 (255 if connected).
17 – Returns X coordinate offset (horizontal).

18 – Returns Y coordinate offset (vertical).
19 – Always 0.
20 – Checks 2nd lightpen (255 if connected or touching the screen). * obs
21 – Returns the X (horizontal) coordinate. * obs
22 – Returns the Y (vertical) coordinate. * obs
23 – Returns the key state (255 if pressed). * obs
Note: Values 20 to 23 require the use of the CALL ADJUST statement beforehand. Only available for MSX2 manufactured by Daewoo.

PAINT (statement, 1-2)
Format: PAINT {(X, Y) | STEP (X, Y)} [,<color> [,<border color>]]
Function: Fills the area enclosed by a line with the color <border color> with the color <color>.

PDL (function, 1)
Format: X = PDL (<paddle number>)
Function: Returns in X the state of the specified paddle. The paddle number can be:
1, 3, 5, 7, 9, 11 – Paddles connected to port 1.
2, 4, 6, 8, 10, 12 – Paddles connected to port 2.

PEEK (function, 1)
Format: X = PEEK (<address>)
Function: Returns in X the value of the byte contained in <address>.

PLAY (macro statement, 1)
Format: PLAY <expr$ 1> [,<expr$ 2> [, expr$ 3>]]
Function: Plays the notes specified by <expr$> on PSG. The valid commands for <expr$> are as follows:
An~Gn Plays an encrypted note with duration n (1~64, default is 4).
Rn Pause of duration n (1~64, default is 4).
or + Sustain
– Flat
. Duration increase by 50%
On Octave (default is 4)

Ln Duration of notes (1~64, default is 4)
Tn Time and quarters of note per minute (32~255)
Vn Volume (0~15, default is 8)
Nn Absolute grade (1~96)
Mn Wrap period (1~65 535, default is 255)
Sn Waveform (0~15, default is 0)
Xserie Executes macro in serie.
Ex. `A$ = "ABC#" → PLAY "XA$"`
= <variable> – Place a parameter as an integer after the command. Ex. `A$ = A$ = "ABC#", S = 10, → PLAY "XA$; R = S; V = S"`

PLAY (function, 1)
Format: X = PLAY (<n>)
Function: Returns in X the state of the voice <n> (playing = –1; not playing = 0).

PLAY# (macro declaration, M-4)
Format: PLAY# <n>, <expr$1> [,<expr$2> ..... [, expr$12>]]]]]]]]]]]]]]]
Function: Play the notes specified by <expr$> on PSG, OPLL, MSX-Audio or MSX MIDI.
The <n> value can be:
0 Play only PSG (same as PLAY)
1 Play through the MIDI interface.
2 or 3 Play through PSG and OPLL / MSX-Audio, depending on the chip activated with CALL MUSIC (OPLL) or CALL AUDIO (MSX-Audio).
⟶ The commands valid for <expr$> are the same as for the PLAY statement, plus those described below for OPLL (Note: Mn and Sn are exclusive to PSG):
Qn Sound width division (1~8, default is 8)
> Increase an octave
< Decrease one octave
= x; Sets parameters to x
& Ligature
{} n Define the notes between {} in n. (n=1~8, default is Ln)
@n Change the instrument (1~64)

→ For battery parts, the commands are as follows:
B Bass Drum
S Snare Drum
W Tom Tom
C Cymbals
H Hi hat
@Vn Detailed volume change (0~127)
@Nn Maintains the duration defined by n (1~64, default Ln)
n The "n"th note is paused (1~64)
! Accentuates the previous note
@An Sets the volume for accented voices (0~15)
→ Obs .: Tn, Vn, @Vn, Rn, X, = x; and . are identical to the other instruments.

POINT (function, 1)
Format: X = POINT (X, Y)
Function: Returns in X the color code of the point (X, Y) of the graphic screen.

POKE (statement, 1)
Format: POKE <address>, <data>
Function: Writes a byte of data to the memory <address>. <data> must be a numeric value between 0 and 255.

POS (system variable, 1)
Format: X = POS (0)
Function: Stores the horizontal position of the cursor in text mode.

PRESET (statement, 1-2)
Format: PRESET {(X, Y) | STEP (X, Y)} [,<color> [,<logical oper>]]
Function: Turns off the point specified by (X, Y) on the graphic screen. "STEP", if specified, defines the offset.

PRINT (declaration, 1)
Format: PRINT [<exprA> [{; | ,} <exprA> ...]]
Function: Displays the characters corresponding to the expressions <exprA> on the screen. ";" does not generate linefeed and "," advances to the next tab position.

PRINT# (declaration, 1-D)
Format: PRINT# <file number>, [<exprA> [{; | ,} <exprA> ...]]
Function: Writes the value of <exprA> to the specified file. ";" does not generate linefeed and "," advances to the next tab position.

PRINT USING (statement, 1)
Format: PRINT USING <"format">; <exprN> [{; | ,} <exprN> ...]
PRINT USING <"expr$ format">
Function: Displays on the screen the characters corresponding to the expressions <exprN> or <expr$>, formatting. ";" does not generate linefeed and "," advances to the next tab position. The formatting characters are described below:
→ Numeric formatting:
Space for one digit
. Includes decimal point
+ Indicates + or −; used before or after the number
− Indicates −; used after the number
$$ Put "$" to the left of the number
** Replaces leading spaces with asterisks
**$ Places a "$" to the left preceded by asterisks
^^^^ Displays the number in scientific notation
→ Alphanumeric formatting:
\ \ Character space
! Space for a character & Variable spacing
_ Next character will print normally
other Print character

PRINT# USING (declaration, 1-D)
Format: PRINT# <file number> USING <"format">; <exprA> [{; | ,} <exprA>...]
Function: Writes the value of <exprA> to the specified file, formatting. The formatting characters are the same as those for PRINT USING.

PSET (statement, 1)
Format: PSET {(X, Y) | STEP (X, Y)} [,<color> [,<logical operation>]]
Function: Draws the point specified by (X, Y) on the graphic screen. "STEP", if specified, defines the offset.

PUT (statement, D)
Format: PUT [#]<file number> [,<record number>]
Function: Writes a record to a random file.

PUT HAN (declaration, Daewoo CPC 400 / 400S)
Format: PUT HAN [(X, Y)],<Hangul code> [,<color> [,<logical operation> [,<mode>]]]
Function: Displays a Korean Hangul character on the screen.
<mode> sets the size of the Hangul character:
0 – 16x16 points
1 – 16x8 points (shows only odd lines)
2 – 16x8 points (shows only even lines)

PUT KANJI (statement, 1-2-Kanji)
Format: PUT KANJI [(X, Y)],<JIS code> [,<color> [,<logical operation> [,<mode>]]]
Function: Displays a Kanji character on the screen. <JIS code> may vary from &H2120 to &H4F53 for JIS1 and from &H5020 to &H7424 for JIS2.
<mode> defines the size of the Kanji:
0 – 16x16 points
1 – 16x8 points (shows only odd lines)
2 – 16x8 points (shows only even lines)

PUT SPRITE (statement, 1-2)
Format: PUT SPRITE <sprite plane> [, {(X, Y) | STEP (X, Y)} [,<color> [,<pattern number>]]]
Function: Displays a sprite on the screen. "STEP", if specified, sets the offset. <sprite plan> is a number from 0 to 31 and specifies the display priority. Larger numbers will be displayed over smaller numbers. <pattern number> defines the pattern to be displayed. It can vary from 0 to 255 for 8x8 sprites and from 0 to 63 for 16x16 sprites. If not specified, it will be the same as <sprite plan>.

READ (statement, 1)
Format: READ <variable name> [,<variable name> ...]
Function: Reads data from the DATA command and stores it in <variable name>'s.

REM (statement, 1)
Format: REM <comments>
Function: Put comments (remarks) in the program.

RENUM (command, 1)
Format: RENUM [<new line number> [,<old line number> [,<increment>]]]
Function: Renumber program lines.

RESTORE (declaration, 1)
Format: RESTORE [<line number>]
Function: Specifies the initial DATA line number to be read by READ.

RESUME (statement, 1)
Format: RESUME {[0] | NEXT | <line number>}
Function: Ends error handling routine.
0 – Execution returns to the same command where the error occurred.
NEXT – Execution continues on the command following the one from which the error occurred.
<line number> – Execution will continue on the specified line.

RETURN (statement, 1)
Format: RETURN [<line number>]
Function: Returns from a subroutine.

RIGHT$ (function, 1)
Format: X$ = RIGHT$ (<expr$>, <exprN>)
Function: Returns the <exprN> right <expr$> characters in X$.

RND (function, 1)
Format: X = RND [(<exprN>)]
Function: Returns in X a random number between 0 and 1. It is advisable to use "–TIME" in <exprN> to obtain better randomness.

RSET (declaration, D)
Format: RSET <string variable> = <expr$>
Function: Stores the content of <expr$> on the right of the string variable defined by the FIELD declaration.

RUN (command, 1-D)
Format: RUN [{<line number> | "filename"]
Function: Run a BASIC program in memory or load a program from disk and execute it. If <line number> is specified, execution will start on that line.

SAVE (command, 1-D)
Format: SAVE "<filename>" [,A]
Function: Saves the BASIC program to disk or tape. If ",A" is specified, save in ASCII form and not in tokenized form.

SCREEN (statement, 1-2-3)
Format: SCREEN <screen mode> [,<sprite size> [,<key click> [,<cassette rate> [,<printer type> [,<interlace>]]]]]
Function: Selects screen mode and other values.
<screen mode> – 0 to 12, depending on the MSX version. "Screen 9" only works on Korean computers or those loaded with Hangul BASIC.
<sprite size> – 0 ⟶ 8x8 sprites (default)
1 ⟶ 8x8 sprites expanded to 16x16
2 ⟶ 16x16 sprites
3 ⟶ 16x16 sprites enlarged to 32x32
<click keys> – 0 ⟶ turn off "click"
1 ⟶ turn on "click" (default)
<cassette rate> – 1 ⟶ write at 1200 baud (default)
2 ⟶ write at 2400 baud
<printer type> – 0 ⟶ MSX printer (default)
1 ⟶ non-MSX printer
<interlace> – 0 ⟶ normal (default)
1 ⟶ interlaced (Screen 0 standard)
2 ⟶ normal (alternate display)
3 ⟶ interlaced (alternate display)

SET ADJUST (statement, 2)
Format: SET ADJUST (<X coordinate>, <Y coordinate>)
Function: Changes the location of the screen. X and Y can vary from –7 to 8.

SET BEEP (declaration, 2)
Format: SET BEEP <timbre>, <volume>
Function: Selects the beep type and volume. <timbre> can vary from 1 to 4 and <volume> from 0 to 15.

SET DATE (declaration, 2)
Format: SET DATE <expr$> [,A]
Function: Changes the date of the clock. [,A] changes the alarm date. <expr$> must contain a valid date specification.

SET HAN (statement, Hangul-BASIC 2nd version)
Format: SET HAN [<size>], [<screen>], [<printer>]
Function: Defines how the Hangul characters will be displayed in Screen 0 to 8 and on the printer.
<size> – 0 ⟶ 8x8 point characters
1 ⟶ 8x16 point characters
<screen> – 0 ⟶ ungrouped characters
1 ⟶ characters grouped in blocks
<printer> – 0 ⟶ ungrouped characters
1 ⟶ characters grouped in blocks

SET PAGE (declaration, 2)
Format: SET PAGE <displayed page>, <active page>
Function: Select video pages. <displayed page> is the page that is being displayed on the screen and <active page> is the page on which the commands will be executed.

SET PASSWORD (declaration, 2)
Format: SET PASSWORD <expr$>
Function: Activates the password. <expr$> must contain a password with a maximum of 6 characters.

SET PROMPT (declaration, 2)
Format: SET PROMPT <expr$>
Function: Activates a new prompt for BASIC. <expr$> must contain the new prompt with a maximum of 6 characters.

SET SYSTEM (statement, Daewoo CPC300 / 400 / 400S)
Format: SET SYSTEM (<mode>) [CPC 300]
Function: Defines how the computer boots.
<mode> – 0 ⟶ starts the additional software in ROM
1 ⟶ starts BASIC
Format: SET SYSTEM [<dummy>], [<screen>], [<printer>] [CPC 400 / 400S]
Function: Defines the initial parameters for the Hangul system.
<dummy> – Null action for any specified value
<screen> – 0 ⟶ ungrouped characters
1 ⟶ characters grouped in blocks
<printer> – 0 ⟶ ungrouped characters
1 ⟶ characters grouped in blocks

SET SCREEN (statement, 2)
Format: SET SCREEN
Function: Writes the data defined in the SCREEN statement to the SRAM of the clock.

SET TIME (statement, 2)
Format: SET TIME <expr$> [,A]
Function: Changes the clock time. [,A] changes the alarm time. <expr$> must contain a valid time specification.

SET TITLE (declaration, 2)
Format: SET TITLE <expr$> [,<title color>]
Function: Defines the title and color of the home screen. <expr$> must contain the title with a maximum of 6 characters. <title color> can vary from 1 to 4

SET VIDEO (declaration, 2, optional)
Format: SET VIDEO [<mode> [,<Ym> [,<CB> [,<sync> [,<audio> [,<video output> [,<AV control>]]]]]]]
Function: Defines superimposition and other modes.
<mode> can range from 0 to 3:
0 – Internal synchronization (default value)
1 – Digitization (external synchronization)
2 – Superimpose (external synchronization)
3 – External video (external synchronization)

<Ym> (external luminance): 0 = normal; 1 = halftone
<CB> (color bus): 0 = Input; 1 = Output
<sync> (synchronization mode): 0 = Internal; 1 = external
<audio> – Select the audio source:
0 – Computer only
1 – Computer + external right channel
2 – Computer + external left channel
3 – Computer + external right and left channels
<video out> – Select the video out mode:
0 – RGB; 1 – Composite video
<AV control> – Selects the RGB output for audio and video.
0 – Not selected; 1 – Selected.

SGN (function, 1)
Format: X = SGN (<exprN>)
Function: Returns the result of the <exprN> sign in X.
–1 ⟶ Negative expression
0 ⟶ The results of the expression is zero
1 ⟶ Positive expression

SIN (function, 1)
Format: X = SIN (<exprN>)
Function: Returns in X the sine value of <exprN> (exprN must be expressed in radians).

SOUND (statement, 1)
Format: SOUND <register number>, <data>
Function: Writes the value of <data> to the PSG register. <register number> can range from 0 to 13 and <data> from 0 to 255.

SPACE$ (function, 1)
Format: X$ = SPACE$ (<exprN>)
Function: Returns a string with <exprN> spaces in X$.

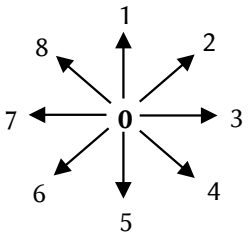
SPC (function, 1)
Format: PRINT SPC (<exprN>)
Function: Prints <exprN> spaces.

SPRITE (statement, 1)
Format: SPRITE {ON | OFF | STOP}
Function: Enables, disables or suspends interruption due to sprite collision.

SPRITE$ (statement or system variable, 1)
Format: SPRITE$ (<sprite number>) = <expr$>
X$ = SPRITE$ (<sprite$>)
Function: Sets or reads the sprites pattern.

SQR (function, 1)
Format: X = SQR (<exprN>)
Function: Returns in X the square root value of <exprN>.

STICK (function, 1)
Format: X = STICK (<joystick port number>)
Function: Examines the direction of the joystick and loads the result in X.
<joystick port number> – 0 ⟶ Keyboard
1 ⟶ port #1
2 ⟶ port #2
The "X" value is illustrated below, according direction:



STOP (statement, 1)
Format: STOP
Function: It paralyzes the execution of a program.
Format: STOP {ON | OFF | STOP}
Function: Enables, disables or supposes interruption by pressing the CTRL+STOP keys.

STRIG (function / declaration, 1)
Format: X = STRIG (<joystick port number>)

Function: Examines the state of the trigger buttons and returns the result in X. The value will be –1 if it is being pressed or 0 otherwise.
<joystick port number> can be:
0 = Space bar
1 = joystick on port 1, button A
2 = joystick on port 2, button A
3 = joystick on port 1, button B
4 = joystick on port 2, button B

Format: STRIG (<joystick port number>) {ON | OFF | STOP}
Function: Enables, disables or suspend interruption by pressing the trigger buttons.

STR$ (function, 1)
Format: X$ = STR$ (<exprN>)
Function: Converts the value of <exprN> to a decimal string and returns the value obtained in X$.

STRING$ (statement, 1)
Format: X$ = STRING$ (<exprN1>, {<expr$> | <exprN2})
Function: Returns in X$ a string of length <exprN1>, where all characters are equal, formed by the first character of <expr$> or by the character whose ASCII code is represented by <exprN2>.

SWAP (declaration, 1)
Format: SWAP <variable name>, <variable name>
Function: Exchanges the content of two variables. The variables must be of the same type.

TAB (function, 1)
Format: PRINT TAB (<exprN>)
Function: Produces <exprN> spaces for the PRINT instructions.

TAN (function, 1)
Format: X = TAN (<exprN>)
Function: Returns in X the tangent value of <exprN> (exprN must be expressed in radians).

TIME (system variable, 1)
Format: X = TIME
Function: Returns the value of TIME (it is an integer variable that is continuously incremented 60 times per second in NTSC machines and 50 times in PAL machines).
Format: TIME = <exprN>
Function: Assigns the value of <exprN> to the TIME variable. It must be an integer value.

TROFF (command, 1)
Format: TROFF
Function: Turn off the line tracking of the running program.

TRON (command, 1)
Format: TRON
Function: Turn on the line tracking of the running program.

USR (function, 1)
Format: X – USR [<number>] (<argument>)
Function: Executes an assembly routine. <number> can be a value from 0 to 9.

VAL (function, 1)
Format: X = VAL (<expr$>)
Function: Converts <expr$> to a numeric value and stores it in X.

VARPTR (function, 1-D)
Format: X = VARPTR (<variable name> | #<file number>)
Function: Returns in X the address where the <variable> is stored or the FCB address of <file number>

VDP (system variable, 1-2-3)
Format: X = VDP (<register number>)
VDP (<register number>) = <data>
Function: Read or write data in a VDP register. <data> must be a numeric value between 0 and 255.

VPEEK (function, 1-2)
Format: X = VPEEK (<address>)
Function: Returns in X the content of the VRAM byte specified by <address>.

VPOKE (statement, 1-2)
Format: VPOKE <address>, <data>
Function: Writes a data byte in the <address> of the VRAM. <data> must be a numeric value between 0 and 255.

WAIT (statement, 1)
Format: WAIT <port number>, <exprN1> [,<exprN2>]
Function: Stops the execution of the program until the specified port value matches the value of <exprN1> or <exprN2>.

WIDTH (declaration, 1-2)
Format: WIDTH <number>
Function: Specifies the number of characters per line in text modes (Screen 0 and 1).

XOR (logical operator, 1)
Format: <exprA1> XOR <exprA2>
Function: Performs logical XOR operation between <exprA1> and <exprA2>.

```
0 xor 0 → 0      1 xor 0 → 1
0 xor 1 → 1      1 xor 1 → 0
```

3.3 – EXTENDED COMMANDS

The CALL command allows the MSX-BASIC instructions to be expanded indefinitely, allowing access to new devices in cartridges or new features. Below is described a majority of the instructions available through the CALL command.

DM-System2 BASIC
(Installable extension for BASIC)

BGMOFF	BLOCK	COS	FILES
BGMON	CALL	DMM	FSIZE
BGMTMP	CELLO	DMMINI	HELP
BGMTRS	CHGCPU	DMMOFF	HMMM
BGMVOL	CHGDRV	DMMON	HMMV
BGMWAIT	CHGPLT	EXT	INTWAIT
BINLOAD	COLOR=	EXTCOPY	KBOLD

KCOLOR	PACSAVE	SEOFF	UPPER
KINIT	PAUSE	SEON	VCOPY
KPRINT	PCMON	SETBIN	VDPWAIT
KPUT	PEEK	SETPLT	VMOFF
KSIZE	PEEKS	SETSE	VMON
LMMM	PEEKW	SIN	VMWAIT
LMMV	POKE	STATUS	WAIT
LOAD	POKES	SYSOFF	XY
MALLOC	POKEW	SYSON	YMMM
PACLOAD	SAVE	SYSTEM	

FM-X BASIC
(Available with the Fujitsu MB22450 interface when inserted into the FM-X expansion slot)

CALL MON CALL PRINTERSETUP

FormatMaster-BASIC
(Available with an installable extension that comes on the Future Magazine Extra disc. FORMAT.BIN – FORMAT.MEM – FORMAT.TXT files)
CALL FORMAT

GR8NET BASIC
(Available with the installation of the GR8NET cartridge)

DSK	FLUPDATE	NETDUMP	NETGETMAP
DSKCFG	NET	NETEXPRT	NETGETMASK
DSKFMT	NETBITOV	NETFIX	NETGETMD
DSKGETIMG	NETBLOAD	NETFKOPLLR	NETGETMEM
DSKLDIMG	NETBROWSE	NETFWUPDATE	NETGETMIX
DSKHELP	NETBTOV	NETGETCLK	NETGETMMV
DSKLDIMG	NETCDTOF	NETGETCLOUD	NETGETNAME
DSKSETIMG	NETCFG	NETGETDA	NETGETNTP
DSKSVIMG	NETCODE	NETGETDNS	NETGETOPL
DSKSTATE	NETDHCP	NETGETGW	NETGETPATH
FLINFO	NETDIAG	NETGETHOST	NETGETPORT
FLLIST	NETDNS	NETGETIP	NETGETPSG

NETGETQSTR	NETRESST	NETSETMEM	NETSTAT
NETGETTSHN	NETSAVE	NETSETMIX	NETSYSINFO
NETGW	NETSDCRD	NETSETMMV	NETRCHKS
NETHELP	NETSETCLK	NETSETNAME	NETTELNET
NETIMPRT	NETSETCLOUD	NETSETNTP	NETTERM
NETIP	NETSETDA	NETSETOPL	NETTGTMAP
NETLDBUF	NETSETDM	NETSETPATH	NETTSYNC
NETLDRAM	NETSETDNS	NETSETPORT	NETVARBRSTR
NETMASK	NETSETGW	NETSETPSG	NETVARBSIZE
NETNTP	NETSETHOST	NETSETQSTR	NETVARRWTH
NETPLAYBUF	NETSETIP	NETSETTSHN	NETVARUDTO
NETPLAYVID	NETSETMAP	NETSNDDTG	NETVER
NETPLAYWAV	NETSETMASK	NETSNDVOL	

Hangul-BASIC
(Disponível em micros coreanos. Veja também PUT HAN, SET HAN e SET SYSTEM)

CLS	HELP	KLEN	REBOOT
ENG	KCHR	KMID	RTCINI
FONT	KCODE	KTYPE	VER
HANOFF	KEXT	MODE9	
HANON	KINSTR	PALETTE	

Hitachi-BASIC
(Disponível em alguns micros Hitachi)
- A versão 1 está disponível no modelo MB-H1 (MSX1)
- A versão 2 está disponível no modelo MB-H2 (MSX1)
- A versão 3 está disponível no modelo MB-H3 (MSX2)

AUTOMUTE	CMT	MON	REW
BLSCAN	CSCAN	MUTE	SCOPY
CCOPY	CSCOPY	NSCAN	STDBY
CDCOPY	FF	PAUSE	STOP
CFILES	HCOPY	PLAY	TABOFF
CHCOPY	IDTRACE	REC	TABON

Kanji-BASIC
(Disponível em micros japoneses MSX2+ ou superior. Veja também PUT KANJI)

AKCNV	KACNV	KLEN	PALETTE
ANK	KANJI	KMID	SJIS
CLS	KEXT	KNJ	
JIS	KINSTR	KTYPE	

Mega Assembler
(Disponível com a instalação do cartucho Mega-Assembler)

ASM	START

MSX-Aid BASIC
(Disponível com a instalação do cartucho MSX-AID)

CFILES	MESSAGE	TRACE ON
FIND	MON	VARLIST
HELP	TRACE OFF	XREF

MSX-Audio BASIC
(Disponível com a instalação de cartuchos com o MSX-Audio BIOS)

APEEK	COPY PCM	MK VOICE	RECMOD
APOKE	INMK	MK VOL	REC PCM
APPEND MK	KEY OFF	PCM FREQ	SAVE PCM
AUDIO	KEY ON	PCM VOL	SET PCM
AUDREG	LOAD PCM	PITCH	STOPM
BGM	MK PCM	PLAY	TEMPER
CONT MK	MK STAT	PLAY MK	TRANSPOSE
CONVA	MK TEMPO	PLAY PCM	VOICE
CONVP	MK VEL	REC MK	VOICE COPY

MSX-Music BASIC
(Disponível através de cartuchos ou internamente)

AUDREG	MUSIC	STOPM	VOICE
BGM	PITCH	TEMPER	VOICE COPY
MDR (*)	PLAY	TRANSPOSE	

(*) Disponível apenas no MSX turbo R FS-A1GT

Network-BASIC
(Extensão BASIC disponível apenas nos computadores MSX2 Yamaha YIS-503IIIR e YIS-805/128R2, usados em escolas da União Soviética)

BRECEIVE	NETEND	PON	SNDRUN
BSEND	NETINIT	RCVMAIL	STOP
CHECK	OFFLINE	RECEIVE	TALK
DISCOM	ONLINE	RUN	WHO
ENACOM	PEEK	SEND	
HELP	POFF	SNDCMD	
MESSAGE	POKE	SNDMAIL	

NewModem-BASIC
(Disponível para os modems Philips NMS 1255 e Micro Technology MT-Telcom II)

ANSWER	FILEOUT	LOGFILE	RECFILE
BREAK	GET	LS	RTSOFF
CARRIER	INIMDM	MC	RTSON
CHKMDM	INITMD	MRING	SENDFILE
CONNECT	LINEOFF	MSTART	SPEAKEROFF
DTROFF	LINEON	MSTOP	SPEAKERON
DTRON	LB	OFFHOOK	TDIAL
ECHOOFF	LEN	ONHOOK	TERMINAL
ECHOON	LO	PDIAL	

Nextor-BASIC
(Disponível através de cartuchos IDE com Nextor)

CURDRV	DRVINFO	MAPDRV	NEXTOR
DRIVERS	LOCKDRV	MAPDRVL	USR

Pioneer-BASIC (P-BASIC)
(Disponível nos micros da Pioneer PX-7, PX-V7 e PX-V60)

BLIND	FRAME	MUTE	SEARCH
CHAPTER	FRAME OFF	PAN	SYMBOL
CHAPTER OFF	IMPOSE	REMOTE	VIDEO
DEF UNIV	LCOPY	SCLOAD	
EXTV	LD	SCSAVE	

Printer-BASIC
(Disponível nos micros da Toshiba HX20 até HX23F e HX31 até HX34)

LCOPY	SPOLOFF	SPOLON

QuickDisk BASIC
(Disponível nos micros Daewoo (versão 1.0) e Casio, Philips e Sanyo (versão 1.1))

BLOAD	LOAD	QDFORMAT	RUN
BSAVE	MERGE	QDKEY	SAVE
CASQD	QDFILES	QDKILL	

RMSX BASIC
(Extensão que vem com o emulador RMSX para o MSX Turbo R)

?	CHCAS	FILES	LICENSE
CASAUTOREW	CHDIR	HELP	MUTE
CASREW	CHDSK	HZ	PALETTE
CASRUN	EXIT	IOSOUND	RESET

RookieDrive BASIC
(Disponível com a instalação do cartucho RookieDrive)

CREATEDISK	HELP	REBOOT	USBRESET
EJECT	INSERTDISK	USBCD	
FNAME	LOADROM	USBERROR	
FORMAT	MOUNT	USBFILES	

SFG-BASIC
(Disponível com a instalação do cartucho Yamaha FM Music Macro)

CANCEL	PATTERN	STOP
CLDVOICE	PHRASE	SYNCOUT
ERASE	PLAY	TEMPO
EVENT ON/OFF/STOP	RCANCEL	TIMER
INIT	REPORT	TRACK
INMKEY	RHYTHM	TRANSPOSE
INST	RSTOP	TSTOP
LENGTH	SELPATTERN	TUNE
LFO	SELVOICE	USERHYTHM
LOOK	SOUND	VLIST
MODINST	STANDBY	WAIT
ON EVENT...GOSUB	START	

StudioFM BASIC
(Disponível com a instalação do StudioFM para tocar músicas .MUS geradas pelo FAC Soundtracker. Usar BLOAD"SFMDRV1.BIN",R)

MFADE	MLOAD	MPLAY	MSTOP

SVI-Modem BASIC
(Disponível apenas no modem Spectravideo SVI-737)

COMBREAK	COMOFF	COMTERM	TDIAL
COMDTR	COMON	OFFLINE	
COM...GOSUB	COMSTAT	ONLINE	
COMINI	COMSTOP	PDIAL	

X-BASIC
(Disponível com a instalação do compilador em tempo real X-BASIC)

'#C	CALL BC	CALL TURBO ON
'#I	CALL RUN	CALL TURBO OFF
'#N		

3.3.1 – Commands Description

? (statement, RMSX-BASIC)
Format: CALL ?
Function: Displays help for RMSX-BASIC. It is equivalent to the CALL HELP command.

ADJUST (command, Daewoo)
Format: CALL ADJUST
Function: Enables the internal lightpen interface. Available only for MSX2 manufactured by Daewoo.

AKCNV (statement, Kanji-BASIC)
Format: CALL AKCNV (<variable>,"<character string>")
Function: Converts single-byte characters to 2-byte Kanji.
<string variable> receives the converted characters.
<character string> are the ASCII characters to be converted.

ANK (statement, Kanji-BASIC)
Format: CALL ANK
Function: Exits Kanji mode (the memory used by the Kanji driver is not released).

ANSWER (function, New Modem BASIC)
Format: CALL ANSWER (<speed>)
Function: Detects the speed of a connection. This instruction works only in BBS programs. The detected speeds are:

Value	Standard	Recep. speed	Transm. speed
1	V21	300 baud	300 baud
2	V23	1200 baud	75 baud
3	V23	75 baud	1200 baud

APEEK (function, MSX-Audio)
Format: CALL APEEK (<address>, X)
Function: Returns in X the value of the byte corresponding to the memory address of MSX-Audio. The address can range from 0000H to 7FFFH.

APOKE (statement, MSX-Audio)
Format: CALL APOKE (<address>, <data>)
Function: Writes a byte of data to the <address> of the audio memory. <data> must be a numeric value between 0 and 255. The address can range from 0000H to 7FFFH.

APPEND MK (statement, MSX-Audio)
Format: CALL APPEND MK (<matrix name>)
CALL APPEND MK (<start address>, <end address>)
CALL APPEND MK (A), where the sequence A must be previously declared in the DIM and REC MK instructions.
Function: Adds an additional recording played on the musical keyboard.

ASM (command, Mega Assembler)
Format: CALL ASM
Function: Calls the Mega Assembler without initializing the variables. To call the MA by initializing the variables, use CALL START.

AUDIO (statement, MSX-Audio)

Format: CALL AUDIO (<mode>, <channels with instruments>, <channels for string 1>, <channels for string 2>, ...,<channels for string 9>)

<mode> defines the use of MSX-Audio. The default is 1.

Mode	FM melody	PCM	FM rhythm	Type
0	9 voices			Normal
1	6 voices		3 voices	Normal
2	9 voices	1 voice		Normal
3	6 voices	1 voice	3 voices	Normal
4	9 voices			CSM
5	6 voices		3 voices	CSM
6	9 voices	1 voice		CSM
7	6 voices	1 voice	3 voices	CSM

In CSM mode, the control of all FM sounds (melody and rhythm) is invalid. CSM stands for Composite Sinusoidal Modeling. Using all operators in parallel, this mode can be used to synthesize speech.

<channels with instruments> defines how many channels will be assigned to an instrument.

<channels for string n> defines how many channels will be used for each string related to the FM melody in the PLAY instruction.

AUDREG (declaration, MSX-Music and MSX-Audio)

Format: CALL AUDREG <register>, <data> [,<channel>]

Function: Writes the value of <data> in the OPLL or MSX-Audio register. <channel> specifies the channel to be used (MSX-Audio only). It can be 0 or 1, the default is 0.

Note: Previous use of CALL MUSIC or CALL AUDIO is required.

AUTOMUTE (command, Hitachi-BASIC version 2)

Format: CALL AUTOMUTE

Function: Adds a 4 second pause before activating the internal data reader of some Hitachi computers.

BGM (declaration, MSX-Music and MSX-Audio)
Format: CALL BGM (n)
Function: Arrow executing commands while the music is being played. <n> can be 0 or 1, as below:
0 – No commands can be executed during the music.
1 – Commands can be executed during music (default).

BGMOFF (declaration, DM-System2 BASIC)
Format: CALL BGMOFF (<fade>)
Function: Mutes the music played by OPLL / MSX Music. Requires BGM driver.
<fade> – 1 ⟶ without fade (immediate stop)
2 ⟶ with fade out

BGMON (declaration, DM-System2 BASIC)
Format: CALL BGMON (<start address> [,<num of repetitions>])
Function: Plays music using the BGM driver. Requires BGM driver.
<start address> is a pointer to the BGM data in the Main RAM.
<repetition number> is the number of times the song will be played. "0" indicates infinite repetitions.

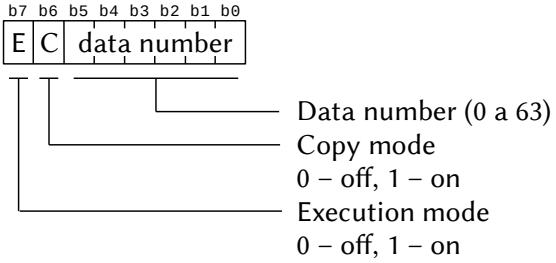
BGMTMP (declaration, DM-System2 BASIC)
Format: CALL BGMTMP (<time>)
Function: Adjusts the "tempo" of the song. Requires BGM driver.
<time> is a value between 0 and 255 representing the percentage. The default value is 100.

BGMTRS (declaration, DM-System2 BASIC)
Format: CALL BGMTRS (<transpose>)
Function: Adjusts the key of the music. Requires BGM driver.
<transpose> is a one-byte value between –128 and +127. The default value is 0.

BGMVOL (declaration, DM-System2 BASIC)
Format: CALL BGMVOL ([<Master>] [,<OPLL>] [,<PSG>] [,<SCC>])
Function: Adjusts the volume of the various sound generators individually. It can vary from 0 to 15 for each one and the default value for all is 15. Requires BGM driver.

BGMWAIT (declaration, DM-System2 BASIC)
Format: CALL BGMWAIT
Function: Pause or restart the BGM. Requires BGM driver.

BINLOAD (command, DM-System2 BASIC)
Format: CALL BINLOAD (<flag> [,<flag 2>] [,<destination address>] [,<size>])
Function: Transfers concatenated data from the binary table in VRAM to RAM, being able to execute it.



<flag 2> – Value of a byte that replaces the same flag in the binary table. By default, the table flag is used.
<destination address> – 2-byte value that specifies the initial destination address for the data.
<size> – 2-byte value that specifies the number of bytes to be transferred.
Obs. – <destination address> and <size> must be omitted when the data format has the same format as the COPY instruction (the first byte is the X coordinate and the second is the Y coordinate).

BLIND (statement, Pioneer-BASIC)
Format: CALL BLIND ([<string>], [S | L])
Function: Delete or re-enable the display of the screen. Works only on Screen 2.
<string> can range from 0 to 9 and specifies the sequence in which the screen will be cleared or enabled.
S – Saves the screen while it is being erased
L – Loads the screen previously saved by "S".

BLOAD (command, QuickDisk BASIC)
Format: CALL BLOAD ("[QD [n]:]"<filename>" {[,R] | [,S]} [, offset])
Function: Load the binary code from the QuickDisk device.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8.3 characters.
", R" automatically executes the binary code of the loaded file
", S" uploads the content to VRAM.
<displacement> indicates that the program will be loaded at the start address + offset. This parameter also affects the execution address.

BLOCK (command, DM-System2 BASIC)
Format: CALL BLOCK ([@]<source address>, [@]<destination address>, <size>)
Function: Copy data between Main RAM and VRAM. If the <address> is preceded by "@", VRAM will be specified. To avoid error messages, decimal numbers should be used for addresses greater than FFFFH (65 535).

BLSCAN (command, Hitachi-BASIC version 2)
Format: CALL BLSCAN
Function: Makes the internal data reader of the Hitachi micro MB-H2 searches for files recorded with BSAVE and that can be loaded with BLOAD.

BREAK (command, New Modem BASIC)
Format: CALL BREAK
Function: Assign the interrupt call to the CODE key. This will allow you to interrupt the RING and DIAL routines just by pressing the CODE key.

BRECEIVE (command, Network-BASIC)
Format: CALL BRECEIVE ([[<unit name>:]<filename>] [,<student number>] [,<start adress>] [,<end adress>] [,S])

Function: Receives binary data in RAM or VRAM from (other) students' computers. This instruction can be used by the teacher and students who have been authorized by the teacher through CALL ENACOM. The short version _BREC can be used. <unit name> can be "A:" or "B:"; <student number> can range from 0 to 15; <start address> and <end address> can range from &H0000 to &HFFFF and ", S" specifies VRAM.

BSAVE (command, QuickDisk BASIC)

Format: CALL BSAVE ("[QD [n]:]<filename>",<start address>, <end adress> [,<run adress>])
CALL BSAVE ("[QD [n]:]<filename>",<start address>, <end address>, S)

Function: Saves a memory area on the specified QuickDisk device.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8.3 characters.
<start address>, <end address> and <run address> can vary from &H0000 to &HFFFF. If <run address> is omitted,<start address> will be used instead.
",S" is used to save VRAM content.

BSEND (command, Network-BASIC)

Format: CALL BSEND ([[<unit name>:]<filename>] [,<student number>] [,<starting address>] [,<end address>] [, S])

Function: Sends binary data from RAM or VRAM from (other) students' computers. See BRECEIVE for more information.

CALL (declaration, DM-System2 BASIC)

Format: CALL CALL (<address> [,<AF>] [,<HL>] [,<DE>] [,<BC>] [,<IX>] [,<IY>])

Function: Calls a machine language routine in Main-RAM, unless the address is less than 2000H (below that it will be called Main-ROM to allow access to BIOS routines). If <AF> is less than 256, the value will be loaded into register A.

CANCEL (declaration, SFG-BASIC)
Format: CALL CANCEL (<instrument number>)
Function: Cancels an instrument. <instrument number> can vary from 1 to 4. Short version: _CANC.

CARRIER (statement, New Modem BASIC)
Format: CALL CARRIER (: GOSUB <line number>)
Function: Specifies the GOSUB routine to be performed when the operator is absent for an unknown reason or because the caller has just hung up. This instruction is only useful in BBS programs.

CASAUTOREW (command, RMSX-BASIC)
Format: CALL CASAUTOREW [ON] | [OFF]
Function: Enables or disables the automatic rewinding of a tape image (CAS file) back to the beginning. Without a parameter, this instruction switches between the two options.
ON – Enables automatic rewinding of the tape image.
OFF – Disables automatic rewinding

CASQD (command, QuickDisk BASIC)
Format: CALL CASQD [("[CAS:]" <filename1> "] [,"[QD [n]:] [" <filename2>] ")]
Function: Transfer the specified file from cassette to QuickDisk. Without parameters, this command transfers the tape file to the standard QuickDisk device with the same name.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename1> is the name of the file to be copied from the tape.
<filename2> is the name of the file to be written to the Quick Disk. The format is limited to 6 characters with no extension. If <filename2> is omitted,<filename1> is repeated.

CASREW (command, RMSX-BASIC)
Format: CALL CASREW
Function: Manually rewind a tape image (CAS file) back to the beginning.

CASRUN (command, RMSX-BASIC)
Format: CALL CASRUN [("[<drive letter>:] [\<path> \]
<filename.CAS>]")]
Function: Load and execute files contained in the specified tape image (CAS file). If <filename.CAS> is omitted, the first file in the image inserted with CALL CHCAS will be executed. <Drive letter>: can go from A: to H:.

CCOPY (command, Hitachi-BASIC version 3)
Format: CALL CCOPY
Function: Sends to the printer a darker copy of a graphic screen in Screens 2, 4 or 5 simulating shades of gray.

CDCOPY (command, Hitachi-BASIC version 3)
Format: CALL CDCOPY
Function: Sends to the printer a copy of a graphical screen in Screens 2, 4 or 5 using only white and black dots.

CELLO (declaration, DM-System2 BASIC)
Format: CALL CELLO (<X0>, <Y0>) – [STEP] (<X1>, <Y1>) [,<n0>]
[,<n1>] [,<n2>]
Function: Changes the colors of a specified rectangular area of the screen. Does not work on Screens 0 to 4.
Screens 5, 6, 7: Replaces one color with another
Screen 8: Modify colors according to RGB
Screen 10, 11, 12: Modify the Y values of the colors
<X0>, <Y0> coords of the starting corner of the rectangle
<X1>, <Y1> coordinates of the final corner of the rectangle
<n0> color to replace in Screens 5 to 7 (0~15);
value to add to the red in Scr 8 (–7~+ 7);
value to add Y to Scr 10 to 12 (–31~+ 31).
<n1> new color for Screens 5 to 7 (0~15);
value to add to green on Screen 8 (–7~+ 7).
<n2> value to add to the blue on Screen 8 (–7~+ 7).

CFILES (command, Hitachi-BASIC, MSX Aid BASIC)
Format: CALL CFILES [Hitachi-BASIC 2]
Function: Lists the contents of the tape inserted in the internal data reader of some Hitachi computers. It is recommended to rewind the tape with the CALL REW command first.

Format: CALL CFILES [MSX Aid BASIC]
Function: Lists the contents of the tape inserted in the data reader connected to the MSX. The list specifies whether a file is binary (OBJ), BASIC (BAS) or ASCII (ASC). It also returns the size of the binary files. It is recommended to rewind the tape first.

CHAPTER (statement, Pioneer-BASIC)
Format: CALL CHAPTER (<chapter num>, GOSUB <line num>)
CALL CHAPTER OFF
Function: Specifies the subroutine line number that will be executed when the chapter <chapter number> is reached. <chapter number> should be in the range between 50 and 54,000. If "OFF" is specified, cancel the line number assignment. This command is specific for use with the Pioneer Laser Vision Player LD-700 and cannot be used in conjunction with the FRAME command.

CHCAS (control, RMSX-BASIC)
Format: CALL CHCAS ("[<drive letter>:] [\<path>\]<filename.CAS>")
Function: Assembles (inserts) the specified tape image (CAS file) in the virtual cassette player of the MSX1 or MSX2 computer emulated on a Turbo R with the rMSX emulator.
<drive letter>: can range from A: to H :.

CHCOPY (command, Hitachi-BASIC version 3)
Format: CALL CHCOPY
Function: Sends to the printer a lighter copy of a graphic screen in Screens 2, 4 or 5 simulating shades of gray.

CHDIR (declaration, Disk-BASIC 2nd version, RMSX-BASIC)
Format: CALL CHDIR ([<drive letter>:] [\]<path>) [Disk-BASIC 2]
Function: Change subdirectory. The argument can also be just ".." or "." to return directories.
Format: CALL CHDIR ([<drive letter>:] [\]<path>) [RMSX-BASIC]
Function: Change the current working directory of an actual disk on an MSX Turbo R drive, used as a host for an MSX1/MSX2 computer emulated on this machine with the rMSX emulator. The argument can also be just ".." or "." to return directories.

CHDRV (command, Disk-BASIC 2nd version)
Format: CALL CHDRV (<drive letter> :)
Function: Change the drive according to "<drive letter>:". If Nextor is installed, the argument can be replaced with a number (1 = A:, 2 = B:, etc.); otherwise, you must indicate the drive letter (A: up to H:).

CHKDSK (command, RMSX-BASIC)
Format: CALL CHKDSK (see parameters below)
Function: Mount (insert) the specified disk image (DSK file) on the virtual disk drive of the MSX1 / MSX2 computer emulated on a Turbo R with the rMSX emulator and / or activate a specified disk number (the Turbo disk drive R can also be used with a real disc).
→ To mount the disk image to the current disk number (if the parameter is not provided or is empty, the actual disk unit will be used)
CALL CHDSK [("[<drive letter>:] [\<path>\] <filename.DSK> ")]
→ To mount the disk image on the specified disk number (without activation)
CALL CHDSK ("[<drive letter>:] [\<path>\] <filename.DSK> "),<disk number>
→ To activate the specified disc number and eventually mount the disc image on it
CALL CHDSK (<drive letter>), [("[<device name>:] [\<path>\]<filename.DSK>")]

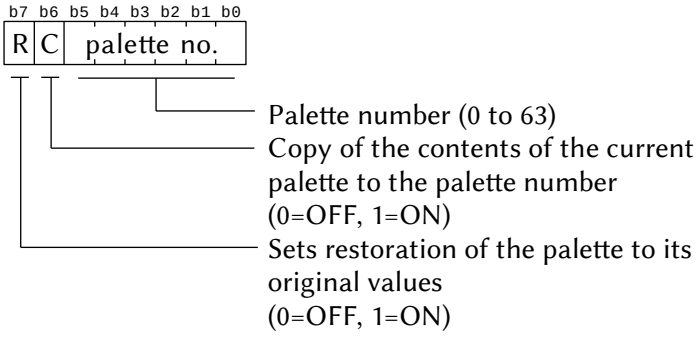
CHECK (command, Network-BASIC)
Format: CALL CHECK ([<connection variable>] [,<communication variable>])
Function: Checks which students are connected to the network and / or which students are able to communicate with others. This instruction is only available to the teacher. <connection variable> contains the binary representation of connected / unconnected students. <communication variable> contains the binary representation of students with extended communication enabled or disabled. Both are 16-bit integer variables, where bit 0 is associated with

student 1, bit 1 is associated with student 2 and so on, up to bit 14. "0" means connected or active student and "1" means disconnected or inactive student.

CHGCPU (command, DM-System2 BASIC)
Format: CALL CHGCPU ([<mode>] [,<variable>])
Function: Switch or return CPU mode on MSX turbo R.
<mode> – Mode to be applied
<variable> – Value before being changed to <mode>
The two parameters have the following format:

CHGDRV (command, DM-System2 BASIC)
Format: CALL CHGDRV ([<drive number>] [,<variable>])
Function: Change or return the current drive unit.
<drive number> – Must be a number between 1 and 8, where 1 = A :, 2 = B :, etc.
<variable> – Numeric variable that will receive the file size.

CHGPLT (declaration, DM-System2 BASIC)
Format: CALL CHGPLT (<number>)
Function: Change the colors of the palette. The palette data must be previously placed in RAM.
<number> is a one-byte value with the following format:



CKHMDM (function, New Modem BASIC)
Format: CALL CHKMDM (<numeric variable>)
Function: Checks whether the modem is present. If <numeric variable> is 0, the modem has been detected, otherwise there is no modem.

CLDVOICE (command, SFG-BASIC)
Format: CALL CLDVOICE [((<display name>), (<device>))]
Function: Loads the voice data into memory, from the cassette or the cartridge memory. <display name> specifies whether to display the voice name during loading. If it is "0" the name will not be displayed and if it is "1" it will be displayed. <device> can be: 0 – Cassette; 1 – Cartridge.
Short version: _CLDV.

CLS (declaration, Kanji-BASIC and Hangul-BASIC 4)
Format: CALL CLS
Function: Clears the screen in Kanji mode.

CMT (command, Hitachi-BASIC version 2)
Format: CALL CMT
Function: Starts the "Tape Utility" on the Hitachi MB-H2 computer.

COLOR= (declaration, DM-System2 BASIC)
Format: CALL COLOR = (<palette number>, <red level>, <green level>, <blue level>)
Function: Change the colors of the palette to a single color. The level can vary from 0 to 7 for each primary color. Changes are stored only in (NEWPLT) and not in the VRAM palette table.

COMBREAK (remote, BASIC Modem, SVI BASIC Modem)
Format: CALL COMBREAK ([<port number> [,<number of characters>]])
Function: Sends instruction to block messages. <port number> can range from 0 to 4 and <character number> to be blocked can range from 3 to 32767.

COM GOSUB (declaration, Modem BASIC, SVI Modem BASIC)
Format: CALL COM ([<port number> GOSUB <line number>])
Function: Specifies the subroutine that will be called when an interruption occurs in RS232-C. <port number> can vary from 0 to 4; if omitted it will be 0. The subroutine will start at the specified <line number>.

COMINI (command, Modem BASIC, SVI Modem BASIC)
Format: CALL COMINI ([<data>] [,<reception speed>] [,<transmission speed>] [,<timeout>])
Function: Initializes the modem with the <data> provided.
<data> is an alphanumeric string of up to 10 characters, the default of which, if omitted is "0: 8N1XHNNN". The starting number is the RS232C port followed by ":" and the following characters represent:
3rd - word size (5 to 8) or "del"
4th - parity (E-even, O-odd, I-ignore, N-without parity) or "ins"
5th - size. stop bit: 1- 1 bit, 2- 1.5 bits, 3- 2 bits
6th - XON / XOFF: X- xon, N- xoff
7th - H- handshaking, N- without handshaking
8th - LF: A- inserts LF, N- does not insert LF
9th - LF: A- delete LF, N- do not delete LF
10th – Shift in / out: S- enable, N- disabled
<reception speed> can vary from 50 to 1200. Valid values are: 50, 75, 110, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19 200. If omitted, it will assume 1200.
<transmission speed> can vary from 50 to 1200. If omitted, it assumes the same as <speed. reception>.
<timeout> is specified in seconds and can range from 0 to 255. If omitted, it will assume 0.

COMOFF (remote, BASIC Modem, SVI BASIC Modem)
Format: CALL COMOFF ([<"port number:">])
Function: Disables the interruption coming from the RS232-C port.

COMON (command, Modem BASIC, SVI Modem BASIC)
Format: CALL COMON ([<"port number:">])
Function: Enables the interruption coming from the RS232-C port.

COMSTAT (function, Modem BASIC, SVI Modem BASIC)
Format: CALL COMSTAT ([<"port number:">],<integer variable>)
Function: Returns the status of the RS232-C port.
<"Port number:"> should vary and 0: to 4 :. If omitted, it will be 0 :.

<integer variable> returns the following values:

bit 15: Receive buffer overflow error
0- no error, 1- error occurred

bit 14: Timeout error
0- no error, 1- error occurred

bit 13: Framing error (binary bit "0" was received instead of the stop bit.)
0- no error, 1- error occurred

bit 12: Saturation error (data received before the receive buffer is empty)
0- no error, 1- error occurred

bit 11: Parity error
0- no error, 1- error occurred

bit 10: Pressing [CTRL] + [STOP]
0- not pressed, 1- pressed

bit 9: Reserved

bit 8: Reserved

bit 7: CS signal status (CTS)
0- off, 1- on

bit 6: Timer/counter set for timeout error detection
0- not defined, 1-defined

bit 5: Reserved

bit 4: Reserved

bit 3: DR signal status (DSR)
0- off, 1- on

bit 2: Stop sequence detected while COMSTAT is executed
0- not detected, 1- detected

bit 1: Reserved

bit 0: CD signal status
0- off, 1- on

COMSTOP (command, Modem BASIC, SVI Modem BASIC)

Format: CALL COMSTOP ([<"port number:">])

Function: Suspend the interruption coming from the RS232-C port. <"Port number:"> should vary and 0: to 4:. If omitted, it will be 0:.

COMTERM (command, Modem BASIC, SVI Modem BASIC)
Format: CALL COMTERM ([<"port number:">])
Function: Puts the MSX in terminal mode. To exit terminal mode, press CTRL+STOP together. <"Port number:"> should vary and 0: to 4 :. If omitted, it will be 0 :. Once in terminal mode, use the following keys:
[SHIFT] + [F1] – Displays the received control codes.
[SHIFT] + [F2] – Displays the pressed keys.
[SHIFT] + [F3] – Displays and prints the pressed keys.
[STOP] – Press and hold to send the interrupt sequence to the host.

CONNECT (command, New Modem BASIC)
Format: CALL CONNECT (<numeric variable>)
Function: Establishes connection in a Terminal program, with the speed defined in CALL INIMDM.
<numeric variable> stores the result of the operation:
0 – An error occurred while trying to connect
1 – Operator detected – The modem is connected
2 – The routine was aborted by pressing the CODE key
3 – The phone number is busy

CONT MK (command, MSX-Audio)
Format: CALL CONT MK
Function: Continues a playback or recording of the musical keyboard that was canceled by the STOPM command.

CONVA (declaration, MSX-Audio)
Format: CALL CONVA (<source file>, <destination file)
Function: Convert PCM data to ADPCM data. <source file> and <destination file> are defined by a number that can vary from 0 to 15.

CONVP (declaration, MSX-Audio)
Format: CALL CONVP (<source file>, <destination file)
Function: Convert PCM data to ADPCM data. <source file> and <destination file> are defined by a number that can vary from 0 to 15.

COPY PCM (remote, MSX-Audio)
Format: CALL COPY PCM (<source file>, <dest file>, [<source offset>, [<file size>, [<dest offset>]]])
Function: Copies ADPCM and PCM data.
<source file> and <dest file> are defined by a number that can vary from 0 to 15.
<source offset> and <dest offset> define the offset in units of 256 bytes.
<file size> is the file size in bytes.

COS (function, DM-System2 BASIC)
Format: CALL COS (<variable>, <angle>, <value>)
Function: Returns the cosine of an angle. The result is obtained by multiplying the cosine of the angle by a numerical value.
<variable> – Numeric variable that will receive the result.
<angle> – Is the angle value in degrees.
<value> – Number of two bytes (integer value).

CREATEDISK (command, RookieDrive BASIC)
Format: CALL CREATEDISK (<disc name>)
Function: Creates a new disk image without formatting it. The created disk image is full of 0XFFH characters. Experimental instruction and not fully implemented (limited to 720 Kbytes disks).

CSCAN (command, Hitachi-BASIC version 2)
Format: CALL CSCAN
Function: Makes the internal data reader of the Hitachi micro MB-H2 searches for files recorded with CSAVE and that can be loaded with CLOAD.

CSCOPY (command, Hitachi-BASIC version 3)
Format: CALL CSCOPY (<c1> [,<c2>, <c3>, <c4>.... <c15>])
Function: Sends to the printer a copy of a graphic screen in Screens 2, 4 or 5 using a formula based on the selected colors. The difference with CALL SCOPY is unknown.

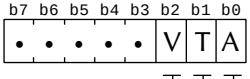
CURDRV (statement, Nextor)
Format: CALL CURDRV
Function: Displays the active drive unit.

DEF UNIV (command, Pioneer-BASIC)
Format: CALL DEF UNIV (<device number>, <device code>)
Function: Defines the device to be controlled by the REMOTE command. <device number> can range from 3 to 15 and <device code> can range from 1 to 255.

DISCOM (command, Network-BASIC)
Format: CALL DISCOM (<student number>)
Function: Disables the sending of messages from a student. This instruction is only available to the teacher. By default, after initialization, students can only send messages to the teacher. <student number> can vary from 1 to 15. Short version: _DISC.

DMM (function, DM-System2 BASIC)
Format: CALL DMM (<variable> [,<time>]) [,S])
Function: Performs the device entry and returns the result in <variable>. It can be aborted by CTRL+STOP. (Requires DEV driver).
<variable> must be numeric. The return values are:
0 – Not pressed 10 – GRAPH 13 – ESC
1 to 8 – 8 directions 11 – STOP 14 – HOME
9 – Space 12 – TAB 15 – SELECT
<time> that the command awaits, in units of 1/60 sec.
[, S] – If specified, the sprite defined in "CALL DMMINI" will be moved automatically.

DMMINI (declaration, DM-System2 BASIC)
Format: CALL DMMINI ([<mode>] [,<sprite number>])
Function: Defines the device entry. When DM-System2 is started, the mouse and joypad are configured as input devices. (Requires DEV driver).
<mode> is a one-byte value with the following format (default values are 0):



Action to be taken:
0 – coordinate update
1 – return value
Screen output:
0 – loop, 1 – don't move
Vertical loop range:
0 – 192/212, 1 – 256

<sprite number> is a 1-byte value that specifies the sprite displayed when running CALL DMM. If omitted,"0" is used.

DMMON (command, DM-System2 BASIC)
Format: CALL DMMON ([<address>])
Function: Activates continuous device verification, placing the result in the DM-System2 information area. <address> defines the execution address when a device event is detected. Requires DEV driver.

DMMOFF (command, DM-System2 BASIC)
Format: CALL DMMOFF
Function: Disables continuous device verification. Requires DEV driver.

DRIVERS (statement, Nextor)
Format: CALL DRIVERS
Function: Displays information about the drivers available for Nextor and MSXDOS.

DRVINFO (statement, Nextor)
Format: CALL DRVINFO
Function: Displays information about all available drive letters.

DSK (command, GR8NET-BASIC)
Format: CALL DSK
Function: Displays help and status and makes diagnostics.

DSKCFG (command, GR8NET-BASIC)
Format: CALL DSKCFG (<max num of pages>, <num of pages>)
Function: Get or manage the state of the disk image.
<max number of pages> is a variable that receives the maximum number of logical pages from RAM-Disk.
<number of pages> is a variable or constant that defines the size of the RAM-Disk. It must be between 0 and <max num of pages>.

DSKFMT (command, GR8NET-BASIC)
Format: CALL DSKFMT
Function: Initializes the image in RAM-Disk.

DSKGETIMG (function, GR8NET-BASIC)
Format: CALL DSKGETIMG [(<string variable>)]
Function: Gets the current location of the disk image and returns the path in the <string variable>.

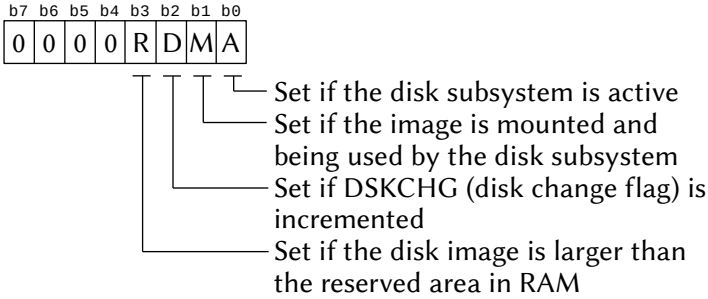
DSKHELP (declaration, GR8NET-BASIC)
Format: CALL DSKHELP
Function: Displays help for GR8NET.

DSKLDIMG (command, GR8NET-BASIC)
Format: CALL DSKLDIMG
Function: Load the current disk image into the GR8NET buffer.

DSKSETIMG (command, GR8NET-BASIC)
Format: CALL DSKSETIMG [(<path>)]
Function: Defines the location of the image according to <path>, which can be a string variable or an alphanumeric expression.

DSKSVIMG (command, GR8NET-BASIC)
Format: CALL DSKSVIMG [(<path>)]
Function: Saves the disk image of the GR8NET buffer to the SD card. If <path> is omitted, the path defined by DSKSETIMG will be used.

DSKSTATE (command, GR8NET-BASIC)
Format: CALL DSKSTATE (<status>, <flags>)
Function: Gets or sets disk subsystem.
<status> defines the state of the system. If it is 0, the disk will be disabled; if it is 1, it will be activated. The change will take effect on the next warm start of the system. Hardware boot will force the return to the default.
<flags> will return with the following values:



DTROFF (command, New Modem BASIC)
Format: CALL DTROFF
Function: Disables the DTR (Data Terminal Ready) signal.

DTRON (command, New Modem BASIC)
Format: CALL DTRON
Function: Enables the DTR (Data Terminal Ready) signal.

ECHOOFF (command, New Modem BASIC)
Format: CALL ECHOOFF
Function: Send characters to the phone line only. The screen will display only received characters.

ECHOON (remote, New Modem BASIC)
Format: CALL ECHOON
Function: Enables the sending of characters simultaneously to the telephone line and to the screen.

EJECT (command, RookieDrive BASIC)
Format: CALL EJECT
Function: Ejects the currently inserted disk image and deletes its name from the USBMSX.INI file.

ENACOM (command, Network-BASIC)
Format: CALL ENACOM (<student number>)
Function: Enables the sending of messages to a student. This instruction is only available to the teacher. By default, after initialization, students can only send messages to the teacher. <student number> can vary from 1 to 15. Short version: _ENAC.

ENG (command, Hangul-BASIC 3)
Format: CALL ENG
Function: Returns to Screen 0 text mode.

ERASE (command, SFG-BASIC)
Format: CALL ERASE (<track number>)
Function: Deletes the content of the specified track. <track number> can range from 1 to the number specified by CALL TRACK. Short version: _ERAS.

EVENT (command, SFG-BASIC)
Format: CALL EVENT ([<event number>]) ON | OFF | STOP
Function: Enables, disables or interrupts the interruption by event specified in ON EVENT... GOSUB. <event number> can be:
1~4 – Stops when playback of the specified instrument ends.
5 – Stops when rhythm playback ends.
6 – Interrupts according to the time programmed in the FM unit timer.
If <event number> is omitted, the command will be applied to all events. Short version: _EVEN.

EXIT (command, RMSX-BASIC)
Format: CALL EXIT
Function: Exits the rMSX emulator and returns to the 'normal' use of the MSX Turbo R computer.

EXT (command, DM-System2 BASIC)
Format: CALL EXT ([@]<source address>, [@]<dest address>)
Function: Extract compressed data in BPE format.
<source address> is the address of the compressed data.

<destination address> is the destination address of the unzipped data.
Note: If "@" is specified, it means VRAM.

EXTCOPY (command, DM-System2 BASIC)
Format: CALL EXTCOPY ([@]<source address>, <X>, <Y> [,<direction>]) [,<logical operator>]
Function: Unzips data in BPE format to a rectangular area on the screen.
<source address> – Address of the compressed data. If "@" is specified, it means VRAM (maximum 64K).
<X> – Horizontal destination coordinate (0 to 511)
<Y> – Vertical destination coordinate (0 to 1023)
<direction> – Is the unpacking direction on the screen.
0 – Right and down (default)
1 – Left and down
2 – Right and up
3 – Left and up
<logical operator> can be [T]PSET, [T]PRESET, [T]XOR, [T]OR or [T]AND. The default is PSET.

EXTV (remote, Pioneer-BASIC)
Format: CALL EXTV (<variable>)
Function: Checks if there is an external video signal at the input terminal and returns the result in <variable>, that can be:
0 – There is no video signal at the input
1 – External video signal detected

FF (command, Hitachi-BASIC version 2)
Format: CALL FF
Function: Puts the Hitachi MB-H2 computer's built-in data reader into fast search mode.

FILEOUT (command, New Modem BASIC)
Format: CALL FILEOUT ("[<device>:]<filename>]",<variable>)
Function: Send a text file or the typed text directly to a terminal.
<variable> contains an option on the question "More? (Y/ N)"and then stores a control code.

The values in <variable> can be:
0 – This option is on
1 – The option is off (required for ASCII upload)
Control codes
0 – Text was sent correctly
3 – The operation was aborted with CTRL+C or C
7 – The file was not found

FILES (declaration, DM-System2 BASIC, RMSX BASIC)
Format: CALL FILES ("[<device>:] [\ <path>] [[\]<filename>]", <variable>) [DM2-BASIC]
Function: Returns filenames and places them in the DM-System2's work area (address 7A00H). The filenames will be placed one after the other every 12 bytes.
<device> can be drive A: to H: or COM: for computers connected with RS232C.
<path> specifies the location of the folder or file
<filename> accepts wildcards (* and?)
<variable> is a numeric variable that will receive the number of files found.
Format: CALL FILES ("[<device>:] [\ <path>] [[\]<filename>]) [RMSX BASIC]
Function: Lists the contents of an actual disk in an MSX Turbo R drive, used to host an MSX1 or MSX2 emulation with the rMSX emulator.
<device> can be drive A: to H :.
<path> specifies the location of the folder or file.
<filename> accepts wildcards (* and ?).

FIND (command, MSX-Aid BASIC)
Format: CALL FIND ("<variable>" [, [<starting line number>], [<ending line number>], [P]])
Function: List part of the MSX-BASIC program that is in memory, where an alphanumeric variable or specific string is used. <variable> must be one or two characters. You can also specify the <starting line number> and <ending line number> of the BASIC program to be listed. If [P] is specified, the listing will be sent to the printer.

FLINFO (declaration, GR8NET-BASIC)
Format: CALL FLINFO
Function: Displays information about the serial flash memory.

FLLIST (declaration, GR8NET-BASIC)
Format: CALL FLLIST
Function: Lists the contents of the serial flash memory.

FLUPDATE (command, GR8NET-BASIC)
Format: CALL FLUPDATE (<sector> [,F])
Function: Updates the contents of the serial flash memory. <sector> specifies the sector number where the update will begin. If parameter ", F" is included, the update will start immediately without asking for confirmation.

FNAME (declaration, RookieDrive BASIC)
Format: ?
Function: ?

FONT (remote, Hangul-BASIC 4)
Format: CALL FONT
Function: Enables alternation between Korean characters (HANGUL key) and non-Korean characters available through the KANA, CYRILLIC or CODE keys. It will return error if used in Screen 9.

FORMAT (command, Disk-BASIC, FormatM. BASIC, RookieD. BASIC)
Format: CALL FORMAT [Disk-BASIC]
Function: Formats a floppy disk. It offers two options:
1 – 1 side, double track (single face, 360K)
2 – 2 sides, double track (double side, 720K)
Format: CALL FORMAT [FormatMaster-BASIC]
Function: Formats a floppy disk offering additional options. Disk-BASIC version 1 required (this instruction is not compatible with Disk-BASIC version 2).
1) 40 trails – 8 sectors per trail – FA
2) 80 tracks – 8 sectors per track – FB
3) 40 tracks – 9 sectors per track – F8
4) 80 tracks – 9 sectors per track – F9

Format: CALL FORMAT [RookieDrive BASIC]
Function: Formats the disc inserted in a standard USB floppy drive or disc image connected to a Rookie Drive interface. Offers 4 options:
1) 720K, full format
2) 720 K, fast format
3) 1.44M, full format
4) 1.44M, fast format

FRAME (statement, Pioneer-BASIC)
Format: CALL FRAME (<frame number>, GOSUB <line number>)
CALL FRAME OFF
Function: Specifies the subroutine line number that will be executed when the frame <frame number> is reached. <frame number> must be in the range between 50 and 54,000. If "OFF" is specified, cancel the line number assignment. This command is specific for use with the Pioneer Laser Vision Player LD-700 and cannot be used in conjunction with the CHAPTER command.

FSIZE (function, DM-System2 BASIC)
Format: CALL FSIZE ("[<device>:] [\ <path>] [[\]<filename>]", <variable>)
Function: Returns the file size
<device> can be drive A: to H: or COM: for computers connected with RS232C.
<path> specifies the location of the folder or file
<filename> accepts wildcards (* and?)
<variable> receives the file size.

GET (function, New Modem BASIC)
Format: CALL GET (<variable>)
Function: Retrieves the ASCII code of a character pressed on the keyboard or received on the telephone line (if that line has not been deactivated with CALL LINEOFF). <variable> stores the ASCII code. Special shortcuts in a BBS program: Pause: CTRL+S or S; continue (after a pause): any key; stop: CTRL+C or C.

HANOFF (command, Hangul-BASIC 1)
Format: CALL HANOFF
Function: Disables the feature to group characters in blocks (characteristic of Hangul characters, used in Korea, available after pressing the HANGUL key). Returns error if used in Screen 9.

HANON (command, Hangul-BASIC 1)
Format: CALL HANON
Function: Enables the feature to group characters in blocks (characteristic of Hangul characters, used in Korea, available after pressing the HANGUL key). Returns error if used in Screen 9.

HCOPY (command, Hitachi-BASIC version 2-3)
Format: CALL HCOPY
Function: Sends to the printer a copy of the text screen (Screens 0 or 1), on Hitachi MB-H2 and MB-H3 computers.

HELP (declaration, DM-System2 BASIC, Hangul-BASIC 4, MSX Aid BASIC, Netw. BASIC, RMSX BASIC, RookieD. BASIC)
Format: CALL HELP
Function: Shows help on BASIC in use.

HIRO (remote, MSX turbo R model FS-A1ST)
Format: CALL HIRO
Function: Calls the menu for programs in ROM on the MSX turbo R model FS-A1ST. For FS-A1GT, use CALL MWR.

HMMM (declaration, DM-System2 BASIC)
Format: CALL HMMM (<X0>, <Y0>) – [STEP] (<X1>, <Y1>) TO (<X2>, <Y2>)
Function: Executes the VDP HMMM (quick copy in bytes) command. Available for Screens 5 to 12.
<X0> – X coordinate of the first point in the source area.
<Y0> – Y coordinate of the first point in the source area.
<X1> – X coordinate of the second point in the source area.
<Y1> – Y coordinate of the second point in the source area.
<X2> – Left X coordinate of the target area.
<Y2> – Upper Y coordinate of the target area.
STEP, if specified, indicates relative coordinates.
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.

HMMV (declaration, DM-System2 BASIC)
Format: CALL HMMV (<X0>, <Y0>) – [STEP] (<X1>, <Y1>) <v>
Function: Executes the VDP's HMMV command (Quick Paint VRAM). Available for Screens 5 to 12.
<X0> – X coordinate of the first point in the area.
<Y0> – Y coordinate of the first point in the area.
<X1> – X coordinate of the second point in the area.
<Y1> – Y coordinate of the second point in the area.
<v> - byte to be sent to VRAM. Specifies one point for Screens 8 to 12, two points for Screens 5 and 7, and four points for Screen 6.
STEP, if specified, indicates relative coordinates.
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.

HZ (control, RMSX-BASIC)
Format: CALL HZ [50 | 60]
Function: Select the screen refresh rate (VDP frequency). No parameter toggles rates.
50 - The VDP will have a frequency of 50 Hz (European, Russian or Arabic MSX).
60 - The VDP will have a frequency of 60 Hz (Japanese, Korean or Brazilian MSX).

IDTRACE (command, Hitachi-BASIC version 2)
Format: CALL IDTRACE
Function: Puts the built-in data reader of the Hitachi MB-H2 in ID tracking mode to verify if the correct tape has been inserted in the reader.

IMPOSE (remote, Pioneer-BASIC)
Format: CALL IMPOSE (<mode>)
Function: Selects the video mode.
<mode> can be:
0 – Computer screen (internal synchronization)
1 – Superimpose (composite video)
2 – External video

INIMDM (command, New Modem BASIC)
Format: CALL INIMDM (<variable>)
Function: Initializes the modem speed according the <variable>, whose value is described in the table below:

Value	Standard	Recep. Speed	Transm. Speed	Note
0	V21	300 baud	300 baud	Caller
1	V21	300 baud	300 baud	Receiver
2	V23	1200 baud	75 baud	-
3	V23	75 baud	1200 baud	-
4	V23	1200 baud	75 baud	for bad connection
5	V23	75 baud	1200 baud	for bad connection
6	V23	600 baud	75 baud	-
7	V23	75 baud	600 baud	-

INIT (command, SFG-BASIC)
Format: CALL INIT
Function: Initializes the FM Music Macro.

INITMD (command, New Modem BASIC)
Format: CALL INITMD
Function: Initializes the X8N1 communication protocol.
X = Xon / Xoff protocol enabled
8 = 8 bits of data
N = without parity
1 = One stop bit

INMK (function, MSX-Audio)
Format: CALL INMK [([<variable 1>] [, [<variable 2>]] [,<variable 3>]])]
Function: Reports changes when using the musical keyboard.
<variable 1> – key number (0 to 127)
<variable 2> – key status (0 if pressed, otherwise 1)
<variable 3> – Frequency of ADPCM corresponding to the key pressed.

INMKEY (function, SFG-BASIC)
Format: CALL INMKEY (<variable>)
Function: Checks if any key on the musical keyboard is being pressed. <variable> returns the key code. If it is 0, no key is being pressed. Short version: _INMK.

INSERTDISK (command, RookieDrive BASIC)
Format: CALL INSERTDISK ("<disk name>")
Function: Insert a new disk image into the USB virtual drive. Currently, it is limited to disk images with a maximum of 720 Kbytes.

INST (statement, SFG-BASIC)
Format: CALL INST (<instrument number> [,<number of voices>] [,<MIDI>] [,<MIDI channel>])
Function: Defines the instruments to be used by the FM Music Macro Up to 4 instruments can be defined by this command.
<instrument number> can vary from 1 to 4.
<number of voices> specifies the number of simultaneous voices used by the instrument. It can vary from 1 to 8.
<MIDI> specifies whether the data will be sent to the MIDI interface. "MIDI ON" uses the MIDI interface and "MIDI OFF" does not (default).
<MIDI channel> can range from 1 to 16. If omitted, channel 1 will be used.

INTWAIT (command, DM-System2 BASIC)
Format: CALL INTWAIT
Function: Pauses the system until the next interruption of DM System2. It can be aborted by CTRL+STOP.

IOSOUND (command, RMSX-BASIC)
Format: CALL IOSOUND [ON] | [OFF]
Function: Enables or disables sounds from emulated cassettes and disks.
ON – Activates all I/O sounds.
OFF – Disables all I/O sounds.

JIS (statement, Kanji-BASIC)
Format: CALL JIS (<string variable>, <character string>)
Function: Converts the first character in a string to a 4-digit hexadecimal JIS code.
<string variable> receives the hexadecimal code
<character string> contains the characters to be converted.

KACNV (statement, Kanji-BASIC)
Format: CALL KACNV (<string variable>, <character string>)
Function: Converts two-byte Kanji characters to one-byte characters.
<string variable> receives the converted characters
<character string> contains the Kanji characters to be converted.

KANJI (command, Kanji-BASIC)
Format: CALL KANJI [<n>]
Function: Activates Kanji mode. <n> can range from 0 to 3, but modes 1 to 3 only work on an MSX2 or higher. When in Kanji mode, press CTRL + SPACE or GRAPH + SELECT to activate Kanji input mode.
0 – 13 lines of 32 or 64 characters (16x16 or 8x16)
1 – 13 lines of 40 or 80 characters (12x16 or 6x16)
2 – 24 lines of 32 or 64 characters in interlaced mode (16x16 or 8x16)
3 – 24 lines of 40 or 80 characters in interlaced mode (16x16 or 8x16)
Note: In Kanji mode, the commands CLS, COLOR= **A**nd SCREEN 9 are disabled.

KBOLD (declaration, DM-System2 BASIC)
Format: CALL KBOLD ([<width>] [,<height>] [,<X edge>] [,<Y edge>] [,<X shadow>] [,<Y shadow>])
Function: Defines the style of the text characters. (Requires FNT driver).
<width> of the character (1 to 16, default is 1)
<height> of the character (1 to 16, default is 1)
<edge X> – X edge thickness (1 to 8, default is 1)
<edge Y> – Y edge thickness (1 to 8, default is 1)
<shadow X> – Horizontal thickness of the shadow character (1 to 32, default is 1)
<shadow Y> – Vertical thickness of the shadow character (1 to 32, default is 1)
If the thickness of the shadow is 0, it will be determined automatically.

KCHR (function, Hangul-BASIC 3)
Format: CALL KCHR (<string variable>, <hexadecimal code>)
Function: Returns in <string variable> the Korean character specified by the 4-digit <hexadecimal code>.

KCODE (function, Hangul-BASIC 3)
Format: CALL KCHR (<string variable>, <string>)
Function: Returns in <string variable> the 4-digit hexadecimal code of the first Korean character in <string>.

KCOLOR (declaration, DM-System2 BASIC)
Format: CALL KCOLOR ([<character color>] [,<background color>] ,<border color>] [,<shadow color>])
Function: Defines the text characters colors. (Requires FNT driver).
<character color> can range from 0 to 15 (default: 15)
<background color> can range from 0 to 15 (default: 0)
<border color> works only for the "border" function. It can vary from 0 to 15 and the default is 1.
<shadow color> works only for the "shadow" function. It can vary from 0 to 15 and the default is 14.

KEXT (function, Kanji-BASIC, Hangul-BASIC 3)
Format: CALL KEXT (<string variable>, <char string>, <function>)
Function: Extracts only 2 bytes or 1 byte characters from a string.
<string variable> receives the extracted characters
<character string> contains the characters to be extracted.
<function> – If 0, only one byte character will be extracted for Kanji-BASIC or non-Korean characters for Hangul-BASIC. If it is 1, only 2 byte characters will be extracted for Kanji-BASIC or Korean characters for Hangul-BASIC.

KEY ON / OFF (function, MSX-Audio)
Format: CALL KEY ON (<key number>, <speed>)
CALL KEY OFF (<key number>)
Function: Informs if the key is pressed or released regardless of its real condition.
<key number> can range from 0 to 127.
<speed> can range from 0 to 15 (8 is the default).

KINIT (declaration, DM-System2 BASIC)

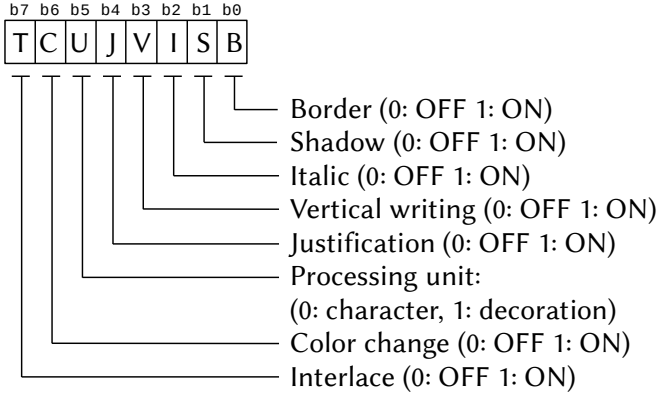
Format: CALL KINIT

Function: Returns all text definitions, including KBOLD and KSIZE, to their default values. (Requires FNT driver).

Format: CALL KINIT ([<configuration>] [,<shadow X>] [,<shadow Y>] [,<italic>] [,<color table>])

Function: Defines all the decoration options for the character.

<configuration> is a one-byte value, with the following flags (the initial value for all is 0):



<shadow X> and <shadow Y> define the position of the shadow in relation to the upper left corner of the character, which can vary between −128 and 127.

<italic> defines the offset to the right of each line of the character, including the border, which can vary between −128 and 127. If omitted, system values will be used.

<color table> defines the colors for each character line from the top, including the border. If omitted, the system color table will be used. The default value is C000H.

KINSTR (function, Kanji-BASIC, Hangul-BASIC 3)

Format: CALL KINSTR (<numeric variable> [<search start>], <string 1>, <string 2>)

Function: Searches for the occurrence of <string 2> in <string 1> and returns the position in <numeric variable>. If there are no occurrences, it returns 0. <search start> is an optional value and indicates the position of the start char for the search.

KLEN (function, Kanji-BASIC, Hangul-BASIC 3)
Format: CALL KLEN (<numeric variable>, <character string>, [<function>]) [Kanji-BASIC]
Format: CALL KLEN (<numeric variable>, <character string>) [Hangul-BASIC]
Function: Returns in <numeric variable> the size of the <character string>. If <function> is 0 or omitted, returns the total number of characters; if it is 1, it returns the number of 1-byte characters and if it is 2 it returns the number of 2-byte characters. Hangul-BASIC does not allow the <function> parameter.

KMID (function, Kanji-BASIC, Hangul BASIC 3)
Format: CALL KMID (<string variable>, <character string>, <shif [,<size>])
Function: Extract <size> characters from position <shift> of the <character string> and put it in <variable string>.

KNJ (statement, Kanji-BASIC)
Format: CALL KNJ (<string variable>, <character string>)
Function: Assigns the <string variable> a kanji character equivalent to the 4-digit hexadecimal kanji code specified in <character string>. When the kanji code is less than 8000H, it will be considered as JIS; when it is larger it will be considered as a JIS shift.

KPRINT (declaration, DM-System2 BASIC)
Format: CALL KPRINT (<character string>, [<limit character>] ,<logic operation code>]
Function: Prints a kanji string on the screen.

KPUT (declaration, DM-System2 BASIC)
Format: CALL KPUT (<string> [,<number of characters>])
Function: Displays a string at high speed. (Requires FNT driver).
<string> is the string to be displayed.
<number of characters> is the maximum number of characters to be displayed. If omitted, all characters will be displayed.

KSIZE (declaration, DM-System2 BASIC)
Format: CALL KPUT (<width>, <height> [,<separation>])
Function: Defines the character size of a byte. (Requires FNT driver).
<width> can vary from 1 to 32 (default: 8)
<height> can vary from 1 to 64 (default: 16)
<partition> defines the space between characters and can vary from 0 to 15 (default: 0)

KTYPE (function, Kanji-BASIC, Hangul-BASIC 3)
Format: CALL KTYPE (<numeric variable>, <character string>, <character position>)
Function: Returns in the <numeric variable> the value 0 if the character corresponding to the <character position> in the <character string> is one byte and the value 1 if the character is 2 bytes.

LB (command, New Modem BASIC)
Format: CALL LB (<string variable>) [;]
Function: Sends a text to the screen and/or the phone line according to the following table:

Eco	Line	Screen	Phone
CALL ECHOON	CALL LINEON	Yes	Yes
CALL ECHOON	CALL LINEOFF	Yes	No
CALL ECHOOFF	CALL LINEON	No	Yes

The text can be paused with CTRL+S but cannot be interrupted.

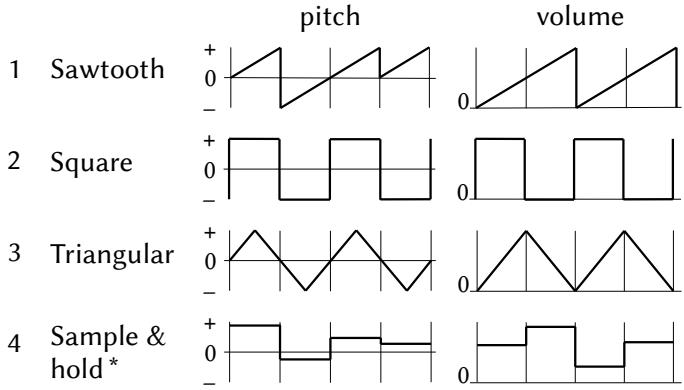
LCOPY (remote, Printer-BASIC or Pioneer-BASIC)
Format: CALL LCOPY [Printer-BASIC]
Function: Prints the data that is still in the temporary buffer of 32 Kbytes when the print spooler is used.
Format: CALL LCOPY (<mode>) [Pioneer-BASIC]
Send a copy of the Screen 2 to the printer. If <mode> is 0, make a positive copy and if it is 1, make a negative copy.

LD (remote, Pioneer-BASIC)
Format: CALL LD
Function: Executes the interactive software present on a CPE (Computer Program Encoded) disk.

LEN (function, New Modem BASIC)
Format: CALL LEN (<string variable>),<numeric variable>
Function: Returns the length of a string, without the final control chars (space, tab, return, etc.). The <numeric variable> will return the length in printable characters of the <string var>.

LENGTH (function, SFG-BASIC)
Format: CALL LENGTH ([<track 1>] [,<track 2>]... [,<track 8>])
Function: Returns the size of the data on a music track. The units returned correspond to 1/192 of an entire note. <track 1> to <track 8> are numeric variables.

LFO (declaration, SFG-BASIC)
Format: CALL LFO (<waveform number> [,<speed>] [,<tremolo>] [,<vibrate>])
Function: Defines the LFO (Low Frequency Oscilator) data. <waveform number> can vary from 1 to 4:



* "Sample &Hold" values are random.

<speed> specifies the frequency of the LFO in relation to the volume. It can vary from 1 to 100. The higher, the higher the frequency and the speed.

<tremolo> specifies the modulation in relation to the volume. It can vary from 1 to 100. The higher, the more the volume will be changed.

<vibrate> specifies the frequency modulation (pitch). It can vary from 1 to 100. The higher, the more the pitch will be changed.

LICENSE (declaration, RMSX-BASIC)
Format: CALL LICENSE
Function: Displays license information about the used version of the rMSX emulator, developed by the Finnish NYYRIKKI for Turbo R. computers.

LINEOFF (command, New Modem BASIC)
Format: CALL LINEOFF
Function: Hang up the phone line but keep the connection active.

LINEON (remote, New Modem BASIC)
Format: CALL LINEON
Function: Connects the telephone line.

LMMM (declaration, DM-System2 BASIC)
Format: CALL LMMM (<X0>, <Y0>) – [STEP] (<X1>, <Y1>) TO (<X2>, <Y2>) [,<logical operator>]
Function: Executes the VDP command LMMM (logical copy in points). Available for Screens 5 to 12.
<X0> – X coordinate of the first point in the source area.
<Y0> – Y coordinate of the first point in the source area.
STEP, if specified, indicates relative coordinates.
<X1> – X coordinate of the second point in the source area.
<Y1> – Y coordinate of the second point in the source area.
<X2> – Left X coordinate of the target area.
<Y2> – Upper Y coordinate of the target area.
<logical operator> can be [T] PSET, [T] PRESET, [T] XOR, [T] OR or [T] AND. The default is PSET.
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.

LMMV (declaration, DM-System2 BASIC)
Format: CALL LMMV (<X0>, <Y0>) – [STEP] (<X1>, <Y1>),<color> [,<logical operator>]
Function: Executes the LMMV command (logical copy in points from VDP to VRAM). Available for Screens 5 to 12.
<X0> – X coord of the first point in the destination area.
<Y0> – Y coord of the first point in the destination area.
STEP, if specified, indicates relative coordinates.

<X1> – X coord of the second point in the destination area.
<Y1> – Y coord of the second point in the destination area.
<color> specifies the color of the rectangle to be painted.
<logical operator> can be [T] PSET, [T] PRESET, [T] XOR, [T] OR or [T] AND. The default is PSET.
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.

LO (command, New Modem BASIC)
Format: CALL LO (<string variable>) [;],<variable>
Function: Sends a text to the screen and/or the phone line according to the following table:

Eco	Line	Screen	Phone
CALL ECHOON	CALL LINEON	Yes	Yes
CALL ECHOON	CALL LINEOFF	Yes	No
CALL ECHOOFF	CALL LINEON	No	Yes

Any characters received are ignored, except for the pause, abort and continue keys. The key buffers remain empty; no keys are stored. The final control codes are sent with the text.
<variable> parameter returns the result of the execution:
0 = the text was sent correctly
3 = the operation was aborted with CTRL+C or C

LOAD (command, DM-System2 BASIC, QuickDisk BASIC)
Format: CALL LOAD ("[<device>:] [\ <path>] [\]<filename>", [@] <destination address> [,<size>] [,<offset>]) [DM-System2 BASIC]
Function: Reads a file or part of it.
<device> can be drive A: to H: or COM: for computers connected with RS232C.
<path> specifies the location of the folder or file.
<filename> is the file to be read.
<destination address> is the destination address for the data. If preceded by "@" it means VRAM.
<size> specifies the number of bytes to read.
<offset> specifies the offset in the source file.

Format: CALL LOAD ([QD[n]:]"<filename>"[,R])
[QuickDisk-BASIC]
Function: Load a non-binary file from the specified Quick Disk device. It can be a BASIC file in tokenized mode or in ASCII text.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8 chars + "." + 3 chars.
[,R], if specified, runs the BASIC file right after loading.

LOAD PCM (command, MSX-Audio)
Format: CALL LOAD PCM (<"filename">, <file number>)
Function: Load ADPCM and PCM data from disk.
<filename> – filename on disk.
<file number> – File number in the audio memory. It can range from 0 to 15.

LOADROM (command, RookieDrive BASIC)
Format: CALL LOADROM ("<filename>")
Function: Loads an 8kb, 16kb or 32kb ROM file into RAM and starts its execution by restarting the computer. The ROM file must be located in the root directory of the USB device.
<filename> must be in 8.3 format.

LOCKDRV (command, Nextor)
Format: CALL LOCKDRV (<drive letter>: N)
Function: Lock or unlock drive letters, or display the list of blocked drives. If "N" is 0, it unlocks the drive; any other number locks.

LOGFILE (command, New Modem BASIC)
Format: CALL LOGFILE (<filename>) [;],<variable>
Function: Stores everything on the screen in a text file. <filename> must be a string variable.

LOOK (function, SFG-BASIC)
Format: CALL LOOK ([<instrument 1>] [,<instrument 2>]
[,<instrument 3>] [,<instrument 4>])
Function: Returns the status of the instrument, whether it is being played or not. The <instrument 'n'> parameters are numeric variables. For instrument not defined by _INST, it returns 0.

LS (remote, New Modem BASIC)

Format: CALL LS (<string variable>) [;],<variable>

Function: Sends a text to the screen and / or the phone line according to the following table:

Eco	Line	Screen	Phone
CALL ECHOON	CALL LINEON	Yes	Yes
CALL ECHOON	CALL LINEOFF	Yes	No
CALL ECHOOFF	CALL LINEON	No	Yes

Any characters received are ignored, except for the pause, abort and continue keys. The key buffers remain empty; no keys are stored. The final control codes are sent with the text. The <variable> parameter returns the result of the execution:

0 = the text was sent correctly.

3 = the operation was aborted with CTRL+C or C.

MALLOC (command, DM-System2 BASIC)

Format: CALL MALLOC ([<number of pages>] [,<variable>])

Function: Enables access to the Memory Mapper.

<number of pages> is the number of pages to be allocated. If it is 0, the allocated pages will be released. If omitted, the current number of allocated pages will return in <variable>.

<variable> is a numeric variable that will contain the number of pages actually allocated.

MAPDRV (command, Nextor)

Format: CALL MAPDRV (<drive> [,<partition> [,<device> [,<slot> | 0]])

Function: Maps a drive unit in the Nextor system.

<drive> letter or drive number to be mapped

<partition> is a number as following:

- 0 – The drive will be mapped from the device's absolute zero sector.
- 1 – First primary partition
- 2 to 4 – Refer to extended partitions 2.1 to 2.4, if partition 2 is extended; otherwise, they refer to primary partitions.
- 5 – Onwards refer to extended partitions.

<device> – Device index (1 to 7)
<slot> – Slot number (0 to 3). If the slot is expanded, use the formula <main slot> + 4 * <subslot>. If "0" is specified, the primary unit slot will be selected.

MAPDRVL (command, Nextor)
Format: CALL MAPDRVL (<drive> [,<partition> [,<device> [,<slot> | 0]])
Function: Maps a drive unit in the Nextor system and locks the specified drive. The parameters are identical to MAPDRV.

MC (declaration, New Modem BASIC)
Format: CALL MC (<string variable>)
Function: Converts the alphabetic characters of the <string variable> to uppercase.

MDR (command, MSX turbo R model FS-A1GT)
Format: CALL MDR
Function: Activates the MSX-MUSIC output to the MIDI interface. Only MSX turbo R model FS-A1GT.

MEMINI (command, 2)
Format: CALL MEMINI [(RAM disk size)]
Function: Activates the RAM disk in the lower 32K of memory.

MERGE (command, QuickDisk BASIC)
Format: CALL MERGE ("[QD [n]:]<filename>")
Function: Merges a BASIC or DATA program saved in ASCII on the QuickDisk device with the program that is in the MSX memory.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8 chars + "." + 3 chars.

MESSAGE (statement, MSX-Aid BASIC, Network BASIC)
Format: CALL MESSAGE [MSX-Aid BASIC]
Function: Displays an encouraging message for programmers using MSX-Aid.

Format: CALL MESSAGE (<message>, [<student number>])
[Network BASIC]

Function: Sends a message of up to 56 characters to a specific student. This instruction is only available to the teacher. <student number> can range from 1 to 15. The short version: _MESS can be used.

MFADE (declaration, StudioFM BASIC)

Format: CALL MFADE (<degree of fade>)

Function: Produces a fade-out when playing back .MUS files in the Studio FM. <degree of fade> can vary between 0 and 255, with 0 no fade and 255 will produce the longest fade-out.

MFILES (command, 2)

Format: CALL MFILES

Function: Lists the RAM disk files of the lower 32K of memory.

MK PCM (declaration, MSX-Audio)

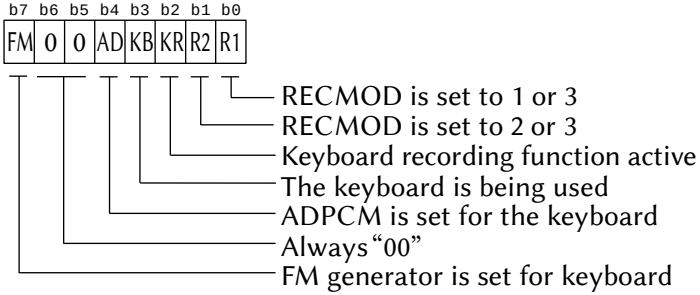
Format: CALL MK PCM (<file number>)
CALL MK PCM OFF

Function: Defines which ADPCM file will be played as an instrument. If specified OFF, it cancels the previously defined instrument. <file number> can range from 0 to 15.

MK STAT (function, MSX-Audio)

Format: CALL MK STAT (<variable>)

Function: Returns the recording or playback status of the musical keyboard.
<variable> is a numerical value defined according to the figure below.



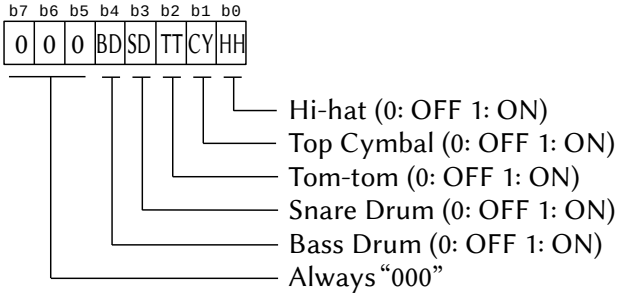
MK TEMPO (statement, MSX-Audio)

Format: CALL MK TEMPO (<speed>, <percussion map>)

Function: Specifies the recording / playback speed of the musical keyboard or activates the metronome function. In this case, the AUDIO command must be previously defined. This command affects the speed of the MK PLAY, MK REC and MK APPEND instructions.

<speed> must be in the range 25~360, the initial value being 120.

<percussion map> is a numerical value defined according to the figure below.



MK VEL (statement, MSX-Audio)

Format: CALL MK VEL (<speed>)

Function: Specifies the speed, or pressure force, that is applied to a key on the musical keyboard.
<speed> can range from 0 to 15, the initial value being 8.

MK VOICE (statement, MSX-Audio)

Format: CALL MK VOICE ([@]<instrument number>)

Function: Defines the instrument to be associated with the musical keyboard. <instrument number> is a numeric variable that defines the instrument number, which can vary from 0 to 63. If there is no @, the variable will be assumed to be a matrix, where the values in sequence define the instrument.

MK VOL (statement, MSX-Audio)

Format: CALL MK VOL (<volume>)

Function: Defines the volume associated with the musical keyboard. <volume> can range from 0 to 63.

MKDIR (command, Disk-BASIC 2nd version)
Format: CALL MKDIR (<subdirectory>)
Function: Creates the <subdirectory> with the specified name.

MKILL (command, 2)
Format: CALL MKILL ("<filename>")
Function: Deletes the file <filename> from the RAM disk of the lower 32K of memory.

MLOAD (command, StudioFM BASIC)
Format: CALL MLOAD ("<filename>",<address>)
Function: Loads a song in the Studio FM format (.MUS).

MNAME (command, 2)
Format: CALL MNAME ("<filename1>" AS "<filename2>")
Function: Renames file <filename1> with <filename2> on the RAM disk of the lower 32K of memory.

MODE9 (remote, Hangul-BASIC 4)
Format: CALL MODE9
Function: Go to Screen 9 automatically using Width 80.

MODINST (declaration, SFG-BASIC)
Format: CALL MODINST (<instrument> [,<voice>]
[,<transposition>] [,<volume>] [,<portamento>]
[,<portamento speed>] [,<support>] [,<trigger mode >]
[,<LFO sync>] [,<tremolo>] [,<vibrate>])
Function: Changes the instrument data. Short version: _MODI.
<instrument> specifies the tone of the instrument defined by _INST.
<voice> specifies the pitch of the instrument.
1~48 – Selects the tone of the FM Sound Synthesizer ROM. 47 and 48 are reserved and do not contain tones.
49~56 – Select tones defined by _SEL.
<transposition> allows the various instruments to be transposed separately. It can range from –12 to +12 in half-step intervals.

<volume> sets the volume separately for each instrument. It can vary from 0 to 100, with 100 being the maximum volume.

<portamento> can take two values:

0 – Portamento only during playback

1 – Portamento all the time

<portamento speed> can vary from 0 to 100, with 100 being the slowest. 0 turns off the portamento.

<support> can take on two values:

0 – Standard lift time

1 – Support time is doubled

<trigger mode> determines whether to trigger when the keys pressed are released.

0 – "attack" when the keys are released

1 – No "attack" during legacy playback

<LFO sync> determines whether there will be synchronization with the LFO when the pressed keys are released.

0 – No synchronization

1 – The LFO starts the waveform whenever a key is released.

<tremolo> specifies the degree of the tremolo. It can vary from 0 to 100, with 100 being the highest degree of sensitivity. (despite the range 0 to 100, there are only 4 degrees of tremolo).

<vibrate> specifies the degree of vibrato. It can vary from 0 to 100, with 100 being the highest degree of sensitivity. (despite the range 0 to 100, there are only 8 degrees of vibrato).

MON (command, FM-X BASIC, Hitachi BASIC, MSXAid BASIC)

Format: CALL MON [FM-X BASIC]

Function: Starts the monitor when the Fujitsu FM-X is connected to the FM-7 machine with the MB22450 interface.

Format: CALL MON [Hitachi BASIC 1-2]

Function: Starts the System Monitor Utility command line on the Hitachi MB-H1 and MB-H2 computers. For a list of all commands available in this utility, type H on the command line.

Format: CALL MON [MSXAid BASIC]
Function: Starts the internal monitor of the MSX-Aid utility. To get help using this monitor, first enter a RAM address and then press F6.

MOUNT (command, RookieDrive BASIC)
Format: ?
Function: ?

MPLAY (statement, StudioFM BASIC)
Format: CALL MPLAY (<memory address>)
Function: Plays a song in the StudioFM (.MUS) format loaded in memory with _MLOAD.

MRING (command, New Modem BASIC)
Format: CALL MRING (<numeric variable>)
Function: Checks whether the phone is ringing. <numeric variable> can return the following values:
0 – The phone is ringing
1 – The phone is not ringing
2 – Routine interrupted (CODE key pressed)

MSTART (command, New Modem BASIC)
Format: CALL MSTART (<numeric variable>)
Function: Repare the modem for data communication. If <numeric variable> returns 0, everything is fine; otherwise there was an error.

MSTOP (command, New Modem BASIC, StudioFM BASIC)
Format: CALL MSTOP [New Modem BASIC]
Function: Interrupts the modem's functions.
Format: CALL MSTOP [StudioFM BASIC]
Function: Stops playback of a song in StudioFM format (.MUS) played in the background.

MUSIC (command, MSX-Music)
Format: CALL MUSIC [(<n1> [, 0 [,<n3> ... [, n9]]]]]]]]])]]

Function: Starts MSX-MUSIC and determines which voices will be used and how.
<n1> can be:
0 – Select pure melody mode (n3~n9 can be specified)
1 – Select melody + battery mode (n3~n6 can be specified)
<n3> to <n9> can be:
1 – Select melody
2 – Select battery

MUTE (control, Hitachi-BASIC, Pioneer-BASIC, RMSX-BASIC)
Format: CALL MUTE [Hitachi-BASIC 2]
Function: Adds a 4 second pause before recording data to the internal register of the Hitachi HB-M2.
Format: CALL MUTE [R | L] [Pioneer-BASIC]
CALL MUTE OFF
Function: Mutes the right (R), left (L) audio channels or both if there is no channel specification. If OFF is specified, the mute function is canceled.
Format: CALL MUTE [ON] | [OFF] [RMSX-BASIC]
Function: Enables or disables the audio output. If the parameter is omitted, it just reverses the state.

MWP (command, MSX turbo R model FS-A1GT)
Format: CALL MWP
Function: Calls the menu for programs in ROM on the MSX turbo R model FS-A1GT. For FS-A1ST, use CALL HIRO.

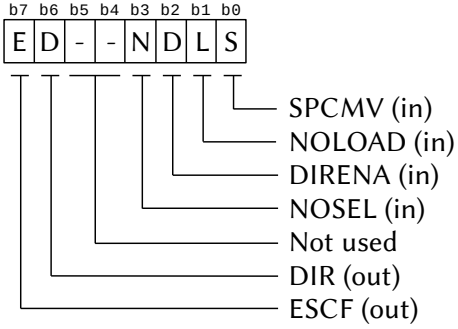
NET (command, GR8NET-BASIC)
Format: CALL NET
Function: Displays GR8NET help.

NETBITOV (command, GR8NET-BASIC)
Format: CALL NETBITOV (<page>, <address>, <VRAM bank>, <VRAM address>)
Function: Transfer the image of the GR8NET buffer icon to VRAM.
<page> is the logical page number of the icon data.
<address> is the address of the icon data and can only vary from 6000H to 7FFFH.

<VRAM bank> must be 0 for 0000H~FFFFH or 1 for 10000H~1FFFFH.
<VRAM address> is the address within the selected VRAM bank.

NETBLOAD (command, GR8NET-BASIC)
Format: CALL NETBLOAD (<url>, <execution flag>, <logical page>, <GR8NET address>)
Function: Load binary file from SD card or remote web server using HTTP.
<url> is the URL string for remote access. For the first partition on the SD card, use "SDC: //".
<execution flag> indicates the action to be taken for executable files.
0 – Data will not be loaded (default value)
1 – Data will be loaded but not executed
2 – The file will be loaded and executed.
<logical page> of the GR8NET (00H to 7FH)
<GR8NET address> can vary from 6000H to 7FFFH.

NETBROWSE (command, GR8NET-BASIC)
Format: CALL NETBROWSE (<url>, <flags>)
Function: Calls the web browser and SD card.
<url> is the starting URL string.
<flags> is a one-byte value with the following meanings:



SPCMV: if this bit is set, when the user presses the SPACE key in the selection, the url will be loaded into the GR8NET RAM, but no action will be performed and the browser exits;

NOLOAD: if this bit is set, the browser will not load the selected url in the GR8NET RAM.

DIRENA: if this bit is set, when pressing the space bar in the directory it will be selected and the browser will be closed; if this bit is reset, pressing space in the directory will load the content and navigation will continue.

NOSEL: if set, it does not force the source device selection page (Internet / SD card), and navigation proceeds directly to the device identified by the URL string.

DIR: this bit returns set if the content is a directory entry or a page with a list of directories generated by the WEB server.

ESCF: this bit returns set when the browser is closed with the ESC key.

NETBTOV (command, GR8NET-BASIC)

Format: CALL NETBTOV (<VRAM bank>, <offset address>)

Function: Moves binary data from the GR8NET buffer to the VRAM.
<VRAM bank> must be 0 for 0000H~FFFFH or 1 for 10000H~1FFFFH.
<address offset> is the offset of the address specified in the header of the binary file.

NETCDTOF (command, GR8NET-BASIC)

Format: CALL NETCDTOF

Function: Copy the DHCP configuration to the fixed IP address configuration.

NETCFG (command, GR8NET-BASIC)

Format: CALL NETCFG

Function: Activates the interactive configuration of the GR8NET.

NETCODE (function, GR8NET-BASIC)

Format: CALL NETCODE (<error_code>, [<http_oper>])

Function: Returns the status of the last operation and the HTTP response code.

NETDHCP (command, GR8NET-BASIC)
Format: CALL NETDHCP
Function: Performs DHCP search and makes its dynamic configuration.

NETDIAG (command, GR8NET-BASIC)
Format: CALL NETDIAG (<V>)
Function: Turns on / off diagnostic mode. If <V> is 0, it turns off; otherwise it turns on.

NETDNS (command, GR8NET-BASIC)
Format: CALL NETDNS [([<A>], [<B>], [<C>], [<D>])]
Function: Gets the IP of the current DNS domain. If there are no arguments, print the address on the screen.

NETDUMP (command, GR8NET-BASIC)
Format: CALL NETDHCP
Function: Performs DHCP search and makes its dynamic configuration.

NETEND (command, Network-BASIC)
Format: CALL NETEND
Function: Disables the MSX network (MSX Network). Short version: _NETE.

NETEXPRT (command, GR8NET-BASIC)
Format: CALL NETEXPRT
Function: Create BASIC program containing GR8NET config data.

NETFIX (command, GR8NET-BASIC)
Format: CALL NETFIX
Function: Configure fixed IP address information for the network.

NETFKOPLLR (command, GR8NET-BASIC)
Format: CALL NETFKOPLLR
Function: Load the OPLL ROM (MSX-Music) into the mapped memory. This command is intended for software that runs in GR8NET mapper modes 1 to 6 (game mapper) and cannot be run in mode 8 when MSX-Music ROM is available in GR8NET subslot 3.

NETFWUPDATE (command, GR8NET-BASIC)
Format: CALL NETFWUPDATE ([<argument>])
Function: Updates the GR8NET firmware. If <argument> is omitted or is 0, it only displays information about the current firmware. If it is 1 (one), it updates only the main firmware and if it is 3 (three) it also updates the configuration area.

NETGETCLK (function, GR8NET-BASIC)
Format: CALL NETGETCLK ([<source>],<frequency>)
Function: Returns the clock frequency. If <source> returns zero, the frequency will be that of the MSX main bus; if it is different from zero, the frequency of the GR8NET internal oscillator will return. <frequency> must be a numeric variable.

NETGETCLOUD (command, GR8NET-BASIC)
Format: CALL NETGETCLOUD
Function: Prints the status of the GR8cloud virtual volume on the screen.

NETGETDA (function, GR8NET-BASIC)
Format: CALL NETGETDA (<adapter number>, <active adapters>)
Function: Returns the number of the standard adapter in <adapter number> and the list of active adapters in <active adapters>.
<adapter number> must be a numeric variable.
<active adapters> must be a numeric variable where bits 0 to 3 will receive the state of the adapters (the respective bit will be set if the device is active.

NETGETDNS (command, GR8NET-BASIC)
Format: CALL NETGETDNS ([<A>], [<B>], [<C>], [<D>])
Function: Gets the DNS address of the fixed IP. [<A>], [<B>], [<C>] and [<D>] must be numeric variables.

NETGETGW (function, GR8NET-BASIC)
Format: CALL NETGETGW ([<A>], [<B>], [<C>], [<D>])
Function: Get fixed IP address of the gateway. [<A>], [<B>], [<C>] and [<D>] must be numeric variables.

NETGETHOST (command, GR8NET-BASIC)
Format: CALL NETGETHOST (<flag>, <name> | <A>, <B>, <C>, <D>)
Function: Gets the name and IP address of the remote host. <A>, <B>, <C> and <D> and <flag> must be numeric variables and <name> must be an alphanumeric variable.

NETGETIP (function, GR8NET-BASIC)
Format: CALL NETGETIP ([<A>], [<B>], [<C>], [<D>])
Function: Gets the fixed IP address. [<A>], [<B>], [<C>] and [<D>] must be numeric variables.

NETGETMAP (function, GR8NET-BASIC)
Format: CALL NETGETMAP (<flags>)
Function: Gets the type of Memory Mapper and other data. <flags> must be a 16-bit numeric variable, where bits 0 to 7 contain the current logical page of the Memory Mapper and bits 8, 13 and 4 are bits of the system mode register.

NETGETMASK (function, GR8NET-BASIC)
Format: CALL NETGETMASK ([<A>], [<B>], [<C>], [<D>])
Function: Obtains the fixed IP address mask. [<A>], [<B>], [<C>] and [<D>] must be numeric variables.

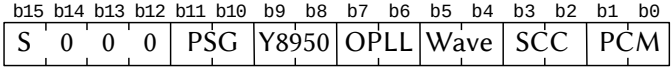
NETGETMD (function, GR8NET-BASIC)
Format: CALL NETGETMD (<logical page>, <address>, variable>)
Function: Get a 4-byte (32-bit) word from memory, convert and store it in a BASIC variable.
<logical page> is the number of the logical page in bank 1 of the GR8NET (6000-7FFF).
<address> is the address visible by the Z80
<variable> is a BASIC variable capable of accommodating the read value (single or double precision).

NETGETMEM (function, GR8NET-BASIC)
Format: CALL NETGETMEM (<logical page>, <address>, [<A>], [<B>], [<C>], [<D>])
Function: Reads a sequence of 4 bytes in memory.
<logic page> logic page number in bank 1 of the GR8NET (6000H~7FFFH).
<address> is the memory address visible to the Z80.
<A> = Address, <B> = Address + 1, etc.

NETGETMIX (function, GR8NET-BASIC)

Format: CALL NETGETMIX ([<number>])

Function: Returns the configuration of the audio mixer. If bit 15 is 0, the audio is mono, if it is 1 it is stereo. If <number> is omitted, the setting will be printed on the screen.
<number> is a 16-bit numeric value:



Where each 2 bits represent the following:
00 – Mute 10 – Right channel
01 – Left channel 11 – Both channels

NETGETMMV (function, GR8NET-BASIC)

Format: CALL NETGETMMV ([<user home page>], [<top of RAM>] [<disk image start page>], [maximum no. of pages], [<init page of Y8950 RAM]

Function: Returns the configuration of the memory manager. All are numeric values and any can be omitted. If all are omitted, the command prints the values on the screen.

NETGETNAME (function, GR8NET-BASIC)

Format: CALL NETGETNAME ([<filename>])

Function: Returns the filename of the remote resource.

NETGETNTP (function, GR8NET-BASIC)

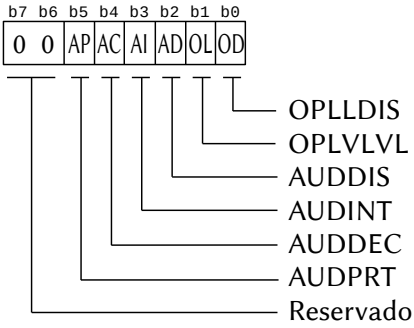
Format: CALL NETGETNTP ([<A>], [<B>], [<C>], [<D>])

Function: Gets the properties on the NTP server in the fixed IP address configuration. [<A>], [<B>], [<C>] and [<D>] must be numeric variables.

NETGETOPL (function, GR8NET-BASIC)

Format: CALL NETGETOPL (<OPL state>, <num sample RAM pages>, <sample RAM size>)

Function: Gets the status of the OPLL / Y8950 and the size of the Sample RAM.
<OPL state> is a byte of flags with the following format:



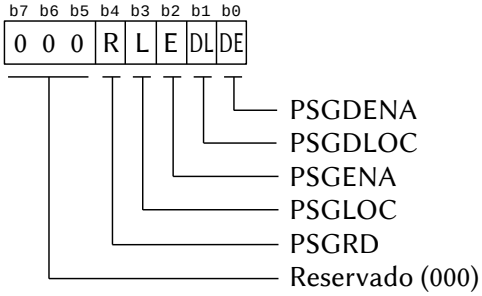
OPLLDIS – 0 – OPLL output is active (default)
1 – OPLL output turned off
OPLVLVL – 0 – Normal volume OPLL / Y8950 (default)
1 – Duplicated volume OPLL / Y8950
AUDDIS – 0 – MSX-Audio is enabled (default)
1 – MSX-Audio disabled
AUDINT – 0 – Interrupt. MSX-Audio enabled (default)
1 – MSX-Audio interrupts disabled
AUDDEC – 0 – MSX-Audio unconfigured / unavailable
1 – MSX-Audio configured in AUDPRT
AUDPRT – 0 – MSX-Audio configured port C0-C1
1 – MSX-Audio configured port C2-C3
<num sample RAM pages> allocated (8K each)
<sample RAM size> required for 8K pages

NETGETPATH (function, GR8NET-BASIC)
Format: CALL NETGETPATH ([<path>])
Function: Returns the <path> of the remote resource. <path> must be an alphanumeric variable. If omitted, print the path on the screen.

NETGETPORT (function, GR8NET-BASIC)
Format: CALL NETGETPATH ([<remote port>], [<local port>])
Function: Returns the <remote port> and <local port> (must be numeric variables).

NETGETPSG (function, GR8NET-BASIC)
Format: CALL NETGETPSG ([<flags>])

Function: Returns some data from the PSG. <flags> is a data byte with the following format:



Where:

- PSGDENA and PSGDLOC are the desired initial state of PSG (see _NETSETPSG);
- PSGENA and PSGLOC are the real state (1 = Activated) and the location (0 = 0xA0, 1 = 0x10) of the PSG;
- If PSGRD is 1, the PSG registers can be read and if it is 0 the PSG is in write-only mode.

NETGETQSTR (function, GR8NET-BASIC)

Format: CALL NETGETQSTR ([<query string>])

Function: Returns the <query string> defined for the remote resource. If the argument is omitted, print the result on the screen.

NETGETTSHN (function, GR8NET-BASIC)

Format: CALL NETGETTSHN ([<time server>])

Function: Returns the host name of the <time server>. If the argument is omitted, print the result on the screen.

NETGW (function, GR8NET-BASIC)

Format: CALL NETGW ([<A>], [<B>], [<C>], [<D>])

Function: Returns the configuration of the current gateway. <A>, <B>, <C> and <D> must be numeric variables.

NETHELP (command, GR8NET-BASIC)

Format: CALL NETHELP <command>

Function: Displays help for the specified GR8NET <command>. If <command> is omitted, print a list of all available commands.

NETIMPRT (command, GR8NET-BASIC)
Format: CALL NETIMPRT
Function: Fill the GR8NET system variables with data from the BASIC program created by NETEXPRT.

NETINIT (command, Network-BASIC)
Format: CALL NETINIT
Function: Initializes the MSX network. Use only after CALL NETEND, as the network is automatically started when the computer is turned on. Short version: _NETI

NETIP (function, GR8NET-BASIC)
Format: CALL NETIP ([<A>], [<B>], [<C>], [<D>])
Function: Gets the IP address of the current adapter. <A>, <B>, <C> and <D> must be numeric variables.

NETLDBUF (command, GR8NET-BASIC)
Format: CALL NETLDBUF (<adapter page>, <adapter address>, <block size>, <RAM address>, [<mapper type>])
Function: Copy data from main memory to the adapter's buffer.

NETLDRAM (command, GR8NET-BASIC)
Format: CALL NETLDRAM (<adapter page>, <adapter address>, <block size>, <RAM address>)
Function: Downloads data from the adapter's buffer to main memory. <adapter address> must be between &H6000 and &H7FFF.

NETMASK (function, GR8NET-BASIC)
Format: CALL NETMASK ([<A>], [<B>], [<C>], [<D>])
Function: Gets the subnet mask of the current adapter. <A>, <B>, <C> and <D> must be numeric variables.

NETNTP (function, GR8NET-BASIC)
Format: CALL NETNTP (<A>, <B>, <C>, <D>, <TZF>)
Function: Obtains the effective configuration of the NTP server. <A>, <B>, <C>, <D> and <TZF> must be variables or constants numeric. If bit 7 of TZF is set, the RTC will be synchronized with the NTP server. Bits 0 to 6 represent a positive or negative value (–64 [40H] to +63 [3FH]) and define the time zone in 15-minute increments.

NETPLAYBUF[#]A (command, GR8NET-BASIC)
Format: CALL NETPLAYBUF[#]A (<logical page>, <address>, <size>)
Function: Defines the address and initial size of buffer No. [#] for the PCM. <logical page> must be between 00H~7FH, <address> between 6000H~7FFFH and <size> must be calculated so as not to exceed the GR8NET memory of 1 MB. [#] must be between 0 and 9.

NETPLAYBUF[#]C (command, GR8NET-BASIC)
Format: CALL NETPLAYBUF[#]C
Function: Continue playback by refilling PCM buffer [#], which must be between 0 and 9.

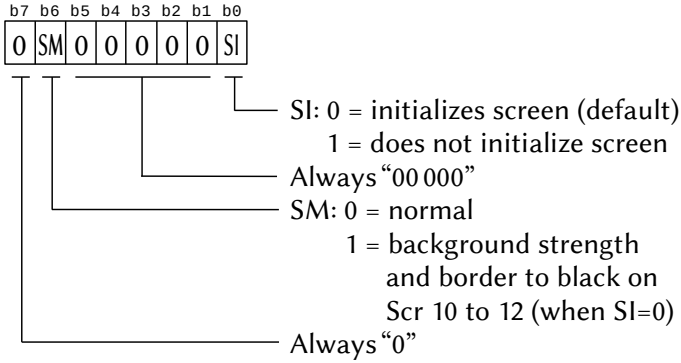
NETPLAYBUF[#]P (command, GR8NET-BASIC)
Format: CALL NETPLAYBUF[#]P (<size>, <frequency>)
Function: Start the reproduction of the data pre-stored in the PCM buffer nº [#]. <size> can be 8 or 16 bits and <frequency> can range from 1 to 65,536. [#] must be between 0 and 9. To prevent the buffer from emptying, the command _NETPLAYBUF[#]C must be used.

NETPLAYBUF[#]R (command, GR8NET-BASIC)
Format: CALL NETPLAYBUF[#]R
Function: Reset the buffer [#] reproduction mechanism, which must be between 0 and 9.

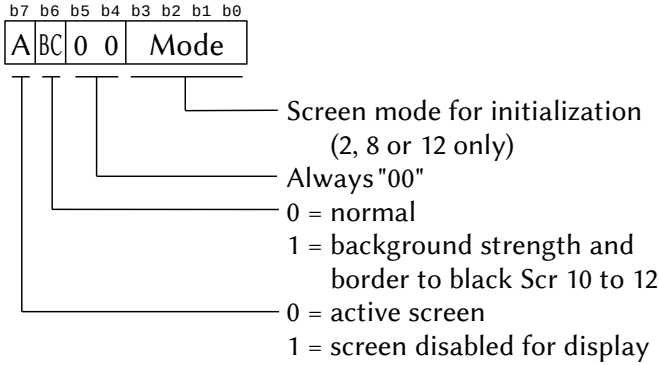
NETPLAYBUF[#]S (command, GR8NET-BASIC)
Format: CALL NETPLAYBUF[#]S (<state>)
Function: Return the playback status of buffer [#]. <state> must be a numeric variable. If <state> returns –1, playback has ended and if it returns 0, data is still being played. [#] must be between 0 and 9.

NETPLAYVID (command, GR8NET-BASIC)
Format: CALL NETPLAYVID (<path> [,<flags>])
CALL NETPLAYVID (<screen mode>)
Function: Play video from the SD card. It works in two ways, depending on the first argument. If string, specify the

<path> of the video file on the SD card. <flags> is an 8-bit value whose meanings are described below:



If the first argument is an integer, its lowest 8 bits are flags with the following meanings:



NETPLAYWAV (command, GR8NET-BASIC)

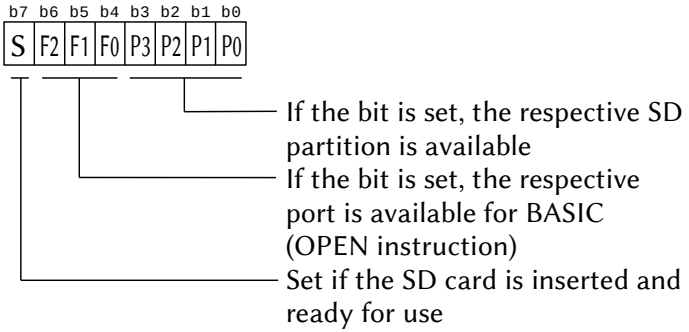
Format: CALL NETPLAYWAV (<path>)

Function: Play audio in wave format. <path> is a name or string variable that identifies the location of the URI of the remote wave file.

NETRESST (command, GR8NET-BASIC)

Format: CALL NETRESST (<flags>)
CALL NETRESST (<inbound URI>, <outbound URI>, <flags>, <size>)

Function: Return the resource's state. If the first argument is an integer variable, the lowest 8 bits will contain the flags as described below. If it is alphanumeric, <input URI> and <output URI> will contain the respective paths and <size> is a numeric variable that returns the size of the resource. <flags> is an integer variable whose lowest 8 bits are mapped as follows:



NETSAVE (command, GR8NET-BASIC)

Format: CALL NETSAVE

Function: Save the current configuration of the ROM configuration page.

NETSDCRD (command, GR8NET-BASIC)

Format: CALL NETSDCRD (<logical page>, <address>, <sector>, <number of sectors to read>)

Function: Read sectors from the SD card. <logical page> is the page number in bank 1 of the GR8NET (6000H~7FFFH), <address> is the visible address for the Z80 and <sector> is the number of the first sector to be read.

NETSETCLK (command, GR8NET-BASIC)

Format: CALL NETSETCLK (<source>)

Function: Defines the frequency source for speed measurement. If <source> is 0, the frequency of the MSX internal bus will be used; if different from 0, the frequency of the GR8NET internal oscillator (3.579545 MHz) will be used.

NETSETCLOUD (command, GR8NET-BASIC)
Format: CALL NETSETCLOUD (<hostname: port>, <password>)
CALL NETSETCLOUD (<activation flag>)
Function: Configure access to the GR8NET virtual volume.
<hostname: port> can be up to 70 characters long, with the port number separated by a colon. The access <password> can be up to 16 characters. To enable the GR8cloud subsystem, <activation flag> must contain the numeric value 1, but the volume will only be fully accessible after the restart.

NETSETDA (command, GR8NET-BASIC)
Format: CALL NETSETDA (<adapter number>)
Function: Defines the number of the standard adapter.
<adapter number> must be a value from 0 to 3.

NETSETDM (command, GR8NET-BASIC)
Format: CALL NETSETDM (<logical page>, <address>, <variable>)
Function: Gets the value of a BASIC variable, converts it to a 32-bit value and stores it in memory.
<logic page> logic page number in bank 1 of the GR8NET (6000H~7FFFH).
<address> is the memory address visible to the Z80.
<variable> can be a number, an expression or a numeric variable of any type.

NETSETDNS (command, GR8NET-BASIC)
Format: CALL NETSETDNS ([<A>], [<B>], [<C>], [<D>])
Function: Defines fixed IP address. At least one of the <A>, <B>, <C> or <D> values must be defined.

NETSETGW (command, GR8NET-BASIC)
Format: CALL NETSETGW ([<A>], [<B>], [<C>], [<D>])
Function: Defines the fixed IP address of the gateway. At least one of the <A>, <B>, <C> or <D> values must be defined.

NETSETHOST (command, GR8NET-BASIC)
Format: CALL NETSETHOST (<URI>)
CALL NETSETHOST (<A>, <B>, <C>, <D>)

Function: Defines the name of the remote host and, if necessary, performs a simple DNS query. The <URI> must be written without a protocol definition and without the final slash (eg "www.gr8bit.ru")

NETSETIP (command, GR8NET-BASIC)
Format: CALL NETSETIP ([<A>], [<B>], [<C>], [<D>])
Function: Defines fixed IP address. At least one of the <A>, <B>, <C> or <D> values must be defined.

NETSETMAP (command, GR8NET-BASIC)
Format: CALL NETSETMAP [(<A>, <M>, <MRPD>)]
Function: Defines the type of Memory Mapper and restarts the system.
<A> identifies the type of memory mapped and the location of the special register set.
<M> 0 – Reading disabled.
1 – Reading enabled.
2 – Automatic detection (default)
<MRPD> RAM mapped with pending disable bit (0 – Enable; 1 – Disable).

NETSETMASK (command, GR8NET-BASIC)
Format: CALL NETSETMASK ([<A>], [<B>], [<C>], [<D>])
Function: Sets the mask for the fixed IP address. At least one of the <A>, <B>, <C> or <D> values must be defined.

NETSETMEM (command, GR8NET-BASIC)
Format: CALL NETSETMEM (<logical page>, <address>, [<A>], [<B>], [<C>], [<D>])
Function: Writes a sequence of 4 bytes in the memory.
<logic page> logic page number in bank 1 of the GR8NET (6000H~7FFFH).
<address> is the memory address visible to the Z80.
<A> = Address, <B> = Address + 1, etc.

NETSETMIX (command, GR8NET-BASIC)
Format: CALL NETSETMIX (<number>)
CALL NETSETMIX (<string>)

Function: Configures the audio mixer.
<number> – 16-bit numeric value with following format:

b15	b14	b13	b12	b11	b10	b9	b8	b7	b6	b5	b4	b3	b2	b1	b0
0	0	0	0	PSG		Y8950		OPLL		Wave		SCC		PCM	

Each 2 bits represent the following:

00 – Mute
01 – Left channel
10 – Right channel
11 – Both channels

If it is a 6-character string, each char means:

M – Mute
L – Left channel
R – Right channel
B – Both channels

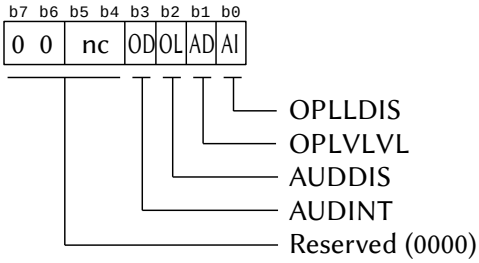
Another character, the setting will be preserved.

NETSETMMV (command, GR8NET-BASIC)
Format: CALL NETSETMMV (<numeric variable>)
Function: Defines the value of the memory manager. You can manage only the home page of RAM protected by the user.

NETSETNAME (command, GR8NET-BASIC)
Format: CALL NETSETMMV (<filename>)
Function: Defines the filename of the remote resource. The maximum length of the filename is 63 characters.

NETSETNTP (command, GR8NET-BASIC)
Format: CALL NETSETNTP (<A>, <B>, <C>, <D>, <TZF>)
Function: Defines the properties of the NTP server within the configuration of fixed IP and time setting flags. <A>, <B>, <C> and <D> define the IP address of the NTP server and <TZF> is the time zone update flag (see NETNTP command).

NETSETOPL (command, GR8NET-BASIC)
Format: CALL NETSETOPL (<flags>, <memory size>)
Function: Enables or disables the OPLL/Y8950, controls the doubling of the output amplitude and defines the size of the Y8950's audio memory. <memory size> defines the size of the audio memory in 8 Kbyte increments, the maximum and default value being 32 (32 * 8 = 256K). <flags> is a 1-byte value whose structure is described in the next page.



OPLLDIS – 0 – Enable OPLL output
1 – Disable OPLL output
OPLVLVL – 0 – Normal volume OPLL / Y8950 (default)
1 – Duplicated volume OPLL / Y8950
AUDDIS – 0 – Enables MSX-Audio (default)
1 – Disable MSX-Audio
AUDINT – 0 – Enables interrupt. MSX-Audio (default)
1 – Disable MSX-Audio interrupts

NETSETPATH (command, GR8NET-BASIC)
Format: CALL NETSETPATH (<path>)
Function: Defines the path of the remote resource. <path> is a name or string variable with a maximum length of 239 characters and no trailing bar and represents the absolute value.

NETSETPORT (command, GR8NET-BASIC)
Format: CALL NETSETPORT (<remote port>, <local port>)
Function: Defines the communication port numbers in the standard URI structure. If <local port> is 0, dynamic port number (default value) will be used. This command does not check the validity of the ports.

NETSETPSG (command, GR8NET-BASIC)
Format: CALL NETSETPSG (<value>)
Function: Configure the PSG.
<value> is a variable or bitmap constant, where bit set 0 defines whether PSG should be activated in (re) configuration and bit set 1 designates the port location to 0x10 if, on reset, the port is 0xA0 (built-in mirrored PSG). If the argument is omitted, the PSG will be reconfigured.

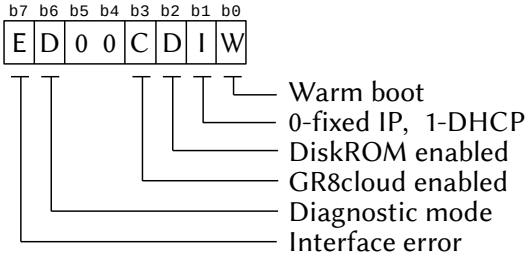
NETSETQSTR (command, GR8NET-BASIC)
Format: CALL NETSETQSTR (<parameter>)
Function: Defines the sequence of queries for processing remote resources. <parameter> is a variable or string constant that must start with the character "?" and have a maximum of 63 characters.

NETSETTSHN (command, GR8NET-BASIC)
Format: CALL NETSETTSHN (<name>)
Function: Defines the name of the time server. <name> is a variable or string constant that must have a maximum of 63 chars.

NETSNDDTG (command, GR8NET-BASIC)
Format: CALL NETSNDDTG (<file number> [, <A>, <B>, <C>, <D>] [, <RP>])
Function: Sends pending datagram / data to the remote host. <file number> is the BASIC file number. <A>, <B>, <C> and <D> represent the IP address of the remote device (can be omitted). RP is the port number of the remote device and can also be omitted.

NETSNDVOL (command, GR8NET-BASIC)
Format: CALL NETSNDVOL (<principal>, <SCC>, <waveform>, <PCM>, <OPLL>, <Y8950>, <PSG>)
Function: Read or change the volume of the audio generators. All arguments must be in the range 0 (mute) to 128 (maximum volume) and anyone can be omitted.

NETSTAT (command, GR8NET-BASIC)
Format: CALL NETSTAT (<Mode:>)
Function: Displays adapter status information. "Mode:" is a one-byte value with the following meaning:



NETSYSINFO (command, GR8NET-BASIC)

Format: CALL NETSYSINFO (<MSX version>, <clock frequency>, <T cycle performance>, <VDP version>, <vertical rate / VRAM size>)

Function: Returns system performance information and data.
<MSX version> – 0=MSX1; 1=MSX2; 3=MSX2+, 4 = MSX TR. For MSX turbo R, bit5 = 0 ⟶ R800; bit5 = 1 ⟶ Z80 and bit6 = 0 ⟶ DRAM mode; bit6 = 1 ⟶ ROM mode
<clock frequency> returns the slot clock (3 579 560)
<T cycle performance> returns the total number of times a 51 T cycle instruction (plus 8 of the M1 cycle) is executed in one second (60 671 * (51 + 8))
<VDP version> – 0 = TMS; 1 = V9938; 2 = V9958
<vertical rate / VRAM size> – Value of two bytes, where the lowest byte returns the frame rate (0 = 60 Hz, 1 = 50 Hz, 255 = error) and the highest byte returns the size of the block VRAM (1 = 8K; 2 = 16K; 4 = 32K; 8 = 64K; 16 = 128K; 255 = error).

NETRCHKS (command, GR8NET-BASIC)

Format: CALL NETRCHKS (<data block>, <address>, <number of bytes> <[, checksum]>)

Function: Calculates the 16-bit ckecksum of the contents of the RAM buffer of <data block> in bank 1 of the GR8NET. If the <checksum> variable is provided, it will receive the checksum, otherwise, the sum will be printed on the screen.

NETTELNET (command, GR8NET-BASIC)

Format: CALL NETTELNET ([<url | IP: port>],<signal>)

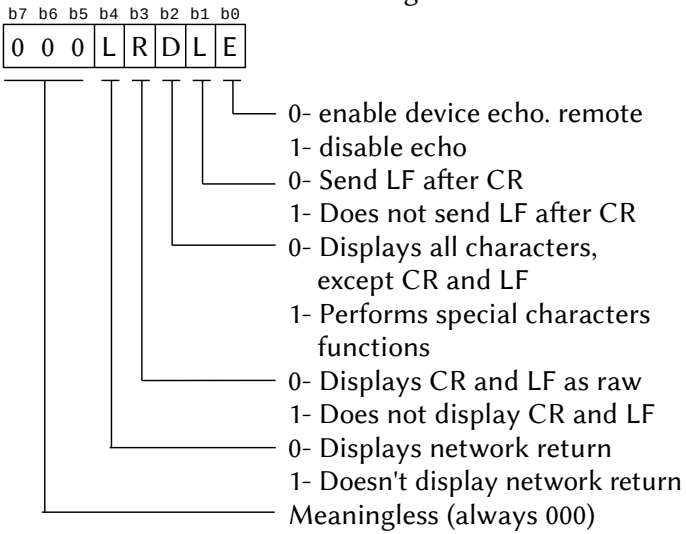
Function: Run telnet session using TCP. <signal> is a one-byte value where only bit 1 has meaning. If it is 1, it means that the telnet application does not add the character LF after the CR; if it is 0, pressing RETURN will send CR + LF to the remote host.

NETTERM (command, GR8NET-BASIC)

Format: CALL NETTERM ([<url | IP: port>],<flags>)

Function: Run telnet session using TCP. This command does not perform special translation of ESC code (&H1B). <flags>

must be kept at 0 if the remote device echoes what it receives back to the GR8NET. The meaning of the bits is as follows:



NETTGTMAP (command, GR8NET-BASIC)
Format: CALL NETTGTMAP [(<A>, <M>, <MRPD>)]
Function: Defines the type of Memory Mapper. Unlike NETSETMAP and that this command does not restart the machine.
<A> identifies the type of memory mapped and the location of the special register set.
<M> 0 – Reading disabled
1 – Reading enabled
2 – Automatic detection (default)
<MRPD> RAM mapped with pending disable bit
(0 – Enable; 1 – Disable)

NETTSYNC (command, GR8NET-BASIC)
Format: CALL NETTSYNC
Function: Displays and synchronizes the system time.

NETVARBRSTR (function, GR8NET-BASIC)
Format: CALL NETVARBRSTR (<alphanumeric variable>)
Function: Get the URL string of the location selected by the user in the browser and store it in <alphanumeric variable>. If the length exceeds 254 characters, an error will be generated.

NETVARBSIZE (function, GR8NET-BASIC)
Format: CALL NETVARBSIZE (<numeric variable>)
Function: Gets the size of the data loaded in bytes and stores it in <numeric variable>.

NETVARRWTH (command, GR8NET-BASIC)
Format: CALL NETVARRWTH (<current value>, <new limit>)
Function: Define the limit of the network RX window. <current value> must be a numeric variable that receives the current size (default is 0). <new limit> can be variable or numeric constant between 0 and 2047.

NETVARUDTO (command, GR8NET-BASIC)
Format: CALL NETVARUDTO (<current value>, <new limit>)
Function: Set UDP packet timeout for DHCP and DNS operations. <current value> is a variable that receives the current timeout and DHCP request retry count value. <new limit> is variable or constant, setting a new timeout value and counting DHCP request retries. Bits 7~0 identify the UDP timeout value (0-255), in periods of 100 ms. The default is 20 (2s). Bits 11~8 identify the number of DHCP request attempts attempted when GR8NET is started.

NETVER (function, GR8NET-BASIC)
Format: CALL NETVER
Function: Displays the GR8NET firmware version in the screen.

NEXTOR (command, Nextor)
Format: CALL NEXTOR
Function: Displays the list of commands added by Nextor.

NSCAN (command, Hitachi-BASIC version 2)
Format: CALL NSCAN
Function: Causes the Hitachi MB-H2 micro's built-in data reader to search for empty parts on the tape.

OFFHOOK (command, New Modem BASIC)
Format: CALL OFFHOOK
Function: Lift the phone handset.

OFFLINE (command, Network-BASIC, SVI-Modem BASIC)
Format: CALL OFFLINE [Network-BASIC]
Function: Disconnects the computer from the network. This instruction is only available to students and must be preceded by CALL NETINIT. Short version: _OFFL.
Format: CALL OFFLINE [SVI-Modem BASIC]
Function: Take the modem offline.

ON EVENT (n) GOSUB (statement, SFG-BASIC)
Format: CALL ON EVENT (<event number>) GOSUB
Function: Defines the subroutine that will be executed when a specific event occurs. <event number> can be:
1~4 – Stops when playback of the specified instrument ends.
5 – Stops when rhythm playback ends.
6 – Interrupts according to the time programmed in the FM unit timer.
If <event number> is omitted, the command will be applied to all events. The priority of the events is as follows:
1st – BASIC
2nd – Instrument 1
3rd – Instrument 2
4th – Instrument 3
5th – Instrument 4
6th – Rhythm
7th – Timer
Short version: _ON EVEN (<event number>) GOSUB.

ONHOOK (command, New Modem BASIC)
Format: CALL ONHOOK
Function: Hangs up the telephone handset.

ONLINE (command, Network-BASIC, SVI-Modem BASIC)
Format: CALL ONLINE [Network-BASIC]
Function: Connect the computer to the network. This instruction is only available to students. Eventually it may be necessary to run CALL NETINIT beforehand. Short version: _ONLI.
Format: CALL ONLINE [SVI-Modem BASIC]
Function: Put the modem in online mode.

PACLOAD (command, DM-System2 BASIC)
Format: CALL PACLOAD (<PAC address>, [@]<destination address> [,<length>])
Function: Reads data from the PAC SRAM (Pana Amusement Cartridge).
<PAC address> is the absolute address of the PAC cartridge and can vary from 0000H to 1FFDH.
<destination address> is the address to which the scanned data will be transferred. If preceded by "@", it means VRAM.
<length> is the number of bytes read. If omitted, 1024 bytes (1 block) will be read.

PACSAVE (command, DM-System2 BASIC)
Format: CALL PACSAVE ([@]<starting address>, <PAC address> ,<length>])
Function: Writes data in the PAC SRAM (Pana Amusement Cartridge).
<start address> is the address from which the data will be read. If preceded by "@", it means VRAM.
<PAC address> is the absolute address of the PAC cartridge and can vary from 0000H to 1FFDH.
<length> is the number of bytes to write. If omitted, 1024 bytes (1 block) will be written.

PALETTE (declaration, 3, Kanji-BASIC, Hangul-BASIC, RMSX BASIC)
Format: CALL PALETTE (<palette number>, <R>, <G>, <B>) [MSX-BASIC version 3, Kanji-BASIC, Hangul-BASIC]
Function: Specifies the colors for the palette. <palette number> can range from 0 to 15 and "<R>, <G>, <B>" from 0 to 7. All BASIC versions have the same syntax, except RMSX-BASIC.
Format: CALL PALETTE <palette/monitor> [RMSX BASIC]
Function: Selects the palette or monitor emulation to be used on the MSX1 computer emulated on a Turbo R machine by the rMSX emulator. <palette/monitor> can be MSX1, MSX2, GREEN or GRAY.

PAN (statement, Pioneer-BASIC)
Format: CALL PAN (<X axis>, <volume>, <string>)
Function: Generates sound according to the parameters provided.
<X axis> – Location of the generated sound. It can range from 0 to 255, with 0 corresponding to the extreme left and 255 to the extreme right.
<volume> can range from 0 to 7.
<string> – Macrocommands identical to those of the PLAY instruction for PSG, except for V, S, M and X. The string can contain up to 79 characters.

PATTERN (statement, SFG-BASIC)
Format: CALL PATTERN (<standard number>, <alphanum var (n)>)
Function: Defines the patterns of the rhythms through an alpha numeric matrix of one dimension (vector). Short version: _PATT.
<standard number> can be 7 or 8 (only two patterns can be defined).
<alfanum var (n)> points to a vector whose indices define different aspects:
xx$ (0) defines the size:
"3" – A quarter note times 3
"4" – A quarter note times 4
"8" – A quarter note times 8
xx$ (1) defines "close high-hat"
xx$ (2) defines "open high hat"
xx$ (3) defines "bass drum"
xx$ (4) defines "high tomtom"
xx$ (5) defines "low tomtom"
"Close high-hat" and "open high hat" cannot be played simultaneously.
The string for indices (1) to (5) must be composed of a sequence of "0" s and "1" s that represent units of 1/12 of a quarter note. The size depends on the value defined by xx$ (0):
xx$ (0) = "3" ⟶ 36 characters
xx$ (0) = "4" ⟶ 48 characters
xx$ (0) = "8" ⟶ 96 characters

PAUSE (command, MSX-BASIC 4, ChakkariCopy BASIC, DM- System2 BASIC, Hitachi BASIC 2)

Format: CALL PAUSE (<time>) [MSX-BASIC 4] [DM-System2 BASIC]

Function: Pauses the execution of the program in BASIC. <time> is specified in milliseconds and can range from 0 to 65,535. It can be aborted by CTRL+STOP.

Format: CALL PAUSE [ChakkariCopy BASIC]

Function: Put the Chakkari Copy cartridge in pause mode.

Format: CALL PAUSE [Hitachi BASIC 2]

Function: Put the internal data reader of the Hitachi MB-H2 micro in pause mode.

PCM FREQ (command, MSX-Audio)

Format: CALL PCM FREQ (<frequency>)

Function: Defines the sampling frequency for ADPCM. <frequency> can range from 1,800 to 49,716 Hz.

PCM VOL (controller, MSX-Audio)

Format: CALL PCM VOL (<volume>)

Function: Sets the reproduction volume for ADPCM and PCM. <volume> can range from 0 to 63. Initial values are 55 for ADPCM and 32 for PCM.

PCMON (declaration, DM-System2 BASIC)

Format: CALL PCMON ([@]<starting address>, [@]<ending address>, <rate> [,<loop>])

Function: Plays data through PCM on MSX2 onwards. Requires PCM driver.

<start address> of the data to be reproduced. If preceded by "@", it means VRAM.

<final address> of the data to be reproduced. If preceded by "@", it means VRAM.

<rate> can be: 0 ⟶ 15.75 Khz 2 ⟶ 5.25 KHz
1 ⟶ 7.875 Khz 3 ⟶ 3.9375 KHz

<loop> defines the number of times the data will be played. It can vary from 1 to 255. 0 = Infinite loop.

PCMPLAY (declaration, 4)
Format: CALL PCMPLAY (@<start adress>, <end adress>, <sampling rate> [,S])
Function: Plays PCM data stored in RAM or VRAM.
<sampling rate> can be 0 to 3.
<start adress> and <end adress> are the starting and ending addresses.
[,S] specifies VRAM.

PCMREC (command, 4)
Format: CALL PCMREC (@ <start adress>, <end adress>, <sampling rate>, [[<trigger level>], [<save>],S])
Function: Writes PCM data to RAM or VRAM.
<start adress> and <end adress> can range from 0000H to FFFFH,
<sampling rate> can vary from 0 to 3,
<trigger level> can vary from 0 to 127,
<save> can be 0 (unsaved) or 1 (saves in RAM),
[,S] records in the VRAM.

PDIAL (remote, New Modem BASIC, SVI Modem BASIC)
Format: CALL PDIAL (<string variable>),<numeric variable> [New Modem BASIC]
Function: Calls a specific phone number via pulse dialing. <string variable> stores the phone number to be called, where only numbers are allowed and the character "–", which add 1 second wait. <numeric variable> returns the state: if it is 0, the input value is correct; if it is "1" it is not.
Format: CALL PDIAL ("<phone number>") [SVI Modem BASIC]
Function: Calls a specific phone number via pulse dialing. <phone number> must be in quotes and only numeric characters are allowed.

PEEK (function, DM-System2 BASIC, Network-BASIC)
Format: CALL PEEK ([@]<address> [,<variable>]) [DM-System2 BASIC]
Function: Reads a byte from any area of Main RAM, VRAM or Memory Mapper.

<address> – Is the address to be read. If greater than 65 534, the Memory Mapper specifications will be used. If it is preceded by "@", it indicates VRAM.
<variable> is a numeric variable that will receive the value of the byte read. If omitted, the value will be displayed on the screen.

Format: CALL PEEK (<variable>, <address>[, <student number>] [,<N>]) [Network-BASIC]

Function: Reads a byte of data from NetRAM (student or teacher) or NetRAM / RAM (teacher only).
<variable> receives the value of the read byte.
<address> must be between 7800H and 7FFFH for NetRAM.
<student number> is a number between 1 and 15. Only the teacher can use this parameter.
<N> must be used to read an address in NetRAM. Useful only for the teacher.

PEEKS (function, DM-System2 BASIC)

Format: CALL PEEKS ([@]<address>, <size>, <string variable>)

Function: Reads several consecutive bytes in Main RAM, VRAM or Memory Mapper, converts them to characters and stores them in a string variable.
<address> – Is the address to be read. If greater than 65 534, the Memory Mapper specifications will be used. If it is preceded by "@", it indicates VRAM.
<size> – Number of bytes to be read, ranging from 1 to 255.
<string variable> receives the bytes read.

PEEKW (function, DM-System2 BASIC)

Format: CALL PEEKW ([@]<address> [,<variable>])

Function: Reads two consecutive bytes in Main RAM, VRAM or Memory Mapper.
<address> – Is the address to be read. If greater than 65 534, the Memory Mapper specifications will be used. If it is preceded by "@", it indicates VRAM.
<variable> receives the value read. If not specified, the value read will be displayed on the screen in hex format.

PHRASE (macro-declaration, SFG-BASIC)
Format: CALL PHRASE (<track number>, <playlist> [,<brand>])
Function: Writes the audio playback data to the specified track.
Short version: _PHRA.
<track number> can vary from 1 to the value defined by _PLAY.
<brand> is a number that can vary from 1 to 254. If omitted, it will be considered equal to <track number>.
<playlist> contains the music macros.

A~G	Plays an encrypted note with duration n (1~64, pattern 4).
# or +	Sustain.
–	Flat.
!	Returns the note to its original value (K and S commands).
On	Octave (n ⟶ 1 to 8; the default is 3).
Nn	Pitch (n ⟶ 25 to 120).
Ln	Note length (n ⟶ 1 to 64, default: 4).
.	Duration increased by 50%.
Rn	Pause of duration n (n ⟶ 1 to 64, default is 4).
Wn	Note duration in 1/96 units (n: 1 to 96).
Tn	Time (n ⟶ 1 to 200, specified by _TEMPO).
Vn	Volume (n ⟶ 1 to 100, the default is 50)
&	Ligature
Mn	Period in units of n / 4 (n ⟶ 3 to 8)
/ /	The string between two bars will be considered a block of duration specified by Mn.
Sn	"n" specifies the number of "Sharps" (#).
Kn	"n" specifies the number of "Flats" (b).
% n	"Staccato" and "tenuto". "n" specifies the time proportion of the note will be played (0% to 100%).
[ ]	Wake up. Comma-separated strings inside the bracket are played simultaneously.
{ }n	Define the notes between {} in n. (n = 1~64, default is Ln)

PITCH (declaration, MSX-Music)
Format: CALL PITCH (<n>)
Function: Fine adjustment of the sound. <n> can range from 410 to 459, the default value being 440 (central "La" tone).

PLAY (macro-declaration, MSX-Music/Audio, Hitachi-BASIC, SFG-BASIC)

Format: CALL PLAY (<n>, <numeric variable>) [Music /Audio]

Function: Returns in the <numeric variable> the state of the voice <n> of the OPLL (touching [–1] or not [0]). <n> can range from 0 to 9. If 0, all voices are checked. 1 to 9 checks the respective voice.

Format: CALL PLAY [Hitachi-BASIC 2]

Function: Puts the internal data reader of the Hitachi MB-H2 micro in playback mode. This instruction does not support files in ASCII mode (BASIC or data).

Format: CALL PLAY (<instrument>, <range> [,<brand>]) [SFG-BASIC]

Function: Play the previously written song with the CALL PHRASE instruction.
<instrument> is a number from 1 to 4
<range> is a number from 1 to 9, with 2 to 8 having to be previously defined by the CALL TRACK instruction and 9 when reproducing via the musical keyboard.
<brand> is a number between 1 and 255 to specify the CALL PHRASE tag used for reproduction. If omitted, the same <track> number is considered.

PLAY MK (statement, MSX-Audio)

Format: CALL PLAY MK (<matrix name>)
CALL PLAY MK (<start address>, <end address>)
CALL PLAY MK (A), where the sequence A must be previously declared in the DIM and REC MK instructions.

Function: Plays file recorded by the musical keyboard.

PLAY PCM (statement, MSX-Audio)

Format: CALL PLAY PCM (<file number>, <offset>, <size>, <sampling frequency>)

Function: Plays an audio file via PCM /ADPCM.
<file number> – Audio file number (0 to 15).
<offset> – Offset in units of 256 bytes.
<size> – Size in bytes of the audio file.
<sampling frequency> – Can range from 1,800 to 49,716 Hz for ADPCM and from 1,800 to 16,000 Hz for PCM.

POKE (statement, DM-System2 BASIC, Network-BASIC)
Format: CALL POKE ([@]<address>, <value>)
[DM-System2 BASIC]
Function: Writes a byte in any area of Main RAM, VRAM or Memory Mapper.
<address> – Is the address to be read. If greater than 65 534, the Memory Mapper specifications will be used. If it is preceded by "@", it indicates VRAM.
<value> is the value to be written. It must be in decimal between 0 and 255, it can also be an expression.
Format: CALL POKE (<value>, <address> [,<student number>] [,<N>]) [Network-BASIC]
Function: Writes a byte of data in NetRAM (student or teacher) or NetRAM / RAM (teacher only).
<value> must be a decimal number between 0 and 255.
<address> must be between 7800H and 7FFFH for NetRAM.
<student number> is a number between 1 and 15. Only the teacher can use this parameter.
<N> must be used to write an address on NetRAM. Useful only for the teacher.

POKES (declaration, DM-System2 BASIC)
Format: CALL POKES ([@]<address>, <string>)
Function: Converts a string to a sequence of bytes and saves them in any area of Main RAM, VRAM or Memory Mapper.
<address> – Is the writing start address. If greater than 65 534, the Memory Mapper specifications will be used. "@" Indicates VRAM.
<string> = String of characters to be converted to bytes.

POKEW (declaration, DM-System2 BASIC)
Format: CALL POKEW ([@]<address>, <value>)
Function: Writes two consecutive bytes to Main RAM, VRAM or Memory Mapper.
<address> – Is the address to be written. If greater than 65 534, the Memory Mapper specifications will be used. If it is preceded by "@", it indicates VRAM.
<value> is the value to be written. It must be in decimal between 0 and 65,535, it can also be an expression.

PON (command, Network-BASIC)
Format: CALL PON
Function: Starts the student search. This instruction is only available to the teacher.

PRINTERSETUP (command, FM-X BASIC)
Format: CALL PRINTERSETUP
Function: It allows printing hiragana and graphic characters on the printer connected to the Fujitsu FM-7 computer when this machine is connected to the micro FM-X using the MB22450 interface.

QDFILES (command, QuickDisk BASIC)
Format: CALL QDFILES [("QD [n]:")]
Function: Lists the contents of the specified Quick Disk device in long format, with filenames, attributes and file sizes. The listed attributes are as follows:
01 – MainRAM binary file
02 – BASIC in tokenized format
03 – BASIC or DATA in ASCII format
0B – VRAM binary file
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.

QDFORMAT (command, QuickDisk BASIC)
Format: CALL QDFORMAT
Function: Formats a QuickDisk excluding all existing files. The data is recorded on a spiral track on a 2.8 inch disc. A QuickDisk can save a maximum of 20 files and has a capacity of 64 Kbytes each side, with a maximum capacity of 128 Kbytes.

QDKEY (command, QuickDisk BASIC)
Format: CALL QDKEY (<parameter>)
Function: Modifies the content of the function keys, except F7, when a QuickDisk unit is connected. When one MSX is started with a connected QuickDisk drive, the contents of most function keys are modified. CALL QDKEY allows you to switch between content.

```
Key      Content        New Command
F1       _RUN           CALL RUN
F2       _LOAD          CALL LOAD
F3       _BLOAD         CALL BLOAD
F4(*)    list           LIST
F5(*)    run            RUN + [RETURN]
F6(**)   color 15,4,7   COLOR 15,4,7 + [RETURN]
F7       _QDKEY         CALL QDKEY
F8       _SAVE ("QD:    CALL SAVE ("QD:
F9       _BSAVE ("QD:   CALL BSAVE ("QD:
F10      _QDFILES       CALL QDFILES
```

(*) Generally unchanged

(**) Unchanged on Japanese, Korean machines, Philips VG-8000 and VG-8010 (not on version 8010F), Sanyo PHC-28S

<parameter> – If 0, the default key content will be reloaded (except F7). With any other number or without a parameter the QuickDisk content will be loaded.

QDKILL (command, QuickDisk BASIC)

Format: CALL QDKEY (["QD [n]:]<filename>")

Function: Deletes the last QuickDisk file. Attempting to delete another file will return an error message.

QD [n] specifies the QuickDisk device to be accessed. It can range from 0 to 7, the default being 0.

<filename> is the file to be deleted and must be in the format 8.3 characters.

RAMDISK (command, Disk-BASIC 2nd version)

Format: CALL RAMDISK (<max size>, [<created size>])

Function: Creates a RAMDISK with <maximum size> and optionally returns the actual <created size>. RAMDISK is accessed via the H: drive.

RCANCEL (command, SFG-BASIC)

Format: CALL RCANCEL

Function: Cancels the rhythm instruments. Using this instruction, the total of simultaneous voices goes from 6 to 8.

RCVMAIL (command, Network-BASIC)
Format: CALL RCVMAIL (<student number>)
Function: Receives data from a student's sending mailbox in the teacher's receiving mailbox. This instruction is only available to the teacher. Mailboxes are special 256-byte areas reserved in the teacher and student's NetRAM. <student number> can vary from 1 to 15. Short version: _RCVM.

REBOOT (remote, Hangul-BASIC 4, Rookie Drive BASIC)
Format: CALL REBOOT
Function: Causes a "hot" restart of the system.

REC (command, Hitachi-BASIC version 2)
Format: CALL REC
Function: Puts the internal data reader of the Hitachi MB-H2 micro in recording mode. This instruction does not support files in ASCII mode (BASIC or data).

REC MK (remote, MSX-Audio)
Format: CALL REC MK (<matrix name>)
CALL REC MK (<start address>, <end address>)
Function: Records a file played by the musical keyboard.

REC PCM (command, MSX-Audio)
Format: CALL REC PCM (<file number> [, SYNC] [,<offset>
[,<size>] [,<sampling frequency>]
Function: Record audio in memory through the microphone.
<file number> – Number of the file to be written (0 to 15)
SYNC – If 0, MSX-Audio waits until an audio signal is detected. If it is 1, recording starts immediately.
<offset> – Offset in units of 256 bytes
<size> – Size of the audio file
<sampling frequency> – Can range from 1,800 to 49,716 Hz for ADPCM and from 1,800 to 16,000 Hz for PCM.

RECEIVE (command, Network-BASIC)
Format: CALL RECEIVE ([[<drive letter>:] <filename>],
<student number>)

Function: Receives the BASIC program from a student's computer. This instruction can be used by the teacher and students who have been authorized by the teacher with CALL ENACOM. <drive letter> can be "A:" or "B:" and can only be used by the teacher. <student number> can vary from 1 to 15. Short version: _RECE.

RECFILE (command, New Modem BASIC)
Format: CALL RECFILE (<string variable>), <numeric variable>
Function: Receive a file using a specific protocol
<string variable> contains the name of the file to be received (may include the name of the drive, if omitted, the file will be saved to the current active drive). If a file with the same name already exists on the disk, the first letter will be replaced by "$", which will occur up to the fourth character.
<numeric variable> stores the protocol:
0 – Xmodem or Xmodem-1K
3 – Ymodem (allows you to receive multiple files simultaneously)
Upon return, <numeric variable> will contain the status:
0 – Receipt was done correctly
1 – Signal dropped (usually the connection is broken)
2 – Timed out (download has not started)
3 – Aborted operation with CTRL + X
4 – Many breaks (waiting times)
5 – Not used (no effect)
6 – Disk full
7 – File not found
8 – Recording error (write-protected disc or there is no disc)
9 – Empty file
10 – Too many attempts

RECMOD (command, MSX-Audio)
Format: CALL RECMOD (<recording mode>)
Function: Sets the recording mode for the musical keyboard.
<recording mode> is a value from 0 to 3:

0 - Mute (does not record)
1 - Records the melody played on the keyboard (def)
2 - Records, in another area, the reproduction of a melody already recorded
3 - Records the performance and playback of a melody already recorded

REMOTE (remote, Pioneer-BASIC)
Format: CALL REMOTE (<device number>, <string>)
Function: Controls external devices.
<device number> can range from 0 to 15, but devices 0, 1 and 2 are already assigned (Commands 3 to 15 must be assigned with CALL DEF UNIV.):
0 – Pioneer Laser Vision Player LD-700
1 – Pioneer Laser Vision Player LD-1100
2 – Pioneer Component Display SD-26
<string> contains a character code of up to 16 commands according to the following table (the "+" character can be omitted):

Functions of the LD-700 model (device 0):

A+	48	Repeat A	M+	58	Multi-speed forward
A-	44	Repeat B	M–	55	Multi-speed reverse
C+	47	Inc. multi-speed	P+	17	Play
C–	46	Dec. multi-speed	P@	16	Reject
D+	43	Presents number of frame/chapter	P/	18	Pause
			S+	54	Pauses frame to frame forward
F+	10	Quick search			
F–	11	Rev. quick search	S–	50	Pauses frame to frame reverse
L+	4B	Audio 1 / left			
L–	49	Audio 2 / right	T+	51	Fast forward (3x)
L@	4A	Estéreo	T–	59	Fast rew (3x)
			X+	45	Clear

Functions of the LD-1100 model (device 1):

D+	Displays frame num	P+	Play
D–	Displays chapter num	P@	Reject
F+	Quick search	P/	Pausa
F–	Rev. quick search	S+	Pauses frame to frame forward
L+	Audio 1 / left		
L–	Audio 2 / right	S–	Pauses frame to frame reverse
M+	Slow search ahead		
M–	Slow search reverse	T+	Fast forward (3x)
		T–	Fast rew (3x)

Functions of the SD-26 model (device 2):

1	01	Channel A	F+	10	Increment channel (+)
2	02	Channel B	F–	11	Decrement channel (–)
3	03	Channel C	K1	0C	Input: TV
4	04	Channel D	K2	0D	Input: Video-Disc
5	05	Channel E	K3	0E	Input: Video 1
6	06	Channel F	K4	0F	Input: Video 2
7	07	Channel G	O@	1C	Turns on / off
8	08	Channel H	V+	0A	Increases volume (+)
9	00	Channel I	V–	0B	Decrease volume (–)
0	00	Channel J		48	Sleep
C–	46	Channel K		49	Mute
C+	47	Channel L		4A	Display call
	4D	Channel M			
	4E	Channel N			
	4F	Channel O			
	50	Channel P			

Other functions:

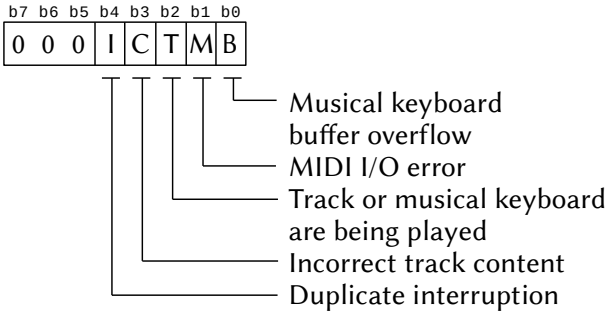
M@	ID	Turns on / off tape monitor
P–	15	Reverse play (for tape-deck)
W		Hold video (for laser vision player)
R+	14	Records (for tape-deck)
R–	12	Mute record (for tape-deck)

REPORT (System variable, SFG-BASIC)

Format: CALL REPORT ([<error flag>] [, <mark number>] [, <number of repetitions remaining>])

Function: Returns the system variables. Short version: _REPO.

<error flag> integer variable (only the lowest 5 bits are valid):



<mark number> is an integer variable that returns the mark number of the last reproduced section.

<number of repetitions remaining> returns the number of times the rhythm will still be played.

RESET (command, RMSX BASIC)
Format: CALL RESET
Function: Restart the MSX1 or MSX2 computer emulated on a Turbo R machine by the rMSX emulator. It is a hot reset, as with DEFUSR=0: X=USR (0).

REW (command, Hitachi-BASIC version 2)
Format: CALL REW
Function: Causes the internal data reader of the Hitachi MB-H2 micro to rewind the tape.

RHYTHM (statement, SFG-BASIC)
Format: CALL RHYTHM (<repetition number> [,<brand number>])
Function: Reproduces the rhythm patterns selected by the CALL SELPATTERN command. Short version: _RHYT.
<number of repetitions> specifies the number of repetitions in 1/4 note units.
<brand number> can range from 1 to 254. If omitted, the value 10 will be used.

RMDIR (command, Disk-BASIC 2nd version)
Format: CALL RMDIR (<subdirectory>)
Function: Removes the specified <subdirectory>.

RSTOP (declaration, SFG-BASIC)
Format: CALL RSTOP
Function: Stops rhythm playback. Short version: _RSTO.

RTCINI (command, Hangul-BASIC 3)
Format: CALL RTCINI
Function: Resets the content of the RTC SRAM to the initial standard corresponding to MSX1.

RTSOFF (command, New Modem BASIC)
Format: CALL RTSOFF
Function: Turns off the carrier wave (RTS = Request To Send). This instruction works only when the DTR (Data Terminal Ready) signal is active (CALL DTRON).

RTSON (command, New Modem BASIC)
Format: CALL RTSON
Function: Turns on the carrier wave (RTS = Request To Send). This instruction works only when the DTR (Data Terminal Ready) signal is active (CALL DTRON).

RUN (command, Network-BASIC, QuickDisk-BASIC, X-BASIC)
Format: CALL RUN [([<student number>], [<line number>])] [Network-BASIC]
Function: Executes the BASIC program that is in the memory of a student's computer. This instruction is only available to the teacher. <student number> can vary from 0 to 15. If omitted or equal to 0, the programs of all computers will be executed The program will run from line <line number>, if specified.
Format: CALL RUN ("[QD [n]:]<filename>") [QuickDisk-BASIC]
Function: Load a BASIC file from the specified Quick Disk device into MSX memory and execute it.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8.3 characters.
Format: CALL RUN [X-BASIC]
Function: Compiles and executes the BASIC program present in the MSX memory.

SAVE (command, DM-System2 BASIC, QuickDisk-BASIC)
Format: CALL SAVE ("[<device>:] [\ <path>] [\]<filename>", [@]<source address>, <size> [,<offset>]) [DM-System2 BASIC]
Function: Saves data to a new file or somewhere in an existing file.
<device> can be drive A: to H: or COM: for computers connected with RS232C.
<path> specifies the location of the folder or file.
<filename> is the name of the file to be saved.
<destination address> is the source address of the data. If preceded by "@" it means VRAM.
<size> specifies the number of bytes to save.
<offset> specifies the offset in the target file.

Format: CALL SAVE ("[QD [n]:]" <filename> "[,A])
[QuickDisk-BASIC]
Function: Saves data from memory or a BASIC program to a QuickDisk device. The data will always be saved in ASCII text. The BASIC program can be saved in ASCII or tokenized text.
QD [n] specifies the QuickDisk device to be used. It can range from 0 to 7, the default being 0.
<filename> must be in the format 8.3 characters.
[,A], if specified, saves the BASIC file as ASCII text.

SAVE PCM (command, MSX-Audio)
Format: CALL SAVE PCM (<filename>, <file number>)
Function: Save audio file to disk.
<filename> is name of the file to be written to the disc
<file number> is the file number in the audio memory. It can range from 0 to 15.

SCLOAD (command, Pioneer-BASIC)
Format: CALL SCLOAD [(<filename>)]
Function: Load data from the cassette to VRAM for display on the screen (only available for Screen 2)

SCOPY (command, Hitachi-BASIC version 3)
Format: CALL SCOPY (<c1> [,<c2>, <c3>, <c4>.... <c15>])
Function: Sends to the printer a copy of a graphic screen in Screens 2, 4 or 5 using a formula based on the selected colors. The difference with CALL CSCOPY is unknown.

SCSAVE (remote, Pioneer-BASIC)
Format: CALL SCSAVE (<filename>, [<baud rate>])
Function: Records VRAM data on the cassette. <baud rate> can be 1 (for 1200 baud) or 2 (for 2400 baud). If not specified, the baud rate defined in SCREEN will be used. Command available only for Screen 2.

SEARCH (remote, Pioneer-BASIC)
Format: CALL SEARCH (<type>, {F | C}, <frame/chapter number>)

Function: Search the specified frame or chapter in the Laser Vision Player. <type> can be 0 for LD-700 or 1 for LD-1100. "F" search for a frame and "C" search for a chapter. <frame/chapter number> can vary between 0 and 54 000.

SELPATTERN (statement, SFG-BASIC)

Format: CALL SELPATTERN (<standard number>)

Function: Select the rhythm patterns for playback. Short version: _SELP.
<pattern number> can vary from 1 to 8, with 1 to 6 being the ROM patterns and 7 and 8 being the patterns defined by the _PATTERN command. The ROM defaults are:

1 – 16 beats	4 – Rock
2 – Slow rock	5 – Disco
3 – Waltz	6 – Swing

SELVOICE (statement, SFG-BASIC)

Format: CALL SELVOICE ([<voice 1>] [, <voice 2>]... [, <voice 8>])

Function: Select up to 8 voices chosen from the data loaded by the _CLDVOICE command and execute them. <voice x> must correspond to the voice number created by the FM Voicing Program. The numbers from 49 to 56 are voices defined by the _MODISNT command (these numbers will be used by default if the voice parameters are omitted).

SEND (command, Network-BASIC)

Format: CALL SEND [([<unit name>:] <filename>] [, <student number>])]

Function: Sends the BASIC program to (other) students' computers. This instruction can be used by the teacher and students who have been authorized by the teacher with CALL ENACOM. <unit name> can be "A:" or "B:" and <student number> can range from 0 to 15. If <filename> is omitted, the BASIC program that is in the micro sender's memory will be sent.

SENDFILE (command, New Modem BASIC)
Format: CALL SENDFILE (<string variable>), <numeric variable>
Function: Send a file using a specific protocol.
<string variable> contains the name of the file to be sent (may include the name of the drive, if omitted, the file will be read from the current active drive).
<numeric variable> stores the protocol to be used:
1 – Xmodem.
2 – Ymodem-1K.
3 – Ymodem (allows only one file at a time).
Upon return, <numeric variable> will contain the status:
0 – Submission was successful.
1 – Signal dropped (usually the connection is broken).
2 – Timed out (upload has not started).
3 – Aborted operation with CTRL + X.
4 – Many breaks (waiting times).
5 – Not used (no effect).
6 – Disk full.
7 – File not found.
8 – Recording error (write-protected disc or no disc).
9 – Empty file.
10 – Too many attempts.

SEOFF (declaration, DM-System2 BASIC)
Format: CALL SEOFF
Function: Stops the playback of the sound effect. Requires SE driver.

SEON (declaration, DM-System2 BASIC)
Format: CALL SEON (<number>)
Function: Reproduces a sound effect from a table. Requires SE driver. <number> is the number of the sound effect to be played. It can range from 0 to 255, with 0 interrupting playback.

SET PCM (command, MSX-Audio)
Format: CALL SET PCM (<file number>, <device number>, <mode>, <parameter 1>, <parameter 2>, <sampling freq>)
Function: Defines parameters for the audio files. The parameters are defined for the following commands:

CONVA CONVP COPY PCM
LOAD PCM MK PCM PLAY PCM
REC PCM SAVE PCM

<file number> – File number in the audio memory. It can range from 0 to 15.

<device number> follows the table below:

device	Device name	Mode	Parameter 1	Parameter 2
0	External RAM	0/1	–	Size
5	VRAM	0/1	Adress	Size

The address and size are defined in units of 256 bytes.

<mode> can be: 0 – ADPCM, 1 – PCM

<sampling freq> can range from 1,800 to 49,716 Hz for ADPCM and from 1,800 to 16,000 Hz for PCM.

SETBIN (command, DM-System2 BASIC)
Format: CALL SETBIN (@<address>)
Function: Specifies the starting address of the binary table according to the binary system. <address> is the starting address of the binary table. The least significant bit is ignored. It is necessary to use "@" in front of <address> to put it in VRAM, otherwise an error will occur because the table cannot be placed in the Main RAM.

SETPLT (command, DM-System2 BASIC)
Format: CALL SETPLT (<address>)
Function: Defines the starting address of the color palette table. The table is 32 bytes long and the default address is C0000H.

SETSE (command, DM-System2 BASIC)
Format: CALL SETSE (<address>)
Function: Defines the starting address of the sound effects table. (Requires SE driver). <address> is the starting address of the table (0 to FFFFH), the initialization value being C000H.

SIN (function, DM-System2 BASIC)
Format: CALL SIN (<variable>, <angle>, <value>)
Function: Returns the sine of an angle. The result is obtained by multiplying the sine of the angle by a numerical value.

<variable> is a numeric variable that will receive the result.
<angle> is the angle value in degrees.
<value> is a number of two bytes (integer value).

SJIS (statement, Kanji-BASIC)
Format: CALL SJIS (<string variable>, <Kanji characters>)
Function: Converts a character in JIS code to a value of 4 hexadecimal digits.
<string variable> receives the 4 hexadecimal digits in ASCII
<Kanji characters> is a 2-byte Kanji character string where only the first one will be converted.

SNDCMD (command, Network-BASIC)
Format: CALL SNDCMD (<instruction>, [<student number>])
Function: Sends BASIC instructions to the student's computer and executes them. CHR$(13) is sent at the end of the instruction and it is possible to send several by separating them with CHR$(13). This instruction is only available to the teacher. <student number> can vary from 1 to 15. Short version: _SNDC.

SNDMAIL (command, Network-BASIC)
Format: CALL SNDMAIL (<student number>)
Function: Sends data from the teacher's mailbox to a student's mailbox. This instruction is only available to the teacher. Mailboxes are special 256-byte areas reserved in the teacher and student NetRAM. <student number> can vary from 1 to 15. Short version: _SNDM.

SNDRUN (command, Network-BASIC)
Format: CALL SNDRUN ([[<unit name>:]<filename>] [,<student number>])
Function: Send the BASIC program to the student's computer and execute it. This instruction is only available to the teacher. If a student already has a BASIC program in memory, it will be deleted and the student will receive a new one. <unit name> can be "A:" or "B:" and <student number> can range from 0 to 15. If <filename> is omitted, the BASIC program that is in the micro sender's memory will be sent. Short version _SNDR.

SOUND (statement, SFG-BASIC)
Format: CALL SOUND (<instrument number>, <control mode>
[,<pitch>] [,<fine tuning>] [,<speed>] [,<volume>])
Function: Controls instruments directly.
<instrument number> chooses the instrument from those defined by the _INST instruction.
<control mode> can be:
0 – No key on / offline key
1 – Key on (note is audible)
2 – Key off (the note is at zero volume)
<pitch> can range from 25 to 120.
<fine adjustment> of the pitch. It can range from 0 to 100.
<volume> can range from 0 to 100, with 100 being the maximum volume (default).

SPEAKEROFF (remote, New Modem BASIC)
Format: CALL SPEAKEROFF
Function: Turns off the speaker.

SPEAKERON (remote, New Modem BASIC)
Format: CALL SPEAKERON
Function: Turn on the speaker.

SPOLOFF (command, Printer-BASIC)
Format: CALL SPOLOFF
Function: Disables the print spooler but does not empty the 32 Kbyte buffer. To clear the temporary buffer, it is necessary to use LPRINT or LLIST.

SPOLON (command, Printer-BASIC)
Format: CALL SPOLON
Function: Activates the print spooler, reserving a 32 Kbyte buffer for it.

STANBY (statement, SFG-BASIC)
Format: CALL STANDBY
Function: Temporarily stop playback. Short version: _STAN.

START (command, Mega Assembler, SFG-BASIC)
Format: CALL START [Mega Assembler]
Function: Calls the Mega Assembler by initializing its variables. To call the MA without initializing, use _ASM.
Format: CALL START [SFG-BASIC]
Function: Resumes playback interrupted by _STANDBY. Short version: _STAR.

STATUS (declaration, DM-System2 BASIC)
Format: CALL STATUS
Function: Displays the list of installed drivers for DM-System2.

STDBY (command, Hitachi-BASIC version 2)
Format: CALL STDBY
Function: Puts the internal data reader of the Hitachi MB-H2 micro in standby/suspend mode to save battery power.

STOP (command, Hitachi-BASIC, Network-BASIC, SFG-BASIC)
Format: CALL STOP [Hitachi-BASIC 2]
Function: Stops the tape movement in the internal data reader of the Hitachi MB-H2 micro.
Format: CALL STOP (<student number>) [Network-BASIC]
Function: Stops the BASIC program running on the student's computer. This instruction is only available to the teacher. <student number> can vary from 1 to 15. If omitted or equal to zero, execution will be interrupted on all student computers.
Format: CALL STOP (<instrument>) [SFG-BASIC]
Function: Suspend the playback of a specific instrument and, optionally, the digitization of the musical keyboard (when assigned to an instrument instead of a track by the CALL PLAY instruction). <instrument> must be a number between 1 and 4.

STOPM (statement, MSX-Audio, MSX-Music)
Format: CALL STOPM
Function: Stops the music played by MSX-Audio or MSX-Music.

SYMBOL (statement, Pioneer-BASIC)
Format: CALL SYMBOL (X, Y), CHR$ (<character code>), [<hor>], [<vert>], [<color>], [<rotation>]
Function: Displays a character in Screen 2 in the coordinates (X, Y). Optional parameters are as follows:
<character code> – ASCII character code
<hor> – Horizontal size multiplier. It can be between 1 and 32. If omitted, the value used will be 1.
<vert> – Vertical size multiplier. It can be between 1 and 24. If omitted, the value used will be 1.
<color> – Color code from 0 to 15. If omitted, the color defined by the COLOR command will be used.
<rotation> defines the character rotation.
0 – No rotation
1 – 90 degree rotation to the right
2 – 180 degree rotation to the right
3 – 270 degree rotation to the right

SYNCOUT (command, SFG-BASIC)
Format: CALL SYNCOUT
Function: Sends a synchronization signal to the cassette register. Short version: _SYNC.

SYSOFF (command, DM-System2 BASIC)
Format: CALL SYSOFF
Function: Uninstall DM-System2 BASIC and return to standard MSX-BASIC.

SYSON (command, DM-System2 BASIC)
Format: CALL SYSON
Function: Initializes the DM-System2 BASIC.

SYSTEM (command, Disk-BASIC, DM-System2 BASIC)
Format: CALL SYSTEM [Disk-BASIC version 1]
Function: Calls MSXDOS.
Format: CALL SYSTEM [("[<device>:] [\ <path>] [[\]<filename>]")] [Disk-BASIC version 2]
Function: Calls MSXDOS2, optionally executing the specified file or entering the subdirectory.

Format: CALL SYSTEM [DM-System2 BASIC]
Function: Uninstall DM-System2 and return to the standard MSX-BASIC. If preceded by CALL SYSOFF, uninstall DM-System2 and call MSXDOS.

TABOFF (command, Hitachi-BASIC version 3)
Format: CALL TABOFF
Function: Disables the Drawing Tablet application on the Hitachi MB-H3 micro.

TABON (command, Hitachi-BASIC version 3)
Format: CALL TABON
Function: Launch the Drawing Tablet application on the Hitachi micro MB-H3.

TALK (statement, Network BASIC)
Format: CALL TALK (<message>, [<micro number>])
Function: Sends a message of up to 56 characters to the teacher or another student, when allowed by the teacher with CALL ENACOM. This instruction is only available to students. <micro number> can vary from 1 to 15 and can be obtained through CALL WHO. If it is 0, the message will be sent to the teacher. If, after sending the message,<micro number> contains 255, it failed, if it contains 0, the message was sent successfully.

TDIAL (remote, New Modem BASIC, SVI Modem BASIC)
Format: CALL TDIAL (<string variable>),<numeric variable> [New Modem BASIC]
Function: Calls a specific phone number via tone dialing. This instruction can only be used in a Terminal program.
<string variable> stores the phone number to be called, where only the characters "0 123 456 789-AaBbCcDd *#" are allowed (the character "–" corresponds to a 1 second wait).
<numeric variable> returns the state: if it is 0, the input value is correct; otherwise not.
Format: CALL TDIAL ("<phone number>") [SVI Modem BASIC]
Function: Calls a specific phone number via tone dialing. <phone number> must be in quotation marks and only the characters "0 123 456 789AaBbCcDd" are allowed.

TEMPER (declaration, MSX-Music and MSX-Audio)
Format: CALL TEMPER (<n>)
Function: Sets the battery mode for OPLL. <n> can range from 0 to 21, the meaning of which is as follows:

0 – Pythograph	11 – Pure Rhythm Cis + (B-)
1 – Mintone	12 – Pure rhythm D + (H-)
2 – Welkmeyster	13 – Pure rhythm Es + (C-)
3 – Welkmeyster (adjusted)	14 – Pure rhythm E + (Cis-)
4 – Welkmeyster (separate)	15 – Pure rhythm F + (D-)
5 – Kilanbuger	16 – Pure rhythm Fis + (Es-)
6 – Kilanbuger (adjusted)	17 – Pure rhythm G + (E-)
7 – Velotte Young	18 – Pure Rhythm Gis + (F-)
8 – Lamour	19 – Pure rhythm A + (Fis-)
9 – Perfect rhythm (default)	20 – Pure rhythm B- (G-)
10 – Pure rhythm C + (A-)	21 – Pure rhythm H- (Gis-)

TEMPO (statement, SFG-BASIC)
Format: CALL TEMPO (<time value>)
Function: Defines the "time" in quarter note units that will be played in one minute. It can range from 0 to 200, with 0 interrupting playback. Short version: _TEMP.

TERMINAL (command, New Modem BASIC)
Format: CALL TERMINAL (<numeric variable>)
Function: Allows you to communicate with a BBS. Almost all keys pressed are sent over the phone line and what comes from the phone line is displayed on the screen. This instruction can only be used in a Terminal program. <numeric variable> stores the state:

- 1 – The signal has dropped (usually broken connection).
- 5 – The automatic login character (from the BBS) was received.
- 11 – The HOME key was pressed to return to BASIC. It can be used to return to the Terminal menu.
- 220 – GRAPH+I was pressed to return to BASIC. They can be used to manually send the name and password to a BBS without automatic login.

TIMER (command, SFG-BASIC)
Format: CALL TIMER (<period> [,<brand number>])
Function: Starts and sets the timer period.
<period> is defined in units of 1/100 seconds and can vary from 1 to 24,000.
<brand number> can be any number between 1 and 254. If omitted, the number 11 will be used.

TRACE OFF (command, MSX Aid BASIC)
Format: CALL TRACE OFF
Function: Stops program execution tracking.

TRACE ON (command, MSX Aid BASIC)
Format: CALL TRACE ON
Function: It starts tracking the execution of the program in the same way as the TRON instruction, but whenever the execution skips to another line, it sends the number of the executed line to the printer.

TRACK (declaration, SFG-BASIC)
Format: CALL TRACK (<number of tracks>)
Function: Defines the number of tracks used by _PHRASE or _PLAY. <number of tracks> for varying from 1 to 8; if omitted, the value will be 1. Short version: _TRAC.

TRANSPOSE (declaration, MSX-Music and MSX-Audio, SFG-BASIC)
Format: CALL TRANSPOSE (<n>) [MSX Music /Audio]
Function: Changes the key. <n> can vary from –12 799 to +12 799, with 100 units corresponding to halftone. The default value is 0.
Format: CALL TRANSPOSE (<n>) [SFG-BASIC]
Function: Changes the key. <n> can range from –12 to +12 in halftone increments. The default value is 0.

TSTOP (command, SFG-BASIC)
Format: CALL TSTOP
Function: Stops the timer. The _INIT statement also interrupts the timer. Short version: _TSTO.

TUNE (command, SFG-BASIC)
Format: CALL TUNE (<numeric value>)
Function: Tunes the FM Tone Generation system with the other instruments. <numeric value> can range from −100 to +100, which corresponds to a semitone.

UPPER (function, DM-System2 BASIC)
Format: CALL UPPER (<variable>, <alphanumeric string>)
Function: Converts the alphabetic characters of the <alphanumeric string> to uppercase and returns it in the <variable>.

USBCD (remote, RookieDrive BASIC)
Format: CALL USBCD ("<directory>")
Function: Change the active directory on the USB device.

USBERROR (statement, RookieDrive BASIC)
Format: CALL USBERROR
Function: Displays the stored error code whenever a USB transaction fails for any reason. Only the error of the last executed USB transaction is stored.

USBFILES (remote, RookieDrive BASIC)
Format: CALL USBFILES
Function: Displays the list of disk images that are in the root directory of the USB virtual drive. The execution of this instruction takes the disk to the "offline" state. To return to the "online" state, use CALL INSERTDISK or CALL REBOOT.

USBRESET (remote, RookieDrive BASIC)
Format: CALL USBRESET
Function: Repeat the initialization procedure that is performed when a standard USB floppy drive is connected to a Rookie Drive interface.

USERHYTHM (statement, SFG-BASIC)
Format: CALL USERHYTHM
Function: Enables the rhythm instruments (drums) for use. These instruments use two FM voices; so the number of available voices drops from 8 to 6 with the rhythm enabled. Short version: _USER.

USR (command, Nextor)
Format: CALL USR (<execution address>, [<registers])
Function: Calls a routine in Assembler, optionally loading registers with specific values beforehand.
<execution address> is the starting address of the routine. If "–1" is specified, the routine will only return without error (useful for detecting Nextor in BASIC).
<registradores> is a pointer to a 12-byte buffer where the values of the registers are specified in the sequence "F, A, C, B, E, D, L, H, IXl, IXh, IYl, Iyh".

VARLIST (command, MSX Aid BASIC)
Format: CALL VARLIST [(["<variable>"] [, P])]
Function: Displays a list of all variables already used by the MSX-BASIC program that is in memory. If <variable> is specified (1 or 2 characters), line numbers with the variable in question will be listed. If the second character is an asterisk (*), all variables that start with the first character are considered. With parameter P, the data will be sent to the printer. Without any parameters, the complete list will be displayed on the screen.

VCOPY (declaration, DM-System2 BASIC)
Format: CALL VCOPY (<X0>, <Y0>) – (<X1>, <Y1>) [,<PgF>] TO (<X2>, <Y2> – <X3>, <Y3>) [,<PgD>] [,<R>] [ON (<X4>, <Y4>)] [,<logical operator>]
Function: Copies a rectangular area of VRAM to another with zoom in / out and rotation.
<X0> – X coordinate of the first point in the source area.
<Y0> – Y coordinate of the first point in the source area.
<X1> – X coordinate of the second point in the source area.
<Y1> – Y coordinate of the second point in the source area.
<PgF> – VRAM source page.
<X2> – X coord of the first corner of the destination area.
<Y2> – Y coord of the first corner of the destination area.
<X3> – X coord of the opposite corner of the target area.
<Y3> – Y coord of the opposite corner of the target area.
<PgD> – VRAM's landing page.

<R> – Clockwise rotation in degrees.
<X4> – X coordinate of the rotation axis (X2 is standard).
<Y4> – Y coordinate of the rotation axis (Y2 is standard).
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.
<LO> is the logical operator and can be [T]PSET, [T]PRESET, [T]XOR, [T]OR or [T]AND. The default is PSET.

VDPWAIT (command, DM-System2 BASIC)
Format: CALL VDPWAIT
Function: Wait until the VDP finishes executing the command.

SEE (statement, Hangul-BASIC 4)
Format: CALL VER
Function: Displays the version of Hangul-BASIC.

VIDEO (function, Pioneer-BASIC)
Format: CALL VIDEO (<variable>)
Function: Returns in the <variable> the type of video selection currently active. The returned value can be:
0 – Computer screen (internal synchronization)
1 – Superimpose
2 – External video

VLIST (declaration, SFG-BASIC)
Format: CALL VLIST
Function: Displays the instrument table on the screen.

VMOFF (control, DM-System2 BASIC)
Format: CALL VMOFF [(<segment number>)]
Function: Aborts the macro operation.

VMON (command, DM-System2 BASIC)
Format: CALL VMON (<start address>, [<start value>])
Function: VDP processing macro operation.
<start address> specifies the start of the macro code.
<initial value>, if specified, causes the macro operation to start only after being stored in the VDP macro variable.

VMWAIT (command, DM-System2 BASIC)
Format: CALL VMWAIT
Function: Puts the system on hold until the VDP macro operation is complete. CTRL+STOP can be used to exit this command.

VOICE (declaration, MSX-Music and MSX-Audio)
Format: CALL VOICE ([@ <n1>], [@ <n2>], ... [@ <n9>])
Function: Specifies the instruments to be used in each voice. <nx> can range from 0 to 63. The default value is 0.

VOICE COPY (statement, MSX-Music and MSX-Audio)
Format: CALL VOICE COPY (@<n1>, – <n2>)
Function: Copies data related to the instruments to / from a matrix variable type DIM A%(16). <n1> is the source and <n2> the destination. <n1> can range from 0 to 63 and <n2> can only be 63, or <n1> and <n2> can be a matrix variable.

WAIT (command, DM-System2 BASIC, SFG-BASIC)
Format: CALL WAIT (<time>) [DM-System2 BASIC]
Function: Wait for a defined time. It can be aborted by CTRL+STOP. <time> is defined in 1/60 second units and can range from 0 to 32767.
Format: CALL WAIT (<event number>) [SFG-BASIC]
Function: Suspend the interruption when the melody is being played. <event number> can be:
1~4 – Suspend while playing the respective instrument.
5 – Suspend during rhythm playback.
6 – Suspend until the timer time is zero.

WHO (statement, Network BASIC)
Format: CALL WHO (<micro number>)
Function: Returns the number of the computer in the MSX network. <micro number> can range from 0 to 15, where 0 is the teacher's micro.

XREF (statement, MSX Aid BASIC)
Format: CALL XREF [([<line number>] [, P])]

Function: Displays a list with all the linked lines of an MSX-BASIC program that is in memory (GOSUB, GOTO, RESUME, RESTORE, RETURN instructions). <line number> is used to limit the list to a specified line number. With parameter P, the data will be sent to the printer. Without any parameters, the complete list will be displayed on the screen.

XY (command, DM-System2 BASIC)
Format: CALL XY (<X coordinate>, <Y coordinate>)
Function: Changes the coordinates of the graphic accumulator.

YMMM (declaration, DM-System2 BASIC)
Format: CALL YMMM (<X0>, <Y0>) – [STEP] (<X1>, <Y1>) TO (<X2>, <Y2>)
Function: Executes the YMMM command (quick copy in bytes in the Y direction) of the VDP. Available for Screens 5 to 12.
<X0> – X coordinate of the first point in the source area.
<Y0> – Y coordinate of the first point in the source area.
<X1> – X coordinate of the second point in the source area.
<Y1> – Y coordinate of the second point in the source area.
<X2> – Left X coordinate of the target area.
<Y2> – Upper Y coordinate of the target area.
STEP, if specified, indicates relative coordinates.
Note: <X> can vary from 0 to 511 and <Y> from 0 to 1023.

3.4 – MSX-BASIC ERROR CODES

01 NEXT without FOR
02 Syntax error
03 RETURN without GOSUB
04 Out of DATA
05 Illegal function call
06 Overflow
07 Out of memory
08 Undefined line number
09 Subscript out of range
10 Redimensioned array
11 Division by zero

12 Illegal direct
13 Type mismatch
14 Out of string space
15 String too long
16 String formula too complex
17 Can't CONTINUE
18 Undefined user function
19 Device I/O error
20 Verify error
21 No RESUME
22 RESUME without error
23 Unprintable error
24 Missing operand
25 Line buffer overflow
26~49 Unprintable error
50 FIELD overflow
51 Internal error
52 Bad file number
53 File not found
54 File already open
55 Input past end
56 Bad filename
57 Direct statement in file
58 Sequential I/O only
59 File not OPEN
60 Bad FAT
61 Bad file mode
62 Bad drive name
63 Bad sector
64 File still open
65 File already exists
66 Disk full
67 Too many files
68 Disk write protected
69 Disk I/O error
70 Disk offline
71 RENAME across disk
72 File write protected

73 Directory already exists
74 Directory not found
75 RAM disk already exists
76 Invalid device driver *
77 Invalid device or LUN *
78 Invalid partition number *
79 Partition already in use *
80~255 Unprintable error

Obs. The codes marked with "*" (76 a 79) are for Nextor only.

4 – MSXDOS

COMMAND NAME (command type, COMMAND version)
  Format: Valid formats for the command
  Function: Command operation mode

Internal commands are commands executed directly by COMMAND.COM, and external commands are loaded from the disk.

The COMMAND version indicates the version for which the command is implemented. Values separated by "–" indicate that there are differences in syntax or behavior for different versions. Next there is a short description of the versions.

1 – MSXDOS version 1.0
2 – MSXDOS version 2.0 (Command up to version 2.3)
2.41 – MSXDOS version 2.0 (Command version 2.41)
K – Kanji-ROM required

4.1 – FORMAT NOTATION

<filename> – Filename in the form: A:\ dir1 \ dir2 \ file.ext
<compound filename> – Multiple filenames in the above format
<path> – Path in the form: A:\ dir1 \ dir2 \
[ ] delimits optional parameter.
 | it means that only one of the items can be used (OR).
{ } delimits option.

Chars in parentheses after some options for some commands indicate the version of COMMAND for which that option is available.

A <device> can be:

CON	Console (Keyboard)
CRT	Video
PRN	Printer
NULL	Null
AUX	Auxiliary
COM	Serial port

Or whatever is installed.

4.1.1 – Description of filenames extensions

ACC Music Creator accompaniment data files

ARC File(s) compressed in ARC format by System Enhancement Associates (SEA). Tools to extract are UNARC.COM (v1.6) and UNP.COM (v1.0 by Pierre Gielen).

ARC File(s) compressed in Russian ARC format, incompatible with SEA's ARC format. Tools to extract are XARC.COM (v1.01) and ARCDE.COM (v1.03).

ARJ File(s) compressed in ARJ format. Tool to extract are UNARC.COM (v1.10) and UNP.COM (v1.0 by Pierre Gielen).

APT Studio FM pattern data file.

ASC Plain text (ASCII format) that can contain a BASIC program or data.

ASM Assembler text file.

ASN Assignment files for MIDI Blaster

BAS BASIC program listing tokenized. These files can be executed from MSX-DOS with the BASIC name.bas command.

BAT Batch files (plain text) interpreted by MSX-DOS.

BGM MuSICA binary music file. MuSICA is a software developed by ASCII to create music on 17 voices with PSG, FM and Konami's SCC.

BGM MSX-FAN music file. Not to be confused with MuSICA files. Songs in this format were contained in all their disk magazines, and later a specific player was released that even supported playback on MIDI.

BGM Bloadable MSX-MUSIC file created by the BIT2BGM.COM utility of Uwe Schröder that converts Synth Saurus musical files.

BIN Binary file created with the BASIC BSAVE instruction. Loads with BLOAD. The header have a length of 7 bytes (FEh + Start address + End address + execution address). It can contain machine language and data.

BMP Image file in format . Viewable in SCREEN 7 or 8 with BMP.COM (v1.01 by SEIGA).

BOK MSX View Picture book.

BTM Batch files supported by MSX-DOS 2 v.2.40 or later.

CAS BIOS level cassette image for emulators, needs a separate tool to run or write to cassette. SofaCas allows to convert software on tape to CAS file and also play it using a homemade cable PC sound output to MSX cassette input. On MSX turbo R, we can use TRCAS (by Martos) to run CAS played by SofaCas.

CMP Compressed screen 5 image, including palette, created with DD-Graph (Dot Designer's Graph) (aka DD-Graph).

CMP Compressed image, including palette, created with GIOS. GIOS, aka Graphical Input/Output System

COM Command containing a binary executable under MSX-DOS. Can be also an executable file compressed with POPCOM.COM (v1.0 by Perpermint-Star).

CPM .COM file renamed to .CPM, either to be used in some CPM emulators, or to be able to workaround GMail's nanny protection against executable files. Just rename those back to .COM to be able to run them.

DRM File for the drums editor of the First Rate Music Hall tracker.

DAT Synthesizer configuration file for MIDI Blaster

DSK Disk image for emulators, needs a separate tool to run or write to normal disk. Can be launched on real MSX with SofaRunit or using Nextor's EMUFILE command.

DUA Music-BOX dual data file (melody + sample)

EDI File for the song editor of the First Rate Music Hall tracker.

EMx Disk image for the floppy disk emulator (HDDEMU.COM) for MSX Turbo R by Tsuyoshi. Internal structure is same as in DSK-files. Protected disks have additional information stored to files with HED-extension.

EVA	Video file in EVA format.
EVG	Yamaha SFG-05 event data file.
FM	MSX-MUSIC BASIC file.
FMP	MSX-MUSIC BASIC file.
FMS	Synth Saurus sound file.
FNT	Font file for the Scroll Power utility.
FON	MSX View font.
G9B	Library graphic format for GFX-9000.
GE5	Synonym for .SC5. See the .SCx file description.
GEN	Plain text that contain Z80 assembly source code, used with GEN80 compiler.
GIF	Graphics Interchange Format. They can be viewed with GIFI.COM (by Kakami Hiroyuki) and converted in MSX format with ENGIF.COM (v1.2 by Pierre Gielen), SHOWEM.COM (by Steven van Loef) and GIFDUMP.COM (by Francesco Duranti)
GLx	Graph Saurus image file like BASIC instruction COPY. Can be used under BASIC.
GRA	Image file in QLD format. The viewer BLS.COM (v2.00 by SEIGA) support it.
GRP	Synonym for .SC2. See the .SCx file description. Can also be a compressed image for Graph Saurus.
GZ	File compressed in GZIP format by PC gzipers. Tool to extract is GUNZIP.COM.
HLP	MSX-DOS 2 help file (plain text).
INS	File for the instruments editor of the First Rate Music Hall tracker.
IPS	Patch for file. Needs IPS patcher.
ISH	Compressed file.

JPG Compressed image file in format JPEG. Some viewer can show image until 1024x1024: JPD.COM (v0.23 by APi), JLD.COM (v1.11 by SEIGA) or BLS.COM (v2.00 by SEIGA). JPEG file can be produced on MSX from SCREEN 12 images with JSV.COM (v0.1 by SEIGA).

KSS MSX music file that contains also player code. Use KSS-PLAY.COM (by NYYRIKKI) to play it.

LDR Tokenized Basic file usually BASIC program LoaDeR used to load and run a program consisting of several BAS files.

LHA File(s) compressed in LHA format. Tools to create a LHA archive are LHPACK.COM (v1.03 by H.Saito) or LHA.COM (v1.05a by Kyouju). Tools to extract are PMM.COM (v1.20 by Iita), LHARC.COM and LHEXT.COM (v1.33 by Kyouju).

LPF Loop file for the Scroll Power utility.

LZH Synonym for LHA.

MAG Maki-chan V2 image file maintly used on PC-9801 and Sharp X68 000. Viewable with BLS.COM (v2.00 by SEIGA)

MAX Synonym for MAG.

MBK Sample kit file for the music tracker MoonBlaster.

MBM Music file for the music tracker MoonBlaster.

MBS Sample file for the music tracker MoonBlaster.

MBV Voice file for the music tracker MoonBlaster.

MBW Wave song for the music tracker MoonBlaster.

MCM Micro Cabin music file. Played by MCDRV.EXE.

MDT MSX Music-System music data file.

MDX Music file in a format designed for Sharp X68 000. These files can be played by MPX2.COM (when driver installed with MXDRV.COM). Optional PDX files are PCM samples. Require the YAMAHA SFG-01/05 cartridge or the MFP PCM cartridge.

MEG Plain text that contain Z80 assembly MegaAssembler source code. Extension also used for Mega-Rom images.

MEL Music-BOX melody data file.

MFM FM song for MoonBlaster.

MGS Music file in format developed by AIN. Played by MGSEL.COM (when driver installed with MGSDRV.COM).

MID Standard MIDI file (can be played using MIDI-interface or MoonSound software)

MIF Compressed image file (MSX Image Format). Can be viewed with MIFVIEW.COM.

MIO MIODRV Music file. Played by MIODRV Player.

MKI Maki-chan V1 image file maintly used on Sharp X68 000. Viewable with BLS.COM (v2.00 by SEIGA).

MOD Amiga MOD file. Can be played on MSX turbo R or MoonSound.

MP3 MPEG Audio Layer III file. MP3s can be played with Sunrise MP3 player, MPX Cartridge r1.1 by Junsoft or SE-ONE by TMT logic.

MPK Music Player K-kaz song. Require WAMPK Player.

MSx Synth Saurus score file.

MSD MuSICA source music file (MML). We can also use KINROU4 (by Masarun), an alternative compiler.

MUE HAL Music Editor MUE music file. There's a patch to add mouse support here.

MUS FAC Soundtracker music file.

MUS MGSDRV source MML file. Needs to be compiled to a MGS file with MGSC.COM. OTOH, MGSCR.COM can decompile MGS files back to the MUS source.

MUS Studio FM music file (not recommended).

MWK MoonSound Wave sample kit.

MWM MoonBlaster for MoonSound Wave song.

OPX OPLL driver music format.

PAC Dump of SRAM contents (save games) of PAC or FM-PAC cartridge.

PAT Studio FM pattern file.

PCM Sound sample file for MSX-Turbo R.

PCK Packaged file for First Rate Music Hall tracker. Includes 4 songs with all instruments and drums data.

PCT Dynamic Publisher page files.

PDX Optional PCM sample file used with an MDX file. You can play a PDX with PDXLOAD.COM by AIN. See also MDX extension.

PIC Phillips Video Graphics image. Synonym for . SC8, so check the .SCx description. Also specific image format used of X68 000, it is viewable with BLS.COM (v2.00 by SEIGA)

PLx Graph Saurus colors palette file in Raw format (contains 8 sets of palettes with two bytes by color RG 0B). It's a companion for the respective .SRx file, so both files must always be copied together.

PMA File(s) compressed in PMARC format. Tools to create an archive are PMARC.COM, PMARC2.COM (v2.0 by Sybex) and UNP.COM (v1.0 by Pierre Gielen). Tools to extract are PMM.COM (v1.20 by Iita), PMEXE.COM (v2.0) and PMEXT.COM (v2.22). PMEXT has been ported on Windows (v1.21 by Yoshihiko Mino). To extract a PMA file on a Mac use The Unarchiver.

PRO Music file for Pro-Tracker (by Tyfoon Soft).

PSG PSG Sampler sample file.

RDT MSX Music-System rhythm data file.

RLT Music Creator real time data files.

ROM Raw ROM image dump. Used by ROM loaders or emulators.

RTM Synth Saurus rhythm file.

S1x Contains the odd lines of an interlace image. For more info, see the SCx file.

S3M See MOD file.

SAM Music-BOX sample data file (used as drumkit file in Music Creator)

SBM Music data for SCC and PSG soundchips.

SBS Instruments data for SCC and PSG soundchips.

SCx Screen-x binary image file. Can have a companion .S1x file that will contain the extra interlaced lines to double the vertical resolution. Used by image editors or BLOAD instruction with the parameter S. SC2 images can be viewed under MSX-DOS with SC2VIEW.COM (by GDX), and SC5 to SCC files with BLS.COM (by SEIGA).

SCR Screen-2 image created with Graphos III. It's an executable file with a loader that produces an effect. Loadable on MSX-BASIC with BLOAD"file",R.

SDT MSX Music-System sound data file (= voice data)

SDT SCMD Music file for MSX made a MML compiler for Windows. The player is SC.COM.

SEE Sound Effect data, for use in Sound Effect Editor (Shareware by Fuzzy Logic)

SEQ Music Creator sequence data files.

SFM Studio FM music file.

SMx FAC Soundtracker sample file.

SMP Sample file for Covox/SIMPL or MSX Turbo R.

SNG Music file for the music editor SCC-Musixx by Tyfoon Soft.

SPT	Music Creator step time data files.
SPT	Text file for the Scroll Power utility.
SRx	Graph Saurus Image file. Requires the respective .PLx file. Can be optionally compressed with run-lenght. Uncompressed files can be loaded on MSX-BASIC with a BLOAD"FILE.SRx",S, but the external palette will have to be loaded with OPEN#1.
STP	Dynamic Publisher stamp files. Contains an image that can be loaded on a page (.STP).
TIx	Graph Saurus tile file.
TSR	Terminate and Stay Resident programs to be used with MemMan 2.0 and higher.
TXT	Plain text file generaly coded in ASCII, Ank or JIS.
VCD	Voice file for the MSX Voice Recorder (HAL Laboratory).
VCD	MuSICA voice file.
VGM	Music file that supports many sound chips, playable by VGMPLAY.COM (by Laurens Holst)
VOC	Music Creator voice data files.
VOC	Studio FM voice data file.
VOG	Yamaha SFG-05 voice data file.
WAV	Sound sample file. Can be played with the MPX Cartridge r1.1 by Junsoft.
WB	Assembler Project file. For the The WBASS2 Z80 Assembler.
XM	See MOD file.
XPC	ROM Patch file for EXECROM.COM (A&L Software)
ZIP	File(s) compressed in ZIP format by PC zipers. The best tool to extract is SUZ.COM (v1.3 by Loutrax).

4.2 – DESCRIPTION OF COMMANDS

ALIAS (internal, 2.41)
Format: ALIAS [/P] [name] [=] [value] | /R | {/L | /S} <filename>
Function: Displays or sets the alias command.
[/P] Pauses the listing when completing a screen.
[/R] Removes all defined aliases.
[/L] Loads an alias defined in <filename>.
[/S] Saves the current alias to file <filename>.
[name] is the name of the new command.
[value] is the command or string that will be assigned to [name]
<filename> is the file on disk to which it will be written or where the defined alias will be retrieved.

ASSIGN (internal, 2)
Format: ASSIGN [d1: [d2:]]
Function: Redirects access to drive d1: to drive d2:.

ATDIR (internal, 2)
Format: ATDIR + | –H [/H] [/P] <filename>
Function: Enables / disables hidden directory attributes.
[/P] pauses error messages when completing a screen.
+H marks file as hidden.
–H turns off the hidden file attribute, and must be followed by /H.

ATTRIB (internal, 2-2.41)
Format: ATTRIB {+ | –H | + | –R | + | –S | + | –A} [/H] [/P] <filename>
Function: Change attributes of hidden file (H) read-only (R), system file (S, 24.1 only) or archived (A, 2.41 only). "–H" must be used with "/H".
[/P] pauses error messages when completing a screen.

BASIC (internal, 1)
Format: BASIC [<prog name>]
Function: It transfers control to the BASIC interpreter and optionally loads and executes the program <prog name>.

BEEP (internal, 2.41)
Format: BEEP
Function: Generates a beep.

BOOT (internal, 2.41)
Format: BOOT [drive]
Function: Exchanges the MSXDOS boot drive from BASIC.

BUFFERS (internal, 2)
Format: BUFFERS [number]
Function: Displays or sets the number of system I/O buffers.

CD (internal, 2)
Format: CD [[d:] [path] | –]
CHDIR [[d:] [path] | –]
Function: Display or change the current subdirectory. If "–" is specified, returns to the previous directory.

CDD (internal, 2.41)
Format: CDD [[d:] [path] | –]
Function: Displays or changes the current subdirectory and drive. If "–" is specified, it returns to the previous drive / directory.

CDPATH (internal, 2.41)
Format: CDPATH [[+ | –] [d:] path [[d:] path ...]]]
Function: Displays or sets the search path.

CHDIR (internal, 2)
Format: The same as the CD command.
Function: The same as the CD command.

CHKDSK (internal, 2)
Format: CHKDSK [d:] [/F]
Function: Checks the integrity of the files on the disk. If [/F] is specified, files will not be corrected; only information about the integrity fault will be shown.

CLS (internal, 2)
Format: CLS
Function: Clears the screen.

COLOR (internal, 2.41)
Format: COLOR <front color> [<backgrd color> [<border color>]]
Function: Change the colors of the screen.

COMMAND2 (internal, 2)
Format: COMMAND2 [command]
Function: Executes a MSXDOS2 command.

CONCAT (internal, 2-2.41)
Format: CONCAT [/H] [/S] [/P] [/A] [/B] [/V] <source files> <destination files>
Function: Concatenates all source files into a single file.
[/H] Hidden files will also be concatenated.
[/S] System files will also be concatenated (only 2.41).
[/P] Pause messages when completing a screen.
[/B] Concatenates without interpretation.
[/A] Reverses the effect of [/B].
[/V] Check concatenated file created.

COPY (internal, 1-2-2.41)
Format: COPY [/H] [/S] [/P] [/A] [/B] [/V] [/T] <source files> <destination files>
Function: Copies files.
[/H] Hidden files will also be copied (2).
[/S] System files will also be copied (2.41).
[/P] Pause messages when completing a screen.
[/A] Makes an ASCII copy (adds Ctrl+Z to the end of the file).
[/B] Reverses the effect of [/A].
[/V] Checks copied file.
[/T] Changes the date and time of the copied file to the current one.

CPU (internal, 2.41)
Format: CPU [number]
Function: Display or change the CPU for the MSX turbo R (0 = Z80; 1 = R800 ROM; 2 = R800 DRAM).

DATE (internal, 1-2.41)
Format: DATE [date]
Function: Displays or changes the system date. [date] must be in format "mm–dd–yyyy" or in format defined by SET DATE.

DEL (internal, 1)
Format: DEL [/S] [/H] [/P] <compound filename>
ERA [/S] [/H] [/P] <compound filename>
ERASE [/S] [/H] [/P] <compound filename>
Function: Delete one or more files.
[/S] System files will also be deleted (2.41).
[/H] Hidden files will also be deleted.
[/P] Pause messages when completing a screen.

DELALL (external, Nextor)
Format: DELALL <drive letter>:
Function: Quick format for a drive unit.

DEVINFO (external, Nextor)
Format: DEVINFO <driver slot> – [<driver subslot>]
Function: Displays information about devices controlled by Nextor.

DIR (internal, 1-2-2.41)
Format: DIR [/S] [/H] [/W] [/P] [/2] [<compound filename>]
Function: Displays the filenames of the disk.
[/S] System files will also be listed (2.41)
[/H] Hidden files will also be listed
[/W] List filenames only
[/P] Pauses the listing when completing a screen
[/2] List in two columns (2.41)

DISKCOPY (external, 2)
Format: DISKCOPY [d1: [d2:]] [/X]
Function: Copies an entire disk (d1:) to another (d2:)
[/X] Suppress messages during copying.

DRIVERS (external, Nextor)
Format: DRIVERS
Function: Displays information about the drivers available for Nextor and MSXDOS.

DRVINFO (external, Nextor)
Format: DRVINFO
Function: Displays information about all available drive letters.

DSKCHK (internal, 2.41)
Format: DSKCHK [ON | OFF]
Function: Displays or sets the check status of the disc.

ECHO (internal, 1)
Format: ECHO [text]
Function: Prints a text while executing a batch file with line feed at the end.

ECHOS (internal, 1)
Format: ECHOS [text]
Function: Prints a text during the execution of a batch file without line feed at the end.

ELSE (internal, 2.41)
Format: ELSE [command]
Function: Conditional command execution. Without the [command] parameter, toggle Command Mode between ON/OFF.

END (internal, 2.41)
Format: END
Function: Ends a batch file (batch).

ENDIFF (internal, 2.41)
Format: ENDIFF [command]
Function: Increase a level and restore Command Mode.

ERA (internal, 1)
Format: The same as the DEL command.
Function: The same as the DEL command.

ERASE (internal, 1)
Format: The same as the DEL command.
Function: The same as the DEL command.

EXIT (internal, 2)
Format: EXIT [number]
Function: Exits the program executed by the COMMAND2 command. [number] is the user's error code (the default is 0).

FASTOUT (external, Nextor)
Format: FASTOUT [ON | OFF]
Function: Turns on/off the quick output for the STROUT routine, or displays the current STROUT status.

FIXDISK (external, 2)
Format: FIXDISK [d:] [/S]
Function: Updates a disk to the MSXDOS2 format.
[/S] Update complete.

FORMAT (internal, 1-2.41)
Format: FORMAT [d:] (1)
FORMAT [d: [option [/X]]] (2.41)
Function: Formats a disc. If [option] is specified, it formats with that option, without displaying a list of options.
[/X] Starts immediate formatting, without displaying a message.

FREE (internal, 2.41)
Format: FREE [d:]
Function: Displays the total, free and used space of the disk.

GOSUB (internal, 2.41)
Format: GOSUB~label
Function: Executes a subroutine within a batch file.

GOTO (internal, 2.41)
Format: GOTO~label
Function: Jump to the label within a batch file.

HELP (internal, 2)
Format: HELP [<filename>]
Function: Displays the help file <filename>.HLP or lists all if there is no argument.

HISTORY (internal, 2.41)
Format: HISTORY [/P]
Function: Displays the command history.
[/P] pauses history when completing a screen.

IF (internal, 2.41)

Format: IF [NOT] EXIST [d:] [<path>] <filename> [THEN] <command>
or
IF [NOT] <expr1> == | EQ | LT | GT <expr2> [AND | OR |
XOR [NOT] <expr3> == | EQ | LT | GT <expr4>
[AND | OR | XOR ...]] [THEN] <command>

Function: Executes a command if the given equation is true.
EQ → Equivalence (equality)
LT → Less than
GT → Greater than

IFF (internal, 2.41)

Format: IFF [NOT] EXIST [d:] [<path>] <filename> [THEN] <command>
⋮
ENDIFF [<command>]
or
IFF [NOT] <expr1> == | EQ | LT | GT <expr2> [AND | OR |
XOR [NOT] <expr3> == | EQ | LT | GT <expr4>
[AND | OR | XOR ...]] [THEN] <command>
⋮
ENDIFF [<command>]

Function: Turn on Command Mode if the given equation is true and turn off otherwise.
EQ → Equivalence (equality)
LT → Less than
GT → Greater than

INKEY (internal, 2.41)

Format: INKEY [<string>] %%<environment variable>

Function: Reads the value of a key pressed and stores the value read in the <environment variable>.

INPUT (internal, 2.41)

Format: INPUT [<string>] %%<environment variable>

Function: Reads a string from the keyboard or device and stores the value read in the <environment variable>.

KMODE (external, 2-K)

Format: KMODE [mode | OFF] [/S] [d:]

Function: Select or turn off the Kanji mode or update the boot to automatically install the Kanji driver.
[/S] Updates the boot code for the [d:] drive.

LOCK (external, Nextor)
Format: LOCK [<drive letter>: [ON | OFF]]
Function: Lock or unlock drive letters, or display the list of locked drives.

MAPDRV (external, Nextor)
Format: MAPDRV [/L] <drive>: <partition> | d | u [<disp index> – <LUN index>] [<driver slot> [– <subslot of the driver>]]]
Function: Maps a drive unit in the Nextor system.
[/L] Locks the unit right after mapping
<drive> drive letter to be mapped
<partition>: 0 – the drive will be mapped from the device's absolute zero sector.
1 – First primary partition
2 to 4 – Refer to extended partitions 2.1 to 2.4 if partition 2 is extended; otherwise, they refer to primary partitions.
5 or more refer to extended partitions.
d – The default unit will be mapped
u – The drive will not be mapped

MD (internal, 2)
Format: MD [d:] <path>
MKDIR [d:] <path>
Function: Create a subdirectory

MEMORY (internal, 2.41)
Format: MEMORY [/K] [/P]
Function: Displays information about the system's RAM.
[/K] Displays in Kbytes.
[/P] Pause messages when completing a screen.

MKDIR (internal, 2)
Format: The same as the MD command.
Function: The same as the MD command.

MODE (internal, 1-2.41)
Format: MODE <number of characters> [<lines>]
Function: Changes the number of characters per horizontal line (1, 2 and 2.41) and the number of screen lines (only 2.41).

MORE (external, 1-2.41)
Format: <command> | MORE
Function: Display command. The output of the <command> is redirected to the MORE command. At the end of the screen, the display is paused with the message MORE until a key is pressed. <ESC> or <N> abort the command.

MOVE (internal, 2)
Format: MOVE [/H] [/P] [/S] <filename> <path>
Function: Moves files to another part of the disk.
[/H] Hidden files will also be moved.
[/S] System files will also be moved (2.41).
[/P] Pause messages when completing a screen.

MVDIR (internal, 2)
Format: MVDIR [/H] [/P] <filename> <path>
Function: Moves directories to another part of the disk.
[/H] Hidden directories will also be moved.
[/P] Pause messages when completing a screen.

NSYSVER (external, Nextor)
Format: NSYSVER <major version>. <Minor version>
Function: Changes the version number of DOS returned by the system.

PATH (internal, 2)
Format: PATH [[+ | −] [d:] <path> [[d:] <path> ...]]]
Function: Displays or sets the search path for .COM and .BAT execution files.
+ Delete paths with the same name and recreate them
− Delete the specified paths
Note: without +/−, delete all existing paths and create the specified path.

PAUSE (internal, 2)
Format: PAUSE [comment]
Function: Stops the execution of a batch file (batch) until a key is pressed.

POPD (internal, 2.41)
Format: POPD [/N]
Function: Retrieves the current drive and directory.
[/N] only the last drive/directory are removed from the list.

PUSHD (internal, 2.41)
Format: PUSHD [d:] [<path>]
Function: Change the default directory and drive, saving the chains.

RAMDISK (internal, 2)
Format: RAMDISK [=] [<size> [K]] [/D]
Function: Displays the size or creates a RAMDISK.
[/D] Delete the existing RAMDISK and create another one.

RALLOC (external, Nextor)
Format: RALLOC [<drive letter>: [ON | OFF]]
Function: Enables or disables the reduction of allocation space for a drive unit, or displays the list of drives with reduced allocation.

RD (internal, 2)
Format: RD [/H] [/P] <filename>
RMDIR [/H] [/P] <filename>
Function: Removes one or more subdirectories.
[/H] Hidden files will also be moved
[/P] Pause messages when completing a screen

REM (internal, 1)
Format: REM [comments]
Function: Insert comments in a batch file.

REN (internal, 1-2.41)
Format: REN [/H] [/P] [/S] <filename 1> <filename 2>
RENAME [/H] [/P] [/S] <filename 1> <filename 2>
Function: Rename the file <filename 1> to <filename 2>.
[/H] Hidden files will also be renamed
[/S] System files will also be renamed (2.41)
[/P] Pause messages when completing a screen

RENAME (internal, 1)
Format: The same as the REN command.
Function: The same as the REN command.

RESET (internal, 2.41)
Format: RESET
Function: Reset the system.

RETURN (internal, 2.41)
Format: RETURN [~label]
Function: Returns from a subroutine in a batch file.

RMDIR (internal, 2)
Format: The same as the RD command.
Function: The same as the RD command.

RNDIR (internal, 2)
Format: RNDIR [/H] [/P] <directory name 1> <directory name 2>
Function: Renames the subdirectory <directory name 1> with <directory name 2>.
[/H] Hidden files will also be renamed.
[/P] Pause messages when completing a screen.

SET (internal, 2-2.41)
Format: SET [/P] [name] [=] [value]
Function: Defines or displays environment items.
[/P] Pause messages when completing a screen.
The default values are as follows:
EXPAND = ON (2.41)
SEPAR = ON (2.41)
ALIAS = ON (2.41)
REDE = ON
LOWER = ON (2.41)
UPPER = OFF
ECHO = OFF
EXPERT = ON (2.41)
PROMPT =% _CWD%> (modified in 2.41)
CDPATH =; (2.41)
PATH =;

TIME = 24
DATE = yy-mm-dd
TEMP = A:\
HELP = A:\ HELP
SHELL = A:\ COMMAND2.COM

SHIFT (internal, 2.41)
Format: SHIFT [/<number>]
Function: Shifts the arguments of the batch file one position to the left. If /<number> is specified, the argument in this position will be the first to be moved; previous arguments will not be affected.

THEN (internal, 2.41)
Format: THEN [<command>]
Function: Executes a command (THEN is ignored).

TIME (internal, 1)
Format: TIME [<time>]
Function: Displays or changes the system time.

TO (internal, 2.41)
Format: TO <part_name_subdirectory> [/N] [/X | F | P | L]
TO [d:] /S [/H]
TO [d:] ...
TO [d:] −n
TO [d:] \
TO [d:] <directory_name> /M | /C [/H]
TO [d:] <directory_name> /D
TO [d:] <old name> <new name> /R
TO [d:] <source_dir> <dest_dir> /V
Function: Change, create, delete, rename or remove a directory.
[/N] Lists the directories containing <part_name_subdir>.
[/C] Create a new directory and enter it.
[/D] Remove directory.
[/F] Searches only at the beginning of the name.
[/H] Makes /S also search for hidden files
or /M or /C create a hidden directory.
[/L] Searches only at the end of the name.

[/M] Create a new directory.
[/P] Searches for the entire name.
[/R] Rename first directory.
[/S] Searches all directories and creates the TO.LST file.
[/V] Moves subdirectory.
[/X] Only exact names are searched.
–n Level of subdirectories.
\ Go to the root directory.

TREE (internal, 2.41)
Format: TREE [d:] [<path>] [/P] [/?]
Function: Displays the directory tree on the disk.
[/P] Pauses the listing when completing a screen.
[/?] Displays a help screen.

TYPE (internal, 1-2.41)
Format: TYPE [/S] [/H] [/P] [/B] <filename> |“>” <device>
TYPE <device> <filename>
Function: Displays data from a file / device or create a file from specified device. To end <device> to <filename> entry, press CTRL+Z and RETURN.
[/S] System files will also be displayed (only 2.41).
[/H] Hidden files will also be displayed.
[/P] Pauses the presentation when completing a screen.
[/B] Disables checking of control codes.

TYPEWW (external, 1-2.41)
Format: TYPEWW <filename> [/S] [/H] [/B]
Function: Displays data from a file. Unlike the TYPE command, <filename> cannot be ambiguous.
[/S] System files will also be displayed (only 2.41).
[/H] Hidden files will also be displayed.
[/P] Pauses the presentation when completing a screen.

UNDEL (external, 2)
Format: UNDEL [<filename>]
Function: Recovers deleted files.

SEE (internal, 2)
Format: SEE
Function: Displays the system version.

VERIFY (internal, 2)
Format: VERIFY [ON | OFF]
Function: Displays or changes the writing verification status.

VOL (internal, 2)
Format: VOL [d:] [<volume name>]
Function: Displays or changes the volume name of the disk.

XCOPY (external, 2)
Format: XCOPY [<source filename> [<dest filename>]] [/T] [/A] [/M] [/S] [/E] [/P] [/W] [/V]
Function: Copies files and directories. The options are:
[/T] Changes the date of the copied file to the current one
[/A] Only files with the "file" attribute set are copied.
[/M] Similar to /A, but the "file" attribute is reset after copying.
[/S] Subdirectories are also copied.
[/E] Makes /S create all subdirectories, even empty ones.
[/P] Pause after copying each file.
[/W] Pause after copying some files.
[/V] Checks copied files.

XDIR (external, 2)
Format: XDIR [<filename>] [/H]
Function: Lists all files in the current subdirectory, in a tree.
[/H] Hidden files will also be listed.

Z80MODE (external, Nextor)
Format: Z80MODE <slot driver> [– <subslot driver>]] [ON | OFF]
Function: Enables or disables Z80 access mode for the specified MSXDOS driver.

4.3 – BDOS CALLS

4.3.1 – I/O Handling

CONIN (01H)
Function: Keyboard input.
Setup: None.
Output: A – keyboard character code.
Note: Character input with wait and echo on screen. The following control sequences are checked by this routine:
CTRL+C ⟶ Return the system to the command level.
CTRL+P ⟶ Turn on echo for the printer. Anything written on the screen will be output to the printer.
CTRL+N ⟶ Turns off echo for the printer.
CTRL+S ⟶ Stops displaying characters until a key is pressed.

CONOUT (02H)
Function: Displays the character contained in the E-register on the screen. The control sequences described above are checked.
Input: E – Character code.
Output: None.

AUXIN (03H)
Function: External auxiliary device input (modem, for example). The four control sequences are checked.
Input: None.
Output: A – Auxiliary device character code.

AUXOUT (04H)
Function: Output to external device. This function checks the four control sequences.
Input: E – Code of the character to be sent.
Output: None.

LSTOUT (05H)
Function: Character output to printer. This function checks the four control sequences.
Input: E – Code of the character to be sent.
Output: None.

DIRIO (06H)

Function: String input or output. Does not support control characters, but checks all four control sequences.

Input: E – Character code to be printed on the screen.
If it is FFH, the character will be received.

Output: If E is FFH on input, the ASCII code of the key will return in A. If A returns 00H, no key was pressed.

DIRIN (07H)

Function: Reads a character from keyboard (with wait) and prints to screen. This function does not support control characters.

Input: None.

Output: A – ASCII code of the character.

INNOE (08H)

Function: Reads a character from the keyboard (with wait) but does not print to the screen. This function does not support control characters.

Input: None.

Output: A – ASCII code of the character.

STROUT (09H)

Function: String output. The 24H ASCII character ($) marks the end of the string and will not print to the screen. This function checks the four control sequences.

Input: DE – Starting address of the string to be sent.

Output: None.

BUFIN (0AH)

Function: String input. The reading of characters ends when the RETURN key is pressed. If the number of characters exceeds the maximum indicated by DE, these will be ignored and a "beep" will be emitted for each extra character. This function checks the four control sequences.

Input: DE must point to a buffer with the following structure:
DE+0 ⟶ number of characters to read.
DE+1 ⟶ number of characters actually read.
DE+2 onwards: codes of the characters read.

Output: The second byte of the buffer pointed to by DE contains the number of characters actually read and from the third byte onwards are the ASCII codes of the characters read.

CONST (0BH)
Function: Check keyboard status. This function checks the four control sequences.
Input: None.
Output: If any key was pressed, register A returns with FFH, otherwise, it returns with the value 00H.

4.3.2 - Definition and reading of parameters

TERM0 (00H)
Function: System reset. When this function is called under DOS, it will cause MSXDOS to reload. When called under DISK-BASIC, it will cause a full reset.
Input: None.
Output: None.

CPMVER (0CH)
Function: Reading the system version. In the case of MSX, it will always return the value 0022H, indicating compatibility with CP/M 2.2.
Input: None.
Output: HL – System version.

DSKRST (0DH)
Function: Disk reset. All buffers are cleared (FCB, DPB, etc.), the current drive will be A: and DTA will be 0080H.
Input: None.
Output: None.

SELDSK (0EH)
Function: Select current drive. The current drive number is stored in address 0004H.
Input: E – Drive number (A:=00H, B:=01H, etc.).
Output: A – Number of available drives (1 to 8).

LOGIN (18H)
Function: Reading of drives connected to the computer.
Input: None.
Output: HL – Connected drives
H ⟶ Always "00 000 000"
L ⟶ ı b7 ı b6 ı b5 ı b4 ı b3 ı b2 ı b1 ı b0 ı
drive: ı H: ı G: ı F: ı E: ı D: ı C: ı B: ı A: ı
The bit will contain 0 if the drive is not connected and 1 if it is. If B: = 1 and A: = 0 (b1=1 and b0=0), it means that there is just one physical drive connected as A: and B:

CURDRV (19H)
Function: Reading the current (current) drive.
Input: None.
Output: A – Current drive number (A:=00H; B:=01H, etc.).

SETDTA (1AH)
Function: Sets the address for data transfer.
Input: DE – Start address of DTA (Disk Transfer Address).
Output: None.
Note: At system reset, DTA is set to 0080H.

ALLOC (1BH)
Function: Reading information about the disk.
Input: E – Desired drive number (0=current; 1=A:; etc.).
Output: A = FFH ⟶ Drive specification is invalid; otherwise:
A – Number of logical sectors per cluster;
BC – Sector size in bytes (typically 512);
DE – Total number of clusters on disk;
HL – Number of free (unused) clusters;
IX – Starting address of DPB in RAM;
IY – Starting FAT address in RAM.

GDATE (2AH)
Function: Returns the system date.
Input: None.
Output: HL – Year (1980 to 2079);
D – Month (1=January, 2=February, etc.);
E – Day of the month (1 to 31)
A – Day of the week (0=Sunday, 1=Monday, etc.).

SDATE (2BH)
Function: Modify system date.
Input: HL – Year (1980 to 2079);
D – Month (1=January, 2=February, etc.);
E – Day of the month (1 to 31).
Output: A = 00H ⟶ Date specification is valid;
FFH ⟶ The specification is invalid.

GTIME (2CH)
Function: Returns system time.
Input: None.
Output: H – Hours;
L – Minutes;
D – Seconds;
E – Hundredths of a second.

TIME (2DH)
Function: Modify system time.
Input: H – Hours;
L – Minutes;
D – Seconds;
E – Hundredths of a second.
Output: A = 00H ⟶ The time specification is valid;
FFH ⟶ The specification is invalid.

VERIFY (2EH)
Function: Disk write check.
Input: E = 0 ⟶ Disables disk write verification mode.
E ≠ 0 ⟶ Enable disk write check.
Output: None.

4.3.3 – Absolute reading/writing of sectors

RDABS (2FH)
Function: Read logical sectors from disk. The sectors read are placed from the DTA.
Input: DE – Number of the first logical sector to read;
H – Number of sectors to read;
L – Drive number (0=A:, 1=B:, etc.).
Output: A = 0 ⟶ The reading was successful;
A ≠ 0 ⟶ Error code.

WRABS (30H)
Function: Writing of logical sectors to disk. The data to be written to disk will be read into RAM from DTA.
Input: DE – Number of the first logical sector to be written;
H – Number of sectors to write;
L – Drive number (0=A:, 1=B:, etc.).
Output: A = 0 ⟶ The writing was successful;
A ≠ 0 ⟶ Error code.

4.3.4 – Accessing files by using FCB

FOPEN (0FH)
Function: Open file (FCB).
Input: DE – Start address of an unopened FCB.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

FCLOSE (10H)
Function: Close file (FCB).
Input: DE – Start address of an open FCB.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

SFIRST (11H)
Function: Search for the first file. This function accepts wildcard characters (* and ?).
Input: DE – Start address of an unopened FCB.
Output: A = 0 ⟶ The file was found;
A ≠ 0 ⟶ The file was not found.

SNEXT (12H)
Function: Search for the next file. This function accepts wildcard characters (* and ?).
Input: None.
Output: A = 0 ⟶ The file was found;
A ≠ 0 ⟶ The file was not found.

FDEL (13H)
Function: Delete files. Wildcard characters (* and ?) can be used.
Input: DE – Start address of an open FCB.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

RDSEQ (14H)
Function: Sequential reading.
Input: DE – Start address of an open FCB.
Current block in FCB – Initial block for reading.
Current record in FCB – Initial record for reading.
Output: A = 0 ⟶ The reading was successful;
A ≠ 0 ⟶ Error code.

WRSEQ (15H)
Function: Sequential writing.
Input: DE – Start address of an open FCB.
Current block in FCB – Initial block for writing.
Current record in FCB – Initial record for writing.
Initial 128 bytes of DTA – Data to be written.
Output: A = 0 ⟶ The writing was successful;
A ≠ 0 ⟶ Error code.

FMAKE (16H)
Function: Create files.
Input: DE – Start address of an unopened FCB.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

FREN (17H)
Function: Rename files. The wildcard character "?" can be used to rename multiple files simultaneously.
Input: DE – FCB start address with the name of the file to be renamed. In the first position of the FCB, the drive number must be placed followed by the name of the file to be renamed. From the 18th byte (FCB+11H) to the 28th, the new filename must be entered.
Output: A = 0 ⟶ The renaming was successful;
A ≠ 0 ⟶ Error code.

RDRND (21H)
Function: Random reading. The read record will be placed in the area indicated by the DTA and has a fixed size of 128 bytes.
Input: DE – Start address of an open FCB.
Random register in FCB – number of the register to read.
Output: A = 0 ⟶ The reading was successful;
A ≠ 0 ⟶ Error code.

WRRND (22H)
Function: Random writing.
Input: DE – Start address of an open FCB.
Random register in FCB – Register number to be written.
128 bytes from DTA – Data to be written.
Output: A = 0 ⟶ The writing was successful;
A ≠ 0 ⟶ Error code.

FSIZE (23H)
Function: Read file size. The size returns in the first three bytes in the FCB's random file size field in 128-byte increments. So, if a file contains 1 to 128 bytes, the returned value will be 1; if it contains 129 to 256 bytes, the value is 2, and so on.
Input: DE – Start address of an open FCB.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

SETRND (24H)
Function: Set field of random record.
Input: DE – Start address of an open FCB.
Current block in FCB – Number of desired block.
Current FCB record – Number of desired record.
Output: The desired current register position, calculated from the register and block contained in the FCB, is placed in the random register field. The first three random record bytes contain valid values.

WRBLK (26H)
Function: Block random writing. The random record number is automatically incremented after writing, and its size can range from 1 to 65 535 bytes.

Input: DE – Start address of an open FCB.
HL – Number of records to be written.
DTA – Data to be written.
FCB record size – Record size to be written.
FCB random record – First record number to be written.
Output: A = 0 ⟶ The operation was successful;
A ≠ 0 ⟶ Error code.

RDBLK (27H)
Function: Block random access.
Input: DE – Start address of an open FCB.
HL – Number of records to read.
DTA – Starting address for the read data.
FCB record size – Record size to be read.
FCB random record – First record number to be read.
Output: A = 0 ⟶ The reading was successful;
A ≠ 0 ⟶ Error code.
HL – Number of records actually read if end of file is reached before all records are read.

WRZER (28H)
Function: Random writing with 00H bytes. This function is the same as 22H (WRRND), except that it fills the remaining records of the file with 00H bytes if the specified record is not the last one in the file.
Input: DE – Start address of an open FCB.
Random record in FCB – Record to be written.
128 bytes from DTA – Data to be written.
Output: A = 0 ⟶ The writing was successful;
A ≠ 0 ⟶ Error code.

4.3.5 – Functions added by MSXDOS2

DPARM (31H)
Function: Read parameters from disk.
Input: DE – Start address of a 32-byte buffer.
L – Drive number (0=current, 1=A:, etc).

Output: A – Error code (if it is 0, there was no error).
DE – Start address of the parameter buffer.

+0	Physical drive number (1=A:, etc.).
+1~2	Sector size in bytes (usually 512).
+3	Number of sectors per cluster.
+4~5	Number of reserved sectors.
+6	Number of FATs (usually 2).
+7~8	Number of directory entries.
+9~10	Total number of logical sectors.
+11	Disk ID.
+12	Number of sectors per FAT.
+13~14	First sector of the directory.
+15~16	First sector of data area.
+17~18	Faximum number of clusters.
+19	Dirty disk flag.
+20~23	Volume ID (–1 = No volume ID).
+24~31	Reserved (usually 0).

FFIRST (40H)
Function: Search for first entry in directory.
Input: DE – Initial address of the FIB or an ASCII string "drive/path/file", which may contain the wildcard characters "?" and "*".
HL – Starting address of filename (only when DE points to FIB).
B – Attributes to search (same as directory).
IX – Starting address of a new FIB.
Output: A – Error code (if it is 0, there was no error).
IX – Starting address of the new filled FIB.

FNEXT (41H)
Function: Searches for next directory entry. This function should only be used after the 40H function. It accepts the wildcard characters "?" and "*" set to 40H and searches for all files that have equal parts of their name specified by wildcard characters, one after another.
Input: IX – FIB start address.
Output: A – Error code (if it is 0, there was no error).
IX – Starting address of the new filles FIB.

FNEW (42H)

Function: Search for new entry.

Input: DE – Starting address of the FIB or an ASCII string "drive/path/file". If there are wildcard characters in the filename, they will be replaced with appropriate characters. If the "directory" bit is set on the input (register B), a subdirectory will be created. The other bits will be copied.
HL – Starting address of a filename (only if DE points to FIB).
B – b0~b6 → Attributes;
b7 → Create new flag.
IX – Start address of new FIB containing default filename.

Output: A – Error code (if 0, there was no error)
IX – Starting address of the FIB filled with the new entry.

OPEN (43H)

Function: Open handle file. If the "inheritable" bit of A is set, the handle file must be opened by another process (see function 60H).

Input: DE – FIB start address or ASCII string "drive/path/file".
A – Open mode:
b0=1 – Not written;
b1=1 – Not reading;
b2=1 – Inheritable;
b3~b7 – Must be "0".

Output: A – Error code (if it is 0, there was no error).
B – New handle file.

CREATE (44H)

Function: Create handle file. The file created by this function will automatically open (function 43H)

Input: DE – Drive/path/file or ASCII string.
A – Open mode:
b0=1 – Not written;
b1=1 – Not reading;
b2=1 – Inheritable;
b3~b7 – Must be "0".
B – b0~b6 = Attributes;
b7 = Create new flag.

Output: A – Error code (if it is 0, there was no error).
B – New handle file.

CLOSE (45H)
Function: Close file handle.
Input: B – File handle to close.
Output: A – Error code (if it is 0, there was no error).

ENSURE (46H)
Function: Protect file handle (the current file pointer cannot be modified).
Input: B – Handle file to be protected.
Output: A – Error code (if it is 0, there was no error).

DUP (47H)
Function: Duplicate handle file.
Input: B – Handle file.
Output: A – Error code (if it is 0, there was no error).
B – New handle file.

READ (48H)
Function: Read from handle file. The four control sequences (Ctrl+P, Ctrl+N, Ctrl+S and Ctrl+C) are checked.
Input: B – Handle file.
DE – Start address of the buffer.
HL – Number of bytes to read.
Output: A – Error code (if it is 0, there was no error).
HL – Number of bytes actually read.

WRITE (49H)
Function: Write by a handle file. If the end of file is found, it will be extended up to the required value.
Input: B – Handle file.
DE – Start address of the buffer.
HL – Number of bytes to write.
Output: A – Error code (if it is 0, there was no error).
HL – Number of bytes actually written.

SEEK (4AH)
Function: Move handle file pointer.
Input: B – Handle file.
A – Method code:
0 – Relative to the beginning of the file;
1 – Relative to the current position;
2 – Relative to the end of the file.
DE:HL – Offset signalling.
Output: A – Error code (if it is 0, there was no error).
DE:HL – New file pointer.

IOCTL (4BH)
Function: Control for I/O devices.
Input: B – Handle file.
A – Subfunction code:
00H – Read status from handle file;
01H – Set ASCII/binary mode;
02H – Tests if device. is ready for entry;
03H – Tests if device. is ready for exit;
04H – Calculates screen size.
DE – Other parameters.
Output: A – Error code (if it is 0, there was no error).
DE – Other return values.
Note: If A equals 0 on input, then the DE register must be loaded with the following parameters:
⟶ For devices:
b0=1 – Input device;
b1=1 – Output device;
b2~b4 – Reserved;
b5=1 – ASCII mode;
=0 – Binary mode;
b6=1 – End of file;
b7=1 – Device (always 1);
b8~b15 – Reserved.
⟶ For files:
b0~b5 – Drive number (0=A:, etc.);
b6=1 – End of file;
b7=0 – Disk file (always 0);
b8~b15 – Reserved.

On return, DE will have the same values. If A=1, only bit 5 of DE must be specified; other bits will be ignored. If A is equal to 2 or 3, register E will return with the value 00H if the device is not ready and with FFH if the device is ready. If A equals 4, DE returns the logical screen size value for the handle file (D=number of rows and E=number of columns). For devices other than the screen, DE will return with the value 0000H.

HTEST (4CH)

Function: Test handle file.
Input: B – Handle file.
DE – Pointer to FIB or to ASCII string "drive/path/file".
Output: A – Error code (if it is 0, there was no error).
B – 00H = not the same file;
FFH = Is the same file.

DELETE (4DH)

Function: Delete file or subdirectory. A subdirectory can only be deleted if it does not contain any files. If a device name is specified it will not return an error, but of course the device will not be "erased".
Input: DE – Pointer to FIB or to ASCII string "drive/path/file".
Output: A – Error code (if it is 0, there was no error).

RENAME (4EH)

Function: Rename file or subdirectory.
Input: DE – Pointer to FIB or to ASCII string "drive/path/file".
HL – Pointer to the new ASCII name.
Output: A – Error code (if it is 0, there was no error).

MOVE (4FH)

Function: Move file or subdirectory. A file cannot be moved if the respective file handle is open. The FIB of the moved file will not be updated.
Input: DE – Pointer to FIB or to ASCII string "drive/path/file".
HL – Pointer to new ASCII string path.
Output: A – Error code (if it is 0, there was no error).

ATTR (50H)

Function: Set or read attributes of a file. The attributes of a file cannot be modified if the corresponding handle file is open.

Input: DE – Pointer to FIB or to ASCII string "drive/path/file".
A = 0 – Read attributes;
1 – Write attributes.
L – New attribute byte (if A = 1).

Output: A – Error code (if it is 0, there was no error).
L – Current attribute byte.

FTIME (51H)

Function: Read or set date and time in a file.

Input: DE – Pointer to FIB or to ASCII string "drive/path/file".
A = 0 – Read date and time;
1 – set date and time.
IX – New time (if A=1).
HL – New date (if A=1).

Output: A – Error code (if it is 0, there was no error).
DE – Time of current file.
HL – Current file date.

HDELET (52H)

Function: Delete handle file. If there is another handle file open for the same file, it cannot be deleted.

Input: B – Handle file.

Output: A – Error code (if it is 0, there was no error).

HRENAM (53H)

Function: Rename handle file. The file cannot be renamed if there is another file handle open for the same file. This function is identical to function 4EH, except that the HL register cannot point to a FIB.

Input: B – Handle file.
HL – New ASCII filename.

Output: A – Error code (if it is 0, there was no error).

HMOVE (54H)

Function: Move handle file. The file cannot be moved if there is another file handle open for the same file. This function is identical to function 4FH, except that the HL register cannot point to a FIB.

Input: B – Handle file.
HL – New path in ASCII.
Output: A – Error code (if it is 0, there was no error).

HATTR (55H)
Function: Read or set handle file attributes. Attribute byte cannot be modified if there is another handle file opened for the same file.
Input: B – Handle file.
A = 0 – read attributes;
1 – set attributes.
L – New attribute byte (if A=1).
Output: A – Error code (if it is 0, there was no error).
L – Current attribute byte.

HFTIME (56H)
Function: Read or change time and date from handle file. If there is another handle file open for the same file, the date and time cannot be changed. This function is identical to function 51H, except that there is no pointer; only the handle file.
Input: B – Handle file.
A = 0 – Read date and time;
1 – Set date and time.
IX – New time (if A=1).
HL – New date (if A=1).
Output: A – Error code (if it is 0, there was no error).
DE – Current file time.
HL – Current file date.

GETDTA (57H)
Function: Read start address of the DTA (Disk Transfer Area).
Input: None.
Output: DE – DTA start address.

GETVFY (58H)
Function: Read write verification flag.
Input: None.
Output: B = 0 – Write check disabled;
1 – Write check enabled.

GETCD (59H)

Function: Read current directory or subdirectory.
Input: B – Drive number (0=current; 1=A:, etc.).
DE – Start address of a 64-byte buffer.
Output: A – Error code (if it is 0, there was no error).
DE – Points to the filled buffer. The drive name and "\" character are not included. If there is no current directory, the buffer will be filled with 00H bytes.

CHDIR (5AH)

Function: Change current subdirectory.
Input: DE – ASCII string "drive/path/name".
Output: A – Error code (if it is 0, there was no error).

PARSIS (5BH)

Function: Parses pathname (path name).
Input: B – Volume name flag (bit 4).
bit4 = 0 ⟶ string "drive/path/file"
1 ⟶ string "drive/volume"
DE – ASCII string for analysis.
Output: A – Error code (if it is 0, there was no error).
DE – Pointer to the ending character.
HL – Pointer to the beginning of the last item.
B – Analysis flags.
b0=1 if any character points to another drive name;
b1=1 if any path directory is specified;
b2=1 if drive name is specified;
b3=1 if master file is specified in the last item;
b4=1 if filename extension is specified in the last item;
b5=1 if the last item is ambiguous;
b6=1 if the last item is "." or "..";
b7=1 if the last item is "...".
C – Logical drive (1=A:, etc.).
Note: The value returned in HL will point to the first character of the last item in the string. For example, for the string "A:\XYZ\P.Q /F", DE will point to the white space before "/F" and HL will point to "P".

PFILE (5CH)
Function: Parse filename.
Input: DE – ASCII string to be parsed, no drive specification. Wildcard characters (? and *) can be used.
HL – Pointer to an 11-byte buffer.
Output: A – Always 00H.
DE – Pointer to the final character.
HL – Pointer to the filled buffer.
B – Analysis flags. The values are identical to the 5BH function, except that bits 0, 1 and 2 will always be 0.

CHKCHR (5DH)
Function: Check character. 16-bit characters are also checked.
Input: D – Character flags.
b0=1 to suppress the character. In this case, the character returned in E will always be the same.
b1=1 if it is the first byte of a 16-bit character;
b2=1 if it is the second byte of a 16-bit character;
b3=1 if it is volume name or preferably filename;
b4=1 if it is invalid file/volume character;
b5~b7 are reserved (always 0).
E – Character to be checked.
Output: A – Always 00H.
D – Updated character flags.
E – Checked character.

WPATH (5EH)
Function: Read complete string path, without drive specification and "\" character. For greater reliability, call function 40H or 41H first and then call WPATH twice, as other functions may change the data.
Input: DE – Pointer to a 64-byte buffer.
Output: A – Error code (if it is 0, there was no error).
DE – Buffer filled with the complete path string.
HL – Pointer to the beginning of the last item.

FLUSH (5FH)
Function: Unload disk buffers.

Input: B – Drive specification (0=current; 1=A:, etc. If FFH, unload all drives).
D = 00H – Unload only;
FFH – Unload and invalidate.
Output: A – Error code (if it is 0, there was no error).

FORK (60H)
Function: Branch files into tree.
Input: None.
Output: A – Error code (if it is 0, there was no error).
B – Branch process ID.

JOIN (61H)
Function: Join files in tree. This function returns to the original handle file the handle file copied by the previous function. The copied file is automatically closed and the original handle file is reactivated.
Input: B – Branch process ID (or 0).
Output: A – Error code (if it is 0, there was no error).
B – Branch primary error code.
C – Branch secondary error code.

TERM (62H)
Function: End with error code.
Input: B – Error code for termination.
Output: None.

DEFAB (63H)
Function: Set abort routine. Only available if called by 0005H.
Input: DE – Starting address of the abort routine; the default address is 0000H.
Output: A – Always 00H.

DEFER (64H)
Function: Set user routine for disk error.
Input: DE – Start address of disk error routine. The default value is 0000H.
Output: A – Always 00H.

Note: The specification of parameters and results of the created routine are as follows:

Input: A – Error code;
B – Physical drive number;
C – b0=1 if writing error;
b1=1 if you ignore the error (not recommended);
b2=1 if automatic abort is suggested;
b3=1 if the sector number is valid.
DE – Disk sector number (if b3 of C is 1).

Output: A = 0 – Call system error routine;
1 – Abort;
2 = Try again;
3 = Ignore.

ERROR (65H)

Function: Catch error code in advance to prevent the kind of error that might occur on the next function call.

Input: None.

Output: A – Always 00H.
B – Function error code.

EXPLN (66H)

Function: Return error code message.

Input: B – Error code.
DE – Pointer to a 64-byte buffer.

Output: A – Always 00H.
B – Error code or 00H.
DE – Buffer filled with error message in ASCII format.

FORMAT (67H)

Function: Format a disk.

Setup: B – Drive number (0=current; 1=A:, etc.).
A = 00H – Return choice message;
01H~09H – Formats with this choice;
0AH~0DH – Illegal;
FEH – Update parameters for MSXDOS2;
FFH – Full update for MDXDOS2.
HL – Pointer to the buffer (if A = 1~9).
DE – Buffer size (if A = 1~9).

Output: A – Error code (if it is 0, there was no error).
B – Chosen message slot (only if A=0 on input).
HL – Address of the chosen message (only if A=0).

RAMD (68H)
Function: Create or delete ramdisk on drive "H:".
Input: B – 00H = Clear the ramdisk;
01H~FEH = Create new ramdisk with xxH 16K logical segments.
FFH = Returns ramdisk size in 16K segments.
Output: A – Error code (if it is 0, there was no error).
B – Ramdisk size.

BUFFER (69H)
Function: Allocate buffers (each is 16K).
Input: B = 0 – Returns number of allocated buffers;
1 to 20 – allocates the specified number of buffers.
21 or more (more than 15H) – Invalid
Output: A – Error code (if it is 0, there was no error).
B – Total number of allocated buffers

ASSIGN (6AH)
Function: Assign logical drive to a physical drive.
Input: B – Logical drive number (1=A:, 2=B:, etc.):
0 – Cancel all assignments (for D = 0);
1 to 7 – Assign/cancel respective logical drive;
D – Physical drive number (1=A:, 2=B:, etc.).
0 – Cancel assignment (for B = 1 to 7):
1 to 7 – Assign respective physical drive;
FFH – Only return logical drive on D.
Output: A – Error code (if it is 0, there was no error).
D – Physical drive number.

GENV (6BH)
Function: Read external item.
Input: HL – Pointer to ASCII string name.
DE – Buffer pointer to value string.
B – Buffer size. If the buffer is small, the return value will be truncated and terminated in 00H. A 255-byte buffer will always sufficient.

Output: A – Error code (if it is 0, there was no error).
DE – Pointer to the filled buffer.

SENV (6CH)
Function: Set external item.
Input: HL – Pointer to ASCII name.
DE – Pointer to the value string to be set. Must be up to 255 characters long and end in 00H. If the string is null, the outer item will be removed.
Output: A – Error code (if it is 0, there was no error).

FENV (6DH)
Function: Search for external item.
Input: DE – External item number.
HL – Buffer pointer to ASCII name.
Output: A – Error code (if it is 0, there was no error).
HL – Pointer to the filled buffer, the end of which is marked with a 00H byte.

DSKCHK (6EH)
Function: Enable or disable disk checking. When the check is active, the system will reload boot, FAT, FIB, FCB, etc. From the disk every time it is changed.
Input: A = 00H – Read check value from disk;
01H – Set disk check value.
B = 00H – Active (if A=01H);
01H – Disables (if A=01H).
Output: A – Error code (if it is 0, there was no error).
B – Current disk check value.

DOSVER (6FH)
Function: Read MSXDOS version number. Values returned in registers BC and DE will be in BCD. So if the version is 2.34, the returned value will be 0234H. For compatibility with MSXDOS1 check first if there was any error (A≠0).
Input: None.
Output: A – Error code (if it is 0, there was no error).
BC – DOS Kernel version.
DE – MSXDOS2.SYS version.

REDIR (70H)
Function: Read or set redirection state. The effect of this function is temporary, in the case of A=01H and B=00H at the input.
Input: A = 00H – Read redirection status;
01H – Set redirection status.
B – New state:
b0 – Standard input;
b1 – Standard output.

4.3.6 – Functions added by NEXTOR

FOUT (71H)
Function: Enables or disables quick STROUT mode. When enabled, only the first 511 characters will be printed.
Input: A = 00H – Get fast STROUT mode;
01H – Set fast STROUT mode.
B = 00H – Disable (only if A = 01H);
FFH – enable (only if A = 01H).
Output: A – Error code (if it is 0, there was no error).
B – Current fast STROUT mode.

ZSTROUT (72H)
Function: Print a zero-terminated string. This function is affected by the quick STROUT mode.
Input: DE – String address.
Output: A = 0 (never returns an error).

RDDRV (73H)
Function: Read the absolute sectors of the unit. This function is able to read sectors regardless of the file system viewed on the drive (FAT12, FAT16 or an unknown file system), and even when there is no file system. The read sectors will be placed from the current disk's DTA.
Input: A – Unit number (0 = A:, 1=B:, etc.).
B – Number of sectors to read.
HL:DE – Sector number.
Output: A – Error code (if it is 0, there was no error).

WRDRV (74h)

Function: Write absolute sectors to disk. This function is able to record sectors regardless of the file system viewed on the drive (FAT12, FAT16 or an unknown file system), and even when there is no file system. The sectors will be written from the current disk's DTA.

Input: A – Unit number (0 = A:, 1=B:, etc.).
B – Number of sectors to read.
HL:DE – Sector number.

Output: A – Error code (if it is 0, there was no error).

RALLOC (75H)

Function: Get or set reduced allocation information mode vector. The vector assigns a bit to each drive; bit 0 of L is for A :, bit 1 of L is for B :, etc. This bit is 1 if the reduced allocation mode is currently enabled (when getting the vector) or to be enabled (when setting the vector) for the drive, 0 when the mode is disabled or to be disabled.

Input: A = 00H – Get current vector;
01H – Define vector.
HL – New vector (only if A = 01H).

Output: A – 0 (never returns an error).
HL – Current vector.

DSPACE (76H)

Function: Get disk space information. The extra value in BC will be nonzero only when the unit's minimum allocation unit is not an integer in Kbytes.

Input: E – Unit number (0 = Standard, 1 = A :, etc)
A = 00H – Get free space;
01H – Get full space.

Output: A – Error code (if it is 0, there was no error).
HL:DE – Space in Kbytes.
BC – Extra space in bytes.

LOCK (77H)

Function: Lock / unlock a unit or get the lock status of a unit. When a drive is locked, Nextor will assume that the media on that drive will never change and therefore will never ask the associated driver for media change status; thus resulting in

an overall increase in media access speed. This is useful when using removable devices as the primary storage device.

Input: E – Physical unit (0 = A :, 1 = B :, etc)
A = 00H – Get lock status;
01H – Set lock status.
B = 00H – Unlock unit (only if A = 01H);
FFH – Unit lock (only if A = 01H).

Output: A – Error code (if 0, no error).
B – Current blocking state, same as input.

GDRVR (78H)

Function: Get information about a device driver.

Input: A – Driver index)0 to specify slot and segment) D driver slot number (only if A = 0).
E – Driver segment number, FFH for drivers in ROM (only if A = 0).
HL – Pointer to 64-byte data buffer.

Output: A – Error code (if it is 0, there was no error).
HL – Points to buffer filled with driver data.
+0: Driver slot number.
+1: Driver segment number, FFh if the driver is built into a Nextor or MSX-DOS kernel ROM (always FFH in the current version).
+2: Number of drive letters assigned to this driver at boot time.
+3: First drive letter assigned to this driver at boot time (A: = 0, etc). Not used if no drive is assigned at boot time.
+4: Driver Flags:
bit 7: 1 ⟶ Nextor driver;
0 ⟶ MSX-DOS driver (built into MSX-DOS kernel ROM).
bits 6-3: Not used, always "0000".
bit 2: 1 ⟶ Driver implements DRV_CONFIG routine.
bit 1: Not used, always zero.
bit 0: 1 ⟶ Device-based driver;
0 ⟶ Drive-based driver.
+5: Driver version number (MSB).

+6: Driver version number (LSB).
+7: Driver revision number.
+8: Driver name, left-justified, complete with spaces (32 bytes).
+40 ~ +63: Reserved (currently always zero).

GDLI (79H)
Function: Get information about a drive letter
Input: A – Physical unit (0 = A :, 1 = B :, etc)
HL – Pointer to 64-byte data buffer
Output: A – Error code (if 0, no error).
HL – Pointer to filled buffer
+0: Unit Status
0 – Not assigned
1 – Assigned to a storage device connected to a Nextor or MSX-DOS driver
2 – Not used
3 – A file is mounted on the drive
4 – Assigned RAMdisk (other fields will be zero)
+1: Driver slot number
+2: Driver segment number (FFH if the driver is built into a Nextor or MSX-DOS kernel ROM)
+3: Relative drive number within driver (for drive--based drivers only FFH if driver is device-based)
+4: Device index (only for device-based drivers; 0 for MSX-DOS drivers)
+5: Logical unit index (only for device-based drivers; 0 for MSX-DOS drivers)
+6 ~ +9: Device's first sector number (only for device-based drivers; 0 for MSX-DOS drivers)
+10 ~ +63: Reserved (currently always zero)
⟶ If a file is mounted on the drive, the information returned in the data buffer will be inserted as follows:
+1: Drive where mounted file is located:
(0 = A:, 1 = B:, etc)
+2: Flags:
bit0 = 0 – Read and write, 1 – Read only
+3: Always 0
+4: filename in print format (up to 12 characters, plus a terminating zero)

GPART (7AH)

Function: Get information about a device partition. This function only works on device-based drivers.

Input:
- A – Driver slot number.
- B – Driver segment number, FFH for drivers in ROM.
- D – Device index.
- E – Logical unit index.
- H – Primary partition number (1 to 4).
- H:7 = 0 – Get partition information;
  - 1 – Get the sector number of the device containing the partition table entry.
- L – Extended partition number (0 for an entry in the primary partition table).

Output:
- A – Error code (if 0, no error).
- ⟶ If partition information is requested:
- B – Partition type code:
  - 0 – None (specified partition does not exist).
  - 1 – FAT12.
  - 4 – FAT16, less than 32 MB (obsolete).
  - 5 – Extended (handles more than 4 partitions).
  - 6 – FAT16 (CHS).
  - 14 – FAT16 (LBA).
- C – Partition status byte.
- HL:DE – Partition absolute sector starting number
- IX:IY – Partition size in sectors.
- ⟶ If the sector number of the partition table entry is requested:
- HL:DE – Sector number of the device containing the partition table entry.

CDRVR (7BH)

Function: Call a routine in a device driver. This function works in MSX-DOS 1 mode.

Input:
- A – Driver slot number.
- B – Driver Segment No., FFH for drivers in ROM.
- DE – Routine address.
- HL – Pointer to an 8-byte buffer with input values. The order of registers is as follows: F, A, C, B, E, D, L, H.

Output: A – Error code (if 0, no error).
BC, DE, HL – Routine results.
IX – AF value returned by the routine.

MAPDRV (7CH)
Function: Map a drive letter to a device driver
Input: A – Physical unit (0 = A :, 1 = B :, etc)
B – Action to be taken:
0 – Remove drive mapping.
1 – Map the drive to its default state.
2 – Map the drive using specific mapping data.
3 – Mount a file on the drive.
HL – If B=2:
Address of an 8-byte buffer with mapping data with the following structure:
+0 – Driver slot number
+1 – Driver segment number (FFH if the driver is embedded in a Nextor kernel ROM)
+2 – Device number
+3 – Logical unit number
+4 ~ +7 – Home sector
If B=3:
Pointer to filename or FIB address.
D – File mount type (if B = 3).
0 – Automatic (read only if file has this attribute set, read and write otherwise).
1 – Read only.
Output: A – Error code (if 0, no error).

Z80MODE (7DH)
Function: Enable or disable the Z80's access to a driver. This function works only on MSX Turbo R computers.
Input: A – Driver slot number.
B = 00H – Get current Z80 access mode;
01H – Set Z80 access mode.
D = 00H – Disable Z80 access mode (only if B = 01H);
FFH – Enable Z80 access mode (only if B = 01H).
Output: A – Error code (if 0, no error).
D – Current Z80 access mode for specified driver (same as input).

GETCLUS (7EH)
Function: Get information for a cluster on a FAT drive.
Input: A – Unit number (0 = Standard, 1 = A: etc.).
DE – Cluster number.
HL – Pointer to a 16-byte buffer.
Output: A – Error code (if 0, no error).
HL – Pointer to the filled buffer:
+0 – FAT sector number containing the entry for the cluster (2 bytes).
+2 – Offset in the FAT sector where the entry for the cluster is located (0-511).
+4 – First number of the data sector to which the cluster refers (4 bytes).
+8 – FAT input value for cluster (2 bytes).
+10 – Size of a cluster in sectors for the unit (1 byte).
+11 – Flags (1 byte):
bit 0 = 1 if the drive is FAT12.
bit 1 = 1 if the drive is FAT16.
bit 2 = 1 if the FAT entry to the cluster is an odd entry (FAT12 only).
bit 3 = 1 if the cluster is the last of a file.
bit 4 = 1 if the cluster is free.
bits 5-7: Unused, always zero.
+12 ~ +15: Unused, always zero.
The FAT entry value for cluster has the following meanings:
0 ⟶ Free cluster.
0FF8H-0FFFH for FAT12 and FFF8H-FFFFH for FAT16 ⟶
⟶ Cluster is the last of a file.
Other value ⟶ Number of the next cluster where data for a file continues.

4.4 – MSXDOS ERROR CODES

50 FIELD overflow
51 Internal error
52 Bad file number
53 File not found
54 File open

55 End of file
56 Bad filename
57 Direct statement in file
58 Sequential I/O only
59 File not OPEN
60 Disk error
61 Bad file mode
62 Bad drive name
63 Bad sector
64 File still open
65 File already exists
66 Disk full
67 Too many files
68 Write protected disk
69 Disk I/O error
70 Disk offline
71 RENAME across disk

4.5 – MSXDOS2 ERROR CODES

4.5.1 – Disk Errors

FFH Incompatible disk
FEH Write error
FDH Disk error
FCH Not ready
FBH Verify error
FAH Data error
F9H Sector not found
F8H Write protected disk
F7H Unfomatted disk
F6H Not a DOS disk
F5H Wrong disk
F4H Wrong disk for file
F3H Seek error
F2H Bad file allocation table
F1H No message
F0H Cannot format this drive

4.5.2 – MSXDOS Functions Errors

DFH Internal error
DEH Not enough memory
DDH -
DCH Invalid MSX-DOS call
DBH Invalid drive
DAH Invalid filename
D9H Invalid pathname
D8H Pathname too long
D7H File not found
D6H Directory not found
D5H Root directory full
D4H Disk full
D3H Duplicate filename
D2H Invalid directory move
D1H Read only file
D0H Directory not empty
CFH Invalid attributes
CEH Invalid . or .. operation
CDH System file exists
CCH Directory exists
CBH File exists
CAH File already in use
C9H Cannot transfer above 64K
C8H File allocation error
C7H End of file
C6H File access violation
C5H Invalid process id
C4H No spare file handles
C3H Invalid file handle
C2H File handle not open
C1H Invalid device operation
C0H Invalid environment string
BFH Environment string too long
BEH Invalid date
BDH Invalid time
BCH RAM disk already exists
BBH RAM disk does not exist

BAH File handle has been deleted
B9H Internal error
B8H Invalid sub-function number

4.5.3 – Errors Added by Nextor

B6H Invalid device driver
B5H Invalid device or LUN
B4H Invalid partition number
B3H Partition is already in use
B2H File is mounted
B1H Bad file size
B0H Invalid cluster number

4.5.4 – End Programs Errors

9FH Ctrl-STOP pressed
9EH Ctrl-C pressed
9DH Disk operarion aborted
9CH Error on standard output
9BH Error on standard input

4.5.5 – Command Errors

8FH Wrong version off COMMAND
8EH Unrecognized command
8DH Command too long
8CH Internal error
8BH Invalid parameter
8AH Too many parameters
89H Missing parameter
88H Invalid option
87H Invalid number
86H File for HELP not found
85H Wrong version of MSX-DOS
84H Cannot concatenate destination file
83H Cannot create destination file
82H File cannot be copied onto itself
81H Cannot overwrite previous destination file

5 – SYMBOS

5.1 – KERNEL ROUTINES

5.1.1 – Kernel Restarts

RST 08H (MSGSLP) – Message_Sleep_And_Receive

Description: Checks for a new message from another process. If there is no message, the process will be switched into sleep mode, until a message is available. For more information about receiving message, see RST 18H (MSGGET).

Input: IXl – Receiver process ID (your own one).
IXh – Sender process ID (–1 to check any process).
IY – Pointer to message buffer (14 bytes).

Output: IXl – 0 ⟶ no message available, 1 ⟶ msg received.
IXh – Sender process ID (if IXl=1).

Registers: AF, BC, DE, HL.

RST 10H (MSGSND) – Message_Send

Description: Sends a message to another process. IXl must contain your own process ID and IXh the ID of the receiver. If the message queue is full or the receiver does not exist, it will not be sent. The message must always be placed between C000H and FFFFH (transfer RAM area) and can have a maximum size of 14 bytes.

Input: IXl – Sender process ID (your own one).
IXh – Receiver process ID.
IY – Pointer to the message (1–14 bytes).

Output: IXl – 0 ⟶ message queue is full.
1 ⟶ message has been sent successfully.
2 ⟶ receiver process does not exist.

Registers: AF, BC, DE, HL.

RST 18H (MSGGET) – Message_Receive

Description: Checks for a new message from another process. The message buffer must have a size of 14 bytes and always be placed between C000H and FFFFH (transfer RAM area).

Input: IXl – Receiver process ID (your own one).
IXh – Sender process ID (–1 to check any process).
IY – Pointer to message buffer (14 bytes).
Output: IXl – 0 ⟶ no message available, 1 ⟶ msg received.
IXh – Sender process ID (if IXl=1).
Registers: AF, BC, DE, HL.

0H (BNKSCL) – Banking_SlowCall
Description: Calls a routine, which is placed in the first RAM bank. All registers will be transfered unmodified to and from the routine. The address of the routine has to be specified
Input: (SP+0) – Destination address.
AF, BC, DE, HL, IX, IY – Registers for the dest routine.
Output: AF, BC, DE, HL, IX, IY – Registers from the dest routine.
Registers: –
Example: rst 20H : dw 8130H
⟶ Calls the routine at 8130H in the first RAM bank.

RST 28H (BNKFCL) – Banking_FastCall
Description: Calls a routine, which is placed in the first RAM bank. DE,IX and IY will be transfered unmodified to and from the routine. It is faster than RST 20H (BNKSCL). Don't use this function, if the routine does make bank switching or requires more registers than DE,IX,IY.
Input: HL – Destination address.
DE, IX, IY – Registers for the destination routine.
Output: DE, IX, IY – Registers from the destination routine.
Registers: AF, BC, HL.
Example: ld hl, 08 109H : rst 028H
⟶ Calls the routine at 8109H in the first RAM bank.

RST 30H (MTSOFT) – Multitasking_SoftInterrupt
Description: Releases the CPU time for the operating system. If the process currently has nothing to do (it is waiting for something, you should call this function, so that other processes can get CPU time, too. A process, which called this function, is marked as "idle".
Input: –
Output: –
Registers: –

RST 38H (MTHARD) – Multitasking_HardInterrupt
Description: You shouldn't call this function by yourself. It is called by the hardware interrupt, which comes 50 or 300 times per second, depending on the computer system.
Input: –
Output: –
Registers: –

5.1.2 – Kernel Commands (Multitasking Management)

Kernel commands are triggered via a message, which has to be sent with RST 10H (MSGSND) to the kernel process. The kernel process always has the ID 1.

ID: 001 (MSC_KRL_MTADDP) – Multitasking_Add_Process_Command
Description: Adds a new process with a given priority and starts it immediately. Application processes usually will be started with priority 4.
Library: SyKernel_MTADDP
Message:
00 1B 001
01 1W Stack address (see notes below).
03 1B Priority (1=highest, 7=lowest).
04 1B RAM bank (0~15).
Example (stack):

```
        ds 128  ;Stack buffer
stack_ptr:
        dw 0    ;initial value for IY
        dw 0    ;initial value for IX
        dw 0    ;initial value for HL
        dw 0    ;initial value for DE
        dw 0    ;initial value for BC
        dw 0    ;initial value for AF
        dw process_start ;process start adr
process_id: db 0 ;kernel writes the ID here
```

Response: See MSR_KRL_MTADDP

ID: 002 (MSC_KRL_MTDELP) – Multitasking_Delete_Process_Command
Description: Stops an existing process and deletes it.
Library: SyKernel_MTDELP.

Message: 00 1B 002.
01 1B Process ID.
Response: See MSR_KRL_MTDELP.

ID: 003 (MSC_KRL_MTADDT) – Multitasking_Add_Timer_Command
Description: Adds a new timer and starts it immediately. Timers will be called 50 or 60 times per second, depending on the screen vsync frequency. Please see MSC_KRL_MTADDP for information about the stack.
Library: SyKernel_MTADDT.
Message: 00 1B 003.
01 1W Stack address.
04 1B RAM bank (0~15).
Response: See MSR_KRL_MTADDT.

ID: 004 (MSC_KRL_MTDELT) – Multitasking_Delete_Timer_Command
Description: Stops an existing timer and deletes it.
Library: SyKernel_MTDELT.
Message: 00 1B 004.
01 1B Timer ID.
Response: See MSR_KRL_MTDELT.

ID: 005 (MSC_KRL_MTSLPP) – Multitasking_Sleep_Process_Command
Description: Puts an existing process into the sleep mode. It is stopped and does not run anymore, until it receives a message, or until it will be wacked up again (see MSC_KRL_MTWAKP).
Library: SyKernel_MTSLPP.
Message: 00 1B 005.
01 1B Process ID.
Response: See MSR_KRL_MTSLPP.

ID: 006 (MSC_KRL_MTWAKP) – Multitasking_WakeUp_Process_Command
Description: Wakes up a process, which was sleeping before. A process will be wacked up, too, when another process is sending a message to it.
Library: SyKernel_MTWAKP.
Message: 00 1B 006.
01 1B Process ID.
Response: See MSR_KRL_MTWAKP

ID: 007 (MSC_KRL_TMADDT) – Timer_Add_Counter_Command
Description: Adds a counter for a process. You need to specify a byte anywhere in the memory. This byte then will be increased every [P5]/50 seconds. This is much easier and faster than setting up an own timer.
Library: SyKernel_TMADDT.
Message: 00 1B 007.
01 1W Counter byte address.
03 1B Counter byte RAM bank (0~15).
04 1B Process ID.
05 1B Speed (counter will be increased every x/50 secs).
Response: See MSR_KRL_TMADDT.

ID: 008 (MSC_KRL_TMDELT) – Timer_Delete_Counter_Command
Description: Stops the specified counter. Please note, that this will be done automatically, if the process should be deleted.
Library: SyKernel_TMDELT.
Message: 00 1B 008.
01 1W Counter byte address.
03 1B Counter byte RAM bank (0~15).
Response: See MSR_KRL_TMDELT.

ID: 009 (MSC_KRL_TMDELP) –
– Timer_Delete_AllProcessCounters_Command
Description: Stops all counters of one process. Please note, that this will be done automatically, if the process should be deleted.
Library: SyKernel_TMDELP.
Message: 00 1B 009.
01 1B Process ID.
Response: See MSR_KRL_TMDELP.

ID: 010 (MSC_KRL_MTPRIO) –
– Multitasking_Process_Priority_Command
Description: Changes the priority of a process. A process is able to change its own priority.
Library: SyKernel_MTPRIO.
Message: 00 1B 010.
01 1B Process ID.
01 1B New Priority (1=highest, 7=lowest).
Response: See MSR_KRL_MTPRIO.

5.1.3 – Kernel Responses (Multitasking Mangement)

Kernel responses are coming as a message, which has to be received with RST 18H (MSGSND) or RST 08H (MSGSLP) from the Kernel process. which always has the ID 1.

ID: 129 (MSR_KRL_MTADDP) – Multitasking_Add_Process_Response
Description: The kernel sends this message after trying to add a new process (see MSC_KRL_MTADDP). You shouldn't add another process until you receive this message.
Message: 00 1B 129.
01 1B Error status (0=successful, 1=failed).
02 1B Process ID (if P1=0).

ID: 130 (MSR_KRL_MTDELP) – Multitasking_Delete_Process_Response
Description: The kernel sends this message after deleting an existing process (see MSC_KRL_MTDELP).
Message: 00 1B 130.

ID: 131 (MSR_KRL_MTADDT) – Multitasking_Add_Timer_Response
Description: The kernel sends this message after trying to add a new timer (see MSC_KRL_MTADDT). You shouldn't add another timer until you receive this message.
Message: 00 1B 131.
01 1B Error status (0=successful, 1=failed).
02 1B Timer ID (if P1=0).

ID: 132 (MSR_KRL_MTDELT) – Multitasking_Delete_Timer_Response
Description: The kernel sends this message after deleting an existing timer (see MSC_KRL_MTDELT).
Message: 00 1B 132.

ID: 133 (MSR_KRL_MTSLPP) – Multitasking_Sleep_Process_Response
Description: The kernel sends this message after puting a process into sleep mode (see MSC_KRL_MTSLPP).
Message: 00 1B 133.

ID: 134 (MSR_KRL_MTWAKP) –
– Multitasking_WakeUp_Process_Response
Description: The kernel sends this message after wacking up a process (see MSC_KRL_MTWAKP).
Message: 00 1B 134.

ID: 135 (MSR_KRL_TMADDT) – Timer_Add_Counter_Response
Description: The kernel sends this message after trying to add a new counter (see MSC_KRL_TMADDT).
Message: 00 1B 135.
01 1B Error status (0=successful, 1=failed).

ID: 136 (MSR_KRL_TMDELT) – Timer_Delete_Counter_Response
Description: The kernel sends this message after deleting a counter (see MSC_KRL_TMDELT).
Message: 00 1B 136.

ID: 137 (MSR_KRL_TMDELP) –
– Timer_Delete_AllProcessCounters_Response
Description: The kernel sends this message after deleting all counters of one process (see MSC_KRL_TMDELP).
Message: 00 1B 137.

ID: 138 (MSR_KRL_MTPRIO) – Multitasking_Process_Priority_Response
Description: The kernel sends this message after changing the priority of a process (see MSC_KRL_MTPRIO).
Message: 00 1B 138.

5.1.4 – Kernel Functions (Memory Management)

All kernel memory functions have to be called with RST 20H (BNKSCL) or RST 28H (BNKFCL).

MEMSUM (8100H) – Memory_Summary
Description: Gives back the size of the total existing memory (=D*65 536+65 536) and the amount of bytes (=E*65 536+IX), which are still available.
How to call: ld hl,8100H : rst 028H
Input: –

Output: E, IX – Free memory in bytes.
D – Number of existing 64K extended RAM banks.
Registers: A, BC, IY.

MEMINF (8121H) – Memory_Information
Description: Searches for the largest free area inside a 64K bank. If you don't specify the RAM bank (A=0) the system is searching for the largest area inside the whole memory.
How to call: rst 20H : dw 8121H
Input: A – RAM bank (1–15, 0 means search in any bank)
E – Memory type:
0 – Total (code area).
1 – Within a 16K block (data area).
2 – Within the last 16K block (transfer area).
Output: BC – Length of the largest free area.
A, HL – Total free memory in bytes.
Registers: F, DE

MEMGET (8118H) – Memory_Get
Description: Reserves the requested amount of memory in any or a special RAM bank. If the memory type is 1, it will be reserved inside a 16k block, if it is 2, inside the last 16K block of the RAM bank.
How to call: rst 20H : dw 8118H.
Input: A – RAM bank (1–15, 0 means search in any bank).
E – Memory type:
0 – Total (code area).
1 – Within a 16K block (data area).
2 – Within the last 16K block (transfer area).
BC – Length in bytes.
Output: A – RAM bank (1–15).
HL – Address.
CY – Error state (CY=1 ⟶ Not enough memory free).
Registers: BC, DE.

MEMFRE (811BH) – Memory_Free
Description: Frees the specified memory. Please note, that because of performance and resources reasons the system will free it in any way, so be sure, that you really free only the memory you reserved by yourself.

How to call: rst 20H : dw 811BH.
Input: A – RAM bank (1–15).
HL – Address.
BC – Length in bytes.
Output: –
Registers: AF, BC, E, HL.

MEMSIZ (811EH) – Memory_Resize
Description: Changes the length of a reserved memory area. You will always have success, if the new length is smaller than the old one.
How to call: rst 20H : dw 811EH.
Input: A – RAM bank (1–15).
HL – Address.
BC – Old length in bytes.
DE – New length in bytes.
Output: CY – Error state (CY=1 → not enough free memory).
Registers: AF, BC, DE, HL.

5.1.5 – Kernel Functions (Banking Management)

Most kernel banking functions have to be called with RST 20H (BNKSCL) or RST 28H (BNKFCL). The interbank functions have to be called directly.

BNKRWD (8124H) – Banking_ReadWord
Description: Reads a word from an address in any RAM bank.
How to call: rst 20H : dw 8124H.
Input: A – RAM bank (0~15).
HL – Address.
Output: BC – Content of A, HL.
HL – Address+2.
Registers: –

BNKWWD (8127H) – Banking_WriteWord
Description: Writes a word to an address in any RAM bank.
How to call: rst 20H : dw 8127H.
Input A – RAM bank (0~15).
HL – Address.
BC – Word.

Output: HL – Address+2.
Registers: BC

BNKRBT (812AH) – Banking_ReadByte
Description: Reads a byte from an address in any RAM bank.
How to call: rst 20H : dw 812AH.
Input: A – RAM bank (0~15).
HL – Address.
Output: B – Content of A, HL.
HL – Address+1.
Registers: –

BNKWBT (812DH) – Banking_WriteByte
Description: Writes a byte to an address in any RAM bank.
How to call: rst 20H : dw 812DH.
Input: A – RAM bank (0~15).
HL – Address.
B – Byte.
Output: HL – Address+1.
Registers: BC.

BNKCOP (8130H) – Banking_Copy
Description: Copies a memory area from an address in any RAM bank to any other place in memory. The low nibble of the A register (bit 0–3) specifies the source bank, the high nibble (bit 4–7) the destination bank.
How to call: rst 20H : dw 8130H
Input: A – bit0–3 ⟶ Source RAM bank (0~15)
bit4–7 ⟶ Destination RAM bank (0~15)
HL – Source address
DE – Destination address
BC – Length
Output: –
Registers: AF, BC, DE, HL

BNKGET (8133H) – Banking_GetBank
Description: Gives back the number of the RAM bank, where the process is running.

How to call: rst 20H : dw 8133H.
Input: –
Output: A – RAM bank (1–15).
Registers: F.

BNK16C (8142H) – Banking_Call_Application16KRoutine
Description: Allows you to execute an application routine in the first RAM bank. The routine must be placed inside a 16K block (data RAM area), that will be switched to 4000H–7FFFH, and the routine will be called. After this the old memory configuration will be restored. The application has to relocate the routine by itself first, by setting bit15=0 and bit14=1 for every address pointer. The routine needs an own temporary stack during its execution, that´s must be placed in the same 16K block.
How to call: ld hl,8142H : rst 28H.
Input: IX – Pointer to data structure (between C000H–FFFFH).
00 1B Routine RAM bank (0~15).
01 1W Routine address.
03 1W Address of the temporary stack.
DE, IY – Will be handed over unmodified to the routine.
Ouput: DE, IX, IY – Will be received unmodified by the routine.
Registers: AF, BC, HL.

BNKCLL (FF03H) – Banking_Interbank_Call
Description: Switches to a routine into another 64K RAM bank. This allows to have code areas placed in multiple 64K RAM banks and to jump easily between them. The code must be relocated and its stack and transfer area must be placed between C000H and FFFFH as usual.
How to call: call FF03H.
Input: IX – Routine address.
B – Routine RAM bank (0~15).
IY – Address of the routines stack.
DE, HL – Will be handed over unmodified to the routine.
Registers: AF, BC, IY.

BNKRET (FF00H) – Banking_Interbank_Return
Description: Returns from a routine inside another 64K RAM bank to the caller in the primary bank. See BNKCLL.
How to call: jp FF00H.
Input: C, DE, HL, IX – Will be handed over unmodified to the caller.
Registers: AF, B, IY.

5.1.6 – Kernel Functions (Miscellaneous)

All miscellaneous kernel functions have to be called with RST 20H (BNKSCL) or RST 28H (BNKFCL). For more information see KERNEL FUNCTIONS (MEMORY MANAGEMENT).

MTGCNT (8109H) – Multitasking_GetCounter
Description: Gives back the system counter (=IY*65 536+IX) and the counter of the idle process. The system counter is increased 50 or 60 times per second. The idle process increases its counter every 64 microseconds, when it owns the CPU time.
How to call: ld hl,8109H : rst 28H
Input: –
Output: IY, IX – System counter
DE – Idle counter
Registers: –

5.2 – DESKTOP MANAGER COMMANDS

ID: 032 (MSC_DSK_WINOPN) – Window_Open_Command
Description: Opens a new window. Its data record must be placed in the transfer RAM area (between C000H and FFFFH).
Library: SyDesktop_WINOPN.
Message: 00 1B 32.
01 1B Window data record RAM bank (0–8).
02 1W Window data record address (C000H–FFFFH).
Response: See MSR_DSK_WOPNER and MSR_DSK_WOPNOK.

ID: 033 (MSC_DSK_WINMEN) – Window_Redraw_Menu_Command
Description: Redraws the menu bar of a window. If you changed your menus you should call this command to update the screen display. Works only if window has focus.
Library: SyDesktop_WINMEN.
Message: 00 1B 033.
01 1B Window ID.

ID: 034 (MSC_DSK_WININH) – Window_Redraw_Content_Command
Description: Redraws one, all or a specified number of controls inside the window content. Works only if window has focus.
Library: SyDesktop_WININH.
Message: 00 1B 034.
01 1B Window ID.
02 1B −1 ⟶ Control ID or negative number of controls.
000~239 ⟶ The control with the specified ID will be redrawed.
240~254 ⟶ Redraws −P2 controls, starting from control P3. As an example, if P2 is −3 (253) and P3 is 5, the controls 5, 6 and 7 will be edrawed.
255 ⟶ Redraws all controls inside the window content.
⟶ If P2 is between 240 and 254:
03 1B ID of the first control, which should be redrawed.

ID: 035 (MSC_DSK_WINTOL) – Window_Redraw_Toolbar_Command
Description: Redraws one, all or a specified number of controls inside the window toolbar. Use this command to update the screen display, if you made changes in the toolbar. Works only if window has focus.
Library: SyDesktop_WINTOL
Message: 00 1B 035
01 1B Window ID
02 1B −1 ⟶ Control ID or negative number of controls.
000~239 ⟶ The control with the specified ID will be redrawed.

240~254 ⟶ Redraws −P2 controls, starting from control P3. As an example, if P2 is −3 (253) and P3 is 5, the controls 5, 6 and 7 will be edrawed.
255 ⟶ Redraws all controls inside the window content.

⟶ If P2 is between 240 and 254:
03 1B ID of the first control, which should be redrawed.

ID: 036 (MSC_DSK_WINTIT) – Window_Redraw_Title_Command
Description: Redraws the title bar of a window. Use this command to update the screen display, if you changed the text of the window title. Works only if window has focus.
Library: SyDesktop_WINTIT.
Message: 00 1B 036.
01 1B Window ID.

ID: 037 (MSC_DSK_WINSTA) – Window_Redraw_Statusbar_Command
Description: Redraws the status bar of a window. Use this command to update the screen display, if you changed the text of the status bar. Works only if window has focus.
Library: SyDesktop_WINSTA.
Message: 00 1B 037.
01 1B Window ID.

ID: 038 (MSC_DSK_WINMVX) – Window_Set_ContentX_Command
Description: If the size of the window content is larger than the visible part, you can scroll its X offset with this command. The command works also, if the window is not resizeable by the user. Works only if window has focus.
Library: SyDesktop_WINMVX.
Message: 00 1B 038.
01 1B Window ID.
02 1W New X offset of the visible window content.

ID: 039 (MSC_DSK_WINMVY) – Window_Set_ContentY_Command
Description: If the size of the window content is larger than the visible part, you can scroll its Y offset with this command.

The command works also, if the window is not resizeable by the user. Works only if window has focus.
Library: SyDesktop_WINMVY.
Message: 00 1B 039.
01 1B Window ID.
02 1W New Y offset of the visible window content.

ID: 040 (MSC_DSK_WINTOP) – Window_Focus_Command
Description: Takes the window to the front position on the screen. Works in all conditions.
Library: SyDesktop_WINTOP.
Message: 00 1B 040.
01 1B Window ID.

ID: 041 (MSC_DSK_WINMAX) – Window_Size_Maximize_Command
Description: Maximizes a window. A maximized window has a special status, where it can't be moved to another screen position Works only if the window is minimized or restored.
Library: SyDesktop_WINMAX.
Message: 00 1B 041.
01 1B Window ID.

ID: 042 (MSC_DSK_WINMIN) – Window_Size_Minimize_Command
Description: Minimizes a window. It will disappear from the screen and can only be accessed by the user via the task bar. Works only if the window is minimized or restored.
Library: SyDesktop_WINMIN.
Message: 00 1B 042.
01 1B Window ID.

ID: 043 (MSC_DSK_WINMID) – Window_Size_Restore_Command
Description: Restores the window or the size of the window, if it was minimized or maximized before. Works only if the window is maximized or minimized.
Library: SyDesktop_WINMID.
Message: 00 1B 043.
01 1B Window ID.

ID: 044 (MSC_DSK_WINMOV) – Window_Set_Position_Command

Description: Moves the window to another position on the screen. Works only, if the window is not maximized.

Library: SyDesktop_WINMOV.

Message:
00 1B 044.
01 1B Window ID.
02 1W New X window position.
04 1W New Y window position.

ID: 045 (MSC_DSK_WINSIZ) – Window_Set_Size_Command

Description: Resizes a window. This command will always work, even if the window is not resizeable by the user. Please note, that the size always refers to the visible content of the window, not to the whole window including the control elements. So with title bar, scroll bars etc. a window can have a bigger size on the screen. Works always.

Library: SyDesktop_WINSIZ.

Message:
00 1B 045.
01 1B Window ID.
02 1W New window width.
04 1W New window height.

ID: 046 (MSC_DSK_WINCLS) – Window_Close_Command

Description: Closes the window. The desktop manager will remove it from the screen. Works in all conditions.

Library: SyDesktop_WINCLS.

Message:
00 1B 046.
01 1B Window ID.

ID: 047 (MSC_DSK_WINDIN) –
– Window_Redraw_ContentExtended_Command

Description: Redraws one, all or a specified number of controls inside the window content. This command is identical with MSC_DSK_WININH with the exception, that it always works but with less speed. See MSC_DSK_WININH.

Library: SyDesktop_WINDIN

Message:
00 1B 047
01 1B Window ID

02 1B Control ID, –1 (all) or negative number of controls.
000~239 ⟶ The control with the specified ID will be redrawed.
240~254 ⟶ Redraws –P2 controls, starting from control P3. As an example, if P2 is –3 (253) and P3 is 5, the controls 5, 6 and 7 will be edrawed.
255 ⟶ Redraws all controls inside the window content.
⟶ If P2 is between 240 and 254:
03 1B ID of the first control, which should be redrawed.

ID: 048 (MSC_DSK_DSKSRV) – Desktop_Service_Command
Description: Please read the desktop manager service description below for more information.
Library: See DESKTOP MANAGER SERVICES.
Message: 00 1B 048.
01 1B Service ID.
02~05 See desktop manager service description below.
Response: See MSR_DSK_DSKSRV.

ID: 049 (MSC_DSK_WINSLD) – Window_Redraw_Slider_Command
Description: Redraws the two slider of the window, with which the user can scroll the content. Sliders will only be displayed, if the window is resizeable. Works if window has focus.
Library: SyDesktop_WINSLD.
Message: 00 1B 049.
01 1B Window ID.

ID: 050 (MSC_DSK_WINPIN) –
– Window_Redraw_ContentArea_Command
Description: This command works like MSC_DSK_WINDIN, but it updates only a specified area inside the window content. Changes outside the area won't be updated. For more information see MSC_DSK_WINDIN and MSC_DSK_ _WININH. This command works in all conditions.
Library: SyDesktop_WINPIN
Message: 00 1B 050

01 1B Window ID
02 1B Control ID, –1 (all) or negative number of controls
000~239 ⟶ The control with the specified ID will be redrawed.
240~254 ⟶ Redraws –P2 controls, starting from control P3. As an example, if P2 is –3 (253) and P3 is 5, the controls 5, 6 and 7 will be redrawed.
255 ⟶ Redraws all controls inside the window content.
04 1W Area X begin inside the window content.
06 1W Area Y begin.
08 1W Area X length.
10 1W Area Y length.
⟶ If P2 is between 240 and 254:
03 1B ID of the first control, which should be redrawed.

ID: 051 (MSC_DSK_WINSIN) –
– Window_Redraw_SubControl_Command
Description: This command works like MSC_DSK_WINDIN, but it updates only one sub control inside a control collection. This command currently doesn't support the redrawing of multiple sub controls. For additional information see also MSC_DSK_WINDIN. This command works always.
Library: SyDesktop_WINSIN.
Message: 00 1B 051.
01 1B Window ID.
02 1B Control collection ID.
03 1B ID of the sub control inside the control collection.

5.2.1 – Desktop Manager Responses

ID: 160 (MSR_DSK_WOPNER) – Window_OpenError_Response
Description: The window couldn't be opened, because the maximum number of windows (32) has already been reached.
Message: 00 1B 160.

ID: 161 (MSR_DSK_WOPNOK) – Window_OpenOK_Response
Description: The window has been opened. The desktop manager sends back its ID. For all following commands regarding the new window you will need this ID.
Message: 00 1B 161.
04 1B Window ID.

ID: 162 (MSR_DSK_WCLICK) – Window_UserAction_Response
Description: The desktop manager is sending this message to the application, if the user has done an interaction with the window or the controls inside the window.
Message: 00 1B 162
01 1B Window ID
02 1B Action type
- 05 – Close button has been clicked or ALT+F4 has been pressed (DSK_ACT_CLOSE).
- 06 – Menu entry has been clicked (DSK_ACT_MENU). P8 will contain the menu entry value.
- 14 – A control of the window content has been clicked and/or modified with the keyboard or mouse (DSK_ACT_CONTENT). P8 will contain the control value, P4/6 the mouse position, if the user used the mouse.
- 15 – A control of the window toolbar has been clicked and/or modified with the keyboard or mouse (DSK_ACT_TOOLBAR). P8 will contain the control value, P4/6 the mouse position, if the user used the mouse.
- 16 – User has pressed a key without modifying any control (DSK_ACT_KEY). P4 will contain the ASCII code.

→ If P2 is 14 or 15:
03 1B Action sub specification
- 00 – Left mousebutton clicked (DSK_SUB_MLCLICK).
- 01 – Right mousebutton clicked (DSK_SUB_MRCLICK).

02 – Left mousebutton double clicked (DSK_SUB_MDCLICK).
03 – Middle mousebutton clicked (DSK_SUB_MMCLICK).
07 – Key has been pressed (DSK_SUB_KEY)

→ If P2 is 14 or 15 and P3 is between 0 and 3:
04 1W Mouse X position (inside the window content/toolbar).
06 1W Mouse Y position.
→ If P2 is 14 or 15 and P3 is 7, or if P2 is 16:
04 1B ASCII code of the pressed key. For information about extended ASCII codes, see the chapter "device manager", EXTENDED ASCII CODES.
→ If P2 is 6, 14 or 15:
08 1W Menu entry value or control value.

ID: 163 (MSR_DSK_DSKSRV) – Desktop_Service_Response
Description: Please read the desktop manager service description below for more information.
Message: 00 1B 163.
01 1B Service ID.
02 –05 See desktop manager service description below.

ID: 164 (MSR_DSK_WFOCUS) – Window_Focus_Response
Description: The desktop manager is sending this message to the application, if the focus status of a window changed.
Message: 00 1B 164.
01 1B Window ID.
02 1B Status: 0 → Window lost focus position.
1 → Window received focus position.

ID: 165 (MSR_DSK_CFOCUS) – Control_Focus_Response
Description: The desktop manager is sending this message to the application, if another control inside a window got the focus. Please note, that the control ID is not the value of the control but its number inside the control group (starting with 1).
Message: 00 1B 165.
01 1B Window ID.

02 1B ID of the new focus control (starting with 1).
03 1B Reason for focus change:
0 ⟶ User clicked the control via mouse or used the mouse wheel.
1 ⟶ User pressed the tab key.

ID: 166 (MSR_DSK_WRESIZ) – Window_Resize_Response
Description: The desktop manager is sending this message to the application, if the user resized the window. This may happen when it has been maximized, restored or resized by keyboard or mouse. Please note, that this message will also be sent, if the user maximizes or restores a window, which was minimized before.
Message: 00 1B 166.
01 1B Window ID.

ID: 167 (MSR_DSK_WSCROL) – Window_Scroll_Response
Description: The desktop manager is sending this message to the application, if the user scrolled the content of the window.
Message: 00 1B 167
01 1B Window ID.

5.2.2 – Desktop Manager Services

Most parts of the device manager can't be accessed by an application directly. All video screen related things will be handled by the desktop manager. Because of this there are the desktop services, which allow an application to change some video screen parameters. Also some more services are offered.

ID: 001 (DSK_SRV_MODGET) – DesktopService_ScreenModeGet
Description: Returns the current screen resolution and number of possible colours.
Library: SyDesktop_MODGET.
Message: 00 1B 048.
01 1B 001.
Response: 00 1B 163.
01 1B 001.

02 1B Screen mode; the available modes depend on the computer platform.
PCW 0 – 720 X 255, 2 colours (PCW standard mode).
CPC 1 – 320 X 200, 4 colours (CPC standard mode).
2 – 640 X 200, 2 colours.
EP 1 – 320 X 200, 4 colours (EP standard mode).
2 – 640 X 200, 2 colours.
MSX 5 – 256 X 212, 16 colours.
6 – 512 X 212, 4 colours.
7 – 512 X 212, 16 colours (MSX standard mode).
G9K 8 – 384 X 240, 16 colours.
9 – 512 X 212, 16 colours (G9K standard mode).
10 – 768 X 240, 16 colours.
11 – 1024x 212, 16 colours.
⟶ If G9K:
03 1B Virtual desktop width.
0 – No virtual desktop.
1 – 512.
2 – 1000.

ID: 002 (DSK_SRV_MODSET) – DesktopService_ScreenModeSet
Description: Sets the screen resolution and number of possible colors.
Library: SyDesktop_MODSET
Message: 00 1B 048
01 1B 002
02 1B Bit0~6 Screen mode (the available modes depend on the computer platform):
PCW 0 – 720 X 255, 2 colours (PCW standard mode).
CPC 1 – 320 X 200, 4 colours (CPC standard mode).
2 – 640 X 200, 2 colours.
EP 1 – 320 X 200, 4 colours (EP standard mode).
2 – 640 X 200, 2 colours.
MSX 5 – 256 X 212, 16 colours.
6 – 512 X 212, 4 colours.
7 – 512 X 212, 16 colours (MSX standard mode).
G9K 8 – 384 X 240, 16 colours.
9 – 512 X 212, 16 colours (G9K standard mode).
10 – 768 X 240, 16 colours.
11 – 1024x 212, 16 colours.

→ If G9K:
03 1B Virtual desktop width.
0 – No virtual desktop.
1 – 512.
2 – 1000.

Response: The desktop manager does not send a response message.

ID: 003 (DSK_SRV_COLGET) – DesktopService_ColourGet

Description: Returns the definition of a colours. Please note, that you always have a range of 4096, even if the computer is not a CPC PLUS, as the system recalculates the colour for the other machines.

Library: SyDesktop_COLGET.

Message:
00 1B 048.
01 1B 003.
02 1B Colour number (0~15).

Response:
00 1B 163.
01 1B 003.
02 1B Colour number (0~15).
03 1B bit0~3 → Blue component (0~15).
bit4~7 → Green component (0~15).
04 1B bit0~3 → Red component (0~15).

ID: 004 (DSK_SRV_COLSET) – DesktopService_ColourSet

Description: Defines one colour. Please note, that you always have a range of 4096, even if the computer is not a CPC PLUS, as the system recalculates the colour for other machines.

Library: SyDesktop_COLSET.

Message:
00 1B 048.
01 1B 004.
02 1B Colour number (0~15)
03 1B bit0~3 → Blue component (0~15).
bit4~7 → Green component (0~15).
04 1B bit0~3 → Red component (0~15).

Response: The desktop manager does not send a response message.

ID: 008 (DSK_SRV_DSKBGR) – DesktopService_RedrawBackground

Description: Reinitialize and redraws the desktop background.

Library: SyDesktop_DSKBGR.

Message: 00 1B 048.
01 1B 008.
Response: The desktop manager does not send a response message.

ID: 009 (DSK_SRV_DSKPLT) – DesktopService_RedrawComplete
Description: Reinitialize the desktop background and redraws the complete screen. The background, the task bar and all windows will be updated.
Library: SyDesktop_DSKPLT
Message: 00 1B 048
01 1B 009
Response: The desktop manager does not send a response message.

5.2.3 – Desktop Manager Functions

The desktop manager functions have to be called with RST 20H (BNKSCL).

BUFPUT (814EH) – Clipboard_Put
Description: Copies data into the clipboard. If the clipboard already contained data, it will be deleted first.
How to call: rst 20H : dw 814EH
Input: IX – Source data address.
E – Source data RAM bank (0~15).
IY – Length of source data.
D – Type of source data.
1 – Text.
2 – Graphic (extended).
3 – Item list (format not yet defined).
4 – Desktop icon shortcut.
Output: CY – Error state (0 ⟶ Ok, 1 ⟶ Memory full)
Registers: AF, BC, DE, HL.

BUFGET (8151H) – Clipboard_Get
Description: Copies data from the clipboard to the destination memory area. This will only be done, if the clipboard contains data of the requested type and if the data inside the clipboard is not larger than the destination area.

How to call: rst 20H : dw 8151H
Input: IX – Destination address.
E – Destination RAM bank (0~15).
IY – Maximum length of destination area.
D – Type of required data.
Output: CY – Error state.
0 ⟶ Ok (IY – Length of copied data).
1 ⟶ Error: A – 0 ⟶ Clipboard is empty,
1 ⟶ Wrongdata type,
2 ⟶ Data is too large.
Registers: AF, BC, DE, HL.

BUFSTA (8154H) – Clipboard_Status
Description: Reads the status of the clipboard (data type and length).
The address and bank of the data is returned as well,
though an application shouldn't access it directly, as it
may be changed by another process in the meantime.
How to call: rst 20H : dw 8154H
Input: –
Output: D – Data type (0 – clipboard is empty).
IY – Data length.
IX – Data address.
E – Data RAM bank (0~15).
Registers: –

5.2.4 – Desktop Manager Data Records

If "recalculation" for a control group is activated every coordinate and size value of a control will be recalculated, if the user changes the size of the window. The calculation is:
position or size – Static_part + window_size * multiplier / divider

5.2.4.1 – Window Data Record

00 1B Status (0=closed, 1=normal, 2=maximized, 3=minimized,
+128=open window centered, will be always reset after opening).

01 1B bit0: Display 8x8 pixel application icon (in the upper left edge).
bit1: Window is resizeable.
bit2: Display close button.
bit3: Display tool bar (below the menu bar).
bit4: Display title bar.
bit5: Display menu bar (below the title bar).
bit6: Display status bar (at the lower side of the window).
bit7: Used internally (set to 0).
02 1B bit0: Adjust X size of the window content to the X size of the window.
bit1: Adjust Y size of the window content to the Y size of the window.
bit2: Window will not be displayed in the task bar.
bit3: Window is not moveable.
bit4: Window is a modal window: other windows, who point on it (see byte 51), can't get the focus position.
bit5: Reserved (set to 0).
bit6: Used internally (set to 0).
bit7: Used internally (set to 0).
03 1B Process ID of the windows owner
04 2W X/Y position, if window is not maximized
08 2W X/Y size, if window is not maximized.
12 2W X/Y offset of the displayed window content.
16 2W Full X/Y length of the total window content.
20 2W Minimal possible X/Y size of the window.
24 2W Maximal possible X/Y size of the window.
28 1W Address of the application icon (graphic object).
30 1W Address of the title line text (terminated by 0).
32 1W Address of the status line text (terminated by 0).
34 1W Address of the MENU DATA RECORD.
36 1W Address of the CONTROL GROUP DATA RECORD of the window content.
38 1W Address of the CONTROL GROUP DATA RECORD of the tool bar content.
40 1W Height of the tool bar.
42 9B Used during runtime, so it has to be reserved.
51 1B "0" or number of modal window + 1.
52 140B Used during runtime, so it has to be reserved.

5.2.4.2 – Control Group Data Record

00 1B Number of controls (has to be >0; notice that you have to fill the background of the form by yourself, too!)
01 1B Process ID of the control group owner
02 1W Address of the CONTROL DATA RECORDS
04 1W Address of the position/size CALCULATION RULE DATA RECORD (0 means, no re–calculation)
06 2B Not used, set to 0.
08 1B Object to click, when user hits return (1~255, 0=not defined; works only for window content, not for the toolbar)
09 1B Object to click, when user hits escape (1~255, 0=not defined; works only for window content, not for the toolbar)
10 4B Reserved, set to 0.
14 1B Focus object (1~255, 0=no focus on any object; only for window content)
15 1B Not used, set to 0.

5.2.4.3 – Control Data Records

[Number of controls] * [
00 1W Control ID/value; this will be sent to the application, if the user clicks or modifies the control. As an example you could store the address of a sub routine here, which you call, if the user clicks the control.
02 1B CONTROL TYPE; for the type IDs see below. The IDs are between 0 and 63. IDs > 63 will be ignored, so you can set bit 6 and/or 7 to 1, if you want to hide an object, and reset it to 0 if you want to show it again.
03 1B Bank number, where the extended control data record is located (0~15); "–1" means, that the control is placed in the same bank like the window data record.
04 1W Either a parameter to specify the control properties or, if one word is not enough, a pointer to the extended control data record; this depends on the control, so see the control description for information, what to write here.
06 2W X/Y position of the control (related to the upper left edge of the content or tool bar); if the window is using a CALCULATION RULE DATA RECORD, you can write 0 here.

10 2W X/Y size of the control (related to the upper left edge of the content or tool bar); if the window is using a CALCULATION RULE DATA RECORD, you can write 0 here
14 2B Not used, set to 0.
]

5.2.4.4 – Calculation Rule Data Record

00 1W X position (static part).
02 1B Window X size multiplier.
03 1B Window X size divider.
04 1W Y position (static part).
06 1B Window Y size multiplier.
07 1B Window Y size divider.
08 1W X size (static part).
10 1B Window X size multiplier.
12 1B Window X size divider.
13 1W Y size (static part).
14 1B Window Y size multiplier.
15 1B Window Y size divider.

5.3 – CONTROL TYPES

5.3.1 – Paint

ID: 00 (PLF) – paint_area
Description: Fills an area with a specified colour.
Parameter: bit0–3: Pen.
bit7: Colour mode:
0 ⟶ 4 colour indexed, 1 ⟶ 16 colour.
Data record: –
Size: Not limited.

ID: 01 (PLT) – paint_text
Parameter: Pointer to data record.
Data record: 00 1W Text address (terminated by 0).
03 1B bit0–1: Alignment (0=left, 1=right, 2=center).
bit5: If 1, don't prepare background (MSX only).
bit7: Colour mode:
0 ⟶ 4 colour indexed, 1 ⟶ 16 colour.

→ If 4 colour mode:
02 1B bit0–1: Paper, bit2–3: Pen,
bit7: If 1, fill background.
→ If 16 colour mode:
02 1B bit0–3: Paper, bit4–7: Pen.
03 1B bit6: If 1, fill background.

Size: Width is not limited, height must be equal like the height of the current font; if the text is larger than the control width, it will only be cut, if the "fill background" option is activated.

ID: 02 (PLR) – paint_frame

Description: Plots a text with the standard system font with 4 or 16 colours for background and foreground. If "fill background" is activated first the whole area of the control will be filled with the paper–colour, and the text will be clipped to the defined area. Otherwise it would exceed the area, if it's too long. Iif the background has already been filled with the paper colour before, bit 5 of byte 3 can be used to increase the performance on the MSX platform.

Parameter: bit7: Colour mode:
0 → 4 colour indexed, 1 → 16 colour.
bit6: If 1, fill area inside frame.
→ If 4 colour mode:
bit4–5: Pen of area inside frame (only used, if bit6=1).
bit0–1: Pen of upper and left line.
bit2–3: Pen of lower and right line.
→ If 16 Colour mode:
bit0–3: Pen of area inside frame (only used, if bit6=1).
bit8–11: Pen of upper and left line.
bit12–15: Pen of lower and right line.

Data record: –

Size: Equal or greater than 3x3.

ID: 03 (PLX) – paint_frame_with_title

Description: Plots a frame with a text title. Notice, that the lines have a distance of 3 pixels to the border of the control. The area inside the frame will not be filled.

Parameter: Pointer to data record.
Data record: 00 1W Text address (terminated by 0)
02 1B bit7: Colour mode:
0 ⟶ 4 colour indexed, 1 ⟶ 16 colour.
⟶ If 4 colour mode:
02 1B bit0–1: Indexed paper of text;
bit2–3: Indexed pen of text and line.
⟶ If 16 colour mode:
02 1B bit0–3: Pen of line.
03 1B bit0–3: Paper of text;
bit4–7: Pen of text.
Size: Equal or greater than 16x16.

ID: 04 (PLP) – paint_progress
Description: Plots a progress bar. The second byte of the parameter specifies the progress in 1/255 steps.
Parameter: bit0–1: Indexed colour of upper and left line.
bit2–3: Indexed colour of lower and right line.
Bit4–5: Indexed colour of filled area inside frame.
bit6–7: Indexed colour of empty area inside frame.
bit8–15: Progress (0=0%, 255=100%).
Data record: –
Size: Equal or greater than 3x3.

ID: 05 (PLA) – paint_text_with_alternative_font
Description: Plots a text with an self specified alternative font. The font must be placed in the same 16K area and RAM bank like the text. For the description how a font is stored in the memory see below (FONTS). If "fill background" is activated first the whole area of the ontrol will be filled with the paper–colour.
Parameter: Pointer to data record.
Data record: 00 1W Text address (terminated by 0).
02 1B bit0–1: Paper, bit2–3: Pen (if 4 colour mode).
bit0–3: Paper, bit4–7: Pen (if 16 colour mode).
03 1B bit0–1: Alignment (0=left, 1=right, 2=center).
bit7: Colour mode:
0 ⟶ 4 colour indexed, 1 ⟶ 16 colour.
04 1W Font address.

Size: Width is not limited, height must be equal like the height of the current font; if the text is larger than the control width, it will only be cut, if the "fill background" option is activated.

ID: 06 (PLC) – paint_text_with_control_codes

Description: Plots a text, which can include control codes (0–31). The following control codes are currently accepted:
- 00 – End of text
- 01 – Set text colour
  Parameters: 1byte (bit0–3=paper, bit4–7Pen)
- 02 – Set font
  Parameters: 1word (font address; must be placed in the same 16K area and RAM bank like the text; if the address is –1, the standard font will be used)
- 03 – Switch underline mode on
- 04 – Switch underline mode off
- 05 – Insert additional space between the current and the next char
  Parameters: 1byte (amount of pixels)
- 06 to 07 – *not yet supported* (will be ignored)
- 08 to 11 – Skip next bytes ((code–8)*2+1 bytes)
- 12 to 31 – Insert additional space between the current and the next char (code–8 pixels)

Parameter: Pointer to data record

Data record:
- 00 1W Text address (terminated by 0)
- 02 1W Maximum number of bytes (control codes included)
- 04 1W Font address (–1=Standard)
- 06 1B bit0–3: paper, bit4–7: Pen
- 07 1B [bit0] =if 1, underlined

Size: Not limited

5.3.2 – Graphics

ID: 08 (ICN) – Graphic_simple

Description: Plots a graphic. For the description how a graphic object is stored in the memory see below (GRAPHICS, "Standard graphics"). The control must have the same size like the graphic.

Parameter: Graphic address.
Data record: –
Size: Same as the graphic object.

ID: 09 (ICT) – Graphic_with_text
Description: Plots a graphic with one or two textlines below. It is used for displaying icons. When there is a 0 instead of a text address, the line will stay empty. The graphic itself must have a size of 24x24.
Parameter: Pointer to data record.
Data record:
00 1W Graphic address (standard graphic) or address of the graphic header (extended graphic).
02 1W "0" or address of text for line 1 (terminated by 0).
04 1W "0" or address of text for line 2 (terminated by 0).
06 1B bit4: Graphic mode (0 – Standard, 1 – extended).
bit5: Text colour mode:
0 ⟶ 4 colour indexed, 1 ⟶ 16 colour.
bit6: Flag, if extended options.
bit7: Flag, if icon can be moved by the user.
⟶ If 4 colour text mode:
06 1B bit0–1: Paper, bit2–3: Pen.
⟶ If 16 colour text mode:
07 1B bit0–3: Paper, bit4–7: Pen.
⟶ If extended options:
08 1B bit0: Flag, if this icon can be marked.
bit1: Flag, if this icon is marked.
Size: 48x40.

ID: 10 (ICX) – Graphic_extended
Description: Plots a graphic with an extended header. For the description how a graphic object is stored in the memory see below (GRAPHICS, "Graphics with extended header"). The control must have the same size like the graphic.
Parameter: Address of the graphic header.
Data record: –
Size: Same as the graphic object.

5.3.3 – Buttons

ID: 16 (BTN) – button_simple
Description: Plots a button with a centered text inside. Indexed colour 2 is used for the background, indexed colour 1 for text colour and right/lower lines, indexed colour 3 for left/upper lines.
Parameter: Text address (terminated by 0).
Data record: –
Size: Width is not limited, height must always be 12.

ID: 17 (BTC) – button_check
Description: Plots a check box followed by a textline. The status byte contains 1, if the box is checked, otherwise it contains 0.
Parameter: Pointer to data record.
Data record:
00 1W Address of status byte (this byte can be 0 or 1)
02 1W text address (terminated by 0)
04 1B bit0–1: Indexed text paper;
bit2–3: Indexed text pen.
Size: Width is not limited, height must always be 8.

ID: 8 (BTR) – button_radio
Description: Plots a radio button followed by a textline. If the global status byte has the same value as the own status, this radio button is checked. The 4byte coordinate buffer has to contain –1,–1,–1,–1 at the beginning. It stores the coordinates of the current checked radio button. Radio buttons, which are connected to each other, have to point to the same global status byte and the same coordinate buffer.
Parameter: Pointer to data record.
Data record:
00 1W Address of global status byte.
02 1W Text address (terminated by 0).
04 1B bit0–1: Indexed text paper,
bit2–3: Indexed text pen.
05 1B Value of the own status.
06 1W Pointer to a global 4-byte coordinate buffer.
Size: Width is not limited, height must always be 8.

ID: 19 (BTP) – button_hidden
Description: This just defines an area on which the user can click. Nothing will be displayed.
Parameter: –
Data record: –
Size: Not limited.

ID: 20 (BTT) – button_tabs
Description: Plots a tab line. If –1 is set as the width of one tab title the system will calculate the needed width by itself and overwrites the –1 with the correct value. As soon as the text of a tab is changed the application has to set the value to –1 again.
Parameter: Pointer to data record.
Data record:

00	1B	Number of tabs
01	1B	bit0–1: Indexed paper, bit2–3: Indexed pen, bit4–5: Indexed colour of left/upper lines, bit6–7: Indexed colour of right/lower lines.
02	1B	Selected tab.
03	1W	Text address of tab 1 title (terminated by 0).
05	1B	–1 or width of tab 1 title.
06	1W	Text address of tab 2 title (terminated by 0).
08	1B	–1 or width of tab 2 title.
⋮		
??	1W	Text address of tab n title.
??	1B	–1 or width of tab n title.

Size: Width is not limited, height must always be 11.

5.3.4 – Miscellaneous

ID: 24 (SLD) – Slider_simple
Description: Plots a slider. It can be used to control a value or to move inside a window or list.
Parameter: Pointer to data record.
Data record:

00	1B	bit0: Alignment (0=vertical, 1=horizontal). bit1: 0=value control, 1=window section control. bit7: Reserved for internal use, set to 0.

01 1B Not used, set to 0.
02 1W Current value/position
04 1W Maximum value/position (range is 0 – maximum)
06 1B Value increase, if the user clicks the down/left button
07 1B Value decrease, if the user clicks the up/right button

Size: Depending on the alignment, one component must have a minimum of 24 pixels; the other one must be always 8.

ID: 25 (SUP) – control_collection

Description: Plots a collection of sub controls. A control collection behaves like a sub content inside the content of a window.
Parameter: Pointer to data record.
Data record: 00 1W Pointer to sub control group data record.
02 1W Full width of the control collection area.
04 1W Full height of the control collection area.
06 1W Current X offset.
08 1W Current Y offset.
10 1B bit0: Flag, if X slider should be displayed, bit1: Flag, if Y slider should be displayed.
Size: If sliders are activated, the size must be more than 32x32; there are no other limitations.

5.3.5 – Textinput

ID: 32 (TXL) – textinput_line

Description: Plots a textinput line. The user can use several key functions for editing the text (see below) as well as a context menu, which opens on right mouseclick. If the user modified the text, bit 7 of byte 12 of the data record will be set to 1.
Parameter: Pointer to data record.
Data record: 00 1W Address of text (has to be large enough, see below;text has to be placed anywhere inside a 16K aligned data area).
02 1W First displayed character.
04 1W Cursor position.

06 1W Number of selected characters (0 ⟶ no selection, <0 ⟶ cursor is placed at the end of the selection, >0 ⟶ cursor is placed at the beginning of the selection).
08 1W Length of the current text.
10 1W Possible maximum text length (doesn't include the 0 terminator at the end of the text).
12 1B bit0: Flag, if Password (all chars will be displayed as '*').
bit1: Text is read only.
bit2: Use alternative colours.
bit7: Will be set to 1, if text has been modified.
⟶ If usage of alternative colours:
13 1B bit0–3: Text paper.
bit4–7: Text pen.
14 1B bit0–3: Pen of upper and left line.
bit4–7: Pen of lower and right line.

Size: Width is not limited, height must always be 12.

Key functions:

SHFT+LEFT/RIGHT	(De)select parts of the text.
CTRL+LEFT/RIGHT	Jump word wise left/right.
CTRL+UP/DOWN	Jump to line begin/end.
CTRL+A	Select the complete text.
CTRL+C	Copy selected text.
CTRL+X	Cut selected text (copy and delete).
CTRL+V	Paste copied text.

ID: 33 (TXB) – textinput_box

Description: Plots a textinput box. If the user modified the text, bit 7 of byte 12 of the data record will be set to 1.

Parameter: Pointer to data record.

Data record: 00 1W Address of text (has to be large enough and has to be placed anywhere inside a 16K aligned data area). See below:
02 1W Not used.
04 1W Cursor position (inside the complete text)
06 1W Number of selected characters (0 ⟶ no selection, <0 ⟶ cursor is placed at the end of the selection, >0 ⟶ cursor is placed at the beginning of the selection)

08 1W Length of the current text
10 1W Possible maximum text length (doesn't include the 0 terminator at the end of the text)
12 1B bit1: Text is read only
bit2: Use alternative colours
bit3: Use alternative font
bit7: Will be set to 1, if text has been modified
13 1B bit0–3: Text paper, bit4–7: Text pen (only when using alternative colours)
14 1B Not used.
15 1W Font address (only when using alternative font)
17 1B Reserved, set to 0.
18 1W Current number of lines
20 1W Maximum pixel width of one line for word wrapping (–1 → unlimited).
22 1W Maximum number of lines;
24 1W Used internally: X size of visible area. (–8=force reformatting)
26 1W Used internally: Y size of visible area.
28 1W Address of this data record
30 1W Used internally: Total X size.
32 1W Used internally: Total Y size.
34 1W Used internally: X offset of visible area.
36 1W Used internally: Y offset of visible area.
38 1B bit0: Word wrapping (0=at window border, 1=at maximum pixel position, see byte20)
bit1: 1 (has to be set always)
39 1B Tab stop width (1~255; 0=no tab stop)
40 4B Message buffer for additional control commands
44 4B Reserved, set to 0.
48 [maximum number of lines]W
Line length table; this table contains a word for each line with the length in chars; that may also include potential carriage return/line feed (CR+LF) codes at the end of a line; bit15 is set, if a line contains the CR+LF codes.

Key functions: SHFT+LEFT/RIGHT (De)select parts of the text.
CTRL+LEFT/RIGHT Jump word wise left/right.

CTRL+UP/DOWN Jump to line begin/end.
CTRL+A Select the complete text.
CTRL+C Copy selected text.
CTRL+X Cut selected text (copy and delete).
CTRL+V Paste copied text.

Commands: The textinput box control provides additional functions, which can be accessed by sending special keyboard codes to the control. This is done by using the KEYPUT function (see Device Manager documentation) while the control has focus position. If a command requires additional parameters, they have to be stored at byte 40 in the data record before sending the code. Here you will also find the results, if the command returns. The following commands are available:

Code 29: Get cursor position; this command returns the current cursor position
Output: (buffer+0)=column (starting at 0)
(buffer+2)=line (starting at 0)

Code 30: Text has been modified; this command forces the control to reformat and update the text.

Code 31: Set cursor position and text selection; the visible area of the textinput box will be scrolled to the new position, if necessary.
Input: (buffer+0)=new cursor position
(buffer+2)=new number of selected chars

5.3.6 – Lists

ID: 40 (LST) – List_title

Description: Plots the title line of a list.
Parameter: Pointer to data record.
Data record:
00 1W Number of lines
02 1W First displayed line of the list
04 1W Pointer to data record for the list content
06 2B Not used, set to 0.
08 1B Number of columns (1~64).
09 1B bit0–5: Index of sorted column.
bit6: Sort list on start.
bit7: Sort order (0=ascending, 1=descending).

10 1W Pointer to data record for the columns.
12 1W Last clicked line.
14 1B bit0: Flag, if list slider will be displayed.
bit1: Flag, if multiselections are possible.
15 1B Not used, set to 0.

Column record: [Number of columns] * [
00 1B bit0–1: Allignment (0=left, 1=right, 2=center)
bit2–3: Type (0=text, 1=graphic, 2=16-bit number, 3=32-bit number)
01 1B Not used, set to 0.
02 1W Width of this column in pixel.
04 1W Text address of the title (terminated by 0).
06 2B Not used, set to 0.
]

List record: [Number of lines] * [
1W bit0–12: Value of this line
bit13: Colour of the first row (1=alternative)
bit14: Set to 0, it is internally used for "selection update".
bit15: Flag, if this line is marked.
[Number of columns] *
1W Text/data address or value for this cell.
]

Size: Width is not limited, height must always be 10.

ID: 41 (LSI) – List_content
Description: Plots the list itself without the title.
Parameter: Pointer to data record
Data record: See ID 40.
Size: Width must be equal or larger than 11, height must be equal or larger than 16

ID: 42 (LSP) – List_dropdown
Description: Plots a dropdown list. Only one line of the list will be displayed. If the user clicks on this control, the complete list will drop down and the user can choose one of the entries.
Parameter: Pointer to data record.

Data record: See ID 40.
12 1W Last clicked line (this always represents the selected line).
14 1B bit0: Flag, if list slider will be displayed (should be set to 1, if list has more than 10 entries).
bit1: Flag, if multiselections are possible (always set to 0)
Size: Width must be equal or larger than 11, height must always be 10.

ID: 43 (LSC) – List_complete
Description: Plots the list title and the list itself together. This is the combination of ID 40 and ID 41.
Parameter: Pointer to data record.
Data record: See ID 40.
Size: Width must be equal or larger than 11, height must be equal or larger than 26

5.3.7 – Pulldown Menus

You can define up to 8 sub menu levels. The WINDOW DATA RECORD points to the highest menu level. These are the entries you see in the menu bar of a window. These entries usually point to their sub menus, which contain entries, too, which are clickable or which point to an additional sub menu again.

00 1W Number of entries
[Number of entries] * [
00 1W bit0: Flag, if the menu entry is active. Deactivated entries can't be clicked by the user and will appear in a different colour.
bit1: Flag, if there is a check mark behind the entry.
bit2: Flag, if the entry opens a sub menu.
bit3: Flag, if there is no entry but a separator line.
02 1W Text address (terminated by 0). If bit3 of the previous word is set, you have to use 0 here.
04 1W Value, if the entry is clickable, or address of the sub menu data record, if bit2 of the first word is set.
06 1W Reserved, set to 0.
]

5.4 – FONTS AND GRAPHICS

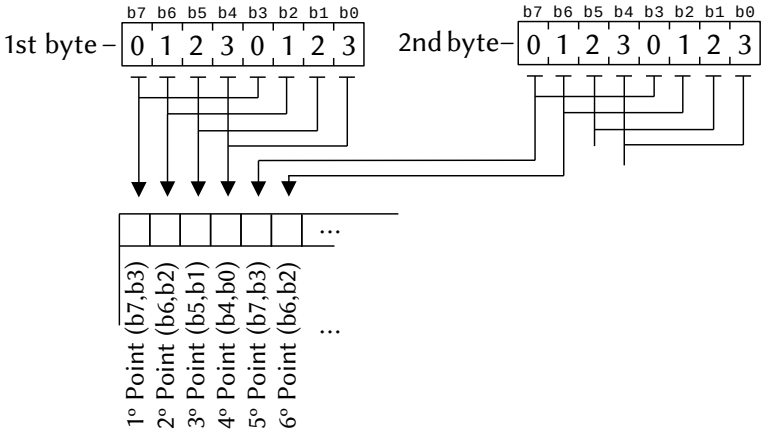
5.4.1 – Standard graphics

A SymbOS standard graphic has 4 colours and can have a maximum size of 255x255 pixel. Each graphic object starts with a 3 byte header:

00 1B bit0–6: Width of the graphic in bytes.
bit7: Encoding type (0=CPC, 1=MSX).
01 1B Width of the graphic in pixel.
02 1B Height of the graphic in pixel.

Directly behind the header the amount of [Width_in_bytes] * [Height_in_pixel] bytes is following containing the graphic data. Every graphic is stored line by line like a sprite. The pixels have to be encoded in CPC format (Mode 1). Graphics on a MSX system will automatically be converted to the MSX format, when they are displayed the first time. bit 7 of header byte 0 contains the current encoding format. Please note, that it is not allowed to store an original graphic in MSX format, as a CPC system is not able to handle such graphics!

The following is a description of the CPC encoding format. Each byte contains 4 pixels:



Only applications, which have to modify a graphic after it has been displayed the first time, should take care about the encoding type and the MSX format.

5.4.2 – Graphics with extended header

As the width of a graphic is limited to 255 pixel, it wouldn't be possible to store a complete screen (like 320 X 200 in CPC Mode 1) in one piece. Such a screen needs to be splittet in two pieces (eg. 2 X 160 X 200), which makes it very difficult to write graphic modification routines.

Extended graphics do not have this limitation and also allow more than 4 colours. They can only be used for control ID 10, "graphic_extended". A graphic can be stored in one piece with a width of up to 1020 pixel. The control "graphic_extended" then is able to display a part of such a big linear stored graphic.

The extended header is build like this:

00 1B Width of the complete graphic in bytes (this has to be an even value!).
01 1B Width of the graphic area, which should be displayed, in pixel.
02 1B Height of the graphic area, which should be displayed, in pixel.
03 1W Address of the graphic data, including the area offset.
05 1W Address of the encoding information byte (see below). Please note: This single byte has ALWAYS to be placed directly in front of the complete graphic data!
07 1W Size of the complete graphic.
?? 1B Encoding information:
bit0–1: Colour encoding (0 → CPC, 1 → MSX).
bit2–3: Colour depth (0 → 4 colours, 1 → 16 colours).
Only the following two initial values are allowed:
0 → 4 colours, CPC format; an MSX system will convert the graphic to MSX format, when it is displayed the fist time.
5 → 16 colours, MSX format; a CPC and PCW system will render down the complete graphic to 4 colours (CPC format), when it is displayed the fist time.
??+1 X Graphic data.

The graphic header doesn't need to be stored directly in front of the graphic, it just needs to be located in the same 16K data area like the graphic itself. You can use this type of graphic:

→ If your graphic is larger than 255 pixel
→ If you only want to display a part of the graphic
→ If you don't want to store the header directly in front of the graphic
→ If you want to use 16 colour graphics

In any other case you should use standard graphics, as they are a little bit faster. The graphic itself ("graphic_data") is stored in one piece in memory (without header). Then we have two headers ("graphic_ header_for_area_1" and "graphic_header_for_area_2") which are poin-ting to two different areas of the big graphic.

5.4.3 – Fonts

A font defines the appearance of the characters used for printing texts in SymbOS. A font starts with a simple 2 byte header:

00	1B	Height of each character in pixel. This value can be between 1 and 15. The usual value is 8.
00	1B	First character in the font. This value can be between 0 and 255. To save memory the usual value is 32 (="space", the first printable ASCII char), as the first 32 chars normally won't be printed. Please note, that the SymbOS system font always starts with 32 and consists of 98 chars (32–129).

After the header the char definitions follow (1568 bytes):

00	1B	Width of the first char in pixel.
01	1B	bit mask of the 1st pixel line of the first char.
02	1B	bit mask of the 2nd pixel line of the first char.
⋮		
15	1B	bit mask of the 15th pixel line of the first char.
16	1B	Width of the second char in pixel.
17	1B	bit mask of the 1st pixel line of the second char.
⋮		

5.5 – SYSTEM MANAGER

The system manager is responsible for starting and stopping applications and for general system jobs. It provides several dialogue services and it owns the file manager, which can only be accessed via the system manager process (for more information see the "FILE MANAGER" chapter). System manager commands are triggered via a message, which has to be sent with RST 10H (MSGSND) to the system manager process. The system manager process always has the ID 3.

5.5.1 – Application Management

ID: 016 (MSC_SYS_PRGRUN – Program_Run_Command
Description: Loads and starts an application or opens a document with a known type by loading the associated application first. If bit 7 of P3 is not set, the system will open a message box, if an error occurs during the loading process.
Library: SySystem_PRGRUN.
Message:
00 1B 016.
01 1W File path and name address.
03 1B Bit0–3: File path and name RAM bank (0~15).
Bit7: Flag, if system error message should be suppressed.
Response: See MSR_SYS_PRGRUN

ID: 144 (MSR_SYS_PRGRUN) – Program_Run_Response
Description: The system manager sends this message after trying to load an application or after opening an associated document. If the operation was successful, you will find the application ID and the process ID in P8 and P9. If it failed because of loading problems P8 contains the file manager error code.
Message:
00 1B 144.
01 1B Success status.
0 – OK.
1 – File does not exist.
2 – File is not an executable and its type is not associated with an application.
3 – Error while loading (see P8 for error code).
4 – Memory full.

→ If success status is 0:
08 1B Application ID.
09 1B Process ID (the applications main process).
→ If success status is 3:
08 1B File manager error code.

ID: 017 (MSC_SYS_PRGEND) – Program_End_Command
Description: Stops an application and releases all its used system resources. Please note, that this command can't release memory, stop processes and timers or close windows, which are not registered for the application.
Library: SySystem_PRGEND
Message: 00 1B 017
01 1B Application ID.
Response: The system manager does not send a response message.

ID: 020 (MSC_SYS_PRGSTA) – Program_Run_Dialogue_Command
Description: Opens the "run" dialogue. The user then can select an application or a document.
Message: 00 1B 020.
Response: The system manager does not send a response message.

ID: 024 (MSC_SYS_PRGSET) – Program_Run_ControlPanel_Command
Description: Starts the control panel application or one of its two sub modules.
Message: 00 1B 024.
01 1B Control panel sub module.
0 → Main window.
1 → Display settings.
2 → Time and date settings.
Response: The system manager does not send a response message.

ID: 025 (MSC_SYS_PRGTSK) – Program_Run_TaskManager_Command
Description: Starts the task manager application.
Message: 00 1B 025.
Response: The system manager does not send a response message.

ID: 030 (MSC_SYS_PRGSRV) – Program_SharedService_Command
Description: Search, start and release shared services.

Message: 00 1B 030.
04 1B Command type:
0 ⟶ Search application or shared service.
1 ⟶ Search, start and use shared service.
2 ⟶ Release shared service.
⟶ If P4 is 0 or 1:
01 1W Address of the 12-byte application ID string.
⟶ If P4 is 0 or 1:
03 1B RAM bank (0~15) of the 12-byte application ID string.
⟶ If P4 is 2:
03 1B Application ID of shared service.
Response: See MSR_SYS_PRGSRV.

ID: 158 (MSR_SYS_PRGSRV) – Program_SharedService_Response
Description: Command type 0 ("search") will return 5 (not found) or 0 (OK). In the latter case you will find the application and process ID in P8 and P9. Command type 1 ("search, start and use") will return 0 (OK) if the shared services has been found or loaded successfully. In the other case it will return a loading error code of 1, 2, 3 or 4, which is identical with these of MSR_SYS_PRGRUN. Command type 2 ("release") will always return 0 (OK).
Message: 00 1B 158.
01 1B Result status:
0 ⟶ OK.
5 ⟶ Application or shared service not found (can only occur on command type 0).
1~4 ⟶ Error while starting shared service; same codes like in MSR_SYS_PRGRUN, please read there for a detailed description.
⟶ If command type was 0 or 1, and result status is 0:
08 1B Application ID of shared service.
09 1B Process ID (the applications main process).
⟶ If result status is 3:
08 1B File manager error code.

5.5.2 – System Management

The system manager will not send response messages after processing the following commands.

ID: 018 (MSC_SYS_SYSWNX) –
– System_Dialogue_NextWindow_Command
Description: Opens the dialogue for changing the current window. The next window is preselected. THIS COMMAND IS NOT IMPLEMENTED YET.
Message: 00 1B 018.

ID: 019 (MSC_SYS_SYSWPR) –
– System_Dialogue_PreviousWindow_Command
Description: Opens the dialogue for changing the current window. The previous window is preselected. THIS COMMAND IS NOT IMPLEMENTED YET.
Message: 00 1B 019.

ID: 021 (MSC_SYS_SYSSEC) –
– System_Dialogue_SystemSecurity_Command
Description: Opens the "SymbOS security" dialogue.
Message: 00 1B 021.

ID: 022 (MSC_SYS_SYSQIT) – System_Dialogue_ShutDown_Command
Description: Opens the "shut down" dialogue.
Message: 00 1B 022.

ID: 023 (MSC_SYS_SYSOFF) – System_ShutDown_Command
Description: Resets the computer.
Message: 00 1B 023.

ID: 028 (MSC_SYS_SYSCFG) – System_Configuration_Command
Description: Loads or saves the configuration or reinitializes the desktop background or the screen saver.
Message: 00 1B 028.
01 1B Action type:
0 ⟶ Reload configuration.
1 ⟶ Save current configuration.
2 ⟶ Reload/reinitialize desktop backg. picture.
3 ⟶ Reload or reinitialize screen saver.

5.5.3 – Dilogue Services

ID: 029 (MSC_SYS_SYSWRN) – Dialogue_Infobox_Command

Description: Opens an info, warning or confirm box and displays three line f text and up to three click buttons.

Library: SySystem_SYSWRN

Message:
- 00 1B 029
- 01 1W Content data address
- 03 1B Content data RAM bank (0~15)
- 04 1B bit0–2: Number of buttons (1–3)
  - 1 ⟶ “OK” button
  - 2 ⟶ “Yes”, “No” buttons
  - 3 ⟶ “Yes”, “No”, "Cancel" buttons
- bit3–5: Titletext
  - 0 ⟶ Default (bit7=[0]“Error!”/[1]“Info“)
  - 1 ⟶ “Error!”
  - 2 ⟶ “Info”
  - 3 ⟶ “Warning”
  - 4 ⟶ “Confirmation”
- bit6: Flag, if window should be modal window.
- bit7: Box type:
  - 0 ⟶ Default (warning [!] symbol).
  - 1 ⟶ Info (own symbol will be used).

Content data:
- 00 1W Address of text line 1.
- 02 1W 4 * (text line 1 pen) + 2.
- 04 1W Address of text line 2.
- 06 1W 4 * (text line 2 pen) + 2.
- 08 1W Address of text line 3.
- 10 1W 4 * (text line 3 pen) + 2.

⟶ If bit7 of P4 is 1:
- 12 1W Address of symbol (24x24px 4col SymbOS graphic format).

Response: See MSR_SYS_SYSWRN.

ID: 157 (MSR_SYS_SYSWRN) – Dialogue_Infobox_Response

Description: The system manager sends back this message to the application, when a infobox should be opened, or if the user clicked one of the buttons.

Message: 00 1B 157.
01 1B Message type:
0 ⟶ The infobox is currently used by another application. It can only be opened once at the same time, if it's not a pure info msg (one button, not a modal window). The user should close the other infobox first before it can be opened again by the app.
1 ⟶ The infobox has been opened successful as a modal window. This message won't be sent for non–modal window infoboxes.
2 ⟶ The user clicked "OK".
3 ⟶ The user clicked "Yes".
4 ⟶ The user clicked "No".
5 ⟶ The user clicked "Cancel" or close button.
⟶ If P1 is 1:
02 1B Number of the infobox window + 1. The application should store this number as the modal window ID of its own window, so that the infobox will be handled as the modal window of the application window. As long as it is open the application window can't get the focus position. For more information about the window data structure and modal windows see the chapter "desktop manager".

ID: 031 (MSC_SYS_SELOPN) – Dialogue_FileSelector_Command
Description: Opens the file selection dialogue. You can filter the entries of the directory by attributes and filename extension. We recommend always to set Bit3 of the attribute filter byte. The File mask/path/name string (260 bytes) must always be placed in the transfer RAM area (C000H~FFFFH).
Library: SySystem_SELOPN.
Message: 00 1B 031.
06 1B bit0–3: File mask, path and name RAM bank (0~15).
bit6: Flag, if "open" (0) or "save" (1) dialogue.
bit7: Flag, if file (0) or directory (1) selection.

07 1B Attribute filter:
bit0 = 1 ⟶ Don't show read only files.
bit1 = 1 ⟶ Don't show hidden files.
bit2 = 1 ⟶ Don't show system files.
bit3 = 1 ⟶ Don't show volume ID entries.
bit4 = 1 ⟶ Don't show directories.
bit5 = 1 ⟶ Don't show archive files.
08 1W File mask, path and name address (C000H–FFFFH).
00 3B File extension filter (e.g. "*").
03 1B 0.
04 256B Path and filename.
10 1W Maximum number of directory entries.
12 1W Maximum size of directory data buffer.

Response: See MSR_SYS_SELOPN.

ID: 159 (MSR_SYS_SELOPN) – Dialogue_FileSelector_Response

Description: The system manager sends back this message to the application, when a file selection dialogue should be opened. If opening was successful the application will first receive a type "–1" message and then, after the user choosed his file or aborted, a type 0 or 1 message. If opening failed the application will directly receive a type 2, 3 or 4 message.

Message: 00 1B 159
01 1B Message type
0 ⟶ The user choosed a file or directory and closed the dialogue with "OK". The complete file path and name can be found in the filepath buffer of the application.
1 ⟶ The user aborted the file selection. The content of the applications filepath buffer is unchanged.
2 ⟶ The file selection dialogue is currently used by another application. It can only be opened once at the same time. The user should close the dialogue first before it can be opened again by the application.

3 ⟶ Memory full. There was not enough memory available for the directory buffer and/or the list data structure.
4 ⟶ No window available. The desktop manager couldn't open a new window for the dialogue, as the maximum number of windows (32) has already been reached.
−1 ⟶ The dialogue has been opened successful and the user is doing his file selection right now.

⟶ If P1 is −1:

02 1B Number of the dialogue window + 1. The application should store this number as the modal window ID of its own window, so that the file selection dialogue will be handled as the modal window of the application window. As long as it is open the application window can't get the focus position. For more information about the window data structure and modal windows see the chapter "desktop manager".

5.5.4 – System Manager Functions

The system manager functions have to be called with RST 28H (BNKFCL).

SYSINF (8103H) – System_Information
Description: This function is mainly used by the task manager and the control panel application. Request types 0~2 are not documented yet.
How to call: ld hl,8103H : rst 28H
Input: E – Request type (see below):
0 ⟶ Get general information.
1 ⟶ Get application information.
2 ⟶ Get task information.
3 ⟶ Load mass storage device configuration.
4 ⟶ Save mass storage device configuration.
5 ⟶ Load a part of the configuration.

6 ⟶ Save a part of the configuration.
7 ⟶ Get config memory address.
8 ⟶ Get font and version string memory address.
D, IX, IY – Sub specification (see below).
Output: DE, IX, IY – Result data (see below).
Registers: AF, BC, HL.

Request type: 3 (Load mass storage device configuration).
Description: Loads the complete device configuration into the applications memory (8*16 bytes). For a description of the data structure please see "Configuration Data/Core Area Part/Mass Storage Devices".
Input: E = 3.
IX – Destination address (must be placed inside the transfer RAM area).
Output: –

Request type: 4 (Save mass storage device configuration).
Description: Saves the complete device configuration from the applications memory (8*16 bytes).
Input: E = 4.
IX – Source address (must be placed inside the transfer RAM area).
Output: –

Request type: 5 (Load a part of the configuration core area).
Description: Loads a part of the core area into the applications memory. For a description of the data structure please see "Configuration Data/Core Area Part".
Input: E = 5.
D – Number of bytes.
IX – Destination address (transfer RAM area).
IY – Source offset (starting from byte 163 [=system path] in the core part).
Output: –

Request type: 6 (Save a part of the configuration core area).
Description: Saves a part of the core area from the applications memory.

Input: E = 6.
D – Number of bytes
IX – Source address (transfer RAM area)
IY – Destination offset (starting from byte 163 [=system path] in the core part).
Output: –

Request type: 7 (Get config memory address).
Description: Sends back the address of the core area part (including the 6byte–header, so you have to add 6 to have the starting ddress), which is always placed in RAM bank 0, and the data area part together with the data area parts RAM bank number.
Input: E = 7.
Output: DE – Core area address (including 6byte–header; RAM bank 0).
IX – Data area address.
IYl – Data area RAM bank (0~8).

Request type: 8 (get font and version string memory address)
Description: Sends back the address and total size of the font, which is always placed in RAM bank 0, and the address of the version tring, which is placed in the same RAM bank like the data area part (see request type 7).
Input: E = 8.
Output: DE – Font address (RAM bank 0)
IX – Font length (=2 byte header + 98*16 byte char bitmaps)
IY – Address of the version information (this is placed in the data area RAM bank).
The version information has a length of 32 bytes:
00 1B Version Major.
01 1B Version Minor.
02 30B Version String (terminated by 0).

5.6 – FILE MANAGER

The file manager is owned by the system manager process, which is the only one, who is allowed to call file manager functions. If an application wants to use the file manager, it needs to send a special message to the system manager process, which includes all registers. The system manager then will call the specified file manager function and sends a message with the result back to the caller application. The system manager process always has the ID 3. Please note, that in SymbOS all texts must be terminated with a 0 byte. This is true for the pathes and filenames used in the file manager, too.

5.6.1 – System Manager Messages

ID: 026 (MSC_SYS_SYSFIL) – System_Filemanager_Command

Description: An application has to send this message to the system manager (process ID 3) to call a file manager function.

Message:

00	1B	026.
01	1B	File manager function ID.
02	1W	Input for AF.
04	1W	Input for BC.
06	1W	Input for DE.
08	1W	Input for HL.
10	1W	Input for IX.
12	1W	Input for IY

ID: 154 (MSR_SYS_SYSFIL) – System_Filemanager_Response

Description: The system manager sends this message back to the application, after the file manager function has been called.

Message:

00	1B	154
01	1B	File manager function ID
02	1W	Output for AF
04	1W	Output for BC
06	1W	Output for DE
08	1W	Output for HL
10	1W	Output for IX
12	1W	Output for IY

5.6.2 – Error Codes

Nearly all file–manager functions return the success status in the carry flag. If the carry flag is not set, the operation was successful. If it is set, an error occured. In this case, the A–register contains the error code number. The following is a list of all possible error codes.

000 – Device does not exist.
001 – OK.
002 – Device not initialised.
003 – Media is damaged.
004 – Partition does not exist.
005 – Unsupported media or partition.
006 – Error while sector read/write.
007 – Error while positioning.
008 – Abort while volume access.
009 – Unknown volume error.
010 – No free filehandler.
011 – Device does not exist.
012 – Path does not exist.
013 – File does not exist.
014 – Access is forbidden.
015 – Invalid path or filename.
016 – Filehandler does not exist.
017 – Device slot already occupied.
018 – Error in file organisation.
019 – Invalid destination name.
020 – File/path already exist.
021 – Wrong sub command code.
022 – Wrong attribute.
023 – Directory full.
024 – Media full.
025 – Media is write protected.
026 – Device is not ready.
027 – Directory is not empty.
028 – Invalid destination device.
029 – Not supported by file system.
030 – Unsupported device.

031 – File is read only.
032 – Device channel not available.
033 – Destination is not a directory.
034 – Destination is not a file.
255 – Undefined Error.

5.6.3 – Mass Storage Device Functions

ID: 000 (STOINI) – Storage_Init
Description: Removes all mass storage devices.
Input: –
Output: –
Registers: BC, DE, HL.

ID: 001 (STONEW) – Storage_New
Description: Adds a new mass storage device.
Input: A – Device (0~7).
C – Sub drive.
DE – Driver address.
L – Removeable media flag (1 ⟶ Removeable).
B – Drive letter ("A"–"Z").
IX – Device name (11 characters).
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 002 (STORLD) – Storage_Reload
Description: Reloads a mass storage device, if its "removeable media" status is activated. The format and the filesystem type will be loaded again.
Input: A – Device (0~7).
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 003 (STODEL) – Storage_Delete
Description: Removes an existing mass storage device.
Input: A – Device (0~7).
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL.

ID: 004 (STOINP) – Storage_ReadSector

Description: Reads a sector from a mass storage device (no memory banking).
Input: A – Device (0~7).
IY, IX – First sector number.
B – Number of sectors.
DE – Destination address.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 005 (STOOUT) – Storage_WriteSector

Description: Writes a sector to a mass storage device (no memory banking).
Input: A – Device (0~7).
IY, IX – First sector number.
B – Number of sectors.
DE – Source address.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 006 (STOACT) – Storage_Activate

Description: Loads the format and the file system type of a mass storage device.
Input: A – Device (0~7).
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 007 (STOINF) – Storage_Information

Description: Returns information about a mass storage device.
Input: A – Device (0~7).
Output: A – Type:
00 ⟶ Device does not exist.
01 ⟶ Device is ready.
02 ⟶ Device is not initialized.
03 ⟶ Device is corrupt.
B – Medium:
01 ⟶ Floppy disc single side (Amsdos, PCW).
02 ⟶ Floppy disc double side (FAT 12).
08 ⟶ RAM disc (*not supported yet*).
16 ⟶ IDE HD or CF card (FAT 12, FAT 16, FAT 32).

C – File system:
01 ⟶ Amsdos Data.
02 ⟶ Amsdos System.
03 ⟶ PCW 180K.
16 ⟶ FAT 12.
17 ⟶ FAT 16.
18 ⟶ FAT 32.
D – Sectors per cluster:
IY, IX – Total number of clusters.
Registers: E, HL.

ID: 08 (STOTRN) – Storage_DataTransfer
Description: Reads or writes a number of sectors (512 bytes) from/to the mass storage device. Sector 0 is the first sector of the partition of the device.
Input: A – Device (0~7).
IY, IX – First sector number.
B – Number of sectors.
C – Direction (0=read, 1=write).
HL – Source/destination address.
E – Source/destination RAM bank (0~15).
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

5.6.4 – File Management Functions

ID: 016 (FILINI) – File_Init
Description: Initialises the whole file manager. You should never call this function by yourself, as it resets everything!
Input: –
Output: –
Registers: AF, BC, DE, HL.

ID: 017 (FILNEW) – File_New
Description: Creates a new file and opens it for read/write access. If the file was already existing, it will be emptied first. The operation will be aborted, if the existing file is read only or an sub directory. For additional information see 018 (FILOPN).

Library: SyFile_FILNEW
Input: IXh – File path and name RAM bank (0~15)
HL – File path and name address.
A – Attributes:
bit0 = 1 ⟶ Read only.
bit1 = 1 ⟶ Hidden.
bit2 = 1 ⟶ System.
bit5 = 1 ⟶ Archive.
Output: A – Filehandler ID.
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: F, BC, DE, HL, IX, IY.

ID: 018 (FILOPN) – File_Open
Description: Opens an existing file for read/write access. You can open up to 7 different files at the same time. The media will be reloaded first, if the device is set to "removeable media" and there is no other open file on the same device.
Library: SyFile_FILOPN
Input: IXh – File path and name RAM bank (0~15).
HL – File path and name address.
Output: A – Filehandler ID.
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: F, BC, DE, HL, IX, IY.

ID: 019 (FILCLO) – File_Close
Description: Closes an opened file. If there is unwritten data in the sector cache, it will be written to disc at once. This command closes a file in any case, even if an error ocured. If an error occured during file reading/writing you must close the file, too, to make the filehandler free again!
Library: SyFile_FILCLO.
Input: A – Filehandler ID.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 020 (FILINP) – File_Input
Description: Reads a specified amount of bytes out of an opened file. After read byte. If you try to read more bytes than available, the zero flag will be reset. In any case BC contains the amount of read bytes (which could also be 0).

Library: SyFile_FILINP.
Input: A – Filehandler ID.
HL – Destination address.
E – Destination RAM bank (0~15).
BC – Number of bytes.
Output: BC – Number of read bytes.
Z = 1 ⟶ All requested bytes have been read.
0 ⟶ The end of the file has been reached, and less bytes than requested have been read (check BC).
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, DE, HL, IX, IY.

ID: 021 (FILOUT) – File_Output
Description: Writes a specified amount of bytes into an opened file. After this operation the file pointer will be moved behind the last written byte.
Library: SyFile_FILOUT.
Input: A – Filehandler ID.
HL – Source address.
E – Source RAM bank (0~15).
BC – Number of bytes.
Output: BC – Number of written bytes.
A = 0 ⟶ All bytes have been written.
1 ⟶ The device is full, and less bytes have been written (check BC).
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, DE, HL, IX, IY.

ID: 022 (FILPOI) – File_Pointer
Description: Moves the file pointer to another position. The difference is specified with IY and IX, IY is the high word, IX the low word (difference – 65 536 * IY + IX).
Ex.: IY=0, IX=1, C=1 ⟶ Increases the position by 1.
IY=65 535, IX=–10, C=2 ⟶ Sets the pointer before the last 10 bytes of the file.
Library: SyFile_FILPOI
Input: A – Filehandler ID.
IY, IX – Difference.

C – Reference point:
0 ⟶ File begin (difference is unsigned).
1 ⟶ Current pointer position (difference is signed).
2 ⟶ File end (difference is signed).

Output: IY, IX – New absolute pointer position.
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).

Registers: AF, BC, DE, HL.

ID: 023 (FILF2T) – File_Decode_Timestamp

Description: Decodes the file timestamp, which is used for the file system. You can use this function after reading the timestamp of a file with 035 (DIRPRR) or 038 (DIRINP).

Library: SyFile_FILF2T.

Input: BC – Time code:
bit 0– 4 – Second/2.
bit 5–10 – minute.
bit 11–15 – hour.
DE – Date code:
bit 0– 4 – Day (starting from 1).
bit 5– 8 – month (starting from 1).
bit 9–15 – year–1980.

Output: A – Second.
B – Minute.
C – Hour.
D – Day (starting from 1).
E – Month (starting from 1).
HL – Year.

Registers: F.

ID: 024 (FILT2F) – File_Encode_Timestamp

Description: Encodes the file timestamp, which is used for the file system. You can use this function before changing the timestamp of a file with 034 (DIRPRS).

Library: SyFile_FILT2F

Input: A – Second.
B – Minute.
C – Hour.
D – Day (starting from 1).
E – Month (starting from 1).
HL – Year.

Output: BC – Time code (see FILF2T)
DE – Date code (see FILF2T)
Registers: AF, HL, IX, IY.

ID: 025 (FILLIN) – File_LineInput
Description: Reads one text line out of an opened file. A text line is terminated by a single 13, a single 10, a combination of 13+10, a combination of 10+13 or by a single 26 (“end of file” code).
Library: SyFile_FILLIN
Input: A – Filehandler ID.
HL – Destination buffer address (size must be 255 bytes).
E – Destination buffer RAM bank (0~15).
Output: C – Number of read bytes (0–254; without terminator).
B – Flag, if line/file end reached (0=no, 1=yes).
Z = 0 ⟶ 1 or more bytes have been loaded.
1 ⟶ EOF reached, nothing has been loaded.
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, DE, HL, IX, IY.

5.6.5 – Directory Management Functions

ID: 032 (DIRDEV) – Directory_Device
Description: Selects the current drive.
Library: SyFile_DIRDEV.
Input: A – Driveletter ("A"–"Z").
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 033 (DIRPTH) – Directory_Path
Description: Selects the current path for the current or a different drive.
Library: SyFile_DIRPTH.
Input: IXh – File path RAM bank (0~15).
HL – File path address.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 034 (DIRPRS) – Directory_Property_Set

Description: Changes a property of a file or a directory. You can set the attribute, the "created" time and the "modified" time. For more information about the time and date code see 023 (FILF2T).

Library: SyFile_DIRPRS

Input: IXh – File path and name RAM bank (0~15).
HL – File path and name address.
A – Property type.
0 – Attribute.
⟶ C – Attribute:
bit0 = 1 ⟶ Read only.
bit1 = 1 ⟶ Hidden.
bit2 = 1 ⟶ System.
bit5 = 1 ⟶ Archive.
1 – Timestamp modified.
⟶ BC – Time code, DE – Date code.
2 – Timestamp created.
⟶ BC – Time code, DE – Date code.
BC,DE – See above.

Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).

Registers: AF, BC, DE, HL, IX, IY.

ID: 035 (DIRPRR) – Directory_Property_Get

Description: Reads a property of a file or a directory. For more information about the time and date code see 023 (FILF2T).

Library: SyFile_DIRPRR.

Input: IXh – File path and name RAM bank (0~15).
HL – File path and name address.
A – Property type:
0 – Attribute.
1 – Timestamp modified.
2 – Timestamp created.

Output: C – Attributes (if requested):
bit0 = 1 ⟶ Read only.
bit1 = 1 ⟶ Hidden.
bit2 = 1 ⟶ System.
bit3 = 1 ⟶ Volume ID.
bit4 = 1 ⟶ Directory.
bit5 = 1 ⟶ Archive.

BC, DE – Time and date code (if requested).
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, HL, IX, IY.

ID: 036 (DIRREN) – Directory_Rename
Description: Renames a file or a directory. The new filename must not include a path.
Library: SyFile_DIRREN.
Input: IXh – RAM bank (0~15) of old and new filename.
HL – Address of file path and old filename.
DE – Address of new filename.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 037 (DIRNEW) – Directory_New
Description: Creates a new directory.
Library: SyFile_DIRNEW
Input: IXh – Directory path and name RAM bank (0~15)
HL – Directory path and name address
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 038 (DIRINP) – Directory_Input
Description: Reads the content of a directory. You can specify a name filter by adding a file mask to the path (* and ? are allowed) and an attribute filter. We recommend always to set Bit3 (volume ID) of the attribute filter byte. Filenames don't contain spaces. For a more powerful function see 013 (DEVDIR).
Library: SyFile_DIRINP.
Input: IXh – Directory path RAM bank (0~15).
HL – Directory path address (may include a search mask).
IXl – Attribute filter:
bit0 = 1 ⟶ Don't show read only files.
bit1 = 1 ⟶ Don't show hidden files.
bit2 = 1 ⟶ Don't show system files.
bit3 = 1 ⟶ Don't show volume ID entries.
bit4 = 1 ⟶ Don't show directories.
bit5 = 1 ⟶ Don't show archive files.

A – Destination buffer RAM bank (0~15)
DE – Destination buffer address
BC – Destination buffer length
IY – Number of entries, which should be skipped

Output: HL – Number of read entries
BC – Remaining unused space in the destination buffer
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).

Registers: AF, DE, IX, IY.

Data structure:

00	4B	File length (32bit double word).
04	1W	Date code, see 023 (FILF2T).
06	1W	Time code, see 023 (FILF2T).
08	1B	Attributes, see 035 (DIRPRR).
09	?B	File or sub directory name.
??	1B	0 terminator.

ID: 039 (DIRDEL) – Directory_DeleteFile

Description: Deletes one or more files. You can delete multiple files by using a file mask (* and ? are allowed). Files, which are read only, can't be deleted. This function also can't be used for deleting directories. Use 040 (DIRRMD), if you want to delete directories.

Library: SyFile_DIRDEL.

Input: IXh – File path and name/mask RAM bank (0~15).
HL – File path and name/mask address.

Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).

Registers: AF, BC, DE, HL, IX, IY.

ID: 040 (DIRRMD) – Directory_DeleteDirectory

Description: Deletes a sub directory. The sub directory has to be empty and not read only, otherwise the operation will be aborted.

Library: SyFile_DIRRMD.

Input: IXh – Directory path and name RAM bank (0~15).
HL – Directory path and name address.

Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).

Registers: AF, BC, DE, HL, IX, IY.

ID: 041 (DIRMOV) – Directory_Move

Description: Moves a file or sub directory into another directory of the same drive. You can either move files or sub directories with this function, in both cases the source path+name must not end with a "/".
Library: SyFile_DIRMOV.
Input: IXh – File/directory old and new path RAM bank (0~15).
HL – File/directory source path and name address.
DE – File/directory destination path address.
Output: CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, HL, IX, IY.

ID: 042 (DIRINF) – Directory_DriveInformation

Description: Returns information about one drive.
Library: SyFile_DIRINF
Input: A – Driveletter ("A"–"Z").
C – Information type.
0 – General drive information.
1 – free and total amount of memory.
Output: ⟶ Information type 0:
A – Type:
00 – Device does not exist.
01 – Device is ready.
02 – Device is not initialized.
03 – Device is corrupt.
B – Medium:
01 – Floppy disc single side (Amsdos, PCW).
02 – Floppy disc double side (FAT 12).
08 – RAM disc.
16 – IDE hard disc or CF card (FAT 16, FAT 32).
C – File system:
01 – Amsdos Data.
02 – Amsdos System.
03 – PCW 180K.
16 – FAT 12.
17 – FAT 16.
18 – FAT 32.
D – Sectors per cluster
IY, IX – Total number of clusters.

⟶ Information type 1:
HL, DE – Number of free 512Byte sectors.
IY, IX – Total number of clusters.
C – Sectors per cluster
⟶ Information type 0 and 1:
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: F.

ID: 013 (DEVDIR) – Directory_Input_Extended
Description: It reads the content of a directory and converts it into ready to use list control data. First you have to reserve two memory areas in the same RAM bank. One area needs to be reserved inside the data RAM area. It will contain the texts (filenames, dates etc.) and numbers (file sizes) for the list control. You can choose any size, but we recommend at least 4000 Bytes. BC must contain its size, when you call the function. DE contains the address, and the low nibble of A the RAM bank number. The second area needs to be reserved inside the transfer RAM area of the same bank. It contains the data structure of the list control. It size is calculated like this:

Size = Maximum_number_of_entries * (4 + Additional_columns * 2)

So when you have two additional columns (like size and attributes) and want to load up to 100 entries, you need to reserve 800 bytes. As there are no more Z80–registers available, the address of this memory area and the maximum number of entries must be written to the beginning of the other memory area. For additional information about reading directories see 038 (DIRINP).
Library: SyFile_DEVDIR.
Input: A – bit0–3 ⟶ Destination buffer RAM bank (0~15).
bit4–7 ⟶ Directory path RAM bank (0~15).
HL – Directory path address (may include a search mask).
DE – Destination buffer address. This must first contain 2 words with additional information at the beginning:
00 1W Address of list control table
02 1W Maximum number of entries
The function will overwrite this information and fill the buffer with the directory data.

BC – Maximum size of destination buffer.
IXl – Attribute filter:
bit0 = 1 ⟶ Don't show read only files.
bit1 = 1 ⟶ Don't show hidden files.
bit2 = 1 ⟶ Don't show system files.
bit3 = 1 ⟶ Don't show volume ID entries.
bit4 = 1 ⟶ Don't show directories.
bit5 = 1 ⟶ Don't show archive files.
IY – Number of entries, which should be skipped.
IXh – Additional columns:
bit0 = 1 ⟶ File size.
bit1 = 1 ⟶ Date and time (last modified).
bit2 = 1 ⟶ Attributes.
Output: HL – Number of read entries.
CY – Error state (0 – Ok, 1 – Error; A ⟶ error code).
Registers: AF, BC, DE, IX, IY.

5.6.6 – Device Manager Functions

The device manager functions have to be called with RST 20H (BNKSCL).

TIMGET (810CH) – Device_TimeGet
Description: Returns the current time.
How to call: rst 20H : dw 810CH
Input: –
Output: A – Second (0 ~ 59).
B – Minute (0 ~59).
C – Hour (0 ~ 23).
D – Day (1 ~ 31).
E – Month (1 ~ 12).
HL – Year (1900 ~ 2100).
IXl – Timezone (–12 ~ +13).
Registers: F, IY.

TIMSET (810FH) – Device_TimeSet
Description: Sets the current time.
How to call: rst 20H : dw 810FH

Input: A – Second (0 ~ 59).
B – Minute (0 ~ 59).
C – Hour (0 ~ 23).
D – Day (1 ~ 31).
E – Month (1 ~ 12).
HL – Year (1900 ~ 2100).
IXl – Timezone (–12 ~ +13).
Output: –
Registers: AF, BC, DE, HL, IY.

SCRSET (8136H) – Device_ScreenModeCPCSet
Description: Sets the current CPC screen mode. This function is CPC specific only.
How to call: ld hl,8136H : rst 28H
Input: E – CPC screen mode (0, 1, 2).
Output: –
Registers: –

SCRGET (8139H) – Device_ScreenMode
Description: Returns the current screen mode, colour depth and resolution.
How to call: ld hl,8139H : rst 28H
Input: –
Output: E – Screen mode:
CPC/EP: 1, 2 MSX: 5, 6, 7
PCW: 0 G9K: 8, 9, 10, 11
D – Number of colours (2–16).
IX – X resolution.
IY – Y resolution.
Registers: –

MOSGET (813CH) – Device_MousePosition
Description: Returns the current position of the mouse pointer.
How to call: rst 20H : dw 813CH
Input: –
Output: DE – X position.
HL – Y position.
Registers: –

MOSKEY (813FH) – Device_MouseKeyStatus
Description: Returns the current status of the mouse keys.
How to call: rst 20H : dw 813FH
Input: –
Output: A – Key Status
bit 0 = 1 ⟶ Left mouse button is pressed.
bit 1 = 1 ⟶ Right mouse button is pressed.
bit 2 = 1 ⟶ Middle mouse button is pressed.
Registers: F.

KEYTST (8145H) – Device_KeyTest
Description: Returns the current status of a key. For the scan codes see KEYBOARD SCAN CODES.
How to call: ld hl,8145H : rst 28H
Input: E – Keyboard scan code.
Output: E – Key status.
0 ⟶ Key is currently not pressed;
1 ⟶ Key is currently pressed.
Registers: AF, BC, D ,HL ,IX ,IY.

KEYSTA (8148H) – Device_KeyStatus
Description: Returns the status of the shift/control/alt/capslock keys.
How to call: ld hl,8148H : rst 28H
Input: –
Output: E – bit0 = 1 ⟶ Shift pressed.
bit1 = 1 ⟶ Control pressed.
bit2 = 1 ⟶ Alt pressed.
D – Caps lock status (1 ⟶ Locked).
Registers: AF, BC, HL, IX, IY.

KEYPUT (814BH) – Device_KeyPut
Description: Puts a char back into the keyboard buffer.
How to call: rst 20H : dw 814BH
Input: A – Char (ASCII code).
Output: CY – Status (1 ⟶ Keyboard buffer full).
Registers: AF, BC, HL.

IOMINP (8157H) – Device_IO_MultiIn [CPC only]
Description: Reads multiple bytes from a hardware port in a very fast way and writes them to a destination address in memory.

This function is only available in SymbOS CPC due to its limited banking abilities.
How to call: rst 20H : dw 8157H
Input: DE – Destination address.
IY – bit12–15 ⟶ Destination bank (0~15).
bit0–11 ⟶ Length.
IX – Port address.
Registers: AF, BC, DE, HL.

IOMOUT (815AH) – Device_IO_MultiOut [CPC only]
Description: Writes multiple bytes to a hardware port in a very fast way from a source address in memory. See also IOMINP. This function is only available in SymbOS CPC due to its limited banking abilities.
How to call: rst 20H : dw 8157H
Input: DE – Source address.
IY – bit12–15 ⟶ Source bank (0~15).
bit0–11 ⟶ Length.
IX – Port address.
Registers: AF, BC, DE, HL.

5.7 – SYMSHELL TEXT TERMINAL

SymShell commands are triggered via a message, which has to be sent with RST 10H (MSGSND) to the SymShell process. SymShell will pass its process ID and the text screen resolution to the application via the command line.

5.7.1 - Text terminal commands

ATTRIB
Format: ATTRIB <file name>
Function: Displays current file attributes.
Format: ATTRIB <filename>%–R, %+R, %–H, %+H, %–S, %+S
Function: Add (%+) or remove (%–) file attributes.
H ⟶ Hidden file.
S ⟶ System file.
R ⟶ Read-only.

CD

Format: CD
Function: Displays directory.
Format: CD [[d:][directory]]
Function: Change the current subdirectory.
Format: CD [\ | ..]
Function: Returns only one subdirectory (..) or returns to the root directory (\ ).

CLS

Format: CLS
Function: Clears the screen.

COPY

Format: COPY <source file> <destination file>
Function: Copies files. Accepts the wildcard character '*' (for example, "file.*", "*.ext", or "*.*").

DATE

Format: DATE [date]
Function: Displays or changes the system date. The date can also be changed by double-clicking the time on the taskbar.

DEL

Format: DEL <filename>
Function: Deletes one or more files. Accepts the wildcard character '*' (e.g. "*.ext" or "fil*.ex*")

DIR

Format: DIR [%S] [%H] [%F] [%/P] [<file name>]
Function: Displays the names of files on the disk. Accepts the wildcard character '*' (for example, "*.ext" or "fil*.ex*")
[/S] Lists system files.
[/H] List hidden files.
[/F] Calculates free disk space.
[/P] Pause the listing when completing a screen.

HELP
Format: HELP [<command>]
Function: Displays the help file for the specified <command> or lists them all if there is no argument.

MKDIR
Format: MKDIR [d:] <path>
Function: Creates a subdirectory.

MOVE
Format: MOVE <filename> <destination-path>
Function: Move files to another part of disk. Accepts the wildcard character '*' (example: "file.* destination.*")

RMDIR
Format: RMDIR <directory name>
Function: Removes an empty subdirectory.

REN
Format: REN <old name> <new name>
Function: Renames file <old name> to <new name>. Accepts the wildcard character '*' (example: "ren *.txt *.bak").

TIME
Format: TIME [<time>]
Function: Displays or changes the system time. The time can also be changed by double-clicking the time on the taskbar.

5.7.1.1 - Default Applications

DIMON.COM – Command line disk monitor to view disks content in hexadecimal.

NETSTAT.COM – Displays active network connections with their status.

TELNET.COM – Telnet client. Type TELNET <domain name> or <IP>. By default, TCP port 23 is used. If SYMTEL is started without a domain name or IP, the console is loaded, where the settings can be changed. To connect to a host, simply enter the IP or

DNS name and press <RETURN>. To disconnect press CTRL–C or log off the remote system. Also press CTRL–C to exit SYMTEL To switch between full screen and windowed mode, press GRAPH+RETURN. In full screen mode the performance is better than in a window.

UNZIP.COM – Extract .ZIP or .GZ files. Type UNZIP H to display help.

WGET.COM – Command line application for downloading files under HTTP protocol using network daemon.

5.7.2 – SymShell Commands and responses

ID: 064 (MSC_SHL_CHRINP) – SymShell_CharInput_Command

Description: Requests a char from an input source. The input source can be the standard channel or the console keyboard. If the keyboard is used, SymShell waits for the user and won't send a response as long as no key is pressed.

Library: SyShell_CHRINP.

Message:
- 00 1B 064.
- 01 1B Channel (0 ⟶ Standard, 1 ⟶ Keyboard).

Response: See MSR_SHL_CHRINP.

ID: 192 (MSR_SHL_CHRINP) – SymShell_CharInput_Response

Description: If a char could be received from the keyboard, a file or another source, it will be sent to the application via this response message. If the user pressed Control+C or if the end of the file (EOF) has been reached, the EOF flag will be set.

Message:
- 00 1B 192.
- 01 1B EOF flag (If ≠ 0 ⟶ EOF reached, no char available!).
- 02 1B Char.
- 03 1B Error state.
  - 254 – Unknown process (SymShell doesn't know the process, which sent the command, so it won't provide any service).
  - 253 – Destination device full.
  - 252 – Internal ring buffer full.

251 – Too many processes (SymShell can't handle the amount of processes running at the same time in its text terminal environment).
Any other: See "Error Codes" in chapter "File Manager".

ID: 065 (MSC_SHL_STRINP) – SymShell_StringInput_Command
Description: Requests a string from an input source. The input source can be the standard channel or the console keyboard. The maximum lenght of a string is 255 chars, so the buffer must have a size of 256 bytes (255 + terminator). A string is always terminated by 0.
Library: SyShell_STRINP.
Message:
00 1B 065.
01 1B Channel (0 ⟶ Standard, 1 ⟶ Keyboard).
02 1B Destination buffer RAM bank (0~15).
03 1W Destination buffer address.
Response: See MSR_SHL_STRINP.

ID: 193 (MSR_SHL_STRINP) – SymShell_StringInput_Response
Description: If a text line could be received from the keyboard, a file or another source (terminated by 13/10), it will be sent the application via this response message. If the user pressed Control+C or if the end of the file (EOF) has been reached, the EOF flag will be set.
Message:
00 1B 193.
01 1B EOF flag (If ≠ 0 ⟶ EOF reached, no string available!).
03 1B Error state (see above "SymShell_CharInput_Response").

ID: 066 (MSC_SHL_CHROUT) – SymShell_CharOutput_Command
Description: Sends a char to the output destination. The output destination can be the standard channel or the console text screen.
Library: SyShell_CHROUT.
Message:
00 1B 066.
01 1B Channel (0 ⟶ Standard, 1 ⟶ Screen).
02 1B Char.
Response: See MSR_SHL_CHROUT.

ID: 194 (MSR_SHL_CHROUT) – SymShell_CharOutput_Response

Description: Informs the application, if the char has be sended correctly. An application shouldn't send more than one char at the same time, before such a response has been received.

Message:
00 1B 194.
03 1B Error state (see above "SymShell_CharInput_Response")

ID: 067 (MSC_SHL_STROUT) – SymShell_StringOutput_Command

Description: Sends a string to the output destination. The output destination can be the standard channel or the console text screen. A string has always to be terminated by 0. The lenght, which has to be specified, must not include the 0–terminator.

Library: SyShell_STROUT.

Message:
00 1B 067.
01 1B Channel (0 ⟶ Standard, 1 ⟶ Screen).
02 1B String RAM bank (0~15).
03 1W String address.
05 1B String length (without 0–terminator).

Response: See MSR_SHL_STROUT.

ID: 195 (MSR_SHL_STROUT) – SymShell_StringOutput_Response

Description: Informs the application, if the string has be sended correctly. An application shouldn't send more than one string at the same time, before such a response has been received.

Message:
00 1B 195
03 1B Error state (see above "SymShell_CharInput_Response")

ID: 068 (MSC_SHL_EXIT) – SymShell_Exit_Command

Description: The application informs SymShell about an exit event. If an application quits itself, SymShell has to be informed about that, so that it can remove the application from its internal management table. In this case the exit type has to be 0 ("quit").

Library: SyShell_EXIT

Message: 00 1B 068.
01 1B Exit type:
0 ⟶ Application quits itself
1 ⟶ Application releases focus and goes into blur mode
Response: SymShell does not send a response message.

ID: 069 (MSC_SHL_PTHADD) – SymShell_PathAdd_Command
Description: ...
Library: SyShell_PTHADD.
Message: 00 1B 069.
01 1W Address of base path (0 ⟶ default).
03 1W Address of additional path component.
05 1W Address of new full path.
07 1B Pathes RAM bank (0~15).
Response: See MSR_SHL_PTHADD.

ID: 197 (MSR_SHL_PTHADD) – SymShell_PathAdd_Response
Description: ...
Message: 00 1B 197.
01 1W Position behind last char in new path.
03 1W Position behind last / in new path.
05 1B bit0=1 ⟶ New path ends with /.
bit1=1 ⟶ New path contains wildcards.

5.7.3 – Symshell Text Terminal Control

00 Stop textoutput and ignore remaining part of the line.
01 –
02 Switch cursor off. This will make the cursor invisible.
03 Switch cursor on.
04 Save current cursor position.
05 Restore last saved cursor position.
06 Activate textoutput (see also 21).
07 –
08 Move cursor one char to the left.
09 Move cursor one char to the right.
10 Move cursor one char downwards.
11 Move cursor one char upwards.
12 Clear screen and place cursor at position 1/1.

13 Move cursor to the beginning of the current line.
14 Move cursor by multiple chars (P1 – Direction and steps)
1~80 ⟶ cursor will move 1~80 chars to the right.
81~160 ⟶ cursor will move 1~ 80 chars to the left (parameter–80).
161~185 ⟶ cursor will move 1~25 chars downwards (parameter–160).
186~210 ⟶ cursor will move 1~25 chars upwards (parameter–185).
The cursor will not cross any borders.
15 –
16 Clear char at cursor position (using space [32]).
17 Clear line from cursor left.
18 Clear line from cursor right.
19 Clear screen from cursor up.
20 Clear screen from cursor down.
21 Deactivate textoutput. No more chars will be printed until a code 06 appears.
22 Set a tab at the current column.
23 Clear a tab at the current column.
24 Clear all tabs.
25 Jump to next tab.
26 Fill screen area with a specified char. **This control code is not implemented yet.**
P1 – Char.
P2 – X start. P4 – X end.
P3 – Y start. P5 – Y end.
27 –
28 Set terminal window size. The minimum size is 10x4, the maximum is 80x25 (MSX: 80x24). After the window has been resized, the screen will be cleared and the cursor placed in the upper left corner (1/1).
P1 – Width.
P2 – Height.
29 Scroll window up or down one line. This will not influence the current cursor position.
P1 – Direction (1 ⟶ up, 2 ⟶ down).
30 Move cursor to the upper left corner (1/1).
31 Move cursor to a specified screen location.
P1 – X pos (1~80).
P2 – Y pos (1~25).

5.7.4 – Extended ASCII Codes

136 – cursor up
137 – cursor down
138 – cursor left
139 – cursor right
140 – F0
141 – F1
142 – F2
143 – F3
144 – F4
145 – F5
146 – F6
147 – F7
148 – F8
149 – F9
150 – F.
151 – Alt + @
152 – Alt + A
153 – Alt + B
154 – Alt + C
155 – Alt + D
156 – Alt + E
157 – Alt + F
158 – Alt + G
159 – Alt + H
160 – Alt + I
161 – Alt + J
162 – Alt + K
163 – Alt + L
164 – Alt + M
165 – Alt + N
166 – Alt + O
167 – Alt + P
168 – Alt + Q
169 – Alt + R
170 – Alt + S
171 – Alt + T
172 – Alt + U
173 – Alt + V
174 – Alt + W
175 – Alt + X
176 – Alt + Y
177 – Alt + Z
178 – Alt + 0
179 – Alt + 1
180 – Alt + 2
181 – Alt + 3
182 – Alt + 4
183 – Alt + 5
184 – Alt + 6
185 – Alt + 7
186 – Alt + 8
187 – Alt + 9

5.7.5 – Keyboard Scan Codes

The scan code are used in the "Device_KeyTest" function. Please note, that they are equal on all supported platforms.

00 – Cursor Up
01 – Cursor Right
02 – Cursor Down
03 – F9
04 – F6
05 – F3
06 – Enter
07 – F.
08 – Cursor Left
09 – Alt
10 – F7
11 – F8
12 – F5
13 – F1
20 – F4
21 – Shift
22 – \
23 – Control
24 – ^
25 – –
26 – @
27 – P
28 – ;
29 – :
30 – /
31 – .
32 – 0
33 – 9
40 – 8
41 – 7
42 – U
43 – Y
44 – H
45 – J
46 – N
47 – Space
48 – 6
49 – 5
50 – R
51 – T
52 – G
53 – F
60 – S
61 – D
62 – C
63 – X
64 – 1
65 – 2
66 – Esc
67 – Q
68 – Tab
69 – A
70 – Capslock
71 – Z
72 – Joystick Up
73 – Joystick Down

14 – F2	34 – O	54 – B	74 – Joystick Left
15 – F0	35 – I	55 – V	75 – Joystick Right
16 – Clr	36 – L	56 – 4	76 – Fire 2
17 – [	37 – K	57 – 3	77 – Fire 1
18 – Return	38 – M	58 – E	78 – [not used]
19 –]	39 – ,	59 – W	79 – Del

5.8 – SYSTEM CONFIGURATION

The SYMBOS.INI file is divided into 5 parts:
→ Header, which contains the identifier and the length of three following parts
→ Core area part, which contains data loaded in the first RAM bank
→ Data area part, which contains additional data usually loaded in a different RAM bank
→ Tranfer area part (currently empty)
→ Font

5.8.1 – Header

0000 2B Identifier, which also contains the version of the config file [byte0]="S", [byte1]=1 (current version).
0002 1W Length of the header (=8 bytes) plus the core area part of the SymbOS system configuration (will be always loaded to RAM–bank 0).
0004 1W Length of the data area part (the RAM–bank depends on the computer platform)
0006 1W Length of the transfer area part (not used, aleays 0)

5.8.2 – Core Area Part

5.8.2.1 – Mass storage devices

0000 128B Device configuration; this consists of 8 data records at 16 bytes for each device
- 00 1B Drive letter (upper case) or 0, if device slot is empty.
- 01 1B bit0–3: Type (0=Floppy, 1=IDE/SCSI) → Driver slot.
  bit4–6: Reserved (set to 0).
  bit7: Flag, if removeable media (1=yes).

02 1B Sub drive:
→ If the device is a floppy disc:
bit0–1: Drive.
bit2: Head.
bit3: Flag, if double step.
bit4–7: Reserved (set to 0).
→ If the device is an IDE/SCSI/SD device:
bit0–3: partition (0=not partitioned).
bit4–7: IDE → channel (master=0, slave=1).
SCSI → sub device (0~15).
03 1B Reserved (set to 0).
04 12B Device name (terminated by 0).

5.8.2.2 – Display and miscellaneous (1)

0128 17W Colour palette (the border is defined by the 17th word)
For each entry:
bit0–3: Blue component.
bit4–7: Green component.
bit8–11: Red component.

0162 1B Screen mode:

0	PCW (768x255x2)	7	MSX (512x212x16)
1	CPC,EP (320x200x4)	8	G9K (384x240x16)
2	CPC,EP (640x200x2)	9	G9K (512x212x16)
5	MSX (256x212x16)	10	G9K (768x240x16)
6	MSX (512x212x4)	11	G9K (1024x212x16)

0163 32B System path.
0195 1B Time zone (−12 to +12).
0196 1B Background type (0~15=plain colour, −1=background graphic).
0197 32B Background graphic path and filename, terminated by 0 (only, if "background type" = −1).

5.8.2.3 – Keyboard (1) and mouse

0229 1B Keyboard delay (in 1/50s; between first an second char).
0230 1B Keyboard repeat speed (delay between every following chars).
0231 1B Joystick mouse delay (until mouse reaches full speed).

0232	1B	Joystick mouse speed (in pixel)
0233	1B	Mouse speed (CPC–SYMBiFACE [PS/2] and MSX) factor (final_movement – original_movement * mouse_speed / 16).
0234	1B	Mouse double click delay (maximum time in 1/50s, when a double click is recognized)
0235	1B	Flag, if swap left/right mouse keys.
0236	1B	Mouse wheel speed (currently only CPC–SYMBiFACE [PS/2] and MSX)

5.8.2.4 – Miscellaneous (2) and Desktop Links

0237	1B	SYMBOS.INI drive ("A", ...)
0238	1B	Miscellaneous flags. bit0: Autosave config.
0239	1B	Flag (1), if SymbOS extension module should be loaded.
0240	1B	Flag for extended hardware (+1=Mouse, +2=Real time clock, +4=IDE/SCSI interface, +16=M4Board).
0241	1B	Virtual desktop (0=no virtual desktop, bit0–3 ⟶ X–resolution, 1=512, 2=1000, bit4–7 ⟶ Y–resolution, not yet defined).
0242	1B	Number of desktop icons.
0243	1B	Number of start menu/programs entries.
0244	1B	Number of taskbar short–cut entries (currently not supported)
0245	1B	Machine type: 0=CPC 464 — 7=MSX1 1=CPC 664 — 8=MSX2 2=CPC 6128 — 9=MSX2+ 3=CPC 464+ — 10=MSX turboR 4=CPC 6128+ — 12=PCW8xxx 6=Enterprise — 13=PCW9xxx
0246	16W	Desktop icon positions; for each of the 8 icons there are two words, the first contains the X–, the second the Y–position.
0278	32B	Path and filename of the autoexec command line file.
0310	1B	Flag, if autoexec command line file should be executed.

5.8.3 – Data Area Part

5.8.3.1 – Desktop Links (2)

0000 400B Start menu program entry names (20 entries at 20 bytes, each terminated by 0).
0400 640B Start menu program entry pathes and filenames (20 entries at 32 bytes, each terminated by 0).
1040 256B Desktop icon pathes and filenames (8 entries at 32 bytes, each terminated by 0).
1296 192B Desktop icon names (8 entries, each consists of 2 lines at 12 bytes, each line is terminated by 0).
1488 1176B Desktop icon graphics (8 entries, each consists of the 3 bytes graphic header and the 144 byte (6*24) bitmap).
2664 768B File extension association (16 entries at 48 bytes)
- 00 3B Extension 1 (uppercase; if byte0=1, then the whole entry is not defined).
- 03 3B Extension 2 (if byte0=1, then this one entry is not defined).
- 06 3B Extension 3 (s.a.).
- 09 3B Extension 4 (s.a.).
- 12 3B Extension 5 (s.a.).
- 15 33B Application path and filename, which will be started, if a file with one of the above listed extensions has been opened.

5.8.3.2 – Screen Saver

3432 1B Flag, if screen saver is present.
3433 1B Duration of user inactivity, after which the screen saver will be started.
3434 33B Screen saver application path and filename (terminated by 0).
3467 64B Screen saver specific configuration data (can be stored and read here).

5.8.3.3 – Keyboard (2)

3531 80B Keyboard definition (normal).
3611 80B Keyboard definition (shift).
3691 80B Keyboard definition (control).
3771 80B Keyboard definition (alt).

5.8.3.4 – Security

3851 16B Security username.
3867 16B Security password.
3883 1B Security flags [not used yet, set to 0].

5.9 – SCREENSAVER APPLICATIONS

This is a list of commands, which will be sent to the screen saver application. Usually they will be sent by the desktop manager or by the control panel. The creensaver must be able to handle these commands and one additional response message for a proper interaction.

ID: 001 (MSC_SAV_INIT) – ScreenSaver_Init_Command
Description: The caller process, which has started the screensaver (usually the desktop manager or the control panel) has sent an nitialisation command. The screensaver now should store the sender process ID to be able to send a configuration response message later (see MSR_SAV_CONFIG). Then it has to copy the configuration data into its own memory area. This data can have a size of up to 64 bytes and is stored in the SYMBOS.INI file together with the other system settings. If the screensaver requires more than 64 bytes for its configuration it has to manage its own config file.
Library: ScrSav_MAIN.
Message: 00 1B 001.
01 1B Config data (64 byte) RAM bank (0~7).
02 1W Config data (64 byte) address.
Response: No response from the screensaver expected.

ID: 002 (MSC_SAV_START) – ScreenSaver_Start_Command
Description: The caller process asks the screensaver to start its animation. The animation should be shown as long as no key has been pressed and the mouse hasn't been moved.
Library: ScrSav_MAIN
Message: 00 1B 002.
Response: No response from the screensaver expected.

ID: 003 (MSC_SAV_CONFIG) – ScreenSaver_Config_Command
Description: The caller process asks the screensaver to open a

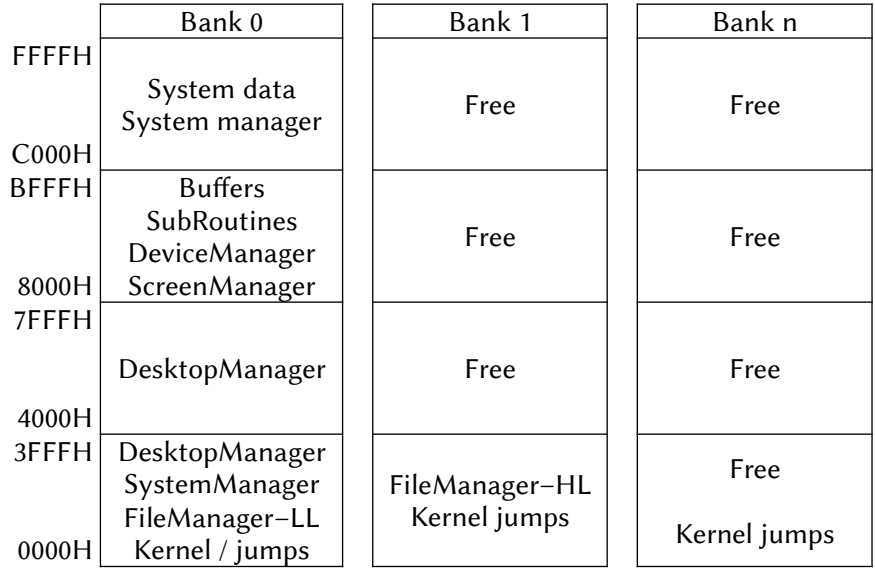
configuration dialogue. In such a window the user has the possibility to modify the screensaver settings. If there is nothing to configure at all, the screensaver can ignore this command or just open an info window.
Library: ScrSav_MAIN.
Message: 00 1B 003.
Response: See MSR_SAV_CONFIG.

ID: 004 (MSR_SAV_CONFIG) – ScreenSaver_Config_Response
Description: The user has finished modifying the settings and clicked on the "OK" button of the configuration dialogue.
Library: ScrSav_CFGSAV
Message: 00 1B 001.
01 1B Config data (64 byte) RAM bank (0~7).
02 1W Config data (64 byte) address.

5.10 – SYMBOS MEMORY MAP

5.10.1 – General Memory Usage

The following diagram shows, in which way the different memory banks and blocks are used in SymbOS.



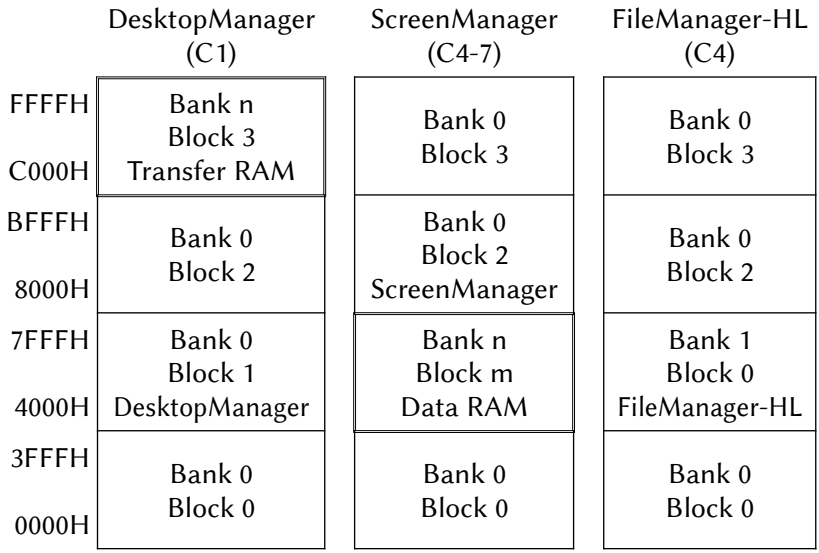
5.10.2 – Application Memory Usage

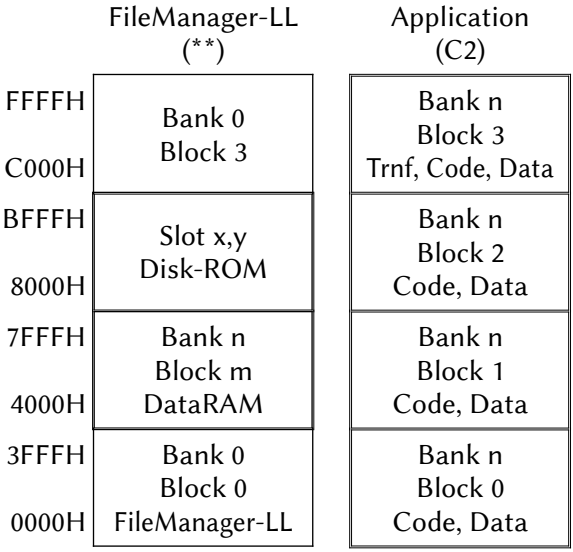
The memory inside an application RAM bank (1–n) is used in the following way:

1. 0000–03FF Kernel jumps, Kernel multitasking and banking routines.
2. 0400–FFFF Application code and internal application data.
3. 0400–3FFF Application data used by the screen manager.
   4000–7FFF (One object has to be inside one 16K block).
   8000–BFFF
   C000–FFFF
4. C000–FFFF Application "transfer" data, used by the desktop manager, message buffer, stack.

5.10.3 – Memory Configurations

The following diagram shows, how the memory is configured during the activity of one of the modules of SymbOS.





5.11 – SCREEN MANAGER

The screen manager contains all routines for the direct access of the video hardware. There is currently only one function, that can be used by applications as well.

TXTLEN (815DH) – Screen_TextLength

Description: Returns the width and height of a textline in pixels, if it would be printed to the screen. You can define the text length (number of chars) in IY. If the text is terminated by 0 or 13 you should use –1 for the maximal text length. Please note, that this function always uses the system font for calculating he width and height.

How to call: rst 20H : dw 815DH.

Input: HL – Text address.
A – Text RAM bank (1~15).
IY – Maximal number of chars (text length).

Output: DE – Text width in pixels.
A – Text height in pixels.

Registers: F, BC, HL, IX.

5.12 – NETWORK DAEMON

The SymbOS network daemon provides all services for full network access. It's running as a shared service process [...]

5.12.1 – Configuration

Config_Get CFGGET 001 130 A – type, E,HL – data buffer
⟶ (buffer has been filled)
Config_Set CFGSET 002 131 A – type, E,HL – config data
⟶ (config has been set)

5.12.2 – Transportation Layer Services

TCP_Open	TCPOPN	016	144	A – mode, HL – local port (IX,IY – remote IP, DE – remote port) CY=0 ⟶ ok, A – handle
TCP_Close	TCPCLO	017	145	A – handle CY=0 ⟶ ok, A – handle
TCP_Status	TCPSTA	018	146	A – handle CY=0 ⟶ ok, A – handle, L – status (BC – received bytes, IX,IY – remote IP, DE – remote port)
TCP_Receive	TCPRCV	019	147	A – handle, BC – length, E,HL – memory CY=0 ⟶ ok, A – handle, BC – number of remaining bytes, Z=1 ⟶ all bytes have been received
TCP_Send	TCPSND	020	148	A – handle, BC – length, E,HL – memory CY=0 ⟶ ok, A – handle, BC – number of sent bytes, HL – number of remaining bytes, Z=1 ⟶ all bytes have been sent
TCP_Skip	TCPSKP	021	149	A – handle, BC – length CY=0 ⟶ ok, A – handle
TCP_Flush	TCPFLS	022	150	A – handle CY=0 ⟶ ok, A – handle
TCP_Disconnect	TCPDIS	023	151	A – handle

				CY=0 ⟶ ok, A – handle
TCP_Event	TCPEVT		159	A – handle, L – status (BC – received bytes, IX,IY – remote IP, DE – remote port)
UDP_Open	UDPOPN	032	160	HL – local port, E – memory bank CY=0 ⟶ ok, A – handle
UDP_Close	UDPCLO	033	161	A=handle CY=0 ⟶ ok, A – handle
UDP_Status	UDPSTA	034	162	A=handle CY=0 ⟶ ok, A – handle, L – status (BC – received bytes, IX,IY – remote IP, DE – remote port)
UDP_Receive	UDPRCV	035	163	A – handle, HL – memory CY=0 ⟶ ok, A – handle
UDP_Send	UDPSND	036	164	A – handle, BC – length, HL – memory, IX,IY – remote IP, DE=remote port CY=0 ⟶ ok, A=handle
UDP_Skip	UDPSKP	037	165	A – handle CY=0 ⟶ ok, A – handle
UDP_Event	UDPEVT		175	A – handle, L – status (BC – received bytes, IX,IY – remote IP, DE – remote port)

5.12.3 – Application Layer Services

DNS_Resolve	DNSRSV	112	240	E,HL – address CY=0 ⟶ Ok, IX,IY – IP
DNS_Verify	DNSVFY	113	241	E,HL – address A – type of address (0 ⟶ no valid address, 1 ⟶ IP address, 2 ⟶ domain address)

5.13 – SYMBOS CONSTANTS

5.13.1 – Process-IDs

PRC_ID_KERNEL	equ 1	Kernel process.
PRC_ID_DESKTOP	equ 2	Desktop manager process.
PRC_ID_SYSTEM	equ 3	System manager process.

5.13.2 – Messages

MSC_GEN_QUIT	equ 0	Application is beeing asked, to quit itself.
MSC_GEN_FOCUS	equ 255	Application is beeing asked, to focus its window.

5.13.3 – Kernel Commands

MSC_KRL_MTADDP	equ 1	Add process (P1/2=stack, P3=priority (7 high – 1 low), P4=RAM bank (0~8))
MSC_KRL_MTDELP	equ 2	delete process (P1=ID)
MSC_KRL_MTADDT	equ 3	add timer (P1/2=stack, P4=RAM bank (0~8))
MSC_KRL_MTDELT	equ 4	delete timer (P1=ID)
MSC_KRL_MTSLPP	equ 5	set process to sleep mode
MSC_KRL_MTWAKP	equ 6	wake up process
MSC_KRL_TMADDT	equ 7	add counter service (P1/2=address, P3=RAM bank, P4=process, P5=frequency)
MSC_KRL_TMDELT	equ 8	delete counter service (P1/2=address, P3=RAM bank)
MSC_KRL_TMDELP	equ 9	delete all counter services of one process (P1=process ID)

5.13.4 – Kernel Responses

MSR_KRL_MTADDP	equ 129	process has been added (P1=0/1⟶ok/failed, P2=ID)
MSR_KRL_MTDELP	equ 130	process has been deleted
MSR_KRL_MTADDT	equ 131	timer process has been deleted (P1=0/1⟶ok/failed, P2=ID)
MSR_KRL_MTDELT	equ 132	timer has been removed
MSR_KRL_MTSLPP	equ 133	process is sleeping now
MSR_KRL_MTWAKP	equ 134	process has been waked up
MSR_KRL_TMADDT	equ 135	counter service has been added (P1=0/1⟶ok/failed)
MSR_KRL_TMDELT	equ 136	counter service has been deleted
MSR_KRL_TMDELP	equ 137	all counter services of a process have been deleted

5.13.5 – System Commands

MSC_SYS_PRGRUN	equ 16	load application or document (P1/2=address filename, P3=RAM bank filename)
MSC_SYS_PRGEND	equ 17	quit application (P1=ID)
MSC_SYS_SYSWNX	equ 18	open dialogue to change current window (next) (–)
MSC_SYS_SYSWPR	equ 19	open dialogue to change current window (previously) (–)
MSC_SYS_PRGSTA	equ 20	open dialogue to load application or document (–)
MSC_SYS_SYSSEC	equ 21	open system secuity dialogue (–)
MSC_SYS_SYSQIT	equ 22	open shut shown dialogue (–)
MSC_SYS_SYSOFF	equ 23	shut down (–)
MSC_SYS_PRGSET	equ 24	start control panel (P1=submodul → 0=main window, 1=display settings, 2=date/time)
MSC_SYS_PRGTSK	equ 25	start taskmanager (–)
MSC_SYS_SYSFIL	equ 26	call filemanager function (P1=number, P2–13=AF, BC, DE, HL, IX, IY.)
MSC_SYS_SYSHLP	equ 27	start help (–)
MSC_SYS_SYSCFG	equ 28	call config function (P1=number, 0=load, 1=save, 2=reload background)
MSC_SYS_SYSWRN	equ 29	open message/confirm window (P1/2=adresse, P3=RAM bank, P4=number of buttons)
MSC_SYS_PRGSRV	equ 30	shared service function (P4=type [0=search, 1=start, 2=release], P1/2=address 12char ID, P3=RAM bank 12char ID or P3=program ID, if type=2)
MSC_SYS_SELOPN	equ 31	open fileselect dialogue (P6=filename RAM bank, P8/9=filename address, P7=forbidden attributes, P10=max entries, P12=max buffer size)

5.13.6 – System Responses

MSR_SYS_PRGRUN	equ 144	application has been started (P1=result ⟶ 0=ok, 1=file doesnt exist, 2=file is not executable, 3=error while loading [P8=filemanager error code], 4=mem. full, P8=app ID, P9=process ID)
MSR_SYS_SYSFIL	equ 154	filemanager function returned (P1=num, P2–13=AF, BC, DE, HL, IX, IY)
MSR_SYS_SYSWRN	equ 157	message/confirm window response (P1 ⟶ 0=already in use, 1=opened [P2=number], 2=ok, 3=yes, 4=no, 5=cancel/close)
MSR_SYS_PRGSRV	equ 158	shared service function response (P1=state [5=not found, other codes see MSR_SYS_PRGRUN], P8=app ID, P9=process ID)
MSR_SYS_SELOPN	equ 159	message from fileselect dialogue (P1 ⟶ 0=Ok, 1=cancel, 2=already in use, 3=no memory free, 4=no window free, –1=open ok, modal window has been opened [P2=number])

5.13.7 – Desktop Commands

MSC_DSK_WINOPN	equ 32	open window (P1=RAM bank, P2/3=address data record)
MSC_DSK_WINMEN	equ 33	redraw menu bar (P1=window ID) [only if focus]
MSC_DSK_WININH	equ 34	redraw window content (P1=window ID, P2=–1/–Num/Object, P3=Object) [only if focus]
MSC_DSK_WINTOL	equ 35	redraw window toolbar (P1=window ID) [only if focus]
MSC_DSK_WINTIT	equ 36	redraw window title (P1=window ID) [only if focus]
MSC_DSK_WINSTA	equ 37	redraw window status lien (P1=window ID) [only if focus]

MSC_DSK_WINMVX	equ 38	set content X offset (P1=window ID, P2/3=XPos) [only if focus]
MSC_DSK_WINMVY	equ 39	set content Y offset (P1=window ID, P2/3=XPos) [only if focus]
MSC_DSK_WINTOP	equ 40	takes window to the front (P1=window ID) [always]
MSC_DSK_WINMAX	equ 41	maximize window (P1=window ID) [always]
MSC_DSK_WINMIN	equ 42	minimize window (P1=window ID) [always]
MSC_DSK_WINMID	equ 43	restore window size (P1=window ID) [always]
MSC_DSK_WINMOV	equ 44	moves window to a new position (P1=window ID, P2/3=XPos, P4/5=YPos) [always]
MSC_DSK_WINSIZ	equ 45	resize the window (P1=window ID, P2/3=XPos, P4/5=YPos) [always]
MSC_DSK_WINCLS	equ 46	closes and removes window (P1=window ID) [always]
MSC_DSK_WINDIN	equ 47	redraw window content, even if it has not focus (P1=window ID, P2=−1/−Num/Object, P3=Object) [always]
MSC_DSK_DSKSRV	equ 48	desktop service request (P1=type, P2−P5=parameters)
MSC_DSK_WINSLD	equ 49	redraw window scrollbars (P1=window ID) [only if focus]
MSC_DSK_WINPIN	equ 50	redraw window content part (P1=window ID, P2=−1/−Num/Object, P3=Object, P4/5=Xbeg, P6/7=Ybeg, P8/9=Xlen, P10/11=Ylen) [always]
MSC_DSK_WINSIN	equ 51	redraw content of a super control (P1=window ID, P2=super control ID, P3=SubObject) [always]

5.13.8 – Desktop Responses

MSR_DSK_WOPNER	equ 160	open window failed; the maximum of 32 windows has been reached
MSR_DSK_WOPNOK	equ 161	open window successfull (P4=number)

```
MSR_DSK_WCLICK    equ 162   window has been clicked (P1=window
                            number, P2=action, P3=subspecifica-
                            tion, P4/5,P6/7,P8/9=parameters)
MSR_DSK_DSKSRV    equ 163   desktop service answer (P1=type
                            P2–P5=parameters)
MSR_DSK_WFOCUS    equ 164   window got/lost focus (P1=window
                            number, P2=type [0=blur, 1=focus])
MSR_DSK_CFOCUS    equ 165   control focus changed (P1=window
                            number, P2=control number, P3=reason
                            [0=mouse click/wheel, 1=tab key])
MSR_DSK_WRESIZ    equ 166   window has been resized
                            (P1=window number)
MSR_DSK_WSCROL  equ 167     window content has been scrolled
                            (P1=window number)
MSR_DSK_EXTDSK    equ 168   command for extended desktop
                            (used internally; P1=command,
                            P2–x=parameters)
FNC_DXT_DSKBGR    equ 001   background has been updated
FNC_DXT_FILRUN  equ 002     file has been opened via prgrun
                            (P2/3=address, P4=bank)
FNC_DXT_FILBRW    equ 003   file has been selected via file browser
                            (P2/3=address, P4=bank)
FNC_DXT_MENCLK    equ 004   startmenu has been clicked
                            (P2/3=value)
FNC_DXT_DSKCLK    equ 005   desktop window has been clicked
                            (P2=action, P3=subespecification,
                            P4/5,P6/7,P8/9=parameters)
```

5.13.9 – Shell Commands

```
MSC_SHL_CHRINP    equ 64    char is requested (P1=channel
                            [0 → Standard, 1 → Keyboard])
MSC_SHL_STRINP    equ 65    line is requested (P1=channel
                            [0 → Standard, 1 → Keyboard],
                            P2=RAM bank, P3/4=address)
MSC_SHL_CHROUT    equ 66    char should be writtten (P1=channel
                            [0 → Standard, 1 → Screen], P2=char)
MSC_SHL_STROUT    equ 67    line should be writtten (P1=channel
                            [0 → Standard, 1 → Screen], P2=RAM
```

		bank, P3/4=address, P5=length)
MSC_SHL_EXIT	equ 68	application released focus or quit itself (P1 ⟶ 0=quit, 1=blur)

5.13.10 – Shell Responses

MSR_SHL_CHRINP	equ 192	char has been received (P1=EOF–flag [0=no EOF], P2=char, P3=error status)
MSR_SHL_STRINP	equ 193	line has been received (P1=EOF–flag [0=no EOF], P3=error status)
MSR_SHL_CHROUT	equ 194	char has been written (P3=error status)
MSR_SHL_STROUT	equ 195	line has been written (P3=error status)

5.13.11 – Screensaver Messages

MSC_SAV_INIT	equ 1	initialises the screen saver (P1=bank of config data, P2/3=address of config data [64bytes])
MSC_SAV_START	equ 2	start screen saver
MSC_SAV_CONFIG	equ 3	open screen savers own config window (at the end the screen saver has to send the result back to the sender)
MSR_SAV_CONFIG	equ 4	returns user adjusted screen saver config data (P1=bank of config data, P2/3=address of config data [64bytes])

5.13.12 – Desktop Actions

DSK_ACT_CLOSE	equ 5	close button has been clicked or ALT+F4 has been pressed
DSK_ACT_MENU	equ 6	menu entry has been clicked (P8/9=menu entry value)
DSK_ACT_CONTENT	equ 14	a control of the content has been clicked (P3=sub spec [see dsk_sub...], P4=key or P4/5=Xpos within the window, P6/7=Ypos, P8/9=control value)
DSK_ACT_TOOLBAR	equ 15	A control of the toolbar has been clicked (see DSK_ACT_CONTENT)

DSK_ACT_KEY	equ 16	Key has been pressed without touching/ modifying a control (P4=ASCII Code)
DSK_SUB_MLCLICK	equ 0	left mouse button has been clicked
DSK_SUB_MRCLICK	equ 1	right mouse button has been clicked
DSK_SUB_MDCLICK	equ 2	double click with the left mouse button
DSK_SUB_MMCLICK	equ 3	middle mouse button has been clicked
DSK_SUB_KEY	equ 7	keyboard has been clicked and did modify/click a control (P4=ASCII Code)
DSK_SUB_MWHEEL	equ 8	mouse wheel has been moved (P4=Offset)

5.13.13 – Desktop Services

DSK_SRV_MODGET	equ 1	get screen mode (output P2=mode, P3=virtual desktop)
DSK_SRV_MODSET	equ 2	set screen mode (input P2=mode, P3=virtual desktop)
DSK_SRV_COLGET	equ 3	get colour (input: P2=number, output: P2=number, P3/4=RGB value)
DSK_SRV_COLSET	equ 4	set colour (input: P2=number, P3/4=RGB value)
DSK_SRV_DSKSTP	equ 5	Freeze desktop (input P2=type [0=Pen0, 1=Raster, 2=background, 255=no screen modification, switch off mouse])
DSK_SRV_DSKCNT	equ 6	continue desktop
DSK_SRV_DSKPNT	equ 7	clear desktop (Eingabe P2=Typ [0=Pen0, 1=Raster, 2=background])
DSK_SRV_DSKBGR	equ 8	initialize and redraw desktop background
DSK_SRV_DSKPLT	equ 9	redraw the complete desktop

5.13.14 – Jumps

jmp_memsum	equ 8100H	MEMSUM
jmp_sysinf	equ 8103H	SYSINF
jmp_clcnum	equ 8106H	CLCNUM
jmp_mtgcnt	equ 8109H	MTGCNT
jmp_timget	equ 810CH	TIMGET
jmp_timset	equ 810FH	TIMSET

```
jmp_memget      equ 8118H   MEMGET
jmp_memfre      equ 811BH   MEMFRE
jmp_memsiz      equ 811EH   MEMSIZ
jmp_meminf      equ 8121H   MEMINF
jmp_bnkrwd      equ 8124H   BNKRWD
jmp_bnkwwd      equ 8127H   BNKWWD
jmp_bnkrbt      equ 812AH   BNKRBT
jmp_bnkwbt      equ 812DH   BNKWBT
jmp_bnkcop      equ 8130H   BNKCOP
jmp_bnkget      equ 8133H   BNKGET
*empty*         equ 8136H   *empty*
jmp_scrget      equ 8139H   SCRGET
jmp_mosget      equ 813CH   MOSGET
jmp_moskey      equ 813FH   MOSKEY
jmp_bnk16c      equ 8142H   BNK16C
jmp_keytst      equ 8145H   KEYTST
jmp_keysta      equ 8148H   KEYSTA
jmp_keyput      equ 814BH   KEYPUT
jmp_bufput      equ 814EH   BUFPUT
jmp_bufget      equ 8151H   BUFGET
jmp_bufsta      equ 8154H   BUFSTA
jmp_iominp      equ 8157H   IOMINP (cpc only)
jmp_iomout      equ 815AH   IOMOUT (cpc only)

jmp_bnkcll      equ FF03H   BNKCLL
jmp_bnkret      equ FF00H   BNKRET
```

5.13.15 – Filemanager Functions (call via MSC_SYS_SYSFIL)

```
FNC_FIL_STOINI      equ 000
FNC_FIL_STONEW      equ 001
FNC_FIL_STORLD      equ 002
FNC_FIL_STODEL      equ 003
FNC_FIL_STOINP      equ 004
FNC_FIL_STOOUT      equ 005
FNC_FIL_STOACT      equ 006
FNC_FIL_STOINF      equ 007
FNC_FIL_STOTRN      equ 008
```

```
FNC_FIL_DEVDIR  equ 013
FNC_FIL_DEVINI  equ 014
FNC_FIL_DEVSET  equ 015

FNC_FIL_FILINI  equ 016
FNC_FIL_FILNEW  equ 017
FNC_FIL_FILOPN  equ 018
FNC_FIL_FILCLO  equ 019
FNC_FIL_FILINP  equ 020
FNC_FIL_FILOUT  equ 021
FNC_FIL_FILPOI  equ 022
FNC_FIL_FILF2T  equ 023
FNC_FIL_FILT2F  equ 024
FNC_FIL_FILLIN  equ 025

FNC_FIL_DIRDEV  equ 032
FNC_FIL_DIRPTH  equ 033
FNC_FIL_DIRPRS  equ 034
FNC_FIL_DIRPRR  equ 035
FNC_FIL_DIRREN  equ 036
FNC_FIL_DIRNEW  equ 037
FNC_FIL_DIRINP  equ 038
FNC_FIL_DIRDEL  equ 039
FNC_FIL_DIRRMD  equ 040
FNC_FIL_DIRMOV  equ 041
FNC_FIL_DIRINF  equ 042
```

6 – UZIX

6.1 – COMMANDS

6.1.1 – Conventions

COMMAND NAME (type of command)
  Format: Valid formats for the command
  Function: Way of operating the command
  Details: Describes some details about the format

Uzix commands are all loaded from disk. This guide describes all commands and utilities that are installed by default on UZIX 2.0.

6.1.1.1 – Format Notations

<filename>
Filename in the form: dir1/dir2/file

<filenames>
Several filename in the form: dir1/dir2/file

<dirname>
Directory name in the form: /dir1/dir2/

[ ] Delimits optional parameter.
 | It means that only one of the items can be used.

A <device> can be:

fd0~fd7	Disk drives.
null	Null device.
lpr	Printer.
tty/tty0~tty2	Monitor.
console	Keyboard.
mem/kmem	Memory.
sga0~sga(n)	Hard disk partitions.
sge(n)	Hard disk partition where the UZIX is.

6.1.2 – Commands Description

ADDUSER (Administration Utility)
Format: adduser
Function: Add a user to the system.

ALIAS (Shell Utility)
Format: alias [<name> [<command> [<command> ...]]]
Function: Presents or sets an alias command.

BANNER (Uzix Utility)
Format: banner <message>
Function: Print a message in big chars.

BASENAME (Shell Utility)
Format: basename <name> [suffix]
Function: Removes component orientation from a directory.

BOGOMIPS (System Utility)
Format: bogomips
Function: Prints processing speed on BogoMips.

CAL (Uzix Utility)
Format: cal [month] year
Function: Shows a calendar.

CAT (Files Utility)
Format: cat <filenames>
Function: Concatenate files and print to standard output.

CD (Files Utility)
Format: cd [<dirname>]
Function: Change directories.

CDIFF (Text Utility)
Format: cdiff [–c n] <file1> <file2>
Function: Prints the difference between two files with context.
Details: [–c] Produces output containing n lines of context.

CGREP (Text Utility)
Format: cgrep [–a n] [–b n] [–f] [–l n] [–n] [–w n] <pattern> [<arqs>...]
Function: Search a string and print the lines where it was found.
Details: [–a] Number of lines to print after the found line.
[–b] Number of lines to print before the found line.
[–f] Suppress filename on output.
[–l] Truncates lines at length n before comparison.
[–n] Supress linenumbers on output.
[–w] Set the window size (same as –a e –b)

CHGRP (Files Utility)
Format: chgrp <gid> <filename>
Function: Changes the group owning user for each file.

CHMOD (Files Utility)
Format: chmod <modo_ascii> | <modo_octal> <filenames>
Function: Change file access permissions.
Details: The symbolic format (ASCII) for the mode is as follows:
[ugoa] [+ | –] [rwx], where:

u ⟶ user	a ⟶ all	x ⟶ record
g ⟶ group	r ⟶ read	+ ⟶ add permission
o ⟶ others	w ⟶ write	– ⟶ remove permission

The numeric format (octal) is the following:
1º octal digit: 1 – save image text of attributes
2 – group ID
4 – user ID
2º octal digit: 1 – execution
2 – write
4 – read

CHOWN (Files Utility)
Format: chown <uid> <filename>
Function: Changes the regular user and the group owning user to the specified file.

CHROOT (Files Utility)
Format: chroot <dirname>
Function: Change the root directory.

CKSUM (Files Utility)
Format: cksum [<filename> [filename ...]]
Function: Shows the checksum and file size.

CLEAR (Shell Utility)
Format: clear
Function: Clears the screen.

CMP (Files Utility)
Format: cmp <filename1> <filename2>
Function: Compare files.

CRC (Files Utility)
Format: crc [<filename> [filename ...]]
Function: Shows the checksum of the data files.

CP (Files Utility)
Format: cp [–pifsmrRvx] <filename1> <filename2>
cp [–pifsrRvx] <filename1> [<filename2>...] <dir>
Function: Copy files.
Details: [–p] Preserves all attributes of the original file.
[–i] Checks the destination for file with the same name.
[–f] Remove files in destination.
[–s] Copies only some attributes.
[–m] Copies multiple subdirectories into one.
[–r] Copy directories recursively.
[–R] Copies directories and treats special files as ordinary.
[–v] Displays file names before copying.
[–x] Skip directories that are on file systems other than where copying started.

CPDIR (Files Utility)
Format: cpdir [–ifvx] <dirname1> <dirname2>
Function: Copy directories.
Details: [–i] Checks the destination for file with the same name.
[–f] Remove files in destination.
[–v] Displays file names before copying.
[–x] Skips subdirectories that are on file systems other than where copying started.

DATE (Uzix Utility)
Format: date
Function: Displays the current system date and time.

DD (Files Utility)
Format: dd [if=<filename>] [of=<filename>] [ibs=<bytes>] [obs=<bytes>] [bs=<bytes>] [cbs=<bytes>] [files=<number>] [skip=<blocks>] [seek=<blocks>] [count=<blocks>] [conv={ascii | ebcdic | ibm | lcase | ucase | swab | noerror | sync}]
Function: Copy file converting it.
Details:

[if=<filename>]	Read from file
[of=<filename>]	Write to file
[ibs=<bytes>]	Read <bytes> bytes at a time
[obs=<bytes>]	Write <bytes> bytes at a time
[bs=<bytes>]	Reads and writes <bytes> bytes at a time
[cbs=<bytes>]	Converts <bytes> bytes at a time
[files=<num.>]	Copies <num.> files
[skip=<blocks>]	Skip <blocks> blocks of "bs" size at the beginning of the entry
[seek=<blocks>]	Skip <blocks> blocks of"bs" size at start of output
[count=<blocks>]	Copies only <blocks> of size"bs" into the input

conv=conversion[,conversion...] → converts the file according to the following arguments:

ascii	Convert from EBCDIC to ASCII.
ebcdic	Convert from ASCII to EBCDIC.
ibm	Convert from ASCII to alternative EBCDIC.
lcase	Converts all characters to lowercase.
ucase	Converts all characters to uppercase.
swab	Swaps a pair of input bytes.
noerror	Continue after detecting an error.
sync	Completes a"bs" block with 00H bytes.

DF (Files Utility)
Format: df [–ikn]
Function: Print the free disk space in units of 512 bytes.
Details: [–i] List information used by inodes.
[–k] Print in units of 1 Kbyte.
[–n] Not access /etc/mtab to obtain information.

DHRY (System Utility)
Format: dhry
Function: Displays processing speed in dhrystones.

DIFF (Text Utility)
Format: diff [–c | –e | –C n] [–br] <filename1> <filename2>
Function: Print the difference between two files.
Details: [–C n] Produces output containing n lines of context.
[–b] Ignores white space in the comparison.
[–c] Produces an output containing 3 lines of context.
[–e] Produces an "ed–script" to convert.
[–r] Applies diff recursively.

DIRNAME (Shell Utility)
Format: dirname <filename>
Function: Print the filename suffix.

DOSDEL (Uzix Utility)
Format: dosdel <drivedos><filenamedos>
Function: Erase a file in MSXDOS disks.

DOSDIR (Uzix Utility)
Format: dosdir [–lr] <drivedos>
Function: List files of the an MSXDOS disk.
Details: [–l] Long listing.
[–r] Prints subdirectories recursively and descending.

DOSREAD (Uzix Utility)
Format: dosread [–a] <drivedos><filenamedos> [<filenameuzix>]
Function: Read file from MSXDOS disk.
Details: [–a] ASCII file.

DOSWRITE (Uzix Utility)
Format: doswrite [–a] <drivedos><filenamedos> [<filenameuzix>]
Function: Write a file to MSXDOS disk.
Details: [–a] ASCII file.

DU (Uzix Utility)
Format: du [–as] [–l n] <dirname> ...
Function: Print space occupied by directories and subdirectories.
Details: [–a] Print space used by all files.
[–s] Summary only.
[–l] List n subdirectories levels.

ECHO (Shell Utility)
Format: echo [–ne] [<string> [<string>...]]
Function: Print a text line.
Details: [–n] Does not feed a line at the end of the text
[–e] Enables interpretation of the following characters:
\a Alert (bell).
\b Backspace.
\c Suppress line feed.
\f Form feed.
\n New line.
\r Carriage return.
\t Horizontal tab.
\v Vertical tab.
\ \ Ignores space in the text between \ \ (backslash).
\nnn Print char of ASCII code nnn (octal).
\xnn Print char of ASCII code nn (hex).

ED (Text Utility)
Format: ed [–Ghs] [–p string] [arquivo]
Function: Execute a standard text editor.
–G Forces retrocompatibility.
–h Shows the program help.
–s Supress diagnostics.
–p Sets a command prompt.

EXIT (Administration Utility)
Format: exit [<status>]
Function: Exit the current session.

FALSE (Shell Utility)
Format: false
Function: Null; only returns with error status "1".

FGREP (Text Utility)
Format: fgrep [–cfhlnsv] [<string_file>] [<string>] <filename>]...
Function: Searches for a string and prints the lines where it was found.
Details: [–c] Prints only the number of lines.
[–f] Searches for string in file <filename>.
[–h] Omit file headers from output.
[–l] Lists file names only once.
[–n] Prints line numbers for each line.
[–s] Status only.
[–v] Print only lines without the <string>.

FILE (Uzix Utility)
Format: file <filename> [<filename>...]
Function: Makes an assumption about what type the file is.

FLD (Text Utility)
Format: fld –u –z* –[b t s? i? fm1.n1,m2.n2] {<input_file>
[<output_file>] }
Function: Reads and concatenates fields from a file.
Details: [–?] Show help. Same as [–h].
–u Unzips tabs.
[–p] Compress tabs.
–z* Skip the first * spaces.
[–b] Skip the starting spaces of the field.
[–t] Removes excessive spaces from the field.
[–s?] Field separator on output will be "?".
[–i?] Field separator on input will be "?".
[–fm1.n1,m2.n2] Field definition:
m1.n1 = beginning of field and m2.n2 = end of field,
where m = number of fields and n = number of chars.
[–f#] Get the user input field.

FORTUNE (Uzix Utility)
Format: fortune
Function: Randomly prints a proverb.

GREP (Text Utility)
Format: grep –cnfv {–p<padrão>] <filenames>
Function: Searches for a string and prints the lines where found.
Details: [–c] Prints only the number of lines.
[–f] Print file names.
[–n] Prints line numbers for each line
[–v] Print only lines without the <string>
[–p] Sets the string (default). The following control characters can be used:

- x Ordinary character.
- \ Quote any character.
- ^ Start of line.
- $ End of line.
- . Any character.
- :l Lowercase.
- :u Capital letters.
- :a Alphabetical.
- :d Digits (numeric).
- :n Alphanumeric.
- :r Russian characters.
- :s Space.
- :t Tab.
- :c Control characters (except LF and TAB).
- :e Starts sub-expression.
- * Repeats zero or more.
- + Repeats one or more.
- – Optionally search for expression.
- [..] Any of these (in the FROM–TO range).
- [^..] Any except these.
- \nnn Numeric value (C style).

HEAD (Text Utility)
Format: head [–n] [<filenames> ...]
Function: Prints the beginning of the text.
Details: [–n] Number of lines ti print (the standard is 10).

HELP (Uzix Utility)
Format: help
Function: Prints some commands in their format.

INIT (Administration Utility)
Format: /bin/init
Function: Process startup control.

KILL (Uzix Utility)
Format: kill [–signal] pid [pid...]
Function: Ends system processes.
Details: [–signal] is a signal to be sent to a process that is running (eg HUP, INT, QUIT, KILL or 9).

LOGIN (Administration Utility)
Format: login <username>
Function: Start a session.

LN (Text Utility)
Format: ln [–ifsSmrRvx] <filename1> <filename2>
ln [–ifsSrRvx] <filename> [<filename>...] <dirname>
Function: Add links between files.
Details: [–i] Warn before removing existing destination files.
[–f] Removes existing destination files.
[–s] Add symbolic link.
[–S] Add symbolic link while trying normal link.
[–m] Interleaves trees.
[–r] Adds recursive link to directories.
[–R] Same as [–r].
[–v] Print file name before adding link.
[–x] Skips subdirectories that are on file systems other than where adding links started.

LOGOUT (Uzix Utility)
Format: logout
Function: Ends a session.

LS (Files Utility)
Format: ls [–1ACFLRacdfgiklqrstu] [<filename> [<filename>...]]
Function: List the directory contents.
Details: [–1] Use only one column in the output.
[–A] Lists all files except "." and "..".

[–C] Sorts files in the listing (in columns).
[–F] Does not identify the file type.
[–L] Lists files by symbolic links.
[–R] Lists the contents of directories recursively.
[–a] Lists all files including “.” and "..".
[–c] Sorts files by change date.
[–d] List directories like other files.
[–f] Does not sort files and directories.
[–g] Prints the name of the user who owns the group.
[–i] Prints the inode number of files.
[–k] Print file size in Kbytes.
[–l] Print file attributes.
[–q] Prints question marks in place of special characters.
[–r] Sort files and directories in reverse order.
[–s] Print file size in bytes.
[–t] Sorts files by creation date.
[–u] Sorts files by last access date.

MAN (System Utility)
Format: man –wqv [seção] <commandname>
Function: Presentes qhe on-line manual.
Details: –w Displays only manual with exact section/name.
–q Silent mode, for faulty formatter commands.
–v Formatted presentation mode (verbose).

MKDIR (Files Utility)
Format: mkdir [–p] [–m <mode>] <dirname>
Function: Create directories.
Details: [–p] Create parent directories according to the mask.
[–m] Sets the mode (0666 minus umask bits).

MKNOD (Files Utility)
Format: mknod [–m <mode>] <filename> {b | c | u} <major> <minor>
Function: Create special files.
Details: [–m] Define the mode.
b Bufferized file (block).
c/u Not bufferized file (character).

MORE (Uzix Utility)
Format: more <filenames>
Function: Paging utility.
Details: When the prompt is present, use the following keys:
space Displays the next page.
return Displays the next line.
n Go to the next file if it exists.
p Go to the previous file if it exists.
q Quits the 'more' command.

MOUNT (Uzix Utility)
Format: mount [–r] <device> <path>
Function: Mounts the <device> in the specified <path>.
Details: [–r] Mounts in the read only mode.

MV (Files Utility)
Format: mv [–isfmvx] <filename1> <filename2>
mv [–ifsvx] <filename> [<filename> ...] <dirname>
Function: Rename or move files.
Details: [–i] Warn before overwriting files with same name.
[–f] Removes existing target files.
[–s] Creates symbolic link and does not move the file.
[–m] Merge directories without searching target directory.
[–v] Print file name before moving.
[–x] Skips subdirectories that are on file systems other than where file movement started.

PASSWD (Administration Utility)
Format: passwd [<login>]
Function: Change user password.

PROMPT (Shell Utility)
Format: prompt <string>
Function: Change the Uzix prompt.

PS (Uzix Utility)
Format: ps [–] [lusmahrn]
Function: Prints a process status report.

Details: [–l] Long format.
[–u] User format (username and start time).
[–s] Signal format.
[–m] Memory information.
[–a] Displays processes from other users as well.
[–h] No header.
[–r] Only running processes.
[–n] Numeric output for user.

PWD (Shell Utility)
Format: pwd
Function: Prints the path of the current working directory.

QUIT (Administration Utility)
Format: quit
Function: Ends current session.

REBOOT (Administration Utility)
Format: reboot
Function: Restart the computer.

RM (Files Utility)
Format: rm <filename>
Function: Remove files.

RMDIR (Files Utility)
Format: rmdir [–p] <dirname>
Function: Remove directories.
Details: [–p] Remove parent directory if empty after removal from the specified directory.

SASH (Application Utility)
Format: sash
Function: It's a kind of shell with built-in commands.

SET (Administration Utility)
Format: [<name> [<value>]]
Function: Displays or sets environment variables.

SLEEP (Administration Utility)
Format: sleep [<seconds>]
Function: Makes the system "sleep" for <seconds> seconds.

SU (Administration Utility)
Format: su [<username>]
Function: Temporarily connect as superuser or other user.

SOURCE (Uzix Utility)
Format: source <filename>
Function: Displays the source of the file.

SUM (Files Utility)
Format: sum [<filename> [<filename>...]]
Function: Analyze the checksum and block counter of the file.

SYNC (Programming Utility)
Format: sync
Function: Unloads file system buffers.

TAIL (Text Utility)
Format: tail [−c n | −n n] [−f] [<filename> [<filename>]]
Function: Prints the last lines of a file.
Details: [−c] Print n characters.
[−f] In FIFO or special file, read after EOF.
[−n] Print n lines.

TAR (Files Utility)
Format: tar [cxt] [voFfpD] <filenametape> [<filename> [<filename>...]
Function: Concatenate/extract files for storage.
Details: [c] Create new tar file.
[x] Extract files from the tar archive.
[t] Lists the contents of the tar file.
[v] Verbose mode.
[o] Defines original user and owner on extraction.
[F] Ignores errors.
[f] Next argument is the name of the tar file.
[p] Restore file modes, ignore mask.
[D] Don't recursively add directories.

TEE (Shell Utility)
Format: tee <filename>
Function: Reads from standard input and writes to a file.

TIME (Uzix Utility)
Format: time <command> [<command arguments>]
Function: Executes the command and prints the real time, user time, and system time (hours–minutes–seconds).

TOP (Uzix Utility)
Format: top [–d <delay>] [–q] [–s] [–i]
Function: Lists the most active processes.
Details: [–d] Specifies the time for screen refresh.
[–q] Specifies update without any delay.
[–s] Safe Mode (disables interactive commands).
[–i] Ignore idle processes.

TOUCH (Files Utility)
Format: touch [–c] [–d <time/date>] [–m] <filename>
Function: Change the time and date of files.
Details: [–c] Does not create files that do not exist.
[–d] Change to <time/date> instead of using current time/date. Format: HH:MM:SS DD:MM:YY.
[–m] Changes only the file modification time/date.

TR (Text Utility)
Format: tr from to [+<start>] [–<end>] [<inputfile> [<outuputfile>]]
Function: Swaps characters in a file (transliterates).
Details: Escape Sequences:

:z	Empty range	:a	Same as a–zA–Z
:l	Same as a–z	:u	Same as A–Z
:m	Same as á–∩	:b	Same as Ç–f
:r	Same as á–∩Ç–f	:d	Same as 0–9
:n	Same as a–zA–Z0–9	:s	Same as \001–\040
: .	All ASCII range minus \0		

TRACE (Uzix Utility)
Format: trace {on}
Function: Trace mode?

TRUE (Shell Utility)
Format: true
Function: Null; only returns with error status "0".

UMOUNT (Uzix Utility)
Format: umount <device>
Function: Unmounts file system from the specified device.

UMASK (Uzix Utility)
Format: umask [<mask>]
Function: Remove masks.

UNALIAS (Shell Utility)
Format: unalias <name>
Function: Removes an alias command.

UNAME (Shell Utility)
Format: uname [−snrvma]
Function: Prints system information.
Details: [−m] Print machine type.
[−n] Prints client machine name on the network.
[−r] Print operating system distribution.
[−s] Print operating system name.
[−v] Print operating system version.
[−a] Prints all of the above items.

UNIQ (Text Utility)
Format: uniq [−cduzN.M+L] [−<fields>] [+<letters>] [<filename>]
Function: Remove duplicate lines in sorted files.
Details: [−u] Only print unrepeated lines.
[−d] Only print duplicate lines.
[−c] Prints the number of times the line is repeated.
[−z] Same as −c, but prints in octal numbers.
[−N.M] Skip N words and M letters.
[+L] Compares only L letters.

WC (Text Utility)
Format: wc [−bhpw] [<filename>]
Function: Prints the number of bytes, words and lines in a file.

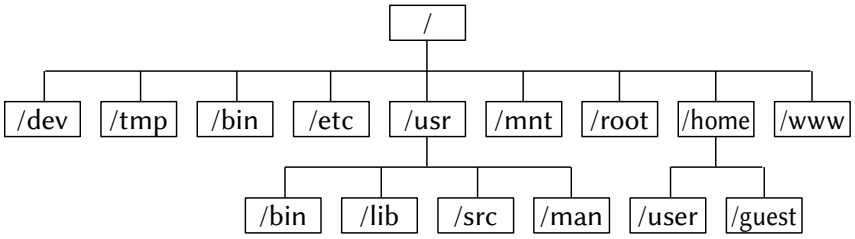
Details: [–b] Open file in binary mode.
[–h] Displays program help.
[–p] Page count.
[–w] Finds the maximum line width.

WHOAMI (Shell Utility)
Format: whoami
Function: Prints the username associated with the current userid.

YES (Shell Utility)
Format: yes [<string>]
Function: Prints "y" or <string> repeatedly to standard output.

6.2 – HIERARCHICAL STRUCTURE

In Uzix there is a pre-defined structure of subdirectories. This structure can be modified by the user, but it is not advisable to do so because it is standard in the Unix world. This structure is as follows:



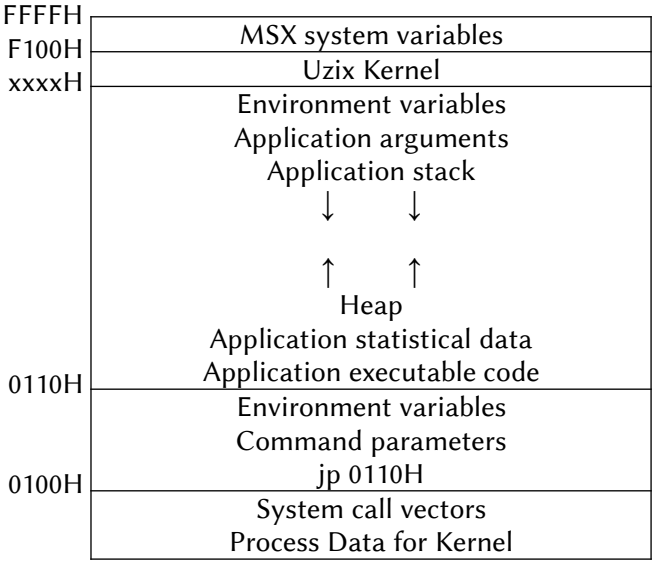
Each of these subdirectories has a specific, but not mandatory, use. A description of each is below.

/ Root directory
/dev Contains the special filenames associated with hardware or software devices.
/tmp Used by all system for creating temporary files.
/bin Contains the most generic applications on the system.
/etc Files used to administer the system.
/usr General system files. This subdirectory contains more 4 subdirectories:

/bin Generic Applications.
/lib Libraries.
/src Source codes.
/man System manuals (text files).
/mnt Used as a connection point for a system of file from another device. Also used for mounting.
/root System administrator working directory.
/home Used by regular users as their desktop.
/user User "user".
/guest User "guest".
/www Internet files.

6.3 – MEMORY MAPPING

Memory mapping is the biggest difference between Uzix 1.0 and 2.0. It is illustrated below, where xxxxH is 8000H for Uzix 1.0 and C000H for 2.0.



Uzix 1.0 is entirely resident in the high memory area, starting at address 8000H. Every process always occupies 32 Kbytes of memory. The

Uzix 2.0 has a resident part on page 3 (from C000H) and makes the additional calls from there. Each process can be 16K, 32K or 48K.

6.4 – SYSTEM CALLS

Uzix is an operating system for MSX that implements AT&T Unix Version 7 functionality. It is a multi-user system and implements preemptive multitasking, while also offering network infrastructure (TCP/IP). However, the following precautions must be taken:

⟶ NEVER use DI and EI instructions;
⟶ NEVER access the hardware directly;
⟶ NEVER access data below 0100H or above the application.

To make a system call it is necessary to stack the parameters in the reverse order of the declaration, then the call number and then making a CALL 08H. It is the application's responsibility to unstack the parameters after the CALL. The 16-bit return value is placed in the DE register. The only exception is the lseek call, whose return value is 32 bits and is placed in HL:DE (HL is the most significant word). The table below lists the direct calls, their parameters and call number.

6.4.1 – Direct System Calls

ACCESS (#00) – Determines the access level of a file.

```
Syntax:     err = access (path, mode)
            int     err
            char    *path
            int     mode
Input:      path:   String pointing to the file to be analyzed.
            mode:   0 – Tests that the file exists and is searchable.
                    1 – Execute.
                    2 – Write.
                    4 – Read.
Output:     err:    0  ⟶  Successful test (if mode = 0).
                    –1 ⟶ Error (error code in errno).
Assembler:  (access = 33.)
            sys access; name; mode
```

ALARM (#01) – Schedules a signal after a specified time.

Syntax:	time = alarm (secs)	
	int	time
	int	secs
Input:	secs:	Time in seconds (maximum 32 767).
Output:	time:	Previous time remaining in alarm.
Assembler:	(alarm = 27.)	
	(seconds in r0)	
	sys alarm	
	(previous amount in r0)	
Note:	Causes the SIGALRM signal to be sent to the calling process after the number of seconds given by the argument. Unless captured or ignored, the signal ends the process. The return value is the amount of time remaining previously.	

BRK (#02) – Change core allocation.

Syntax:	err = brk (addr)	
	int	err
	char	*addr
Input:	addr:	Adress.
Output:	err:	0 ⟶ Command executed successfully.
		−1 ⟶ The program needs more memory than the system limit or overflows the maximum number of segmentation records.
Assembler:	(brk = 17.)	
	sys break; addr	
Note:	Defines, for the system, the lowest location not used by the program (called the range) for addr. Usually only growing programs whose data areas increase need to break.	

CHDIR (#03) – Change default directory.

Syntax:	err = chdir (path)	
	int	err
	char	*path
Input:	path:	String of the directory to be defined.
Output:	err:	0 ⟶ Command executed successfully.
		−1 ⟶ “path” is not a directory or is not searchable.
Assembler:	(chdir = 12.)	
	sys chdir; dirname	

CHMOD (#04) – Change mode of file.

```
Syntax:     err = chmod (path, mode)
            int     err
            char    *path
            int     mode
Input:      path:   String pointing to the file to be changed.
            mode:   Resulting from an OR combining the following
                    values:
                    04 000  Set user ID on execution.
                    02 000  Set group ID on execution.
                    01 000  Save text image after execution.
                    00 400  Read by owner.
                    00 200  Write by owner.
                    00 100  Execute (search on directory) by owner.
                    00 070  Read, write, execute (search) by group.
                    00 007  Read, write, execute (search) by others.
Output:     err:    0 ⟶ Command executed successfully.
                    −1 ⟶ File not found or user not allowed access.
Assembler:  (chmod = 15.)
            sys chmod; name; mode
```

CHOWN (#05) – Change owner and group of a file.

```
Syntax:     err = chown (path, owner, group)
            int     err
            char    *path
            int     owner
            int     group
Input:      path:   String pointing to the file.
            owner:  New file user.
            group:  New file group.
Output:     err:    0 ⟶ Owner is changed.
                    −1 ⟶ Illegal owner changes.
Assembler:  (chown = 16.)
            sys chown; name; owner; group
```

CLOSE (#06) – Close a file.

```
Syntax:     err = close(path)
            int     err
            char    *path
```

Input: path: String pointo to the file to be closed.
Output: err: 0 ⟶ File succesfully closed.
−1 ⟶ Unknown file descriptor.
Assembler: (close = 6.)
(file descriptor in r0)

GETSET (#07) – Implements calls that read or change system variable values.

```
Syntax:    var = getset(operation, ...)
           int     var
           operation ⟶ Depends on the called function.
Input:     getset(0)        ⟶   getpid(void)
           getset(1)        ⟶   getppid(void)
           getset(2)        ⟶   getuid(void)
           getset(3,uid)    ⟶   setuid (uid)
                                 int   uid
           getset(4)        ⟶   geteuid(void)
           getset(5)        ⟶   getgid(void)
           getset(6,gid)    ⟶   setgid(int gid)
           getset(7)        ⟶   getegid(void)
           getset(8)        ⟶   getprio(void)
           getset(9,pid,prio) ⟶ setprio(pid, prio)
                                 int   pid
                                 char  prio
           getset(10)       ⟶   umask(mask)
                                 int   mask
           getset(11,onoff) ⟶   systrace(onoff)
                                 int   onoff
Output:    It depends on the called function.
Assembler: It depends on the called function.
```

DUP (#08) – Duplicate an open file descriptor.

Syntax: newd = dup(oldd)
int newd
int oldd
Input: oldd: Old file descriptor.
Output: newd: New file descriptor.
−1 if the descriptor is invalid or if there are already too many files open.

Assembler: (dup = 41.)
(file descriptor in r0)
(new file descriptor in r1)
sys dup
(file descriptor in r0)

DUP2 (#09) – Duplicate an open file descriptor.
Syntax: err = dup2(oldd, newd)
int newd
int oldd
Nota: The dup2 entry is implemented by adding 0100 to oldd.

EXECVE (#10) – Execute a file.
Syntax: err = execve(name, argv, envp)
int err
char *name
char **argv
char **envp
Input: name: Name of the file to be executed.
argv: Array of pointers to arguments.
envp: Pointer to an array of strings that constitute the process environment.

EXIT (#11) – Ends a process.
Syntax: param = exit(status)
int param
int status
Input: status: Lowest byte (LSB) is passed to "param".
Output: param: Receives the lowest byte of "status".
Assembler: (exit = 1.)
(status in r0)
sys exit

FORK (#12) – Generate a new process.
Syntax: newp = fork(void)
int newp
Input: None.
Output: newp: ID of the created process. If it returns –1, it failed in creating the process.

Assembler: (fork = 2.)
sys fork
(new process return)
(old process return, new process ID in r0)

FSTAT (#13) – Gets file information.
Syntax: stat = fstat(fd, *buf)
int stat
int fd
void *buf
Library: #include <sys/types.h>
#include <sys/stat.h>
Input: fd: File descriptor.
*buf: Pointer to an empty buffer.
Output: stat: Information obtained. These are the same as the open, creat, dup or pipe commands.

GETFSYS (#14) – Get system information.
Syntax: stat = getfsys(dev, *buf)
int stat
int dev
void *buf
Library: –
Input: dev: Device.
*buf: Pointer to an empty buffer.
Output: stat: Obtainet status.

IOCTL (#15) – Device control.
Syntax: err = ioctl(fd, req, ...)
int err
int fd
int req
Library: #include <sgtty.h>
Input: fd: File descriptor.
req: Request.
Output: err: 0 ⟶ Command successful.
–1 ⟶ The file descriptor does not refer to the type of file to which it was directed.
Assembler: (ioctl = 54.)
sys ioctl; fildes; request; argp

KILL (#16) – Sends the "sig" signal to the process specified in "r0".

```
Syntax:     err = kill(pid, sig)
            int     err
            int     pid
            int     sig
Input:      pid:    Process ID.
            sig:    Signal to be sent.
Output:     err:    0 ⟶ The process was ended.
                    −1 ⟶ The process does not exist or does not have
                         the same current userid or is not a superuser.
Assembler:  (kill = 37.)
            (process number in r0)
            sys kill; sig
Note:       If the process number is 0, the signal is sent to all other
            processes in the sender's process group.
```

LINK (#17) – Link to a file.

```
Syntax:     err = link(oldname, newname)
            int     err
            char    *oldname
            char    *newname
Input:      oldname:  Old filename.
            newname:  New filename.
Output:     err:    0 ⟶ Link sucessfully created.
                    −1 ⟶ Link creation failed.
Assembler:  (link = 9.)
            sys link; oldname; newname
```

MKNOD (#18) – Make a directory or a special file.

```
Syntax:     err = mknod(name, mode, dev)
            int     err
            char    *name
            int     mode
            int     dev
Input:      name: String pointing to the new file/directory.
            mode: New file/directory mode.
            dev:  Device.
Output:     err:    0 ⟶ File/directory created successfully.
                    −1 ⟶ File/directory already exists or user is not
                         superuser.
Note:       Only the superuser can use this command.
```

MOUNT (#19) – Mount the filesystem.

```
Syntax:     err = mount(spec, dir, rwflag)
            int     err
            char    *spec
            char    *dir
            int     rwflag
Library:    #include <sys/mount.h>
Input:      spec:   –
            dir:    –
            rwflag: –
Output:     err:    0 ⟶  Command executed successfully.
                    –1 ⟶ Error executing command.
```

OPEN (#20) – Open a file for read or write.

```
Syntax:     err = open(name, flags, mode)
            int     err
            char    *name
            int     flags
            int     mode
Input:      name: Name of file to be open.
            flags: –
            mode: 0 ⟶ Read only.
                  1 ⟶ Read/Write.
Output:     err:  0 ⟶  File successfully open.
                  –1 ⟶ Failed to open file.
Assembler:  (open = 5.)
            sys open; name; mode
            (file descriptor in r0)
```

PAUSE (#21) – Pauses the system.

```
Syntax:     err = pause(void)
            int     err
Input:      None.
Output:     None.
Assembler:  (pause = 29.)
            sys pause
Note:       This command never returns normally. It is used to pause
            the system until it receives a “kill” or “alarm” signal.
```

PIPE (#22) – Create an interprocess channel.

```
Syntax:     err = pipe(fd)
            int     err
            int     *fd
Input:      fd:     File descriptor.
Output:     err:    0 ⟶ Channel successfully created.
                    −1 ⟶ Channel creation failed.
Assembler:  (pipe = 42.)
            sys pipe
            (read file descriptor in r0)
            (write file descriptor in r1)
```

READ (#23) – Read from file.

```
Syntax:     err = read(fd, buf, bytes)
            int     err
            int     fd
            void    *buf
            int     bytes
Input:      fd:     File descriptor.
            buf:    Empty buffer.
            bytes:  Number of bytes to read.
Output:     err:    0 ⟶ End of file reached.
                    −1 ⟶ Read error.
Assembler:  (read = 3.)
            (file descriptor in r0)
            sys read; buffer; nbytes
            (byte count in r0)
```

SBRK (#24) – Change core allocation.

```
Syntax:     err = sbrk(int incr)
Nota:       Ver BRK (#02).
```

LSEEK (#25) – Move read/write pointer.

```
Syntax:     err = lseek(fd, offset, flag)
            int     err
            int     fd
            long    offset
            int     flag
```

Input: fd: File descriptor.
offset: Offset.
Flag: 0 → The pointer is set to offset bytes.
1 → The pointer is set to its current location plus offset.
2 → The pointer is set to the size of the file plus offset.
Output: err: 0 → Command executed successfully.
−1 → Error executing command.
Assembler: (lseek = 19.)
(file descriptor in r0)
sys lseek; offset1; offset2; whence
[Offset1 and offset2 are the high and low offset words; r0 and r1 contain the pointer on return].

SIGNAL (#26) – Catch or ignore signals.
Syntax: err = signal(sig_num, (*func)(int))
int err
char sig_num
void (*func)(int)
Input: sig_num: Signal number.
(*func)(int): –
Output: The value (int)−1 is returned if the given signal is out of range.
Note: List of signals with names as in the "include" file.

1	SIGHUP	Hangup
2	SIGINT	Interrupt
3*	SIGQUIT	Quit
4*	SIGILL	Illegal instruction (not reset when caught)
5*	SIGTRAP	Trace trap (not reset when caught)
6*	SIGIOT	IOT instruction
7*	SIGEMT	EMT instruction
8*	SIGFPE	Floating point exception
9	SIGKILL	Kill (cannot be caught or ignored)
10*	SIGBUS	Bus error
11*	SIGSEGV	Segmentation violation
12*	SIGSYS	Bad argument to system call

13	SIGPIPE	Write on a pipe or link with no one to read it
14	SIGALRM	Alarm clock
15	SIGTERM	Software termination signal
16		Unassigned

The starred signals in the list above cause a core image if not caught or ignored.

Assembler: (signal = 48.)
sys signal; sig; label
(old label in r0)

STAT (#27) – Get file status.

Syntax: err = stat(path, buf)
int err
char *path
void *buf

Library: #include <sys/types.h>
#include <sys/stat.h>

Input: path: Filename path.
buf: Empty buffer.

Output: err: –

STIME (#28) – Set system time.

Syntax: err = stime(tvec)
int err
int *tvec

Input: tvec: Time in seconds from 01/01/1970.

Output: err: 0 ⟶ Date and time set successfully.
−1 ⟶ Error in setting date and time .

Assembler: (stime = 25.)
(time in r0–r1)
sys stime

SYNC (#29) – Update super-block.

Syntax: err = sync(void)
int err

Input: None.

Output: None.

Assembler: (sync = 36.)
sys sync

TIME (#30) – Get time and date.

Syntax: void time(tloc)
int *tloc
Library: #include <sys/types.h>
#include <sys/timeb.h>
Input: None.
Output: tloc: Time in seconds from 01/01/1970.

TIMES (#31) – Get process times.

Syntax: t = times(struct tms *tvec)
int t
Input: None.
Output: See note.
Assembler: (times = 43.)
sys times; buffer
Note: Returns time information for the current process and for terminated child processes of the current process. All times are in 1/Hz seconds, where Hz = 60 or Hz = 50. After the call, the buffer will appear as follows:

```
struct tbuffer {
    long proc_user_time;
    long proc_system_time;
    long child_user_time;
    long child_system_time;
    };
```

UMOUNT (#32) – Unmount filesystem.

Syntax: err = umount(spec)
int err
char *spec
Input: –
Output: –
Assembler: –

UNLINK (#33) – Remove directory entry.

Syntax: err = unlink(path)
int err
char *path
Input: path: String with directory to be removed.

Output: err: 0 ⟶ Directory entry removed successfully.
−1 ⟶ Error removing directory.
Assembler: (unlink = 10.)
sys unlink; name

UTIME (#34) – Set file times.
Syntax: err = utime(path, utimbuf *buf)
int err
char *path
struct utimbuf *buf
Library: #include <sys/types.h>
Input: path: Filename path.
Buf: –
Output: –
Assembler: (utime = 30.)
sys utime; file; timep

WAITPID (#35) – Wait for process to change state.
Syntax: err = waitpid(pid, statloc, options)
int err
int pid
int *statloc
int options
Library: #include <sys/types.h>
#include <sys/wait.h>
Input: pid: Process ID.
statloc: Wait status.
options: <−1 Meaning wait for any child process whose process group ID is equal to the absolute value of pid.
−1 Meaning wait for any child process.
0 Meaning wait for any child process whose process group ID is equal to that of the calling process.
> 0 Meaning wait for the child whose process ID is equal to the value of pid.

WRITE (#36) – Write on a file.

```
Syntax:     err = write(fd, buf, nbytes)
            int     fd
            void    *buf
            int     nbytes
Input:      fd:     File descriptor.
            buf:    Buffer with nbytes contiguous bytes which are
                    written on the output file
            nbytes: Number of bytes to be written.
Output:     err:    0 ⟶ Writing successful.
                    −1 ⟶ Error during writing.
Assembler:  (write = 4.)
            (file descriptor in r0)
            sys write; buffer; nbytes
            (byte count in r0)
```

REBOOT (#37) – Restart system.

```
Syntax:     err = reboot(p1, p2)
            int     err
            char    p1
            char    p2
Library:    #include <sys/reboot.h>
            #include <unistd.h>
Input:      p1:     –
            p2:     –
Output:     err:    0 ⟶ Not applicable.
                    −1 ⟶ Restart error.
```

SYMLINK (#38) – Create a new name for a file.

```
Syntax:     err = symlink(oldname, newname)
            int     err
            char    *oldname
            char    *newname
Library:    #include <fcntl.h>
            #include <unistd.h>
Input:      oldname:    Old filename.
            newname:    New filename.
Output:     err:    0 ⟶ Command executed successfully.
                    −1 ⟶ Error creating name.
```

CHROOT (#39) – Change root directory.

```
Syntax:     err = chroot(path)
            int     err
            char    *path
Library:    #include <unistd.h>
Input:      path:   New path to root directory.
Output:     err:    0 ⟶ Command executed successfully.
                    −1 ⟶ Change error.
```

MOD_REG (#40)

```
Syntax:     err = mod_reg (sig, (func)())
            int     err
            int     sig
            int     (*func)()
```

MOD_DEREG (#41)

```
Syntax:     err = mod_dereg (sig)
            int     err
            int     sig
```

MOD_CALL (#42)

```
Syntax:     err = mod_call (sig, fnc, args, argsz)
            int     err
            int     sig
            int     fnc
            char    *args
            int     argsz
```

MOD_SENDREPLY (#43)

```
Syntax:     err = mod_sendreply (pid, fnc, r, rsz)
            int     err
            int     pid
            int     fnc
            char    *r
            int     rsz
```

MOD_REPLY (#42)

```
Syntax:     err = mod_reply (sig, fcn, r)
            int     err
            int     sig
            int     fcn
            char    *r
```

6.4.2 – Indirect System Call

CREAT – Create a new file.

```
Call:       creat(path, mode)
            char   *path
            int    mode
Syntax:     err =  open(path, 0x301, mode)
Input:      path:  Filename path.
            0x301: O_CREAT | O_TRUNC | O_WRONLY
            mode:  Mode.
Output:     err:   0 ⟶ File created successfully.
                   −1 ⟶ Error creating file.
Assembler:  (creat = 8.)
            sys creat; name; mode
            (file descriptor in r0)
```

6.4.3 – Calls via GETSET

GETPID – Get process ID.

```
Call:       getpid(void)
Syntax:     id = getset(0)
Library:    #include <unistd.h>
Input:      None.
Output:     id:    Process ID.
Assembler:  (getpid = 20.)
            sys getpid
            (pid in r0)
```

GETPPID – Get parent process ID.

```
Call:       getppid(void)
Syntax:     id = getset(1)
Library:    #include <unistd.h>
Input:      None.
Output:     id:    Process ID.
```

GETUID – Returns the real user ID of the current process.

```
Call:       getuid(void)
Syntax:     id = getset(2)
Library:    #include <unistd.h>
```

Input: None.
Output: id: Process ID.
Assembler: (getuid = 24.)
sys getuid
(real user ID in r0, effective user ID in r1)

SETUID – Set user and group ID.
Call: setuid(uid)
Syntax: err = getset(3, uid)
int err
int uid
Library: #include <unistd.h>
Input: uid: User process ID.
Output: err: 0 ⟶ ID successfully set.
−1 ⟶ Error setting ID.
Assembler: (setuid = 23.)
(user ID in r0)
sys setuid
(setgid = 46.)
(group ID in r0)
sys setgid

GETEUID – Returns the effective user ID of the calling process.
Call: geteuid(void)
Syntax: id = getset(4)
Library: #include <unistd.h>
Input: None.
Output: id: Process ID.

GETGID – Get the real group ID.
Call: getgid(void)
Syntax: id = getset(5)
Library: #include <unistd.h>
Input: None.
Output: id: Group ID.
Assembler: (getgid = 47.)
sys getgid
(real group ID in r0, effective group ID in r1)

SETGID – Set group identity.

```
Call:       setgid(gid)
Syntax:     err = getset(6, gid)
            int     err
            int     gid
Library:    #include <unistd.h>
Input:      gid:    Group ID.
Output:     err:    0 ⟶ ID set successfully.
                    −1 ⟶ Error in setting the ID.
Assembler:  (setgid = 46.)
            (group ID in r0)
            sys setgid
```

GETEGID – Return the effective group ID of the calling process.

```
Call:       getegid(void)
Syntax:     id = getset(7)
Library:    #include <unistd.h>
Input:      None.
Output:     id:     Group ID.
```

GETPRIO – Get the priority of a given process.

```
Call:       getprio(void)
Syntax:     prd = getset(8)
Library:    #include <sched.h>
Input:      None.
Output:     prd:    Priority.
```

SETPRIO – Set the priority of a process.

```
Call:       setprio(pid, prio)
Syntax:     err = getset(9, pid, prio)
            int     err
            int     pid
            char    prio
Library:    #include <unistd.h>
Input:      pid:    Process ID.
            prio:   Process priority.
Output:     err:    0 ⟶ Priority set successfully.
                    −1 ⟶ Error setting priority.
```

UMASK – Define a file creation mask.

```
Call:       umask(mask)
Syntax:     oldm = getset(10, mask)
            int     oldm
            int     mask
Input:      mask:  Mode. Only the lowest 9 bits are valid.
Output:     oldm:  Old mask value.
Assembler:  (umask = 60.)
            sys umask; complmode
```

SYSTRACE – Generate and apply protocols for system calls.

```
Call:       systrace(onoff)
Syntax:     err = getset(11, onoff)
            void    err
            int     onoff
Input:      onoff:  New protocol.
Output:     –
```

6.4.4 – TCP/IP module

The TCP/IP module implements a subset of IPv4 and allows Uzix to communicate with other systems that support the protocol. The TCP/IP module signature is 04950H, and it provides the functions listed below.

Call	C prototype	FNC#
ipconnect	int ipconnect(char mode, ip struct t *ipstruct)	1
ipgetc	int ipgetc(uchar socknum)	2
ipputc	int ipputc(uchar socknum, uchar byte)	3
ipwrite	int ipwrite(uchar socknum, uchar *bytes, int len)	4
ipread	int ipread(uchar socknum, uchar *bytes, int len)	5
ipclose	int ipclose(uchar socknum)	6
iplisten	int iplisten(int aport, uchar protocol)	7
ipaccept	int ipaccept(ip struct t *ipstruct, int aport, uchar block)	8
ping	int ping(uchar *IP, unsigned long *unused, uint len)	9
setsocktimeout	int setsocktimeout(uchar socknum, uint timeout)	10
ipunlisten	int ipunlisten(int aport)	11

ipgetpingreply	icmpdata t *ipgetpingreply(void)	12
gettcpinfo	tcpinfo t *gettcpinfo(void)	13
getsockinfo	sockinfo t *getsockinfo(uchar socknum)	14

The data type used are:

```
// protocol numbers (protocol for iplisten)
ICMP_PROTOCOL = 1
TCP_PROTOCOL  = 6
UDP_PROTOCOL  = 17

// open modes
TCP_ACTIVE_OPEN  = 255
TCP_PASSIVE_OPEN = 0

// protocols (ipconnect mode)
IPV4_TCP  = 1
IPV4_UDP  = 2
IPV4_ICMP = 3

// UDP modes
UDPMODE_ASC   = 1
UDPMODE_CKSUM = 2

// error codes
ECONTIMEOUT = 080H
ECONREFUSED = 081H
ENOPERM     = 082H
ENOPORT     = 083H
ENOROUTE    = 084H
ENOSOCK     = 085H
ENOTIMP     = 086H
EPROT       = 087H
EPORTINUSE  = 088H

// allowed states for sockstatus in sockinfo_t
TCP_CLOSED       = 000H
TCP_LISTEN       = 001H
TCP_SYN_SENT     = 042H
TCP_SYN_RECEIVED = 043H
TCP_ESTABLISHED  = 0C4H
TCP_FIN_WAIT1    = 045H
```

```
TCP_FIN_WAIT2    = 046H
TCP_CLOSE_WAIT   = 087H
TCP_CLOSING      = 008H
TCP_LAST_ACK     = 009H
TCP_TIMEWAIT     = 00AH
UDP_LISTEN       = 091H
UDP_ESTABLISHED  = 094H

ip_struct_t = { uchar remote_ip[4],
           uint remote_port,
           uint local_port }

icmpdata_t = {  uchar type,
                uchar icmpcode,
                unsigned long unused,
                uchar data[28], /* pad para 64 bytes */
                uint len;
                uchar sourceIP[4],
                uchar ttl }

tcpinfo_t = {   uchar IP[4],
                uchar dns1ip[4],
                uchar dns2ip[4],
                char datalink[5],
                char domainname[DOMSIZE=128],
                int used_sockets,
                int avail_sockets,
                int used_buffers,
                int avail_buffers,
                int IP_chksum_errors }

sockinfo_t = {  int localport,
                int remoteport,
                uchar remote_ip[4],
                char socketstatus, /* bit 7: permissao
                                         de escrita
                                      bit 6: estado de
                                         listen
                                      bits 3-0: estado
                                   */
                char sockettype,   /* TCP=1, UDP=2 */
                char sockerr,      /* codigo de erro */
                int pid }
```

6.4.5 – Error codes

Uzix system calls return a value greater than 0 on success and less than 0 on error. The error code is placed in the global variable (defined in the stub of the Uzix programs) **errno**. Listed below are possible error codes.

EPERM	1	Operation not permitted
ENOENT	2	No such file or directory
ESRCH	3	No such process
EINTR	4	Interrupted system call
EIO	5	I/O error
ENXIO	6	No such device or address
E2BIG	7	Arg list too long
ENOEXEC	8	Exec format error
EBADF	9	Bad file number
ECHILD	10	No child processes
EAGAIN	11	Try again
ENOMEM	12	Out of memory
EACCES	13	Permission denied
EFAULT	14	Bad address
ENOTBLK	15	Block device required
EBUSY 16	16	Device or resource busy
EEXIST	17	File exists
EXDEV	18	Cross–device link
ENODEV	19	No such device
ENOTDIR	20	Not a directory
EISDIR	21	Is a directory
EINVAL	22	Invalid argument
ENFILE	23	File table overflow
EMFILE	24	Too many open files
ENOTTY	25	Not a typewriter
ETXTBSY	26	Text file busy
EFBIG	27	File too large
ENOSPC	28	No space left on device
ESPIPE	29	Illegal seek
EROFS	30	Read–only file system
EMLINK	31	Too many links

EPIPE	32	Broken pipe
EDOM	33	Math argument out of domain of func
ERANGE	34	Math result not representable
EDEADLK	35	Resource deadlock would occur
ENAMETOOLONG	36	File name too long
ENOLCK	37	No record locks available
EINVFNC	38	Function not implemented
ENOTEMPTY	39	Directory not empty
ELOOP	40	Too many symbolic links encountered
ESHELL	41	It's a shell script
ENOSYS	EINVFNC	

6.5 – VT–5 TERMINAL CODES

Ctrl G	07H	Beep.
Ctrl H	08H	Backspace.
Ctrl I	09H	TAB.
Ctrl J	0AH	Advances one line.
Ctrl K	0BH	Move cursor to origin.
Ctrl L	0CH	Clears screen and moves cursor to origin.
Ctrl M	0DH	Carriage return.
Ctrl \	1CH	Advances cursor one position.
Ctrl]	1DH	Moves cursor back one position.
Ctrl ^	1EH	Move cursor up.
Ctrl _	1FH	Move cursor down.
	7FH	Deletes character and moves cursor to the left.
Esc A	1BH,41H	Moves cursor up.
Esc B	1BH,42H	Moves cursor down.
Esc C	1BH,43H	Move cursor to the right.
Esc D	1BH,44H	Move cursor left.
Esc E	1BH,45H	Clears screen and places cursor at origin.
Esc H	1BH,48H	Places cursor at the origin.
Esc J	1BH,4AH	Erases to the end of the screen, does not move cursor.
Esc j	1BH,6AH	Clears screen and places cursor at origin.
Esc K	1BH,4BH	Erases to end of line, do not move cursor.
Esc L	1BH,4CH	Insert line above cursor, move rest of screen

		down, leave cursor at start of new line.
Esc l	1BH,6CH	Erases to end of line, do not move cursor.
Esc M	1BH,4DH	Erases cursor line, moves rest of screen to line and places cursor at beginning of next line.
Esc x 4	1BH,78H,34H	Selects block cursor.
Esc x 5	1BH,78H,35H	Turns cursor off.
Esc Y n m	1BH,59H,m,n	Move cursor to column m−32 and row y−32.
Esc y 4	1BH,79H,34H	Selects underlined cursor.
Esc y 5	1BH,79H,35H	Turns on cursor.

7 – WiOS

7.1 – FILESYSTEM DRIVER

Description: File-System driver
Driver-Name: “File-System”
GDA: _hfsdrv
Header-File: FSFNC.H

All disk-functions have the same numbers and names as their corresponding DOS 2 functions. These functions were not complete at the time when this documentation was printed.

_SELDSK (Function ‘0x0e’)
Description: Set current drive.
Arguments: None.
Return: None.

_CURDRV (Function ‘0x19’)
Description: Get current drive.
Arguments: None.
Return: None.

_RDABS (Function ‘0x2f’)
Description: Absolute sector read.
Arguments: None.
Return: None.

_WRABS (Function ‘0x30’)
Description: Absolute sector write.
Arguments: None.
Return: None.

_FFIRST (Function ‘0x40’)
Description: Find first entry.
Arguments: char * pointer to filename / file info block.
char * pointer to new file info block.
char search attributes.
Return: None.

_FNEXT (Function '0x41')
Description: Find next entry.
Arguments: char * Pointer to file info block
Return: None.

_OPEN (Function '0x43')
Description: Open file handle.
Char * Arguments pointer to filename
int mode
Return 0 ⟶ error
1~255 ⟶ file handle

_CREATE (Function '0x44')
Description: Create file and open handle.
Arguments: None.
Return: None.

_CLOSE (Function '0x45')
Description: Close file handle.
Arguments: int file handle
Return 0 ⟶ done
1~255 ⟶ error

_READ (Function '0x48')
Description: Read from file handle.
Arguments: int File handle.
char * Destination address (4000H...BFFFh).
uint Len (0~16 384).
T_SEG Destination segment (0~3071).
char Dummy (must be 0) mode (CRC, CRYPT).
Return 0 ⟶ done;
1~255 ⟶ error.

_WRITE (Function '0x49')
Description: Write to file handle.
Arguments: None.
Return: None.

_SEEK (Function '0x4a')
Description: Seek (position file pointer).

Arguments: int File handle.
uint Offset (0~65 535).
char Mode: 0 ⟶ from start of file;
1 ⟶ from end of file;
2 ⟶ relative to current position.
Return 0 ⟶ done.
1~255 ⟶ error.

_DELETE (Function ‘0x4d’)
Description: Delete file or subdirectory.
Arguments: None.
Return: None.

_RENAME (Function ‘0x4e’)
Description: Rename file or subdirectory.
Arguments: None.
Return: None.

_MOVE (Function ‘0x4f’)
Description: Move file or subdirectory.
Arguments: None.
Return: None.

_ATTR (Function ‘0x50’)
Description: Change file or subdirectory attributes.
Arguments: None.
Return: None.

_FTIME (Function ‘0x51’)
Description: Get/set file date and time.
Arguments: None.
Return: None.

_VERIFY (Function ‘0x58’)
Description: Get verify flag setting.
Arguments: None.
Return: None.

_GETCD (Function ‘0x59’)
Description: Get current directory.
Arguments: None.

Return: None.

_CHDIR (Function '0x5a')
Description: Change current directory
Arguments: None.
Return: None.

_PARSE (Function '0x5b')
Description: Parse pathname.
Arguments: None.
Return: None.

_FORMAT (Function '0x67')
Description: Format disk.
Arguments: None.
Return: None.

_D2V (Function '0x80')
Description: Read directly from file handle to port-address (e.g. VRAM).
Arguments: int handle.
uint len.
char port.
Return 0 ⟶ done.
1~255 ⟶ error.

7.2 – EXTERNAL DRIVER

Description: External driver
Driver-Name: "External Driver"
GDA: _hextdrv
Header-File: EXTFNC.H

NONAME (Function '1')
Description: Used internally.
Arguments: None.
Return: None.

RESKEY (Function '2')
Description: Clear keyboard-buffer of calling task.
Arguments: None.

Return: None.

GETKEY (Function '3')
Description: Get next keyboard-character in buffer of calling task.
Arguments: None.
Return: 0 ⟶ no key has been pressed since last GETKEY.
1~255 ⟶ character-code of the key.

7.3 – GRAPHIC I/O DRIVER

Description: Graphic I/O driver
Driver-Name: "Graphic-IO Driver"
GDA: _hgiodrv
Header-File: GIOFNC.H

NONAME (Function '1')
Description: Used internally.
Arguments: None.
Return: None.

NONAME (Function '2')
Description: Used internally.
Arguments: None.
Return: None.

NONAME (Function '3')
Description: Used internally.
Arguments: None.
Return: None.

GET_VHANDLE (Function '4')
Description: Get handle of a file in Videoram-Directory (used for finding font- and icon-handles).
Arguments char ICON for icon.
FONT for font.
char * filename.
Return: 0~65534 ⟶ handle of icon/font;
–1 ⟶ file not found.

ADDFONTLIB (Function '5')
Description: Load a font from disk.

Arguments:	char * filename.
Return	0~65 534 ⟶ handle of loaded font; –1 ⟶ file not found.
Note:	Don't overload memory. No checking is done yet.

NONAME (Function '6')

Description:	Used internally.
Arguments:	None.
Return:	None.

7.4 – GRAPHIC DRIVER

Description:	Graphic driver
Driver-Name:	"Graphic Driver"
GDA:	_hgrpdrv
Header-File:	GRPFNC.H

MOUSEBOX (Function '1')

Description:	Set the box where the mouse can move absolute coordinates – corners must be in the order upper-left / lower-right.		
Arguments:	int	x	coordinates of upper left corner.
	int	y	
	int	x	coordinates of lower right corner.
	int	y	
Return:	None.		

PPOINTER (Function '2')

Description:	Put the mouse-pointer at absolute coordinates on the screen absolute coordinates.		
Arguments:	int	x	coordinates of mouse-pointer.
	int	y	
Return:	None.		
Note:	If the coordinates are outside of the mousebox, it will 'jump' into them at the next interrupt).		

MOUSEMOVE (Function '3')

Description:	Move mouse according to the user-movement (not necessary)
Arguments:	None.

Return: None.

STOREBOX (Function '4')
Description: Store current VDA-coordinates.
Arguments: None.
Return: None.
Note: There is only set of VDA-coordinated to be stored! Since this functions is also used by WiOS, the function RESTOBOX might not restore the coordinates you stores after a polling or calling the window-driver.

RESTOBOX (Function '5')
Description: Restore the last set of VDA-coordinates stored with STOREBOX.
Arguments: None.
Return: None.
Note: See note at STOREBOX.

SETBOX (Function '6')
Description: Set valid display area (VDA).
Arguments: absolute coordinates:
int x Coordinates of first corner.
int y
int x Coordinates of opposite corner.
int y
Return: None.
Note: Corners may be set freely – they are adjusted to upper-left / lower-right.

COPY (Function '7')
Description: Copy graphic area corresponding to VDA.
Arguments: int sx Coord. of upper left corner from source-area.
int sy
int nx Width of area to be copied.
int ny
int dx Destination coordinates.
int dy
Return: 0 ⟶ area was fully copied (destination is in VDA).
1 ⟶ nothing was copied (dest. is totally out of VDA).
2 ⟶ area was cutted (dest. is partially out of VDA).

Note Copy-direction is corrected automatically if source and destination intersect.

PUTICON (Function '8')

Description: Puts icon on screen corresponding to VDA.

Arguments: int 0~32767 Handle of standard-int icon from VRAM.

−1 Icon graphic-data comes from memory and '*_adrblk' must contain the following information:

offset	len	
+0	2	Address.
+2	2	Segment of icon-graphic-data (32768 color mode).
+4	2	Width.
+6	2	Height of icon.

int x Coordinates of destination.

int y

Return: 0 ⟶ icon was fully put (destination is in VDA).

1 ⟶ nothing was put (dest. is totally out of VDA).

2 ⟶ icon was cutted (dest. is partially out of VDA).

EXPICON (Function '9')

Description: Expand icon (fill complete VDA with icon)

Arguments: int 0~32767 Handle of standard-icon from VRAM.

−1 Icon graphic-data comes from memory (see PUTICON for data of '*_adrblk').

int x Coordinates.

int y

Return: None.

Note: x,y-coordinate set the destination of the 'master'-icon. The area around this icon is tiled with the same icon.

SETFONT (Function '10')

Description: Set current font.

Arguments: int Font-handle.

Return Height of font.

Note No error-checking for the font-handle is done. If a non-existing font-handle is sent, the height of the current font is returned and nothing is changed.

SETCOL (Function '11')

Description:	Set the font-color	
Arguments	int	Foreground color.
	int	Background color.
Return:	None.	
Note:	Color 0 is transparent	

SETPOS (Function '12')

Description:	Set current text-position	
Arguments:	int x	Coordinates of text-position.
	int y	
	int	Distance in pixels between characters.
Return:	None.	

WRTTXT (Function '13')

Description:	Write text to the screen according to VDA.	
Arguments:	char *	Address of string.
	T_SEG	Segment of string.
	int	0 ⟶ all characters until byte '0' are written.
		1~32 767 ⟶ characters are written.
	Char	Flags not used yet. Must be 0.
Return:	Total length of text in pixels (NOT the length of displayed text)	
Note:	Distance between characters is set with function 'SETPOS' font color 0 is transparent every underscore will underline the next character	

GETTXTLEN (Function '14')

Description:	Get len of null-terminated string in pixels.	
Arguments:	Char *	Address of string
	T_SEG	Segment of string
Return:	None.	
Note:	Distance between characters is set with function 'SETPOS'. Every underscore will underline the next character.	

GETCHARS (Function '15')

Description:	Get number of characters fitting in given width.	
Arguments:	char *	address of string.
	T_SEG	segment of string.

	int	width of area in pixels.
	char	0 ⟶ direction forward (increment text-pointer).
		1~255 ⟶ backward.
Return:	Number of characters.	

BOX (Function '16')

Description:	Draw colored box on screen according to VDA.	
Arguments:	int x	Coordinates of upper left corner.
	int y	
	int nx	Width of box.
	int ny	
	int 0~32 767 ⟶ color of box.	
	int logical operation (see V9990 technical manual).	
Return:	None.	

GETBOX (Function '17')

Description:	Get current VDA-coordinates.		
Arguments:	None.		
Return:	address of data-block.		
	offset	len	
	+0	2	x-coordinate of upper left corner.
	+2	2	y
	+4	2	x-coordinate of lower right corner.
	+6	2	y
Note:	Use this function to store current VDA-coordinates permanently in your own memory-area (better than STOREBOX when polling or calling the window driver).		

7.5 – MEMORY DRIVER

Description:	Memory driver
Driver-Name:	–
GDA:	_hmemdrv
Header-File:	MEMFNC.H

This driver is in the internal WiOS-part and has no name.

ALLSEG (Function '1')

Description:	Allocate one new segment.
Arguments:	T_TASK 0..252 task□s handle

Return: −1 ⟶ no more free segments
0~32767 ⟶ number of the segment

ENASEG (Function '2')
Description: Enable segment in page 1 / 2.
Arguments T_SEG 0~32767 ⟶ number of the segment.
char 1 ⟶ page 1: 4000H-7FFFH.
2 ⟶ page 2: 8000H-bFFFH.
Return: 0 segment was enabled
−1 ⟶ segment could not be enabled (out of range).
−2 ⟶ segment was not allocated.

FRESEG (Function '3')
Description: Free allocated segment.
Arguments T_SEG 0~32767 ⟶ number of the segment.
Return: 0 ⟶ segment was freed.
−1 ⟶ segment could not be freed (out of range).
−2 ⟶ segment was not allocated by this task.
Note After freeing a segment should be paid attention NOT to write in the area where the segment was if no other segment has been enabled at this page.

GETSEG (Function '4')
Description: Get status of segment.
Arguments T_SEG 0..32767 ⟶ number of the segment.
Return: 0~252 ⟶ task/driver's handle which allocated the segment
253 ⟶ used by a non specified application (was allocated when WiOS started).
254 ⟶ too slow memory for CPU (e.g. Sony mappers in turbo mode)
255 ⟶ segment is free.

NONAME (Function '5')
Description: Used internally.
Arguments: None.
Return: None.

SEGCPY (Function '6')

Description:	Copy a given number of bytes from one segment to another.
Arguments:	Relative addresses in segments: range is from 0~3FFFH:
	T_SEG 0~32 767 ⟶ source segment.
	T_SEG 0~32 767 ⟶ destination segment.
	uint 0~16 383 ⟶ source address.
	uint 0~16 383 ⟶ destination address.
	uint 0~16 384 ⟶ number of bytes to be copied.
Return	0 ⟶ data was copied
	−1 ⟶ out of range (src > 16 383, dest > 16 383 or length of the block exceeds the src or dest segment)
Note:	No checking is done whether the segments are allocated or to which task they belong

SEGNCPY (Function '7')

Description:	Copy bytes from one segment to another until a given byte is found.
Arguments:	Relative addresses in segments: range is from 0~3FFFH.
	T_SEG 0..32767 source segment
	T_SEG 0..32767 destination segment
	uint 0..16383 source address
	uint 0..16383 destination address
	uint 0..16384 number of bytes to be copied
	char 0..255 termination byte
Return:	−1 ⟶ see SEGCPY.
	1~16 384 ⟶ number of bytes copied
Note:	No checking is done whether the byte exists. SEGNCPY will copy data until the termination byte is found – with all consequences. Make sure that it is in a valid range.

7.6 – STANDARD DRIVER

Description:	Standard driver
Driver-Name:	–
GDA:	_hstddrv
Header-File:	STDFNC.H

This driver is in the WiOS-kernel and has no name.

ADDINT (Function '1')
Description: Add an interrupt.
Arguments: T_TASK 0~252 (task/driver's handle).
T_SEG 0~32 767 (segment).
uint 4000H~7FFFH (address of interrupt-routine).
Return: 0~MAXINT ⟶ interrupt handle ('MAXINT' is in DEF.H).
255 ⟶ no more interrupts.
Note: Interrupt-routine may not EI.

DELINT (Function '2')
Description: Remove an interrupt.
Arguments: T_INT interrupt handle.
Return 0 ⟶ Interrupt was removed.
255 ⟶ Interrupt could not be removed.

DRAG (Function '3')
Description: Starts a drag-operation.
Arguments: None.
Return: None.
Note Under construction. Do not use.

7.7 – TASK DRIVER

Description: Task driver
Driver-Name: –
GDA: _htaskdrv
Header-File: TASKFNC.H

This driver is in the internal WiOS-part and has no name.

NONAME (Function '1')
Description: Used internally
Arguments: None.
Return: None.

LOADTASK (Function '2')
Description: Load a task from disk
Arguments char * Pointer to drive+path+filename string
Return: –1 ⟶ task could not be loaded
0~252 ⟶ handle of task

NONAME (Function '3')

Description: Used internally
Arguments: None.
Return: None.

KILLTASK (Function '4')

Description: Remove a task from memory and free all segments and interrupts.
Arguments: T_TASK Task handle.
Return: 0 ⟶ event was sent.
−1 ⟶ task does not exist.
Note: Not yet supported.

S_NAME (Function '5')

Description: Search by name if a task/driver is currently present (loaded)
Arguments: char * Pointer to name-string of task to be searched.
Return: 0 ⟶ not found.
≠0 ⟶ address of data block.

offset	len	
+0	2	Handle of found task/driver.
+2	2	Version number.

Note: String must be between 4000H and 7FFFH.

ADD_EVENT (Function '127')

Description: Send an event to a task.
Arguments: uint Address of event-block

Member	Function
task	Handle of the destination task.
event	Event.
array[ ]	12 info-bytes for the receiving task.

char handle of the sending task (filled by WiOS).
Return: None.
Note Event-block must be outside of page 1 & 2 so the receiving task can access the block without segment switching. It is recommended to use '_eventblk' from GDA to store events.

7.8 – WINDOW DRIVER

Description: Window driver
Driver-Name: “Window Driver”
GDA: _hwindrv
Header-File: WINFNC.H

The function argument-explanation require the definition of struct WINSTR windat; as a global variable in the C source-code to have access to the window-structure via names.

If there are problems with some window-specific words in the following list, refer to the illustration of chapter about Icons.

Since this is the most complex driver, there are some functions which are not yet implemented. Still there is enough functionality to write normal applications and most of the missing functions can be simulated with some ‘work-arounds’.

7.8.1 – Window Structure

Type	Name	Description
int	handle	Window-handle. Identifies each window – no windows have the same handle.
int	x	Absolute x position on screen.
int	y	Absolute y position on screen.
int	nx	Width of visible work-area.
int	ny	Height of visible work-area. The visible work-area is always guaranteed to be free for the task’s use. Window-icons do NOT use this area since WiOS puts them ‘around’ the window – except for the following cases, where the icons are copied into the user-area: • Back-, Close- or Toggle-Size-Icon, if there’s no title-bar. • Resize-Icon, if there is no scrollbar.
Int	vx	Virtual width of window.
int	vy	Virtual height of window. Is only needed for calculation of the scrollbars and may be 0 if you have none.

int	scrx	Horizontal scroll-offset.
int	scry	Vertical scroll-offset. always contains the absolute offset in pixels to the upper left corner of the window (like SET SCROLL in BASIC) and is not needed without scrollbars.
int	minx	Minimum width of window.
int	miny	Minimum height ofwindow. Contains the minimum size of the window, which can be scaled by the user – does only prevent the user to create smaller windows using the mouse, NOT changes made manually from a task.
Int	maxx	Maximum width of window.
int	maxy	Maximum height of window. (See minimum-size).
Int	behind	Handle of the window in front (see below). Can be used to check the level of the window.
T_TASK	task	Task which created this window. Used internally to determine which task has to be informed if the user drags a window and to close all related windows if a task is closed.
char	winflag	Bitmapped window-area flags. bit 7: unused (must be 0). bit 6: not yet implemented (must be 0). bit 5: not yet implemented (must be 0). bit 4: not yet implemented (must be 0). bit 3: auto-repeat on arrow-icons. bit 2: window is pane-window (see 'parent' below). bit 1: window can be dragged (via title bar). bit 0: 0 → redraw from task needed; 1 → WiOS can redraw the whole window (not yet implemented).
char	iconflag	Bitmapped window-icon flags. Specifies the WiOS-controlled icons of a window. bit 7: unused (must be 0). bit 6: horizontal scrollbar. bit 5: adjust-size icon. bit 4: vertical scrollbar. bit 3: toggle-size icon.

		bit 2: title-bar. bit 1: close icon. bit 0: back icon.
char	workflag	Bitmapped window-work-area flags (see below). bit 7: unused (must be 0). bit 6: unused (must be 0) bit 5: double-click notifies task. bit 4: release over work-area notifies task (for drag & drop). bit 3: click notifies task (once). bit 2: click notifies task (always). bit 1: notify task continually while pointer is over work area. bit 0: ignore all clicks.
int	parent	Handle of parent window. –1: window is a parent window (i.e. can have pane windows). 0~255: handle of the parent window, if window is a pane window Pane windows are 'connected' to their parent windows, i.e. if a parent window is opened on front or on back, all pane-windows will stay directly in front of the parent window. If a parent window is closed, all connected pane windows will also be closed. Note that pane windows are NOT moved in the x- and y--position, if the parent window is.
char	statflag	Bitmapped window-status flags. bit7~2: unused (must be 0). bit 1: window is minimized (not yet implemented: 0). bit 0: window is maximized (not yet implemented: 0).
int	realnx	'Real' width of window on screen.
int	realny	'Real' height of window on screen.
char	dummy[25]	These bytes are to fill the window data block up to 64 bytes and are reserved for future use, so always fill them with 0.

7.8.2 – Window Driver Functions

CREATE_WIN (Function '1')
Description: Add window to list and send open-event.

Arguments: uint Address of window-data-block.
Needed members:
x,y
nx,ny
vx,vy
scrx,scry
minx,miny
maxx,maxy
behind −2 ⟶ back.
−1 ⟶ front.
0~255 ⟶ specified window.
If window is a pane window, 'front' always means on top of the pane window block, and 'back' means directly above the parent window (on back of pane window block).
winflag
iconflag
workflag
parent −1 ⟶ window is parent.
0~255 ⟶ handle of parent window.
statflag Must be 0.
dummy[25] All 25 bytes must be 0.

Return: −1 ⟶ window could not be created.
0~255 ⟶ handle of the new window.

Note The members handle, task, realnx and realny are filled by WiOS, no matter if they have valid data or not. This function always creates a new window after 'CREATE_WIN' the window is NOT drawn on the screen. It is only registered and an 'OPEN' event is sent to the task after the next polling.

OPEN_WIN (Function '2')

Description: Change data of existing window in list, calculate new position and send redraw-events to all affected tasks.

Arguments: uint Pointer to window-data (see above).

Return: −1 ⟶ window could not be opened.
1~255 ⟶ number of windows opened (including pane windows).

Note: Window-data must be valid and complete so call GET_WIN_STATE before. If the behind value changes, all pane-windows (if window is a parent) are also changed so they are always directly above the parent window be sure use this function only on windows belonging to your task.

CLOSE_WIN (Function '3')

Description: Remove window and all connected pane-windows from list and send redraw-events lower window's tasks

Arguments: uint 0~255 ⟶ handle of window to be closed.

Return: −1 ⟶ window could not be deleted.
1~255 ⟶ number of windows closed (including pane windows).

Note: Be sure use this function only on windows belonging to your task.

GET_WIN_STATE (Function '4')

Description: Copy window-data from list to task's memory.

Arguments: uint Address where the data has to be copied. member 'handle' must contain the window-handle.

Return: 0~FFFEH ⟶ address of window-data (=the address sent).
−1 ⟶ window could not be found.

DRAWFRAME (Function '5')

Description: Draw window frame.

Arguments: int Destination x-coordinate.
int Destination y-coordinate.
uint Pointer to address of window structure member 'handle' must contain the window-handle.
char Must be '0'.

Return: None.

Note: No validity checking for the window-handle is done is called automatically by 'GETWIB'.

GETWIB (Function '6')

Description: Get window-information-block for redraw.

Arguments: uint 0~255 ⟶ handle of window.

Return: 0~FFFEh ⟶ address of WIB.
−1 ⟶ window could not be found.

7.8.3 – Redrawing Windows

COPYWIN (Function '7')
Description: Copy graphic area to display-screen using the redraw-stack.
Arguments: uint Address of WIB.
Return: None.
Note: VDA is changed. WIB must be outside page 1 and 2 ('GETWIB' always returns a correct address).

NONAME (Function '8')
Description: Used internally.
Arguments: None.
Return: None.

DRW_MENU (Function '9')
Description: Draws a menu corresponding to the task's data.
Arguments: None.
Return: None.

CHK_MENU (Function '10')
Description: Check coordinates and return the item number.
Arguments: None.
Return –1 ⟶ no item at these coordinates.
0~32767 ⟶ number of item.

NONAME (Function '11')
Description: Used internally.
Arguments: None.
Return: None.

NONAME (Function '12')
Description: Used internally.
Arguments: None.
Return: None.

NONAME (Function '13')
Description: Used internally.
Arguments: None.
Return: None.

The redraw-routine has to know a bit more than just the data in the window structure. This data can be ordered by the window driver and is called the window-information-block (WIB). It has the following structure:

T_SEG	stackseg	segment of the redraw-stack
uint	elements	number of rectangular areas in redraw-stack
struct WINSTR	*windat	pointer to window-data (outside page 1 or 2). Needed window-members are: x,y nx,ny realx,realy
uint	sx,sy	start coordinates of window-copy (always 0,212)
uint	offx,offyoffset for work-area	
int	moffx,moffy	move-offset relative to last position
uint	toffx,toffy	title offset
uint	t_nx,t_ny	title width

The redraw stack has the following structure:

uint	x1,y1	Upper left coordinate (absolute)
uint	x2,y2	Lower right coordinate (absolute)

The task's redraw-routine has to look like this:

- Call 'GETWIB' and store return-address
- If elements is equal 0, nothing is to do
- Draw your own data at coordinates (sx+offx , sy+offy) that come with the 'GETWIB' function
- Call 'COPYWIN'

7.9 – EVENT DEFINITION

Event names are in 'EVENTFNC.H'.

E_NULL (Function '0')
Description: Null-Event.
Arguments: None.

E_REDRAW (Function '1')
Description: Window redraw request.
Arguments: array[0] ⟶ handle of window to be redrawn.

E_SCROLL (Function '2')

Description: Window scroll request.
Arguments: task sender (this task).
array[0] ⟶ handle of window to be scrolled.
array[1] ⟶ horizontal scroll offset (absolute).
array[2] ⟶ vertical scroll offset (absolute).
Note: An 'E_WOPEN' event is sent immediately after the scroll event to the task. The task only has to determine whether the new scroll offsets are accepted or not and, if yes, update the window data. Tasks which send a scroll request to windows of other tasks should take care to send the 'E_WOPEN' event immediately after it.

E_WOPEN (Function '3')
Description: Window open request.
Arguments: task Sender (this task).
array[0] Handle of window to be opened.
array[1] Behind value.
array[2] Upper left x-coordinates of window position.
array[3] Upper left y-coordinates of window position.
array[4] Width of work-area.
array[5] Height of work-area.
Note Width and height are the work-area dimensions, not the real window size.

E_WCLOSE (Function '4')
Description: Window close request.
Arguments: Task sender (this task).
array[0] Handle of window to be closed.

E_PNTOUT (Function '5')
Description: Pointer has left window.
Arguments: array[0] Handle of left window.

E_PNTIN (Function '6')
Description: Pointer has entered window.
Arguments: array[0] Handle of entered window.

E_MCLICK (Function '7')
Description: Mouseclick.
Arguments: array[0] Handle of window which has been clicked.
array[1] Coordinates of pointer where
array[2] click was (absolute).
array[3] Bitmapped click-type (bit is set if clicked).
Bit Function
0 ⟶ left button is pressed.
1 ⟶ left button is held.
2 ⟶ right button is pressed.
3 ⟶ right button is held.
array[4] The time (16-bit-counter) when the click was performed.
Note: Coordinates do not give the actual coordinates of the pointer but the coordinates of where the pointer was on click-time.

E_EDRAG (Function '8')
Description: End of a drag operation.
Arguments: Task sender (this task).
array[0] Data-type.
array[1] Detailed information on data.
Note: For the data-type see chapter "Data Type Definition" section.

E_WKAREA* (Function '9')
Description: Pointer is over work-area (only if bit 1 of work-area flag is set).
Arguments: array[0] Handle of window below pointer.

E_MENU* (Function '10')
Description: Not yet supported.
Arguments: None.

E_SHUTDOWN (Function '11')
Description: Requests the task to be closed.

Arguments: None.
Note: Does not need to be handled.

E_USERMSG (Function '16')
Description: User definable.
Arguments: task Sender (this task).
array[0] User-message code.
array[1~5] User definable.
Note. Must be handled even if the task shall not receive external data.

7.9.1 – Data sent with the 'E_EDRAG' event

Data type names in brackets are defined in 'DTDEF.H' and must be sent in array[0], the specifier in array[1].

Data type: 0 ('TEXT').
Specifier: 0 ('STRING').
Description: One or more text strings with no special function.

Data type: 0 ('TEXT').
Specifier: 1 ('FILENAME').
Description: One or more text strings with filenames to be handled / edited / opened.

Data type: 1 ('GRAPHIC').
Specifier: 0~255 bits per pixel.
Description: One rectangular bitmapped image (resolution is defined in the data itself).

7.9.2 – Data sent with the 'E_USERMSG' event

Whenever sending user messages, array[0] holds the user-message code. That is, in this case, code 'DATA', which is defined in 'DTDEF.H'.

Data: **STRING**
Arguments: array[1] Address of data block.
array[2] Segment of data block.
Data Block

Offset	Len	Valid Entries	Description
+0	2	0~65 535	Number of strings.

+2	2	1~3071	Number of segments used for data.
+4,6,...	2	0~3071	Segment numbers with strings.

After segment numbers:

2	0	Null-terminated strings.
	1~255	Number of fields.

- If strings are null-terminated, the strings come directly after the '0'.
- If strings are in fields of fixed length, there follows the length of the field.
- 2 bytes for each field length. After the length data comes the field-formatted string data.

Note: If the strings exceed 16K, it's up to the program to switch the segments.

Data: **FILENAME**

Arguments: array[1] Address of data block.
array[2] Segment of data block.

Data Block

Offset	Len	Valid Entries	Description
+0	2	0~65 535	Number of filenames.

After number of filenames:

Null-terminated strings.

Note: The filename data structure is exactly like the one of null-terminated strings – with the difference that the destination task does not have to insert this data somewhere but to open these filenames. In practice, that means to load these files – either in new windows for each file or for a slide show, one after each other, depending on the program. Also it's up to the destination task to parse the filename extensions or the files whether they are valid files for this task.

Data: **GRAPHIC**

Arguments: array[1] Address of data block.
array[2] Segment of data block.

Data Block

Offset	Len	Valid Entries	Description
+0	2	Unused.	
+2	2	1~3071	Number of segments used for image.

+4,6,...	2	0~3071	Segment numbers with image-data.
After segment numbers:			
	2	0~65 535	Width of image.
	2	0~65 535	Height of image.
			15-bit-mapped image data (see VDP-spec).

Note: If an image exceeds 16K, it's up to the program to switch the segments

7.10 – GDA REFERENCE

This section describes the list of needed GDA variables.

The GDA area starts at C030h. it's end is open since it can be expanded in future. Do not change reserved variables. They are not protected, but changing them may result in system instability or a crash. All GDA variables are pointers to the address of the variable or field. Type definitions are in 'DEF.H'.

Type	Name	Offset	Len	Description
T_SEG	*_tot_seg	+0	2	Total number of segments in computer.
T_SEG	*_free_seg	+2	2	Current number of free segments.
T_SEG	*_tmp_seg	+4	2	Segment of temporary segment.
<unused>		+6	2	
<internal use>		+8	2	
T_TASK	*_p1_task	+10	2	Current handle of task in page 1.
T_SEG	*_p1_seg	+12	2	Current segment in page 1.
char	*_p1rseg	+14	2	Mapper segment number of current segment in page 1.
char	*_p1_slt	+16	2	Slot of current segment in page 1
<unused>		+18	2	
T_SEG	*_p2_seg	+20	2	Current segment in page 2.
char	*_p2rseg	+22	2	Mapper segment number of current segment in page 2.
char	*_p2_slt	+24	2	Slot of current segment in page 2.
<internal use>		+26	2	
uint	(*_caldrv)(.)	+28	2	WiOS-function call entry address.
T_TASK	*_hmemdrv	+30	2	Handle of memory driver.
T_TASK	*_hstddrv	+32	2	Handle of standard driver.

T_TASK	*_htaskdrv	+34	2	Handle of task driver.
T_TASK	*_hfsdrv	+36	2	Handle of file system driver.
T_TASK	*_hgiodrv	+38	2	Handle of graphic I/O driver.
T_TASK	*_hgrpdrv	+40	2	Handle of graphic driver.
T_TASK	*_hwindrv	+42	2	Handle of window driver.
T_TASK	*_hextdrv	+44	2	Handle of external driver.
uint	*_wiosver	+46	2	Current version of internal WiOS part.
VOID	(*_wiosend)()	+48	2	WiOS shutdown call (only in Alpha-version).
uint	(*_poll)()	+50	2	Poll entry address.
<internal use>		+52 to +106		
VOID	(*_dump)()	+108	2	Entry call to make a memory dump on V9958 (argument is address).
struct EBSTR	*_eventblk	+110	2	Address of event-block holding the data after polling
struct WIBSTR	*_wibblk	+112	2	Address of window information Block struct
WINSTR	*_winblk	+114	2	Address of window data block
uint	*_act_font	+116	2	Handle of current font
uint	*_std_font	+118	2	Handle of standard (□system□) font
<internal use>		+120	2	
char	*_busyflag	+122	2	Status of direct-write VDP status
uint	*_adrblk	+124	2	Address of global, segment-switch independent address block
char	*_mb	+126	2	Current state of mouse buttons (bitmapped). bit description 0 ⟶ left trigger state (1=pressed). 1 ⟶ right trigger state (1=pressed).
char	*_cltype	+128	2	Type of last mouse click (bitmapped). bit description 0 ⟶ left single click (1=clicked). 1 ⟶ left double click (1=clicked). 2 ⟶ right single click (1=clicked). 3 ⟶ right double click (1=clicked).
uint	*_cltime	+130	2	Time of mouse click (array of 2). offset description +0 time of last left-click. +2 time of last right-click.

int	*_coord	+132	2	Current mouse coordinates (array of 2).
uint	(*_cal_seg)()	+134	2	Entry call for tasks to call multiple-part functions.
<internal use>		+136	2	
uint	(*_dcal_seg)()	+138	2	Entry call for drivers to call multiple-part functions
<internal use>		+140	2	
<internal use>		+142	2	

... to be expanded in future

7.11 – MENU BLOCK STRUCTURE

The menu block is one big field of data where one or more menu types can be defined. The following symbols are used in the structure definition:

prefix
- b Byte (1 byte).
- w Word (2 bytes).
- t Text (variable byte length).
- # Definition of valid values for 'D' (see suffix).

suffix
- N Any number.
- D Number from a pre-defined list.
- S And character string.
- 0 Must be this value.

Valid border styles:
- 0 No drawn border.
- 1 Filled box with background color.

Definition

bN number of item blocks

type
bD
#0 Option

Once for each option block:

wN Font-handle.
wN X-offset.

wN Y-offset.
w0 Width of option-block (filled by WiOS).
w0 Height of option block (filled by WiOS).
bD Border style.
bN Distance of border in pixels.
bD Direction.
#0 Top to bottom.
#1 Left to right.

For each entry:

bD Entry Flags (bitmapped):

		(0)	(1)
entry status	bit 0	end of list	valid entry
valid	bit 1	disabled	enabled (selectable)
selection	bit 2	not selected	selected
	bit 3~7	unused (must be reset)	

tS Null-Terminated String.

Continued at Entry Flags (until bit 0 is reset).

#1 Radio

Once for each radio block:

wN Font-handle.
wN X-offset.
wN Y-offset.
w0 Width of option-block (filled by WiOS).
w0 Height of option block (filled by WiOS).
bD Border style.
bN Distance of border in pixels.
bD Direction.
#0 Top to bottom.
#1 Left to right.

For each entry:

bD Entry Flags (bitmapped)

		(0)	(1)
entry status	bit 0	end of list	valid entry
valid	bit 1	disabled	enabled (selectable)
selection	bit 2	not selected	selected
	bit 3~7	unused (must be reset)	

tS Null-Terminated String

continued at Entry Flags (until bit 0 is reset)

#2 List

Once for each list block

wN	Font-handle.	
wN	X-offset.	
wN	Y-offset.	
w0	Width of option-block (filled by WiOS).	
w0	Height of option block (filled by WiOS).	
bD	Border style.	
bN	Distance of border in pixels.	

For each entry:

bD	Entry Flags (bitmapped)			
		(0)	(1)	
entry status	bit 0	end of list	valid entry	
valid	bit 1	disabled	enabled (selectable)	
sub list	bit 2	no sub list	sub list (arrow on the right)	
	bit 3~7	unused (must be reset)		
tS	Null-Terminated String			

continued at Entry Flags (until bit 0 is reset)

#3 Scrollbar

bD	Entry Flags (bitmapped)		
	(0)	(1)	
Entry status	bit 0	not valid	valid
wN	X-offset.		
wN	Y-offset.		
wN	Visible size of field in pixels.		
wN	Virtual size of field in pixels.		
wN	Real size of field in pixels.		
wN	Scroll offset of scrollbar in pixels.		
bD	Direction.		
#0	Top to bottom.		
#1	Left to right.		

#4 Box

bD	Entry Flags (bitmapped)		
	(0)	(1)	
Entry status	bit 0	not valid	valid
wN	X-offset.		
wN	Y-offset.		
wN	Width.		

wN Height.
bD Border style.
bN Distance of border in pixels.

8 – SYSTEM VARIABLES

8.1 – SYSTEM AREA FOR MSXDOS1

F1C1H, 1 – Countdown timer for the drives. By setting this counter to 0, the drives' motors are stopped.

F1C2H, 1 – Sub-counter of the countdown timer for the drive.

F1C3H, 1 – Countdown counter sub-counter for the drive.

F1C4H, 1 – Number of the currently active drive.

F1C5H, 1 – Track number where the drive head A: is.

F1C6H, 1 – Track number where the head of drive B: is.

F1C7H, 1 – Logic drive active.

F1C8H, 1 – Number of physical drives present.

F1C9H, 24 – Routine for printing on the screen a string ending with "$". DE – Starting address of the string.

F1CAH~F1E1H – ?

F1E2H, 6 – Routine to abort the program in case of error.

F1E8H, 12 – Calls the address pointed by (HL) in RAM and returns with the DOS Kernel (BDOS) page active.

F1F4H, 3 – Jump to the filename check routine.
HL – Address of the first character of the filename.

F1F7H, 4 – Device name "PRN".

F1FBH, 4 – Device name "LST".

F1FFH, 4 – Device name "NUL".

F203H, 4 – Device name "AUX".

F207H, 4 – Device name "CON".

F20BH, 11 – Reserved for new device or filenames.

F216H, 1 – Current device number:
−5 ⟶ PRN; −4 ⟶ LST; −3 ⟶ NUL; −2 ⟶ AUX; −1 ⟶ CON.

F217H~F220H – ?

F221H, 2 – FCB date of the current file.

F223H, 2 – FCB time of the current file.

F22BH, 12 – Table containing the number of days in the months.

F22BH [31] January	F231H [31] July
F22CH [28] February	F232H [31] August
F22DH [31] March	F233H [30] September
F22EH [30] April	F234H [31] October
F22FH [31] May	F235H [30] November
F230H [30] June	F236H [31] December

F237H, 4 – Used internally by function 10 of BDOS.

F23BH, 1 – Flag to indicate whether the characters should go to the printer. (0 = no; other value, yes)

F23CH, 2 – Current DTA address.

F23EH, 1 – ?

F23FH, 4 – Current sector number of the disk.

F243H, 2 – Pointer to the DPB address of the current drive.

F245H, 1 – Current relative sector of the directory starting from the first (0).

F246H, 1 – Drive that contains the current sector of the directory (0 = A :, 1 = B :, etc.)

F247H, 1 – Default drive (0 = A :, 1 = B :, etc)

F248H, 1 – Day

F249H, 1 – Month

F24AH, 1 – Year-1980 (add 1980 to obtain the correct year)

F24BH, 1 – ?

F24CH, 2 – Hour and minutes

F24EH, 1 – Day of the week (0 = Sunday, 1 = Monday, etc.)

8.1.1 – Hooks called by disk routines

F24FH, 3 – Routine that displays the message "Insert disk for drive".
A – Drive number (41H = A :, 42H = B :, etc)

F252H, 3 – Get the FAT content.

F255H, 3 – Filename repair routine.

F258H, 3 – Directory search routine.

F25BH, 3 – Increment the directory entry (last entry in A).

F25EH, 3 – Routine that calculates the next sector of the directory.

F261H, 3 – Filename repair routine.

F264H, 3 – 'OPEN' function routine.

F267H, 3 – Returns the last FAT.

F26AH, 3 – 'GETDPB' routine of the disk interface (SFIRST).

F26DH, 3 – Routine function 'CLOSE' (writes FAT).

F270H, 3 – Routine function 'RDABS – 2FH' (HL = DMA, DE = Sector,
B = number of sectors). H.DISKREAD.

F273H, 3 – Error handling routine when accessing the disk.

F276H, 3 – 'WRABS' function routine (writes sector).

F279H, 3 – Routine function 'WRABS' (HL = DMA, DE = Sector,
B = number of sectors).

F27CH, 3 – **M**ultiplication routine (HL = DE * BC).

F27FH, 3 – Division routine (BC = BC/DE; HL = Rest).

F282H, 3 – Returns the absolute cluster.

F285H, 3 – Returns the next absolute cluster.

F288H, 3 – Disk sector reading.

F28BH, 3 – Sector writing on the disk.

F28EH, 3 – Starts the operation of reading blocks (records) of the disk.

F291H, 3 – Finalizes the option of reading blocks (records) of the disk.

F294H, 3 – End of the operation of reading blocks (records) of the disk.

F297H, 3 – Error in the operation with blocks (records).

F29AH, 3 – Starts the operation of writing blocks (records) in the disk.

F29DH, 3 – Finalizes the option of writing blocks (records) in the disk.

F2A0H, 3 – Calculates sequential sectors.

F2A3H, 3 – Gets the number of sectors in a cluster.

F2A6H, 3 – Allocate a sequence of FATs.

F2A9H, 3 – Releases a sequence of FATs.

F2ACH, 3 – 'BUFIN' function (adds data to the buffer)

F2AFH, 3 – 'CONOUT' function (BDOS 02H).

F2B2H, 3 – Get the time and date of the file.

F2B5H, 3 – February identification routine (28/29 days).

8.1.2 – Other DOS data

F2B8H, 1 – Number of the current directory entry.

F2B9H, 11 – Filename / extension of the current file.

F2C4H, 1 – Byte of file attributes of the last directory entry read.
If bit 7 is set, files with a NOT attribute of 0 can be opened.
This can be done by setting bit 7 of the FCB-drive byte, by calling the BDOS OPEN routine. (FCB+0).

F2C5H~F2CEH – ?

F2CFH, 2 – Time of the current file.

F2D1H, 2 – Date of the current file.

F2D3H, 2 – Initial cluster of the current file.

F2D5H, 4 – Size of the current file.

F2D9H~F2DBH – ?

F2DCH, 1 – Files with attributes other than F2DCH are also accepted. (F2C4H bit-7 overrides that!)

F2DDH~F2E0H – ?

F2E1H, 1 – Current drive for writing and reading absolute sectors.

F2E2H~F2FDH – ?

F2FEH, 2 – Sub-counter from the countdown timer to the drive.

F301H, 1 – ?

F302H, 2 – Pointer to the MSXDOS abort handler routine.

F304H, 2 – Stores the value of the SP (Stack Pointer) register.

F306H, 1 – Default drive for MSXDOS (0 = A :, 1 = B :, etc).

F307H, 2 – Stores the value of the DE register (FCB address).

F309H, 2 – Used by the DPB for searching (First / Next).

F30BH, 2 – Current sector of the directory.

F30DH, 1 – Check flag (0 = Off; other value, on).

F30EH, 1 – Date format (0- yymmdd; 1- mmdday; 2- ddmyy).

F30FH, 4 – Area used by Kanji mode.

F313H, 1 – Contains the version of the ROM of the MSXDOS.
00H = version 1.x; 20H = version 2.0; 21H = 21H = version 2.1; etc.
Obs: The Nextor returns 99H.

F314H~F322H – ?

F323H, 2 – Address of the disk error handler.

F325H, 2 – Address of the handler of the CTRL+C keys.

8.1.3 – Hooks for the 'COM:' port

F327H, 5 – Routine 'AUXINP' (A = byte read from the AUX device).

F32CH, 5 – 'AUXOUT' routine (A = byte to be sent to the AUX device).

F331H, 5 – Routine for manipulating BDOS functions.

8.1.4 – Keyboard

F336H, 1 – Key pressed flag. Contains FFH if any key is pressed and 03H for CTRL+STOP.

F337H, 5 – Contains the ASCII code of the key pressed and 03H for CTRL+STOP pressed together.

8.1.5 – MSXDOS Variables

F338H, 1 – Flag to indicate the presence of an internal clock (0 = no; another value, yes).

F339H, 7 – Routine used by the internal clock.

F340H, 1 – REBOOT
If it is 0, DOS will reset all variables again.

F341H, 1 – RAMAD0
Slot of page 0 of RAM (format equal to RDSLT – 000CH /BIOS).

F342H, 1 – RAMAD1
Slot of page 1 of RAM (format equal to RDSLT – 000CH /BIOS).

F343H, 1 – RAMAD2
Slot of page 2 of the RAM (format equal to RDSLT – 000CH /BIOS).

F344H, 1 – RAMAD3
Slot on page 3 of the RAM (format equal to RDSLT – 000CH /BIOS).

F345H, 1 – Number of free buffers (025H).

F346H, 1 – Flag to indicate whether the system was booted from MSXDOS on a floppy disk. (0 = no; other value, yes)

F347H, 1 – NMBDRV
Total number of logical drives in the system.

F348H, 1 – MASTER
DOS Kernel slot ID (format equal to RDSLT – 000CH /BIOS).

F349H, 2 – HIMSAV
Pointer to a copy of the FAT of the last connected logical drive (1.5 Kbytes) followed by a copy of the FAT of the next to last connected logical drive (1.5 Kbytes) and so on, up to drive A :. It also indicates the highest memory area available for DOS.

F34BH, 2 – Final address of the MSXDOS Kernel (start for COMMAND.COM). The MSXDOS Kernel start address is stored at 0006H/0007H.

F34DH, 2 – SECBUF
Pointer to a copy of the FAT of the default drive (1.5K).

F34FH, 2 – BUFFER
Pointer to a 512-byte buffer used as Disk BASIC's DTA.

F351H, 2 – DIRBUF
Pointer to a 512-byte buffer used for transferring sectors of the disk (used by DSKI$ and DSKO$ of BASIC).

8.1.6 – DPB addresses

F353H, 2 – DPBBASE

Pointer to the DPB of the current file.

F355H, 16 – DPBLIST
F355H, 2 – DPB address of drive A :.
F357H, 2 – DPB address of drive B :.
F359H, 2 – DPB address of drive C :.
F35BH, 2 – DPB address of drive D :.
F35DH, 2 – DPB address of drive E :.
F35FH, 2 – DPB address of drive F :.
F361H, 2 – DPB address of drive G :.
F363H, 2 – DPB address of drive H :.

8.1.7 – Routines used by MSXDOS

F365H, 3 – Jump from the primary slot reading routine.
(A – Primary slot state)

F368H, 3 – SETROM
Jump to the routine that switch DISK-ROM (BDOS) to page 1 (not available from Disk BASIC)

F36BH, 3 – SETRAM
Jump to the routine that switch RAM to page 1 (not available from Disk BASIC).

8.1.8 – Inter-slot movement routines

F36EH, 3 – SLTMOV
Jump to LDIR from RAM on page 1 (not available from Disk BASIC).

F371H, 3 – AUXINP
Jump to the auxiliary device entry routine.
Output: A – Value read (1AH when CTRL+Z).

F374H, 3 – AUXOUT
Jump to the auxiliary device exit routine.
Input: A – Amount to send.

F377H, 3 – BLDCHK
Jump to the 'BLOAD' command routine. The address pointed to by F378H/F379H is the highest RAM address available for Disk BASIC. Contains JP 0000H under MSXDOS.

F37AH, 3 – BSVCHK
Jump to the 'BSAVE' command routine (Contains JP 0000H under MSXDOS). Input: c – Number of the routine to be called.

F37DH, 3 – ROMBDOS
Jump to BDOS command handler.

*** See also addresses F85FH to F87EH and FB20H to FB34H.

8.2 – SYSTEM AREA FOR MSXDOS2

8.2.1 – Physical information about disks

F1C1H, 1 – Countdown timer for the drives. By setting this counter to 0, the drive motors are stopped.

F1C2H, 1 – 1st sub-counter of the countdown timer for the drive.

F1C3H, 1 – 2nd sub-counter of the countdown timer for the drive.

F1C4H, 1 – Number of the currently active drive.

F1C5H, 1 – Track number where the drive head A: is.

F1C6H, 1 – Track number where the head of drive B: is.

F1C7H, 1 – Logic drive active.

F1C8H, 1 – Number of physical drives connected.

8.2.2 – Hooks called by disk routines (1)

F1C9H, 24 – Routine for printing on the screen a string ending with "$". DE – Starting address of the string.

F1E2H~F1E4H – ?

F1E5H, 3 – Jump to the interrupt handler (only when processing BDOS functions).

F1E8H, 3 – Jump to the BIOS routine 'RDSLT-000CH' (only when processing BDOS functions).

F1EBH, 3 – Jump to the BIOS routine 'WRSLT-0014H' (only when processing BDOS functions).

F1EEH, 3 – Jump to the BIOS routine 'CALSLT-001CH' (only when processing BDOS functions).

F1F1H, 3 – Jump to the BIOS routine 'ENASLT-0024H' (only when processing BDOS functions).

F1F4H, 3 – Jump to the BIOS routine 'CALLF-0030H' (only when processing BDOS functions).

F1F7H, 3 – Jump to the routine for switching to "DOS Mode" (pages 0 and 2 for system segments).

F1FAH, 3 – Jump to the switching routine to "User Mode".

F1FDH, 3 – Jump to the routine that selects the DOS Kernel segments on page 1.

F200H, 3 – Jump to the routine that allocates a segment of 16 Kbytes of RAM.

F203H, 3 – Jump to the routine that releases a segment of 16 Kbytes of RAM.

F206H, 3 – Jump to the BIOS routine 'RDSLT-000CH'.

F209H, 3 – Jump to the BIOS routine 'WRSLT-0014H'.

F20CH, 3 – Jump to the BIOS routine 'CALSLT-001CH'.

F20FH, 3 – Jump to the BIOS routine 'CALLF-0030H'.

F212H, 3 – Jump to the routine that places a 16 Kbyte segment on the page indicated by HL.

F215H, 3 – Jump to the routine that reads the page of the current 16 Kbytes segment. HL – Page read.

F218H, 3 – Jump to the routine that enables the 16 Kbyte segment of the mapped memory on page 0.

F21BH, 3 – Jump to the routine that reads the current 16 Kbytes segment of the mapped memory on page 0.

F21EH, 3 – Jump to the routine that enables the 16 Kbyte segment of the mapped memory on page 1.

F221H, 3 – Jump to the routine that reads the current 16 Kbytes segment of the mapped memory on page 1.

F224H, 3 – Jump to the routine that enables the 16 Kbyte segment of the mapped memory on page 2.

F227H, 3 – Jump to the routine that reads the current 16 Kbytes segment of the mapped memory on page 2.

F22AH, 3 – Page 3 does not support segment change.

F22DH, 3 – Jump to the routine that reads the current 16 Kbytes segment of the mapped memory on page 3.

F230H~F23BH – ?

8.2.3 – Logical information about disks

F23CH, 1 – Current logical drive (0 = A:, 1 = B:, etc.).

F23DH, 2 – Current DTA address.

F23FH, 4 – Current sector number for access.

F243H, 2 – DPB address of the current drive.

F245H, 1 – Relative number of the current sector of the directory area.

F246H, 1 – Drive number of the current directory (0 = A :, 1 = B :, etc.).

F247H, 1 – Default drive number (0 = A :, 1 = B :, etc.).

F248H, 3 – +0 = Day / +1 = Month / +2 = Year-1980 (Add 1980 to obtain the correct year)

F24CH, 1 – ?

F24CH, 2 – Hour

F24EH, 1 – Day of the week

8.2.4 – Hooks called by disk routines

F24FH, 3 – H.PROM

Jump to the routine that displays the message "Insert disk for drive". A – Drive pain number (41H = A :, 42H = B :, etc)

F252H, 3 – Hook called before the execution of a BDOS function. Page 0 – Block map (F2D0H). Page 2 – Block map (F2CFH).

F255H, 3 – Hook of the filename repair routine.

F258H, 3 – Hook of the disk BASIC subdirectory manipulation routine. Used by several other routines.

F25BH, 3 – Hook of the routine that increments the directory entry. The new entry is stored in AF

F25EH, 3 – Hook of the routine that loads the next sector of the directory.

F261H, 3 – Hook of function 02H of BDOS.

F264H, 3 – OPEN Routine

F267H, 3 – Returns the last FAT

F26AH, 3 – Looking for the first FCB (SFIRST)

F26DH, 3 – Writes the FAT.

F270H, 3 – Hook of the sector direct reading routine (function 2FH of BDOS). HL – DMA, DE – Initial sector, B – Number of sectors.

F273H, 3 – Disk error.

F276H, 3 – Write hook in the subdirectory sector (folder)

F279H, 3 – Hook of the direct sector writing routine (BDOS function 30H). HL – DMA, DE – Initial sector, B – Number of sectors.

F27CH, 3 – Hook of the multiplication routine (HL = DE * BC).

F27FH, 3 – Hook of the division routine (BC = BC/DE; HL = Rest).

F282H~F282H – ?

8.2.5 – MSXDOS2 variables

F2B3H, 2 – User defined TPA address. The initial 32 bytes of the TPA are used for special functions:

Off set	Description
00H~02H	Reserved
03H	Used by VDP speed (bit 3 of F2B6H)
04H~1FH	Reserved
20H	BDOS expansion and interruption routines

F2B5H, 1 – ?

F2B6H, 1 – Byte of flags:
b0~b2 – Reserved
b3 – Fast VDP (0 = yes; 1 = no)
b4 – User TPA address (0 = yes; 1 = no)
b5 – Reset (0 = no; 1 = yes)
b6 – BusReset (0 = yes; 1 = no)
b7 – Reboot (0 = no; 1 = yes)

F2B7H, 1 – Version number (usually 10H = v1.0).

F2B8H, 1 – Number of the current directory entry.

F2B9H~F2BFH – ?

F2C0H, 5 – Second hook of the interrupt routine (used by Disk-ROM).

F2C5H, 2 – Mapping table address.

F2C7H, 1 – Current logical page of the mapper on physical page 0.

F2C8H, 1 – Current logical page of the mapper on physical page 1.

F2C9H, 1 – Current logical page of the mapper on physical page 2.

F2CAH, 1 – Current logical page of the mapper on physical page 3 (cannot be changed).

F2CBH, 1 – Copy of F2C7H during the execution of BDOS routines.

F2CCH, 1 – Copy of F2C8H during the execution of BDOS routines.

F2CDH, 1 – Copy of F2C9H during the execution of BDOS routines.

F2CEH, 1 – Copy of F2CAH during the execution of BDOS routines.

F2CFH, 1 – Number of the last available 16K block of the memory mapper. During the execution of the BDOS routines, the blocks are exchanged on page 2 (buffer segment).

F2D0H, 1 – Number of the last available 16K block of the memory mapper. During the execution of the BDOS routines, the blocks are exchanged on page 0 (code segment).

F2D1H~F2D4H – ?

F2D5H, 5 – Second EXTBIO hook (hook routine FCALL [FFCAH]).

F2DAH, 4 – Address of the second BDOS ROM for handling functions.

F2DEH, 4 – BDOS ROM address for handling functions.

F2E2H~F2E5H – ?

F2E6H, 2 – Buffer used for temporary storage of register IX.

F2E8H, 2 – Buffer used for temporary storage of the SP register.

F2EAH, 1 – Status of the primary slots after the execution of a BDOS function.

F2EBH, 1 – Same as F2EAH, but for secondary slots

F2ECH, 1 – Flag for checking the disk status. (00H = Off, FFH = On).

F2EDH~F2FAH – ?

F2FBH, 2 – Pointer to a temporary buffer when interpreting an error code.

F2FDH, 1 – Drive from which MSXDOS2.SYS should be loaded. (01H = A :, 02H = B :, etc).

F2FEH, 2 – Address of the top of the DOS buffer stack.

F300H, 1 – Check flag (00H = Off, FFH = On).

F301H~F30CH – ?

F30DH, 1 – Check disk flag (00H = Off, FFH = On).

F30EH~F312H – ?

F313H, 1 – Contains the version of the ROM of the MSXDOS.
00H = version 1.x; 20H = version 2.0; 21H = 21H = version 2.1; etc.
Obs: The Nextor returns 99H.

F314H~F322H – ?

F323H, 2 – DISKVE
Disk error handler address.

F325H, 2 – BREAKV
Address of the CTRL+C key handler.

F327H~F33CH – ?

F33DH, 3 – Jump to the BASIC 'LEN' command (random file access).

F341H, 1 – RAMD0
Slot of page 0 of RAM (format equal to 'RDSLT' – 000CH / Main).

F342H, 1 – RAMD1
Slot of page 1 of RAM (format equal to 'RDSLT' – 000CH / Main).

F343H, 1 – RAAD2
Slot of page 2 of RAM (format equal to 'RDSLT' – 000CH / Main).

F344H, 1 – RAAD3
Slot of page 3 of RAM (format equal to 'RDSLT' – 000CH / Main).

F345H, 1 – ?

F346H, 1 – MSXDOS
Flag to indicate whether the system was booted from MSXDOS on a floppy disk. (0 = no; other value, yes)

F347H, 1 – ?

F348H, 1 – MASTER
Primary DOS Kernel slot ID (master). In the case of DOS2 it is the primary interface that contains the DOS2 ROM. The format is the same as RDSLT – 000CH /BIOS).

8.2.6 – Pointers and buffers (FAT, DTA, FCB, DPB)

F349H, 2 – HIMSAV
Pointer to a copy of the FAT of the last connected logical drive (1.5 Kbytes) followed by a copy of the FAT of the penultimate connected logical drive (1.5 Kbytes) and so on, up to drive A :. It also indicates the highest area of memory available to the user.

F34BH~F34CH – ?

F34DH, 2 – SECBUF
Pointer to a copy of the FAT of the default drive (1.5 Kbytes).

F34FH, 2 – BUFFER
Pointer to a 512-byte area used as the DTA of the Disk-BASIC.

F351H, 2 – DIRBUF
Pointer to a 512-byte buffer used for transferring sectors of the disk.

F353H, 2 – FCBBASE
Point to the FCB of the current file.

F355H, 16 – DPBLIST
List of pointers to the DPBs of all eight possible drives, reserving two bytes for each one.

F355H, 2 – Drive A:	F35DH, 2 – Drive E:
F357H, 2 – Drive B	F35FH, 2 – Drive F:
F359H, 2 – Drive C:	F361H, 2 – Drive G:
F35BH, 2 – Drive D:	F363H, 2 – Drive H:

F364H~F377H – ?

8.2.7 – System jumps

F378H – BLDCHK+1
Routine address of the 'BLOAD' command handler.

F37AH, 3 – Secondary jump to the system segment on p. 0.

F37DH – BDOS
Jump to the BDOS function handler.

*** See also addresses F85FH to F87EH and FB20H to FB34H.

8.3 – INTER-SLOT SUBROUTINES

RDPRIM (F380H)
Function: Reads a byte from any address in any slot.
Input: A – Primary slot to be read
D – Current return slot
Output: E – Byte read
Code:

```
F380H RDPRIM: OUT   (0A8H), A
F382H         LD    E, (HL)
F383H         JR    WRPRM1
```

WRPRIM (F385H)
Function: Writes a byte to any address in any slot.
Input: A – Primary slot to be read
D – Current return slot
E – Byte to be written
Output: None
Code:
```
F385H WRPRIM: OUT  (0A8H),A
F387H         LD   (HL), E
F388H WRPRM1: LD   A,D
F389H         OUT  (0A8H),A
```

CLPRIM (F38CH)
Function: Calls an address in any slot.
Input: A – Primary slot containing the routine
IX – Address to be called
PUSH AF – Current return slot (in A)
Output: Depends on the routine called
Code:
```
F38CH CLPRIM: OUT  (0A8H),A
F38EH         EX   AF, AF'
F38FH         CALL CLPRM1
F392H         EX   AF,AF'
F393H         POP  AF
F394H         OUT  (0A8H),A
F396H         EX   AF,AF'
F397H         RET
F398H CLPRM1: JP   (IX)
```

8.4 – USR FUNCTION AND TEXT MODES

USRTAB (F39AH, 20)
Initial value: FCERR
Content: There are ten system variables of two bytes each that point to the execution address of an assembly routine to be called by the USR function. The first position points to USR0, the second to USR1, and so on. The initial value points to the error generator routine.

LINL40 (F3AEH, 1)
Initial value: 39
Content: Screen width in Screen 0 text mode.

LINL32 (F3AFH, 1)
Initial value: 29
Content: Screen width in Screen 1 text mode.

LINLEN (F3B0H, 1)
Initial value: 39
Content: Current width of the text screen.

CRTCNT (F3B1H, 1)
Initial value: 24
Content: Number of lines in text modes.

CLMSLT (F3B2H, 1)
Initial value: 14
Content: Horizontal location in the case of items divided by commas in the PRINT command.

8.5 – AREA USED BY THE SCREEN

8.5.1 – Screen 0

TXTNAM (F3B3H, 2)
Initial value: 0000H
Content: Address in the VRAM of the pattern name table.

TXTCOL (F3B5H, 2) – Variable not used

TXTCGP (F3B7H, 2)
Initial value: 0800H
Content: Address in the VRAM of the character pattern table.
Note: In this variable lies the only bug found on MSX2 computers. When Screen 0 is given the command WIDTH up to 40, the value will be correct. However, if the WIDTH command is 41 to 80, the correct value will be 1000H, but this variable will continue with 0800H value. In this case, when working with an assembly program from BASIC, an ADD HL,HL instruction must be used to correct the value. In the MSX2+ and MSX turbo R models, the correct value for this variable is 0000H, so that the instruction shown does not affect compatibility, despite the fact that this bug does not exist in these models.

TXTATR (F3B9H, 2) – Variable not used

TXTPAT (F3BBH, 2)
Initial value: 0000H
Content: Variable not used

8.5.2 – Screen 1

T32NAM (F3BDH, 2)
Initial value: 1800H
Content: Address of the pattern name table.

T32COL (F3BFH, 2)
Initial value: 2000H
Content: Address in the VRAM of the color table.

T32CGP (F3C1H, 2)
Initial value: 0000H
Content: Address in VRAM of the pattern table.

T32ATR (F3C3H, 2)
Initial value: 1B00H
Content: Address in VRAM of the table of attributes of the sprites.

T32PAT (F3C5H, 2)
Initial value: 3800H
Content: Address in VRAM of the sprite pattern table.

8.5.3 – Screen 2

GRPNAM (F3C7H, 2)
Initial value: 1800H
Content: Address in the VRAM of the pattern name table.

GRPCOL (F3C9H, 2)
Initial value: 2000H
Content: Address in the VRAM of the color table.

GRPCGP (F3CBH, 2)
Initial value: 0000H
Content: Address in the VRAM of the pattern table.

GRPATR (F3CDH, 2)
Initial value: 1B00H
Content: Address in VRAM of the table of attributes of the sprites.

GRPPAT (F3CFH, 2)
Initial value: 3800H
Content: Address in VRAM of the sprite pattern table.

8.5.4 – Screen 3

MLTNAM (F3D1H, 2)
Initial value: 0800H
Content: Address of the pattern name table.

MLTCOL (F3D3H, 2) – Variable not used.

MLTCGP (F3D5H, 2)
Initial value: 0000H
Content: Address in VRAM of the standards table.

MLTATR (F3D7H, 2)
Initial value: 1B00H
Content: Address in VRAM of the table of attributes of the sprites.

MLTPAT (F3D9H, 2)
Initial value: 3800H
Content: Address in VRAM of the sprite standards table.

8.5.5 – Other Screen Values

CLIKSW (F3DBH, 1)
Initial value: 1
Content: Turn key click on / off (0 = Off; other value, on). It can be changed by the SCREEN command.

CSRY (F3DCH, 1)
Initial value: 1
Content: Y (vertical) coordinate of the cursor in text modes.

CSRX (F3DDH, 1)
Initial value: 1
Content: X (horizontal) coordinate of the cursor in text modes.

CNSDFG (F3DEH, 1)
Initial value: 0
Content: Turns on / off the display of the function keys (0 = On, other value, off). It can be changed by the KEY ON/OFF command.

8.6 – VDP REGISTERS AREA

RG0SAV (F3DFH, 1)
Content: Copy of the VDP register R#0.

RG1SAV (F3E0H, 1)
Content: Copy of the VDP register R#1.

RG2SAV (F3E1H, 1)
Content: Copy of the VDP register R#2.

RG3SAV (F3E2H, 1)
Content: Copy of the VDP register R#3.

RG4SAV (F3E3H, 1)
Content: Copy of the VDP register R#4.

RG5SAV (F3E4H, 1)
Content: Copy of the VDP register R#5.

RG6SAV (F3E5H, 1)
Content: Copy of the VDP register R#6.

RG7SAV (F3E6H, 1)
Content: Copy of the VDP register R#7.

STATFL (F3E7H, 1)
Content: Copy of the VDP status register. On MSX2 or upper, stores the contents of the S#0 register.

8.6.1 – Area used for the V9938

RG8SAV (FFE7H, 1)
Content: Copy of the VDP register R#8.

RG9SAV (FFE8H, 1)
Content: Copy of the VDP register R#9.

R10SAV (FFE9H, 1)
Content: Copy of the VDP register R#10.

R11SAV (FFEAH, 1)
Content: Copy of the VDP register R#11.

R12SAV (FFEBH, 1)
Content: Copy of the VDP register R#12.

R13SAV (FFECH, 1)
Content: Copy of the VDP register R#13.

R14SAV (FFEDH, 1)
Content: Copy of the VDP register R#14.

R15SAV (FFEEH, 1)
Content: Copy of the VDP register R#15.

R16SAV (FFEFH, 1)
Content: Copy of the VDP register R#16.

R17SAV (FFF0H, 1)
Content: Copy of the VDP register R#17.

R18SAV (FFF1H, 1)
Content: Copy of the VDP register R#18.

R19SAV (FFF2H, 1)
Content: Copy of the VDP register R#19.

R20SAV (FFF3H, 1)
Content: Copy of the VDP register R#20.

R21SAV (FFF4H, 1)
Content: Copy of the VDP register R#21.

R22SAV (FFF5H, 1)
Content: Copy of the VDP register R#22.

R23SAV (FFF6H, 1)
Content: Copy of the VDP register R#23.

8.6.2 – Area used for the V9958

R25SAV (FFFAH, 1)
Content: Copy of the VDP register R#25 (V9958).

R26SAV (FFFBH, 1)
Content: Copy of the VDP register R#26 (V9958).

R27SAV (FFFCH, 1)
Content: Copy of the VDP register R#27 (V9958).

8.7 – MISCELLANEOUS

TRGFLG (F3E8H, 1)
Initial value: 11110001B
Content: Status of the joystick buttons. (0 = pressed, 1 = not pressed). This variable is constantly updated by the interrupt handler.

FORCLR (F3E9H, 1)
Initial value: 15
Content: Front and character color. It can be changed by the COLOR command.

BAKCLR (F3EAH, 1)
Initial value: 4
Content: Background color. It can be changed by the COLOR command.

BDRCLR (F3EBH, 1)
Initial value: 7
Content: Color of the border. It can be changed by the COLOR command.

MAXUPD (F3ECH, 3)
Initial value: JP 0000H (C3H, 00H, 00H)
Content: Used internally by the CIRCLE command.

MINUPD (F3EFH, 3)
Initial value: JP 0000H (C3H, 00H, 00H)
Content: Used internally by the CIRCLE command.

ATRBYT (F3F2H, 1)
Initial value: 15
Content: Color code used for graphics.

8.8 – AREA USED BY PLAY COMMAND

QUEUES (F3F3H, 2)
Initial value: QUETAB (F959H)
Content: Pointer to the PLAY command execution queue.

FRCNEW (F3F5H, 1)
Initial value: 255
Content: Used internally by the BASIC interpreter.

MCLTAB (F956H, 2)
Content: Address of the command table to be used by the PLAY and DRAW macro commands.

MCLFLG (F958H, 1)
Content: Flag to indicate which command is being processed (0 = DRAW; not zero, PLAY).

QUETAB (F959H, 24)
Content: This table contains the data for the three musical queues and RS232C queue, with six bytes reserved for each one.
+0: position to place
+1: position to get
+2: return indication
+3: size of the buffer in the queue
+4: address of the buffer in the queue (high)
+5: address of the buffer in the queue (low)

F959H = voz A
F95FH = voz B
F965H = voz C
F96AH = RS232C

QUEBAK (F971H, 4)
Content: Used for replacement characters table of queues
F971H +0 – Voice A
+1 – Voice B
+2 – Voice C
+3 – RS232C (MSX1 only)

PRSCNT (FB35H, 1)
Content: Used internally by the PLAY command to count the number of operands completed. Bit 7 will be turned on after each of the three operands is analyzed.

SAVSP (FB36H, 2)
Content: Saves the value of the SP register before executing the PLAY command.

VOICEN (FB38H, 1)
Content: Number of the voice currently being processed (0, 1 or 2).

SAVVOL (FB39H, 2)
Content: Saves the volume when generating a pause.

MCLPTR (FB3CH, 2)
Content: Address of the operand being analyzed.

QUEUEN (FB3EH, 1)
Content: Used by the interrupt handler to contain the number of the musical queue that is currently being processed.

MUSICF (FB3FH, 1)
Content: Flag to indicate which musical queues will be used.

PLYCNT (FB40H, 1)
Content: Number of strings stored in the PLAY command queue.

MCLLEN (FB3BH, 1)
Content: Length of the string being analyzed.

8.8.1 – Play statement queues

VOICAQ (F975H, 128)
Initial value: DEFS 128 (00H ..... 00H)
Content: Queue for voice A.

VOICBQ (F9F5H, 128)
Initial value: DEFS 128 (00H ..... 00H)
Content: Queue for voice B.

VOICCQ (FA75H, 128)
Initial value: DEFS 128 (00H ..... 00H)
Content: Queue for voice C.

8.8.2 – Offset for PLAY buffer parameter control

METREX	(+00, 2)	Duration counter
VCXLEN	(+02, 1)	String length
VCXPTR	(+03, 2)	String address
VCXSTP	(+05, 2)	Data address on the stack
QLENGX	(+07, 1)	Size of the musical packet in bytes
NTICSX	(+08, 2)	Music package
TONPRX	(+10, 2)	Tone period
AMPRX	(+12, 1)	Volume and envelope
ENVPRX	(+13, 2)	Envelope period
OCTAVX	(+15, 1)	Octave
NOTELX	(+16, 1)	Tone length
TEMPOX	(+17, 1)	Time
VOLUMX	(+18, 1)	Volume
ENVLPX	(+19, 14)	Envelope waveform
MCLSTX	(+33, 3)	Reserved for the battery
MCLSEX	(+36, 1)	Initialization of the stack
VCBSIZ	(+37, 1)	Size of the parameter buffer

8.8.3 – Data area for the parameter buffer

VCBA (FB41H, 37)
Content: Parameters for voice A.
+00, 2 – Duration counter
+02, 1 – String length
+03, 2 – String address
+05, 2 – Data address on the stack
+07, 1 – Music package size
+08, 7 – Music package
+15, 1 – Eighth
+16, 1 – Length
+17, 1 – Weather
+18, 1 – Volume
+19, 2 – Wrap period
+21,16 – Stack data space

VCBB (FB66H, 37)
Content: Parameters for voice B.
(Structure identical to voice A).

VCBC (FB8BH, 37)
Content: Parameters for the C voice.
(Structure identical to voice A).

8.9 – KEYBOARD AREA

SCNCNT (F3F6H, 1)
Initial value: 1
Content: Interval for scanning the keys.

REPCNT (F3F7H, 1)
Initial value: 50
Content: Delay time for the start of the auto-repeat of the keys.

PUTPNT (F3F8H, 2)
Initial value: KEYBUF (FBF0H)
Content: Points to the write address of the keyboard buffer.

GETPNT (F3FAH, 2)
Initial value: KEYBUF (FBF0H)
Content: Points to the reading address of the keyboard buffer.

8.10 – AREA USED BY CASSETTE

CS1200 (F3FCH, 5)
Initial value: +0 ⟶ 53H – first half for bit 0
+1 ⟶ 5CH – second half for bit 0
+2 ⟶ 26H – first half for bit 1
+3 ⟶ 2DH – second half for bit 1
+4 ⟶ 0FH – cycle count for short header
[cycles = (0F400H) * 2/256]
Content: Parameters for the 1200 baud cassette.

CS2400 (F401H, 5)
Initial value: +0 ⟶ 25H – first half for bit 0
+1 ⟶ 2DH – second half for bit 0
+2 ⟶ 0EH – first half for bit 1
+3 ⟶ 16H – second half for bit 1
+4 ⟶ 1FH – count of cycles for short header
[cycles = (0F405H) * 4/256]
Content: Parameters for the cassette for 2400 baud.

LOW (F406H, 2)
Initial value: +0 ⟶ 53H – first half for bit 0
+1 ⟶ 5CH – second half for bit 0
Content: Width for bit 0 of the current baud rate.

HIGH (F408H, 2)
Initial value +0 ⟶ 26H – first half for bit 1
+1 ⟶ 2DH – second half for bit 1
Content: Width for bit 1 of the current baud rate.

HEADER (F40AH, 1)
Initial value: 0FH
Content: Count of cycles for current short header.

8.11 – AREA USED BY CIRCLE COMMAND

ASPCT1 (F40BH, 2)
Content: 256 / aspect ratio. Can be changed by SCREEN command for use of the CIRCLE command.

ASPCT2 (F40DH, 2)
Content: 256 * aspect ratio. Can be changed by SCREEN command for use of the CIRCLE command.

ASPECT (F931H, 2)
Initial value: 0
Content: Aspect ratio.

CENCNT (F933H, 2)
Initial value: 0
Content: Count of points of the final angle.

CLINEF (F935H, 1)
Initial value: 0
Content: Flag used to indicate the drawing of a line from the center of the circle. Bit 0 will be turned on if a line is required from the start angle and bit 7 will be turned on if the line is required from the end angle.

CNPNTS (F936H, 2)
Initial value: 0
Content: Number of points within a 45 degree segment of the circumference.

CPLOTF (F938H, 1)
Content: Used internally by the CIRCLE command.

CPCNT (F939H, 2)
Content: Y coordinate within the current 45 degree segment of the circumference.

CPCNT8 (F93BH, 2)
Initial value: 0
Content: Total point count of the current position.

CPCSUM (F93DH, 2)
Initial value: 0
Content: Counter of points for computation.

CSTCNT (F93FH, 2)
Initial value: 0
Content: Count of points of the initial angle of the circumference.

CSCLXY (F941H, 1)
Initial value: 0
Content: Scales between X and Y. Used by the CIRCLE command.

CSAVEA (F942H, 2)
Content: Area reserved for ADVGRP.

CSAVEM (F944H, 1)
Content: Area reserved for ADVGRP.

CXOFF (F945H, 2)
Content: X coordinate from the center of the circle.

CYOFF (F947H, 2)
Content: Y coordinate from the center of the circle.

8.12 – AREA INTERNALLY USED BY BASIC

ENDPRG (F40FH, 5)
Initial value: ":"; 00H; 00H; 00H; 00H.
Content: False end of program line for RESUME and NEXT commands.

ERRFLG (F414H, 1)
Initial value: 00H
Content: Area to save the error number.

LPTPOS (F415H, 1)
Initial value: 00H
Content: Stores the current position of the printer head.

PRTFLG (F416H, 1)
Initial value: 00H
Content: Flag to select output for screen or printer (0 = Screen; other value, printer).

NTMSXP (F417H, 1)
Initial value: 00H
Content: Flag to select the type of printer. (0 = Standard MSX printer; other value, non-MSX printer). It can be changed by the SCREEN command.

RAWPRT (F418H, 1)
Initial value: 00H
Content: Flag to determine whether the graphic control characters will be modified when sent to the printer (0 = modify; another value, does not modify).

VLZADR (F419H, 2)
Initial value: 0000H
Content: Character address for the VAL function.

VLZDAT (F41BH, 1)
Initial value: 00H
Content: Character that should be replaced by 0 with VAL function.

CURLIN (F41CH, 2)
Initial value: FFFFH
Content: Current line number of the BASIC interpreter. The FFFFH value indicates direct mode.

8.12.1 – BASIC text buffers

KBFMIN (F41EH, 1)
Initial value: ":"
Content: This byte is a fictitious prefix for the tokenized text contained in KBUF.

KBUF (F41FH, 318)
Initial value: 00H, 00H,... 00H
Content: This buffer stores the collected tokenized BASIC line by the interpreter.

BUFMIN (F55DH, 1)
Initial value: ":"
Content: Fictitious prefix for the text contained in BUF. It is used to synchronize the INPUT instruction handler when it starts analyzing the collected text.

BUF (F55EH, 258)
Initial value: 00H, 00H,... 00H
Content: This buffer stores, in ASCII format, the characters collected from the keyboard by the standard INLIN routine.

ENDBUF (F660H, 1)
Initial value: 00H
Content: Byte to prevent BUF overflow (F55EH).

8.12.2 – General data

TTYPOS (F661H, 1)
Content: Used by the PRINT command to store the virtual cursor position.

DIMFLG (F662H, 1)
Content: Used internally by the DIM command.

VALTYP (F663H, 1)
Content: Stores the type of variable contained in DAC (F3F6H):
2 = Integer; 3 = String; 4 = Simple precision; 8 = Double prec.

DORES (F664H, 1)
Content: Used internally by the DATA command to keep the text in ASCII format.

DONUM (F665H, 1)
Content: Flag used internally by BASIC.

CONTXT (F666H, 2)
Content: Stores the address of the text used by the CHRGTR routine.

CONSAV (F668H, 1)
Content: Stores the token of a numeric constant; used by the GHRGTR routine.

CONTYP (F669H, 1)
Content: Stores the type of a numeric constant found in the BASIC program text. It is used by the prairie CHRGTR routine.

CONLO (F66AH, 8)
Content: Stores a numeric constant used by standard routine CHRGTR.

MEMSIZ (F672H, 2)
Content: Highest memory address that can be used by BASIC.

STKTOP (F674H, 2)
Content: Address of the top of the Z80 stack. Used internally by BASIC.

TXTTAB (F676H, 2)
Initial value: 8000H
Content: Starting address of the BASIC text area.

TEMPPT (F678H, 2)
Initial value: TEMPST (F67AH)
Content: Address of the next free position in TEMPST.

TEMPST (F67AH, 30)
Initial value: DEFS 30 (00H, ... 00H)
Content: Buffer used to store string descriptors.

DSCTMP (F698H, 3)
Initial value: 00H, 00H, 00H
Content: Saves the string descriptor during processing.

FRETOP (F69BH, 2)
Initial value: F168H
Content: Address of the next free position in the string area.

TEMP3 (F69DH, 2)
Initial value: 0000H
Content: Used internally by the interpreter for temporary storage of multiple routines.

ENDFOR (F6A1H, 2)
Initial value: 0000H
Content: Address for the FOR command.

DATLIN (F6A3H, 2)
Initial value: 00H
Content: Line number of the DATA command to use the READ command.

SUBFLG (F6A5H, 1)
Initial value: 00H
Content: Flag used to control the processing of indexes when searching for matrix type variables.

FLGINP (F6A6H, 1)
Initial value: 00H
Content: Flag used by the INPUT and READ commands (0 = INPUT; another value, READ).

TEMP (F6A7H, 2)
Content: Used internally by the interpreter.

8.12.3 – BASIC lines control at runtime

PTRFLG (F6A9H, 1)
Content: Used internally by the interpreter to convert line numbers to pointers (0 = Operand not converted; another value, operand converted).

AUTFLG (F6AAH, 1)
Content: Flag for the AUTO command (0 = AUTO command inactive; another value, AUTO command active).

AUTLIN (F6ABH, 2)
Content: Number of the last BASIC line entered.

AUTINC (F6ADH, 2)
Initial value: 10
Content: Increment value for the AUTO function.

SAVTXT (F6AFH, 2)
Content: Stores the current BASIC text address during the execution.

SAVSTK (F6B1H, 2)
Content: Stores the current address of the Z80 stack. Used by the error handler and the RESUME statement.

ERRLIN (F6B3H, 2)
Content: BASIC line number where an error occurred.

DOT (F6B5H, 2)
Content: Last line number during processing. Used internally by the interpreter and the error handler.

ERRTXT (F6B7H, 2)
Content: Address of the BASIC text where an error occurred. Used by the RESUME command.

ONELIN (F6B9H, 2)
Content: Address of the program line that must be executed when an error occurs. Set by the ON ERROR GOTO command.

ONEFLG (F6BBH, 1)
Content: Flag to indicate the execution of an error routine (0 = not executing; another value, routine in execution).

TEMP2 (F6BCH, 2)
Content: Used internally by the interpreter.

OLDLIN (F6BEH, 2)
Content: Stores the last line executed by the program. It is updated by the END and STOP commands to be used by the CONT command.

8.12.4 – BASIC text storage adresses

OLDTXT (F6C0H, 2)
Content: Stores the address of the last instruction in the BASIC text.

VARTAB (F6C2H, 2)
Content: Address of the first byte of the storage area of the BASIC variables.

ARYTAB (F6C4H, 2)
Content: Address of the first byte of the storage area of the BASIC arrays.

STREND (F6C6H, 2)
Content: Address of the first byte after the storage area for arrays, variables or BASIC text.

DATPTR (F6C8H, 2)
Content: Address of the current DATA command to use the READ command.

DEFTBL (F6CAH, 26)
Content: Storage area of the variable type by names in alphabetical order. They can be changed by the command group "DEF xxx".

8.12.5 – Area for user functions

PRMSTK (F6E4H, 2)
Content: Previous definition of the block in the Z80 stack.

PRMLEN (F6E6H, 2)
Content: Length of the current "FN" parameter block in PARM1.

PARM1 (F6E8H, 100)
Content: Buffer for storing the variables of the "FN" function being evaluated.

PRMPRV (F74CH, 2)
Initial value: PRMSTK (F6E4H)
Content: Address of the previous "FN" parameter block.

PRMLN2 (F74EH, 2)
Content: Length of the "FN" parameter block being assembled in PARM2.

PARM2 (F750H, 100)
Content: Buffer used for the variables of the current "FN" function.

PRMFLG (F7B4H, 1)
Content: Flag to indicate when PARM1 is being searched.

ARYTA2 (F7B5H, 2)
Content: Last address to search for a variable.

NOFUNS (F7B7H, 1)
Content: Flag to indicate to the "FN" function the existence of local variables (0 = there are no variables; another value, there are variables).

TEMP9 (F7B8H, 2)
Content: Used internally by the interpreter.

FUNACT (F7BAH, 2)
Content: Number of "FN" functions currently active.

SWPTMP (F7BCH, 8)
Content: Buffer used to contain the first operand of a SWAP command.

TRCFLG (F7C4H, 1)
Content: Flag for the TRACE command (0 = TRACE OFF, another value, TRACE ON).

8.12.6 – Interpreter data area

FNKSTR (F87FH, 160)
Content: Reserved area to store the content of the function keys (10 positions with 16 characters each).

CGPNT (F91FH, 3)
Content: Address of the character font. The first byte is the slot ID and the other two is the address.

NAMBAS (F922H, 2)
Content: Address of the name table in the current text mode.

CGPBAS (F924H, 2)
Content: Address of the pattern generator table in the current text mode.

PATBAS (F926H, 2)
Content: Current address of the sprite generator table.

ATRBAS (F928H, 2)
Content: Current address of the sprites attribute table.

CLOC (F92AH, 2)
Content: Used internally by graphic routines.

CMASK (F92CH, 1)
Content: Used internally by graphic routines.

MINDEL (F92DH, 2)
Content: Used internally by the LINE command.

MAXDEL (F92FH, 2)
Content: Used internally by the LINE command.

8.13 – MATH-PACK AREA

FBUFFR (F7C5H, 43)
Content: Used internally by MATH-PACK.

DECTMP (F7F0H, 2)
Content: Used to transform an integer into a floating point number.

DECTM2 (F7F2H, 2)
Content: Used internally by the division routine.

DECCNT (F7F4H, 1)
Content: Used internally by the division routine.

DAC (F7F6H, 16)
Content: Primary accumulator that contains a number during a mathematical operation.

HOLD8 (F806H, 48)
Initial value: 00H, 00H... 00H
Content: Storage area for decimal multiplication.

HOLD2 (F836H, 8)
Initial value: 00H, 00H... 00H
Content: Used internally by MATH-PACK.

HOLD (F83EH, 8)
Initial value: 00H, 00H... 00H
Content: Used internally by MATH-PACK.

ARG (F847H, 16)
Content: Secondary accumulator that contains the number to be calculated with DAC (F7F6H).

RNDX (F857H, 8)
Content: Stores the last double-precision random number. Used by the RND function.

8.14 – DISK SYSTEM DATA AREA

MAXFIL (F85FH, 1)
Content: Number of existing I/O buffers. It can be changed by the MAXFILES statement.

FILTAB (F860H, 2)
Content: Initial address of the data area of the files.

NULBUF (F862H, 2)
Content: Points to the buffer used by commands SAVE and LOAD.

PTRFIL (F864H, 2)
Content: Address of the data of the currently active file.

RUNFLG (F866H, 0)
Content: Non-zero, if any programs have been loaded and executed. Used by the ", R" operand of the LOAD command.

FILNAM (F866H, 11)
Content: Area for storing a filename.

FILNM2 (F871H, 11)
Content: Area for storing a filename to be compared with FILNAM.

NLONLY (F87CH, 1)
Content: Flag to indicate whether a program is being loaded or not (0 = program is not being loaded; another value, program is being loaded).

SAVEND (F87DH, 2)
Content: Used by the BSAVE command to contain the final address of the assembly program to be saved.

HOKVLD (FB20H, 1)
Initial value: 01H
Content: Bit 0 of this byte indicates the presence of an extended BIOS. (0 = No Extended BIOS, 1 = There is at least one BIOS that can be called at address 0FFCAh (EXTBIO)).

DRVINV (FB21H, 9)
Initial value: variable
Content: Slot ID and num of drives connected to the disk interfaces.
DRVINV +0 = Number of drives connected to the primary disk interface.
+1 = Master disk interface slot ID.
+2 = Number of units connected to the master disk interface.
+3 = 2nd disk interface slot ID.
+4 = Number of units connected to the 2nd disk interface.
+5 = 3rd disk interface slot ID.
+6 = Number of units connected to the 3rd disk interface.
+7 = Slot ID of the 4th disk interface.
+8 = Number of units connected to the 4th disk interface.

DRVINT (FB29H, 12)
Initial value: variable
Content: Slot ID and address of each disk interface interrupt handler (3*4 bytes).
DRVINT+0 = slot ID of each interrupt handler on the main interface.

+1, +2 = Address of the interrupt handler of the main interface.
+3 = Slot ID of each interrupt handler on the 2nd interface.
+4, +5 = 2nd interface interrupt handler address.
+6 = Slot ID of each interrupt handler on the 3rd interface.
+7, +8 = 3rd interface interrupt handler address.
+9 = Slot ID of each interrupt handler on the 4rd interface.
+10 = 4th interface interrupt handler address.

8.15 – AREA USED BY PAINT COMMAND

LOHMSK (F949H, 1)
Initial value: 0
Content: Leftmost position of the LH tour.

LOHDIR (F94AH, 1)
Initial value: 0
Content: Painting direction required by the LH tour.

LOHADR (F94BH, 2)
Initial value: 0000H
Content: Leftmost position of the LH tour.

LOHCNT (F94DH, 2)
Initial value: 0
Content: Size of the LH tour.

SKPCNT (F94FH, 2)
Initial value: 0
Content: Hop counter returned by SCANR (012CH).

MOVCNT (F951H, 2)
Initial value: 0
Content: Movement counter returned by SCANR (012CH).

PDIREC (F953H, 1)
Content: Painting direction: 40H, down; C0H, upward; 00H, finish.

LFPROG (F954H, 1)
Content: Flag used by the PAINT command to indicate whether there was progress to the left (0 = there was no progress; another value, there was progress).

RTPROG (F955H, 1)
Content: Flag used by the PAINT command to indicate whether there has been progress on the right (0 = there was no progress; another value, there was progress).

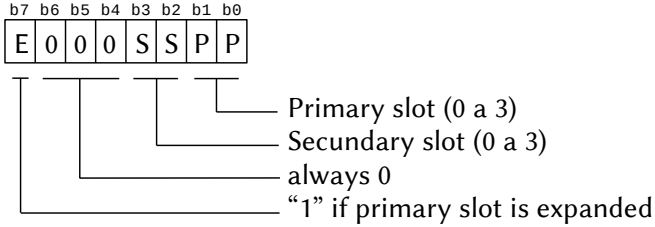
8.16 – ADDED AREA FOR MSX2

DPPAGE (FAF5H, 1)
Initial value: 0
Content: Video page that is currently being displayed.

ACPAGE (FAF6H, 1)
Initial value: 0
Content: Active page for receiving commands.

AVCSAV (FAF7H, 1)
Initial value: 0
Content: Used by the AV control port.

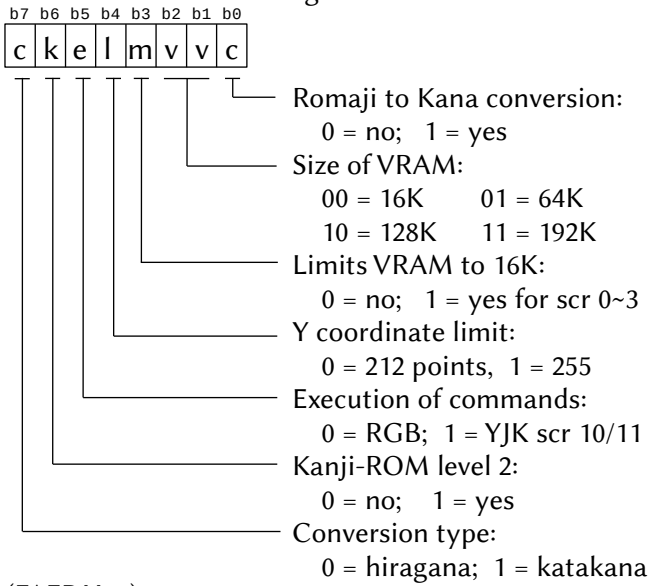
EXBRSA (FAF8H, 1)
Initial value: 10 000 111B
Content: Sub-ROM slot, in the format below:



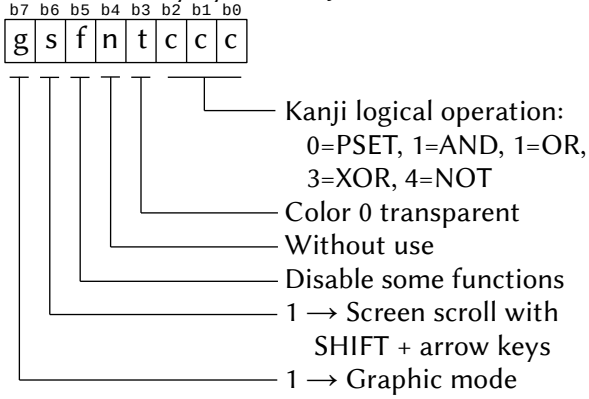
CHRCNT (FAF9H, 1)
Initial value: 0
Content: Character counter in the buffer. Used for Roman-Kana transition (0, 1 or 2).

ROME (FAFAH, 2)
Initial value: 0
Content: Stores the character of the buffer for the Roman-Kana transition (Japanese version only).

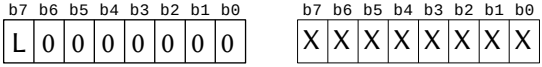
MODE (FAFCH, 1)
Initial value: 1000 1001B
Content: VRAM size and mode flag:



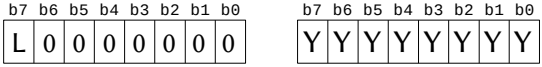
NORUSE (FAFDH, 1)
Initial value: 00H
Content: Used internally by the Kanji-driver.



XSAVE (FAFEH, 2)
Initial value: 00 000 000B, 00 000 000B



YSAVE (FB00H, 2)
Initial value: 00 000 000B, 00 000 000B



L = 1 ⟶ lightpen interruption request
0 000 000 = meaningless
XXXXXXXXX = X coordinate
YYYYYYYY = Y coordinate

LOGOPR (FB02H, 1)
Initial value: 00H
Content: Logical operation code for the VDP.

8.17 – AREA USED BY RS232C

RSTMP (FB03H, 1)
Initial value: 00H
Content: Temporary storage for the RS232C.
Note: Same address as TOCNT.

TOCNT (FB03H, 1)
Initial value: 00H
Content: Counter used by the RS232C interface.
Note: Same address as RSTMP.

RSFCB (FB04H, 2)
Initial value: 0000H
Content: FCB address of RS232C.

RSIQLN (FB06H, 1)
Initial value: 00H
Content: Used internally by the RS232C.

MEXBIH (FB07H, 5)
Initial value: C9H, C9H, C9H, C9H, C9H
Content: Used internally by the RS232C.
FB07H +0: RST 030H
+1: Slot ID byte
+2: Address (low)
+3: Address (high)
+4: RET

OLDSTT (FB0CH, 5)
Initial value: C9H, C9H, C9H, C9H, C9H
Content: Used internally by the RS232C.
FB0CH +0: RST 030H
+1: Slot ID byte
+2: Address (low)
+3: Address (high)
+4: RET

OLDINT (FB12H, 5)
Initial value: C9H, C9H, C9H, C9H, C9H
Content: Used internally by the RS232C.
FB12H +0: RST 030H
+1: Slot ID byte
+2: Address (low)
+3: Address (high)
+4: RET

DEVNUM (FB17H, 1)
Content: Offset byte.

DATCNT (FB18H, 3)
Content: FB18H +0: Slot ID
+1: Pointer
+2: Pointer

ERRORS (FB1BH, 1)
Initial value: 00H
Content: RS232C error code.

FLAGS (FB1CH, 1)
Initial value: 00000011B
Content: Flags used by the RS232C.

ESTBLS (FB1DH, 1)
Initial value: FFH
Content: Boolean bit for use with RS232C.

COMMSK (FB1EH, 1)
Initial value: C1H
Content: RS232C mask.

LSTCOM (FB1FH, 1)
Initial value: E8H
Content: Used internally by the RS232C.

8.18 – GENERAL DATA AREA

ENSTOP (FBB0H, 1)
Content: Flag to enable a forced output for the interpreter when detecting the CTRL + SHIFT + GRAPH + CODE keys pressed together (0 = Disabled; another value, enabled).

BASROM (FBB1H, 1)
Initial value: 00H
Content: Location of BASIC text (0 = RAM; other value, ROM).

LINTTB (FBB2H, 24)
Content: There are 24 flags to indicate whether each line of text screen has advanced to the next line (0 = Advanced; another value, has not advanced).

FSTPOS (FBCAH, 2)
Content: First location of the character collected by the BIOS INLIN (00B1H) routine.

CODSAV (FBCCH, 1)
Initial value: 00H
Content: Character replaced by the cursor in the text screens.

FNKSW1 (FBCDH, 1)
Initial value: 01H
Content: Flag to indicate which function keys are shown when enabled by KEY ON (1 = F1 to F5; 0 = F6 to F10).

FNKFLG (FBCEH, 10)
Content: Flags to enable, inhibit or stop the execution of a line defined by the ON KEY GOSUB command. They are modified by KEY (n) ON / OFF / STOP
(0 = KEY (n) OFF / STOP; 1 = KEY (n) ON).

ONGSBF (FBD8H, 1)
Content: Flag to indicate whether any device required a program interruption (0 = normal; another value = Active interrupt).

CLIKFL (FBD9H, 1)
Content: Click flag of the keys. Used by the interrupt handler.

OLDKEY (FBDAH, 11)
Content: Previous state of the keyboard matrix.

NEWKEY (FBE5H, 11)
Content: New state of the keyboard matrix.

KEYBUF (FBF0H, 40)
Content: Circular buffer containing the decoded keyboard chars.

LINWRK (FC18H, 40)
Content: Buffer used by the BIOS to contain a full line of characters on the screen.

PATWRK (FC40H, 8)
Content: Buffer used by the BIOS to contain an 8x8 char pattern.

BOTTOM (FC48H, 2)
Content: Lowest address used by the interpreter, usually 8000H.

HIMEM (FC4AH, 2)
Content: Highest available RAM address for BASIC interpreter. It can be modified by the CLEAR command.

TRPTBL (FC4CH, 78)
Content: This table contains the current state of the interrupting devices. Each device allocates three bytes in the table. The first byte contains the state of the device (bit 0 = On; bit 1 = stopped; bit 2 = Active). The other two bytes contain the address of the program line to be executed in the interruption event.

FC4CH / FC69H	(3 x 10 bytes)	ON KEY GOSUB
FC6AH / FC6CH	(3 x 1 byte)	ON STOP GOSUB
FC6DH / FC6FH	(3 x 1 byte)	ON SPRITE GOSUB
FC70H / FC7EH	(3 x 5 bytes)	ON STRIG GOSUB
FC7FH / FC81H	(3 x 1 byte)	ON INTERVAL GOSUB
FC82H / FC99H	Reserved for expansion	

RTYCNT (FC9AH, 1)
Content: Used internally by BASIC.

INTFLG (FC9BH, 1)
Content: If CTRL+STOP are pressed, this variable is set to 03H and processing is interrupted; if STOP is pressed, the value is 04H.

PADY (FC9CH, 1)
Content: Y coordinate of the paddle.

PADX (FC9DH, 1)
Content: X coordinate of the paddle.

JIFFY (FC9EH, 2)
Content: This variable is continuously incremented by the interrupt handler. Its value can be read or assigned by the TIME function. It is also used internally by the PLAY command.

INTVAL (FCA0H, 2)
Initial value: 0000H
Content: Duration of the interval used by ON INTERVAL GOSUB.

INTCNT (FCA2H, 2)
Initial value: 0000H
Content: Counter for the ON INTERVAL GOSUB instruction.

LOWLIM (FCA4H, 1)
Initial value: 31H
Content: Minimum duration for the starting bit when reading the cassete.

WINWID (FCA5H, 1)
Initial value: 22H
Content: Duration of the discrimination of the high / low cycle during the reading of the cassette.

GRPHED (FCA6H, 1)
Content: Flag for sending a graphic character.
(0 = normal; 1 = graphic character).

ESCCNT (FCA7H, 1)
Content: Escape codes counting area.

INSFLG (FCA8H, 1)
Content: Flag to indicate the insertion mode (0 = normal; another value, insertion mode)

CSRSW (FCA9H, 1)
Content: Flag to indicate whether the cursor will be shown (0 = no; another value, yes). Can be modified by the LOCATE command.

CSTYLE (FCAAH, 1)
Content: Cursor shape (0 = block; other value, sub-aligned).

CAPST (FCABH, 1)
Content: Status of the CAPS LOCK key (0 = Off; another value, on).

KANAST (FCACH, 1)
Content: Status of the KANA key (0 = Off; another value, on).

KANAMD (FCADH, 1)
Content: Flag used only on Japanese machines.

FLBMEM (FCAEH, 1)
Content: Flag to indicate program loading in BASIC (0 = Is loading; another value, is not).

SCRMOD (FCAFH, 1)
Content: Number of the current screen mode.

OLDSCR (FCB0H, 1)
Content: Screen mode of the last text mode.

CASPRV (FCB1H, 1)
Initial value: 00H
Content: Used by the cassette on MSX1, MSX2 and MSX2+. In the MSX turbo R, it stores the A7H port value.

BDRATR (FCB2H, 1)
Content: Color code of the border. Used by PAINT.

GXPOS (FCB3H, 2)
Content: Graphic X coordinate.

GYPOS (FCB5H, 2)
Content: Graphic Y coordinate.

GRPACX (FCB7H, 2)
Content: Graphic accumulator for the X coordinate.

GRPACY (FCB9H, 2)
Content: Graphic accumulator for the Y coordinate.

DRWFLG (FCBBH, 1)
Content: Flag used by the DRAW command.

DRWSCL (FCBCH, 1)
Content: Scale factor for the DRAW command. A value of 0 indicates that the scale will not be used.

DRWANG (FCBDH, 1)
Content: Angle for the DRAW command.

RUNBNF (FCBEH, 1)
Content: Flag to indicate whether the BLOAD or BSAVE command is running (disk system only).

SAVENT (FCBFH, 2)
Initial value: 0000H
Content: Initial address for the BSAVE and BLOAD commands (disk system only)

8.19 – BIOS EXPANSION ROUTINES

EXTBIO (FFCAH)
Purpose: To directly expand the system BIOS.
Input: A – Always 0.
D – device identifier (device number 0 is used to get installed extensions).
E – function to be called.
Note: Refer to the "EXTENDED BIOS ROUTINES" section for more details.

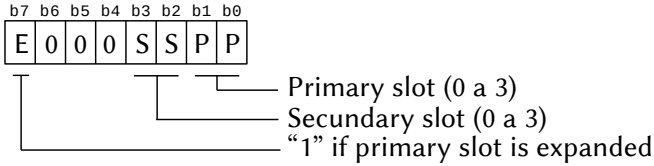
DISINT (FFCFH)
Purpose: Called by function 2 of the "broadcast".
Input: None.

ENAINT (FFD4H)
Purpose: Called by function 2 of the "broadcast".
Input: None.

FFD9H~FFE6H → Contains the code for the DISINT and ENAINT routines.

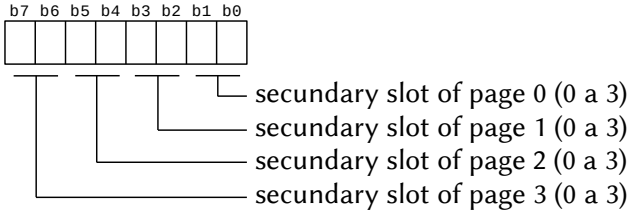
8.20 – DATA AREA FOR SLOTS AND PAGES

EXPTBL (FCC1H, 4)
Initial value: Variable.
Content: Table of flags to indicate whether the primary slots are expanded:
FCC1H → primary slot 0 (Main-ROM slot).
FCC2H → primary slot 0 (Main-ROM slot).
FCC3H → primary slot 0 (Main-ROM slot).
FCC4H → primary slot 0 (Main-ROM slot).
The structure of each flag is described below:



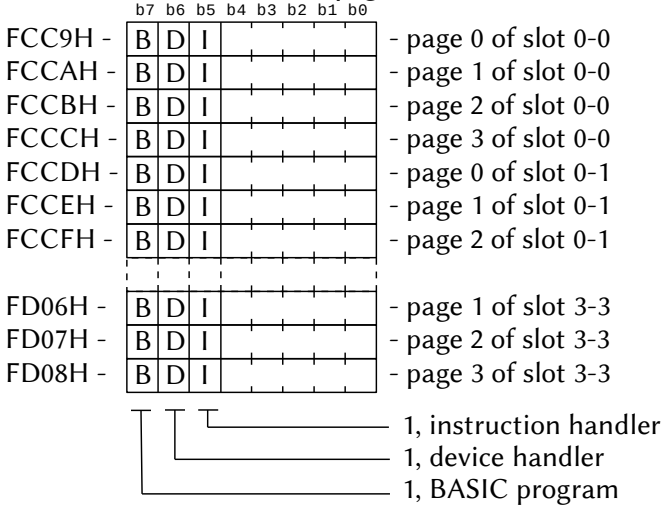
SLTTBL (FCC5H, 4)

Content: These four bytes contain the possible state of the four primary slot registers, in case the slot is expanded.
FCC5H → status for primary slot 0
FCC6H → status for primary slot 1
FCC7H → status for primary slot 2
FCC8H → status for primary slot 3
The structure of each flag is described below:

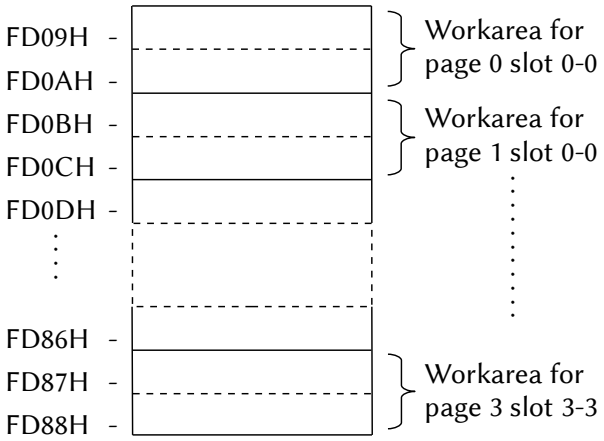


SLTATR (FCC9H, 64)

Content: Table of attributes for each page of each slot.



SLTWRK (FD09H, 128)
Content: This table allocates two bytes as a working area for each page of each slot.



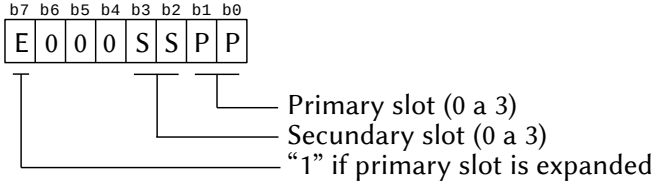
PROCNM (FD89H, 16)
Content: Stores the name of an expanded instruction (CALL command) or device expansion (OPEN command). The end of the name is marked with a byte 0.

DEVICE (FD99H, 1)
Content: Stores the device ID in a cartridge (0 to 3).

FD9AH~FFC9H → Hook area (listed further ahead)

8.20.1 – Main-ROM slot

MINROM (FFF7H, 1)
Content: Main-ROM slot, in the format below:

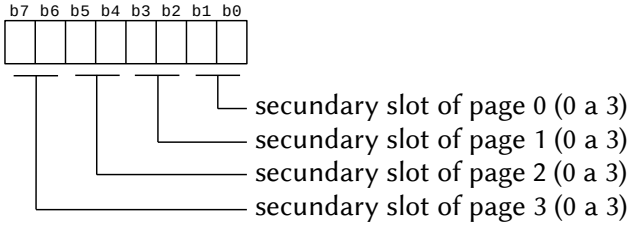


FFF8H~FFF9H → Not used

FFFDH~FFFEH → Not used

8.20.2 – Secondary slot register

SLTSL (FFFFH, 1)
Content: Secondary slot register, in the format below:



8.21 – HOOKS DESCRIPTION

HKEYI (FD9AH)
Called by: Beginning of the interrupt handler (KEYINT, 0038H).
Purpose: Add interrupt handling routines. It can also be used to test when the interruption is caused by a device other than VDP.

HTIMI (FD9FH)
Called by: The interrupt routine (KEYINT, 0038H) immediately after reading the VDP status register 0.
Purpose: Add interrupt handling routines. It can also be used to synchronize the graphical display, adding graphics during Vblank.

HCHPU (FDA4H)
Called by: Beginning of the CHPUT routine (00A2H).
Purpose: Connect other console devices besides the screen. Reg. A contains the character code when this hook is called.

HDSPC (FDA9H)
Called by: Beginning of the DSPSCR routine (with cursor).
Purpose: Connect other console devices besides the screen.

HERAC (FDAEH)
Called by: Beginning of ERASCR routine (deletes cursor).
Purpose: Connect other console devices besides the screen.

HDSPF (FDB3H)
Called by: Beginning of the DSPFNK routine (features function keys).
Purpose: Connect other console devices besides the screen.

HERAF (FDB8H)
Called by: Beginning of routine ERAFNK (clears function keys).
Purpose: Connect other console devices besides the screen.

HTOTE (FDBDH)
Called by: Beginning of TOTEXT routine (forces screen to text mode).
Purpose: Connect other console devices besides the screen.

HCHGE (FDC2H)
Called by: Beginning of the CHGET routine (takes a character).
Purpose: Connect other console devices besides the keyboard.

HINIP (FDC7H)
Called by: Beginning of the INIPAT routine (initialization of the character patterns).
Purpose: To use another character table.

HKEYC (FDCCH)
Called by: Beginning of the routine KEYCOD (keyboard character decoder).
Purpose: To change the keyboard configuration. When this hook is called, register A contains: (line number) × 8 + column number of the key pressed in the keyboard matrix.

HKEYA (FDD1H)
Called by: Beginning of MSXIO NMI (KEY EASY)
Purpose: To change the way a key is interpreted.

HNMI (FDD6H)
Called by: Beginning of the non-masking interrupt handler (NMI, 0066H).
Purpose: NMI is disabled on a standard MSX; therefore, this hook has no use.

HPINL (FDDBH)
Called by: Beginning of PINLIN routine (get a line)
Purpose: To use other input devices and / or methods, such as 80 columns of text or other input devices in addition to the keyboard.

HQINL (FDE0H)
Called by: Beginning QINLIN routine (get a line with "?").
Purpose: To use other input devices and / or methods, such as 80 columns of text or other input devices in addition to the keyboard.

HINLI (FDE5H)
Called by: Beginning of the INLIN routine.
Purpose: To use other input devices and / or methods, such as 80 columns of text or other input devices in addition to the keyboard.

HONGO (FDEAH)
Called by: Beginning of the handler of the ON GOTO and ON GOSUB commands.
Purpose: To divert access to these BASIC instructions.

HDSKO (FDEFH)
Called by: Beginning of BASIC command "DSKO$".
Purpose: Used by Disk-ROM to write a sector to the disk.

HSETS (FDF4H)
Called by: Beginning of the BASIC "SET" command.
Purpose: On MSX1, the instruction SET has no other effect than calling this hook and returning an error. On MSX2 or newer, the instructions SET SCREEN, SET ADJUST, SET TIME, etc call this hook to be treated.

HNAME (FDF9H)
Called by: Beginning of BASIC command "NAME".
Purpose: To connect disk devices.

HKILL (FDFEH)
Called by: Beginning of the BASIC "KILL" command.
Purpose: To connect disk devices.

HIPL (FE03H)
Called by: Beginning of BASIC command "IPL" (Initial Program Loading).
Purpose: Reserved. There is no known use for this instruction, but this hook can be used to add functions to the IPL instruction.

HCOPY (FE08H)
Called by: Beginning of the BASIC "COPY" command.
Purpose: To connect disk devices.

HCMD (FE0DH)
Called by: Beginning of BASIC command "CMD" (Expanded Commands).
Purpose: Reserved. There is no known use for this instruction, but this hook can be used to add functions to the CMD instruction.

HDSKF (FE12H)
Called by: Beginning of BASIC command "DSKF".
Purpose: To connect disk devices.

HDSKI (FE17H)
Called by: Beginning of BASIC command "DSKI$".
Purpose: To connect disk devices.

HATTR (FE1CH)
Called by: Beginning of the BASIC "ATTR$" command handler.
Purpose: To connect disk devices.

HLSET (FE21H)
Called by: Beginning of the BASIC "LSET" command handler.
Purpose: To connect disk devices.

HRSET (FE26H)
Called by: Beginning of the BASIC "RSET" command handler.
Purpose: To connect disk devices.

HFIEL (FE2BH)
Called by: Beginning of the FIELD command handler.
Purpose: To connect disk devices.

HMKI$ (FE30H)
Called by: Beginning of the MKI$ command handler.
Purpose: To connect disk devices.

HMKS$ (FE35H)
Called by: Beginning of the MKS$ command handler.
Purpose: To connect disk devices.

HMKD$ (FE3AH)
Called by: Beginning of the MKD$ command handler.
Purpose: To connect disk devices.

HCVI (FE3FH)
Called by: Beginning of the CVI command handler.
Purpose: To connect disk devices.

HCVS (FE44H)
Called by: Beginning of the CVS command handler.
Purpose: To connect disk devices.

HCVD (FE49H)
Called by: Beginning of the CVD command handler.
Purpose: To connect disk devices.

HGETP (FE4EH)
Called by: Find FCB (get file pointer).
Purpose: To connect disk devices.

HSETP (FE53H)
Called by: Find FCB (set file pointer).
Purpose: To connect disk devices.

HNOFO (FE58H)
Called by: OPEN command handler (OPEN without FOR).
Purpose: To connect disk devices.

HNULO (FE5DH)
Called by: OPEN command handler (open unused file).
Purpose: To connect disk devices.

HNTFL (FE62H)
Called by: Close I/O buffer 0.
Purpose: To connect disk devices.

HMERG (FE67H)
Called by: Beginning of the MERGE and LOAD commands handler.
Purpose: To connect disk devices.

HSAVE (FE6CH)
Called by: Beginning of the SAVE command handler.
Purpose: To connect disk devices.

HBINS (FE71H)
Called by: Beginning of the SAVE command handler (in binary).
Purpose: To connect disk devices.

HBINL (FE76H)
Called by: Beginning of the LOAD command handler (in binary).
Purpose: To connect disk devices.

HFILE (FE7BH)
Called by: Beginning of the FILES command handler.
Purpose: To connect disk devices.

HDGET (FE80H)
Called by: Beginning of the GET and PUT commands handler.
Purpose: To connect disk devices.

HFILO (FE85H)
Called by: Sequential output handler.
Purpose: To connect disk devices.

HINDS (FE8AH)
Called by: Sequential input handler.
Purpose: To connect disk devices.

HRSLF (FE8FH)
Called by: Handler for pre-selection of the drive.
Purpose: To connect disk devices.

HSAVD (FE94H)
Called by: Reserve current disk (LOC and LOF commands).
Purpose: To connect disk devices.

HLOC (FE99H)
Called by: Beginning of the LOC function handler.
Purpose: To connect disk devices.

HLOF (FE9EH)
Called by: Beginning of the LOF function handler.
Purpose: To connect disk devices.

HEOF (FEA3H)
Called by: Beginning of the EOF function handler.
Purpose: To connect disk devices.

HFPOS (FEA8H)
Called by: Beginning of the FPOS function handler.
Purpose: To connect disk devices.

HBAKU (FEADH)
Called by: Beginning of the LINEINPUT # instruction handler.
Purpose: To connect disk devices.

HPARD (FEB2H)
Called by: Beginning of the routine that analyzes the device name.
Purpose: To expand or add device names.

HNODE (FEB7H)
Called by: Beginning of the NODEVN routine, which is called when no name was found in the device name table.
Purpose: To assign the default device name to another device.

HPOSD (FEBCH)
Called by: Analyze device name (SPCDEV POSDSK).
Purpose: To connect disk devices.

HDEVN (FEC1H)
Called by: Process device name.
Purpose: To expand logical device name.

HGEND (FEC6H)
Called by: Beginning of the routine that assigns the device name.
Purpose: To expand logical device name.

HRUNC (FECBH)
Called by: Beginning of the routine that initializes the interpreter variables for the RUN and NEW commands.
Purpose: Allows to assign new functions to the commands.

HCLEA (FED0H)
Called by: Initialize interpreter variables for CLEAR command.
Purpose: Allows to assign new functions to the command or prevent accidental deletion of variables.

HLOPD (FED5H)
Called by: Initialize interpreter variables (general).
Purpose: To use other default values for variables.

HSTKE (FEDAH)
Called by: Beginning of the STKERR (stack error) routine, used by the BASIC CLEAR command.
Purpose: This hook is called after checking the executable ROMs in each slot when starting MSX, just before the system starts the BASIC or DOS environment. Therefore, it allows you to automatically re-run the ROM after installing the disks.

HISFL (FEDFH)
Called by: Beginning of the ISFLIO routine, which tests whether the file should be written or read.

HOUTD (FEE4H)
Called by: Beginning of the OUTDO routine, which sends a character to the screen or to the printer.

HCRDO (FEE9H)
Called by: Beginning of the routine that sends CR + LF to the OUTDO routine.
Purpose: Allows you to use a printer with automatic line feed, for example.

HDSKC (FEEEH)
Called by: Disk attribute entry.

HDOGR (FEF3H)
Called by: Beginning of the internal DOGRPH routine, used by the BASIC graphical instructions (LINE, CIRCLE, etc.)
Purpose: To change or expand the graphical instructions.

HPRGE (FEF8H)
Called by: End of the execution of a BASIC program.
Purpose: Add a routine to be executed after the BASIC program ends.

HERRP (FEFDH)
Called by: Beginning of the routine of presenting error messages.
Purpose: Add or change error messages.

HERRF (FF02H)
Called by: End of the error message routine.
Purpose: Add routine to be executed after the error message is presented.

HREAD (FF07H)
Called by: "Ok" from the main loop (interpreter ready).
Purpose: Add a routine to be executed after the prompt is presented ("Ok").

HMAIN (FF0CH)
Called by: Beginning of the interpreter's BASIC text execution main loop.
Purpose: Add a routine to be executed whenever the BASIC interpreter is accessed.

HDIRD (FF11H)
Called by: Beginning of direct command execution (direct declaration).
Purpose: Add routine or prevent executions.

HFINI (FF16H)
Called by: Beginning of the FININT routine, which starts the interpretation of a BASIC commands.
Purpose: To change the processing of BASIC instructions.

HFINE (FF1BH)
Called by: End of the FININT routine, which initiates the interpretation of a BASIC statement.

HCRUN (FF20H)
Called by: Beginning of the CRUNCH routine (42B9H), which converts a BASIC text from ASCII form to tokenized form.

HCRUS (FF25H)
Called by: Beginning of the CRUSH routine (4353H), which looks for a reserved word in the alphabetical list of the ROM.

HISRE (FF2AH)
Called by: Beginning of the ISRESV routine (437CH), when a reserved word is found by the CRUSH routine.

HNTFN (FF2FH)
Called by: Beginning of the NTFN2 routine (43A4H), when a reserved word is followed by a line number.

HNOTR (FF34H)
Called by: Beginning of the NOTRSV routine (44EBH), when the sequence of characters examined by the CRUNCH routine is not a reserved word.

HSNGF (FF39H)
Called by: Beginning of the FOR command handler.

HNEWS (FF3EH)
Called by: Beginning of the interpreter's NEWSTT routine (4601H), which executes an tokenized BASIC text.

HGONE (FF43H)
Called by: Beginning of the GONE2 routine, used by the jump instructions (GOTO, THEN, etc.).

HCHRG (FF48H)
Called by: Beginning of the CHRGET routine (character entry via the keyboard).
Purpose: Use another keyboard.

HRETU (FF4DH)
Called by: Beginning of the RETURN command handler.

HPTRF (FF52H)
Called by: Beginning of the PRINT command handler.

HCOMP (FF57H)
Called by: Beginning of the internal COMPRT routine (4A94H), used by the PRINT handler.

HFINP (FF5CH)
Called by: Beginning of the routine that clears PRTFLG and PRTFIL to end the PRINT command.
Purpose: Add a routine to be executed after the PRINT command.

HTRMN (FF61H)
Called by: Beginning of the error handler of the READ and INPUT commands.
Purpose: Error processing.

HFRME (FF66H)
Called by: Routine FRMEVL (4C64H) – Expression Evaluator.
Purpose: Allows you to add new mathematical functions.
Input: HL = pointer to BASIC text
Output: HL = pointer to the expression found
VALTYP (F663H) = value type of expression
DAC (F7F6H) = value found

HNTPL (FF6BH)
Called by: Routine FRMEVL (4CA6H) – Expression Evaluator.
Purpose: Allows you to add new mathematical functions.

HEVAL (FF70H)
Called by: Factor Evaluator (4DD9H)
Purpose: Allows you to add new mathematical functions.

HOKNO (FF75H)
Called by: Beginning of the BASIC interpreter's transcendental function routine (hook removed on MSX turbo R. It was replaced by HMDIN).
Purpose: Allows you to add new mathematical functions.

HMDIN (FF75H)
Called by: Beginning of the MIDI interface interruption handling routine (MSX turbo R with internal MIDI only).
Purpose: Add or change functionality of the MIDI interface.

HFING (FF7AH)
Called by: Factor evaluator.

HISMI (FF7FH)
Called by: Beginning of the MID$ command handler.

HWIDT (FF84H)
Called by: Beginning of the WIDTH command handler.

HLIST (FF89H)
Called by: Beginning of the LIST command handler.

HBUFL (FF8EH)
Called by: De-symbolize for LIST command (532DH).

HFRQI (FF93H)
Called by: Convert to integer (543FH). Hook removed on MSX turbo R. Replaced by HMDTM.

HMDTM (FF93H)
Called by: Beginning of the MIDI interface timer manipulation routine (MSX turbo R with internal MIDI only).
Purpose: Add or change functionality of the MIDI interface.

HSCNE (FF98H)
Called by: Beginning of the BASIC interpreter routine SCNEX2 (5514H) (converting a line number to a memory address and vice versa)

HFRET (FF9DH)
Called by: Searches for a free place to store the next descriptor of an alphanumeric variable (string).

HPTRG (FFA2H)
Called by: Beginning of the PTRGET routine (5EA9H) of the BASIC interpreter, which obtains the pointer of a variable.
Purpose: Use another default value for the variables.

HPHYD (FFA7H)
Called by: Beginning of routine PHYDIO (physical disk input-output).
Purpose: To connect disk devices.

HFORM (FFACH)
Called by: Beginning of the FORMAT (format disk) routine.
Purpose: To connect disk devices.

HERRO (FFB1H)
Called by: Beginning of the error handler.
Purpose: Error handling by application programs.

HLPTO (FFB6H)
Called by: Beginning of the LPTOUT routine (00A5H).
Purpose: Use other printer models.

HLPTS (FFBBH)
Called by: Beginning of the LPTSTT routine (00A5H).
Purpose: Use other printer models.

HSCRE (FFC0H)
Called by: Beginning of the SCREEN command handler.
Purpose: To expand the SCREEN command.

HPLAY (FFC5H)
Called by: Beginning of the PLAY command handler.
Purpose: To expand the PLAY command.

9 – BIOS ROUTINES

This appendix provides a description of the BIOS routines available to the user.

There are several types of BIOS routines, the ones in the Main-ROM, the ones in the Sub-ROM, the Math-Pack routines and the extension routines accessed by EXTBIO on the workarea, in addition to several others made available by cartridges of expansion and of the BASIC interpreter routines.

The notation for routines is as follows:

LABEL (Routine address / location)
Function: describes the function of the routine.
Input: describes the parameters for calling the routine.
Output: describes the parameters for returning the routine.
Registers: lists the registers modified by the routine.

9.1 – Main-ROM ROUTINES

9.1.1 – RST Routines

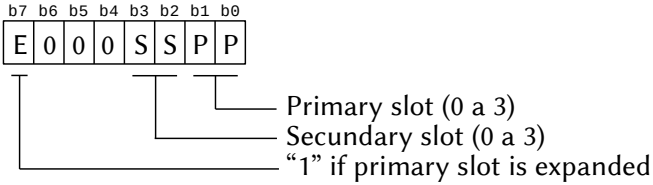
CHKRAM (0000H / Main)
Function: Tests RAM and initializes system variables. A call to this routine will cause a software reset.
Input: None.
Output: None.
Registers: All

SYNCHR (0008H / Main)
Function: Tests if the character pointed by (HL) is the one specified. If not, it generates "Syntax error"; otherwise it calls CHRGTR (0010H).
Input: The character to be tested must be in (HL) and the character for comparison after the RST instruction (online parameter), as shown in the example below:

```
LD HL, CARACT
RST 008H
DEFB 'A'
 |
CARACT: DEFB 'B'
```

Output: HL is incremented by 1 and A receives (HL). When the tested character is numeric, the CY flag is set; the end of declaration (00H or 3AH) sets the Z flag.
Registers: AF, HL.

RDSLT (000CH / Main)
Function: Reads a memory byte in the slot specified in A. Interrupts are disabled during reading.
Input: A –



HL – memory address to be read.
Output: A – contains the value of the read byte.
Registers: AF, BC, DE.

CHRGTR (0010H / Main)
Function: Get a character (token) from the BASIC text.
Input: HL – address of the character to be read.
Output: HL is incremented by 1 and A receives (HL). When the character is numeric, the CY flag is set; the end of the declaration (00H or 3AH) sets the Z flag.
Registers: AF, HL.

WRSLT (0014H / Main)
Function: Writes a memory byte in the slot specified in A. Interrupts are disabled during writing.
Input: A – slot indicator (same as RDSLT – 000CH).
HL – address for writing the byte.
E – byte to be written.
Output: None
Registers: AF, BC, D.

OUTDO (0018H / Main)
Function: Sends a byte to the current device.
Input: A – Byte to be sent. If PRTFLG (F416H) is different of 0, the byte will sent to the printer; if PTRFIL (F864H) is

different of 0, the byte will sent to the file specified by PTRFIL.

Output: None.
Registers: None.

CALSLT (001CH)
Function: Calls a routine in any slot (called an inter-slot).
Input: IY – the slot ID must be specified in the highest 8 bits in the same format as RDSLT (000CH).
IX – address of the routine to be called.
Output: It depends on the called routine.
Registers: It depends on the routine called.

DCOMPR (0020H)
Function: Compare HL with DE.
Input: HL, DE.
Output: Set the Z flag if HL = DE; set the flag CY if HL < DE.
Registers: AF.

ENASLT (0024H)
Function: Enables a page in any slot. Only pages 1 and 2 can be enabled by this routine; 0 and 3 do not. Interruptions are deactivated during enabling.
Input: A – Slot indicator (same as RDSLT – 000CH).
Output: None.
Registers: All.

GETYPR (0028H / Main)
Function: Gets the type of operand contained in DAC.
Input: None
Output: Flags CY, S, Z and P / V, as shown in the table below:

Integer:	C=1	S=1*	Z=0	P/V=1
Simple precision:	C=1	S=0	Z=0	P/V=0*
Double precision:	C=0*	S=0	Z=0	P/V=1
String:	C=1	S=0	Z=1*	P/V=1

Note: The types can be recognized by using only by the flags marked with "*".
Registers: AF.

CALLF (0030H / Main)
Function: Calls a routine in any slot using inline parameters. Very useful for calling routines through system hooks. The call sequence is as follows:

```
RST 030H ; calls CALLF
DEFB n   ; n is slot ID (same as RDSLT)
DEFW nn  ; nn is the address to be called
RET      ; return to the system
```

Input: By the method described.
Output: It depends on the called routine.
Registers: Depends on the called routine (plus AF).

KEYINT (0038H / Main)
Function: Performs the routine of interrupting and scanning the keyboard.
Input: None.
Output: None.
Registers: None.

9.1.2 – Routines for I/O initialization

HOME (0000H / Main)
Function: Initializes the input and output devices.
Input: None.
Output: None.
Registers: All.

INIFNK (003EH / Main)
Function: Initializes the contents of the function keys.
Input: None.
Output: None.
Registers: All.

9.1.3 – Routines for accessing the VDP

DISSCR (0041H / Main)
Function: Disables the screen presentation.
Input: None.
Output: None.
Registers: AF, BC.

ENASCR (0044H / Main)
Function: Enables the screen presentation.
Input: None.
Output: None.
Registers: AF, BC.

WRTVDP (0047H / Main)
Function: Writes a byte of data to a VDP register.
Input: C – register that will receive the data. It can vary from 0 to 7 for MSX1, from 0 to 23/32 to 46 for MSX2 and from 0 to 23/25 to 27/32 to 46 for MSX2+ or higher.
B – data byte
Output: None.
Registers: AF, BC.

RDVRM (004AH / Main)
Function: Read a VRAM byte. This routine reads only the lowest 14 address bits (16K for MSX1's TMS9918). To access the entire VRAM it is necessary to use the routine NRDVRM (0174H).
Input: HL – VRAM address to be read.
Output: A – byte read.
Registers: AF.

WRTVRM (004DH / Main)
Function: Writes a VRAM byte. This routine writes only the lowest 14 address bits (16K for MSX1's TMS9918). To access the entire VRAM, it is necessary to use the routine NWRVRM (0177H).
Input: HL – VRAM address to be written.
A – Byte to be written.
Output: None.
Registers: AF.

SETRD (0050H / Main)
Function: Prepare the VRAM for sequential reading using the VDP address auto-increment function. It is a faster means of reading than using a loop with the RDVRM routine (004AH). This routine accesses only the lowest 14 address bits (16K for MSX1's TMS9918). To access the entire VRAM, it is necessary to use the NSETRD routine (016EH).

Input: HL – Address at VRAM to start reading
Output: None.
Registers: AF.

SETWRT (0053H / Main)
Function: Prepare the VRAM for sequential writing using the VDP address auto-increment function. The characteristics are the same as for SETRD (0050H). To access the entire VRAM it is necessary to use the routine NSTWRT (0171H).
Input: HL – VRAM address to start reading.
Output: None.
Registers: AF.

FILVRM (0056H / Main)
Function: Fills an area of the VRAM with a single byte of data. This routine accesses only the lowest 14 address bits (16K for MSX1's TMS9918). To access the entire VRAM, it is necessary to use the BIGFIL routine (016BH).
Input: HL – VRAM address to start writing.
BC – Number of bytes to be written.
A – Byte to be written.
Output: None.
Registers: AF, BC.

LDIRMV (0059H / Main)
Function: Copies a block of data from VRAM to RAM. All 16 address bits are valid.
Input: HL – Source address at VRAM.
DE – Destination address in RAM.
BC – Block size (length).
Output: None.
Registers: All.

LDIRVM (005CH / Main)
Function: Copies a block of data from RAM to VRAM.
Input: HL – Source address in RAM.
DE – Destination address at VRAM.
BC – Block size (length).

Note: All 16 address bits are valid.
Output: None.
Registers: All.

CHGMOD (005FH / Main)
Function: Switches the screen modes. This routine does not initialize the color palette. For this, it is necessary to use the routine CHGMDP (01B5H / Sub-ROM).
Input: A – 0 to 3 for MSX1, 0 to 8 for MSX2 or 0 to 12 for MSX2+ or higher (Note: Mode 9 is only valid for Korean machines).
Output: None.
Registers: All.

CHGCLR (0062H / Main)
Function: Change the colors of the screen.
Input: FORCLR (F3E9H) – Front color
BAKCLR (F3EAH) – Background color
BDRCLR (F3EBH) – Border color
Output: None.
Registers: All.

NMI (0066H / Main)
Function: Executes the NMI (Non-Maskable Interrupt) routine. On a standard MSX machine, it just makes a call to the HNMI hook (FDD6H) and returns without any processing.
Input: None.
Output: None.
Registers: None.

CLRSPR (0069H / Main)
Function: Initializes all sprites. The sprite pattern table is cleared (filled with zeros), the sprite numbers are initialized with the series 0 ~ 31 and the color of the sprites is equal to the background color. The vertical location of the sprites is set to 209 (for Screens 0 to 3) or 217 (for Screens 4 to 8 / 10 to 12).
Input: SCRMOD (FCAFH) – Screen mode.
Output: None.
Registers: All.

INITXT (006CH / Main)
Function: Initializes the screen in text mode (Screen 0). The color palette is not initialized. To initialize it, it is necessary to call the routine INIPLT (0141H / Sub-ROM).
Input: TXTNAM (F3B3H) – Name table address
TXTCGP (F3B7H) – Pattern table address
LINL40 (F3AEH) – Number of characters per line
Output: None.
Registers: All.

INIT32 (006FH / Main)
Function: Initializes the screen in graphical mode 1 (Screen 1). The color palette is not initialized. To initialize it, it is necessary to call the routine INIPLT (0141H / Sub-ROM).
Input: T32NAM (F3BDH) – Address of the character name table.
T32COL (F3BFH) – Address of the character color table.
T32CGP (F3C1H) – Address of the character pattern table.
T32ATR (F3C3H) – Address of the sprites attribute table.
T32PAT (F3C5H) – Address of the sprites standards table.
Output: None.
Registers: All.

INIGRP (0072H / Main)
Function: Initializes the screen in the high resolution graphic mode of MSX1 (Screen 2). The color palette is not initialized. To initialize it, it is necessary to call the routine INIPLT. (0141H / Sub-ROM).
Input: GRPNAM (F3C7H) – Address of the pattern name table.
GRPCOL (F3C9H) – Address of the color table.
GRPCGP (F3CBH) – Address of the pattern generator table.
GRPATR (F3CDH) – Address of the sprites attribute table.
GRPPAT (F3CFH) – Address of the sprite standards table.
Output: None.
Registers: All.

INIMLT (0075H / Main)
Function: Initializes the screen in the MSX1 multicolor mode (Screen 3). The color palette is not initialized. To initialize it, it is necessary to call the routine INIPLT (0141H / Sub-ROM).

Input: MLTNAM (F3D1H) – Address of the pattern name table.
MLTCOL (F3D3H) – Address of the color table.
MLTCGP (F3D5H) – Adr of the pattern generator table.
MLTATR (F3D7H) – Address of the sprites attribute table.
MLTPAT (F3D9H) – Address of the sprites standards table.
Output: None.
Registers: All.

SETTXT (0078H / Main)
Function: Puts only the VDP in text mode (Screen 0).
Input: Same as INITXT (006CH).
Output: None.
Registers: All.

SETT32 (007BH / Main)
Function: Puts only the VDP in graphical mode 1 (Screen 1).
Input: Same as INIT32 (006FH).
Output: None.
Registers: All.

SETGRP (007EH / Main)
Function: Puts only the VDP in graphical mode 2 (Screen 2).
Input: Same as INIGRP (0072H).
Output: None.
Registers: All.

SETMLT (0081H / Main)
Function: Puts only the VDP in multicolour mode (Screen 3).
Input: Same as INIMLT (0075H).
Output: None.
Registers: All.

CALPAT (0084H / Main)
Function: Returns the address of the sprite pattern generator table.
Input: A – Sprite number.
Output: HL – Address at VRAM.
Registers: AF, DE, HL.

CALATR (0087H / Main)
Function: Returns the address of a sprite's attribute table.
Input: A – Sprite number.
Output: HL – Address at VRAM.
Registers: AF, DE, HL.

GSPSIZ (008AH / Main)
Function: Returns the current size of the sprites.
Input: None.
Output: A – Size of the sprite in bytes. The CY flag is set if the size is 16 x 16 and reset otherwise.
Registers: AF.

GRPPRT (008DH / Main)
Function: Displays a character on a graphic screen.
Input: A – Character ASCII code. When the screen is 5 to 8 or 10 to 12, it is necessary to specify the logical operation code in LOGOPR (FB02H).
Output: None.
Registers: None.

9.1.4 – Routines for access to PSG

GICINI (0090H / Main)
Function: Initializes the PSG and sets the initial values for the PLAY command.
Input: None.
Output: None.
Registers: All.

WRTPSG (0093H / Main)
Function: Writes a byte of data to a PSG register.
Input: A – PSG registrar number.
E – Byte of data to be written.
Output: None.
Registers: None.

RDPSG (0096H / Main)
Function: Reads the contents of a PSG register.
Input: A – PSG registrar number.
Output: A – Byte read.
Registers: None.

STRTMS (0099H / Main)
Function: Tests whether the PLAY command is being executed. If not, start execution.
Input: None.
Output: None.
Registers: All.

9.1.5 – Routines for accessing keyboard, screen and printer

CHSNS (009CH / Main)
Function: Checks the keyboard buffer.
Input: None.
Output: If the Z flag is set, the buffer is empty; otherwise, the Z flag will be reset.
Registers: AF.

CHGET (009FH / Main)
Function: Input of a character by the keyboard with waiting.
Input: None.
Output: A – Character ASCII code.
Registers: AF.

CHPUT (00A2H / Main)
Function: Displays a character on the text screen.
Input: A – ASCII code of the character to be displayed.
Output: None.
Registers: None.

LPTOUT (00A5H / Main)
Function: Send a character to the printer.
Input: A – ASCII code of the character to be sent.
Output: If it fails, CY returns set.
Registers: F.

LPTSTT (00A8H / Main)
Function: Tests the status of the printer.
Input: None.
Output: A = 255 (and Z flag = 0) ⟶ printer ready.
A = 0 (and Z flag = 1) ⟶ printer is not ready.
Registers: AF.

CNVCHR (00A8H / Main)
Function: Tests the graphic header and converts if necessary.
Input: A – ASCII code of the character.
Output: CY = 0 – There is no graphic header.
CY = 1 and Z = 1 – The converted code is placed in A.
CY = 1 and Z = 0 – The unconverted code returns to A.
Registers: AF.

PINLIN (00AEH / Main)
Function: Collect a line of text and store it in a buffer until the RETURN or STOP key is pressed.
Input: None.
Output: HL – initial address of the buffer minus 1.
CY – set if the STOP key was pressed.
Registers: All.

INLIN (00B1H / Main)
Function: Same as PINLIN(00AEH), except that AUTFLG(F6AAH) is set.
Input: None.
Output: HL – initial address of the buffer minus 1.
CY – set if the STOP key was pressed.
Registers: All.

QINLIN (00B4H / Main)
Function: Executes INLIN (00B1H) presenting "?" and a space.
Input: None.
Output: HL – initial address of the buffer minus 1.
CY – set if the STOP key was pressed.
Registers: All.

BREAKX (00B7H / Main)
Function: Tests whether CTRL+STOP are pressed together. During verification, interruptions are disabled.
Input: None.
Output: CY – Set if CTRL+STOP are pressed.
Registers: AF.

BEEP (00C0H / Main)
Function: Generates a beep.
Input: None.
Output: None.
Registers: All.

CLS (00C3H / Main)
Function: Clears the screen.
Input: The Z flag must be set.
Output: None.
Registers: AF, BC, DE.

POSIT (00C6H / Main)
Function: Moves the cursor to a specific coordinate.
Input: H – X coordinate (horizontal)
L – Y coordinate (vertical)
Output: None.
Registers: AF.

FNKSB (00C9H / Main)
Function: Tests whether the commands associated with the function keys are being displayed on the screen by checking the FNKFLG (FBCEH) flag and inverts the display status (if the flag is on, off and if it is off, on).
Input: FNKFLG (FBCEH).
Output: None.
Registers: All.

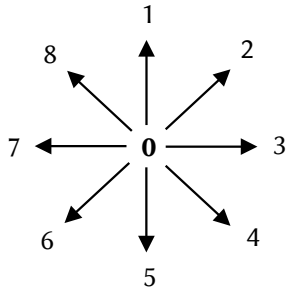
ERAFNK (00CCH / Main)
Function: Turn off the display of the function keys.
Input: None.
Output: None.
Registers: All.

DSPFNK (00CFH / Main)
Function: Turns on the display of the function keys.
Input: None.
Output: None.
Registers: All.

TOTEXT (00D2H / Main)
Function: Forces the screen to text mode (Screen 0 or 1).
Input: None.
Output: None.
Registers: All.

9.1.6 – I/O access routines for games

GTSTCK (00D5H / Main)
Function: Returns the state of the joystick or the cursor keys.
Input: A – 0 = Cursor keys.
1 = joystick on port 1.
2 = joystick on port 2.
Output: A – direction of the joystick or function keys as shown in the illustration below.



Registers: All.

GTTRIG (00D8H / Main)
Function: Returns the state of the mouse, joystick or keyboard bar buttons.
Input: A – 0 = Space bar.
1 = joystick on port 1, button A.
2 = joystick on port 2, button A.
3 = joystick on port 1, button B.
4 = joystick on port 2, button B.

Output: A – 0 = tested button is not pressed.
255 = tested button is pressed.
Registers: AF, BC.

GTPAD (00DBH / Main)
Function Returns the state of a touch pad, trackball or mouse connected to one of the joystick connectors.
Input: A – 0 – Check touch pad on port 1 (255 if connected)
1 – Returns the X coordinate (horizontal).
2 – Returns the Y (vertical) coordinate.
3 – Returns the key state (255 if pressed).
4 – Check touch pad on port 2 (255 if connected).
5 – Returns the X (horizontal) coordinate.
6 – Returns the Y (vertical) coordinate.
7 – Returns the key state (255 if pressed).
8 – Check lightpen (255 if connected or touching pad).
9 – Returns the X (horizontal) coordinate.
10 – returns the Y (vertical) coordinate.
11 – returns the key state (255 if pressed).
12 – check mouse on port 1 (255 if connected).
13 – returns X coordinate offset (horizontal).
14 – returns Y coordinate offset (vertical).
15 – always 0.
16 – check mouse on port 2 (255 if connected).
17 – returns X coordinate offset (horizontal).
18 – returns Y coordinate offset (vertical).
19 – always 0.
20 – checks 2nd lightpen (255 if connected or touching the pad).
21 – returns the X coordinate (horizontal).
22 – returns the Y (vertical) coordinate.
23 – returns the key state (255 if pressed).
Output: A – state or value, as described above.
Registers: All.
Note: For function codes 8 to 23, call NEWPAD (01ADH) in SubROM. For the MSX turbo R, the pen functions (8 to 11) have been eliminated.

GTPDL (00DEH / Main)
Function: Returns the values of paddles connected to the joystick connectors.
Input: A – paddle identification (1 to 12).
1, 3, 5, 7, 9, 11 – Paddles connected to port 1.
2, 4, 6, 8, 10, 12 – Paddles connected to port 2.
Output: A – value read (0 to 255).
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

9.1.7 – I/O access routines for cassette register

TAPION (00E1H / Main)
Function: Read the tape header after starting the cassette motor.
Input: None.
Output: If it fails, the CY flag returns set.
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

TAPIN (00E4H / Main)
Function: Read data from the tape.
Input: None.
Output: A – byte read.
CY – set if the reading fails.
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

TAPIOF (00E7H / Main)
Function: For reading the tape.
Input: None.
Output: None.
Registers: None.
Note: This routine was eliminated in the MSX turbo R.

TAPOON (00EAH / Main)
Function: Writes the header on the tape after starting the cas motor.
Input: A – 0 = Short header; another value = long header.
Output: If it fails, the CY flag returns set.
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

TAPOUT (00EDH / Main)
Function: Writes data to the tape.
Input: A – Byte to be written.
Output: If it fails, the CY flag returns set.
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

TAPOOF (00F0H / Main)
Function: For writing on the tape.
Input: None.
Output: If it fails, the CY flag returns set.
Registers: All.
Note: This routine was eliminated in the MSX turbo R.

STMOTR (00F3H / Main)
Function: Turns the cassette motor on or off.
Input: A – 0 = power on the motor
1 = power off the motor
255 = Inverts the state of the motor
Output: None.
Registers: AF.
Note: This routine was eliminated in the MSX turbo R.

9.1.8 – Routines for the PSG queue

LFTQ (00F6H / Main)
Function: Returns the number of free bytes in a PSG musical queue.
Input: A – queue number (0, 1 or 2).
Output: HL – free space left in the queue.
Registers: AF, BC, HL.

PUTQ (00F9H / Main)
Function: Place a byte in one of the PSG's musical queues.
Input: A – queue number (0, 1 or 2).
E – data byte.
Output: Flag Z set if the queue is full.
Registers: AF, BC, HL.

GETVCP (0150H / Main)
Function: Returns the address of byte 2 in the PSG's voice buffer.
Input: A – Voice number (0, 1 or 2)
Output: HL – address in the voice buffer.
Registers: AF, HL.

GETVC2 (0153H / Main)
Function: Returns the address of any bytes in the PSG's voice buffer.
Input: VOICEN (FB38H) – Voice number (0, 1 or 2).
L – Byte number (0 to 36).
Output: HL – Address in the voice buffer.
Registers: AF, HL.

9.1.9 – Routines for MSX1 graphics screens

RIGHTC (00FCH / Main)
Function: Shifts the current pixel one position to the right.
Input: None.
Output: None.
Registers: AF.

LEFTC (00FFH / Main)
Function: Shifts the current pixel one position to the left.
Input: None.
Output: None.
Registers: AF.

UPC (0102H / Main)
Function: Shifts the current pixel one position up.
Input: None.
Output: None.
Registers: AF.

TUPC (0105H / Main)
Function: Tests the position of the current pixel and, if possible, moves it up one position.
Input: None.
Output: CY = 1 if the pixel could not be moved because it exceeds the upper limit of the screen.
Registers: AF.

DOWNC (0108H / Main)
Function: Shifts the current pixel down one position.
Input: None.
Output: None.
Registers: AF.

TDOWNC (010BH / Main)
Function: Tests the position of the current pixel and, if possible, moves it down one position.
Input: None.
Output: CY = 1 if the pixel could not be moved because it exceeds the lower limit of the screen.
Registers: AF.

SCALXY (010EH / Main)
Function: Limits the pixel coords to the visible area of the screen.
Input: BC – X coordinate (horizontal).
DE – Y coordinate (vertical).
Output: BC – X coordinate limited to the border.
DE – Y coordinate limited to the border.
CY = 1 if the coordinates are limited.
Registers: AF.

MAPXYC (0111H / Main)
Function: Converts a pair of graphic coordinates to the physical address of the current pixel (places the "cursor" on the coord).
Input: BC – X coordinate (horizontal).
DE – Y coordinate (vertical).
Output: None.
Registers: AF, D, HL.

FETCHC (0114H / Main)
Function: Returns the physical address of the current pixel.
Input: None.
Output: A ← content of CMASK (F92CH).
HL ← content of CLOC (F92AH).
Registers: A, HL.

STOREC (0117H / Main)
Function: Establishes the physical address of the current pixel.
Input: A is copied to CMASK (F92CH).
HL is copied to CLOC (F92AH).
Output: None.
Registers: None.

SETATR (011AH / Main)
Function: Establishes the color for the SETC (0120H) and NSETCX (0123H) routines.
Input: A – Color code (0 to 15).
Output: CY – Set if the color code is invalid.
Registers: F.

READC (011DH / Main)
Function: Returns the color code of the current pixel.
Input: None.
Output: A – Color code of the current pixel (0 to 15).
Registers: AF, EI.

SETC (0120H / Main)
Function: Establishes the color of the current pixel.
Input: ATRBYT (F3F2H) – Color code (0 to 15), established by SETATR (011AH).
Output: None.
Registers: AF, EI.

NSETCX (0123H / Main)
Function: Sets the color of multiple horizontal pixels starting from the current pixel, to the right.
Input: ATRBYT (F3F2H) – Color code (0 to 15), established by SETATR (011AH).
HL – Number of pixels to color.
Output: None.
Registers: AF, EI.

GTASPC (0126H / Main)
Function: Returns the aspect ratios of the CIRCLE statement.
Input: None.
Output: DE – Contents of ASPCT1 (F40BH).
HL – Contents of ASPCT2 (F40DH).
Registers: DE, HL.

PNTINI (0129H / Main)
Function: Establishes the outline color for the PAINT instruction.
Input: A – outline color code (0 to 15).
Output: CY – 1 if the color code is invalid.
Registers: AF.

SCANR (012CH / Main)
Function: Used by the PAINT instruction handler to scan an area, from left to right, starting from the current pixel until a color code equal to BDRATR (FCB2H) is found or the edge of the screen is reached.
Input: B – 0 = Does not fill the area covered.
255 = Fills the area covered.
DE – number of hops (pixels of the same color ignored).
Output: HL – number of pixels covered.
DE – number of hops remaining.
Registers: AF, BC, DE, HL, EI.

SCANL (012FH / Main)
Function: Same as SCANR (012CH), except that the route will be from right to left and the area will always be filled.
Input: None.
Output: HL – number of pixels covered.
Registers: AF, BC, DE, HL, EI.

9.1.10 – Miscellaneous

CHGCAP (0132H / Main)
Function: Changes the LED status of Caps Lock.
Input: A = 0 turns off the LED; another value, turn on the LED.
Output: None.
Registers: AF.

CHGSND (0135H / Main)
Function: Changes the state of the sound-generating 1-bit port.
Input: A = 0 turns the bit off, another value turns the bit on.
Output: None.
Registers: AF.

RSLREG (0138H / Main)
Function: Reads the contents of the primary slot register.
Input: None.
Output: A – Value read.
Registers: A.

WSLREG (013BH / Main)
Function: Writes to the primary slot register.
Input: A – Value to be written.
Output: None.
Registers: None.

RDVDP (013EH / Main)
Function: Read the VDP status register.
Input: None.
Output: A – Value read.
Registers: A.

SNSMAT (0141H / Main)
Function: Reads the value of a line from the keyboard matrix.
Input: A – Line to be read.
Output: A – Value read (the bit corresponding to a key pressed is 0).
Registers: AF, C.

ISFLIO (014AH / Main)
Function: Tests whether a device I/O operation is being performed.
Input: None.
Output: A = 0 if the device is active (I/O operation is being performed); another value the device is inactive.
Registers: AF.

OUTDLP (014DH / Main)
Function: Formatted output for the printer. It differs from LPTOUT in the following points:

- If the character sent is a TAB (09H) spaces will be sent until reaching a multiple of 8;
- For non-MSX printers, hiraganas are converted to katakanas and graphic characters are converted to 1-byte characters;
- If there is a failure, an I/O error will occur.

Input: A – Character to be sent.
Output: None.
Registers: F.

KILBUF (0156H / Main)
Function: Clears the keyboard buffer.
Input: None.
Output: None.
Registers: HL.

CALBAS (0159H / Main)
Function: Performs an inter-slot call to any BASIC interpreter routine.
Input: IX – Address to be called.
Output: It depends on the routine called.
Registers: It depends on the routine called.

9.1.11 – Routines for accessing the disk system

PHYDIO (0144H / Main)
Function: Read or write one or more sectors on the specified drive.
Input: CY – 0 = Reading.
1 = writing.
A – drive number (0 = A:, 1 = B:, etc).
B – number of sectors to read or write.
C – Disk formatting ID:
F0H – 63 sectors per track (for HD's)
F8H – 80 tracks, 9 sectors per track, single face.
F9H – 80 tracks, 9 sectors per track, double sided.
FAH – 80 tracks, 8 sectors per track, single face.
FBH – 80 tracks, 8 sectors per track, double sided.
FCH – 40 tracks, 9 sectors per track, single face.
FDH – 40 tracks, 9 sectors per track, double sided.
DE – Number of the first sector to be read or written.
HL – RAM address from which the sectors to be read from the disk will be written or the sectors to be written to the disk will be removed.
Output: CY – set if there was a reading or writing error.

A – error code if CY = 1:
0 – Write-protected.
2 – Not ready.
4 – Data error.
6 – Seek error.
8 – Sector not found.
10 – Writing error.
12 – Invalid parameters.
14 – Insufficient memory.
16 – Undefined error.
B – Number of sectors actually read or written.
Registers: All.
Note: In some HD interfaces, when bit 7 of register C is set, a 23-bit addressing scheme will be used and bits 0-6 of register C must contain bits 23-16 of the number of the sector.

FORMAT (0147H / Main)
Function: Format a floppy disk. When called, a series of questions will be presented that must be answered to start the formatting. There is no standard for these questions; they can be different for each drive interface.
Input: None.
Output: None.
Registers: None.

9.1.12 – Routines added for MSX2

SUBROM (015CH / Main)
Function: Performs an inter-slot call to SubROM.
Input: IX – Address to be called (at the same time put IX on the stack).
Output: It depends on the called routine.
Registers: IY, AF', BC', DE', HL', and the registers modified by the called routine.

EXTROM (015FH / Main)
Function: Performs an inter-slot call to SubROM.
Input: IX – Address to be called.

Output: It depends on the called routine.
Registers: IY, AF', BC', DE', HL', and the registers modified by the called routine.

CHKSLZ (0162H / Main)
Function: Searches for slots for the SubROM.
Input: None.
Output: None.
Registers: All.

CHKNEW (0165H / Main)
Function: Tests the screen mode.
Input: None.
Output: CY = 1 if screen is 5, 6, 7 or 8.
Registers: AF.

EOL (0168H / Main)
Function: Erase until the end of the line.
Input: H – X coordinate of the cursor.
L – Y coordinate of the cursor.
Output: None.
Registers: All.

BIGFIL (016BH / Main)
Function: Fills an area of RAM with a single byte of data. The Screens 0 to 3 are not tested and filling can exceed the 16K limit of these screens. See FILVRM (0056H) on Main ROM.
Input: L – VRAM address to start writing.
BC – Number of bytes to be written.
A – Byte to be written.
Output: None.
Registers: AF, BC.

NSETRD (016EH / Main)
Function: Prepares VRAM for sequential reading using the VDP address auto-increment function.
Input: HL – VRAM address from which the data will be read. All bits are valid.
Output: None.
Registers: AF.

NSTWRT (0171H / Main)
Function: Prepares VRAM for sequential writing using the VDP address auto-increment function.
Input: HL – VRAM address from which the data will be written. All bits are valid.
Output: None.
Registers: AF.

NRDVRM (0174H / Main)
Function: Read the content of one byte of the VRAM.
Input: HL – VRAM address to be read.
Output: A – byte read.
Registers: AF.

NWRVRM (0177H / Main)
Function: Writes one byte of data to VRAM.
Input: HL – VRAM address to be written.
A – byte to be written.
Output: None.
Registers: AF.

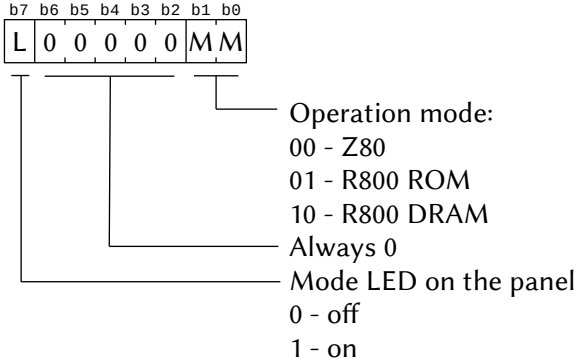
9.1.13 – Routines added for MSX2+

RDRES (017AH / Main)
Function: Returns the reset status.
Input: None.
Output: A – b7 = 0 indicates total reset (by hardware)
b7 = 1 indicates partial reset (by software)
Registers: A.
Note: In the total reset (by hardware) the RAM content is cleared and the MSX logo appears at startup. In the partial reset (by software) the RAM content is not erased (only the desktop is initialized) and the MSX logo does not appear at startup.

WRRES (017DH / Main)
Function: Modifies the reset status.
Input: A – b7 = 0 for total reset (by hardware)
b7 = 1 for partial reset (by software)
Output: None.
Registers: None.

9.1.14 – Routines added for the MSX turbo R

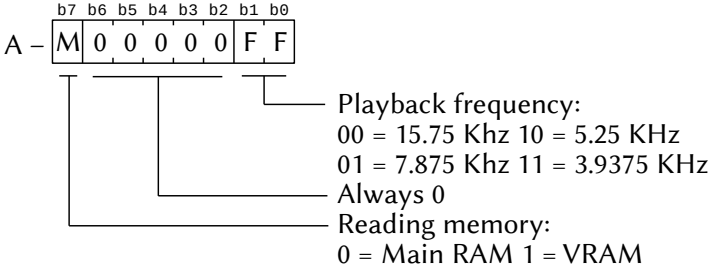
CHGCPU (0180H / Main)
Function: Change the microprocessor (operating mode).
Input: A –



Output: None.
Registers: AF.

GETCPU (0183H / Main)
Function: Returns in which mode the computer is operating.
Input: None.
Output: A – 0 = Z80; 1 = R800 ROM; 2 = R800 DRAM.
Registers: AF.

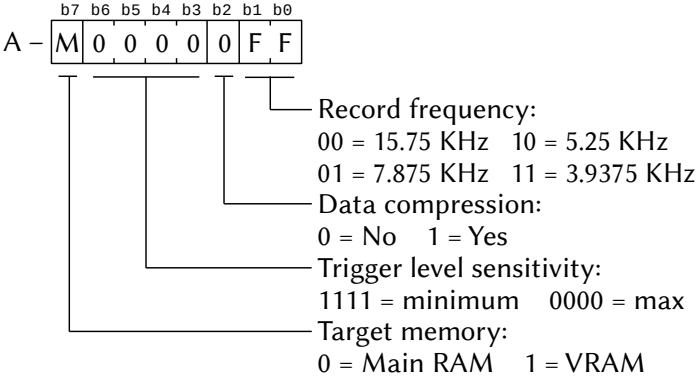
PCMPLY (0186H / Main)
Function: Play sounds through the PCM.
Input: EHL – Address to start reading.
DBC – Size of the block to be reproduced (length).



Note: The 15.75 Khz frequency can only be used in the R800 DRAM mode.

Output: CY – 0 ⟶ Playback OK.
1 ⟶ Playback error.
Cause of error:
A – 0 ⟶ error in specifying the frequency.
1 ⟶ interruption by CTRL+STOP.
EHL – Address as far as it actually reproduced.
Registers: All.

PCMREC (0189H / Main)
Function: Digitize sounds through the PCM.
Input: EHL – Address to start reading.
DBC – Size of the block to be digitized (length).



Note: The 15.75 Khz frequency can only be used in R800 DRAM mode.

Output: CY – 0 ⟶ Record OK.
1 ⟶ Record error.
Cause of error:
A – 0 ⟶ error in specifying the frequency.
1 ⟶ interruption by CTRL+STOP.
EHL – Address as far as it actually recorded.
Registers: All.

9.1.15 – Inter-slot work area routines

RDPRIM (F380H / Work Area)
Function: Reads a byte from any address in any slot.
Input: A – Primary slot to be read.
D – Current return slot.
Output: E – Byte read.

WRPRIM (F385H / Work Area)
Function: Writes a byte to any address in any slot.
Input: A – Primary slot to be read.
D – Current return slot.
E – Byte to be written.
Output: None

CLPRIM (F38CH / Work Area)
Function: Calls an address in any slot
Input: A – Primary slot containing the routine
IX – Address to be called
PUSH AF – Current return slot (in A)
Output: Depends of the called routine

9.2 – SubROM ROUTINES

9.2.1 – Routines for BASIC graphical functions

PAINT (0069H / SubROM) – BASIC Command
Function: Paints an area on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text (parameters of the PAINT command).
Output: HL – Points to the end of the command parameters.
Registers: All.

PSET (006DH / SubROM) – BASIC Command
Function: Draws a point on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text (parameters of the PSET command).
Output: HL – Points to the end of the command parameters.
Registers: All.

ATRSCN (0071H / SubROM) – BASIC Command
Function: Returns color attributes.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

GLINE (0075H / SubROM) – BASIC Command
Function: Draws a line on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

DOBOXF (0079H / SubROM) – BASIC Command
Function: Draws a filled rectangle on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

DOLINE (007DH / SubROM) – BASIC Command
Function: Draws a line on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text
(parameters of the LINE command).
Output: HL – Points to the end of the command parameters.
Registers: All.

BOXLIN (0081H / SubROM) – BASIC Command
Function: Draws a rectangle on a graphic screen.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

PUTSPR (0151H / SubROM) – BASIC Command
Function: Displays a sprite on a graphical screen.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

COLOR (0155H / SubROM) – BASIC command
Function: Change the colors of the screen, sprites or palette.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

SCREEN (0159H / SubROM) – BASIC command
Function: Switches the screen modes.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

WIDTH (015DH / SubROM) – BASIC Command
Function: Changes the number of characters per line in text mode.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

VDP (0161H / SubROM) – BASIC Command
Function: Writes data to a VDP register.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

VDPF (0165H / SubROM) – BASIC Command
Function: Reads data from a VDP register.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

BASE (0169H / SubROM) – BASIC command
Function: Writes data to the VDP base register.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

BASEF (0169H / SubROM) – BASIC Command
Function: Reads data from the VDP base register.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

9.2.2 – Routines for graphical functions

DOGRPH (0085H / SubROM)
Function: Draws a line on a graphic screen.
Input: BC – Initial X coordinate.
HL – Initial Y coordinate.
GXPOS (FCB3H) – Final X coordinate.
GYPOS (FCB5H) – Y coordinate at the end.
ATRBYT (F3F2H) – Attributes.
LOGOPR (FB02H) – Logical operation code.
Output: None.
Registers: AF.

GRPPRT (0089H / SubROM)
Function: Prints a character on a graphical MSX2 screen.
Input: A – Character ASCII code.
ATRBYT (F3F2H) – Attributes.
LOGOPR (FB02H) – Logical operation code.
Output: None.
Registers: All.

SCALXY (008DH / SubROM)
Function: Limits the pixel coordinates to the screen visible area.
Input: BC – X coordinate (horizontal).
DE – Y coordinate (vertical).
Output: BC – X coordinate limited to the border.
DE – Y coordinate limited to the border.
CY = 1 if the coordinates was limited.
Registers: AF.

MAPXYC (0091H / SubROM)
Function: Converts a pair of graphic coordinates to the physical address of the current pixel (puts the "cursor" on the coord).
Input: BC – X coordinate (horizontal).
DE – Y coordinate (vertical).
Output: Screen 3: HL, CLOC (F92AH) – Address at VRAM.
A, CMASK (F92CH) – Mask.
Screen 5~12: HL, CLOC (F92AH) – X coordinate.
A, CMASK (F92CH) – Y coordinate.
Registers: F.

READC (0095H / SubROM)
Function: Read the attributes of a pixel.
Input: CLOC (F92AH) – X Coordinate.
CMASK (F92CH) – Y coordinate.
Output: A – Attribute.
Registers: AF.

SETATR (0099H / SubROM)
Function: Defines attribute in ATRBYT (F3F2H).
Input: A – Attribute.
Output: CY = 1 if there is an error in the attribute.
Registers: F.

SETC (009DH / SubROM)
Function: Defines pixel attribute.
Input: CLOC (F92AH) – X Coordinate.
CMASK (F92CH) – Y coordinate.
ATRBYT (F3F2H) – Attribute.
Output: None.
Registers: AF.

TRIGHT (00A1H / SubROM)
Function: Moves one pixel to the right.
Input: CLOC (F92AH) – X Coordinate.
CMASK (F92CH) – Y coordinate.
Output: CLOC (F92AH) – New X coordinate.
CMASK (F92CH) – New Y coordinate.
CY = 1 if the edge of the screen is reached.
Registers: AF.
Note: Only for Screen 3.

RIGHTC (00A5H / SubROM)
Function: Moves one pixel to the right.
Input: CLOC (F92AH) – X Coordinate.
CMASK (F92CH) – Y coordinate.
Output: CLOC (F92AH) – New X coordinate.
CMASK (F92CH) – New Y coordinate.
Registers: AF.
Note: Only for Screen 3. This routine is the same as TRIGHT (00A1H) except for the absence of the return of the CY flag.

TLEFTC (00A9H / SubROM)
Function: Moves one pixel to the left.
Input: Same as TRIGHT (00A1H / SubROM).
Output: Same as TRIGHT (00A1H / SubROM).
Registers: AF.
Note: Only for Screen 3.

LEFTC (00ADH / SubROM)
Function: Moves one pixel to the left.
Input: Same as RIGHTC (00A5H / SubROM).
Output: Same as RIGHTC (00A5H / SubROM).
Registers: AF.
Note: Only for Screen 3. This routine is the same as TLEFTC (00A9H) except for the absence of the return of the CY flag.

TDOWNC (00B1H / SubROM)
Function: Moves down one pixel.
Input: Same as TRIGHT (00A1H / SubROM).
Output: Same as TRIGHT (00A1H / SubROM).
Registers: AF.
Note: Only for Screen 3.

DOWNC (00B5H / SubROM)
Function: Moves down one pixel.
Input: Same as RIGHTC (00A5H / SubROM).
Output: Same as RIGHTC (00A5H / SubROM).
Registers: AF.
Note: Only for Screen 3. This routine is the same as TDOWNC (00A9H) except for the absence of the return of the CY flag.

TUPC (00B9H / SubROM)
Function: Moves up one pixel.
Input: Same as TRIGHT (00A1H / SubROM).
Output: Same as TRIGHT (00A1H / SubROM).
Registers: AF.
Note: Only for Screen 3.

UPC (00BDH / SubROM)
Function: Moves up one pixel.
Input: Same as RIGHTC (00A5H / SubROM).
Output: Same as RIGHTC (00A5H / SubROM).
Registers: AF.
Note: Only for Screen 3. This routine is the same as TUPC (00B9H) except for the absence of the CY flag return.

SCANR (00C1H / SubROM)
Function: Scan pixels, from left to right, starting from the current pixel until a color code equal to BDRATR (FCB2H) is found or the edge of the screen is reached.
Input: B – 0 = Does not fill the area covered.
255 = Fills the area covered.
C – counter to the edge.
Output: DE – counter to the edge.
C – modified pixel flag.
Registers: All.

SCANL (00C5H / SubROM)
Function: Scan pixels, from right to left, starting from the current pixel until a color code equal to BDRATR (FCB2H) is found or the edge of the screen is reached.
Input: DE – counter to the edge.
Output: DE – counter to the edge.
C – modified pixel flag.
Registers: All.

NVBXLN (00C9H / SubROM)
Function: Draws a lined rectangle.
Input: BC – Initial X coordinate.
HL – Initial Y coordinate.
GXPOS (FCB3H) – Final X coordinate.
GYPOS (FCB5H) – Y coordinate at the end.
ATRBYT (F3F2H) – Attributes.
LOGOPR (FB02H) – Logic operation code.
Output: None.
Registers: All.

NVBXFL (00CDH / SubROM)
Function: Draws a filled rectangle.
Input: Same as NVBXLN (00C9H / SubROM).
Output: None.
Registers: All.

9.2.3 – Duplicate routines (same as MainROM)

CHGMOD (00D1H / SubROM)
Function: Switches the screen modes.
Input: A – 0 to 3 for MSX1, 0 to 8 for MSX2 or 0 to 12 for MSX2+ or higher (Mode 9 is valid only for Korean computers).
Output: None.
Registers: All.

INITXT (00D5H / SubROM)
Function: Initializes the screen in text mode (Screen 0).
Input: TXTNAM (F3B3H) – Name table address.
TXTCGP (F3B7H) – Pattern table address.
LINL40 (F3AEH) – Number of characters per line.
Output: None.
Registers: All.

INIT32 (00D9H / SubROM)
Function: Initializes the screen in Screen 1 mode.
Input: T32NAM (F3BDH) – Characters name table address.
T32COL (F3BFH) – Characters color table address.
T32CGP (F3C1H) – Characters patterns table address.
T32ATR (F3C3H) – Sprites attributes table address.
T32PAT (F3C5H) – Sprites patterns table address.
Output: None.
Registers: All.

INIGRP (00DDH / SubROM)
Function: Initializes the screen in Screen 2 mode.
Input: GRPNAM (F3C7H) – Patterns name table address.
GRPCOL (F3C9H) – Color table address.
GRPCGP (F3CBH) – Patterns generator table address.
GRPATR (F3CDH) – Sprites attributes table address.
GRPPAT (F3CFH) – Sprites patterns table address.

Output: None.
Registers: All.

INIMLT (00E1H / SubROM)
Function: Initializes the screen in the multicolor mode (Screen 3).
Input: MLTNAM (F3D1H) – Patterns name table address.
MLTCOL (F3D3H) – Color table address.
MLTCGP (F3D5H) – Patterns generator table address.
MLTATR (F3D7H) – Sprites attributes table address.
MLTPAT (F3D9H) – Sprites patterns table address.
Output: None.
Registers: All.

SETTXT (00E5H / SubROM)
Function: Puts only the VDP in text mode (Screen 0).
Input: Same as INITXT (00D5H / SubROM).
Output: None.
Registers: All.

SETT32 (00E9H / SubROM)
Function: Puts only the VDP in graphical mode 1 (Screen 1).
Input: Same as INIT32 (00D9H / SubROM).
Output: None.
Registers: All.

SETGRP (00EDH / SubROM)
Function: Puts only the VDP in graphical mode 2 (Screen 2).
Input: Same as INIGRP (00E1H / SubROM).
Output: None.
Registers: All.

SETMLT (00F1H / SubROM)
Function: Puts only the VDP in multicolour mode (Screen 3).
Input: Same as INIMLT (0075H).
Output: None.
Registers: All.

CLRSPR (00F5H / SubROM)
Function: Initializes all sprites. The sprite pattern table is cleared (filled with zeros), the sprite numbers are initialized with the series 0 ~ 31 and the color of the sprites is equal to the background color. The vertical location of the sprites is set to 209 (Screens 0 to 3) or 217 (Screens 4 to 9 or 10 to 12).
Input: SCRMOD (FCAFH) – Screen mode.
Output: None.
Registers: All.

CALPAT (00F9H / SubROM)
Function: Returns the address of the sprite pattern generator table.
Input: A – Sprite number.
Output: HL – Address at VRAM.
Registers: AF, DE, HL.

CALATR (00FDH / SubROM)
Function: Returns the address of a sprite's attribute table.
Input: A – Sprite number.
Output: HL – Address at VRAM.
Registers: AF, DE, HL.

GSPSIZ (0101H / SubROM)
Function: Returns the current size of the sprites.
Input: None.
Output: A – Size of the sprite in bytes. The CY flag is set if the size is 16 x 16 and reset otherwise.
Registers: AF.

9.2.4 – Various routines for MSX2 or higher

GETPAT (0105H / SubROM)
Function: Returns the pattern of a character.
Input: A – ASCII code of the character.
Output: PATWRK (FC40H) – Character standard.
Registers: All.

WRTVRM (0109H / SubROM)
Function: Writes one byte of data to VRAM.
Input: HL – Address of VRAM.
A – byte to be written.
Output: None.
Registers: AF.

RDVRM (010DH / SubROM)
Function: Read the content of one byte of VRAM.
Input: HL – VRAM address to be read.
Output: A – byte read.
Registers: AF.

CHGCLR (0111H / SubROM)
Function: Change the colors of the screen.
Input: FORCLR (F3E9H) – Front color
BAKCLR (F3EAH) – Background color
BDRCLR (F3EBH) – Border color
Output: None.
Registers: All.

CLSSUB (0115H / SubROM)
Function: Clear the screen.
Input: None.
Output: None.
Registers: All.

CLRTXT (0119H / SubROM)
Function: Clear text screen.
Input: None.
Output: None.
Registers: All.

DSPFNK (011DH / SubROM)
Function: Displays the content of the function keys.
Input: None.
Output: None.
Registers: All.

DELLNO (0121H / SubROM)
Function: Deletes a line in text mode.
Input: L – Number of the line to be deleted.
Output: None.
Registers: All.

INSLNO (0125H / SubROM)
Function: Adds a line in text mode.
Input: L – Line number to be added.
Output: None.
Registers: All.

PUTVRM (0129H / SubROM)
Function: Place a character on a text screen.
Input: H – Y coordinate.
L – X coordinate.
Output: None.
Registers: AF.

WRTVDP (012DH / SubROM)
Function: Writes a byte of data to a VDP register.
Input: C – number of the registrar that will receive the data.
B – data byte.
Output: None.
Registers: AF, BC.

VDPSTA (0131H / SubROM)
Function: Read the contents of a VDP register.
Input: A – Number of the register to be read (0 to 9).
Output: A – Value read.
Registers: F.

KYKLOK (0135H / SubROM)
Function: Control of the KANA key and the KANA LED on Japanese computers.
Input: ?
Output: ?
Registers: ?

PUTCHR (0139H / SubROM)
Function: Take a key code, convert it to KANA and put it in a buffer (on Japanese computers).
Input: CY = 0 – Make conversion.
CY = 1 – Does not convert.
Output: ?
Registers: All.

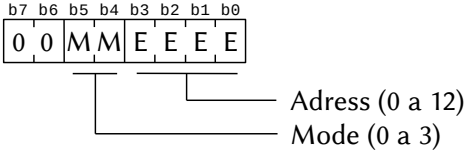
SETPAG (013DH / SubROM)
Function: Defines the video pages.
Input: DPPAGE (FAF5H) – Page shown on the screen.
ACPAGE (FAF6H) – Active page for receiving commands.
Output: None.
Registers: AF.

NEWPAD (01ADH / SubROM)
Function: Returns the state of the mouse or the lightpen.
Input: A – function code:
0 to 7 – No effect.
8 – Check lightpen (255 if connected/touching screen).
9 – Returns the X (horizontal) coordinate.
10 – Returns the Y (vertical) coordinate.
11 – Returns the button state (255 if pressed).
12 – Check mouse on port 1 (255 if connected).
13 – Returns X coordinate offset (horizontal).
14 – Returns Y coordinate offset (vertical).
15 – Always 0.
16 – Check mouse on port 2 (255 if connected).
17 – Returns X coordinate offset (horizontal).
18 – Returns Y coordinate offset (vertical).
19 – Always 0.
20 – Check 2nd lightpen (255 if connected/touch screen).
21 – Returns the X coordinate (horizontal).
22 – Returns the Y (vertical) coordinate.
23 – Returns the button state (255 if pressed).
Output: A – state or value, as described above.
Registers: All.

CHGMDP (01B5H / SubROM)
Function: Switches the screen modes and initializes the color palette.
Input: A – 0 to 3 for MSX1, 0 to 8 for MSX2 or 0 to 12 for MSX2+ or higher (Mode 9 is valid only for Korean computers).
Output: None.
Registers: All.

KNJPRT (01BDH / SubROM)
Function: Writes a Kanji character on a graphic screen (Screens 5 to 8 or 10 to 12). This routine is present only in machines with Kanji ROM.
Input: BC – Kanji character JIS code.
A – Presentation mode:
0 – All lines on the screen.
1 – Even lines.
2 – Odd lines.
Registers: AF.

REDCLK (01F5H / SubROM)
Function: Reads a nibble of data from the clock memory (Clock-IC).
Input: C – SRAM address of the clock, as shown below:



Output: A – Nibble read (4 bits lower).
Registers: AF.

WRTCLK (01F9H / SubROM)
Function: Writes a data nibble to the clock's memory
Input: C – SRAM address of the clock (equal to REDCLK).
A – Nibble to be written (4 bits lower).
Output: None.
Registers: F.

9.2.5 – Color palette handling routines

INIPLT (0141H / SubROM)
Function: Initializes the color palette (the current palette will be saved in VRAM).
Input: None.
Output: None.
Registers: AF, BC, DE.

RSTPLT (0145H / SubROM)
Function: Retrieves the color palette saved in VRAM.
Input: None.
Output: None.
Registers: AF, BC, DE.

GETPLT (0149H / SubROM)
Function: Returns the color levels of the palette.
Input: A – color number in the palette (0 to 15).
Output: B – 4 bits high for the red level.
B – 4 bits low for the blue level.
C – 4 bits low for green level.
Registers: AF, DE.

SETPLT (014DH / SubROM)
Function: Modifies the color levels of the palette.
Input: D – color number in the palette (0 to 15).
A – 4 bits high for the red level.
A – 4 bits low for the blue level.
E – 4 bits low for green level.
Output: None.
Registers: AF.

9.2.6 – Various routines used by BASIC

VPOKE (0171H / SubROM) – BASIC Command
Function: Writes one byte of data to VRAM.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

VPEEK (0175H / SubROM) – BASIC Command
Function: Reads one byte of VRAM data.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

SETS (0179H / SubROM) – BASIC Command
Function: Executes the parameters of the BEEP, ADJUST, TIME and DATE commands
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

BEEP (017DH / SubROM) – BASIC Command
Function: Generates a beep.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

PROMPT (0181H / SubROM) – BASIC Command
Function: Displays the BASIC prompt ("Ok" by default).
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

SDFSCR (0185H / SubROM) – BASIC command
Function: Retrieves the screen parameters of the Clock-IC. When CY = 1, the content of the function keys will be displayed.
Input: CY = 0 after calling MSXDOS.
Output: ?
Registers: All.

SETSCR (0189H / SubROM) – BASIC Command
Function: Retrieves the screen parameters of the Clock-IC and displays a welcome message.
Input: ?
Output: ?
Registers: All.

SCOPY (018DH / SubROM) – BASIC Command
Function: Executes copies between VRAM, BASIC matrices and files on disk.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

GETPUT (01B1H / SubROM) – BASIC command
Function: Executes the parameters of the GET TIME, GET DATE and PUT KANJI commands.
Input: HL – Pointer to the beginning of the BASIC text.
Output: HL – Points to the end of the command parameters.
Registers: All.

9.2.7 – Block transfer routines (bit-blit)

BLTVV (0191H / SubROM)
Function: Transfer data from one VRAM area to another.
Input: HL – Must contain the value F562H.
SX – (F562H, 2) – X coordinate of the source.
SY – (F564H, 2) – Y coordinate of the source.
DX – (F566H, 2) – X coordinate of destination.
DY – (F568H, 2) – Y coordinate of destination.
NX – (F56AH, 2) – Number of pixels in the X direction.
NY – (F56CH, 2) – Number of pixels in the Y direction.
CDUMMY– (F56EH, 1) – Dummy (no data required).
ARGT – (F56FH, 1) – Selects the direction and the expanded VRAM (equal to R # 45 of the VDP).
LOGOP – (F570H, 1) – Logical operation code (same as VDP codes).
Output: CY = 0.
Registers: All.

BLTVM (0195H / SubROM)
Function: Transfer data from Main RAM to VRAM.
Input: HL – Must contain the value F562H.
DPTR – (F562H, 2) – Source address in RAM.
DUMMY – (F564H, 2) – Dummy (no data required).

DX – (F566H, 2) – X coordinate of destination.
DY – (F568H, 2) – Y coordinate of destination.
NX – (F56AH, 2) – Number of pixels in the X direction (no data required; already filled in).
NY – (F56CH, 2) – Number of pixels in the Y direction (no data required; already filled in).
CDUMMY– (F56EH, 1) – Dummy (no data required).
ARGT – (F56FH, 1) – Selects the direction and the expanded VRAM (equal to R # 45 of the VDP).
LOGOP – (F570H, 1) – Logical operation code (same as VDP codes).

Output: CY = 0 – Transfer successful.
CY = 1 – Transfer error.

Registers: All.

Note: The memory space to be allocated, in bytes, must obey the following formulas:
Screen 6: (NX * NY) / 4 + 4
Screens 5 and 7: (NX * NY) / 2 + 4
Screens 8, 10, 11 and 12: (NX * NY) + 4

BLTMV (0199H / SubROM)

Function: Transfer data from VRAM to Main RAM.

Input: HL – Must contain the value F562H.
SX – (F562H, 2) – X coordinate of the source.
SY – (F564H, 2) – Y coordinate of the source.
DPTR – (F566H, 2) – Destination address in RAM.
DUMMY – (F568H, 2) – Dummy (no data required).
NX – (F56AH, 2) – Number of pixels in the X direction.
NY – (F56CH, 2) – Number of pixels in the Y direction.
CDUMMY– (F56EH, 1) – Dummy (no data required).
ARGT – (F56FH, 1) – Selects the direction and the expanded VRAM (equal to R # 45 of the VDP).

Output: CY = 0.

Registers: All.

Note: The memory space to be allocated, in bytes, must obey the following formulas:
Screen 6: (NX * NY) / 4 + 4
Screens 5 and 7: (NX * NY) / 2 + 4
Screens 8, 10, 11 and 12: (NX * NY) + 4

BLTVD (019DH / SubROM)
Function: Transfer data from disk to VRAM.
Input: HL – Must contain the value F562H.
FNPTR – (F562H, 2) – Address of the filename.
DUMMY – (F564H, 2) – Dummy (no data required).
DX – (F566H, 2) – X coordinate of destination.
DY – (F568H, 2) – Y coordinate of destination.
NX – (F56AH, 2) – Number of pixels in the X direction (no data required; already filled in).
NY – (F56CH, 2) – Number of pixels in the Y direction (no data required; already filled in).
CDUMMY– (F56EH, 1) – Dummy (no data required).
ARGT – (F56FH, 1) – Selects the direction and the expanded VRAM (same as R # 45 of the VDP).
LOGOP – (F570H, 1) – Logical operation code (same as VDP codes).
Output: CY = 0 – Transfer successful.
CY = 1 – Error in the transfer or in the parameters.
Registers: All.

BLTDV (01A1H / SubROM)
Function: Transfer data from VRAM to disk.
Input: HL – Must contain the value F562H.
SX – (F562H, 2) – X coordinate of the source.
SY – (F564H, 2) – Y coordinate of the source.
FNPTR – (F566H, 2) – Address of the filename.
DUMMY – (F568H, 2) – Dummy (no data required).
NX – (F56AH, 2) – Number of pixels in the X direction.
NY – (F56CH, 2) – Number of pixels in the Y direction.
CDUMMY– (F56EH, 1) – Dummy (no data required).
Output: CY = 0.
Registers: All.

BLTMD (01A5H / SubROM)
Function: Transfer data from disk to Main RAM.
Input: HL – Must contain the value F562H.
FNPTR – (F562H, 2) – Address of the filename.
DUMMY – (F564H, 2) – Dummy (no data required).
SPTR – (F566H, 2) – Initial data address
EPTR – (F568H, 2) – Final data address

Output: CY = 0
Registers: All.

BLTDM (01A9H / SubROM)
Function: Transfer data from Main RAM to disk.
Input: HL – Must contain the value F562H.
SPTR – (F562H, 2) – Initial data address.
EPTR – (F564H, 2) – Final data address.
FNPTR – (F566H, 2) – Address of the filename.
Output: CY = 0
Registers: All.

9.3 – MATH-PACK ROUTINES

9.3.1 – Floating point mathematical functions

DECSUB	268CH	DAC – DAC – ARG	(double precision)
DECADD	269AH	DAC – DAC + ARG	
DECMUL	27E6H	DAC – DAC * ARG	
DECDIV	289FH	DAC – DAC / ARG	
COS	2993H	DAC – COS (DAC)	
SIN	29ACH	DAC – SIN (DAC)	
TAN	29FBH	DAC – TAN (DAC)	
ATN	2A14H	DAC – ATN (DAC)	
LOG	2A72H	DAC – LOG (DAC)	
SQR	2AFFH	DAC – SQR (DAC)	
EXP	2B4AH	DAC – EXP (DAC)	
SGNEXP	37C8H	DAC – DAC ^ ARG	(single-precision)
DBLEXP	37D7H	DAC – DAC ^ ARG	(double-precision)

9.3.2 – Operations with integer numbers

UMULT	314AH	DE – BC * DE	(Unsigned multiplication)
ISUB	3167H	HL – DE – HL	
IADD	3172H	HL – DE + HL	
IMULT	3193H	HL – DE * HL	
IDIV	31E6H	HL – DE / HL	
IMOD	323AH	HL – DE mod HL	
		DE – DE / HL	
INTEXP	383FH	DAC – DE ^ HL	

9.3.3 – Special functions

DECNRM	26FAH	Normalises DAC, removing excessive zeros from the mantissa. (Ex. 0.00123 ⟶ 0.123E-2).
DECROU	273CH	Rounds DAC
RND	2BDFH	Generates a random number from the number contained in DAC, returning it in DAC.
SIGN	2E71H	A – Sign of DAC.
ABSFN	2E82H	Extracts the absolute value (module) of DAC and stores the result in DAC.
NEG	2E8DH	Inverts the DAC signal.
SGN	2E97H	DAC – Sign of DAC: DAC +2, +3: 0000H = Zero 0001H = Positive FFFFH = Negative

FCOMP	2F21H	Left: CBED	Right: DAC	single precision
ICOMP	2F4DH	Left: DE	Right: HL	integer number
XDCOMP	2F5CH	Left: ARG	Right: DAC	double precision

Results will be in A register. Meanings of A register are:

A = 1	⟶	left < right
A = 0	⟶	left = Right
A = –1	⟶	left > right

9.3.4 – Movement

MAF	2C4DH	ARG ⟵ DAC	Double precision
MAM	2C50H	ARG ⟵ (HL)	
MOV8DH	2C53H	(DE) ⟵ (HL)	
MFA	2C59H	DAC ⟵ ARG	
MFM	2C5CH	DAC ⟵ (HL)	
MMF	2C67H	(HL) ⟵ DAC	
MOV8HD	2C6AH	(HL) ⟵ (DE)	
XTF	2C6FH	(SP) ⟷ DAC	
PHA	2CC7H	ARG ⟵ (SP)	
PHF	2CCCH	DAC ⟵ (SP)	
PPA	2CDCH	(SP) ⟵ ARG	
PPF	2CE1H	(SP) ⟵ DAC	

PUSHF	2EB1H	DAC ← (SP)	Single precision
MOVFM	2EBEH	DAC ← (HL)	
MOVFR	2EC1H	DAC ← (CBED)	
MOVRF	2ECCH	(CBED) – DAC	
MOVRMI	2ED6H	(CBED) – (HL)	
MOVRM	2EDFH	(BCDE) – (HL)	
MOVMF	2EE8H	(HL) ← DAC	
MOVE	2EEBH	(HL) ← (DE)	
VMOVAM	2EEFH	ARG ← (HL)	VALTYP
MOVVFM	2EF2H	(DE) ← (HL)	
VMOVE	2EF3H	(HL) ← (DE)	
VMOVFA	2F05H	DAC ← ARG	
VMOVFM	2F08H	DAC ← (HL)	
VMOVAF	2F0DH	ARG ← DAC	
VMOVMF	2F10H	(HL) ← DAC	

9.3.5 – Conversions

FRCINT	2F8AH	Converts DAC to a 2-byte integer (DAC+2,+3)
FRCSNG	2FB2H	Converts DAC to a single precision real number
FRCDBL	303AH	Converts DAC to a double precision real number
FIXER	30BEH	DAC – SGN(DAC) * INT(ABS(DAC))

FIN 3299H — Stores a string representing the floating-point number in DAC, converting it in real.

Input:
- HL ← Starting address of the string
- A ← First character of the string

Output:
- DAC ← Real number
- C ← FFH: without decimal point
  00H: with decimal point
- B ← Number of digits after the decimal point
- D ← Number of digits

FOUT 3425H — Converts a real number contained in DAC to an unformatted string.

Input:
- A – Always 0
- B – Number of digits before the decimal point
- C – Number of digits after the decimal point, including this one.

Output:
- HL – Address of the first character of the string.

PUFOUT (3426H) Converts a real number contained in DAC to a formatted string.

Input: A – Format:
bit 7 – 0: unformatted 1: formatted
bit 6 – 0: no commas 1: commas every 3 digits.
bit 5 – 0: meaningless 1: fill spaces with "*"
bit 4 – 0: meaningless 1: add "$" before number
bit 3 – 0: meaningless 1: add "+" for positive numbers
bit 2 – 0: meaningless 1: sign after the number
bit 1 – 0: not used
bit 0 – 0: fixed point 1: floating point
B ← Number of digits before decimal point.
C ← Number of digits after decimal point, including this one.

Output: HL ← Starting address of the string.

FOUTB (371AH) Converts an integer contained in DAC to a string expression in binary format.

Input: DAC+2, +3 – Integer number.
VALTYP – 2.

Output: HL – Initial address of the binary string.

FOUTO (371EH) Converts an integer contained in DAC to a string expression in octal format.

Input: DAC+2, +3 – Integer number.
VALTYP – 2.

Output: HL – Initial address of the octal string.

FOUTH (3722H) Converts an integer contained in DAC to a string expression in hexadecimal format.

Input: DAC+2, +3 – Integer number.
VALTYP – 2.

Output: HL – Initial address of the hexadecimal string.

FIN 3299H Converts a string representing a real number to BCD format and stores it in DAC.

Input: HL – Address of the first character of the string.
A – First character of the string.

Output: DAC – Real number in BCD.
C ←FFH – Without decimal point;
0 – With decimal point.
B – Number of digits after the decimal point.
D – Total number of digits.

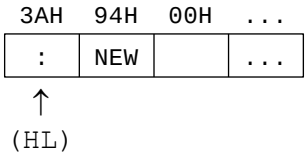
9.4 – BASIC INTERPRETER ROUTINES

9.4.1 – Execution routines

READYR (409BH / Main)
Function: Returns to the command level (BASIC hot start).
Input: None
Output: None

CRUNCH (42B2H / Main)
Function: Converts BASIC text from ASCII form to tokenized form.
Input: HL ← ASCII text address to be converted, ending with a 00H byte.
Output: KBUF (F41FH) – Converted BASIC text.

NEWSTT (4601H / Main)
Function: Executes a BASIC text. The text must be in tokenized form.
Input: HL ← pointer to the beginning of the text to be executed. The text must be in the form illustrated below:



Output: None

CHRGTR (4666H / Main) – From 0010H
Function: Extracts a character from the BASIC text, starting with (HL) +1. Spaces are ignored.
Input: HL ← starting address of the text
Output: HL ← extracted character address
A ← ASCII code of the extracted character
Z = "1" if it is the end of the line (00H or 3AH ":")
CY = "1" if it is a character from 0 to 9

FRMEVL (4C64H / Main)
Function: Evaluates an expression and returns the result.
Input: HL ← start address of the expression in the BASIC text.
Output: HL ← final address of the expression +1.
VALTYP (F663H) ← 2 – Integer variable
4 – Single precision variable
8 – Double precision variable
3 – String variable
DAC (F7F6H) – Result of the evaluated expression.

GETBYT (521CH / Main)
Function: Evaluates an expression and returns a 1-byte result. When the result extrapolates the value of 1 byte, an "Illegal Function Call" error will be generated and the execution will return to the command level.
Input: HL ← starting address of the expression to be evaluated
Output: HL ← final address of the expression +1.
A,E – Evaluation result (A and E contain the same value).

FRMQNT (542FH / Main)
Function: Evaluates an expression and returns a result of 2 bytes (integer). When the result extrapolates the value of 2 bytes, an "Overflow" error will be generated and the execution will return to the command level.
Input: HL ← starting address of the expression to be evaluated
Output: HL ← final address of the expression +1.
DE ← evaluation result

SYNCHR (558CH / Main) – 0008H
Function: Tests if the character pointed by (HL) is the one specified. If not, it generates "Syntax error"; otherwise it calls CHRGTR (4666H / Main).
Input: HL ← points to the character to be tested
The character for comparison must be placed after an instruction "RST 0008H" in the form of a line parameter, as shown in the example below:

```
LD   HL,CHAR
RST  008H
DEFB 'A'
|
CHAR: DEFB 'B'
```

Output: HL is incremented by one and A receives (HL). When the tested character is numeric, the CY flag is set. The end of the declaration (00H or 3AH ":") sets the Z flag.

GETYPR (5597H / Main) – 0028H

Function: Gets the type of operand contained in DAC.

Input: None

Output: Flags CY, S, Z and P / V, as shown in the table below:

Integer:	C=1	S=1 *	Z=0	P/V=1
Simple precision:	C=1	S=0	Z=0	P/V=0 *
Double precision:	C=0 *	S=0	Z=0	P/V=1
String:	C=1	S=0	Z=1 *	P/V=1

Note: The types can be recognized verifying only the flags marked with " * ".

PTRGET (5EA4H / Main)

Function: Gets the address for storing a variable or matrix. The address is also obtained when the variable has not been assigned. When the value of SUBFLG (F5A5H) is different from 0, the starting address of an array will be obtained; otherwise, the address of the array element will be obtained.

Input: HL ← starting address of the variable name in BASIC text
SUBFLG (F6A5H) – 0: single variable, other value: matrix

Output: HL ← address after the variable name
DE ← address of the content of the variable.

FRESTR (67D0H / Main)

Function: Registers the result of a string obtained by FRMEVL (4C64H) and obtains the respective descriptor. When evaluating a string, this routine is usually combined with FRMEVL as described below:

```
CALL FRMEVL
PUSH HL
CALL FRESTR
EX DE, HL
POP HL
LD A, (DE)
...
```

Input: VALTYP (F663H) – Variable type (must be 3).
DAC (F7F6H) – Pointer to the string descriptor.

Output: HL ← pointer to the string descriptor.

9.4.2 – Command and function routines

Command / Token Address Token Address

function function in the routine table

Command/ Function	Token	Function Token	Table Adress	Routine Adress
>	EEH	–	Afat	-
=	EFH	–	Afat	-
<	F0H	–	Afat	-
+	F1H	–	Afat	-
-	F2H	–	Afat	-
*	F3H	–	Afat	-
/	F4H	–	Afat	-
^	F5H	–	Afat	-
$	FCH	–	Afat	-
ABS	06H	FF86H	39E8H	2E82H
AND	F6H	–	Afat	-
ASC	15H	FF95H	3A06H	680BH
ATN	0EH	FF8EH	39F8H	2A14H
ATTR$	E9H	–	Afat	7C43H
AUTO	A9H	–	3973H	49B5H
BASE	C9H	–	39BEH	7B5AH
BEEP	C0H	–	39ACH	00C0H
BIN$	1DH	FF9DH	3A16H	6FFFH
BLOAD	CFH	–	39CAH	6EC6H
BSAVE	D0H	–	39CCH	6E92H
CALL	CAH	–	39C0H	55A8H
CDBL	20H	FFA0H	3A1CH	303AH
CHR$	16H	FF96H	3A08H	681BH
CINT	1EH	FF9EH	3A18H	2F8AH
CIRCLE	BCH	–	39A4H	5B11H
CLEAR	92H	–	3950H	64AFH
CLOAD	9BH	–	3962H	703FH
CLOSE	B4H	–	3994H	6C14H
CLS	9FH	–	396AH	00C3H
CMD	D7H	–	39DAH	7C34H
COLOR	BDH	–	39A6H	7980H
CONT	99H	–	395EH	6424H
COPY	D6H	–	39D8H	7C2FH

COS	0CH	FF8CH	39F4H	2993H
CSAVE	9AH	–	3960H	6FB7H
CSNG	1FH	FF9FH	3A1AH	2FB2H
CSRLIN	E8H	–	Afat	790AH
CVD	2AH	FFAAH	3A30H	7C70H
CVI	28H	FFA8H	3A2CH	7C66H
CVS	29H	FFA9H	3A2EH	7C6BH
DATA	84H	–	3934H	485BH
DEF	97H	–	395AH	501DH
DEFDBL	AEH	–	3988H	4721H
DEFINT	ACH	–	3984H	471BH
DEFSNG	ADH	–	3986H	471EH
DEFSTR	ABH	–	3982H	4718H
DELETE	A8H	–	397CH	53E2H
DIM	86H	–	3938H	5E9FH
DRAW	BEH	–	39A8H	5D6EH
DSKF	26H	FFA6H	3A28H	7C39H
DSKI$	EAH	–	Afat	7C3EH
DSKO$	D1H		39CEH	7C16H
ELSE	A1H	3AA1H	396EH	485DH
END	81H		396EH	63EAH
EOF	2BH	FFABH	3A32H	6D25H
EQV	F9H	–	Afat	-
ERASE	A5H		3976H	6477H
ERL	E1H	–	Afat	4E0BH
ERR	E2H	–	Afat	4DFDH
ERROR	A6H	–	3978H	49AAH
EXP	0BH	FF8BH	39F2H	2B4AH
FIELD	B1H	–	398EH	7C52H
FILES	B7H	–	39AAH	6C2FH
FIX	21H	FFA1H	3A1EH	30BEH
FN	DEH	–	Afat	5040H
FOR	82H	–	3920H	4524H
FPOS	27H	FFA7H	3A2AH	6D39H
FRE	0FH	FF8FH	39FAH	69F2H
GET	B2H	–	3990H	775BH
GOSUB	8DH	–	3948H	47B2H
GOTO	89H	–	393EH	47E8H
GO TO	89H	–	393EH	47E8H
HEX$	1BH	FF9BH	3A12H	65FAH
IF	8BH		3942H	49E5H
IMP	FAH	–	3A20H	7940H

INKEY$	ECH	–	Afat	7347H
INP	10H	FF90H	39FCH	4001H
INPUT	85H	–	3936H	4B6CH
INSTR	E5H	–	39F6H	29FBH
INT	05H	FF85H	39E6H	30CFH
IPL	D5H	–	39D6H	7C2AH
KEY	CCH	–	3964H	786CH
KILL	D4H	–	39D4H	7C25H
LEFT$	01H	FF81H	39DEH	6861H
LEN	12H	FF92H	3A00H	67FFH
LET	88H	–	393CH	4880H
LFILES	BBH	–	39A2H	6C2AH
LINE	AFH	–	398AH	4B0EH
LIST	93H	–	3952H	522EH
LLIST	9EH	–	3968H	5229H
LOAD	B5H	–	3996H	6B5DH
LOC	2CH	FFACH	3A34H	6D03H
LOCATE	D8H	–	39DCH	7766H
LOF	2DH	FFADH	3A36H	6D14H
LOG	0AH	FF8AH	39F0H	2A72H
LPOS	1CH	FF9CH	3A14H	4FC7H
LPRINT	9DH	–	394CH	4A1DH
LSET	B8H	–	399CH	7C48H
MAX	CDH	–	39C6H	7E4BH
MERGE	B6H	–	3998H	6B5EH
MID$	03H	FF83H	39E2H	689AH
MKD$	30H	FFB0H	3A3CH	7C61H
MKI$	2EH	FFAEH	3A38H	7C57H
MKS$	2FH	FFAFH	3A3AH	7C5CH
MOD	FBH	–	Afat	–
MOTOR	CEH	–	39C8H	73B7H
NAME	D3H	–	39D2H	7C20H
NEW	94H	–	3954H	6286H
NEXT	83H	–	3932H	6527H
NOT	E0H	–	Afat	-
OCT$	1AH	FF9AH	3A10H	7C70H
OFF	EBH	–	3A02H	3A02H
ON	95H	–	3956H	48E4H
OPEN	B0H	–	398CH	6AB7H
OR	F7H	–	Afat	-
OUT	9CH	–	3964H	4016H
PAD	25H	FFA5H	3A26H	7969H

PAINT	BFH	–	39AAH	59C5H
PDL	24H	FFA4H	3A24H	795AH
PEEK	17H	FF97H	3A0AH	541CH
PLAY	C1H	–	39AEH	73E5H
POINT	EDH	–	Afat	5803H
POKE	98H	–	395CH	5423H
POS	11H	FF91H	39FEH	4FCCH
PRESET	C3H	–	39B2H	57E5H
PRINT	91H	–	394EH	4A24H
PSET	C2H	–	39B0H	57EAH
PUT	B3H	–	3992H	7758H
READ	87H	–	393AH	4B9FH
REM	8FH	3A8FH	394AH	485DH
RENUM	AAH	–	3980H	5468H
RESTORE	8CH	–	3944H	63C9H
RESUME	A7H	–	397AH	495DH
RETURN	8EH	–	3948H	4821H
RIGHT$	02H	FF82H	39E0H	6891H
RND	08H	FF88H	39ECH	2BDFH
RSET	B9H	–	399EH	7C4DH
RUN	8AH	–	3940H	479EH
SAVE	BAH	–	39A0H	6BA3H
SCREEN	C5H	–	39B6H	79CCH
SET	D2H	–	39D0H	7C1BH
SGN	04H	FF84H	39E4H	2E97H
SIN	09H	FF89H	39EEH	29ACH
SOUND	C4H	–	39B4H	73CAH
SPACE$	19H	FF99H	3A0EH	6848H
SPC(	DFH	–	Afat	-
SPRITE	C7H	–	39BAH	7A48H
SQR	07H	FF87H	39EAH	2AFFH
STEP	DCH	–	Afat	-
STICK	22H	FFA2H	3A20H	7940H
STOP	90H	–	394CH	63E3H
STR$	13H	FF93H	3A02H	6604H
STRIG	23H	FFA3H	3A22H	794CH
STRING$	E3H	–	Afat	6829H
SWAP	A4H	–	3974H	643EH
TAB(	DBH	–	Afat	-
TAN	0DH	FF8DH	39F6H	29FBH
THEN	DAH	–	Afat	-
TIME	CBH	–	39C2H	7911H

TO	D9H	–	Afat	-
TROFF	A3H	–	3972H	6439H
TRON	A2H	–	3970H	6438H
USING	E4H	–	Afat	-
USR	DDH	–	Afat	4FD5H
VAL	14H	FF94H	3A04H	68BBH
VARPTR	E7H	–	39FAH	4E41H
VDP	C8H	–	39BCH	7B37H
VPEEK	18H	FF98H	3A0CH	7BF5H
VPOKE	C6H	–	39B8H	7BE2H
WAIT	96H	–	3958H	401CH
WIDTH	A0H	–	396CH	51C9H
XOR	F8H	–	Afat	-

9.5 – EXTENDED BIOS ROUTINES

9.5.1 – Extended BIOS Entry

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions. Only available if bit 0 of the HOKVLD system flag (FB20H) is set to 1.
Input: A ← Always 00H.
D ← Device ID:
- 00 – Internal commands (broadcast commands)
- 01~03 – Free
- 04 – DOS2 Mapped Memory Handling
- 05~07 – Free
- 08 – RS232C / MSX Modem
- 09 – Free
- 10 – MSX-Audio
- 11 – MSX MIDI
- 12~15 – Free
- 16 – MSX-JE
- 17 – Kanji Driver
- 18~33 – Free
- 34 – UNAPI
- 35~51 – Free
- 52 – MWMPLAY (MoonBlaster 4 Wave Replayer)
- 53~76 – Free

77 – Memman
78 – Nowind
79~204 – Free
205 – MCDRV (Micro Cabin BGM Replayer)
206~239 – Free
240 – MGSDRV (SCC music player)
241~254 – Free
255 – System Exclusive

E ← Function number (0 to 255).

Output: Depends of the device and function called.
CY = 1 if the specified device is not found.

Registers: All.

9.5.2 – Internal commands (broadcast commands)

EXTBIO (FFCAH/Work Area)

Function: Accesses extended BIOS functions.

Input: A – 00H.
D – 00H – Internal command.
E – 00H – Examines the devices present in the system, asks it to record its own number in the table, increments the pointer by 1 and moves to the next device.
B – ID of the slot where the table will be placed.
HL – Table address.

Output: B – Table slot ID.
HL – Table address.
CY = 1 if there are no devices.

Registers: All.

EXTBIO (FFCAH/Work Area)

Function: Accesses extended BIOS functions.

Input: A = 00H.
D = 00H ← Internal command.
E = 01H – Gets the number of MSX BASIC interrupt events. It internally manages up to 26 events, which are:

0~9	ON KEY GOSUB
10	ON STOP GOSUB

11 ON SPITE GOSUB
12~16 ON STRIG GOSUB
17 ON INTERVAL GOSUB
18~23 For expansion devices
24~25 Reserved (prohibited use)

Output: A – Number of active events.
Registers: All.

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A – 00H.
D – 00H ← Internal command.
E – 02H ← Declare interrupt prohibition (disables interrupts for the default time of 1 mS).
Output: None.
Registers: All.

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A – 00H.
D – 00H ← Internal command.
E – 03H ← Declare interrupt permission (enables interrupts blocked by the 02H function).
Output: None.
Registers: All.

9.5.3 –Memory Mapper

EXTBIO (FFCAH/Work Area)
Function: Access extended BIOS functions
Input: A – 00H.
D – 04H ← MSXDOS2 Memory Mapped Handling Device.
E – 01H ← Returns the address of the Memory Mapper variable table.
Output: A – Primary mapper slot ID.
DE – Reserved.

HL – Starting address of the variable table, whose structure is as follows:

+00H	Primary Mapper Slot ID
+01H	Total number of 16K segments
+02H	Number of free 16K segments
+03H	Number of 16k segments allocated by the system (minimum 6 for primary mapper)
+04H	Number of 16K segments allocated to user
+05H~+07H	Reserved (Always 00H)
+08H...	Entries for other mappers in other slots. If there is none, it will contain 00H.

Registers: All.

EXTBIO (FFCAH/Work Area)

Function: Access extended BIOS functions

Input: A – 00H.
D – 04H – Memory Mapper Handling Device.
E – 02H – Returns several parameters related to the Memory Mapper.

Output: A – Total number of segments (logical pages) for the primary mapper.
B – Primary mapper slot ID.
C – Number of free segments (logical pages) in the primary mapper.
DE – reserved.
HL – Start address of a mapper support subroutine call table. The format of this table is as follows:

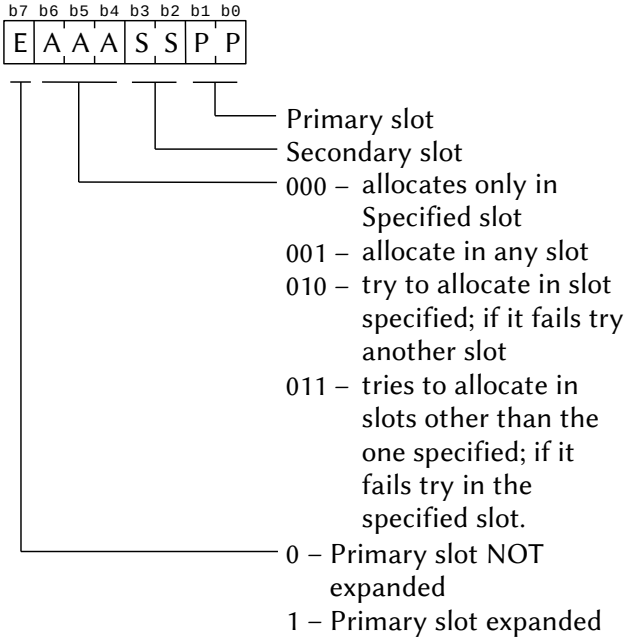
+00H	ALL_SEG	Allocates a 16K segment
+03H	FRE_SEG	Releases a 16K segment
+06H	RD_SEG	Read a byte from the address (A:HL) to A
+09H	WR_SEG	Write the contents of E at the address (A:HL)
+0CH	CAL-SEG	Inter-segment call by address IYh:IX
+0FH	CALLS	Inter-segment call. Parameters in line after the CALL statement
+12H	PUT_PH	Place a segment on the physical page (HL)
+15H	GET_PH	Returns the current segment for the physical page (HL)

+18H PUT_P0	Place a segment on physical page 0
+1BH GET_P0	Returns the current page 0 segment.
+1EH PUT_P1	Place a segment on physical page 1
+21H GET_P1	Returns the current page 1 segment.
+24H PUT_P2	Place a segment on physical page 2
+27H GET_P2	Returns the current page 2 segment.
+2AH PUT_P3	Not supported as page 3 cannot be switched. If called, it just returns.
+2DH GET_P3	Returns the current page 3 segment.

Registers: All.

9.5.3.1 – Memory Mapper Manipulation Routines

ALL_SEG (HL+00H/ExtBIOS) – HL value obtained via EXTBIO
Function: Allocate a 16K segment of the mapper.
Input: A – 00H – Allocates a user segment
01H – Allocates a system segment
B – 00H – Allocates only to primary mapper



Output: CY = 1 ⟶ There are no free segments
0 ⟶ Segment allocated
A – Segment number
B – Segment slot ID
Note: A system segment will only be released using the FRE_SEG routine. In the case of a user segment, whenever the program that uses it is closed, the segments are released, which is not the case with the system segments.

FRE_SEG (HL+03H/ExtBIOS) – HL value obtained via EXTBIO
Function: Free a 16K segment from the mapper.
Input: A – segment number to be released
B – If it is 00H, it releases only on the primary mapper; if it is different from 00H it releases in any other mapper than the primary one (see ALL_SEG).
Output: CY = 0 – Segment released
1 – Error in releasing the segment

RD_SEG (HL+06H/ExtBIOS) – HL value obtained via EXTBIO
Function: Read a byte from the mapper.
Input: A – segment number from which the byte will be read.
HL – address to be read (0000H to 3FFFH).
Output: A – byte read.
All other registers are preserved.

WR_SEG (HL+09H/ExtBIOS) – HL value obtained via EXTBIO
Function: Write a byte to the mapper.
Input: A – segment number where the byte will be written.
HL – address to be written (0000H to 3FFFH).
E – value to write.
Output: A – corrupted while writing.
All other registers are preserved.

CAL_SEG (HL+0CH/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls a routine in any area of the mapper.
Input: IYh – segment number to be called
IX – address to be called (0000H to FFFFH)
AF, BC, DE and HL can contain parameters for the routine. Do not use AF', BC', DE' and HL' as they are corrupted during the call

Output: AF, BC, DE, HL, IX and IY can contain valid return values.
AF', BC', DE' and HL' return corrupted.

CALLS (HL+0FH/ExtBIOS) – HL value obtained via EXTBIO

Function: Calls a routine in any area of the mapper through inline parameters.

Input: AF, BC, DE and HL can contain parameters for the routine. Do not use AF', BC', DE' and HL' as they are corrupted during the call. The call string must be in the following format:

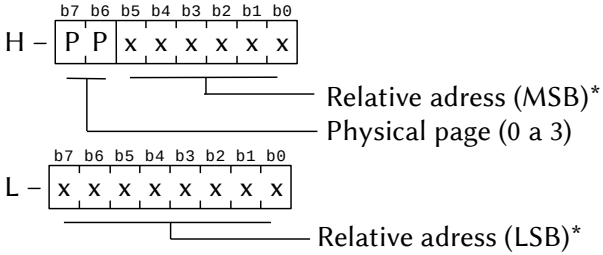
```
CALL CALLS
DEFB SEGMENT
DEFW ADDRESS
```

Output: AF, BC, DE, HL, IX and IY can contain valid return values.
AF', BC', DE' and HL' return corrupted.

PUT_PH (HL+12H/ExtBIOS) – HL value obtained via EXTBIO

Function: Enables a mapper segment on a physical page.

Input: A – Mapper segment

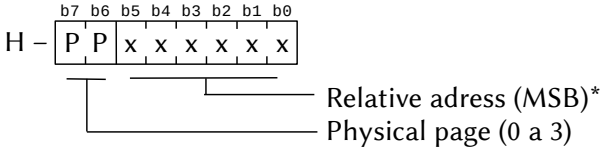


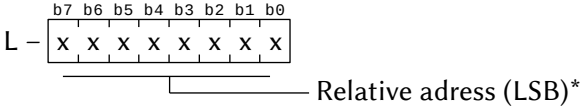
* Relative address is optional.

Output: None. All registers are preserved.

GET_PH (HL+15H/ExtBIOS) – HL value obtained via EXTBIO

Function: Returns the current active segment on a physical page.





* Relative address is optional.

Output: A – Segment number.
All other registers are preserved.

PUT_P0 (HL+18H/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a mapper segment on physical page 0.
Input: A – segment number to be enabled
Output: None. All registers are preserved.

GET_P0 (HL+1BH/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the active segment on physical page 0.
Input: None
Output: A – active segment number
All other registers are preserved.

PUT_P1 (HL+1EH/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a mapper segment on physical page 1.
Input: A – segment number to be enabled
Output: None. All registers are preserved.

GET_P1 (HL+21H/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the active segment on physical page 1.
Input: None
Output: A – active segment number
All other registers are preserved.

PUT_P2 (HL+24H/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a mapper segment on physical page 2.
Input: A – segment number to be enabled
Output: None. All registers are preserved.

GET_P2 (HL+27H/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the active segment on physical page 2.
Input: None
Output: A – active segment number
All other registers are preserved.

PUT_P3 (HL+2AH/ExtBIOS) – HL value obtained via EXTBIO
Function: Not supported since physical page 3 cannot be swapped.
A call to this function has no effect.

GET_P3 (HL+2DH/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the active segment on physical page 0.
Input: None.
Output: A – Active segment number.
All other registers are preserved.

CALL_MAP (HL+30H/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls a routine in any area of the mapped RAM.
Input: IYh – Slot number.
IYl – Segment number.
IX – Routine address, which must necessarily be on page 1 (4000H to 7FFFH).
AF, BC, DE, HL – Parameters for the routine. (Do not use AF', BC', DE' and HL' as they are corrupted in the call).
Output: AF, BC, DE, HL, IX, IY – May contain valid return values.
AF', BC', DE' and HL' return corrupted.
Note: Exclusive routine for NEXTOR.

RD_MAP (HL+33H/ExtBIOS) – HL value obtained via EXTBIO
Function: Reads a byte from a RAM segment.
Input: A – Slot number.
B – Segment number.
HL – Address to read (highest two bits will be ignored).
Output: A – Data byte read.
F, BC, DE, HL, IX, IY return preserved.
Note: Exclusive routine for NEXTOR.

CALL_MAPI (HL+36H/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls routine on a mapped RAM segment (inline parameters).
Input: AF, BC, DE, HL – Parameters for the called routine. Do not use AF', BC', DE' and HL' as they are corrupted during the call. The call string must be in the following format:

```
CALL CALL_MAPI
DEFB SLOT
DEFB ADDRESS
DEFB SEGMENT NUMBER
; It is not necessary to use RET
```

Where
• SLOT – Is the slot to be called, from 0 to 3
• ADDRESS – Address to be called as an index of a table, which can vary from 0 to 63, where 0=4000H, 1=4003H, 2=4006H, etc.
• SEGMENT NUMBER – Can range from 0 to 255.

Output: AF, BC, DE, HL, IX, IY – Parameters returned by the routine.
Note: Exclusive routine for NEXTOR.

WR_MAP (HL+39H/ExtBIOS) – HL value obtained via EXTBIO
Function: Writes a byte to a mapped RAM segment.
Input: A – Slot number.
B – Segment number.
E – Byte to write.
HL – Address to be written (highest two bits are ignored).
Output: A – Data read from the specified address.
F, BC, DE, HL, IX, IY return preserved.
Note: Exclusive routine for NEXTOR.

9.5.4 – RS232C Serial Port and MSX Modem

EXTBIO (FFCAH/Work Area)
Function: Access extended BIOS functions
Input: A – 00H.
D – 08H – RS232C manipulation device.
E – 00H – Returns the address of the input address table of the RS232C routines.
B – Address table slot ID.
HL – Table address.
Output: CY = 1 ⟶ no RS232C interfaces.
0 ⟶ HL is incremented by 4 for each interface found and will point to the end of a table that reserves 4 bytes for each RS232C found. The original value of HL points to the beginning of the table, which has the following structure:

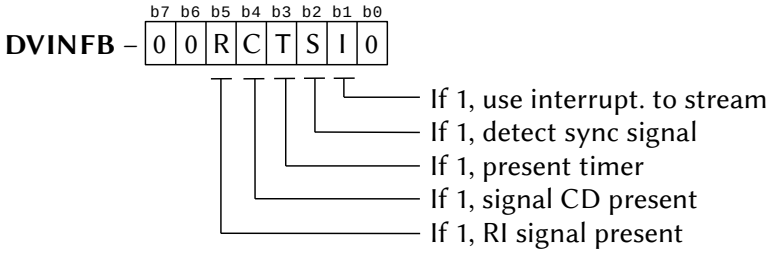
+00H Slot ID.
+01H Lowest address.
+02H Highest address.
+03H Reserved for expansion.

The slot ID (+00H) and the address (+01H,+02H) will point to a table with the following structure:

+00H	DB DVINFB	(optional)
+01H	DB DVTYPE	(optional)
+02H	DB 0	
+03H	JP INIT	Initialize RS232
+06H	JP OPEN	Opens an RS232 port
+09H	JP STAT	Returns various states
+0CH	JP GETCHR	Read a character
+0FH	JP SNDCHR	Sends a character
+12H	JP CLOSE	Closes an RS232 port
+15H	JP EOF	Checks end of file
+18H	JP LOC	Returns the num. char.
+1BH	JP LOF	Return free space
+1EH	JP BACKUP	Save a character
+21H	JP SNDBRK	Send break characters
+24H	JP DTR	On/off DTR line
+27H	JP SETCHN	Select RS232 channel
+2AH	JP NCUSTA	(MSX Modem)
+2DH	JP SPKCNT	(MSX Modem)
+30H	JP LINSEL	(MSX Modem)
+33H	JP DIALST	(MSX Modem)
+36H	JP DIALCH	(MSX Modem)
+39H	JP DTMFST	(MSX Modem)
+3CH	JP RDDTMF	(MSX Modem)
+3FH	JP HOKCNT	(MSX Modem)
+42H	JP CONFIG	(MSX Modem)
+45H	JP SPCIAL	(MSX Modem)

Registers: All.

9.5.4.1 – Parameter Bytes



DVTYPE – 0 ⟶ Multiple channels.
Another value ⟶ Single channel.

9.5.4.2 – RS232C serial port manipulation routines

INIT (HL+03H/ExtBIOS) – HL value obtained via EXTBIO
Function: Initialize RS232C port.
Input: B – ID of the slot from the parameter table.
HL – Address of the parameter table, with the following structure (from +00H to +07H values must be in ASCII code):
+00H – Character length ("5", "6", "7" or "8")
+01H – Parity ("E", "O", "I" or "N")
+02H – Stop bits ("1", "2" or "3")
+03H – XON/XOFF ("X" or "N")
+04H – CTR-RTS hand shake ("H" or "N")
+05H – Auto LF reception ("A" or "N")
+06H – Auto LF transmission ("A" or "N")
+07H – SI/SO Control ("Y" or "N")
+08H – Receive speed (low)
+09H – Receive speed (high) (50 to 19 200 baud)
+0AH – Speed transmission (low)
+0BH – Speed transmission (high) (50 to 19 200 baud)
+0CH – Time counter (0 to 255)
Output: CY = 0 ⟶ RS232C successfully started.
1 ⟶ Parameter error.
Registers: AF.

OPEN (HL+06H/ExtBIOS) – HL value obtained via EXTBIO
Function: Opens an RS232C serial port using FCB.
Input: HL – FCB initial address (greater than 8000H).
C – Buffer size (32 to 254).
E – Open mode:
0 – Entry
2 – Exit
4 – Input/Output and RAW mode
Output: CY = 0 ⟶ Door opened successfully.
1 ⟶ Error in the opening process.
Registers: AF.

STAT (HL+09H/ExtBIOS) – HL value obtained via EXTBIO

Function: Returns status or error data.

Input: None.

Output: HL – Data returned.

Bit 15: 0 – No buffer error. 1 – Buffer overflow.
Bit 14: 0 – No timing error. 1 – Time out.
Bit 13: 0 – Correct framing. 1 – Framing error.
Bit 12: 0 – Correct execution.
1 – Execution error (overrun error).
Bit 11: 0 – No parity error.
1 – Character parity error.
Bit 10: 0 – CTRL+STOP are not pressed.
1 – CTRL+STOP pressed together.
Bit 09: Reserved.
Bit 08: Reserved.
Bit 07: 0 – Clear to Send state is false.
1 – Clear to Send state is true.
Bit 06: 0 – Timer/Counter-2 not confirmed.
1 – Timer/Counter-2 confirmed.
Bit 05: Reserved.
Bit 04: Reserved.
Bit 03: 0 – Data Set Ready state is false.
1 – Data Set Ready state is true.
Bit 02: 0 – Stop not detected.
1 – Stop detected.
Bit 01: 0 – Ring indicator state is false.
1 – Touch indicator state is true.
Bit 00: 0 – Carrier not detected.
1 – Carrier detected.

GETCHR (HL+0CH/ExtBIOS) – HL value obtained via EXTBIO

Function: Returns a character from the receive buffer.

Input: None.

Output: A – Character received.
CY = 1 ⟶ EOF (end of file).
S = 1 ⟶ Error.

Registers: F.

SNDCHR (HL+0FH/ExtBIOS) – HL value obtained via EXTBIO
Function: Sends a character to the RS232C serial port.
Input: A – Character to be sent.
Output: CY = 1 ⟶ CTRL+STOP were pressed together.
Z = 1 ⟶ Error.
Registers: F.

CLOSE (HL+12H/ExtBIOS) – HL value obtained via EXTBIO
Function: Close the RS232C serial port.
Input: None.
Output: CY = 1 ⟶ Error.
Registers: AF.

EOF (HL+15H/ExtBIOS) – HL value obtained via EXTBIO
Function: Checks for end of file.
Input: None.
Output: HL = –1 and CY = 1 ⟶ Next character is EOF (End of file).
HL = 0 and CY = 0 ⟶ Not end of file.
Registers: AF.

LOC (HL+18H/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the number of characters in the receive buffer.
Input: None.
Output: HL – Number of characters in buffer.
Registers: AF.

LOF (HL+1BH/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the free space in the receive buffer.
Input: None.
Output: HL – Free space in bytes.
Registers: AF.

BACKUP (HL+1EH/ExtBIOS) – HL value obtained via EXTBIO
Function: Saves a character in a special buffer. The previous character is lost.
Input: C – Character to be saved.
Output: None.
Registers: F.

SNDBRK (HL+21H/ExtBIOS) – HL value obtained via EXTBIO
Function: Sends the specified number of "break" characters.
Input: DE – Number of "break" characters to be sent.
Output: CY = 1 ⟶ CTRL+STOP were pressed together.
Registers: AF, DE.

DTR (HL+24H/ExtBIOS) – HL value obtained via EXTBIO
Function: Turns the DTR line on/off.
Input: A = 0 ⟶ Disconnect the DTR line.
A ≠ 0 ⟶ Connects the DTR line.
Output: None.
Registers: F.

SETCHN (HL+27H/ExtBIOS) – HL value obtained via EXTBIO
Function: Select the channel number (only for multi-channel interfaces).
Input: A – Channel number.
Output: CY = 1 ⟶ The channel does not exist on the interface.
Registers: AF, BC.

9.5.4.3 – MSX Modem manipulation routines

INIT (HL+03H/ExtBIOS) – HL value obtained via EXTBIO
Function: Initializes MSX Modem.
Input: A – modem type.
0 – BELL 103 300 bps full duplex
1 – BELL 212 A 1200 bps full duplex
2 – CCITT V 21 300 bps full duplex
3 – CCITT V 22 1200 bps full duplex
4 – CCITT V22bis 2400 bps full duplex
5 – CCITT V 23 1200 bps half duplex
6 – CCITT V27ter 4800 bps half duplex
7 – CCITT V 29 9600 bps half duplex
8 – CCITT V32 9600 bps full duplex
9 to 254 – Reserved for future expansions.
255 – System default.

C – Dialing mode:
0 – DTMF (tone prompting)
1 – Reserved for future expansions.
2 – Pulses (20 pps)
3 – Pulses (10 pps)
4 – Automatic
5 to 254 – Reserved for future expansions.
255 – System default.
B – ID of the slot from the parameter table.
HL – Address of the parameter table, with the following structure (from +00H to +07H values must be in ASCII code):
+00H – Character length ("5", "6", "7" or "8")
+01H – Parity ("E", "O", "I" or "N")
+02H – Stop bits ("1", "2" or "3")
+03H – XON/XOFF ("X" or "N")
+04H – CTR-RTS hand shake ("H" or "N")
+05H – Auto LF reception ("A" or "N")
+06H – Auto LF transmission ("A" or "N")
+07H – SI/SO Control ("Y" or "N")
+08H~0BH – Not used
+0CH – Time counter (0 to 255)

Output: CY = 0 ⟶ MSX Modem successfully started.
1 ⟶ Parameter error.

Registers: AF.

NCUSTA (HL+2AH/ExtBIOS) – HL value obtained via EXTBIO

Function: Returns NCU status.

Input: None.

Output: HL – State.
bit 15~bit 9 – Always 0.
bit 8: 0 – No DTMF data.
1 – Receiving DTMF data
bit 7: 0 – External telephone on hook
1 – External telephone off-hook
bit 6: 0 – No ringing tone
1 – 400 Hz ring tone detected
bit 5: locks line polarity inversion

b4,b3: 00 – Loop off
01 – DC loop (LB)
10 – DC loop (LA)
11 – Undefined
b2,b1: dialing mode
00 – DTMF
01 – Pulse (10 pps)
10 – Pulse (20 pps)
11 – Automatic
bit 0: 0 – No bell signal (ring)
1 – Bell signal (ring) present

Registers: All.

SPKCNT (HL+2DH/ExtBIOS) – HL value obtained via EXTBIO
Function: Turns the speaker on/off.
Input: A = 0 ⟶ Turns off the speaker.
A ≠ 0 ⟶ Turns on the speaker.
Output: CY = 1 if this function is not supported.
Registers: F.

LINSEL (HL+30H/ExtBIOS) – HL value obtained via EXTBIO
Function: Switch the line.
Input: A – bit 7~5 – Reserved (always 0).
b4~b3 – releases the line (puts the internal phone on the "hook"). Bit4 releases the speaker and bit3 releases the microphone).
b2~b1 – connect the built-in telephone to the modem to the outside line (bit2 = 1, connect speaker, bit1 = 1, connect microphone).
bit 0 – Switches between modem and external telephone:
b0 = 0 ⟶ connect the internal modem;
b0 = 0 ⟶ connect the telephone connected to the "TEL" port of the modem.
Output: CY = 1 if there is an error in the parameters.
Registers: None.

DIALST (HL+33H/ExtBIOS) – HL value obtained via EXTBIO
Function: Connect the device to the line and "dial".

Input: C – Dial mode:
0 – DTMF (tone dialing)
1 – Reserved for future expansions.
2 – Pulses (20 pps)
3 – Pulses (10 pps)
4 – Automatic
5 to 254 – Reserved for future expansions.
255 – System default.
B – ID of the slot from the parameter table.
HL – Starting address of the dial data to be sent. Valid characters for "dial" are: "0"~"9", "A"~"D", "#", "*", "H", "<", ":" and "T". "H" means 1 second on-hook, "<" means three, "T" selects tone dialing and ":" waits for second dial tone. The data list must end with a 00H byte.
Output: CY = 1 if there is an error in the parameters.
Registers: None.

DIALCH (HL+36H/ExtBIOS) – HL value obtained via EXTBIO
Function: Sends a single character at a time to "dial".
Input: A – character to be sent.
C – dial mode (same as DIALST(HL+33H)).
Output: CY = 1 if there is an error in the parameters.
Registers: None.

DTMFST (HL+39H/ExtBIOS) – HL value obtained via EXTBIO
Function: Checks the status of the DTMF decoder.
Input: None.
Output: Z = 1 if DTMF code is in input mode.
CY = 1 if this function is not supported.
Registers: AF.

RDDTMF (HL+3CH/ExtBIOS) – HL value obtained via EXTBIO
Function: Read data from DTMF decoder.
Input: None.
Output: A – DTMF Code (in ASCII)
CY = 1 if CTRL+STOP are pressed together or if this function is not supported.
Registers: AF.

HOKCNT (HL+3FH/ExtBIOS) – HL value obtained via EXTBIO
Function: Connect or disconnect the line.
Input: A – 0 = On hook
1 = Off the hook
Output: CY = 1 if this function is not supported.
Registers: None.

CONFIG (HL+42H/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns hardware specifications.
Input: A – 0 to 255.
Output: HL – Specifications.
• When A = 0:
bit 15~09: always 0
bit 8 = 1 ⟶ CCITT V 32 9600 bps full duplex
bit 7 = 1 ⟶ CCITT V 29 9600 bps half duplex
bit 6 = 1 ⟶ CCITT V 27ter 4800 bps half duplex
bit 5 = 1 ⟶ CCITT V 23 1200 bps half duplex
bit 4 = 1 ⟶ CCITT V 22a 2400 bps full duplex
bit 3 = 1 ⟶ CCITT V 22 1200 bps full duplex
bit 2 = 1 ⟶ CCITT V 21 300 bps full duplex
bit 1 = 1 ⟶ BELL 212 At 1200 bps full duplex
bit 0 = 1 ⟶ BELL 103 300 bps full duplex
• When A = 1:
bit 15~08: always 0
bit 7 = 1 ⟶ support 10pps↔20pps change by software
bit 6 = 1 ⟶ DTMF – Soft pulse switching
bit 5 = 1 ⟶ supports "H"
bit 4 = 1 ⟶ support for "A" to "D"
bit 3 = 1 ⟶ automatic
bit 2 = 1 ⟶ pulse (20 pps)
bit 1 = 1 ⟶ pulse (10 pps)
bit 0 = 1 ⟶ DTMF
• When A = 2:
bit 15~8: always 0
bit 7 = 1 ⟶ support 10pps↔20pps change by software
bit 6~4: always 0
bit 3 = 1 ⟶ integrated handsfree phone
bit 2 = 1 ⟶ built-in standard telephone
bit 1 = 1 ⟶ internal modem
bit 0 = 1 ⟶ external telephone

• When A = 3:
bit 15~bit 13: always 0
bit 12 = 1 ⟶ long loop detection function
bit 11 = 1 ⟶ carrier control function
bit 10 = 1 ⟶ transmission power switching function
bit 9 = 1 ⟶ RS-232C
bit 8 = 1 ⟶ standard MSX cartridge
bit 7 = 1 ⟶ external telephone hook detection
(on-hook or off-hook).
bit 6 = 1 ⟶ "on hook" / "off hook" function
bit 5 = 1 ⟶ has speaker
bit 4 = 1 ⟶ has DTMF decoder
bit 3 = 1 ⟶ charging pulse detection
bit 2 = 1 ⟶ line polarity detection
bit 1 = 1 ⟶ call progress detection
bit 0 = 1 ⟶ touch signal detection
• When A is 4 to 255:
HL = 0000H

Registers: HL.

SPCIAL (HL+45H/ExtBIOS) – HL value obtained via EXTBIO

Function: Implements special functions for each modem model.

Input: A = 0 ⟶ Send modem power switching function.
C – transmission power value (dBm). If it is 255, it defaults to value.
A = 1 ⟶ Carrier wave control.
C – 0 – Turns off the carrier.
1 – Turns on the carrier.
H – delay time up to RS ON (n * 10 mS)
L – delay time from CS ON to RETURN (n* 10 mS)
A = 2 ⟶ Equalizer setting.
C = 0 – Do not use equalizer.
1 – Use the equalizer.
2 – Automatic equalizer adjustment
255 – Defaults.

Output: CY = 1 if selected function is not supported.

Registers: Depends on the called function.

9.5.5 – MSX-AUDIO

EXTBIO (FFCAH/Work Area)

Function: Access extended BIOS functions

Input:
- A – 00H.
- D – 0AH – MSX-Audio manipulation device.
- E – 00H – Returns the pointer to the MSX-Audio information table.
- B – Address table slot ID.
- HL – Address of a 64-byte buffer for the table (should be on page 3).

Output:
- B – ID of the information table slot.
- HL – HL is incremented by 4 and will point to the end of a table that reserves 4 bytes for MSX-Audio. The original HL value points to the beginning of the table, which has the following structure:
  - +00H – Slot ID
  - +01H – Lowest address
  - +02H – Highest address
  - +03H – Reserved for expansion

  The slot ID (+00H) and the address (+01H,+02H) will point to a table with the following structure:

+00H	VERSION	Software version
+03H	MBIOS	Music BIOS
+06H	AUDIO	Initialization of MSX-Audio
+09H	SYNTHE	Calls the SYNTHE app
+0CH	PLAYF	State instruction PLAY
+0FH	BGM	Enable/cancel BGM mode
+12H	MKTEMP	Set recording time / musical keyboard playback
+15H	PLAYMK	Plays via musical keyboard
+18H	RECMK	Records the notes played on the musical keyboard
+1BH	STOPM	Keyboard playback / recording / ADPCM; stops command PLAY
+1EH	CONTMK	Continue recording by musical keyboard
+21H	RECMOD	Sets recording mode of the musical keyboard

+24H	STPPLY	Stops the PLAY instruction
+27H	SETPCM	Protected Area ADPCM/PCM
+2AH	RECPCM	ADPCM/PCM Recording
+2DH	PLAYPCM	ADPCM/PCM Playback
+30H	PCMFREQ	Changing the frequency of ADPCM/PCM playback
+33H	MKPCM	Set/cancel data ADPCM for musical keyboard
+36H	PCMVOL	Sets the volume of ADPCM/PCM playback
+39H	SAVEPCM	Save ADPCM/PCM data
+3CH	LOADPCM	Load ADPCM/PCM data
+3FH	COPYPCM	Transfers ADPCM/PCM data
+42H	CONVP	Converts ADPCM to PCM data
+45H	CONVA	Converts PCM to ADPCM data
+48H	VOICE	Sets FM data
+4BH	VOICECOPY	Moves FM data

Registers: F.

EXTBIO (FFCAH/Work Area)
Function: Access extended BIOS functions
Input: A – 00H.
D – 0AH – MSX-Audio manipulation device.
E – 01H – Returns how many MSX-Audio cartridges are connected to the MSX (maximum 2).
Output: A – 0 ⟶ There is no MSX-Audio connected.
1 ⟶ There is an MSX-Audio cartridge connected.
2 ⟶ There are two MSX-Audio cartridges connected.
Registers: BC, DE, HL.

9.5.5.1 – Startup routines

VERSION (HL+00H) – HL value obtained via EXTBIO
Function: BIOS version. Usually 00H-00H-00H.

MBIOS (HL+03H) – HL value obtained via EXTBIO
Function: Call the MBIOS routines (Music BIOS).

Input: HL – Address of the MBIOS routine.
IX and IY are used for interslot calling and must be defined in BUF (F55EH) as follows:
BUF +00H/+01H – IX
BUF +02H/+03H – IY
Output: Depends on MBIOS routine.
Registers: It depends on the MBIOS routine.

AUDIO (HL+06H) – HL value obtained via EXTBIO
Function: Initialize MSX-Audio.
Input: Set the following values in BUF (F55EH):
+01H – Switch mode
+02H – Number of FM instruments used to configure MSX-Audio (0 to 9)
+03H – Number of FM sound sources for the first string (0 to 9)
+04H – Number of FM sound sources for the second string (0 to 8)
+05H – Number of FM sound sources for the third string (0 to 7)
+06H – Number of FM sound sources for the fourth string (0 to 6)
+07H – Number of FM sound sources for the fifth string (0 to 5)
+08H – Number of FM sound sources for the sixth string (0 to 4)
+09H – Number of FM sound sources for the seventh string (0 to 3)
+0AH – Number of FM sound sources for the eighth string (0 to 2)
+0BH – Number of FM sound sources for the ninth string (0 to 1)
Output: CY = 1 ⟶ Initialization failed.
Registers: All.

SYNTHE (HL+09H) – HL value obtained via EXTBIO
Function: Calls the built-in SYNTHE application.
Input: None.
Output: None.
Registers: All.

9.5.5.2 – PCM/ADPCM Routines

SETPCM (HL+27H) – HL value obtained via EXTBIO
Function: Initializes the audio file for PCM/ADPCM.
Input: Set the parameters in BUF (F55EH):
+00H – Audio file number (0 to 15).
+01H – Device number (0 to 5, except 4).
0 or 2 ⟶ external RAM
1 or 3 ⟶ external ROM
4 ⟶ CPU (cannot be used)
5 ⟶ VRAM
+02H – Mode (0 or 1).
+03H/+04H – Depends on the device number:
RAM: No need to define
ROM: +3H – File number. audio in ROM
+4H – Always 0.
VRAM: +3H – VRAM Address (LSB)
+4H – VRAM Address (MSB)
+05H/+06H – Length (LSB-MSB).
+07H/+08H – Sampling frequency (LSB-MSB).
+09H – Channel number (0 or 1).
Output: CY = 1 ⟶ Parameter error, not configured.
Registers: All.

RECPCM (HL+2AH) – HL value obtained via EXTBIO
Function: Record audio file.
Input: Set the following parameters in BUF (F55EH):
+00H – Audio file number (0 to 15).
+01H – Synchronization (0 or 1).
+02H/+03H – Displacement (LSB-MSB).
+04H/+05H – Length (LSB-MSB). FFFFH to use values defined by SETPCM (HL+27H).
+06H/+07H – Sampling frequency (LSB-MSB). FFFFH to use values defined by SETPCM (HL+27H).
+08H – Channel number (0 or 1). FFH to use channel defined by SETPCM (HL+27H).
Output: CY = 1 ⟶ Parameter error, write cancelled.
Registers: All.

PLAYPCM (HL+2DH) – HL value obtained via EXTBIO
Function: Play audio file.
Input: Set the parameters in BUF (F55EH):
+00H – Audio file number (0 to 15).
+01H – Repeat flag (0 or 1).
+02H/+03H – Displacement (LSB-MSB).
+04H/+05H – Length (LSB-MSB). FFFFH to use values defined by SETPCM (HL+27H).
+06H/+07H – Sampling frequency (LSB-MSB). FFFFH to use values defined by SETPCM (HL+27H).
+08H – Channel number (0 or 1). FFH to use channel defined by SETPCM (HL+27H).
Output: CY = 1 ⟶ Parameter error, operation cancelled.
Registers: All.

PCMFREQ (HL+30H) – HL value obtained via EXTBIO
Function: Change the playback frequency.
Input: BC – First channel sampling frequency
DE – First channel sampling frequency
The frequency can vary from 1800 to 49,716 Hz. If there is no second channel, set DE value equal to BC.
Output: CY = 1 ⟶ Parameter error. The frequency is not changed.
Registers: All.

PCMVOL (HL+36H) – HL value obtained via EXTBIO
Function: Sets the PCM/ADPCM playback volume.
Input: BC – Volume of the first channel (0 to 63), where 63 is max.
DE – First channel volume (0 to 63), where 63 is max.
The initial value is 63 for ADPCM and 32 for PCM. If there is no second channel, set DE value equal to BC.
Output: CY = 1 ⟶ Parameter error. Volume is not set.
Registers: All.

SAVEPCM (HL+39H) – HL value obtained via EXTBIO
Function: Save PCM/ADPCM audio file to disk.
Input: A – Audio file number.
HL – Pointer to the filename. It must be enclosed in double quotes (22H) and end with byte 00H (Ex. "FILENAME.PCM",00H), as in MSX-BASIC.

Output: CY = 1 ⟶ Wrong audio file number. The file will not be saved.
Registers: All.
Note: If there are any errors during the save, control will be returned to the BASIC interpreter.

LOADPCM (HL+3CH) – HL value obtained via EXTBIO
Function: Load PCM/ADPCM audio file from disk.
Input: A – Audio file number.
HL – Pointer to the filename. It must be enclosed in double quotes (22H) and end with byte 00H (Ex. "FILENAME.PCM",00H), as in MSX-BASIC.
Output: CY = 1 ⟶ Wrong audio file number. The file will not be loaded.
Registers: All.
Note: If there are any errors during loading, control will be returned to the BASIC interpreter.

COPYPCM (HL+3FH) – HL value obtained via EXTBIO
Function: Transfer PCM/ADPCM data between audio files.
Input: Set the parameters in BUF (F55EH):
+00H – Source file number (0 to 15).
+01H – Destination file number (0 to 15).
+02H/+03H – Offset of source file (LSB-MSB).
+04H/+05H – Length (LSB-MSB).
+06H/+07H – Offset destination file (LSB-MSB).
+08H – Font specification (0 or 1).
Output: CY = 1 ⟶ Parameter error, transfer cancelled.
Registers: All.

CONVP (HL+42H) – HL value obtained via EXTBIO
Function: Convert data from PCM format to ADPCM.
Input: Set the parameters in BUF (F55EH):
+00H – Source file number (0 to 15).
+01H – Destination file number (0 to 15).
Output: CY = 1 ⟶ Parameter error, conversion cancelled.
Registers: All.

CONVA (HL+45H) – HL value obtained via EXTBIO
Function: Convert data from ADPCM format to PCM.
Input: Set the parameters in BUF (F55EH):
+00H – Source file number (0 to 15).
+01H – Destination file number (0 to 15).
Output: CY = 1 ⟶ Parameter error, conversion cancelled.
Registers: All.

MKTEMPO (HL+18H) – HL value obtained via EXTBIO
Function: Sets the time for recording and playback through the musical keyboard, with metronome function.
Input: DE – Time in quarter notes per minute (25 to 360).
Output: CY = 1 ⟶ Parameter error, configuration cancelled.
Registers: All.

MKPCM (HL+33H) – HL value obtained via EXTBIO
Function: Specify the ADPCM sound file to play with the musical keyboard.
Input: A – Audio file number (0 to 15). To cancel, use FFH.
Output: CY = 1 ⟶ Parameter error, playback cancelled.
Registers: All.

9.5.5.3 – Musical keyboard routines

PLAYMK (HL+15H) – HL value obtained via EXTBIO
Function: Plays recorded audio via musical keyboard.
Input: DE – Starting address of reproduction.
BC – Final address of reproduction.
Output: None.
Registers: All.

RECMK (HL+18H) – HL value obtained via EXTBIO
Function: Records audio through the musical keyboard.
Input: DE – Starting address for recording.
BC – Final address for recording.
Output: None.
Registers: All.

CONTMK (HL+1EH) – HL value obtained via EXTBIO
Function: Continue recording or playing musical keyboard audio that was interrupted by STOPM.
Input: None.
Output: None.
Registers: All.

RECMOD (HL+21H) – HL value obtained via EXTBIO
Function: Sets the recording mode for the musical keyboard.
Input: A = 0 ⟶ Muting (do not record)
1 ⟶ Record
2 ⟶ Play
3 ⟶ Record and play simultaneously
Output: CY = 1 ⟶ Parameter error, configuration cancelled.
Registers: All.

9.5.5.4 – FM synthesizer routines

PLAYF (HL+0CH) – HL value obtained via EXTBIO
Function: Checks the status of the PLAY instruction.
Input: A – PLAY instruction channel number (0 = All channels).
Output: HL – 0000H ⟶ the specified channel is NOT playing.
FFFFH ⟶ the specified channel is playing (when specified for all channels, HL will return FFFFH if any are active).
Registers: All.

BGM (HL+0FH) – HL value obtained via EXTBIO
Function: Specifies background execution.
Input: 0 – Does NOT perform background processing.
1 – Runs background processing (default). The functions available for the background are: playback via the PLAY command, ADPCM recording and playback via microphone, and recording and playback via the musical keyboard.
Output: None.
Registers: All.

STOPM (HL+1BH) – HL value obtained via EXTBIO
Function: Stop playback and recording.
Input: None.
Output: None.
Registers: All.

STPPLY (HL+1BH) – HL value obtained via EXTBIO
Function: Stop playback of PLAY command only.
Input: None.
Output: None.
Registers: All.

VOICE (HL+48H) – HL value obtained via EXTBIO
Function: Sets the instrument for each FM channel.
Input: Define the following parameters in BUF (F55EH):
+0 ⟶ Voice 1 parameter block
+4 ⟶ Voice 2 Parameter Block
⋮
(n–1)*4 ⟶ Voice n parameter block
n*4 ⟶ End mark (FFH).
• Specifying instruments provided in ROM:
+0 ⟶ Channel number (0 to 8).
+1 ⟶ 00H.
+2 ⟶ Instrument number in ROM (0 to 63).
+3 ⟶ 00H.
• Specifying user instrument:
+0 ⟶ Channel number (0 to 8).
+1 ⟶ FFH.
+2/+3 ⟶ Instrument data address (LSB-MSB).
Output: CY = 1 ⟶ Parameter error, configuration cancelled.
Registers: All.

VOICECOPY (HL+4BH) – HL value obtained via EXTBIO
Function: Transfers data from FM instruments.
Input: Define the following parameters in BUF (F55EH):
• Transfer 0~63 instruments from ROM to 32~63 system instruments:
+0 ⟶ 00H
+1 ⟶ Source instrument number (0~63).
+2~+5 ⟶ 00H

+6 ⟶ Target instrument number (32~63).
+7~+9 ⟶ 00H
• Transfer 0~63 instruments from ROM to user data area:
+0 ⟶ 00H
+1 ⟶ Source instrument number (0~63).
+2~+4 ⟶ 00H
+5 ⟶ FFH
+6~+7 ⟶ Destination address in the data area.
+8~+9 ⟶ 00H
• Transfer instruments from user data area to 32~63 system instruments:
+0 ⟶ FFH
+1~+2 ⟶ Source address in the data area.
+3~+5 ⟶ 00H
+6 ⟶ Target instrument number (32~63).
+7~+9 ⟶ 00H
• Transfer all 32~63 instruments from the system to the user data area:
+0 ⟶ 00H
+1 ⟶ FFH
+2 ~ +4 ⟶ 00H
+5 ⟶ FFH
+6 ~ +7 ⟶ Destination address in the data area.
+8 ~ +9 ⟶ Length of data in bytes.
• Transfers all instruments from the user data area to 32~63 system instruments:
+0 ⟶ FFH
+1 ~ +2 ⟶ Destination address in the data area.
+3 ~ +4 ⟶ Length of data in bytes.
+5 ⟶ 00H
+6 ⟶ FFH
+7 ~ +9 ⟶ 00H.

9.5.5.5 – MBIOS routines (Music BIOS)

The Music BIOS routines must be called through the MBIOS entry of the jump table, setting in HL the call address of the desired Music BIOS routine. The MBIOS format is as follows:

MBIOS (JumpTable+03H) – JumpTable value obtained via EXTBIO

Function: Call the MBIOS routines (Music BIOS).

Input: HL – Address of the MBIOS routine.
IX and IY are used for interslot calling and must be defined in BUF (F55EH) as follows:
BUF+00H/+01H – IX
BUF+02H/+03H – IY

Output: Depends on Music BIOS routine.

Registers: It depends on the Music BIOS routine.

The data tables used by Music BIOS are as follows:

CHDB (32 bytes)

+00	YCAO0_MULTI
+01	YCAO0_LS
+02	YCAO0_AR
+03	YCAO0_RR
+04	YCAO0_VELS
+05	YCAO0_VTL
+06~+07	Unused
+08	YCAO1_MULTI
+09	YCAO1_LS
+10	YCAO1_AR
+11	YCAO1_RR
+12	YCAO1_VELS
+13	YCAO1_VTL
+14~+15	Unused
+16~+17	YCA_VTRANS
+18~+19	YCA_TRANS
+20	YCA_TRIG
+21	YCA_VOL
+22	YCA_FB
+23	YCA_VEL
+24~+25	YCA_PITCH
+26	YCA_VOICE
+27	ZCA_FLAG
+28	ZC_CH
+29	ZC_OP
+30~+31	ZC_COUNT

MIDB (64 bytes)

+00~+01	Unused
+02	YM_TIM 1
+03	YM_TIM 2
+04~+17	Unused
+18	YMA_BIAS
+19~+24	Unused
+25	YMA_AUDIO
+26~+31	Unused
+32~+33	YMA_TRANS
+34	YMA_LFO
+35	YMA_RAM
+36	ZMA_FLAG
+37~+38	YMA_PDB
+39	ZMA_PH_FILTER
+40	ZMA_PH_TL
+41~+42	ZMA_PH_AR
+43~+44	ZMA_PH_DIR
+45	ZMA_PH_SL
+46~+47	ZMA_PH_D 2 R
+48~+49	ZMA_PH_RR
+50~+51	ZMA_PH_EG
+52	ZMA_PH_STAT
+53~+63	Unused

PDB (PCM Data Block)

```
+00     PDB_DEV    Defines the PCM/ADPCM device
+01     Unused
+02~+03 PDB_ADDR   Data start address
+04~+05 PDB_SIZE   Data Block Size
+06~+07 PDB_SAMPLE Sampling frequency
                   (1800 to 16000 for ADPCM
                   or 1800 to 12000 for PCM)
+08~+09 PDB_PCM    Initial value when ADPCM is
                   tracked and converted to PCM.
+10~+11 PDB_STEP   Initial Quantize Width
                   when ADPCM is tracked and
                   converted to PCM
+12~+15 Unused
```

The Music BIOS routines are as follows:

SV_RESET (0090H/MBIOS)
Function: Initialize the MBIOS.
Input: None.
Output: None.
Registers: None.
Note: Interruptions are disabled on return. Before re-enabling them, the MBIOS hook must be set.

SV_DI (0093H/MBIOS)
Function: Disables user interrupts.
Input: None.
Output: None.
Registers: None.

SV_EI (0096H/MBIOS)
Function: Allow user interrupts.
Input: None.
Output: None.
Registers: None.

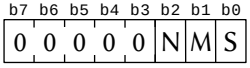
SV_ADW (0099H/MBIOS)
Function: Write a byte of data into a Y8950 register.
Input: IY – Master/slave specification by MIDB address.
A – Byte of data to be written.
C – Registrar number.

Output: CY = 1 ⟶ there was an attempt to write to a non-existent slave device.
Registers: All.

SV_ADW_DI (009CH/MBIOS)
Function: Write a data byte in a Y8950 register, disabling loopback interrupts.
Input: IY – Master/slave specification by MIDB address.
A – Byte of data to be written.
C – Register number.
Output: CY = 1 ⟶ there was an attempt to write to a non-existent slave device.
IFF = 0 (Interrupts Disabled)
Registers: All.

SV_SETUP (00ABH/MBIOS)
Function: Initial setup of various functions.
Input: A – Function code.
0 – SM_AUDIO ⟶ tone setting.
1 – SC_CHDB ⟶ initialize CHDB desktop.
2 – SM_INST ⟶ initialize the instrument function.
3 – SM_MK ⟶ initialize musical keyboard reading.
Other parameters depend on the function.
Output: CY = 1 ⟶ Configuration failed (usually because the routine is called via interrupts).
Registers: All.

SM_AUDIO (00ABH/MBIOS)
Function: Set the FM synthesizer music tone mode.
Input: A – 0.
C –



0 – FM slave mode
1 – CSM slave mode
0 – FM master mode
1 – CSM master mode
0 – 9 channels
1 – 6 chan. + 5 drum pieces

DE – FM synthesizer tone channel.
bit0 = 1 ⟶ instrument in channel 0
bit1 = 1 ⟶ instrument on channel 1
⋮
bit8 = 1 ⟶ instrument on channel 8
(In 6 channel mode + 5 drum pieces, only channels 0 to 5 can be assigned).

Output: None.
Registers: All.
Note: This routine internally calls SC_CHDB and SM_INST.

SC_CHDB (00ABH/MBIOS)
Function: Launches the CHDB desktop.
Input: A – 1.
IX – CHDB address to be initialized.
Output: None.
Registers: All.

SM_INST (00ABH/MBIOS)
Function: Initializes the instrument's tone with timbre #0.
Input: A – 2.
Output: None.
Registers: All.

SM_MK (00ABH/MBIOS)
Function: Initializes music keyboard reading.
Input: A – 3.
B – 1 ⟶ connects the keyboard to the instrument.
0 ⟶ do not connect the keyboard.
C – speed when keys are pressed. 0 is the slowest and 15 is the fastest. Velocity is referenced in SV_MK (musical keyboard scan).
Output: None.
Registers: All.

SV_REAL (00AEH/MBIOS)
Function: Perform real-time operations. This call is divided into several functions designated by codes that are as follows:

00 RM_MOVE_DI – Transfer. ADPCM/PCM data
01 RM_TRACE_DI – ADPCM Data Trace
02 RM_CONV_PCM_DI – ADPCM to PCM Conversion
03 RM_CONV_ADPCM_DI – Conversion PCM→ADPCM
04 RMA_DAC_BIAS – Volume for PCM playback
05 RMA_DAC_DI – PCM Data Playback
06 RMA_ADC_DI – PCM Data Write
07 RMA_ADPCM_BIAS – Configure ADPCM playback
08 RMA_ADPLAY_DI – Playback of ADPCM data
09 RMA_ADREC_DI – Writing ADPCM data
10 RMA_BREAK – Interrupt. playback/recording
11 RMA_ADPLAY – Playback of ADPCM data
12 RMA_ADREC – Writing ADPCM data
13 RMA_PHASE_SET_DI – Converts 256 bytes PCM
14 RMA_PHASE_EG – Configures the envelope
15 RMA_PHASE_EVENT – Pitch Sampling
16 RM_TIMER – Enables/disables interrupt timer.
17 RM_TIM1 – Sets timer 1
18 RM_TIM2 – Sets timer 2
19 RM_TEMPO – Defines the cycle of timer 2
20 RM_DAMP – FM generator stopping force
21 RM_PERC – Plays rhythm sound
22 RMA_MK – Returns musical keyboard status
23 RMA_LFO – Sets vibrato
24 RMA_TRANS – Configures sound transition
25 RMA_CSM_DI – Reproduction of CSM data
26 RM_READ_DI – Transf. 256 bytes ADPCM/PCM
27 RM_WRITE_DI – Transf. 256 bytes ADPCM/PCM
28 RM_UTEMPR – Converts the temperament pitch
29 RM_CTEMPR – Temper setting
30 RM_PITCH – Set current/subsequent pitch
31 RM_TSRAN – Configures sound transposition
32 RC_NOTE – Turns FM voice on/off
33 RC_LEGATO – Turns FM voice legato on/off
34 RC_DAMP – Interrupts FM voice
35 RC_KON – Turns on the FM generator voice
36 RC_LEGATO_ON – Turns on legato FM voice
37 RC_KOFF – Turns off FM generator voice
38 RCA_PARAM – FM real-time configuration

39 RCA_VOICE – Configures voice for the FM channel
40 RCA_VPARAM – Configures voice parameters
41 RCA_VOICEP – Configures voice for the FM channel
42 RMA_ADPLAYLP – Playback ADPCM data repeats
43 RMA_ADPLY_SAMPLE – ADPCM Playback
44 RM_PVEL – Sets rhythm sound speed
⋮
48 RI_DAMP FM – Generator stopping force
49 RI_ALLOFF – Activates all FM channels
50 RI_EVENT – Converts a note
51 RI_PCHB – Setting the pitch position
52 RI_PCHBR – Setting pitch pitch
53 RIA_PARAM – Realtime setting for FM
54 RIA_VOICE – FM Voice Configuration
55 RIA_VPARAM – Voice Definition for FM
56 RIA_VOICEP – Configure tone for FM

Input: A – function code.
Other parameters depend on the called function.
Output: CY = 1 ⟶ error in input parameters.
Other parameters depend on the called function.
Registers: Depends on the called function.

RC_NOTE (00AEH/MBIOS)
Function: Turns on a voice on the FM generator and turns off automatically after a specified time.
Input: A – 32.
IX – CHDB address with FM voice data.
DE – Range (0~32.767). The central value is 15,360 and a semitone corresponds to 256.
C – Speed (0~15). 0 is the slowest and 15 is the fastest.
B – Timer. Turns off when SV_TEMPO is called this number of times.
Output: None.
Registers: All.

RC_LEGATO (00AEH/MBIOS)
Function: Turns on a voice on the FM generator and turns off automatically after a specified time. Unlike RC_NOTE, this function does not start wrapping.

Input: A – 33.
IX – CHDB address with FM voice data.
DE – Range (0~32.767). The central value is 15,360 and a semitone corresponds to 256.
C – Speed (0~15). 0 is the slowest and 15 is the fastest.
B – Timer. Turns off when SV_TEMPO is called this number of times.
Output: None.
Registers: All.

RC_DAMP (00AEH/MBIOS)
Function: Forces the FM voice that is playing to stop.
Input: A – 34.
IX – CHDB address with FM voice data.
Output: None.
Registers: All.

RC_KON (00AEH/MBIOS)
Function: Connects an FM voice.
Input: A – 35.
IX – CHDB address with FM voice data.
DE – Range (0~32.767). The central value is 15,360 and a semitone corresponds to 256.
C – Speed (0~15). 0 is the slowest and 15 is the fastest.
Output: None.
Registers: All.

RC_LEGATO_ON (00AEH/MBIOS)
Function: Connects an FM voice. Unlike RC_KON, this function does not start wrapping.
Input: A – 36.
IX – CHDB address with FM voice data.
DE – Range (0~32.767). The central value is 15,360 and a semitone corresponds to 256.
C – Speed (0~15). 0 is the slowest and 15 is the fastest.
Output: None.
Registers: All.

RC_KOFF (00AEH/MBIOS)
Function: Turns off an FM voice.
Input: A – 37.
IX – CHDB address with FM voice data.
Output: None.
Registers: All.

RCA_PARAM (00AEH/MBIOS)
Function: Adjust real-time parameters for an FM voice.
Input: A – 38.
IX – CHDB address with FM voice data.
C – Offset of the parameter to be adjusted in the CHDB list.
DE – Configuration data.
The data that can be configured with this function are as follows:

YCA_TRANS	YCA_TRIG	YCA_PITCH
YCA_VOL	YCA_VEL	

Output: None.
Registers: All.

RCA_VOICE (00AEH/MBIOS)
Function: Associate an instrument with an FM voice.
Input: A – 39.
IX – CHDB address with FM voice data.
C – Instrument pattern number in ROM (0~63). The available instruments are as follows:

```
0   Piano 1          32  Piano 3
1   Piano 2          33  Electric Piano 2
2   Violin           34  Santool 2
3   Flute 1          35  Brass
4   Clarinet         36  Flute 2
5   Oboe             37  Clavicode 2
6   Trumpet          38  Clavicode 3
7   Pipe Organ 1     39  Koto 2
8   Xylophone        40  Pipe Organ 2
9   Organ            41  PohdsPLA
10  Guitar           42  RohdsPRA
11  Santool 1        43  Orch L
```

```
12 Electric Piano 1 44 Orch R
13 Clavicode 1      45 Synth Violin
14 Harpsicode 1     46 Synth Organ
15 Harpsicode 2     47 Synth Brass
16 Vibraphone       48 Tube
17 Koto 1           49 Shamisen
18 Taiko            50 Magical
19 Engine 1         51 Huwawa
20 UFO              52 Wander Flat
21 Synthesizer bell 53 Hardrock
22 Chime            54 Machine
23 Synthesizer bass 55 Machine V
24 Synthesizer      56 Comic
25 Synth Percussion 57 SE-Comic
26 Synth Rhythm     58 SE-Laser
27 Harm Drum        59 SE-Noise
28 Cowbell          60 SE-Star 1
29 Close Hi-hat     61 SE-Star 2
30 Snare Drum       62 Engine 2
31 Bass Drum        63 Silence
```

Output: None.
Registers: All.

RCA_VPARAM (00AEH/MBIOS)

Function: Adjust parameters of an FM voice.

Input: A – 40.
IX – CHDB address with FM voice data.
C – Offset of the parameter to be adjusted in the CHDB list.
DE – Configuration data.
The data that can be configured with this function are as follows:

```
CAO0_LS         CAO1_LS         YCAO0_MULTI
CAO0_AR         YCAO1_AR        CAO1_MULTI
CAO0_RR         YCAO1_RR        YCA_VTRANS
CAO0_VELS       YCAO1_VELS      YCA_FB
CAO0_VTL        YCAO1_VTL
```

Output: None.
Registers: All.

RCA_VOICEP (00AEH/MBIOS)

Function: Set an FM voice with tone data.

Input: A – 41.

IX – CHDB address with FM voice data.

BC – Pointer to the data, which occupy 32 bytes with the following structure:

```
0~7 V_NAME  Sound name
8~9 V_TRANS Transposition value
10  V_ARG   Several configurations:
    bit7 - Tremolo level:
           0- 1dB; 1- 4,8 dB
    bit6 - Vibrato level:
           0- 7%;  1- 14%
    bit5 - Defines tremolo/vibrato:
           0- no;  1- configure
    bit4 - Defines fixed tone:
           0- normal tone; 1- fixed tone
    bit3~bit1 - Feedback level:
           000 - 0       100 - π/2
           001 - π/16    101 - π
           010 - π/8     110 - 2π
           011 - π/4     111 - 4π
    bit0 - Type of operators connection:
           0- serial; 1- parallel
11~15 - Unused
16   VO0_MULTI - Data to be configured
           for registers 20H (voice 0) to
           35H (voice 1):
    bit7 - Amplitute modulation:
           0- no; 1- yes
    bit6 - Vibrato:
           0- no; 1- yes
    bit5 - EG-TYP (type of envelope):
           0- decaying; 1- sustained
    bit4 - KSR (Key Scale Rate):
           0- no; 1- yes
    bit3~bit0 - Multiple:
           00-½   04-4   08-8   12-12
           01-1   05-5   09-9   13-12
           02-2   06-6   10-10  14-15
           03-3   07-7   11-10  15-15
```

```
17   VO0_TL - Data to be configured for
           registers 40H (voice 0) to 55H
           (voice 1):
     bit7~bit6 - KSL (Key Scale Level):
           00 - 0 dB/octave
           01 - 1,5 dB/octave
           10 - 3 dB/octave
           11 - 6 dB/octave
     bit7~bit6 - Total level:
           bit0 - 0,75 dB
           bit1 - 1,5 dB
           bit2 - 3 dB
           bit3 - 6 dB
           bit4 - 12 dB
           bit5 - 24 dB
18   VO0_AR - Data to be configured for
           registers 60H (voice 0) to 75H
           (voice 1):
     bit7~bit4 - Attack Rate:
            0dB to 96dB: 1111 - 0 mS
                         1110 - 0,2 mS
                         0000 - 2826 mS
            10% to 90%:  1111 - 0 mS
                         1110 - 0,11 mS
                         0000 - 1482 mS
     bit7~bit4 - Decay Rate:
            0dB to 96dB: 1111 - 0 mS
                         1110 - 2,4 mS
                         0000 - 39280 mS
            10% to 90%:  1111 - 0 mS
                         1110 - 0,51 mS
                         0000 - 8212 mS
19   VO0_RR - Data to be configured for
           registers 80H (voice 0) to 95H
           (voice 1):
     bit7~bit4 - Sustain Level:
           bit7 - 24 dB
           bit6 - 12 dB
           bit5 - 6 dB
           bit4 - 3 dB
     bit3~bit0 - Release Rate:
           bit0 - 24 dB
           bit1 - 12 dB
           bit2 - 6 dB
           bit3 - 3 dB
```

```
20   VO0_VELS - Speed sensitivity
        performed by software via MBIOS.
    bit7~bit4 - Unused.
    bit3~bit0 - Sensitivity:
           0000 - Invalid
           0001 - Minimum
           1111 - Maximum
21~23 - Unused.
24   VO1_MULTI (same as VO0_MULTI but
               acts on operator 1)
25   VO1_TL    (same as VO0_TL but acts
               on operator 1)
26   VO1_AR    (same as VO0_AR but acts
               on operator 1)
27   VO1_RR    (same as VO0_RR but acts
               on operator 1)
28   VO1_VELS  (same as VO0_VELS but acts
               on operator 1)
29~31 - Unused.
```

RM_TIMER (00AEH/MBIOS)
Function: Enable/disable timer functions.
Input: A – 16.
C – bit7~bit2 – Unused.
bit1 – Timer 2: 0- Disable; 1- Activate.
bit0 – Timer 1: 0- Disable; 1- Activate.
Output: None.
Registers: All.

RM_TIM1 (00AEH/MBIOS)
Function: Set the value of timer #1.
Input: A – 17.
C – Period with 80 uS step. Corresponds to 20.48 mS when C=0 and 80 uS when C=255.
Output: None.
Registers: All.

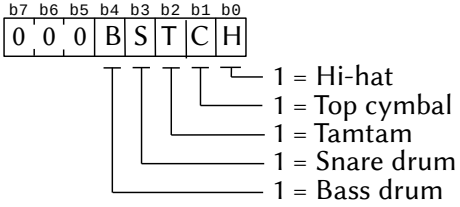
RM_TIM2 (00AEH/MBIOS)
Function: Set the value of timer #2.
Input: A – 18.
C – Period with 80 uS step. Corresponds to 20.48 mS when C=0 and 80 uS when C=255.

Output: None.
Registers: All.

RM_TEMPO (00AEH/MBIOS)
Function: Set timer cycle #2.
Input: A – 19.
C – Number of quarter notes per minute.
Output: None.
Registers: All.

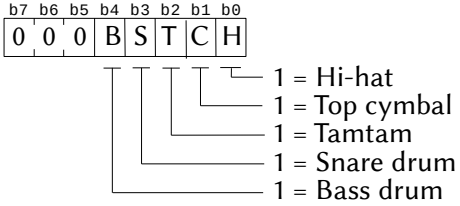
RM_DAMP (00AEH/MBIOS)
Function: Force stop of all active FM generator channels.
Input: A – 20.
Output: None.
Registers: All.

RM_VEL (00AEH/MBIOS)
Function: Sets the speed of the five drum pieces (rhythm). This function can define more than one part at a time.
Input: A – 44.
C –



E – Speed. 0 is the fastest and 31 is the slowest.
Output: None.
Registers: All.

RM_PERC (00AEH/MBIOS)
Function: Activates drum parts sound (rhythm). Several pieces can be played simultaneously.
Input: A – 21.
C –



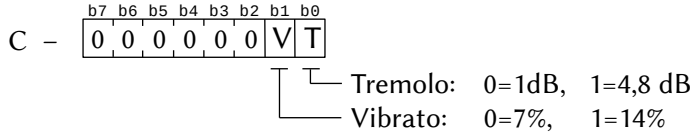
E – Speed. 0 is the fastest and 31 is the slowest.
Output: None.
Registers: All.

RMA_MK (00AEH/MBIOS)
Function: Returns the state of the musical keyboard.
Input: A – 22.
DE – Pointer to a 9-byte buffer.
IY – MIDB pointer indicating master/slave.
Output: The buffer pointed to by DE contains the following structure, where a pressed key corresponds to a set bit:

	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0
0 →	0	C	B	A#	0	A	G#	G
1 →	0	F#	F	E	0	D#	D	C#
2 →	0	C	B	A#	0	A	G#	G
3 →	0	F#	F	E	0	D#	D	C#
4 →	0	C	B	A#	0	A	G#	G
5 →	0	F#	F	E	0	D#	D	C#
6 →	0	C	B	A#	0	A	G#	G
7 →	0	F#	F	E	0	D#	D	C#
8 →	0	C	0	0	0	0	0	0

Note: The note "C" (do) of the byte 8 corresponds to the second octave and "C" of the byte 0 corresponds to the sixth octave.
Registers: All.

RMA_LFO (00AEH/MBIOS)
Function: Sets vibrato and tremolo levels.
Input: A – 23.



IY – MIDB pointer indicating master/slave.
Output: None.
Registers: All.

RMA_TRANS (00AEH/MBIOS)
Function: Sets the transition from the current tone to the subsequent tone.
Input: A – 24.
DE – Transposition value, in units corresponding to 1% of 100/256 (~0.0039).

IY – MIDB pointer indicating master/slave.
Output: None.
Registers: All.

RM_UTEMPR (00AEH/MBIOS)
Function: Converts the temper tone to the set temper tone.
Input: A – 28.
D – Interval (The middle do (C) note is 60).
Output: DE – Converted tone.
Registers: All.

RM_CTEMPR (00AEH/MBIOS)
Function: Selects the temper.
Input: A – 29.
C – Temper:
00 – Pythagoras
01 – Meanone
02 – Werk Meister
03 – Werk Meister (modified)
04 – Werk Meister (another)
05 – Kirunker
06 – Kirunberger (modified)
07 – Valory Young
08 – Lamoo
09 – Balanced temperament (initial value)
10 – C (C minor)
11 – C# (C major)
12 – D (D minor)
13 – D# (D major)
14 – E (mi)
15 – F (F minor)
16 – F# (F major)
17 – G (minor G)
18 – G# (greater sun)
19 – A (minor A)
20 – A# (major)
21 – B (sol)
Output: None.
Registers: All.

RM_PITCH (00AEH/MBIOS)
Function: Adjusts the pitch of the current and subsequent notes. The pitch must be in the range 410~459, the initial value is 440.
Input: A – 30.
BC – Master channel.
DE – Slave channel.
Output: None.
Registers: All.

RM_TRANS (00AEH/MBIOS)
Function: Sets the transition between the current tone and the subsequent tone. The transition value must be in the range –12,799 to +12,799 where the initial value is 0 and the values are in hundredths.
Input: A – 31.
BC – Master channel.
DE – Slave channel.
Output: None.
Registers: All.

RI_DAMP (00AEH/MBIOS)
Function: Force all FM channels to stop.
Input: A – 48.
Output: None.
Registers: All.

RI_ALLOFF (00AEH/MBIOS)
Function: Turns off all assigned FM channels.
Input: A – 49.
Output: None.
Registers: All.

RI_EVENT (00AEH/MBIOS)
Function: Convert pitch and turn FM voice on or off.
Input: A – 50.
D – On: interval + 80H (central value: 60).
Off: interval (central value: 60).
Output: None.
Registers: All.

RI_PCHB (00AEH/MBIOS)
Function: Sets the pitch bender position.
Input: A – 51.
DE – Pitch bender position (the 16 bits are valid in 2's complement, where 7FFFH defines the highest position, 0 the center and 8000H the lowest position)
Output: None.
Registers: All.
Note: This function calls RCA_PARAM (38) internally.

RI_PCHBR (00AEH/MBIOS)
Function: Sets the degree to which the pitch bender will give the pitch.
Input: A – 52.
C – Degree (0 to 12 times).
Output: None.
Registers: All.

RIA_PARAM (00AEH/MBIOS)
Function: Adjusts real-time parameters for active FM voice. The parameters that can be adjusted by this function are YCA_TRANS and YCA_VOL.
Input: A – 53.
IY – MIDB pointer indicating master/slave.
C – Parameter offset in CHDB.
DE – Configuration parameters.
Output: None.
Registers: All.

RIA_VOICE (00AEH/MBIOS)
Function: Assigns an instrument number to an FM channel.
Input: A – 54.
IY – MIDB pointer indicating master/slave and for the FM voice to be assigned.
C – Instrument number (0 to 63).
Output: None.
Registers: All.

RIA_VPARAM (00AEH/MBIOS)

Function: Sets the parameters of an FM channel.

Input: A – 55.
IY – MIDB pointer indicating master/slave and for the FM voice to be assigned.
C – Parameter offset in CHDB.
DE – Configuration parameters.
The following parameters can be set by this function:

CAO0_LS	CAO1_LS	YCAO0_MULTI
CAO0_AR	YCAO1_AR	CAO1_MULTI
CAO0_RR	YCAO1_RR	YCA_VTRANS
CAO0_VELS	YCAO1_VELS	YCA_FB
CAO0_VTL	YCAO1_VTL	

Output: None.

Registers: All.

RIA_VOICEP (00AEH/MBIOS)

Function: Sets an instrument to an FM channel.

Input: A – 56.
IY – MIDB pointer indicating master/slave and for the FM voice to be defined.
BC – Instrument data address.

Output: None.

Registers: All.

RM_MOVE_DI (00AEH/MBIOS)

Function: Transfer PCM/ADPCM data between devices.

Input: A – 0.
IX – PDB address indicating the origin. The following fields are relevant:
PDB_DEV (Device Number)
PDB_ADDR (Starting address)
PDB_SIZE (Transfer Data Size)
IY – PDB address indicating the destination. The following fields are relevant:
PDB_DEV (Device Number)
PDB_ADDR (Starting address)

Output: CY = 1 ⟶ transfer error.

Registers: All.

RM_READ_DI (00AEH/MBIOS)
Function: Transfers 256 bytes of PCM/ADPCM data to RAM.
Input: A – 26
DE – Destination address in RAM.
IX – PDB address indicating the origin. The following fields are relevant:
PDB_DEV (Device Number)
PDB_ADDR (Starting address)
Output: CY = 1 ⟶ transfer error.
Registers: All.

RM_WRITE_DI (00AEH/MBIOS)
Function: Transfers 256 bytes of data from RAM to PCM/ADPCM.
Input: A – 27.
DE – Source address in RAM.
IX – PDB address indicating the destination. The following fields are relevant:
PDB_DEV (Device Number)
PDB_ADDR (Starting address)
Output: CY = 1 ⟶ transfer error.
Registers: All.

RM_TRACE_DI (00AEH/MBIOS)
Function: Track ADPCM data based on initial prediction value and quantize width to find the next predicted value and next quantize width.
Input: A – 1.
C – Mode to start tracking:
0 – initial forecast at 8000H and quantization width at 007FH. The following data must be specified in the PDB:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (transfer size)
1 – initial prediction and quantization width specified in the PDB. In addition to the data for C=0, the following must also be specified:
PDB_PCM (initial predicted value)
PDB_STEP (initial quantization width)

Output: The following fields return valid in the PDB:
PDB_ADDR (next start address)
PDB_PCM (next expected value)
PDB_STEP (next quantization width)
If CY = 1, there was an error in the trace.
Registers: All.

RM_CONV_PCM_DI (00AEH/MBIOS)
Function: Convert ADPCM data to PCM data based on initial prediction value and quantize width.
Input: A – 2.
C – Mode to start tracking:
0 → initial forecast at 8000H and quantization width at 007FH.
1 → initial prediction and quantization width specified in the PDB.
IX – Source PDB address with ADPCM data. The following fields must be completed:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (conversion value)
PDB_SAMPLE (sampling frequency)
• If C=1, also fill in:
PDB_PCM (initial predicted value)
PDB_STEP (initial quantization width)
IY – Destination PDB address with PCM data. The following fields must be completed:
PDB_DEV (device number)
PDB_ADDR (initial address)
Output: The following fields return valid in the source PDB:
PDB_PCM (next expected value)
PDB_STEP (next quantization width)
• If CY = 1, there was an error in the conversion.
Registers: All.

RM_CONV_ADPCM_DI (00AEH/MBIOS)
Function: Convert PCM data to ADPCM data based on initial prediction value and quantize width for ADPCM data.

Input: A – 3.
C – Mode to start tracking:
0 → initial forecast at 8000H and quantization width at 007FH.
1 → initial prediction and quantization width specified in the PDB.
IX – Address of the source PDB with PCM data. The following fields must be completed:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (conversion value)
PDB_SAMPLE (sampling frequency)
• If C=1, also fill in:
PDB_PCM (initial predicted value)
PDB_STEP (initial quantization width)
IY – Destination PDB address with ADPCM data. The following fields must be completed:
PDB_DEV (device number)
PDB_ADDR (initial address)
Output: The following fields return valid in the source PDB:
PDB_SIZE (size after conversion)
PDB_SAMPLE (copy of sampling freq. from PCM source)
PDB_PCM (next expected value)
PDB_STEP (next quantization width)
• If CY = 1, there was an error in the conversion.
Registers: All.

RM_DAC_BIAS (00AEH/MBIOS)
Function: Sets the volume for PCM playback (sets the 17H register of the Y8950).
Input: A – 4.
IY – MIDB pointer indicating master/slave and PCM channel (device) 0 or 1.
C – Volume (1 to 7). Volume 7 is the maximum.
Output: None.
Registers: All.

RMA_DAC_DI (00AEH/MBIOS)
Function: Play PCM data.

Input: A – 5.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with reproduction data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 ⟶ playback error.
Registers: All.

RMA_ADC_DI (00AEH/MBIOS)
Function: Write PCM data.
Input: A – 6.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with recording data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 ⟶ write error.
Registers: All.

RMA_ADPCM_BIAS (00AEH/MBIOS)
Function: Sets the volume for ADPCM playback.
Input: A – 7.
IY – MIDB pointer indicating master/slave and PCM channel (device) 0 or 1.
C – Volume (0 to 63). Volume 63 is the maximum.
Output: None.
Registers: All.

RMA_ADPLAY_DI (00AEH/MBIOS)
Function: Play ADPCM data in non-local mode.
Input: A – 8.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).

IX – PDB address with reproduction data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)

Output: CY = 1 ⟶ playback error.
Registers: All.

RMA_ADPLAY_DI (00AEH/MBIOS)
Function: Record ADPCM audio in non-local mode.
Input: A – 9.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with recording data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 ⟶ write error.
Registers: All.

RMA_ADPAY_SAMPLE (00AEH/MBIOS)
Function: Changes the sampling frequency during playback in local mode.
Input: A – 43.
IY – Pointer to MIDB indicating master/slave.
DE – Frequency of sampling.
Output: None.
Registers: All.

RMA_BREAK (00AEH/MBIOS)
Function: Stop recording or playback in local mode.
Input: A – 10.
IY – Pointer to MIDB indicating master/slave.
Output: None.
Registers: All.

RMA_ADPLAY (00AEH/MBIOS)
Function: Play ADPCM data in local mode.

Input: A – 11.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with reproduction data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 → playback error.
Registers: All.

RMA_ADREC (00AEH/MBIOS)
Function: Play ADPCM data in local mode.
Input: A – 12.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with recording data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 → write error.
Registers: All.

RMA_ADPLAYLP (00AEH/MBIOS)
Function: Play ADPCM data in local loop mode. At the end, playback resumes indefinitely. To break, execute RMA_BREAK (Function 10).
Input: A – 42.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
IX – PDB address with reproduction data. The following fields must be defined:
PDB_DEV (device number)
PDB_ADDR (initial address)
PDB_SIZE (size)
PDB_SAMPLE (sampling frequency)
Output: CY = 1 → playback error.
Registers: All.

RMA_PHASE_SET_DI (00AEH/MBIOS)

Function: Take 256 bytes of PCM data in main RAM as waveform data, convert to ADPCM data and store in external RAM.

Ext RAM adr	Pitch	No. waveforms
0000H~07FFH	24H~36H	16
0800H~0FFFH	37H~42H	32
1000H~17FFH	43H~4EH	64
1800H~1FFFH	4FH~5AH	128

Input: A – 13.
IY – Pointer to MIDB indicating master/slave.
C – Filter specification (see ZMA_PH_FILTER).
DE – PCM data address.

Output: None.

Registers: All.

Note: Before conversion, RMA_BREAK (Func. 10) is executed.

RMA_PHASE_EG (00AEH/MBIOS)

Function: Define the wrap data.

Input: A – 14.
IY – Pointer to MIDB indicating master/slave.
DE – Envelope data address (7 bytes):
+0 Timer#1 value
+1 Total level
+2 Attack rate
+3 Decay rate#1
+4 Sustain Level
+5 Decay rate#2
+6 Release rate

Output: None.

Registers: All.

RMA_PHASE_EVENT (00AEH/MBIOS)

Function: Turn on sampling of the specified tone or turn off keyboard sampling simulation.

Input: A – 115.
IY – Pointer to MIDB indicating master/slave.
D – On: interval + 80H (central value: 60).
Off: interval (central value: 60).
Valid range is 24H~5AH.

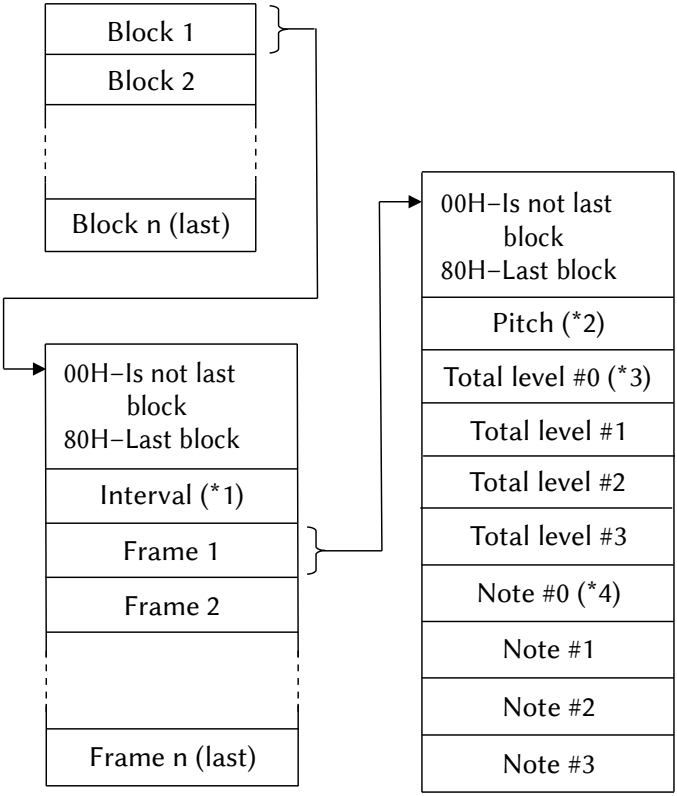
Output: None.

Registers: All.

RMA_CSM_DI (00AEH/MBIOS)

Function: CSM data playback.

Input: A – 25.
IY – Pointer to MIDB indicating master/slave.
B – Volume (0 to 127, where 0 is the maximum volume).
C – Filter specification (see ZMA_PH_FILTER).
DE – Address of the CSM data, which have the following structure:



(*1) Interval: Timer 2 ⟶ 0 = 81.9 mS / 255 = 0.32 mS
(*2) Pitch: Timer 1 ⟶ 0 = 20.9 mS / 255 = 0.08 mS
(*3) Total level (volume data for each channel):
0 = maximum / 127 = minimum
(*4) Note (pitch data for each channel).

The upper 4 bits specify the octave from 0 to 7 and the lower 4 bits specify the scale. Values are:

00 – C#	08 – G
01 – D	09 – G#
02 – D#	10 – A
03 – None	11 – None
04 – E	12 – A#
05 – F	13 – B
06 – F#	14 – C
07 – None	15 – None

Output: None.
Registers: All.

SV_IRQ (00B4H/MBIOS)
Function: MSX-Audio interrupt handling (must configure HKEYI hook (FD9AH)).
Input: None.
Output: None.
Registers: None.

9.5.6 – MSX-JE

EXTBIO (FFCAH/Work Area)
Function: Access extended BIOS functions
Input: A – 00H.
D – 16H – MSX-JE manipulation device.
E – 00H – Returns the pointer to the table of input addresses of the MSX-JE routines.
B – Address table slot ID.
HL – Address of a 64-byte buffer for the table (should be on page 3).
Output: CY = 1 ⟶ there is no MSX-JE.
CY = 0 ⟶ HL is incremented by 4 for each MSX-JE found and will point to the end of a table that reserves 4 bytes for each MSX-JE. The original value of HL points to the beginning of the table, which has the following structure:

+00H – Capacity vector
+01H – Slot ID
+02H – Lowest address
+03H – Highest address
The capacity vector byte has the following structure:
Bit 0 – 0 ⟶ MSX-JE compatible
1 ⟶ incompatible
Bit 1 – 0 ⟶ there is virtual terminal interface
1 ⟶ there is no interface
Bit 2 – 0 ⟶ dictionary interface exists
1 ⟶ there is no dictionary
Bit 3 – 0 ⟶ there is register and deletion function.
1 ⟶ there is no register and deletion function.
bit 4~bit 7 ⟶ always 0.
Slot ID (+01H) and address (+02H,+03H) specify the entry point for MSX-JE functions. The call must be made through the CALSLT (0030H) routine of the Main-ROM, placing the function number in register A.

Registers: All.

9.5.6.1 – Calling MSX-JE functions

INQUIRY (Function 01H / MSX-JE)
Function: Returns the size of the desktop.
Input: A – 01H.
Output: HL – Maximum desktop size limit used by MSX-JE.
DE – Lower limit on the size of the desktop used by MSX-JE.
BC – Minimum size required for MSX-JE to use learning function

INVOKE (Function 02H / MSX-JE)
Function: Initializes the desktop.
Input: A – 02H.
HL – Address of the desktop protected by the AP.
DE – Size of the working area protected by the AP.
Output: None.

RELEASE (Function 03H / MSX-JE)
Function: Frees the memory protected by the AP.
Input: A – 03H.
HL – Address of the desktop protected by the AP.
Output: None.

CLEAR (Function 04H / MSX-JE)
Function: Clears the buffer for Kana – Kanji conversion.
Input: A – 04H.
HL – Address of the desktop protected by the AP.
Output: None.

SET_TTB (Function 05H / MSX-JE – Optional)
Function: Pass the text and read the data to be converted from the AP to the MSX–JE, configuring them in the MSX-JE's internal buffer. This function causes MSX–JE to reconvert the given text.
Input: A – 05H.
HL – Desktop address.
DE – Address of the text to be converted again.
BC – TTB (Transferable Text Block) address.
Output: A = 255 ⟶ function not supported.

DISPATCH (Function 06H / MSX-JE – Optional)
Function: Pass CPU control from AP to MSX–JE.
Input: A – 06H.
HL – Desktop address.
Output: HL – STB (Screen image Text Block) address.
A – Return status:
bit 0 = 1 ⟶ the AP displays the STB.
bit 1 = 1 ⟶ AP can get conversion result.
bit 2 = 1 ⟶ MSX-JE conversion finished.
Possible states with the combined bits:
000 – MSX-JE ignores the key (no key entry).
001 – Entry or conversion in progress.
01x – Partial conversion made.
10x – Conversion stopped and finished.
11x – Fully converted.

GET_RESULT (Function 07H / MSX-JE)
Function: Returns the result of the conversion.
Input: A – 07.
HL – Desktop address.
Output: HL – Start address of the conversion result, ending with a 00H byte.

GET_TTB (Function 08H / MSX-JE – Optional)
Function: Acquires text data obtained by GET_RESULT.
Check-in: A – 08.
HL – Desktop address.
Output: HL – TTB (Transferable Text Block) address. If this function is not supported, (HL) will point to a 00H byte.

INQUIRY_WINDOW_SIZE (Function 09H / MSX-JE – Optional)
Function: Defines the window format.
Input: A – 09:00.
HL – desktop address.
E – maximum length of the "tail".
B – maximum height of the window.
C – maximum width of the window.
Output: HL – address of window specification data.
+00H – Type of window:
1 – Independent.
2 – "tail".
+01H – Window width (1~255).
+02H – Window height (1~255).

CONFLICT_DETECT (Function 0AH / MSX-JE – Optional)
Function: Avoid key collision conflicts.
Input: A – 0AH.
HL – Desktop address.
Output: A – 00H ⟶ conflict not detected.
FFH ⟶ conflict detected.

9.5.6.2 – MSX-JE dictionary interface

HAN_ZEN (Function 40H)
Function: Converts a one-byte character string to a two-byte character string.

Input: A – 40H.
HL – Desktop address.
DE – Source string address (one byte).
BC – Address of the two-byte character string.
Output: A = 0 ⟶ successful conversion.
A ≠ 0 ⟶ conversion error.

ZEN_HAN (Function 41H / MSX-JE)
Function: Converts a two-byte character string to a one-byte character string.
Input: A – 41H.
HL – Desktop address.
DE – Source string address (two byte).
BC – Address of one-byte character string.
Output: A = 0 ⟶ successful conversion.
A ≠ 0 ⟶ conversion error.

HAN_KATA (Function 42H / MSX-JE)
Function: Converts a one-byte character string from the roman alphabet, katakana, or a combination thereof to a two-byte katakana character string.
Input: A – 42H.
HL – Desktop address.
DE – Source string address (one byte).
BC – Address of the katakana character string.
Output: A = 0 ⟶ successful conversion.
A ≠ 0 ⟶ conversion error.

HAN_HIRA (Function 43H / MSX-JE)
Function: Converts a one-byte character string from the roman alphabet, katakana, or a combination thereof to a two-byte hiragana character string.
Input: A – 43H.
HL – Desktop address.
DE – Source string address (one byte).
BC – Address of the hiragana character string.
Output: A = 0 ⟶ successful conversion.
A ≠ 0 ⟶ conversion error.

KATA_HIRA (Function 44H / MSX-JE)

Function: Converts a two-byte katakana character string to a two-byte hiragana character string.
Input: A – 44H.
HL – Desktop address.
DE – Address of the two-byte katakana string.
BC – Address of the hiragana character string.
Output: A = 0 ⟶ successful conversion.
A ≠ 0 ⟶ conversion error.

HIRA_KATA (Function 45H / MSX-JE)

Function: Converts a two-byte hiragana character string to a two-byte katakana character string.
Input: A – 45H.
HL – Desktop address.
DE – Address of the two-byte katakana string.
BC – Address of the hiragana character string.
Output: None.

OPEN_DIC (Function 46H / MSX-JE)

Function: Reserved for future expansions.
Input: A – 46H.
HL – Desktop address.
DE – 0000H
Output: A – Always returns 5.

HENKAN (Function 47H / MSX-JE)

Function: Converts a 2-byte katakana and hiragana character string to a mixed Kanji-Kana string.
Input: A – 47H.
HL – Desktop address.
DE – Address of the string katakana/hiragana.
Output: A – Number of possible conversions. If there is none, it returns 0. The converted strings must be obtained by the JI_KOHO (48H) function.

JI_KOHO (Function 48H / MSX-JE)

Function: Acquires the next conversion obtained by HENKAN (47H).

Input: A – 48H.
HL – Desktop address.
DE – Address of the next converted Kanji-Kana string.
BC – Secondary Kanji-Kana string address.
Output: A = 0 ⟶ No Kanji-Kana conversion acquired.
A > 0 ⟶ Acquired Kanji-Kana conversion number.

ZEN_KOHO (Function 49H / MSX-JE)
Function: Acquires the previous conversion obtained by HENKAN (47H).
Input: A – 49H.
HL – Desktop address.
DE – Address of the former Kanji-Kana string converted.
BC – Secondary Kanji-Kana string address.
Output: A = 0 ⟶ No Kanji-Kana conversion acquired.
A > 0 ⟶ Acquired Kanji-Kana conversion number.

JI_BLOCK (4AH Function / MSX-JE)
Function: Creates a lower priority Kanji-Kana conversion group next to the main group.
Input: A – 4AH.
HL – desktop address.
Output: A = 0 ⟶ group not created.
A > 0 ⟶ number of lowest priority Kanji-Kana conversions.

ZEN_BLOCK (4BH Function / MSX-JE)
Function: Creates a higher priority Kanji-Kana conversion group next to the main group.
Input: A – 4BH.
HL – Desktop address.
Output: A = 0 ⟶ Group not created.
A > 0 ⟶ Highest priority number of Kanji-Kana conversions.

KAKUTEI1 (4CH Function)
Function: Confirms the result of Kanji-Kana conversion.
Input: A – 4CH.
HL – Desktop address.
E – Kanji-Kana conversion number within the group.
BC – Kanji-Kana translation buffer address.
Output: BC – 0AH + "natto curry".

KAKUTEI2 (4DH Function)
Function: Confirms the result of Kanji-Kana conversion.
Input: A – 4DH.
HL – Desktop address.
E – Kanji-Kana conversion number within the group
BC – Kanji-Kana translation buffer address.
Output: A – Size in bytes of the Kanki-Kana string.
BC – 04H + "natto".

CLOSE_DIC (4EH Function)
Function: Function not implemented.
Input: A – 4EH.
Output: A – Always 0.

TOUROKU (4FH Function)
Function: Provides reading data, word data and part of text, and includes the specified word in the dictionary.
Input: A – 4FH.
HL – Desktop address.
DE – Read buffer address.
BC – Address of word inclusion buffer.
Output: A – 00H ⟶ word successfully added.
01H ⟶ insufficient free space.
02H ⟶ word parity overflow.
04H ⟶ incorrect reading data.
08H ⟶ incorrect word data.
10H ⟶ part of text is incorrect.
FFH ⟶ not supported.

SAKUJO (50H Function)
Function: Provides reading data, word data and part of text excluding the specified word from the dictionary.
Input: A – 50H.
HL – Desktop address.
DE – Read buffer address.
BC – Address of the word exclusion buffer.

Output: A – 00H ⟶ word successfully deleted.
01H ⟶ word to be deleted was not found.
04H ⟶ incorrect reading data.
08H ⟶ incorrect word data.
10H ⟶ part of text is incorrect.
FFH ⟶ not supported.

9.5.7 – MSX UNAPI

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A = 00H – Gets the number of implementations of the specified API.
A > 00H ⟶ Returns the parameters of the specified API.
D = 22H ⟶ MSX UNAPI manipulation device.
E = 22H ⟶ Returns data from the specified API.
(F487H) – API specification identifier, which must be an alphanumeric string of up to 15 characters ending in 00H, not case sensitive.
Output: A = 00H ⟶ B – Number of implementations of the specified API.
A > 00H ⟶ A – Implementation routine slot ID.
B – Implementation mapper segment (FFH = not in the mapper).
HL – Entry point address of the implementation routines (if it is on physical page 3, the values of A and B are disregarded).
Registers: AF, BC, HL.

9.5.7.1 – RAM Helper

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A = FFH ⟶ API: RAM helper
D = 22H ⟶ MSX UNAPI manipulation device.
E = 22H ⟶ Returns API parameters.
HL = 0000H

Output: HL = 0000H ⟶ RAM helper not installed.
HL > 0000H ⟶ HL = Jump table address on page 3.
BC – Address of the mapping table.
A – Number of entries in the jump table, which has the following structure:
+00H CALMAP calls routine mapper
+03H RDBYTE reads byte from RAM
+06H CALSEG calls routine in RAM
Registers: AF, BC, HL.

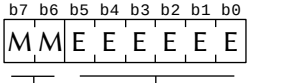
CALMAP (HL+00H) – HL value obtained via EXTBIO
Function: Calls a routine on a mapped RAM segment.
Input: IYh – Slot ID.
IYl – Mapper segment number.
IX – Routine address (must be on physical page 1).
AF, BC, DE, HL – Parameters for the called routine.
Output: AF, BC, DE, HL, IX, IY – Routine return parameters.
Registers: Depends on the called routine.

RDBYTE (HL+03H) – HL value obtained via EXTBIO
Function: Reads a byte from a segment of the mapped RAM.
Input: A – Slot ID.
B – Segment number.
HL – Address to be read (highest two bits are ignored).
Output: A – byte read at specified address.
Registers: A.

CALSEG (HL+06H) – HL value obtained via EXTBIO
Function: Calls a routine a segment of the mapped RAM using inline parameters.
Input: AF, BC, DE and HL can contain parameters for the called routine (do not use IX and IY).
Inline call parameters, in the following format:

```
CALL <routine address>
DB   <routine ID>
DB   <segment number>
```

Routine ID:



Routine to call in jump table (0 to 63)

Mapper slot specified in mapper table (0 to 3)

- The jump table starts at address 4000H, where index 0 means 4000H, index 1 means 4003H, and só on up to the value 63, every three bytes.
- The mapper table occupies 8 bytes, reserving two bytes for each mapper, being able to manage up to 4 mappers (0 to 3), and has the following structure:
  +0 – Slot ID of first mapper
  +1 – Number segments available in the 1st mapper
  +2 – Second mapper slot ID
  +3 – Number segments available in the 2nd mapper
  +4 – Third mapper slot ID
  +5 – Number segments available on the 3rd mapper
  +6 – Fourth mapper slot ID
  +7 – Number segments available on the 4th mapper

  Note: if the mapper has 4 Mbytes, the number of segments will be FEH, since the FFH value has been reserved for the system.

Output: AF, BC, DE, HL, IX and IY can contain valid values.
Registers: Depends on the called routine.

9.5.7.2 – API for Ethernet cartridges

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A = 00H ⟶ Gets the number of implementations of the specified API.
A > 00H ⟶ Returns the parameters of the specified API.
D = 22H ⟶ MSX UNAPI manipulation device.
E = 22H ⟶ Returns data from the specified API.
(F487H) – "ETHERNET"

Output: A = 00H ⟶ B – Number of API implementations.
A > 00H ⟶ A – Implementation routine slot ID.
B – Implementation mapper segment
(FFH = not in the mapper).
HL – Entry point address of the implementation routines
(if it is on physical page 3, the values of A and B are
disregarded).
Registers: AF, BC, HL.

ETH_GETINFO (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the version and name of the implementation.
Input: A = 0.
Output: HL – Implementation name string address.
B – Version of the API implementation (primary).
C – Version of the API implementation (secondary).
D – API version specification (primary).
E – API version specification (secondary).
Registers: All.

ETH_RESET (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the hardware and state variables to their initial
condition (condition right after computer reset).
Input: A = 1.
Output: None.
Registers: All.

ETH_GET_HWADD (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns Ethernet address.
Input: A = 2.
Output: L-H-E-D-C-B – Address.
Registers: All.

ETH_GET_NETSTAT (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Checks the network connection status.
Input: A = 3.
Output: A = 0 ⟶ no connection to an active network.
1 ⟶ there is an active network connection.
Registers: All.

ETH_NET_ONOFF (HL/ExtBIOS) – HL value obtained via EXTBIO

Function: Enables or disables the network.
Input: A = 4.
B = 0 ⟶ returns the current networking state.
1 ⟶ enable networking.
2 ⟶ disable networking.
Output: A = 1 ⟶ network enabled.
2 ⟶ network disabled.
Registers: All.

ETH_DUPLEX (HL/ExtBIOS) – HL value obtained via EXTBIO

Function: Sets duplex mode.
Input: A = 5.
B – 0 ⟶ returns to current mode.
1 ⟶ select half-duplex mode.
2 ⟶ selects full-duplex mode.
Output: A – 1 ⟶ half-duplex mode selected.
2 ⟶ half-duplex mode selected.
3 ⟶ unknown mode or duplex mode is not applicable.

ETH_FILTERS (HL/ExtBIOS) – HL value obtained via EXTBIO

Function: Configures frame reception filters.
Input: A = 6.
B – bit 7 – 0 ⟶ no action.
1 ⟶ returns current setting.
bit 6 – reserved.
bit 5 – reserved.
bit 4 – 0 ⟶ disable promiscuous mode.
1 ⟶ enable promiscuous mode.
bit 3 – reserved.
bit 2 – 0 ⟶ reject "broadcast" frames.
1 ⟶ accepts "broadcast" frames.
bit 1 – 0 ⟶ reject frames smaller than 64 bytes.
1 ⟶ accept frames smaller than 64 bytes.
bit 0 – Reserved.
Output: A – filter configuration after execution (same format as register B on input).
Registers: All.

ETH_IN_STATUS (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Checks the availability of received frames.
Input: A = 7.
Output: A – 0 ⟶ No incoming frames available.
1 ⟶ At least one received frame is available.
BC – Oldest frame size available.
HL – Bytes 12 and 13 of the oldest frame available.
Registers: All.

ETH_GET_FRAME (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Recovers the oldest frame.
Input: A = 8.
HL = 0 ⟶ Discard the frame.
Other value ⟶ frame destination address.
Output: A – 0 ⟶ Frame retrieved or discarded.
1 ⟶ There are no received frames available.
BC – Retrieved frame size.

ETH_SEND_FRAME (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Sends a frame.
Input: A = 9.
HL – Destination address of the frame in memory.
BC – Frame size.
D – Execution mode: 0 – Synchronous.
1 – Asynchronous.
Output: A – 0 ⟶ Frame sent or transmission started.
1 ⟶ Invalid frame size.
2 ⟶ Ignored.
3 ⟶ Lost carrier.
4 ⟶ Excessive number of collisions.
5 ⟶ Asynchronous mode not supported.
Registers: All.

ETH_OUT_STATUS (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Recovers the oldest frame.
Input: A = 10.
Output: A – 0 ⟶ no frame sent since last reset.
1 ⟶ transmitting at this time.

2 ⟶ transmission completed successfully.
3 ⟶ lost carrier.
4 ⟶ excessive number of collisions.

Registers: All.

ETH_SET_HWADD (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Selects Ethernet address.
Input: A = 11.
L-H-E-D-C-B – Ethernet address to be set.
Output: L-H-E-D-C-B – Ethernet address after execution.
Registers: All.

9.5.8 – MemMan

EXTBIO (FFCAH/Work Area)
Function: Accesses extended BIOS functions.
Input: A = 00H
D = 4DH – MEMMAN manipulation device.
E = 32H – Returns information about alternative inputs for MemMan functions.

B – 0 ⟶ Input address for FastUse0 (func. 0)
1 ⟶ Input address for FastUse1 (func. 1)
2 ⟶ Input address for FastUse2 (func. 2)
3 ⟶ Input address for FastTsrCall (fn. 63)
4 ⟶ Input address for BasicCall
5 ⟶ Input address for FastCurSeg (fn. 32)
6 ⟶ Input address for handler of MemMan functions
7 ⟶ Returns MemMan version (VerMM:#H.L)
8 ⟶ Input address for FastXTsrCall (f. 61)

Output: HL – Address or version.
Registers: All.

9.5.8.1 – Fast Calls (Preferred alternative entries)

FastUse0 (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a segment on physical page 0 (0000H~3FFFH). Enabling is only possible if the segment contains the entry points to the standard slot switching routines.

Input: HL – Segment number.
Output: A – 00H ⟶ segment enabled successfully.
FFH ⟶ segment enable failed.
Note: This function is identical to function 0 (Use0).

FastUse1 (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a segment on physical page 1 (4000H~7FFFH).
Input: HL – Segment number.
Output: A – 00H ⟶ segment enabled successfully.
FFH ⟶ segment enable failed.
Note: This function is identical to function 1 (Use1).

FastUse2 (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Enables a segment on physical page 2(8000H~BFFFH).
Input: HL – Segment number.
Output: A – 00H ⟶ segment enabled successfully.
FH ⟶ segment enable failed.
Note: This function is identical to function 2 (Use2).

FastTsrCall (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls a TSR driver routine.
Input: BC – TSR function ID code.
AF, DE, HL – Parameters for the TSR.
Output: AF, BC, DE, HL – TSR return parameters.
Note: This function is identical to function 63 (TsrCall), except here the DE register can be used without problems.

BasicCall (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls a routine from Main-ROM.
Input: IX – Routine address on physical page 0 or 1.
AF, BC, DE, HL – Parameters to be passed to the routine.
Output: AF, BC, DE, HL – Routine return parameters.
Note: Interrupts are disabled.

FastCurSeg (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the current segment number of a page.
Input: B – Physical page (0, 1, 2 or 3).

Output: HL – Segment number.
A – Type of segment: 00H ⟶ PSEG.
FFH ⟶ FSEG.
Note: This function is identical to function 32 (CurSeg).

MemMan (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Directly call a MemMan function.
Input: E – Function number.
AF, BC, HL – Parameters to be passed to the routine.
Output: AF, BC, DE, HL – Routine return parameters.

VerMM (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Returns the MemMan version number.
Input: None.
Output: HL – "H.L" format version.

FastXTsrCall (HL/ExtBIOS) – HL value obtained via EXTBIO
Function: Calls driver input from a TSR.
Input: IX – ID code of the called TSR input.
AF, BC, DE, HL – Parameters to be passed to the routine.
Output: AF, BC, DE, HL – Routine return parameters.
Note: This function is identical to function 61 (XtrsCall).

9.5.8.2 – MemMan Functions

Use0 (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 00H – Use0 function. Enables a segment on physical page 0 (0000H~3FFFH). Enabling is only possible if the segment contains the entry points to the standard slot switching routines.
HL – segment number.
Output: A – 00H ⟶ segment enabled successfully.
FFH ⟶ segment enable failed.
Note: Preferably use the FastUse0 input of the 32H function (Info) of MemMan.

Use1 (FFCAH/Work Area) – Execution via EXTBIO

Function: Accesses extended BIOS functions.

Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 01H – Use0 function. Enables a segment on physical page 1 (4000H~7FFFH).
HL – segment number.

Output: A – 00H ⟶ segment enabled successfully.
FFH ⟶ segment enable failed.

Note: Preferably use the FastUse1 input of the 32H function (Info) of MemMan.

Use2 (FFCAH/Work Area) – Execution via EXTBIO

Function: Accesses extended BIOS functions.

Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 02H – Use2 function. Enables a segment on physical page 2 (8000H~BFFFH).
HL – segment number.

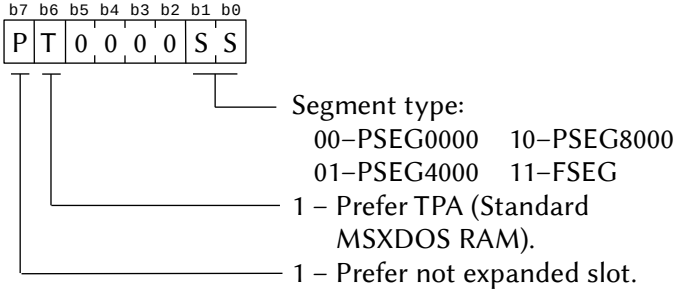
Output: A – 00H ⟶ segment enabled successfully.
FFH ⟶ segment enable failed.

Note: Preferably use the FastUse2 input of the 32H function (Info) of MemMan.

Alloc (FFCAH/Work Area) – Execution via EXTBIO

Function: Accesses extended BIOS functions.

Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 0AH – Alloc function. Allocates a segment.
B – Segment preference code:



Output: HL – Segment number (0000H ⟶ no segments free).

SetRes (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 0BH – SetRes function. Assigns to a segment the "reserved" status.
HL – Segment number.
Output: None.

DeAlloc (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H
D = 4DH – MEMMAN manipulation device.
E = 14H – DeAlloc function. Frees a segment.
HL – Segment number.
Output: None.

IniChk (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = Control code.
D = 4DH – MEMMAN manipulation device.
E = 1EH – IniChk function. Start MemMan before a program.
Output: A – Control code + "M".
DE – Version number in "D:E" format.

Status (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 1FH – Status Function. Return status information of MemMan.
Output: HL – Number if segments available.
BC – Number of free segments.
DE – Number of segments controlled simultaneously through MemMan and DOS2.
A – Hardware connection status:
bit0 – 0 ⟶ DOS2 mapper support not available.
1 ⟶ DOS2 mapper support installed.
bit1~bit 7 ⟶ always 0.

CurSeg (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 20H – CurSeg function. Returns the segment number on a page.
B – Physical page number (0, 1, 2 or 3).
Output: HL – Segment number.
A – Type of segment: 00H ⟶ PSEG.
FFH ⟶ FSEG.
Note: Preferably use the FastCurSeg entry of the 32H (Info) function of MemMan.

StoSeg (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 28H – StoSeg function. Stores the state of the current segment.
HL – Address of a 9-byte buffer.
Output: None.

RstSeg (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 29H – RstSeg function. Reactivates the state of a segment that has been stored.
HL – Address of a 9-byte state buffer.
Output: None.

XtsrCall (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 3DH – XTsrCall function. Call an entry from the TSR driver.
IX = ID code of the called TSR input.
AF, BC, HL – Parameters to be passed to the routine.

Output: AF, BC, DE, HL – Routine return parameters.
Note: Preferably use the FastXtrsCall entry of the 32H (Info) function of MemMan.

GetTsrID (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 3EH - GetTsrID function. Determines the TSR ID code.
HL – Pointer to TsrName. Unused positions must be padded with spaces.
Output: CY = 0 ⟶ Not found; 1 ⟶ ID found.
BC ⟶ TSR ID code.

TsrCall (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 3FH – TsrCall function. Call an entry from the TSR driver.
BC – ID code of the called TSR input.
AF, HL – Parameters to be passed to the routine.
Output: AF, BC, DE, HL – Routine return parameters.
Note: Preferably use the FastTrsCall entry of the 32H (Info) function of MemMan.

HeapAlloc (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 46H – HeapAlloc function. Allocates space in the "heap".
HL – Size of the space to be allocated.
Output: HL – 0000H ⟶ insufficient memory for allocation.
Other value ⟶ start address of allocated space.

HeapDeAlloc (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.

Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 47H – HeapDeAlloc Function. Frees up space allocated in the "heap".
HL – Size of the space to be allocated.
Output: None.

HeapMax (FFCAH/Work Area) – Execution via EXTBIO
Function: Accesses extended BIOS functions.
Input: A = 00H.
D = 4DH – MEMMAN manipulation device.
E = 48H – HeapMax function. Returns the maximum size of space available in the "heap".
Output: HL – Available space in the "heap".

9.5.9 – System commands

EXTBIO (FFCAH/Work Area)
Function: Access extended BIOS functions
Input: A = 00H.
D = FFH – System Device.
E = 00H – Returns the starting address, slot ID and manufacturer code of the device.
B – ID of the slot from the parameter table.
HL – Address of the parameter table.
Output: CY = 1 ⟶ there are no devices.
0 ⟶ there are devices.
B – ID of the slot from the parameter table.
HL – Starting address of the parameter table.
Note: Each device occupies 5 bytes in the table pointed to by HL, with the following structure:
+00H – Reserved. Always 0.
+01H – Manufacturer code.
+02H – MSB address of the jump table.
+03H – LSB address of the jump table.
+04H – Device slot ID.

Manufacturers are as follows:

00 – ASCII
01 – Microsoft
02 – Canon
03 – Casio Computer
04 – Fujitsu
05 – General Fujitsu
06 – Hitachi, Ltd.
07 – Kyocera
08 – Matsushita (Panasonic)
09 – Mitsubishi Electric Corporation
10 – NEC
11 – Yamaha (Nippon Gakki)
12 – Japan Victor Company (JVC)
13 – Philips
14 – Pioneer
15 – Sanyo Electric
16 – Sharp Japan
17 – Sony
18 – Spectravideo
19 – Toshiba
20 – Mitsumi Electric
21 – Telematica
22 – Gradient Brazil
23 – Sharp do Brasil
24 – GoldStar (LG)
25 – Daewoo
26 – Samsung
128 – Image Scanner (Matsushita)
170 – Darky (SuperSoniqs)
171 – Darky (SuperSoniqs) second setting
212 – 1chipMSX / Zemmix Neo (KdL firmware)
254 – MPS2 (ASCII)

9.6 – DISC INTERFACE ROUTINES

9.6.1 – Interface Initialization

The routines below are executed only once during the system initialization at power up or after a reset, in the sequence INIHRD, DRIVES, INIENV. Their calling address is different for each interface.

INIHRD (????H/Disk interface).
Function: Initializes the hardware as soon as control is passed to the disk interface cartridge.
Input: None.
Output: None.
Registers: All.

DRIVES (????H/Disk Interface).
Function: Checks the physical drives connected to the system.
Input: Flag Z = 0 ⟶ Two logical units are assigned to one physical unit.
1 ⟶ Only one logical drive is assigned to a physical drive.
Output: L – Number of connected drives.
Registers: F, HL, IX, IY.

INIENV (????H/Disk Interface).
Function: Initializes the disk interface desktop.
Input: None.
Output: None.
Registers: All.

9.6.2 – Standard interface routines

MALLOC (01CBH/Disk Interface)
Function: Allocates a buffer for a segment for MSXDOS2.
Input: Number of bytes to reserve.
Output: A > 0 ⟶ Allocation error.
A = 0 ⟶ Allocation made.
HL – Buffer start address.
(HL–2, HL–1) – Buffer size + 2.
Registers: All.

DEALOC (2D0FH/Disk Interface)
Function: Reallocates a buffer to a segment for MSXDOS2.
Input: HL – Initial buffer address.
(HL-2, HL-1) ⟶ Buffer size + 2.
Output: Unknown.
Registers: All.

DSKIO (4010H/Disk Interface)

Function: Direct reading/writing of sectors.

Input: HL – Pointer to TPA (clipboard).
DE – Number of the first sector to read or write.
B – Number of sectors to read or write.
A – Drive number (0=A:, 1=B:, 2=C:, etc).
C – Disk formatting ID:
F0H – 63 sectors per track (for HD's).
F8H – 80 tracks, 9 sectors per track, single face.
F9H – 80 tracks, 9 sectors per track, double sided.
FAH – 80 tracks, 8 sectors per track, single face.
FBH – 80 tracks, 8 sectors per track, double sided.
FCH – 40 tracks, 9 sectors per track, single face.
FDH – 40 tracks, 9 sectors per track, double sided.
CY = 0 ⟶ reading.
1 ⟶ writing

Output: B – Number of sectors effectively transferred.
CY = 1 ⟶ Transfer successfully executed.
0 ⟶ Transfer error. The error code is returned in register A.
A – Error code:
00 – Write protected.
02 – Not ready.
04 – CRC error (sector not accessible).
06 – Seek error.
08 – Record not found.
10 – Write fault.
12 – Other errors.
MSXDOS2 or higher only:
18 – Not a DOS disk.
20 – Incorrect MSXDOS version.
22 – Unformatted disk.
24 – Disk swapped.
Remaining: disk error.

Registers: All.

DSKCHG (4013H/Disk Interface)

Function: Check the swap status of the disk.

Input: A – Drive number (0=A:, 1=B:, 2=C:, etc).
B – Always 00H.
C – Disk formatting ID (same as DISKIO/4010H).
HL – Pointer for the respective DPB.

Output: CY = 0 ⟶ successfully verified.
CY = 1 ⟶ execution error.
A – Error code (same as DISKIO/4010H).
B – 00H ⟶ unknown state.
01H ⟶ disk not exchanged.
FFH ⟶ disk swapped.

Registers: All.

GETDPB (4016H/Disk Interface)

Function: Return the disk drive DPB.

Input: A – Drive number
B – First byte of FAT (disk ID).
C – Disk formatting ID (same as DISKIO/4010H).
HL – Pointer to the DPB to be filled (18 bytes).

Output: HL – DPB initial address filled in:

+00H	DRIVE	Drive number (0=A:, etc)
+01H	MEDIA	Media Type (F8H~FFH)
+02H	SECSIZ	Sector Size (must be 2^n)
+04H	DIRMSK	(SECSIZ/32) – 1
+05H	DIRSHFT	Number of bits 1 in DIRMSK
+06H	CLUSMSK	(Sectors per cluster) – 1
+07H	CLUSHFT	(Num of 1 bits in CLUSMSK) – 1
+08H	FIRFAT	Logical sector number of 1st FAT
+0AH	FATCNT	Number of FATs
+0BH	MAXENT	Number of root directory entries
+0CH	FIRREC	First sector data area
+0EH	MAXCLUS	(Total of clusters) + 1
+10H	FATSIZ	Number of sectors used
+11H	FIRDIR	First directory sector
+13H	FATDIR	FAT address in RAM

Registers: All.

CHOICE (4019H/Disk Interface)
Function: Returns the address of the disk format message.
Input: None.
Output: HL – Message address, which ends with a 00H byte. If there is no choice (only one formatting type is supported), HL returns 0000H.
Registers: All.

DSKFMT (401CH/Disk Interface)
Function: Format a disk.
Input: A – Choice of formatting by the user (CHOICE /4019H routine). It can range from 1 to 9.
D – Drive number (00H=A:, 01H=B:, etc).
HL – Starting address of the workspace used by the formatting routine.
BC – Size of the workspace used by the formatting routine.
Output: CY – 0 ⟶ Formatting completed successfully.
1 ⟶ Error during formatting.
A – Error code:
00 – Write protected.
02 – Not ready.
04 – Data error (CRC).
06 – Seek error.
08 – Record not found.
10 – Write fault.
12 – Bad parameter.
14 – Insufficient memory.
16 – Other errors.
Registers: All.

MTROFF (401FH/Disk Interface)
Function: Stop the motor of the drives.
Input: None.
Output: None.
Registers: All.
Note: This function is implemented in only some interfaces. If the interface does not have this function implemented, the value of address 401FH will be 00H. Therefore, it is necessary to verify that the function exists by reading address 401FH before calling it.

CALBAS (4022H/Disk Interface)
Function: Call the BASIC interpreter.
Input: None.
Output: None.
Registers: All.

FORMAT (4025H/Disk Interface)
Function: Format a disk displaying message.
Input: None.
Output: None.
Registers: All.

STPDRV (4029H/Disk Interface)
Function: Stop the motor of the drives.
Input: None.
Output: None.
Registers: All.

SLTDOS (402DH/Disk Interface)
Function: Returns the DOS Kernel slot ID.
Input: None.
Output: A – Slot ID (same as RDSLT (000CH/Main)).
Registers: All.

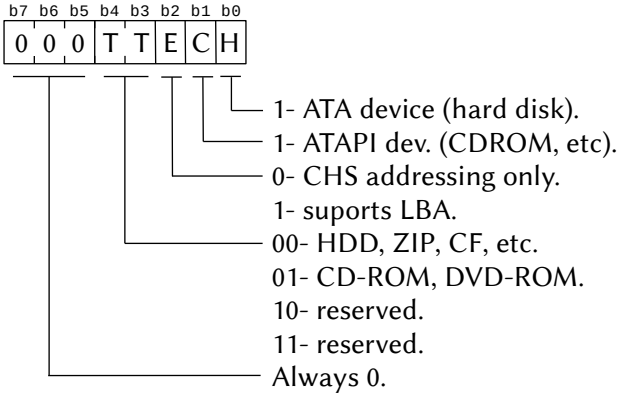
HIGMEM (4030H/Disk Interface)
Function: Returns the highest address available in RAM.
Input: None.
Output: HL – Address.
Registers: All.

BLKDOS (402DH/Disk Interface)
Function: Returns the current MSXDOS2 block.
Input: None.
Output: A – Current block number (0 to 3).
Registers: All.
Note: The 64 Kbytes of MSXDOS2 Kernel ROM are split into 4 segments that can be active only on physical page 1. Therefore, they are constantly swapped during processing.

9.6.3 – Routines for accessing standard IDE Hard-Disks

IDBYT (7F80H/IDE Interface)
Function: Interface ID in 3 bytes. ("ID#" for IDE interfaces).

RDLBLK (7F89H/IDE Interface)
Function: Read logical sectors from disk or device.
Input: CDE – Sector number.
HL – RAM address for read data.
B – Number of sectors to read.
A – Device ID:



Output: HL – Pointer to the read data.
CY = 1 ⟶ Read error.
A – Error code for IDE devices:
00 – Write protected.
02 – Not ready.
04 – CRC error (sector not accessible).
06 – Seek error.
08 – Record not found.
10 – Write fault.
12 – Other errors.
MSXDOS2 or higher only:
18 – Not a DOS disk.
20 – Incorrect MSXDOS version.
22 – Unformatted disk.
24 – Disk swapped.
Remaining: other errors.

Registers: All.
Note: This routine can also read sectors from the CD-ROM, which have 2048 bytes instead of 512 bytes of the HD's.

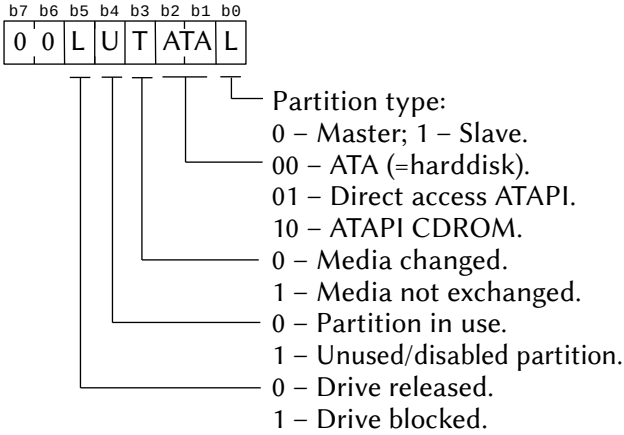
WRLBLK (7F8CH/IDE Interface)
Function: Write logical sectors of the disk.
Input: CDE – Sector number.
HL – Starting address of the data to be written.
B – Number of sectors to write.
A – Device ID. Same as RDLBLK (7F89H).
Output: CY = 1 ⟶ Writing error.
A – Error code. Same as RDLBLK (7F89H).
Registers: All.

SELDEV (7FB9H/IDE Interface)
Function: Select master/slave for ATAPI devices.
Input: A – bit0 = 0 ⟶ Master.
1 ⟶ Slave.
bit1~bit7: reserved. Always 0.
Output: CY = 1 ⟶ time-out error occurs.
Registers: A, BC, IX.

PACKET (7FBCH/IDE Interface)
Function: Send a sequence of ATAPI commands to the selected device.
Input: HL – Pointer to 12-byte ATAPI command packet (cannot be on page 1 – 4000H~7FFFH).
DE – Address for data transfer (if any).
Output: CY = 1 ⟶ execution error.
Z = 1 ⟶ time-out error.
A = Error code. Same as RDLBLK (7F89H).
Registers: All.
Attention: This entry has different function on SCSI interfaces.

DRVADR (7FBFH/IDE Interface)
Function: Returns the desktop address.
Input: A – Unit number (0 to 7).
0~5 – Drive number (0=A:~5 = F:)
6 – Device Y Infobytes.
7 – 18 bytes of free space (used internally for sending ATAPI command strings).

Output: HL – Pointer to start of data:
+00H – Device Codebyte:



+01H~+03H – Partition start sector (bits 0~23).
+04H~+06H – (Size of partition in sectors) – 1 (bits 0~23).
+07H – Additional information about the partition.
For BIOS 3.0 or higher:
+08H – Partition start sector (bits 24~31).
+08H – (Size of partition in sectors) – 1 (bits 24~31).
Registers: AF, BC, DE, HL, IX.

9.6.4 - Routines added by NEXTOR

GSLOT1 (402DH / NEXTOR Kernel)
Function: Returns the current driver slot.
Input: None.
Output: A – Slot identifier.
Registers: AF.
Note: This routine cannot be called directly. It must be called through CALBNK (4042H), as follows:

```
XOR  A
LD   IX,GSLOT1
CALL CALBNK
```

RDBANK (403CH / NEXTOR Kernel)
Function: Reads a byte from any Kernel bank.

Input: A – Bank number.
HL – Address (must be on physical page 1).
Output: A – Byte read.
Registers: AF.
Note: This routine cannot be called directly. It must be called through CALBNK (4042H), as follows:

```
LD   A,<bank number>
LD   HL,<byte address>
LD   IX, RDBANK
CALL CALBNK
```

CALLB0 (403FH / NEXTOR Kernel)
Function: Temporarily switch the Kernel's main bank (usually bank 0, but will be 3 when running in MSX-DOS 1 mode), and then call the routine whose address is in CODE_ADD (F1D0H).
Input: CODE_ADD – Address of the routine to be called.
AF, BC, DE, HL, IX, IY – Parameters for the routine.
Output: AF, BC, DE, HL, IX, IY – Return data from the routine.
Registers: All.

CALBNK (4042H / NEXTOR Kernel)
Function: Call routine in another Kernel bank.
Input: A – Bank number.
IX – Routine address (must be on physical page 1).
AF' – Input parameter for the called routine. (will be passed as AF to the called routine).
BC, DE, HL, IY – Parameters for the called routine.
Output: AF, BC, DE, HL, IX, IY – Routine output values.
Registers: All.

GWORK (4045H / NEXTOR Kernel)
Function: Get the address of the 8-byte SLTWRK input for the past slot or for the current slot on page 1. The first two bytes of this area will contain a pointer to the page 3 work area allocated for this driver (as requested in the routine DRV_INIT), or zero if no workspace has been allocated.
Input: A – Slot number (0 for current slot on page 1).
Output: A – Current page 1 slot (if 0 on input). Unchanged if not 0 in input).
IX – Address of the 8-byte SLTWRK entry.

Registers: F
Note: This routine cannot be called directly. It must be called through CALBNK (4042H), as follows:

```
LD   A,<slot number or 0>
EX   AF, AF'
XOR  A
LD   IX, GWORK
CALL CALBNK
```

K_SIZE (40FEH / NEXTOR Kernel)
Function: This address contains a byte that tells how many banks make up the Kernel Nextor (or alternatively, the driver's first bank number).

CUR_BANK (40FFH / NEXTOR Kernel)
Function: This address contains a byte with the current bank number. For the first driver bank, this value is the same as K_SIZE and increases by one for each additional driver bank (if any).

CHGBNK (7FD0H / NEXTOR Kernel)
Function: Make the specified bank visible on page 1 of the Z80. This routine is available in all banks, not just bank 0. Normally, the driver code will not need to use this routine, but will use CALBNK instead.
Input: A – Bank number.
Output: None.
Registers: AF.

PROMPT (41E8H / Kernel NEXTOR v2.1.0)
Function: Display the message "Insert disk to drive X: and press a key when ready" and wait for the requested action.
Input: None.
Output: None.
Registers: All.
Note: This routine cannot be called directly. It must be called through CALLB0 (403FH), as follows:

```
PROMPT:   EQU 041E8H
CODE_ADD: EQU 0F1D0H
CALLB0:   EQU 0403FH
LD   HL, PROMPT
LD   (CODE_ADD),HL
CALL CALLB0
```

The "zero-based" drive number is obtained from TARGET (F33FH) and the HPROMPT hook (F24FH) is called with the "zero-based" drive number in A before the routine is performed.

9.6.4.1 – Routines for disk device drivers

DRV_SIGN (4100H / NEXTOR Kernel)
Function: Valid driver signature. Used by the Kernel at startup to verify that the driver bank contains a valid driver. Consists of the string "NEXTOR_DRIVER", without the quotes, zero-terminated and capitalized.

DRV_FLAGS (410EH / NEXTOR Kernel)
Function: Flags byte containing driver information:
bit 0: 0 – Drive-based driver.
1 – Device-based driver.
bit 1: Reserved, must be zero.
bit 2: 1 – The driver implements the DRV_CONFIG routine (used by Nextor from version 2.0.5 onwards).
bits 3-7: Reserved, always zero.

RESERVED (410FH / NEXTOR Kernel)
Function: Reserved byte, must be zero.

DRV_NAME (4110H / NEXTOR Kernel)
Function: String containing the driver name. Must consist of 32 printable ASCII characters (codes 32 to 126). It must be left justified and right padded with spaces.

DRV_TIMI (4130H / NEXTOR Kernel)
Function: Entry point for the driver's interrupt routine, called 50 or 60 times per second depending on the selected VDP frequency. If the driver does not need an interrupt, this entry must be filled with RETs. This input will only be called if DRV_INIT returns CY=1 on its first run.
Input: None.
Output: None.
Registers: None.

DRV_VERSION (4133H / NEXTOR Kernel)

Function: Return the driver version.
Input: None.
Output: A – Major version number.
B – Minor version number.
C – Revision number.
Registers: All.

DRV_INIT (4136H / NEXTOR Kernel)

Function: Driver initialization routine. Drive-based drivers should return the number of drive units needed in the output of the first run of this routine. Device-based drivers can optionally request an initial number of drives to be allocated at boot time by implementing the DRV_CONFIG routine, thus replacing the automatic mapping procedure. This routine is called by the Kernel twice:
(1) – First run, for information gathering.
Input: A = 0.
B = Number of available drive letters.
HL = Maximum Allocable Desktop Size on page 3.
C = Boot Flags:
bit 5: Requests a reduced unit count.
Output: A = Number of drive units controlled (for unit-based drivers only).
HL = Required desktop size on page 3
C = Boot Flags:
bit 5: Requests a reduced unit count.
CY = 1 if DRV_TIMI is to be connected to the interrupt timer do, 0 otherwise.
(2) – Second run, for desktop and hardware boot.
Input: A = 1.
B = Number of drive letters actually allocated to this controller.
Registers: All.
Note: If 8 bytes or less are required, this routine should return HL = 0 on its first run, and the 8-byte space reserved by the system for this slot in SLTWRK should be used as the workspace:

```
XOR  A
EX   AF, AF'
XOR  A
LD   IX, GWORK
CALL CALBNK
; IX points to desktop of 8 bytes
```

If more than 8 bytes are needed, this routine should return the space needed in HL, getting pointer to the space allocated from the first two bytes of the space reserved by the system for this slot in SLTWRK:

```
XOR   A
EX   AF, AF'
XOR  A
LD   IX, GWORK
CALL CALBNK
LD   L, (IX)
LD   H,(IX+1)
; Use the space pointed to by HL as
; workspace
```

DRV_BASSTAT (4139H / NEXTOR Kernel)

Function: Input to the Extended Instruction Handler BASIC ("CALLs"). It works in the same way as the standard handlers, except that if the handled instructions have parameters, the MSX BIOS routine CALBAS cannot be used directly; instead, routine CALLB0 on page 0 of the Kernel should be used. If the driver does not handle BASIC extended statements, it must set the transport flag (CY=1) and return.

Input: Depends on the called routine.

Output: Depends on the called routine.

Registers: Depends on the called routine.
CY = 1 if there is no instruction handler.

DRV_BASDEV (413CH / NEXTOR Kernel)

Function: Input to BASIC's Extended Device Handler. It works in the same way as standard handlers. If the driver does not handle BASIC extended devices, it should make CY=1 return.

Input: Depends on the called routine.

Output: Depends on the called routine.
Registers: Depends on the called routine.
CY = 1 if there is no instruction handler.

DRV_EXTBIO (413FH / NEXTOR Kernel)
Function: Extended BIOS Handler. It works in the same way as the standard handlers, except that it must return a value in D' (D in the register set alternatives).
Input: Depends on the called routine.
Output: D' = 0 ⟶ Return immediately.
D' = 0 ⟶ Execute Kernel and/or Extended System BIOS Handler.
Other registers: Depends on the called routine.

DRV_DIRECT0 (4142H / NEXTOR Kernel)
DRV_DIRECT1 (4145H / NEXTOR Kernel)
DRV_DIRECT2 (4148H / NEXTOR Kernel)
DRV_DIRECT3 (414BH / NEXTOR Kernel)
DRV_DIRECT5 (414EH / NEXTOR Kernel)
Function: Inputs for direct driver calls. Calls to any of the five available entry points at addresses 7850H to 785CH in Kernel ROM (bank 0 or 3) will map to a call to the corresponding DRV_DIRECT entry point. All registers except IX and AF' are passed unmodified.
Input: Depends on the called routine.
Output: Depends on the called routine.
Registers: Depends on the called routine.

DRV_CONFIG (4151H / Kernel NEXTOR v2.0.5)
Function: Allow the driver to provide configuration information at boot time. Currently, all configured settings only apply to drivers based on device. This routine is called twice.
Input: A – Configuration index.
BC, DE, HL – Depends on configuration.
Output: A = 0 ⟶ Okay.
1 ⟶ Configuration not available for index provided or unknown config index
BC, DE, HL = Depends on configuration.

(1) – To get the number of units at boot time.
Input: A = 1.
B = 0 for DOS mode 2, 1 for DOS mode 1
C = Boot Flags
bit 5: The user is requesting a unit summary count.
Output: B – Number of units.
(2) – Get the default configuration for the unit
Input: A = 2.
B = 0 for DOS 2 mode, 1 for DOS 1 mode.
C = Relative unit number at boot time.
Output: B = Device Index.
C = LUN Index.

RESERVED (4155H to 415FH / NEXTOR Kernel)
Area is reserved for future expansion (padded with zeros).

DRV_DSKIO (4160H / NEXTOR Kernel)
Function: Read or write sectors from the mass storage device associated with a drive. Unlike standard MSX-DOS routines, this routine will never receive a request to transfer data to/from page 1.
Input: A – Drive unit, starting at 0
CY – 0 for read, 1 for write
B – Number of sectors to read/write
C – First sector number for read/write (bits 22-16) if bit 7 is not set or Media ID byte if bit 7 is set.
DE – First sector for read/write (bits 15-0).
HL – Source/destination address for the transfer.
Output: CY = 1 if there was an error in the operation
A – Error code (only in case of error):
0 – write protected
2 – Not ready
4 – Data error (CRC)
6 – Search error
8 – Record not found
10 – Recording failure
12 – Other mistakes
B – Number of sectors actually read (only in case of error)

DRV_DSKCHG (4163H / NEXTOR Kernel)

Function: Obtain information about the change state of the media associated with a given drive unit.

Input: A – Drive unit, starting at 0.
B = C – Media Descriptor.
HL – Base address for DPB –1.

Output: CY = 1 if there is an error.
A – Error code (only in case of error).
The same codes as DRV_DSKIO are used.
B – Media states (if CY = 0):
1 – The media has not changed since the last time this routine was called.
0 – Unknown.
–1 – The media has changed since the last time this routine was called.

Note: If the state of the media is "Changed" or "Unknown", the routine should generate a DPB to disk and copy it to the address passed in HL plus one. The DPB format is described in the DRV_GETDPB routine.

DRV_GETDPB (4166H)

Function: Get a DPB (Drive Parameter Block) for the media associated with a given driver drive.

Input: A = driver unit, starting at 0.
B = C = Media Descriptor.
HL – Base address for DPB–1.

Output: HL points to the filled DPB (The DPB must be copied to the address passed in HL plus one).
The format of the 18-byte DPB is as follows:
HL+00: Media descriptor byte (F0h to FFh)
+01: Sector size in 2 bytes (must be power of 2).
+03: Directory mask (sector size/32 –1).
+04: Change of directory (No. 1 in the Directory mask).
+05: Cluster mask (sectors per cluster – 1).
+06: Cluster change (# of bits 1 cluster mask + 1).
+07: Number of the first sector of the FAT.
+08: Number of FATs.

+0A: Number of directory entries (maximum 254).
+0B: Number of the first data sector (2 bytes).
+0D: Maximum number of cluster (# of clusters +1).
+0F: Total number of sectors.
+10: Number of the first sector of the root directory.

DRV_CHOICE (4169H)
Function: Returns a format choice string for a disk.
Input: None.
Output: HL – Address of choice string in Kernel slot. This routine is called by the Kernel when a FORMAT command is executed, in order to show the formatting options to the user.

DRV_FORMAT (416CH)
Function: Formats a disk and initializes its boot sector, FAT and root directory.
Input: A – Formatting choice, from 1 to 9 (see DRV_CHOICE).
D – Drive unit, starting at 0.
HL – Address of the desktop in memory.
DE – Size of the workspace.
Output: CY – 1 if there is an error..
A – Error code (only in case of error):
0 – Write protected.
2 – Not ready.
4 – CRC error.
6 – Search error.
8 – Record not found.
10 – Write failure.
12 – Bad parameter.
14 – Insufficient memory.
16 – Other errors.

DRV_MTOFF (416FH)
Function: Stop the motor of all drives. Useful for floppy drives only.
Input: None.
Output: None.

9.6.4.2 – Routines for drivers of other devices

DEV_RW (4160H)

Function: Read or write absolute sectors to or from a device.

Input:
- CY – 0 for reading; 1 for writing.
- A – Device Index (1 to 7).
- B – Number of sectors to read or write.
- C – Logical unit index (1 to 7).
- HL – Transfer address (cannot be on page 1).
- DE – Address for storing the 4-byte sector number (cannot be on page 1).

Output:
- A – Error code (same codes as MSXDOS2):
  - 00H: There was no error.
  - B5H: Invalid logical device/drive number.
  - F3H: Fetch error.
  - F7H: Disk not formatted.
  - F8H: Write protected or read-only drive.
  - F9H: Sector not found.
  - FAH: CRC error while reading.
  - FCH: Not ready.
  - FDH: Disk error.
  - FEH: Writing error.
  - FFH: Incompatible disk.
- B – Number of sectors actually read or written (only in case of error).

DEV_INFO (4163H)

Function: Return information about a device.

Input:
- A – Device Index (1 to 7).
- B – 0 ⟶ Basic information.
  - 1 ⟶ String containing the manufacturer's name.
  - 2 ⟶ String containing the device name.
  - 3 ⟶ String containing the serial number.
- HL – Pointer to buffer (cannot be on page 1).

Output:
- A – Error code:
  - 0 ⟶ There was no error.
  - 1 ⟶ Device / information not available or invalid index.

HL – Buffer filled with the text string or with basic information in the following format:
+0: Number of logical units (1 to 8). Must be 1 if there are no logical drives.
+1: Flags with device capabilities (00H in current version).

DEV_STATUS (4166H)
Function: Check availability and change the state of a device or logical unit.
Input: A – Device Index (1 to 7)
B – Logical unit number (1 to 7) or 0 to return the state of the device itself.
Output: A – State for the specified logical unit or for the device if 0 was specified:
0 – The device or logical unit is not available, or the device / logical unit number is invalid.
1 – The device or logical drive is available and has not changed since the last state check.
2 – The device or logical drive is available and has changed since the last state check (for devices, the device was disconnected and another was connected to which the same index was assigned; for logical drives, the media has been changed).
3 – The device or logical drive is available, but it is not possible to determine whether or not it has changed since the last state check.

LUN_INFO (4169H)
Function: Get information for a logical drive.
Input: A – Device index (1 to 7).
B – Logical unit index (1 to 7).
HL – Pointer to buffer (cannot be on page 1).
Output: A – 0 ⟶ Ok, buffer filled with information.
1 ⟶ Device or logical drive not available or invalid.
HL – 12-byte buffer filled.
+0: Media type:
0 – Lock device.
1 – CD or DVD reader or writer.

2-254 – Not used (reserved for future use).
255 – Another type.
+1: Sector size in 2 bytes (0 if this information does not apply or is not available).
+3: Total available sectors in 4 bytes (0 if this information does not apply or is not available).
+7: Logical drive flags:
bit 0: 1 if the media is removable.
bit 1: 1 if the media is read-only.
bit 2: 1 if the logical drive is a floppy drive.
bit 3: 1 if the logical drive is not to be used for automapping.
bits 4-7: Not used (always 0).
+8: Number of cylinders (2 bytes).
+10: Number of heads (1 byte).
+11: Number of sectors per track (1 byte).

9.6.4 – Routines for accessing standard SCSI Hard-Disks

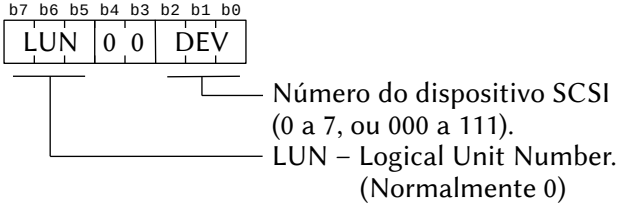
IDBYT (7F80H/SCSI Interface)
Function: Interface ID in 3 bytes. (Ex.: "HD#").

INISYS (7F83H/SCSI Interface)
Function: Starts SCSI interface.
Input: None.
Output: None.
Registers: All.

TRMACT (7F86H/SCSI Interface)
Function: Terminates HDD actions.
Input: None.
Output: A – SCSI interface status. Same as RDLBLK (7F89H).
D – Current Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: AF, DE.

RDLBLK (7F89H/SCSI Interface)
Function: Read logical sectors from disk or device.

Input: CDE – Sector number.
HL – RAM address for read data.
B – Number of sectors to read.
A – Device ID:



Output: HL – Pointer to the read data.
A – SCSI interface status.
00H – There was no error.
02H – Check condition.
04H – "MET" condition.
08H – Device busy.
0CH – Booking conflict.
10H – Intermediate condition.
14H – Intermediate condition "MET".
18H – Reservation conflict.
22H – Command finished.
28H – Full queue.
30H – ACA active.
40H – Operation aborted.
D – Current device status.
00H – There was no error.
02H – Check condition.
04H – "MET" condition.
08H – Device busy.
0CH – Booking conflict.
10H – Intermediate condition.
14H – Intermediate condition "MET".
18H – Reservation conflict.
22H – Command finished.
28H – Full queue.
30H – ACA active.
40H – Operation aborted.

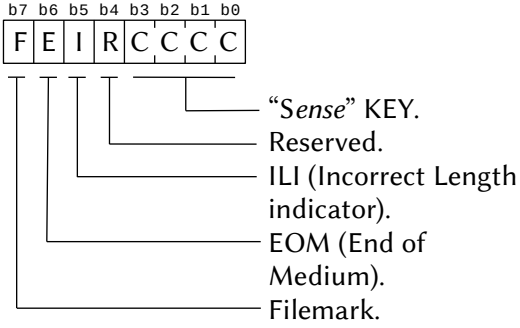
E – Messages:
00H – Complete command.
01H, xx, 00H – Modify given pointers.
01H, xx, 01H – Request for transfer synchronous data.
01H, xx, 03H – Request for transfer total data.
02H – Save data pointers.
03H – Restore pointers.
04H – Disconnect.
05H – Initialization error.
06H – Abort.
07H – Message rejected.
08H – No operation.
09H – Message parity error.
0AH – Command attached complete.
0BH – Complete attached command (with flag).
0CH – Reset on device bus.
0DH – Abort TAG.
0EH – Clean/Empty Queue.
0FH – Start recovery.
10H – Release recovery.
11H – End I/O process.
20H – Single row tag.
21H – Queue header tag
22H – Ordered queue tag.
23H – Ignore waste.
80H ~ 0FFH – Identify.

Registers: All.
Note: This routine can also read sectors from the CD-ROM, which have 2048 bytes instead of 512 bytes from the HD's.

WRLBLK (7F8CH/SCSI Interface)
Function: Write logical sectors of the disk.
Input: CDE – Sector number.
HL – Starting address of the data to be written.
B – Number of sectors to write.
A – Device ID. Same as RDLBLK (7F89H).
Output: HL – Pointer to the read data.
A – SCSI Status. Same as RDLBLK (7F89H).

D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: All.

RQSENS (7F8FH/SCSI Interface)
Function: Returns "sense" information about the SCSI device.
Input: A – Device ID. Same as RDLBLK (7F89H).
Output: A – DOS error code.
IX – Pointer to a buffer filled with "sense" data:
+00H – Error code:
70H – Fixed format, current "sense".
71H – Fixed format, "sense" previous.
72H – Descriptor format, current "sense".
73H – Descriptor format, "sense" previous.
+01H – Segment number.
+02H –



"Sense" key codes:
00H – No sense
01H – Error recovered
02H – Not ready
03H – Media error
04H – Hardware error
05H – Illegal request
06H – Unit requires attention
07H – Data protected
08H – Blank verification
09H – Manufacturer Specific
0AH – Copy aborted
0BH – Command aborted

0DH – Volume overflow
0EH – Agreement error
0FH – Completed
+03H~+06H – Information
+07H – Additional "sense" length (n-7)
+08H~+11H – Command specific information
+12H – Additional "sense" code
+13H – Additional "sense" code qualifier
+14H – Replaceable unit code
+15H – Bit7 = 0 ⟶ There is no valid information
1 ⟶ There is valid information
+15H (bit6~bit0)~+17H – Manufacturer specific information

Registers: AF, BC, DE.

INQIRY (7F92H/SCSI Interface)

Function: Returns SCSI device information.
Input: HL – Buffer address for read information.
A – Device ID.
Output: CY = 1 ⟶ reading error.
A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
CY = 0 ⟶ HL – Points to the beginning of the buffer:
+00H – device code.
+01H – bit7 ⟶ RMB (removable media)
bit6~bit0 ⟶ device type
+02H – Interface version:
00H – Not specified
01H – SCSI 1
02H – SCSI 2
+03H – bit7~bit4 ⟶ Reserved.
bit3~bit0 ⟶ Response data format.
+04H – Additional length, contains how many subsequent bytes are valid.
+05H~+07H – Reserved.
+08H~+15H – Name (Ex. SEAGATE).
+16H~+31H – Device ID (in ASCII).
+32H – Hardware revision.

+33H – Firmware revision.
+34H – ROM revision.
+35H – Reserved.

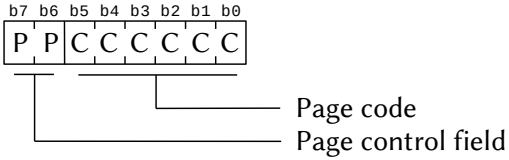
Registers: All.

RDSIZE (7F95H/SCSI Interface)

Function: Returns the total space of the SCSI device.
Input: HL – Buffer address for read information.
A – Device ID. Same as RDLBLK (7F89H).
Output: CY = 1 ⟶ reading error.
A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
CY = 0 ⟶ data read successfully.
(HL+0)~(HL+3) ⟶ total number of sectors (MSB/LSB).
(HL+4)~(HL+7) ⟶ sector size in bytes
(MSB/LSB). Typically 512 (00H-00H-02H-00H).
Registers: All.

MDSENS (7F98H/SCSI Interface)

Function: Returns the "sense" parameters of the current mode.
Input: HL – Buffer address for read information.
A – Device ID. Same as RDLBLK (7F89H).



Output: CY = 1 ⟶ reading error.
A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
CY = 0 ⟶ HL – Points to the beginning of the buffer:
+00H – Operating parameters (SEAGATE).
+01H – Error recovery parameters.
+02H – Disconnected parameters.
+03H – Format parameters.
+04H – Geometry parameters.
+05H~+1FH – Reserved.

+20H – Drive serial number.
+3FH – Returns all pages.

Registers: All.

MDSEL (7F9BH/SCSI Interface)

Function: Mode selection. Used to boot HD.

Input: HL – Buffer address.
A – Device ID. Same as RDLBLK (7F89H).
B – Size of the parameter list.

Output: CY = 1 ⟶ reading error.
A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
CY = 0 ⟶ HL points to the parameter list.

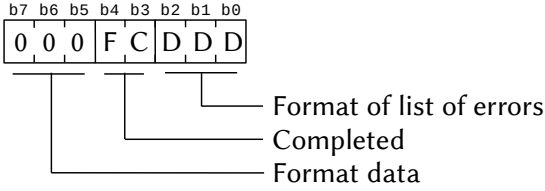
Registers: AF, BC, HL, IX.

HDFORM (7F9EH/SCSI Interface)

Function: Format the SCSI drive.

Input: A – Unit ID.

B –



DE – Interleave (MSB-LSB).
HL – Data address.

Output: CY = 1 ⟶ reading error.
A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
CY = 0 ⟶ Formatted successfully.

Registers: AF, BC, DE, HL.

TESTRD (7FA1H/SCSI Interface)

Function: Tests whether the SCSI device is ready.

Input: A – Device ID. Same as RDLBLK (7F89H).

Output: A = 85H ⟶ device is ready.
A = 42H ⟶ the device is NOT ready.

Registers: All.

SFBOOT (7FA4H/SCSI Interface)
Function: Softboot the SCSI device.
Input: None.
Output: None.
Registers: All.
Note: This entry must not be used.

INSWRK (7FA7H/SCSI Interface)
Function: Mounts SCSI device table (installs desktop).
Input: None.
Output: None.
Registers: All.
Note: This input must not be used (internal routine).

CLRLIN (7FAAH/SCSI Interface)
Function: Cleans to end of line (prints ESC sequence).
Input: None.
Output: None.
Registers: All.

VERIFY (7FADH/SCSI Interface)
Function: Device verification.
Input: A – Device ID. Same as RDLBLK (7F89H).
B – Size to be checked (in blocks).
CDE – Logical block number.
HL – Address.
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: AF, BC, HL, IX.

STRSTP (7FB0H/SCSI Interface)
Function: Starts or stops the drive.
Input: A – Device ID. Same as RDLBLK (7F89H).
B = 0 ⟶ Stops drive.
1 ⟶ Start the drive.
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: All.

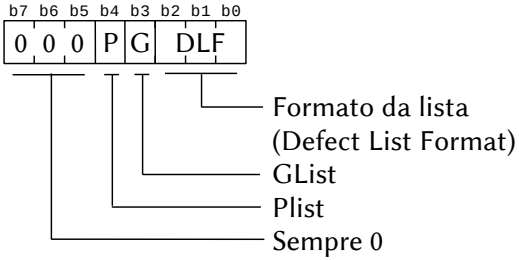
SNDDGN (7FB3H/SCSI Interface)
Function: Sends diagnostics.
Input: A – Device ID. Same as RDLBLK (7F89H).
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: All.

RESERV (7FB6H/SCSI Interface)
Function: Reserved.

RESER2 (7FB9H/SCSI Interface)
Function: Reserved.

COPY (7FBCH/SCSI Interface)
Function: Read "default" list.
Input: A – Device ID. Same as RDLBLK (7F89H).
DE – Length of the parameter list.
HL – Data address.
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Attention: This input has different function on IDE interfaces. It is not advisable to use this call.

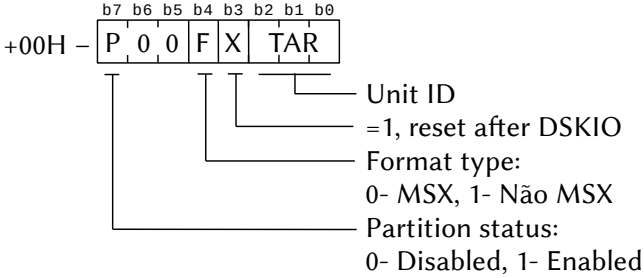
RDEFCT (7FBFH/SCSI Interface)
Function: Returns corrupted data.
Input: A – Device ID. Same as RDLBLK (7F89H).
B –



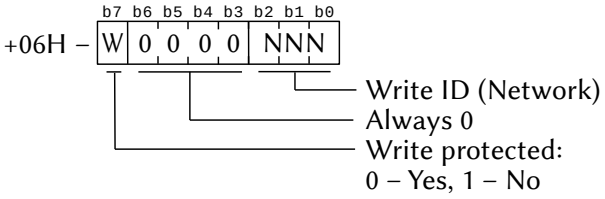
DE – Size of allocated space.
HL – Data address.

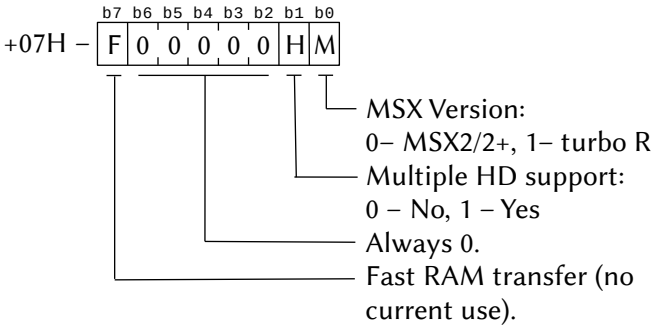
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: All.

GETWRK (7FC2H/SCSI Interface)
Function: Returns the desktop address.
Input: None.
Output: HL = IX = Pointer to start of workspace. 8 bytes are reserved for each logical drive (there can be up to 6 logical drives, A: through F:). The structure for each unit is as follows:



+01H~+03H – Primeiro setor da partição.
+04H~+05H – Número de setores da partição.





Registers: AF, BC, HL, IX.

PRTINF (7FC5H/SCSI Interface)
Function: Returns information about the partition.
Input: A – Drive number
Output: HL = IX = Pointer to the beginning of the workspace of the specified drive. There are 8 bytes with the same structure as GETWRK (7FC2H).
Registers: AF, BC, DE, HL, IX.

GTUNIT (7FC8H/SCSI Interface)
Function: Returns the number of active units.
Input: None.
Output: A – Number of active units.
C – Vector ID.
D – Host ID.
Registers: AF, BC, DE.

HOSTID (7FCBH/SCSI Interface)
Function: Select the Host ID.
Input: A – Host ID (4~7)
Output: CY = 1 ⟶ error.
Registers: AF, D.

TARGID (7FCEH/SCSI Interface)
Function: Select the Target ID.
Input: A – Target ID (0~3)
Output: CY = 1 ⟶ error.
Registers: AF, D.

GTTARG (7FD1H/SCSI Interface)
Function: Returns the Target ID.
Input: None.
Output: A – Target ID.
Registers: AF.

GTHOST (7FD4H/SCSI Interface)
Function: Returns the Host ID.
Input: None.
Output: A – Host ID.
Registers: AF.

GTSENS (7FD7H/SCSI Interface)
Function: Returns "sense" data.
Input: A – Device ID. Same as RDLBLK (7F89H).
Output: A – Key "sense".
C – Additional "sense" code.
D – Target Status
IX – Address data "sense". Same as RQSENS (7F8FH).
Registers: AF, BC, DE.

MEDREM (7FDAH/SCSI Interface)
Function: Prevent media removal.
Input: A – Device ID. Same as RDLBLK (7F89H).
B = 0 ⟶ allows removal
1 ⟶ prevent removal
Output: A – SCSI Status. Same as RDLBLK (7F89H).
D – Device Status. Same as RDLBLK (7F89H).
E – Messages. Same as RDLBLK (7F89H).
Registers: All.

9.7 – MSX-MUSIC ROUTINES (FM/OPLL)

WRTOPL (4110H/FM-BIOS)
Function: Writes a byte of data into an OPLL register.
Input: A – OPLL Register
E – Data byte to be written
Output: None.
Registers: None.

INIOPL (4113H/FM-BIOS)
Function: Initializes the FM-BIOS/OPLL desktop.
Input: HL – Desktop start (must be even).
Output: None.
Registers: All.

MSTART (4116H/FM-BIOS)
Function: Starts playing music.
Input: HL – Music queue address.

A = 0 → Infinite loop.
1~254 → Number of repetitions.
255 → Reserved. Do not use.

The musical queue has the structure described below.

Header for 6 FM voices + 5 drum pieces:

```
+00~+01 0EH, 00H
+02~+03 Address for FM1CH
+04~+05 Address for FM2CH
+06~+07 Address for FM3CH
+08~+09 Address for FM4CH
+10~+11 Address for FM5CH
+12~+13 Address for FM6CH
+14 ... Data area
```

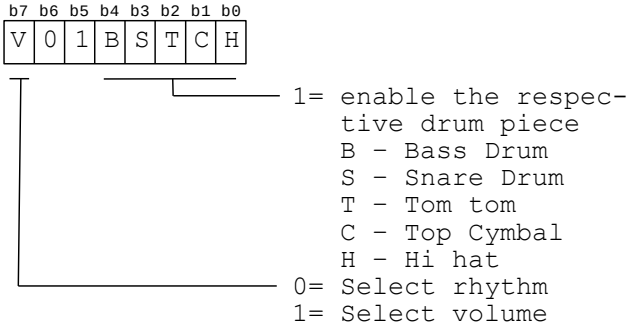
Header for 9 FM voices:

```
+00~+01 12H, 00H
+02~+03 Address for FM1CH
+04~+05 Address for FM2CH
+06~+07 Address for FM3CH
+08~+09 Address for FM4CH
+10~+11 Address for FM5CH
+12~+13 Address for FM6CH
+14~+15 Address for FM7CH
+16~+17 Address for FM8CH
+18~+19 Address for FM9CH
+20 ... Data area
```

Data area for melody:

```
+00H~+5FH Specifies the pitch. This number
  represents all musical scales, including
  the "pitch"
+60H~+6FH Volume
+70H~+7FH Instrument
+80H Release of "Sustain"
+81H Maintenance of "Sustain"
+82H Enable ROM instrument (0 to 63)
+83H Specify User Instrument
+84H Turn off legato
+85H Turn on legato
+86H Q designation (1 to 8). When legato is
     on, the Q assignment is not performed.
+87H~+FEH Not used
+FFH End of data for each voice
```

Data area for rhythm:



FFH → end of rhythm data.

Instrument data storage format:

+0	AM	VIB	EG TYP	KSR	MULTIPLE	
+1						
+2	KSL M		TOTAL LEVEL			
+3	KSL C		XX	DC	DM	FEEDBACK
+4	ATTACK RATE				DECAY RATE	
+5						
+6	SUSTAIN LEVEL				RELEASE RATE	
+7						

Output: None.
Registers: All.

MSTOP (4119H/FM-BIOS)
Function: Stop the music.
Input: None.
Output: None.
Registers: All.

RDDATA (411CH/FM-BIOS)
Function: Returns instrument data from ROM.
Input: HL – Buffer address for read data.
A – Instrument number (0 to 63).
Output: None.
Registers: F.

OPLDRV (411FH/FM-BIOS)

Function: Input for OPLL driver. It is the routine that plays the music and must be called by the interrupt handler via the HTIMI hook.
Input: None.
Output: None.
Registers: None.

TSTBGM (4122H/FM-BIOS)

Function: Checks if there is still data in the music queue.
Input: None.
Output: A = 0 ⟶ no music being played
A ≠ 0 ⟶ music is being played.
Registers: AF.

10 – MSX-HID (Human Interface Device)

Formula for the unique ID byte:

```
HIDID=(byte1<<4|0xF)&(byte2|0xC0)&(byte1<<2|0x3F)&0xFF
```

10.1 – FINGERPRINTS OF MSX DEVICES

Unconnected, or MSX-joystick	3Fh,3Fh,3Fh
Mouse	30h,30h,30h
Trackball	38h,38h,38h
Touchpad(1)	39h,3Dh,39h
Touchpad(2)	3Dh,3Dh,3Dh
Lightgun	2Fh,2Fh,2Fh
Arkanoid Vaus Paddle	3Eh,3Eh,3Eh
Time encoded devices (each bit of "xx" is zero for each analog channel present)	xxh,3Fh,3Fh
MSX-Paddle	3Eh,3Fh,3Fh
Yamaha MMP-01 music pad	3Ch,3Fh,3Fh
IBM-PC DA15 joystick adapter	3Ah,3Fh,3Fh
Atari dual-paddle adapter	36h,3Fh,3Fh
Dual-axis analog controller	30h,3Fh,3Fh

10.2 – FINGERPRINTS OF SEGA COMPATIBLE DEVICES

Megadrive 3-button joypad	3Fh,33h,3Fh
Megadrive 6-button joypad	3Fh,33h,3Fh,33h,3Fh,30h
Megadrive Multi-Tap	33h,3Fh,33h
Saturn digital joypad	3Ch,3Fh,3Ch
Saturn Mouse	30h,3Bh,30h
Sega 3line-handshake device	31h,31h,31h

10.3 – FINGERPRINTS OF DEVICES THAT CONFLICT

The following devices can conflict with other MSX-HID devices. If necessary, both cases can be distinguished from the other device with one extra detection step.

Micomsoft XE1-AP analog mode	2Fh,2Fh,2Fh
Sega-Mouse (Megadrive)	30h,30h,30h

10.4 – HOMEBREW DEVICES

Ninja-tap	`3Fh,1Fh,3Fh`
3D glasses	`3Fh,37h,3Fh`
3D glasses + light gun	`2Fh,27h,2Fh`
Passive PS/2 mouse adapter	`3Fh,3Eh,3Fh`

10.5 – RESERVED FINGERPRINTS (DO NOT USE)

- → Any fingerprints that can be produced by a standard MSX joystick.
- → Any fingerprints that set both the pin-6 and pin-7 of the joystick port to 0 simultaneously on the two first bytes.

11 – Z80/R800 MNEMONICS

11.1 – 8-BIT LOAD GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
LD r,r'	r ← r'	•	•	•	•	•	•	01 r r'	--	04	05	01	01
LD r,n	r ← n	•	•	•	•	•	•	00 r 110 ← n →	-- --	07	08	02	02
LD u,u'	u ← u'	•	•	•	•	•	•	11 011 101 01 u u'	DD --	08	10	02	02
LD v,v'	v ← v'	•	•	•	•	•	•	11 111 101 01 v v'	FD --	08	10	02	02
LD u,n	u ← n	•	•	•	•	•	•	11 011 101 00 u 110 ← n →	DD -- --	11	13	03	03
LD v,n	u ← n	•	•	•	•	•	•	11 111 101 00 v 110 ← n →	FD -- --	11	13	03	03
LD r,(HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	--	07	08	02	04
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11 011 101 01 r 110 ← d →	DD -- --	19	21	05	07
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11 111 101 01 r 110 ← d →	FD -- --	19	21	05	07
LD (HL),r	(HL) ← r	•	•	•	•	•	•	01 110 r	--	07	08	02	04
LD (IX+d),r	(IX+d) ← r	•	•	•	•	•	•	11 011 101 01 110 r ← d →	DD -- --	19	21	05	07
LD (IY+d),r	(IY+d) ← r	•	•	•	•	•	•	11 111 101 01 110 r ← d →	FD -- --	19	21	05	07
LD (HL),n	(HL) ← n	•	•	•	•	•	•	00 110 110 ← n →	36 --	10	11	03	05
LD (IX+d),n	(IX+d) ← n	•	•	•	•	•	•	11 011 101 01 110 110 ← d → ← n →	DD 36 -- --	19	21	05	07

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
LD (IY+d),n	(IY+d) ← n	• • • • • •	11 111 101 01 110 110 ← d → ← n →	FD 36 -- --	19	21	05	07
LD A,(BC)	A ← (BC)	• • • • • •	00 001 010	0A	07	08	02	04
LD A,(DE)	A ← (DE)	• • • • • •	00 011 010	1A	07	08	02	04
LD A,(nn)	A ← (nn)	• • • • • •	00 111 010 ← n → ← n →	1A -- --	13	14	04	06
LD (BC),A	(BC) ← A	• • • • • •	00 000 010	02	07	08	02	04
LD (DE),A	(DE) ← A	• • • • • •	00 010 010	22	07	08	02	04
LD (nn),A	(nn) ← A	• • • • • •	00 110 010 ← n → ← n →	32 -- --	13	14	04	06
LD A,I	A ← I	• ↕ I ↕ • •	11 101 101 01 010 111	ED 57	09	11	02	02
LD A,R	A ← R	• ↕ I ↕ • •	11 101 101 01 011 111	ED 5F	09	11	02	02
LD I,A	I ← A	• • • • • •	11 101 101 01 000 111	ED 47	09	11	02	02
LD R,A	R ← A	• • • • • •	11 101 101 01 001 111	ED 4F	09	11	02	02

	000	001	010	011	100	101	110	111
r,r'	B	C	D	E	H	L	•	A
u,u'	B	C	D	E	IXH	IXL	•	A
v,v'	B	C	D	E	IYH	IYL	•	A

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
I = The IFF content is copied to the P/V flag.

11.2 – 16-BIT LOAD GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
LD dd,nn	dd ← nn	•	•	•	•	•	•	00 dd0 001 ← n → ← n →	-- -- --	10	11	03	03
LD IX,nn	IX ← nn	•	•	•	•	•	•	11 011 101 00 100 001 ← n → ← n →	DD 21 -- --	14	16	04	04
LD IY,nn	IY ← nn	•	•	•	•	•	•	11 111 101 00 100 001 ← n → ← n →	FD 21 -- --	14	16	04	04
LD HL,(nn)	H ← (nn+1) L ← (nn)	•	•	•	•	•	•	00 101 010 ← n → ← n →	2A -- --	16	17	05	07
LD dd,(nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	•	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	ED -- -- --	20	22	06	08
LD IX,(nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	•	•	•	•	11 011 101 00 101 010 ← n → ← n →	DD 2A -- --	20	22	06	08
LD IY,(nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	•	•	•	•	11 111 101 00 101 010 ← n → ← n →	FD 2A -- --	20	22	06	08
LD (nn),HL	(nn+1) ← H (nn) ← L	•	•	•	•	•	•	00 100 010 ← n → ← n →	22 -- --	16	17	05	07
LD (nn),dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	•	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	ED -- -- --	20	22	06	08
LD (nn),IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	•	•	•	•	11 011 101 01 100 010 ← n → ← n →	DD 22 -- --	20	22	06	08

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
LD (nn),IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	•	•	•	•	11 111 101 01 100 010 ← n → ← n →	FD 22 -- --	20	22	06	08
LD SP,HL	SP ← HL	•	•	•	•	•	•	11 111 001	F9	06	07	01	01
LD SP,IX	SP ← IX	•	•	•	•	•	•	11 011 101 11 111 001	DD F9	10	12	02	02
LD SP,IY	SP ← IY	•	•	•	•	•	•	11 111 101 11 111 001	FD F9	10	12	02	02
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H SP ← SP - 2	•	•	•	•	•	•	11 qq0 101	--	11	12	04	06
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H SP ← SP - 2	•	•	•	•	•	•	11 011 101 11 100 101	DD E5	15	17	05	07
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H SP ← SP - 2	•	•	•	•	•	•	11 111 101 11 100 101	FD E5	15	17	05	07
POP qq	qq_H ← (SP+1) qq_L ← (SP) SP ← SP + 2	•	•	•	•	•	•	11 qq0 001	--	10	11	03	05
POP IX	IX_H ← (SP+1) IX_L ← (SP) SP ← SP + 2	•	•	•	•	•	•	11 011 101 11 100 001	DD E1	14	16	04	06
POP IY	IY_H ← (SP+1) IY_L ← (SP) SP ← SP + 2	•	•	•	•	•	•	11 111 101 11 100 001	FD E1	14	16	04	06

	00	01	10	11
dd	BC	DE	HL	SP
qq	BC	DE	HL	AF

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected

11.3 – 8-BIT ARITHMETIC GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
ADD r	A ← A + r	↕	↕	V	↕	0	↕	10 000 r	--	04	05	01	01
ADD p	A ← A + p	↕	↕	V	↕	0	↕	11 011 101 10 000 r	DD --	08	10	02	02
ADD q	A ← A + q	↕	↕	V	↕	0	↕	11 111 101 10 000 r	FD --	08	10	02	02
ADD (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 000 110	86	07	08	02	04
ADD (IX+d)	A←A+(IX+d)	↕	↕	V	↕	0	↕	11 011 101 10 000 110 ← d →	DD 86 --	19	21	05	07
ADD (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101 10 000 110 ← d →	FD 86 --	19	21	05	07
ADD n	A ← A + n	↕	↕	V	↕	0	↕	11 000 110 ← n →	C6	07	08	02	02
ADC r	A ← A+r+CY	↕	↕	V	↕	0	↕	10 001 r	--	04	05	01	01
ADC p	A ← A+p+CY	↕	↕	V	↕	0	↕	11 011 101 10 001 r	DD --	08	10	02	02
ADC q	A ← A+q+CY	↕	↕	V	↕	0	↕	11 111 101 10 001 r	FD --	08	10	02	02
ADC (HL)	A←A+(HL)+CY	↕	↕	V	↕	0	↕	10 001 110	8E	07	08	02	04
ADC (IX+d)	A←A+(IX+d)+CY	↕	↕	V	↕	0	↕	11 011 101 10 001 110 ← d →	DD 8E --	19	21	05	07
ADC (IY+d)	A←A+(IY+d)+CY	↕	↕	V	↕	0	↕	11 111 101 10 001 110 ← d →	FD 8E --	19	21	05	07
ADC n	A ← A+n+CY	↕	↕	V	↕	0	↕	11 001 110 ← n →	CE	07	08	02	02
SUB r	A ← A – r	↕	↕	V	↕	1	↕	10 010 r	--	04	05	01	01
SUB p	A ← A – p	↕	↕	V	↕	1	↕	11 011 101 10 010 r	DD --	08	10	02	02
SUB p	A ← A - q	↕	↕	V	↕	1	↕	11 111 101 10 010 r	FD --	08	10	02	02

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
SUB (HL)	A ← A - (HL)	↕	↕	V	↕	1	↕	10 010 110	96	07	08	02	04
SUB (IX+d)	A←A-(IX+d)	↕	↕	V	↕	1	↕	11 011 101 10 010 110 ← d →	DD 96 --	19	21	05	07
SUB (IY+d)	A←A-(IY+d)	↕	↕	V	↕	1	↕	11 111 101 10 010 110 ← d →	FD 96 --	19	21	05	07
SUB n	A ← A - n	↕	↕	V	↕	1	↕	11 010 110 ← n →	D6	07	08	02	02
SBC r	A ← A-r-CY	↕	↕	V	↕	1	↕	10 011 r	--	04	05	01	01
SBC p	A ← A-p-CY	↕	↕	V	↕	1	↕	11 011 101 10 011 r	DD --	08	10	02	02
SBC p	A ← A-q-CY	↕	↕	V	↕	1	↕	11 111 101 10 011 r	FD --	08	10	02	02
SBC (HL)	A←A-(HL)-CY	↕	↕	V	↕	1	↕	10 011 110	8E	07	08	02	04
SBC (IX+d)	A←A-(IX+d)-CY	↕	↕	V	↕	1	↕	11 011 101 10 011 110 ← d →	DD 8E --	19	21	05	07
SBC (IY+d)	A←A-(IY+d)-CY	↕	↕	V	↕	1	↕	11 111 101 10 011 110 ← d →	FD 8E --	19	21	05	07
SBC n	A ← A-n-CY	↕	↕	V	↕	1	↕	11 011 110 ← n →	CE	07	08	02	02
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r 100	--	04	05	01	01
INC (HL)	(HL)←(HL)+1	•	↕	V	↕	0	↕	00 110 100	--	11	12	04	07
INC (IX+d)	(IX+d)← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101 00 110 100 ← d →	DD 34 --	23	25	07	10
INC (IY+d)	(IY+d)← (IY+d)+1	•	↕	V	↕	0	↕	11 111 101 00 110 100 ← d →	FD 34 --	23	25	07	10
DEC r	r ← r - 1	•	↕	V	↕	1	↕	00 r 101	--	04	05	01	01
DEC (HL)	(HL)←(HL)-1	•	↕	V	↕	1	↕	00 110 101	--	11	03	04	07

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
DEC (IX+d)	(IX+d)← (IX+d)-1	•	↕	V	↕	1	↕	11 011 101 00 110 101 ← d →	DD 34 --	23	25	07	10
DEC (IY+d)	(IY+d)← (IY+d)-1	•	↕	V	↕	1	↕	11 111 101 00 110 101 ← d →	FD 34 --	23	25	07	10
MULB r	HL ← A * r	↕	↕	0	0	•	•	11 101 101 11 r 001	ED --	--	--	14	14

	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	•	A
p	•	•	•	•	IXH	IXL	•	•
q	•	•	•	•	IYH	IYL	•	•

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected.
↕ = Flag affected according operation results.
0 = Flag off.
1 = Flag on.
V = the P/V flag contains the overflow status: V=1 -> overflow; V=0 -> there was no overflow.
P = the P/V flag contains the parity status. P=1 means the parity of the result is even; P=0 means it is odd.

11.4 – 16-BIT ARITHMETIC GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
ADD HL,ss	HL ← HL + ss	↕ • • • 0 ?	00 ss1 001	--	11	12	01	01
ADD IX,pp	IX ← IX + ss	↕ • • • 0 ?	11 011 101 00 pp1 001	DD --	15	17	02	02
ADD IY,rr	IY ← IY + ss	↕ • • • 0 ?	11 111 101 00 rr1 001	FD --	15	17	02	02
ADC HL,SS	HL←HL+ss+CY	↕ ↕ V ↕ 0 ?	11 101 101 01 ss1 010	ED --	15	17	02	02
SBC HL,SS	HL←HL-ss-CY	↕ ↕ V ↕ 1 ?	11 101 101 01 ss0 010	ED --	15	17	02	02
INC ss	ss ← ss + 1	• • • • • •	00 ss0 011	--	06	07	01	01
INC IX	IX ← IX + ss	• • • • • •	11 011 101 00 100 011	DD 23	10	12	02	02
INC IY	IY ← IY + ss	• • • • • •	11 111 101 00 100 011	FD 23	10	12	02	02
DEC ss	ss ← ss - 1	• • • • • •	00 ss1 011	--	06	07	01	01
DEC IX	IX ← IX - ss	• • • • • •	11 011 101 00 101 011	DD 2B	10	12	02	02
DEC IY	IY ← IY - ss	• • • • • •	11 111 101 00 101 011	FD 2B	10	12	02	02
MULW HL,tt	DE:HL ← HL * tt	↕ ↕ 0 0 • •	11 101 101 11 tt0 011	ED --	--	--	36	36

	00	01	10	11
ss	BC	DE	HL	SP
pp	BC	DE	IX	SP
rr	BC	DE	IY	SP
tt	BC	•	•	SP

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected.
↕ = Flag affected according operation results.
0 = Flag off.
1 = Flag on.
? = Flag unknown.
V = the P/V flag contains the overflow status: V=1 -> overflow;
V=0 -> there was no overflow.
P = the P/V flag contains the parity status. P=1 means the
parity of the result is even; P=0 means it is odd.

11.5 – EXCHANGE GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
EX DE,HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	EB	04	05	01	01
EX AF,AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	08	04	05	01	01
EXX	BC ↔ BC' DE ↔ DE' HL ↔ HL'	•	•	•	•	•	•	11 011 001	D9	04	05	01	01
EX (SP),HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	E3	19	20	05	07
EX (SP),IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	DD E3	23	25	06	08
EX (SP),IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	FD E3	23	25	06	08

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected

11.6 – BLOCK TRANSFER GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	↕	•	0	0	11 101 101 10 100 000	ED A0	16	18	04	07
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 {Até BC=0}	•	•	0	•	0	0	11 101 101 10 110 000	ED B0	21 16	23 18	04 04	? 07
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	↕	•	0	0	11 101 101 10 101 000	ED A8	16	18	04	07
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 {Até BC=0}	•	•	0	•	0	0	11 101 101 10 111 000	ED B8	21 16	23 18	04 04	? 07

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
0 = Flag off

NOTE: When there are two descriptions of cycles, they refer to the two conditions that the instruction can assume. Thus, for LDIR, the time in T cycles for the Z80 is 21; when BC reaches 0, 16 T cycles are spent.

11.7 – SEARCH GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
CPI	A – (HL) HL ← HL+1 BC ← BC-1	• ↕ ↕ ↕ 1 ↕	11 101 101 10 100 001	ED A1	16	18	04	06
CPIR	A – (HL) HL ← HL+1 BC ← BC-1 {Até BC=0 ou A=(HL)}	• ↕ ↕ ↕ 1 ↕	11 101 101 10 110 001	ED B1	21 16	23 18	05 05	? 08
CPD	A – (HL) HL ← HL-1 BC ← BC-1	• ↕ ↕ ↕ 1 ↕	11 101 101 10 101 001	ED A9	16	18	04	06
CPDR	A – (HL) HL ← HL-1 BC ← BC-1 {Até BC=0 ou A=(HL)}	• ↕ ↕ ↕ 1 ↕	11 101 101 10 111 001	ED B9	21 16	23 18	05 05	? 08

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
1 = Flag on

NOTE: When there are two descriptions of cycles, they refer to the two conditions that the instruction can assume. Thus, for LDIR, the time in T cycles for the Z80 is 21; when BC reaches 0, 16 T cycles are spent.

11.8 – COMPARISON GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
CP A,r	A – R	↕ ↕ v ↕ 1 ↕	10 111 r	--	04	05	01	01
CP A,p	A – p	↕ ↕ v ↕ 1 ↕	11 011 101 10 111 p	DD --	08	10	02	02
CP A,q	A – q	↕ ↕ v ↕ 1 ↕	11 111 101 10 111 p	FD --	08	10	02	02
CP A,(HL)	A – (HL)	↕ ↕ v ↕ 1 ↕	10 111 110	BE	07	08	02	04
CP A,(IX+d)	A – (IX+d)	↕ ↕ v ↕ 1 ↕	11 011 101 10 111 110 ← d →	DD BE --	19	21	05	07
CP A,(IY+d)	A – (IY+d)	↕ ↕ v ↕ 1 ↕	11 111 101 10 111 110 ← d →	FD BE --	19	21	05	07
CP A,n	A – n	↕ ↕ v ↕ 1 ↕	11 111 110 ← n →	FE --	07	08	02	02

	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	•	A
p	•	•	•	•	IXH	IXL	•	•
q	•	•	•	•	IYH	IYL	•	•

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
↕ = Flag affected according operation results.
1 = Flag on.
V = the P/V flag contains the overflow status: V=1 -> overflow;
V=0 -> there was no overflow.

11.9 – LOGICAL GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
AND r	A ← A ∧ r	0	↕	P	↕	0	1	10 100 r	--	04	05	01	01
AND p	A ← A ∧ p	0	↕	P	↕	0	1	11 011 101 10 100 p	DD --	08	10	02	02
AND q	A ← A ∧ q	0	↕	P	↕	0	1	11 111 101 10 100 p	FD --	08	10	02	02
AND (HL)	A ← A ∧ (HL)	0	↕	P	↕	0	1	10 100 110	A6	07	08	02	04
AND (IX+d)	A ← A ∧ (IX+d)	0	↕	P	↕	0	1	11 011 101 10 100 110 ← d →	DD A6 --	19	21	05	07
AND (IY+d)	A ← A ∧ (IY+d)	0	↕	P	↕	0	1	11 111 101 10 100 110 ← d →	FD A6 --	19	21	05	07
AND n	A ← A ∧ n	0	↕	P	↕	0	1	11 100 110 ← n →	E6 --	07	08	02	02
OR r	A ← A ∨ r	0	↕	P	↕	0	1	10 110 r	--	04	05	01	01
OR p	A ← A ∨ p	0	↕	P	↕	0	1	11 011 101 10 110 p	DD --	08	10	02	02
OR q	A ← A ∨ q	0	↕	P	↕	0	1	11 111 101 10 110 p	FD --	08	10	02	02
OR (HL)	A ← A ∨ (HL)	0	↕	P	↕	0	1	10 110 110	B6	07	08	02	04
OR (IX+d)	A ← A ∨ (IX+d)	0	↕	P	↕	0	1	11 011 101 10 110 110 ← d →	DD B6 --	19	21	05	07
OR (IY+d)	A ← A ∨ (IY+d)	0	↕	P	↕	0	1	11 111 101 10 110 110 ← d →	FD B6 --	19	21	05	07
OR n	A ← A ∨ n	0	↕	P	↕	0	1	11 110 110 ← n →	F6 --	07	08	02	02
XOR r	A ← A ⊕ r	0	↕	P	↕	0	1	10 110 r	--	04	05	01	01
XOR p	A ← A ⊕ p	0	↕	P	↕	0	1	11 011 101 10 110 p	DD --	08	10	02	02
XOR q	A ← A ⊕ q	0	↕	P	↕	0	1	11 111 101 10 110 p	FD --	08	10	02	02
XOR (HL)	A ⊕ (HL)	0	↕	P	↕	0	1	10 110 110	B6	07	08	02	04

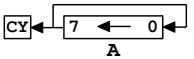
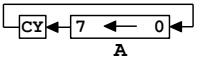
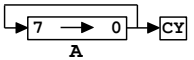
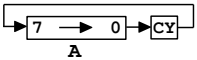
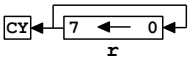
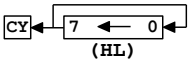
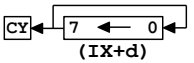
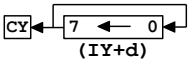
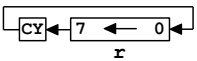
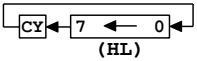
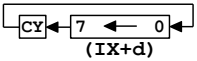
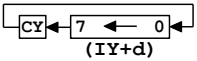
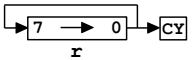
Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
XOR (IX+d)	A ← A ⊕ (IX+d)	0	↕	P	↕	0	1	11 011 101 10 110 110 ← d →	DD B6 --	19	21	05	07
XOR (IY+d)	A ← A ⊕ (IY+d)	0	↕	P	↕	0	1	11 111 101 10 110 110 ← d →	FD B6 --	19	21	05	07
XOR n	A ← A ⊕ n	0	↕	P	↕	0	1	11 110 110 ← n →	F6 --	07	08	02	02

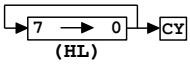
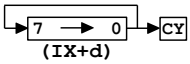
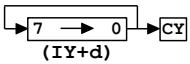
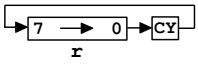
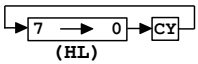
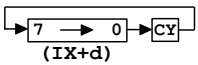
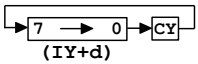
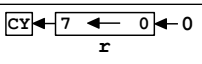
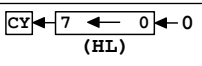
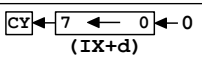
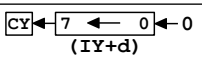
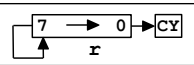
	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	•	A
p	•	•	•	•	IXH	IXL	•	•
q	•	•	•	•	IYH	IYL	•	•

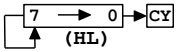
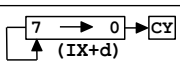
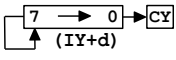
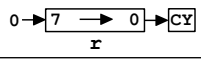
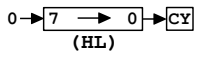
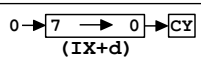
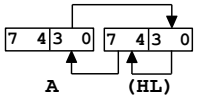
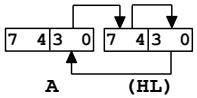
TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
↕ = Flag affected according operation results
0 = Flag off
1 = Flag on
P = the P/V flag contains the parity status. P=1 means the parity of the result is even; P=0 means it is odd.

11.10 – ROTATE AND SHIFT GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
RLCA		↕ • • • 0 0	00 000 111	07	04	05	01	01
RLA		↕ • • • 0 0	00 010 111	0F	04	05	01	01
RRCA		↕ • • • 0 0	00 001 111	17	04	05	01	01
RRA		↕ • • • 0 0	00 011 111	1F	04	05	01	01
RLC r		↕ ↕ P ↕ 0 0	11 001 011 00 000 r	CB --	08	10	02	02
RLC (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 000 110	CB 06	15	17	05	08
RLC (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 000 110	DD CB -- 06	23	25	07	10
RLC (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 000 110	FD CB -- 06	23	25	07	10
RL r		↕ ↕ P ↕ 0 0	11 001 011 00 010 r	CB --	08	10	02	02
RL (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 010 110	CB 16	15	17	05	08
RL (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 010 110	DD CB -- 16	23	25	07	10
RL (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 010 110	FD CB -- 16	23	25	07	10
RRC r		↕ ↕ P ↕ 0 0	11 001 011 00 001 r	CB --	08	10	02	02

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
RRC (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 001 110	CB 0E	15	17	05	08
RRC (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 001 110	DD CB -- 0E	23	25	07	10
RRC (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 001 110	FD CB -- 0E	23	25	07	10
RR r		↕ ↕ P ↕ 0 0	11 001 011 00 011 r	CB --	08	10	02	02
RR (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 011 110	CB 1E	15	17	05	08
RR (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 011 110	DD CB -- 1E	23	25	07	10
RR (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 011 110	FD CB -- 1E	23	25	07	10
SLA r		↕ ↕ P ↕ 0 0	11 001 011 00 100 r	CB --	08	10	02	02
SLA (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 100 110	CB 26	15	17	05	08
SLA (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 100 110	DD CB -- 26	23	25	07	10
SLA (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 100 110	FD CB -- 26	23	25	07	10
SRA r		↕ ↕ P ↕ 0 0	11 001 011 00 101 r	CB --	08	10	02	02

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
SRA (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 101 110	CB 2E	15	17	05	08
SRA (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 101 110	DD CB -- 2E	23	25	07	10
SRA (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 101 110	FD CB -- 2E	23	25	07	10
SRL r		↕ ↕ P ↕ 0 0	11 001 011 00 111 r	CB --	08	10	02	02
SRL (HL)		↕ ↕ P ↕ 0 0	11 001 011 00 111 110	CB 3E	15	17	05	08
SRL (IX+d)		↕ ↕ P ↕ 0 0	11 011 101 11 001 011 ← d → 00 111 110	DD CB -- 3E	23	25	07	10
SRL (IY+d)		↕ ↕ P ↕ 0 0	11 111 101 11 001 011 ← d → 00 111 110	FD CB -- 3E	23	25	07	10
RLD		• ↕ P ↕ 0 0	11 101 101 01 101 111	ED 6F	18	20	05	08
RRD		• ↕ P ↕ 0 0	11 101 101 01 100 111	ED 67	18	20	05	08

	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	•	A

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
0 = Flag off
P = the P/V flag contains the parity status. P=1 means the parity of the result is even; P=0 means it is odd.

11.11 – BIT SET, RESET AND TEST GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
BIT b,r	$Z \leftarrow \overline{r}_b$	0	↕	?	?	0	1	11 001 011 01 b r	CB --	08	10	02	02
BIT b,(HL)	$Z \leftarrow \overline{(HL)}_b$	0	↕	?	?	0	1	11 001 011 01 b 110	CB --	12	04	03	05
BIT b,(IX+d)	$Z \leftarrow \overline{(IX+d)}_b$	0	↕	?	?	0	1	11 011 101 11 001 011 ← d → 01 b 110	DD CB -- --	20	22	05	07
BIT b,(IY+d)	$Z \leftarrow \overline{(IY+d)}_b$	0	↕	?	?	0	1	11 111 101 11 001 011 ← d → 01 b 110	FD CB -- --	20	22	05	07
SET b,r	$\overline{r}_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 11 b r	CB --	08	10	02	02
SET b,(HL)	$\overline{(HL)}_b \leftarrow 1$	•	•	•	•	•	•	11 001 011 11 b 110	CB --	15	17	05	08
SET b,(IX+d)	$\overline{(IX+d)}_b \leftarrow 1$	•	•	•	•	•	•	11 011 101 11 001 011 ← d → 11 b 110	DD CB -- --	23	25	07	10
SET b,(IY+d)	$\overline{(IY+d)}_b \leftarrow 1$	•	•	•	•	•	•	11 111 101 11 001 011 ← d → 11 b 110	FD CB -- --	23	25	07	10
RES b,r	$\overline{r}_b \leftarrow 0$	•	•	•	•	•	•	11 001 011 10 b r	CB --	08	10	02	02
RES b,(HL)	$\overline{(HL)}_b \leftarrow 0$	•	•	•	•	•	•	11 001 011 10 b 110	CB --	15	17	05	08
RES b,(IX+d)	$\overline{(IX+d)}_b \leftarrow 0$	•	•	•	•	•	•	11 011 101 11 001 011 ← d → 10 b 110	DD CB -- --	23	25	07	10

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
RES b,(IY+d)	$\overline{(IY+d)}_b \leftarrow 0$	•	•	•	•	•	•	11 111 101 11 001 011 ← d → 10 b 110	FD CB -- --	23	25	07	10

	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	•	A
b	b0	b1	b2	b3	b4	b5	b6	b7

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
0 = Flag off
1 = Flag on
? = Flag unknown

11.12 – JUMP GROUP

Mnemonic	Operation	C	Z	P/V	S	N	H	Binary	Hex	TZ	Z1	TR	RW
JP nn	PC ← nn	•	•	•	•	•	•	10 000 011 ← n → ← n →	C3 -- --	10	11	03	05
JP cc,nn	If cc=true, PC ← nn	•	•	•	•	•	•	10 cc 011 ← n → ← n →	-- -- --	10 10	11 11	03 03	03 05
JR e	PC ← PC+e	•	•	•	•	•	•	00 011 000 ← e-2 →	18 --	12	13	03	03
JR C,e	If CY=1, PC ← PC+e	•	•	•	•	•	•	00 111 000 ← e-2 →	38 --	07 12	08 13	02 03	02 03
JR NC,e	If CY=0, PC ← PC+e	•	•	•	•	•	•	00 110 000 ← e-2 →	30 --	07 12	08 13	02 03	02 03
JR Z,e	If Z=1, PC ← PC+e	•	•	•	•	•	•	00 101 000 ← e-2 →	28 --	07 12	08 13	02 03	02 03
JR NZ,e	If Z=0, PC ← PC+e	•	•	•	•	•	•	00 100 000 ← e-2 →	20 --	07 12	08 13	02 03	02 03
JP (HL)	PC ← HL	•	•	•	•	•	•	11 101 001	E9	04	05	01	03
JP (IX)	PC ← IX	•	•	•	•	•	•	11 011 101 11 101 001	DD E9	08	10	02	04
JP (IY)	PC ← IY	•	•	•	•	•	•	11 111 101 11 101 001	FD E9	08	10	02	04
DJNZ e	B ← B-1 If B0, PC ← PC+e	•	•	•	•	•	•	00 010 000 ← e-2 →	10 --	08 13	09 14	02 03	02 03

	000	001	010	011	100	101	110	111
cc	NZ	Z	NC	C	PO	PE	P	M

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected

NOTE: When there are two descriptions of cycles, they refer to the two conditions that the instruction can assume. Thus, for LDIR, the time in T cycles for the Z80 is 21; when BC reaches 0, 16 T cycles are spent.

11.13 – CALL AND RETURN GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
CALL nn	(SP-1)←PC_H (SP-2)←PC_L PC ← nn	• • • • • •	11 001 101 ← n → ← n →	CD -- --	17	18	05	08 07
CALL cc,nn	If cc=true, (SP-1)←PC_H (SP-2)←PC_L PC ← nn	• • • • • •	11 cc 100 ← n → ← n →	CD -- --	10 17	11 18	03 05	07 03 08
RET	PC_H←(SP+1) PC_L←(SP)	• • • • • •	11 101 001	C9	10	11	03	05
RET cc	If cc=true, PC_H←(SP+1) PC_L←(SP)	• • • • • •	11 cc 000	--	05 11	06 12	01 03	01 05
RETI	Return from Interrupt	• • • • • •	11 101 101 01 001 101	ED 4D	14	16	05	07
RETN	Return from non maskable Interrupt	• • • • • •	11 101 101 01 000 101	ED 45	14	16	05	07
RST p	(SP-1)←PC_H (SP-2)←PC_L PC_H ← 0 PC_L ← t*8	• • • • • •	11 t 111	--	11	12	04	06 07

	000	001	010	011	100	101	110	111
cc	NZ	Z	NC	C	PO	PE	P	M
p	00H	08H	10H	18H	20H	28H	30H	38H

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected

NOTE: When there are two descriptions of cycles, they refer to the two conditions that the instruction can assume. Thus, for LDIR, the time in T cycles for the Z80 is 21; when BC reaches 0, 16 T cycles are spent.
NOTE1: Tests have shown that a CALL followed by a series of NOPs takes 8 cycles, while if followed by a combined RET or POP AF it takes 12 cycles (7 for CALL + 5 for RET/POP AF). This also applies to the RST (only applicable to the RW highlighted value for the R800).

11.14 – INPUT AND OUTPUT GROUP

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
IN A,(n)	A ← (n)	• • • • • •	11 011 011 ← n →	DB --	11	12	03	10 09
IN r,(C)	r ← (C)	• ↕ P ↕ 0 ↕	11 101 101 01 r 000	ED --	12	14	03	10 09
INI	(HL) ← (C) B ← B-1 HL ← HL+1	• ↕ ? ? 1 ?	11 101 101 10 100 010	ED A2	16	18	04	12 11
INIR	(HL) ← (C) B ← B-1 HL ← HL+1 {Até B=0}	• 1 ? ? 1 ?	11 101 101 10 110 010	ED B2	21 16	23 18	04 03	? 11 12
IND	(HL) ← (C) B ← B-1 HL ← HL-1	• ↕ ? ? 1 ?	11 101 101 10 101 010	ED AA	16	18	04	12 11
INDR	(HL) ← (C) B ← B-1 HL ← HL-1 {Até B=0}	• 1 ? ? 1 ?	11 101 101 10 111 010	ED BA	21 16	23 18	04 03	? 11 12
OUT (n),A	(n) ← A	• • • • • •	11 010 011 ← n →	D3 --	11	03	03	10 9
OUT (C),r	(C) ← r	• • • • • •	11 101 101 01 r 001	ED --	11	12	03	10 9
OUTI	(C) ← (HL) B ← B-1 HL ← HL+1	• ↕ ? ? 1 ?	11 101 101 10 100 011	ED A3	16	18	04	12 11
OTIR	(C) ← (HL) B ← B-1 HL ← HL+1 {Até B=0}	• 1 ? ? 1 ?	11 101 101 10 110 011	ED B3	21 16	23 18	04 03	? 11 12
OUTD	(C) ← (HL) B ← B-1 HL ← HL-1	• ↕ ? ? 1 ?	11 101 101 10 101 011	ED AB	16	18	04	12 11

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
OTDR	(C) ← (HL) B ← B-1 HL ← HL-1 {Até B=0}	• 1 ? ? 1 ?	11 101 101 10 111 011	ED BB	21 16	23 18	04 03	? 11 12

	000	001	010	011	100	101	110	111
r	B	C	D	E	H	L	F	A

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
0 = Flag off
1 = Flag on
? = Flag unknown
P = the P/V flag contains the parity status. P=1 means the parity of the result is even; P=0 means it is odd.

NOTE: In the INI, IND, OUTI e OUTD instructions, the flag Z is set when B-1=0 and is reset otherwise.
NOTE1: In the 'IN A,(n)' e 'OUT (n),A' instructions, 'n' is sent to A0~A7 and A is sent to A8~A15. In the other instructions, 'C' content is sent to A0~A7 and 'B' content is sent to A8~A15.
NOTE2: The I/O instructions are aligned to the bus clock, so an extra wait is inserted depending on the alignment. This means that between two OUTs there may be a reduction of one cycle (applicable only to the RW value highlighted for the R800).

11.15 – GENERAL PURPOSE AND CONTROL GROUPS

Mnemonic	Operation	C Z P/V S N H	Binary	Hex	TZ	Z1	TR	RW
CCF	CY ← $\overline{CY}$	1 • • • 0 ?	00 111 111	3F	04	05	01	01
CPL	A ← $\overline{A}$	• • • • 1 1	00 101 111	2F	04	05	01	01
DAA	Converts A to BCD	↕ ↕ P ↕ • ↕	00 100 111	27	04	05	01	01
DI	IFF ← 0	• • • • • •	11 110 011	F3	04	05	02	02
EI	IFF ← 1	• • • • • •	11 111 011	FB	04	05	01	01
HALT	Halts CPU	• • • • • •	01 110 110	76	04	05	02	02
IM 0	Interrupt mode 0	• • • • • •	11 101 101 01 000 110	ED 46	08	10	03	03
IM 1	Interrupt mode 1	• • • • • •	11 101 101 01 010 110	ED 56	08	10	03	03
IM 2	Interrupt mode 2	• • • • • •	11 101 101 01 011 110	ED 5E	08	10	03	03
NEG	A ← 0 - A	↕ ↕ V ↕ 1 ↕	00 101 101 01 000 100	ED 44	08	10	02	02
NOP	No operation	• • • • • •	00 000 000	00	04	05	01	01
SCF	CY ← 1	1 • • • 0 0	00 110 111	37	04	05	01	01

TZ – Z80 T Cycles
Z1 – Z80 + M1
TR – R800 T Cycles
RW – R800 + Wait

Flags notation:
• = Flag not affected
↕ = Flag affected according operation results
0 = Flag off
1 = Flag on
? = Flag unknown
V = the P/V flag contains the overflow status: V=1 -> overflow; V=0 -> there was no overflow.
P = the P/V flag contains the parity status. P=1 means the parity of the result is even; P=0 means it is odd.

NOTE: IFF indicates the flip-flop of interrupt activation circuit.
CY indicates the flip-flop of the overflow circuit.

12 – STANDARD CHIPS REGISTERS MAPS

12.1 – MAP OF THE REGISTERS OF THE V9918/38/58

Regist.		b7	b6	b5	b4	b3	b2	b1	b0	Short description
R#0	W	•	•	•	•	•	•	m3	EV	Mode Register #1 (9918)
		b7~b2 b1 b0		Not used (always“000 000”) m3: Screen mode (together with R#1) EV: 0=disable external input; 1=enable						
		•	DG	IE2	IE1	m5~m3			•	Mode register #1 (9938/58)
		b7 b6 b5 b4 b3~b1 b0		Not used (always 0) DG: 0=normal; 1=color bus in input mode IE2: Lightpen input (eliminated in the 9958) IE1: 0=enable line interrupt #1; 1=disable M5~M3: Screen mode (together with R#1) Not used (always 0)						
R#1	W	16K	BL	IE0	m2~m1		BC	SI	MA	Mode register #2
		b7 b6 b5 b4~b3		*(9918)* ⟶ 0=4027(4K x 1-bit); 1=4108(8K x 1-bit)/4116(16K x 1-bit) *(9938/58)* ⟶ Not used (always 0) BL: 0=screen off; 1=screen on IE0: 9918: 0=enable interrupt; 1=disable interrupt 9938/58: 0=enable line interrup #0; 1=disable M2~M1: Screen mode (together with R#0)						

```
M5 M4 M3 M2 M1  b4 b3→(de R#25/9958)
 0  0  0  1  0   0  0   Screen 0 Wth 40
 0  1  0  1  0   0  0   Screen 0 Wth 80
 0  0  0  0  0   0  0   Screen 1
 0  0  1  0  0   0  0   Screen 2
 0  0  0  0  1   0  0   Screen 3
 0  1  0  0  0   0  0   Screen 4
 0  1  1  0  0   0  0   Screen 5
 1  0  0  0  0   0  0   Screen 6
 1  0  1  0  0   0  0   Screen 7
 1  1  1  0  0   0  0   Screen 8
 1  1  1  0  0   1  1   Screen 10/11
 1  1  1  0  0   0  1   Screen 12
```

										FAFCH = • • • x x • • • xx=10 ⟶ Screen 10 (MODE) xx=11 ⟶ Screen 11 b2 Bit Cadari b1 SI: 0=8x8 sprites; 1=16x16 sprites b0 MA: 0=normal sprites; 1=expanded sprites
R#2	W	0	a16	a15	a14	a13	a12	a11	a10	Pattern name table adress
R#3	W	a13	a12	a11	a10	a9	a8	a7	a6	Pattern color table adress
R#4	W	0	0	a16	a15	a14	a13	a12	a11	Pattern generator table adr.
R#5	W	a14	a13	a12	a11	a10	a9	a8	a7	Sprites attributes table
R#6	W	0	0	a16	a15	a14	a13	a12	a11	Sprites pattern gen. table adr.
Added registers in the V9938										
R#7	W	f3~f0				b3~b0				f3~f0 – Forecolor (0~15) b3~b0 – Backcolor (0~15)
R#8	W	MS	LP	TP	CB	VR	•	SPD	BW	Mode register #3
										b7 MS: 0=color bus out; 1=color bus in b6 LP: 0=disabel lightpen; 1=eneble (eliminated 9958) b5 TP: 0=0 color transparent; 1=0 color can be defined b4 CB: 0=color bus out; 1=color bus in b3 VR: VRAM 0=16K/1bit, 16K/4bits; 1=64K/1bits, 64K/4bits b2 Not used (always 0) b1 SPD: 0=sprites on ; 1=sprites off b0 BW: 0=color output; 1=grey scale output
R#9	W	LN	•	S1~S0		IL	EO	NT	DC	Registrador de modo#4
										b7 LN: 0=192 lines; 1=212 lines b6 Not used (always 0) b5~b4 S1-S0: 00 – Internal synchronization 01 – Mixed synchronization 10 – external synchronization (digitization) 11 – No synchronization b3 IL: 0=normal mode; 1=interlace mode b2 EO: 0=one screen; 1=two simultaneous screens b1 NT: 0=NTSC (262 lines); 1=PAL (313 lines) b0 DC: 0=*DLCLK out; 1=*DLCLK in

R#10	W	0	0	0	0	0	a16	a15	a14	Pattern color table adress
R#11	W	0	0	0	0	0	0	a16	a15	Sprites attributes table
R#12	W	f3~f0				b3~b0				f3~f0 – Blink forecolor b3~b0 – Blink backcolor
R#13	W	e3~e0 unit: 1/6 second				o3~o0 unit: 1/6 second				Blink R#7/R#12 e3~e0 – Even page o3~o0 – Odd page
R#14	W	0	0	0	0	0	a16	a15	a14	Base VRAM adress
R#15	W	•	•	•	•	0 ~ 9				Status register pointer
R#16	W	•	•	•	•	0 ~ 15				Palette register pointer
R#17	W	AI	•	R#0 a R#46						Pointer control
R#18	W	Vert –8 a +7				Hor –8 a +7				Screen adjust
R#19	W	Line number (0 a 255)								Line interrupt register
R#20	W	0	0	0	0	0	0	0	0	Color Burst #1
R#21	W	0	0	1	1	1	0	1	1	Color Burst #2
R#22	W	0	0	0	0	0	1	0	1	Color Burst #3
R#23	W	Line number (0 a 255)								Vertical scroll / adjust
R#24	W	---.---								This register not exist
Registradores adicionados para o V9958										
R#25	W	•	CMD	VDS	YAE	YJK	WTE	MSK	SP	Mode register #5
		b7	Not used (always 0)							
		b6	0=VDP commands in 5~7 screen modes only 1=VDP commands in all screen modes							
		b5	VDS: 0=CPUCLK output; 1= VDS output							
		b4	YAE: 0=YJK only; 1=YJK+RGB							
		b3	YJK: 0=RGB mode; 1=YJK mode							
		b2	WTE: 0=wait function off; 1=wait function active							
		b1	MSK: 0=scroll mask off; 1=scroll mask active							
		b0	SP: 0=1-page horizontal scroll; 1=2-page hor scroll							

R#26	W	•	•	h8	h7	h6	h5	h4	h3	Horizontal scroll
R#27	W	•	•	•	•	•	h2	h1	h0	
Command Registers (V9938 e V9958)										
R#32	W	x7	x6	x5	x4	x3	x2	x1	x0	Source horizontal coordinate (0 a 511)
R#33	W	•	•	•	•	•	•	•	x8	
R#34	W	y7	y6	y5	y4	y3	y2	y1	y0	Source vertical coordinate (0 a 1023)
R#35	W	•	•	•	•	•	•	y9	y8	
R#36	W	x7	x6	x5	x4	x3	x2	x1	x0	Destination horizontal coordinate (0 a 511)
R#37	W	•	•	•	•	•	•	•	x8	
R#38	W	y7	y6	y5	y4	y3	y2	y1	y0	Destination vertical coordinate (0 a 1023)
R#39	W	•	•	•	•	•	•	y9	y8	
R#40	W	x7	x6	x5	x4	x3	x2	x1	x0	Number of points in horizontal direction(0 a 511)
R#41	W	•	•	•	•	•	•	•	x8	
R#42	W	y7	y6	y5	y4	y3	y2	y1	y0	Number of points in vertical direction (0 a 1023)
R#43	W	•	•	•	•	•	•	y9	y8	
R#44	W	Cor (0~3; 0~15; 0~255)								Color register
R#45	W	•	MC	MD	MS	DIY	DIX	EQ	MAJ	Argument register
		b7 Not used (always0) b6 MXC: 0=VRAM; 1=expanded VRAM (std memory) b5 MXD: 0=VRAM; 1=expanded VRAM (dest memory) b4 MXS: 0=VRAM; 1=expand. VRAM (source memory) b3 DIY: 0=down; 1=up b2 DIX: 0=to right; 1=to left b1 EQ: 0=specified color; 1=other color (end of SRCH) b0 MAJ: 0=horizontal major side; 1=vertical major side								
R#46	W	Command(0~15)				Logopr (0~15)				Command register
		b7~b4	Command code (OP-CODE) 0 0 0 0 STOP Stop command 0 0 0 1~0 0 1 1 Not implemented							

			0 1 0 0 POINT Read point (dot) color code 0 1 0 1 PSET Draw a point and advance coord 0 1 1 0 SRCH Search point color code 0 1 1 1 LINE Draw a line (logical) 1 0 0 0 LMMV Draw a box (logical) 1 0 0 1 LMMM Transf. VRAM → VRAM (logical) 1 0 1 1 LMMC Transf. CPU → VRAM (logical) 1 1 0 0 HMMV Draw a box (bytes) 1 1 0 1 HMMM Transf. VRAM → VRAM (bytes) 1 1 1 0 YMMM Transf. VRAM → VRAM in Y direct. 1 1 1 1 HMMC Transf. CPU → VRAM (bytes)
		b3	Cor transparente: 0=make logical operation 1=not make logical operation
		b2~b0	Logopr 000 IMP DC = SC 001 AND DC = SC and DC 010 OR DC = SC or DC 011 XOR DC = SC xor DC 100 NOT DC = not (SC) 101~111 Not implemented

Status register (TMS9918/V9938/V9958)

S#0	R	FLG	5S	C	5º sprite (0~31)	Status register
		b7	FLG: Vertical interrupt flag			
		b6	5S: 0=normal; 1=more than 4 (ou 8) sprites in same line			
		b5	C: 0=normal; 1=two sprites colliding			
		b4~b0	Number of 5th (ou 9th) sprite			

Status register (V9938 e V9958)

S#1	R	LP	KEY	ID number	HI	Lightpen / ID / Hor.Interrupt
		b7	LPF: 0=LP normal; 1=LP light detect (elim. V9958)			
		b6	LPK: 0=LP key not pressed; 1=it is pressed			
		b5~b1	MSX-video number ID			
		b0	FH: 0=Horizontal interrupt disabled; 1=enabled			

S#2	R	TR	VR	HR	BD	•	•	EO	CE	Command status
		b7	TR: 0=VDP not ready for dada; 1=VDP is ready							
		b6	VR: 0=frame is not being scanned; 1=it is							
		b5	HR: 0=Line is not being scanned; 1=it is							
		b4	BD: 0=SRCH not search; 1=SRCH sucess							

		b3~b2 b1 b0		Not used (always 11) EO: 0=1st screen showed; 1=2nd screen showed CE: 0=VDP free; 1=VDP executing command						
S#3	R	x7	x6	x5	x4	x3	x2	x1	x0	Coordinate X+12 (Sprites collision)
S#4	R	•	•	•	•	•	•	•	x8	
S#5	R	y7	y6	y5	y4	y3	y2	y1	y0	Coordinate Y+8 (Sprites collision)
S#6	R	•	•	•	•	•	•	y9	y8	
S#7	R	Cor (0~3; 0~15; 0~255)								Color of specified point
S#8	R	x7	x6	x5	x4	x3	x2	x1	x0	Horizontal coordinate of the point (SRCH command)
S#9	R	•	•	•	•	•	•	•	x8	

12.1.1 – Access ports for VDPs V9918/38/38

Porta			b7	b6	b5	b4	b3	b2	b1	b0	Short description
P#0	98H	R/W	Byte de dados								Write/read VRAM data
P#1	99H	R	FLG	5S	C	5º sprite (0~31)					Read status register
			b7 b6 b5 b4~b0			FLG: Vertical interrupt flag 5S: 0=normal; 1=more than 4 (ou 8) sprites same line C: 0=normal; 1=two sprites colliding Number of 5th (ou 9th) sprite					
P#1	99H	W	a7~a0 adress								Select VRAM adress f: 0=read; 1=write
			0	f	a13~a8 adress						
			Data byte								Write control reg (9918) Select register. (9938/58)
			1	0	Nº reg. (0~46)						
P#2	9AH	W	0	r	r	r	0	b	b	b	Write in palette registers
			0	0	0	0	0	g	g	g	
P#3	9BH	W	Data byte								Write in indirect register

12.1.2 – Standard color chart

The color chart illustrated below is the standard color chart for the MSX1. For MSX2 onwards, the table is loaded when the computer is reset.

Palette number	Color	Red Level	Blue Level	Green Level
0	Transparent	0	0	0
1	Black	0	0	0
2	Green	1	1	6
3	Light Green	3	3	7
4	Deep Blue	1	7	1
5	Blue	2	7	3
6	Deep Red	5	1	1
7	Light Blue	2	7	6
8	Red	7	1	1
9	Light Red	7	3	3
10	Yellow	6	1	6
11	Light Yellow	6	3	6
12	Deep Green	1	1	4
13	Purple	6	5	2
14	Grey	5	5	5
15	White	7	7	7

12.2 – MAP OF THE V9990 REGISTERS

Reg		b7	b6	b5	b4	b3	b2	b1	b0	Short description
R#0	W	a7	a6	a5	a4	a3	a2	a1	a0	Write adress in VRAM AI = 0 ⟶ autoincrement
R#1	W	a15	a14	a13	a12	a11	a10	a9	a8	
R#2	W	AI	•	•	•	•	a18	a17	a16	
R#3	W	a7	a6	a5	a4	a3	a2	a1	a0	Read adress in VRAM AI = 0 ⟶ autoincrement
R#4	W	a15	a14	a13	a12	a11	a10	a9	a8	
R#5	W	AI	•	•	•	•	a18	a17	a16	

R#6	R/W	DSPM	DCKM	XIMM	CLRM	Screen mode

b7~b6	XIMM	00-Y=256pixels	10-Y=1024pixels
		01-Y=512pixels	11-Y=2048pixels
b5~b4	DSPM	00-P1 mode	10-Bit Map
		01-P2 mode	11-Stand-by
b3~b2	DCKM	00-XTAL=1/4	10-XTAL=1/1
		01-XTAL=1/2	11 – N/A
b1~b0	CLRM	00-2 bits/pixel	10-8 bits/pixel
		01-4 bits/pixel	11-16 bits/pixel

R#7	R/W	•	CM	S1	S2	PL	EO	IL	HS	Screen mode

b7	Not used (always 0)		
b6	C25M	0-other modes	1-B6 mode
b5	SM1	0-262 lines	1-263 lines
b4	SM	0-1H=fsc/228	1-1H=fsc/227.5
b3	PAL	0-NTSC	1-PAL
b2	EO	0-Y=normal	1-Y=doubled
b1	IL	0-no interlace	1-interlaced
b0	HSCN	0-other modes	1-B5-B6 modes

R#8	R/W	DP	SP	YS	VE	VM	DM	V1~V0	System control

b7	DISP	0=Backcolor only, 1=Normal display
b6	SPD	0=Show sprite/cursor, 1=Not show
b5	YSE	0=YS signal out disabled, 1=Enabled
b4	VWTE	0=VRAM serial data bus output; 1=input

		b3		VWM VRAM write digitization: 0=No, 1=While horizontal blank						
		b2		DMAE 0=DREQ output in high level, 1=Enabled						
		b1~b0		VSL1/0 00=64K x 4-bit, 128K total 01=128K x 8-bit, 256K total 10=256K x 4-bit, 512K total						
R#9	R/W	•	•	•	•	•	IE	IH	IV	Interrupt control
		b7~b3		Not used (always "00 000")						
		b2		IECE 0-interrupt disabled, 1-enabled						
		b1		IEH 0-line interrupt disabled, 1-enabled						
		b0		IEV 0-frame interrupt disabled, 1-enabled						
R#10	R/W	I7	I6	I5	I4	I3	I2	I1	I0	Interrupt line num (I9~I0)
R#11	R/W	IE	•	•	•	•	•	I9	I8	IEHM – 0=specific line 1=all lines
R#12	R/W	•	•	•	•	ix3	ix2	ix1	ix0	Interrupt horizontal position (IX) × (64) × (master clock)
R#13	W	PLTM		IE	AI	PLTO5~2				Palette control
		b7~b6		PLTM - 00=palette; 01=256 colors; 10=YJK; 11=YUV						
		b5		YAE – 0=YJK/YUV only; 1=YJK/YUV + RGB						
		b4		PLTAIH – Autoinc palette read (0=Yes, 1=No)						
		b3~b0		PLTO5-2 – Palette offset (0 a 15)						
R#14	W	PLTA5~0						PL2~1		Palette control
		b7~b2		PLTA – Num of the color in the palette (0 a 63)						
		b1~b0		PLTP – 00-Red; 01-Green; 10-Blue; 11-N/C						
R#15	R/W	•	•	b5	b4	b3	b2	b1	b0	Backcolor
R#16	R/W	ADJV (–8 a +7)				ADJH (–8 a +7)				Screen adjust
R#17	R/W	SCAY (b7~b0)								Scroll control Y-coord in the plane "A"
R#18	R/W	ROLL		•	SCAY (b12~b8)					
		b7~b0		SCAY – Y-coordinate of start of display for the "A"						
		b4~b0		plane of P1 mode and the screens of other modes						
		b5		Not used (always "0")						
		b7~b6		ROLL – Direct. Y scroll: 00-full screen 10-512 lines 01-256 lines 11-N/C						

R#19	R/W	•	•	•	•	•	SX (b2~b0)			Scroll control "A" plane X-coord and B0~B7
R#20	R/W	SCAX (b10~b3)								
R#21	R/W	SCBY (b7~b0)								Scroll control "B" plane Y-coord
R#22	R/W	A	B	•	•	•	•	•	b8	
		b7~b0	SCBY – Start Y-coord for showing "B" plane in the P1 mode (b8~b0)							
		b0								
		b7	SDB – If=1, disable sprites and "A" plane							
		b6	SDB – If=1, disable sprites and "B" plane							
		b5~b1	Not used (always "00 000")							
R#23	R/W	•	•	•	•	•	BX(b2~b0)			SCBX - Start X-coord for showing "B" plane
R#24	R/W	•	•	SCBX (b8~b3)						
R#25	R/W	•	•	•	•	a17	a16	a15	•	Pattern sprites adress (P1)
		•	•	•	•	a18	a17	a16	a15	Pattern sprites adress (P2)
R#26	R/W	•	•	•	VR	PNS	PL	PD	PNE	LCD panel control
		b7~b5	Not used (always "000")							
		b4	VRI 0=equal CRT 1=one line vertical blank							
		b3	PNSL 0=400 vertical points 1=480 vert points							
		b2	PLVO 0=Greyscale (D3~0 pins)							
			1=Color (CB7~0 pins)							
		b1	PDUAL 0=one screen 1=two screens							
		b0	PNEN 0=CRT cycle 1=LCD cycle							
R#27	R/W	•	•	•	•	PRY		PRX		P1 mode priority
		b7~b4	Not used (always "0000")							
		b3~b2	PRX – 00 – X=256 10 – X=128							
			01 – X=64 11 – X=192							
		b1~b0	PRY – 00 – Y=256 10 – Y=128							
			01 – Y=64 11 – Y=192							
R#28	W	•	•	•	•	CSPO (b5~b2)				Cursor palette offset
		The R#29 to R#31 registers not exist								

R#32	W	SX, SA, KA (b7~b0)										VDP Commands – SOURCE X,Y-coordinates (SX, SY) Linear adress (SA) Kanji-ROM adress (KA)
R#33	W	•	•	•	•	•	SX(b10~b8)					
R#34	W	SY (b7~b0); SA, KA (b15~b8)										
R#35	W	•	•	•	•	SY (b7~b0)						
		•	•	•	•	•	SA(b18~b16)					
		•	•	•	•	•	•	K(b17~16)				
R#36	W	DX, DA (b7~b0)										VDP Commands – DEST X,Y-coordinates (DX, DY) Linear adress (DA)
R#37	W	•	•	•	•	•	DX(b10~b8)					
R#38	W	DY (b7~b0); DA (b15~b8)										
R#39	W	•	•	•	•	DY (b7~b0)						
		•	•	•	•	•	DA(b18~b16)					
R#40	W	NX, NA, MJ (b7~b0)										VDP Commands – DIVS Number of pixels to transfer XY (NX, NY) No of bytes to transfer (NA) Major side of the line (MJ) Minor side of the line (MI)
R#41	W	•	•	•	•	MJ (b11~b8)						
		•	•	•	•	•	NX(b10~b8)					
R#42	W	NY, MI (b7~b0); NA (b15~b8)										
R#43	W	•	•	•	•	NY,MI(b11~b8)						
		•	•	•	•	•	NA(b18~b16)					
R#44	W	•	•	•	•	DIY	DIX	NEQ	MAJ			Argument reg (write only)
		b7~b4	Not used (always "0000")									
		b3	DIY: 0 – Down; 1 – Up;									
		b2	DIX: 0 – To right; 1 – To left. Note: BMXL and BMLX are fixed em "+" and BMLL, X and Y are fixed in same direction.									
		b1	NEQ (SRCH end): 0=spec color 1=other color									
		b0	MAJ (LINE maj side): 0=horizontal 1=vertical									
R#45	W	•	•	•	T	Logopr						Logical operation
		b7~b5	Alwayse "000"									
		b4	Transparent color: 0=make logical operation 1=not make logical operation									

		b3~b0								Logopr 0 0 0 1 WC = not (SC .or. DC) 0 0 1 1 WC = not (SC) 0 1 1 0 WC = SC .xor. DC 0 1 1 1 WC = not (SC .and. DC) 1 0 0 0 WC = SC .and. DC 1 0 0 1 WC = not (SC .xor. DC) 1 1 0 0 WC = SC 1 1 1 0 WC = SC .or. DC
R#46	W	m7	m6	m5	m4	m3	m2	m1	m0	Write mask Bit=1 ⟶ read alowed Bit=0 ⟶ not alowed
R#47	W	m15	m14	m13	m12	m11	m10	m9	m8	
		In P1 mode, R#46 is used for "A" plane and R#47 for "B" pl.								
R#48	W	f7	f6	f5	f4	f3	f2	f1	f0	Forecolor
R#49	W	f15	f14	f13	f12	f11	f10	f9	f8	
R#50	W	b7	b6	b5	b4	b3	b2	b1	b0	Backcolor
R#51	W	b15	b14	b13	b12	b11	b10	b9	b8	
R#52	W	OP-CODE				AY		AX		Operation control
		b7~b4								Command code (OP-CODE) 0 0 0 0 STOP Stop command 0 0 0 1 LMMC Transf. CPU ⟶ VRAM (coord) 0 0 1 0 LMMV Draw rectangle in the VRAM 0 0 1 1 LMCM Transf. VRAM ⟶ CPU (coord) 0 1 0 0 LMMM Transf. VRAM ⟶ VRAM (coord) 0 1 0 1 CMMC Transf. character CPU ⟶ VRAM 0 1 1 0 CMMK Transf. Kanji ROM ⟶ VRAM 0 1 1 1 CMMM Transf. character. VRAM ⟶ VRAM 1 0 0 0 BMXL VRAM ⟶ VRAM (linear ⟶ coord) 1 0 0 1 BMLX VRAM ⟶ VRAM (coord ⟶ linear) 1 0 1 0 BMLL VRAM ⟶ VRAM (linear ⟶ linear) 1 0 1 1 LINE Draw a line 1 1 0 0 SRCH Search point (dot) color code 1 1 0 1 POINT Read point color code 1 1 1 0 PSET Draw a point and advance coord. 1 1 1 1 ADVN Advanced coord without drawing

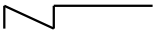
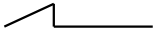
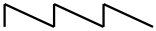
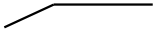
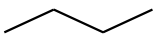
		b3~b2		AY:	00-no move				10-move down	
					01-no move				11-move up	
		b1~b0		AX:	00-(DX,DY) is used				10-move to right	
					01-no movr				11-move to left	
R#53	R	x7	x6	x5	x4	x3	x2	x1	x0	Horizontal coord of the point (SRCH sucessfully)
R#54	R	•	•	•	•	•	x10	x9	x8	

12.2.1 – Access ports to V9990

Porta			b7	b6	b5	b4	b3	b2	b1	b0	Short description
P#0	60H	R/W	Byte de dados								VRAM access
P#1	61H	R/W	$\overline{YS}$	•	•	Red (0~31)					Color palette access
			•	•	•	Green (0~31)					
			•	•	•	Blue (0~31)					
P#2	62H	R/W	Data byte								Hardware commands
P#3	63H	R/W	b7~b0 adress (R#0)								VRAM adress (AI=0 ⟶ autoincrement)
			b15~b8 adress (R#1)								
			AI	•	•	•	•	b18~b16(R#2)			
P#4	64H	W	WI	RI	Register number						Registers selection
P#5	65H	R	TR	VR	HR	BD	•	CS	EO	CE	Status port
			b7	TR: 0=VDP not ready for data; 1=VDP ready							
			b6	VR: 0=vertical non-display period; 1=dispaying							
			b5	HR: 0=horiz. Non-display period; 1=displaying							
			b4	BD: 0=SRCH not found; 1=sucessfully							
			b3	Not used (always 0)							
			b2	MCS: copy of the MCS bit of P#7							
			b1	EO: 0=1st screen is being displayed; 1=2nd screen							
			b0	CE: 0=VDP free; 1=VDP executing command							

P#6	66H	R/W	•	•	•	•	•	CE	HI	VI	Interrupt flag
			b7~b3 b2 b1 b0		Not used (always “00 000”) CE – Command end flag HI – Horizontal interrupt flag VI – Vertical interrupt flag						
P#7	67H	W	•	•	•	•	•	•	SR	MC	System control
			b7~b2 b1 b0		Not used (always “000 000”) SRS: if set in “1”, all VDP ports will put in the reset state (but not this). MCS: select master clock: 0=XTAL pin; 1=MCKIN pin						
P#8	68H	W	•	•	a5	a4	a3	a2	a1	a0	Kanji-ROM adr (low) – 1
P#9	69H	R/W	•	•	a11	a10	a9	a8	a7	a6	Kanji-ROM adress (high) and data – '1'
			Byte de dados								
P#A	6AH	W	•	•	a5	a4	a3	a2	a1	a0	Kanji-ROM adr (low) – 2
P#B	6BH	R/W	•	•	a11	a10	a9	a8	a7	a6	Kanji-ROM adress (high) and data – '2'
			Byte de dados								

12.3 – MAP OF PSG REGISTERS (AY-3-8910)

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short description
R#0	7	6	5	4	3	2	1	0	Channel "A" frequency
R#1	•	•	•	•	11	10	9	8	111860,87 / F_num (b11~b0)
R#2	7	6	5	4	3	2	1	0	Channel "B" frequency
R#3	•	•	•	•	11	10	9	8	111860,87 / F_num (b11~b0)
R#4	7	6	5	4	3	2	1	0	Channel "C" frequency
R#5	•	•	•	•	11	10	9	8	111860,87 / F_num (b11~b0)
R#6	•	•	•	4	3	2	1	0	White noise frequency 111860,87 / F_num (b4~b0)
R#7	ioB	ioA	rC	rB	rA	tC	tB	tA	Enable/disable sounds
	b0~b2 b5~b4 b6 b7		Enable/disable tones (0=enable) Enable/disable white noise (0=enable) Configures "A" I/O port (0=in, 1=out) Configures "B" I/O port (0=in, 1=out)						
R#8	•	•	•	m	v	v	v	v	Volume of channel A
R#9	•	•	•	m	v	v	v	v	Volume of channel B
R#10	•	•	•	m	v	v	v	v	Volume of channel C
	b7~b5 b4 b3~b0		Not used (always "000") 0=Not use envelope; 1=Use envelope 0000=Minimum volume; 1111=Maximum volume						
R#11	7	6	5	4	3	2	1	0	Envelope frequency
R#12	15	14	13	12	11	10	9	8	6983,3 / F_num (b15~b0)
R#13	•	•	•	•	e	e	e	e	Envelope shape
	b7~b4 b3~b0		Not used (always "0000") Defines envelope shape: 00xx =  1011 =  01xx =  1100 =  1000 =  1101 =  1001 =  1110 =  1010 =  1111 = 						
R#14	a7	a6	a5	a4	a3	a2	a1	a0	Read/write "A" I/O port
R#15	b7	b6	b5	b4	b3	b2	b1	b0	Read/write "B" I/O port

12.3.1 – Access ports to PSG

Porta		b7	b6	b5	b4	b3	b2	b1	b0	Short description
A0H	W	•	•	•	•	Reg.num. (0~15)				Select register
A1H	W	Data byte								Write data in the PSG
A2H	R	Data byte								Read data from PSG

12.4 – MAP OF FM-OPLL REGISTERS (YM2413)

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H	AM	VIB	EGT	KSR	Multiple				→ (m) – Modulated wave
$01H	AM	VIB	EGT	KSR	Multiple				→ (c) – Carrier wave
	b7 b6 b5 b4 b0~b3		AM: 0=tremolo off; 1=tremolo on VIB: 0=vibrato off; 1=vibrato on EGT: 0=decaying sound; 1=sustained sound KSR: 0=same level; 1=frequency attenuation (KSL) Multiplication factor (0=1/2, 1=1, 2=2, ..., 15=15)						
$02H	KSL(m)		Total level modul. (m)						Instrument definition
	b6~b7 b6~b0		KSL (m): 00=0dB/oct, 01=1,5dB, 10=3dB, 11=6dB Total level: b0=0,75dB, b1=1,5dB, ..., b5=24dB						
$03H	KSL(c)		•	DC	DM	Feedback			Instrument definition
	b6~b7 b5 b4 b3 b2~b0		KSL (c): 00=0dB/oct, 01=1,5dB, 10=3dB, 11=6dB Not used (always 0) DC: 0=integer carrier wave, 1=half-wave DM: 0=integer modulated wavw, 1=half-wave Feedback: (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)						
$04H	Attack (m)				Decay (m)				Attack (0dB a 48dB → min. 0,14 mS; max 1730 mS)
$05H	Attack (c)				Decay (c)				Decay (0dB a 48dB → min. 1,27 mS; max 20 926 mS)
$06H	Sustain (m)				Release (m)				Sustain (b7=24dB, b6=12dB, b5=6dB, b4=3dB)
$07H	Sustain (c)				Release (c)				Release (0dB a 48dB → min. 1,27 mS; max 28 926 mS)
$0EH	•	•	R	BD	SD	TOM	TCY	HH	Rhythm control
	b7~b6 b5 b0~b4		Not used (always "00") 0=melody mode; 1=rhythm mode 0=rhythm instrument off; 1=on BD–Bass Drum SD–Snare Drum TOM–Tom tom TCY–Top cymbal HH–Hi-hat						

$0FH	b7	•	b5	•	b3	b2	b1	b0	OPLL test register
	b7 b5 b3 b2 b1 b0	1=Reset the LFOs (max amplitude, zero phase deviation) 1=Select waveform 1=Updates the LFO every sample (the frequency of the tremolo is multiplied by 64 and vibrato by 1024) 1=Hold but resets the waveform to "0". 1=Hold but resets the LFO phase to "0". 1=Put the envelope generators (modulator and carrier) in the maximum volume.							
$10H ⋮ $18H	Frequency LSB (8 bits)								Registers used to select the frequencies of the tone generator
$20H ⋮ $28H	•	•	Sustain	Key	Octave			Freq.	Frequency MSB 1 bit Octave Key / Sustain on/off
	b7~b6 b5 b4 b3~b1 b0	Not used (always"00") 0=No sustain; 1=Release Rate will decay gradually 0=key off; 1=key on Octave setting. The fourth is 011. MSB Frequency 1 bit. The A 440 Hz central note is obtained with b0=1 and $10H~18H=00 100 000							
$30H ⋮ $38H	Instruments				Volume				Registers used to select the instruments and volume
	b7~b4	Instruments: 0000 – To be defined 0001 – Violin 0010 – Guitar 0011 – Piano 0100 – Flute 0101 – Clarinet 0110 – Oboé 0111 – Trumpet 1000 – Organ 1001 – Horn 1010 – Sinthesizer 1011 – Harpsichord 1100 – Vibraphone 1101 – Sinthesizer Bass 1110 – Acoustic Bass 1111 – Electric Guitar							

	b3~b0	Volume (0000=minimum; 1111=maximum)
Register map for Rhythm mode ($0EH, b5=1)		
$36H	• • • • BD volume	Volume registers for the rhythm sounds
$37H	HH volume	SD volume
$38H	TOM volume	TCY volume

12.4.1 – Access ports to OPLL

Porta		b7	b6	b5	b4	b3	b2	b1	b0	Short Description
7CH	W	Nº do registrador (00H a 38H)								Select register
7DH	W	Byte de dados								Write data in the OPLL

12.5 – MSX-AUDIO REGISTERS MAP (Y8950)

Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$01H	b7	b6	•	•	b3	b2	b1	b0	Test register
	b7	1=Reset the LFOs (max amplitude, zero phase deviation)							
	b6	1=Affect bit-0 (PCMBSY) of the status register.							
	b3	1=Updates the LFO every sample (the frequency of the tremolo is multiplied by 64 and vibrato by 1024)							
	b2	1=Hold but resets the waveform to "0".							
	b1	1=Hold but resets the LFO phase to "0".							
	b0	1=Put the envelope generators (modulator and carrier) in the maximum volume.							
$02H $03H	1st Timer (80 µS) 2nd Timer (320 µS)								Time registers
$04H	IRQ	T1M	T2M	EOS	BR	•	ST2	ST1	Flags register
	b7	IRQ – If write 1, reset all flags.							
	b6	T1M – If write 1, b0 will reset.							
	b5	T2M – If write 1, b1 will reset.							
	b4	EOS – B3 mask, indicating the end of current operation							
	b3	BR – ADPCM / Audio memory mask (1=enable)							
	b2	Not used (always0)							
	b1	ST2 – $03 Start/stop control (1=start counter)							
	b0	ST1 – $02 Start/stop control (1=start counter)							
$05H $06H	External keyboard (input) External keyboard (output)								Registers for access to external musical keyboard
$07H	STA	REC	MEM	REP	OFF	•	•	RST	Control register (1)
	b7	STA – Must be 1 to start data read/write							
	b6	REC – Must be 1 to write data in the memory							
	b5	MEM – Must be 1 to access audio memory							
	b4	REP – When 1, enable ADPCM data repeat							
	b3	OFF – When 1, cut off audio output							
	b1~b2	Not used (always"00")							
	b0	RST – When 1, puts ADPCM in the initial state							

$08H	CSM	SEL	•	•	SAM	DAD	64K	ROM	Control register (2)
	b7	CSM – 1=composite sinusoisal modulation mode							
	b6	SEL – External keyboard octaves separation point							
	b5~b4	Not used (always "00")							
	b3	SAM – 0=start DA conversion; 1=start AD conversion							
	b2	DAD – 0=conv. AD / mus. output; 1=$15~$16 → output							
	b1	64K – Memory size: 0=256K; 1=64K							
	b0	ROM – Memory type: 0=RAM; 1=ROM							
$09H $0AH $0BH $0CH	Start adress (b7~b0) Start adress (b15~b8) End adress (b7~b0) End adress (b15~b8)								Start and end adresses for CPU and ADPCM acesses
$0DH	f7	f6	f5	f4	f3	f2	f1	f0	ADPCM frequency 3580 / F_num (1,8~16 KHz)
$0EH	•	•	•	•	•	f10	f9	f8	
$0FH	ADPCM data								Data register
$10H	i7	i6	i5	i4	i3	i2	i1	i0	ADPCM interpolation factor (i15~i0) = 1310,72 *sampling rate
$11H	i15	i14	i13	i12	i11	i10	i9	i8	
$12H	ADPCM volume								ADPCM volume (0~255)
$15H	f9	f8	f7	f6	f5	f4	f3	f2	DA conversion data Out: Vcc/2 + Vcc/4*(−1+f9+ f8*2^{-1}+...+f1*2^{-8}+f0*2^{-9}+2-10)*2^{-E} E=S2*4+S1*2+S0*1 (S0+S1+S2>0)
&16H	f1	f0	•	•	•	•	•	•	
$17H	•	•	•	•	•	S2	S1	S0	
$18H	•	•	•	•	I/O control				I/O ports control ($18H → 0=input; 1=output)
$19H	•	•	•	•	I/O data				
$1AH	ADPCM data								Data register
$20H ⋮ $35H	AM	VIB	EGT	KSR	Multiple				Instruments definition
	b7	AM (1=tremolo on – Frequency: 3,7Hz)							
	b6	VIB (1=vibrato on – Frequency: 6,4Hz)							
	b5	EG-TYP (0=decaying sound; 1=sustained sound)							
	b4	If 0, KSR→0~3; If 1, KSR→0~15							
	b3~b0	Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)							

<table>
<tr><td>$40H
⋮
$55H</td><td colspan="2">KSL</td><td colspan="3">Total level</td><td>KSL (00= 0dB/octave,
01=1,5dB, 10=3dB, 11=6dB)
Total level (b0=0,75dB,
b1=1,5dB ..... b5=24dB)</td></tr>
<tr><td>$60H
⋮
$75H</td><td colspan="3">Attack Rate
(AR)</td><td colspan="2">Decay Rate
(DR)</td><td>Attack (0dB a 96dB →
min. 0,2 mS; max 2826 mS)
Decay (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$80H
⋮
$95H</td><td colspan="3">Sustain Level
(SL)</td><td colspan="2">Release Rate
(RL)</td><td>Sustain (b7=24dB, b6=12dB,
b5=6dB, b4=3dB)
Release (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$A0H
⋮
$A8H</td><td colspan="5">Frequency (LSB 8 bits)</td><td>FM frequency (b7~b0)</td></tr>
<tr><td rowspan="2">$B0H
⋮
$B8H</td><td>•</td><td>•</td><td>KEY</td><td>Octave</td><td>Freq.
MSB
2 bits</td><td>FM Frequency (b9~b8)
Octave (FM)
Key on/off (FM)</td></tr>
<tr><td colspan="2">b7~b6
b5
b4~b2
b1~b0</td><td colspan="4">Not used (always“00”)
0=key off; 1=key on (voice on)
Octave definition. The fourth is 011.
MSB Frequency 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 001</td></tr>
<tr><td></td><td colspan="6">Operators (para$20H~$35H e$A0H~$A8H)

Oper: 01 02 03 04 05 06 07 08 09
Voz: 1 2 3 1 2 3 4 5 6
Reg: $20$21$22$23$24$25$28$29$2A
Freq:$A0$A1$A2$A0$A1$A2$A3$A4$A5

Oper: 10 11 12 13 14 15 16 17 18
Voz: 4 5 6 7 8 9 7 8 9
Reg: $2B$2C$2D$30$31$32$33$34$35
Freq:$A3$A4$A5$A6$A7$A8$A6$A7$A8

The operators are associated as below:
$20/$40/$60/$80/$A0/$B0/$C0
ou
$23/$43/$63/$83/$A0/$B0/$C0</td></tr>
<tr><td>$BDH</td><td>AM</td><td>VIB</td><td>BAT</td><td>BD</td><td>SD</td><td>TOM</td><td>TCY</td><td>HH</td><td>FM rhythm control</td></tr>
</table>

	b7	Tremolo level (0=1dB. 1=4,8dB)
	b6	Vibrato level (0=7%; 1=14%)
	b5	0=Melody mode; 1=Rhythm mode
	b4	1=Bass Drum
	b3	1=Snare Drum
	b2	1=Tom-tom
	b1	1=Top Cymbal
	b0	1=High-Hat

$C0H ⋮ $C8H	•	•	•	•	Feedback	CON	FM Feedback factor and connection type
	b7~b4	Not used (always“0000”)					
	b3~b1	Feedback (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)					
	b0	Operators connection type (0=série; 1=paralelo)					

STAT	INT	T1	T2	EOS	BUF	•	•	PCM	Status register
	b7	Will be 1 when one or more bits b3 to b6 contains 1							
	b6	Will be 1 after timer 1 ends counting ($02)							
	b5	Will be 1 after timer 2 ends counting ($03)							
	b4	Will be 1 when ADPCM analysis/synthesis ends							
	b3	Will be 1 at the end of read/write/analysis/synthesis							
	b2-b1	Not used (always“00”)							
	b0	Will be 1 during the ADPCM analysis/synthesis (if b7 of $07 are 1)							

12.5.1 – MSX-Audio access ports

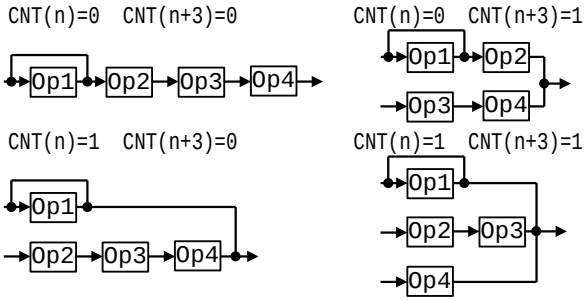
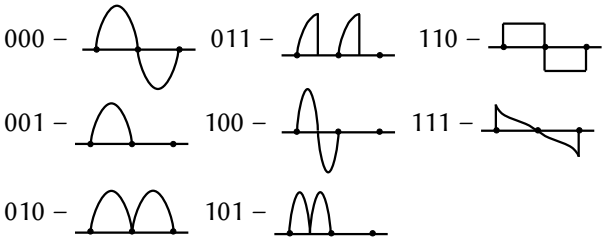
Porta		b7	b6	b5	b4	b3	b2	b1	b0	Short Description
C0H	W	Register number (01H a C8H)								Select register
	R	INT	T1	T2	EOS	BUF	•	•	PCM	Read status register
C1H	W/R	Data byte								Write/read data in the/from MSX Audio

12.6 – MAP OF THE OPL4 REGISTERS (YMF278)

12.6.1 – Register Array #0

FM generator – Register Array 0 (A1 = "1")									
Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H $01H	Test								Test registers
$02H $03H	1st Timer (80,8 µS) 2nd Timer (323,1 µS)								Time registers
$04H	RST	MT1	MT2	•	•	•	ST2	ST1	Flag register
	b7 b6 b5 b4-b2 b1 b0	RST – If write 1, resets b5, b6 and b7. MT1 – If write 1, b0 will be 0. MT2 – If write 1, b1 will be 0. Not used (always"000") ST2 – $03 Start/stop control (1=start counter) ST1 – $032Start/stop control (1=start counter)								
$05H	Only in the Register Array 1								
$08H	•	NTS	•	•	•	•	•	•	Keyboard configuration
	b7 b6 b5~b0	Not used (always"0") NTS – If 0, the separation point is determined by the higher 2 bits of F_number. If 1, the separation point is determined only by the MSB of F_number. Not used (always"000 000")							
$20H ⋮ $35H	AM	VIB	EGT	KSR	Multiple				Instruments definition
	b7 b6 b5 b4 b3~b0	AM (1=tremolo on (Frequency: 3,7Hz) VIB (1=vibrato on (Frequency: 6,4Hz) EG-TYP (0=decaying sound; 1=sustained sound) If 0, KSR⟶0~3; If 1, KSR⟶0~15 Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)								
$40H ⋮ $55H	KSL		Total level						KSL (00= 0dB/octave, 01=1,5dB, 10=3dB, 11=6dB) Total level (b0=0,75dB, b1=1,5dB ..... b5=24dB)
$60H ⋮	Attack Rate (AR)				Decay Rate (DR)				Attack (0dB a 96dB ⟶ min. 0,2 mS; max 2826 mS)

<table>
<tr><td>$75H</td><td colspan="8"></td><td colspan="2"></td><td>Decay (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$80H
⋮
$95H</td><td colspan="8">Sustain Level
(SL)</td><td colspan="2">Release Rate
(RL)</td><td>Sustain (b7=24dB, b6=12dB,
b5=6dB, b4=3dB)
Release (0dB a 96dB →
min. 2,4 mS; max 39 280 mS)</td></tr>
<tr><td>$A0H
⋮
$A8H</td><td colspan="10">Frequency (LSB 8 bits)</td><td>Frequency (b7~b0)</td></tr>
<tr><td rowspan="2">$B0H
⋮
$B8H</td><td colspan="2">•</td><td colspan="2">•</td><td>KEY</td><td colspan="3">Octave</td><td colspan="2">Freq.
MSB
2 bits</td><td>Freq. MSB 2 bits (b9~b8)
Octave (FM)
Key on/off (FM)</td></tr>
<tr><td colspan="11">b7~b6 Not used (always“00”)
b5 0=key off; 1=key on (voice active)
b4~b2 Define the octave. The fourth is 011.
b1~b0 Frequency MSB 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 110</td></tr>
<tr><td rowspan="2">$BDH</td><td>AM</td><td>VIB</td><td>BAT</td><td>BD</td><td>SD</td><td>TOM</td><td>TCY</td><td>HH</td><td colspan="3">Controle da bateria do FM</td></tr>
<tr><td colspan="11">b7 Tremolo level (0=1dB. 1=4,8dB)
b6 Vibrato level (0=7%; 1=14%)
b5 0=Melody mode; 1=Rhythm mode
b4 1=Bass Drum
b3 1=Snare Drum
b2 1=Tom-tom
b1 1=Top Cymbal
b0 1=High-Hat</td></tr>
<tr><td rowspan="2">$C0H
⋮
$C8H</td><td colspan="2">•</td><td colspan="2">•</td><td colspan="2">•</td><td colspan="2">Feedback</td><td>CON</td><td>Feedback factor; connect type</td></tr>
<tr><td colspan="11">b7~b4 Not used (always“0000”)
b3~b1 Feedback factor (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π)
b0 Connection type (0=serie; 1=paralell)
For 4 operators:
A1 Channel CNT(n) CNT(n+3)
0 1 C0H C3H
0 2 C1H C4H</td></tr>
</table>

		0 3 C2H C5H 1 4 C0H C3H 1 5 C1H C4H 1 6 C2H C5H 
	Operators (for $20H~$35H and $C0H~$C8H) Oper: 01 02 03 04 05 06 07 08 09 Voz: 1 2 3 1 2 3 4 5 6 Reg: $20 $21 $22 $23 $24 $25 $28 $29 $2A Freq: $A0 $A1 $A2 $A0 $A1 $A2 $A3 $A4 $A5 Oper: 10 11 12 13 14 15 16 17 18 Voz: 4 5 6 7 8 9 7 8 9 Reg: $2B $2C $2D $30 $31 $32 $33 $34 $35 Freq: $A3 $A4 $A5 $A6 $A7 $A8 $A6 $A7 $A8	The operators are associated as below: $20/$40/$60/$80/$A0/$B0/$C0 or $23/$43/$63/$83/$A0/$B0/$C0
$E0H ⋮ $F5H	• • • • • Wave Select	Waveform select
	b7~b3 b2~b1	Not used (always“00 000”) Waveform select: 

12.6.2 – Register Array #1

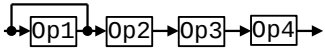
FM generator – Register Array 1 (A1 = “H”)									
Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description

$00H $01H	Test								Test registers
$02H $03H	Only in the Register Array 0								
$04H	•	•	Connection SEL						4-operators selection
	b7~b6 b5~b0		Not used (always“00”) Enable 4-operators mode for the respective slot: Bit: b5 b4 b3 b2 b1 b0 Slot: 6 5 4 3 2 1						
$05H	•	•	•	•	•	•	NEW2	NEW	Expansion register
	b7~b2 b1 b0		Not used (always“000 000”) If 1, enable OPL4 mode (Register Array 1) If 1, enable OPL3 mode (Register Array 0)						
$08H	Only in the Register Array 0								
$20H ⋮ $35H	AM	VIB	EGT	KSR	Multiple				
	b7 b6 b5 b4 b3~b0		AM (1=trêmolo on (frequency: 3,7Hz) VIB (1=vibrato on (frequency: 6,4Hz) EG-TYP (0=decaying sound; 1=sustained sound) If 0, KSR⟶0~3; If 1, KSR⟶0~15 Multiplication factor (0=1/2, 1=1, 2=2, 3=3, ..., 15=15)						
$40H ⋮ $55H	KSL		Nível total						KSL (00= 0dB/octave, 01=1,5dB, 10=3dB, 11=6dB) Total level (b0=0,75dB, b1=1,5dB ..... b5=24dB)
$60H ⋮ $75H	Attack Rate (AR)			Decay Rate (DR)					Attack (0dB a 96dB ⟶ min. 0,2 mS; max 2826 mS) Decay (0dB a 96dB ⟶ min. 2,4 mS; max 39 280 mS)
$80H ⋮ $95H	Sustain Level (SL)			Release Rate (RL)					Sustain (b7=24dB, b6=12dB, b5=6dB, b4=3dB) Release (0dB a 96dB ⟶ min. 2,4 mS; max 39 280 mS)
$A0H ⋮ $A8H	Frequency (LSB 8 bits)								Frequency (b7-b8)

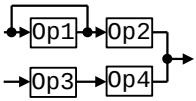
$B0H ⋮ $B8H	•	•	KEY	Octave	Freq. MSB 2 bits	Freq. MSB 2 bits (FM) Octave (FM) Key on/off (FM)	
	b7~b6 b5 b4~b2 b1~b0	Not used (always“00”) 0=key off; 1=key on (active voice) Defines the octave. The fourth is 011. Frequency MSB 2 bits. The C central note of 440 Hz is obtained with b1~b0=10 and $A0H~A8H=01 000 110					
$BDH	Only in the Register Array 0						
$C0H ⋮ $C8H	•	•	•	•	Feedback	CON	Feedback factor/connect type
	b7~b4 b3~b1 b0	Not used (always“0000”) Feedback factor (0=0; 1=π/16; 2=π/8; ...; 6=2π; 7=4π) Connection type (0=serie; 1=paralell) For 4 operators:					

A1	Canal	CNT(n)	CNT(n+3)
0	1	C0H	C3H
0	2	C1H	C4H
0	3	C2H	C5H
1	4	C0H	C3H
1	5	C1H	C4H
1	6	C2H	C5H

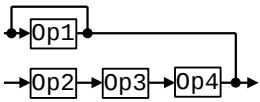
CNT(n)=0 CNT(n+3)=0



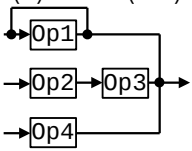
CNT(n)=0 CNT(n+3)=1



CNT(n)=1 CNT(n+3)=0



CNT(n)=1 CNT(n+3)=1



Operators (para$20H~$35H e$C0H~$C8H)

Oper:	19	20	21	22	23	24	25	26	27
Voz:	1	2	3	1	2	3	4	5	6

Reg: $20$21$22$23$24$25$28$29$2A

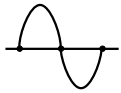
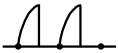
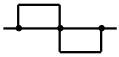
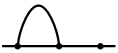
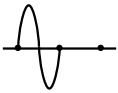
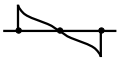
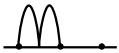
Freq:$A0$A1$A2$A0$A1$A2$A3$A4$A5

The operators are associated as below:

$20/$40/$60/$80/$A0/$B0/$C0

or

$23/$43/$63/$83/$A0/$B0/$C0

	Oper: 28 29 30 31 32 33 34 35 36 Voz: 4 5 6 7 8 9 7 8 9 Reg: $2B$2C$2D$30$31$32$33$34$35 Freq: $A3$A4$A5$A6$A7$A8$A6$A7$A8						
$E0H ⋮ $F5H	•	•	•	•	•	Wave Select	Waveform select
	b7~b3	Not used (always“00 000”)					
	b2~b1	Waveform select 000 –  011 –  110 –  001 –  100 –  111 –  010 –  101 – 					

12.6.3 – Wave synthesis

Wave synthesis									
Reg	b7	b6	b5	b4	b3	b2	b1	b0	Short Description
$00H $01H	Test								Test registers
$02H	ID disp		Wave header				MT	MM	Special functions
	b7~b5	OPL4-ID (b7=0; b6=0; b7=1)							
	b4~b2	Wave table header: 000=0 to 511 (000 000H) 100=384 to 511 (200 000H) 001=384 to 511 (080 000H) 101=384 to 511 (280 000H) 010=384 to 511 (100 000H) 110=384 to 511 (300 000H) 011=384 to 511 (180 000H) 111=384 to 511 (380 000H)							
	b1	Audio memory type (0=ROM; 1=RAM)							
	b0	Audio memnry access (0=OPL4; 1=CPU)							
$03H	•	•	a21	a20	a19	a18	a17	a16	Audio memory adress
$04H	a15	a14	a13	a12	a11	a10	a9	a8	
$05H	a7	a6	a5	a4	a3	a2	a1	a0	

$06H	Memory data					Data register
$08H ⋮ $1FH	Wave table number LSB (n7~n0)					24 registers with the LSB number (n7~n0) of the wave table
$20H ⋮ $37H	F_number (f6~f0)				Tab Wave (n8)	24 registers with the 7 bits LSB frequency and MSB wave table number (n8)
$38H ⋮ $4FH	Octave (o3~o0)		Pseudo-rev	F_number (f9~f7)		Octave (–7 a +7) Pseudo-reverberation Frequency (3 bits MSB)
	b7~b0 b7~b6 b3 b7~b4	With “b0” (n8) select up to 512 samples (0~511) With “b2-b1-b0” (f9~f7) defines the frequency If “1” turn pn the pseudo-reverberation; if “0”, turn off Octave. Range from –7 to +7 (–8 can´t used). With F_number defines the frequency. For octave = 1 and F_number = 0, the frequency is 44,1 Khz. $f(¢) = 1200 * (\text{octave} - 1) + 1200 * \log_2 \frac{1024+\text{F\_number}}{1024}$				
$50H ⋮ $67H	Total level (l6~l0)				DL	Total level 7 bits (l6~l0) Direct level
	b7-b1 b0	Total level (b7=-24dB, b6=-12dB, ... b1=-0,375dB) Direct level (0=change envelope during interpolation; 1=change envelope immediately)				
$68H ⋮ $7FH	Key on	Damp	LFO RST	CH	Panpot	miscellaneous functions and stereo balancing (Panpot)
	b7 b6 b5 b4 b3~b0	0=key on; 1=key off 0=Damp off; 1=Damp on LFO RST (0=turn on the LFO; 1=turn off the LFO) 0=Wave mixed with FM; 1=No mixing				
$80H ⋮	•	•	LFO (s2~s0)		VIB (v2~v0)	Tremolo and vibrato frequency (LFO)

```
Panpot:     0   1   2  …  6   7   8   9  …  13  14  15
Left (dB)   0  -3  -6  … -18 -∞  -∞   0  …   0   0   0
Right(dB)   0   0   0  …  0   0  -∞  -18 …  -9  -6  -3
```

$97H					Vibrato level (VIB)
	b7~b6 b5~b3 b2~b0	Not used LFO (0=0,168Hz, 1=2,019Hz, ... 7=7,066Hz) Vibrato level (0=off, 1=3,378; 2=5,065, ... 7=79,31)			
$98H ⋮ $AFH	Attack Rate		Decay Rate (1)		Attack Rate 10-90% ⟶ (1=3715 mS; 14=0,23 mS) Decay 1 Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)
$B0H ⋮ $C7H	Decay Level		Decay Rate (2)		Decay Level (b7=-24; b6=-12; b5=-6; b4=-3 dB) Decay 2 Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)
$C8H ⋮ $DFH	Rate Correction		Release Rate		Rate Correction Release Rate
	b7~b4 b3~b0	Rate Correction: (RATE = (OCT + RC)*2 + f9 + RD) OCT = Octave (–7 a +7 in $38H~$4FH) RC = Rate correction (0~14 in $C8H~$DFH) f9 = bit “f9” of F_number ($38H~$4FH) RD = AR, D1R, D2R e RR values (0001=04; 0010=08; ...; 1111=63) Release Rate 10-90% ⟶ (1=19 040 mS; 14=1,18 mS)				
$E0H ⋮ $F7H	•	•	•	•	•	AM(a2~a0)	Tremolo level
	b7~b3 b2~b0	Not used Tremolo level (0=off; 1=1,781; ...; 7=11,91)					
$F8H	•	•	Mix FM_R			Mix FM_L	Nível de saída FM
	b7~b6 b5~b3 b2~b0	Not used (always“00”) Right FM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞) Left FM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞)					
$F9H	•	•	Mix PCM_R			Mix PCM_L	PCM output level
	b7~b6 b5~b3 b2~b0	Not used (always“00”) Right PCM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞) Left PCM level (0=0; 1=-3; 2=-6; ... 6=-18dB; 7=∞)					

12.6.4 – OLP4 access ports

Porta		b7	b6	b5	b4	b3	b2	b1	b0	Short Description
C4H	W	Register number (00H a F5H)								Select reg. FM array 0
	R	IRQ	FT1	FT2	•	•	•	LD	BSY	Read status register
	b7	IRQ – Interrupt Request (will be "1" when FT1 or FT2 be "1")								
	b6	FT1 – Will be "1" when timer 1 end counting								
	b5	FT2 – Will be "1" when timer 2 end counting								
	b4~b2	Not used (always"000")								
	b1	LD – Will be "1" during PCM header reading by the OPL4 (Valid when 05H_NEW2 bit of the array 1 is "1".)								
	b0	BUSY – Will be "1" during registers data writing (Valid when 05H_NEW2 bit of the array 1 is "1".)								
C5H	W	Data byte								Write data in the register
C4H	W	Register number (00H a F5H)								Select FM reg array 0
C7H	R	Mirror of C5H								C5H access is preferred
7EH	W	Register number (00H a F9H)								Select PCM registers
7FH	W/R	Data byte								Read/write data in regs

12.6.5 – Wave table synthesis header

End	b7	b6	b5	b4	b3	b2	b1	b0	
00H	d1	d0	s21	s20	s19	s18	s17	s16	d1, d0 ⟶ 00=8bits; 01=12 bits; 10=16 bits
01H	s15	s14	s13	s12	s11	s10	s9	s8	
02H	s7	s6	s5	s4	s3	s2	s1	s0	s21~s0 = start adress
03H	l15	l14	l13	l12	l11	l10	l9	l8	Loop adress
04H	l7	l6	l5	l4	l3	l2	l1	l0	
05H	e15	e14	e13	e12	e11	e10	e9	e8	End adress
06H	e7	e6	e5	e4	e3	e2	e1	e0	

07H	•	•	f2	f1	f0	v2	v1	v0	LFO freq. and vibrato level
08H	ar3	ar2	ar1	ar0	dr3	dr2	dr1	dr0	Attack Rate; Decay 1 Rate
09H	dl3	dl2	dl1	dl0	dr3	dr2	dr1	dr0	Decay Level; Decay 2 Rate
0AH	rc3	rc2	rc1	rc0	rr3	rr2	rr1	rr0	Rate correct; Release Rate
0BH	•	•	•	•	•	am2	am1	am0	AM level (tremolo)

12.6.6 – Wave data lenght

16 bits	d15	d14	d13	d12	d11	d10	d9	d8	+00H
	d7	d6	d5	d4	d3	d2	d1	d0	+01H
12 bits	d11	d10	d9	d8	d7	d6	d5	d4	+00H
	d3	d2	d1	d0	d3	d2	d1	d0	+01H
	d11	d10	d9	d8	d7	d6	d5	d4	+02H
8 bits	d7	d6	d5	d4	d3	d2	d1	d0	+00H

12.7 – MAP OF THE SCC REGISTERS (2212/2312)

Adresses	Short Description (SCC)
9800H~981FH	Waveform of the voice #1
9820H~983FH	Waveform of the voice #2
9840H~985FH	Waveform of the voice #3
9860H~987FH	SCC : Write/read: Waveform of the voices #4 e #5 SCC+: Read: Waveform of the voice #4
9880H~9881H	Frequency of the voice #1
9882H~9883H	Frequency of the voice #2
9884H~9885H	Frequency of the voice #3
9886H~9887H	Frequency of the voice #4
9888H~9889H	Frequency of the voice #5
	Example: 9880=[f7][f6][f5][f4][f3][f2][f1][f0] 9881=[•][•][•][•][f11][f10][f9][f8] $Ftone = \frac{F\_clock}{32 * ((f11\sim f0)+1)}$ (F_clock=3,579545 MHz)
988AH	Volume of the voice #1 (0 to 15)
988BH	Volume of the voice #2 (0 to 15)
988CH	Volume of the voice #3 (0 to 15)
988DH	Volume of the voice #4 (0 to 15)
988EH	Volume of the voice #5 (0 to 15)
998FH	[•][•][•][v5][v4][v3][v2][v1] v5=1 ⟶ turn on voice #5 v4=1 ⟶ turn on voice #4, etc
9890H~989FH	Mirror of 9880H~988FH
98A0H	SCC: no function SCC+: waveform read of the voice #5 (no write alowed)
98A1H~98BFH	Mirrors of 98A0H
98C0H	SCC: Mirror of 98A0H SCC+: Deformation register

98C1H~98DFH	SCC: Mirror of 98A0H SCC+: Mirror of 98C0H
98E0H	SCC: Deformation register R R • • • • P P PP: 11/10→Ftone*16; 01→Ftone*256; 00→ Ftone*1 RR: 11 → white noise voices 4 and 5 cfe waveform 01 → continuous white noise 00 → no white noise SCC+: No function
98E1H~98FFH	Mirrors of 98E0H

12.7.1 – Acess adresses for SCC

Endereços	Short Description (SCC+)
B800H~B81FH	Voice #1 waveform
B820H~B83FH	Voice #2 waveform
B840H~B85FH	Voice #3 waveform
B860H~B87FH	Voice #4 waveform
B880H~B89FH	Voice #5 waveform
B8A0H~B8A1H	Voice #1 frequency
B8A2H~B8A3H	Voice #2 frequency
B8A4H~B8A5H	Voice #3 frequency
B8A6H~B8A7H	Voice #4 frequency
B8A8H~B8A9H	Voice #5 frequency
	Example: B8A0=f7 f6 f5 f4 f3 f2 f1 f0 B8A1=• • • • f11 f10 f9 f8 $Ftone = \frac{F\_clock}{32 * ((f11\sim f0)+1)}$ (F_clock=3,579545 MHz)
B8AAH	Voice #1 volume (0 to 15)
B8ABH	Voice #2 volume (0 to 15)
B8ACH	Voice #3 volume (0 to 15)
B8ADH	Voice #4 volume (0 to 15)

B8AEH	Voice #5 volume (0 to 15)
B8AFH	| • | • | • | v5 | v4 | v3 | v2 | v1 | v5=1 ⟶ turn on voice #5 v4=1 ⟶ turn on voice #4, etc
B8B0H~B8BFH	Mirror of B8A0H~B8AFH
B8C0H	| R | R | • | • | • | • | P | P | – Deformation register PP: 11/10⟶Ftone*16; 01⟶Ftone*256; 00⟶ Ftone*1 RR: 11 ⟶ white noise voices 4 and 5 cfe waveform 01 ⟶ continuous white noise 00 ⟶ no white noise
B8C1H~B8DFH	Mirrors of B8C0H
B8E0H~B8FFH	No function
B900H~BFFDH	???
BFFEH~BFFFH	| • | • | S | M | • | B3 | B2 | B1 | – Mode register S – SCC mode (0=SCC; 1=SCC+) M – Memory mode (0=bank select; 1=RAM) B3 – Memory bank #3 (0=bank select; 1=RAM) B2 – Memory bank #2 (0=bank select; 1=RAM) B1 – Memory bank #1 (0=bank select; 1=RAM)

BIBLIOGRAPHIC REFERENCES

APROFUNDANDO-SE NO MSX
Piazzi – Maldonado – Oliveira (Editora Aleph, 1986)

CARTÃO DE 80 COLUNAS & RS232C – MANUAL DE OPERAÇÕES
Gradiente (1989)

FM MUSIC MACRO YRM-104 – Owner's Manual
Yamaha (1984)

FUDEBA ASSEMBLER – Manual de Referência e Arquitetura MSX
Felipe Bergo (2002)

GR8NET Technical Databook and Programmer´s Guide
Age Labs (2019)

HBI-232MKII – Basic Manual
Age Labs & Ebsoft (2014)

LIVRO VERMELHO DO MSX, O (The Red Book)
McGraw Hill / Avalon Software (1988 / 1985)

MANUAL DE INSTRUÇÕES DA INTERFACE DE DRIVE DDX

MANUAL DO MICROPROCESSADOR Z-80
William Barden Jr. (Editora Campus, 1985)

MIDI MACRO MONITOR YRM-303 – Owner's Manual
Yamaha (1986)

MSX DATAPACK volumes 1, 2 e 3
ASCII Corporation (1991)

MSX-DOS version 2 – The advanced disk operating system for
MSX 2 computers – ASCII Corp (1988)

MSX MAGAZINE, Edição Dezembro de 1990
ASCII Corporation (1990)

MSX MAGAZINE, Edição ???
ASCII Corporation (1990)

MSX MOZAÏK, Edição nº 33
Editora desconhecida, Ano desconhecido

MSX TECHNICAL GUIDE BOOK
Ayumu Kimura (ASCAT Ashigaka, NIPPON, 1992)

MSX TECHNICAL DATA BOOK
Sony Corp (1984)

MSX turbo R TECHNICAL HANDBOOK
ASCII Corpotation (1991)

MSX2 TECHNICAL HANDBOOK
ASCII Corpotation (1985)

NEXTOR 2.0 User Manual
Konamiman (2014)

Nextor 2.1 Driver Development Guide
Konamiman (2020)

OPL4 YMF278B – APPLICATION MANUAL
Yamaha Corporation (1994)

PROGRAMAÇÃO AVANÇADA EM MSX
Figueredo – Maldonado – Rosseto (Editora Aleph, 1986)

PX-7 P-BASIC Reference Manual
Pioneer (1985)

TMS9918A/TMS9928A/TMS9929A Video Display Processors Data Manual – Texas Instruments (1982)

V9938 MSX-VIDEO – APPLICATION MANUAL
Nippon Gakki Co. Ltd. (Yamaha, 1985)

V9938 MSX-VIDEO – TECHNICAL DATA BOOK
Nippon Gakki Co. Ltd. (Yamaha, 1985)

V9958 MSX-VIDEO – TECHNICAL DATA BOOK
Yamaha Corporation (1989)

V9990 E-VDP-III – APPLICATION MANUAL
Yamaha Corporation (1992)

Y9850 MSX-AUDIO – APPLICATION MANUAL
Nippon Gakki Co. Ltd. (Yamaha, 1985)

YM2413 FM OPERATOR TYPE LL (OPLL) – APPLICATION MANUAL
Yamaha Corporation (1987)

https://www.gigamix.jp/ds2/

https://www.msx.org/wiki/I/
O_Ports_List#The_register_of_internal_I.2FO_ports_control

http://msxbanzai.tni.nl/v9990/manual.html

https://github.com/Konamiman/MSX-UNAPI-specification/
tree/master/docs

http://www.symbos.de/

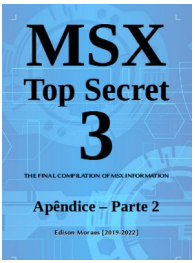
https://www.msx.org/wiki/MSX-HID

OTHER BOOKS BY THE AUTHOR

YBYMARÃ – A Cidade do Outro Lado

MSX Top Secret

MSX Top Secret 2



MSX Top Secret 3 – Appendix