

# MSX2 TECHNICAL HANDBOOK

Published by ASCII CORPORATION Japan in 1987

Adapted to text files by Nestor Soriano (Konami Man) in 1997

-----  
This disk contains the whole MSX2 Technical Handbook (except appendix 7 and appendix 9) plus Turbo-BASIC compiler manual, all in 9 text files with the following format:

- File format: ASCII MS(X)-DOS.
- Line width: 78 characters.
- Carriage return marks at the end of each line.
- Standard tabulation marks are used (in columns with a number multiply of 8)

Format compatibility warranty: all files have been generated with a MSX computer.

ASCII character 92 is used as the inverse bar; therefore, viewing the files in a MSX2+ or Turbo-R computer with the original character set will cause some figures to appear slightly distorted. Anyway, they will still be perfectly readable.

## Files list:

TH-1	PMA	8192 bytes	Chapter 1 - MSX System Overview
		Decompressed: 24930 bytes	
TH-2	TXT	117450 bytes	Chapter 2 - BASIC
TH-3	TXT	108270 bytes	Chapter 3 - MSX-DOS
TH-4A	TXT	131604 bytes	Chapter 4 - VDP and Display Screen (1-5)
TH-4B	TXT	92922 bytes	Chapter 5 - VDP and Display Screen (6)
TH-5A	TXT	97157 bytes	Chapter 5 - Access to Peripherals Through BIOS (1-6)
TH-5B	TXT	46762 bytes	Chapter 6 - Access to Peripherals Through BIOS (7)
TH-AP	TXT	106609 bytes	Appendix 1 - BIOS listing Appendix 2 - MATH-PACK Appendix 3 - Bit Block Transfer Appendix 4 - Work Area Listing Appendix 5 - VRAM Map Appendix 6 - I/O Map Appendix 8 - Control Codes Appendix 10- Escape Sequences
KUNBASIC	TXT	7379 bytes	MSX BASIC-KUN Compiler

\*\* NOT AVAILABLE \*\*      Appendix 7 - Cartridge Hardware  
                                 Appendix 9 - Character Set

Files are not compressed, then you can use this disk directly with any text editor/viewer. Problem: there is not enough room in 713K to afford all the files. Because of this, TH-1 (the less important chapter) is compressed.

This manual was reproduced without the consent of the original authors, that is, ASCII staff. I think that this people forgot the MSX system a long time ago; anyway, for any complaint contact with me in <konamiman@geocities.com>.

Arigatos to Ramon Serna and Javi Lavandeira for lending to me theirs "original" THs in march and october, respectively.

## MSX2 TECHNICAL HANDBOOK

-----

Edited by: ASCII Systems Division  
Published by: ASCII Coprporation - JAPAN  
First edition: March 1987

Text files typed by: Nestor Soriano (Konami Man) - SPAIN  
March 1997

Changes from the original:

none

-----

## CHAPTER 1 - MSX SYSTEM OVERVIEW

The MSX2 was designed to be fully compatible with the MSX1, but there are many enhanced features in the MSX2. Chapter 1 introduces the enhanced features of the MSX2, and shows block figures and standard tables. This information is conceptual, but will be needed to understand dexcriptions in volume 2 and later.

### 1. FROM MSX1 TO MSX2

To begin with, let us took back to the original purpose or intention of MSX and then sum up the transition from MSX1 to MSX2.

#### 1.1 What is MSX?

MSX was announced as a new 8-bit computer standard in the autumn of 1983. In early days the word "compatibility" was not understood correctly and there were misunderstandings that MSX could execute programs from other computers. Since MSX can execute programs only for MSX, it was said that were was no difference from the PC series (NEC) or FM series (Fujitsu) personal computers, which could only execute programs using their format.

Several years passed before personal computers became popular. In the early days only dedicated enthousiasts bought computers, which were difficult to use, and, needless to say, incompatible. They were satisfied to tinker with the computer and study it. But now computer use has expanded to include several classes of users. In other words, the personal computer is becoming a commodity item such as televisions or radio cassette recorders. Therefore,

"compatibility" is coming to be a problem. If each TV station needs a different television set or if each radio cassette recorder needs a different tape, do you suppose they would be popular? Software or programs of the computer as a home electric product must be compatible.

The design team for MSX considered these problems. Since a computer is most powerful when left flexible and easy to expand, a "final" standard format was not practical. There are too many matters to define and hardware is constantly improving. Therefore MSX started with fixing format of the most fundamental hardware and software such as DOS and BASIC, and the hardware bus which is the basis for expansion. Since the computer is used by itself and does not interact with other computers, the problem is small. But formats must be fixed if the computer is to be connected to "peripherals" and handle or accumulate various data. Fortunately MSX had the approval of many home appliance electric companies and an MSX format was established early. This allowed the system to be well known so that several manufacturers could make compatible peripherals for the MSX standard.

Some of the useful features included in the MSX system include the use of double precision BCD for normal BASIC arithmetic and the same file format as MS-DOS. The real capabilities of the MSX machine will come to light as it is used across several fields.

## 1.2 Environment of the MSX

Over one million MSX machines had been sold by December 1985 and are used mainly as game machines or primers by primary and junior high school students. But MSX use has gradually spread to include such uses as communication terminals, Japanese word processing, factory automation, and audio visual control. For improving its capabilities, a disk system and MSX-DOS have been prepared, and languages such as C, FORTH, and LOGO are available. BIOS, which is the collection of input/output routines in BASIC ROM, and BDOS, which resides in the disk interface ROM and has compatibility with CP/M system calls have both been improved. So an excellent programming environment is now available. Chinese Character input, light pen and mouse input, and the RS-232C interface have been standardised, and standardisation of other peripherals is proceeding. The keyboard and character set are consistent with international standards, and there are minor variations to satisfy the needs of individual countries.

Several new peripherals have been developed. Standard devices include printers, disk drives, and mice; audio/visual devices include laser drives, VTRs, synthesizer controllers, and video acquisition systems. Factory Automation devices include robot controllers, room temperature controllers, various adaptors for modem and telephone lines, and a health controller combined with a hemodynamometer has been developed. So you can see that the potential uses for MSX computers has really grown.

Many applications other than games are now supplied on disks and are becoming more practical. There are now Japanese word processors capable of clause transformation, data bases which can exchange data with higher-level systems, and CAI and CAD systems.

## 1.3 Extended Contents of MSX2

MSX2 was announced in May 1985 as a system having upgraded compatibility with MSX. Programs created under the MSX environment can be executed on MSX2 without any modifications, even at the assembly language level. Data and programs stored on cassette tapes or disks can be used without modification. Features added by the MSX2 system are improved screen display, higher resolution, more colours available, and higher graphics speed. A battery-powered clock and RAMDISK feature have also been added. In this manual the name MSX2 refers to the computer made along the MSX2 standard and the name MSX1 refers to the computer made along the previous MSX standard.

System configuration is shown in Figures 1.1 and 1.2 and Table 1.1 and indicate the differences between MSX1 and MSX2. The differences are described as follows:

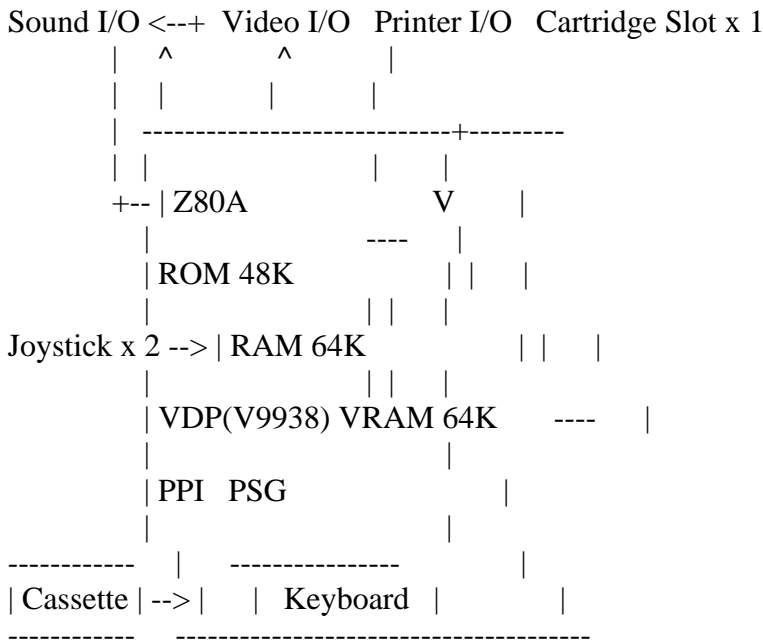
Table 1.1 MSX2/MSX1 standard comparison

	MSX2	MSX1
CPU	Z80A or equivalent (clock 3.579545 MHz +- 1%)	
ROM	48K (MSX-BASIC version 2.0) MAIN-ROM 32K SUB-ROM 16K	32K (MSX-BASIC ver 1.0) MAIN-ROM 32K
MEMORY RAM	64K or more	8K or more
VRAM	64K or 128K	16K
LSI for VDP	V-9938 (MSX-VIDEO)	TMS9918 or equivalent
CMT	FSK 1200/2400 baud	
PSG	8 octaves tri-chord output (AY-3-8910 compatible)	
Keyboard	Alphanumeric Graphic symbols	Alphanumeric Graphic symbols
Floppy disk (*)	Based on MS-DOS format	
Printer	8-bit parallel	(*)
ROM cartridge	I/O bus with slot for game cartridge and expansion bus	
Joystick	2	1 or 2 (*)
CLOCK-IC	Standard	(*)
RAM disk feature	Standard	Different for each maker

(\*) Optional

Figure 1.1 MSX2 system configuration

### 1. Minimum configuration



2. Software support range +-- Lightpen, Superimpose, Video digitize

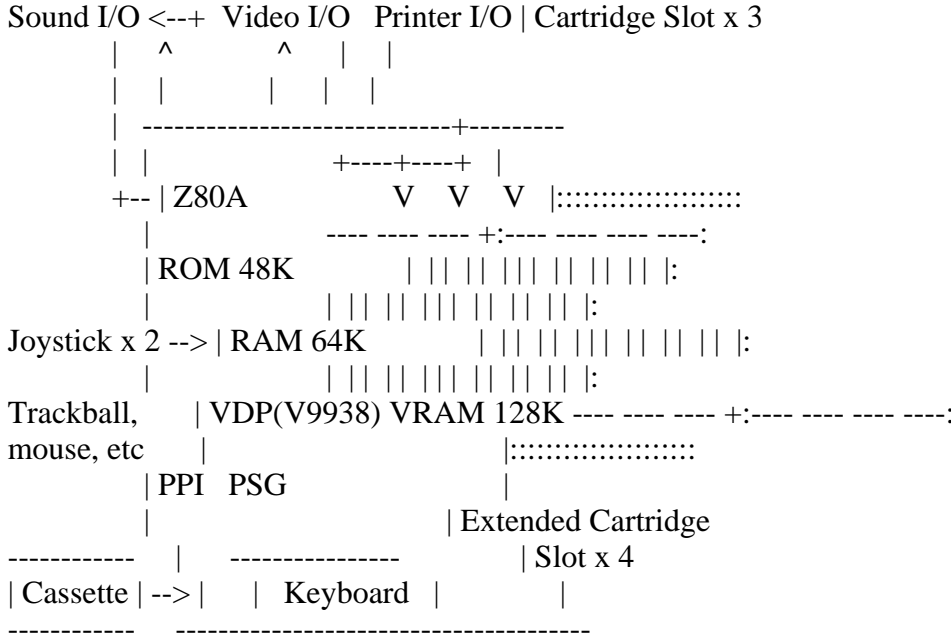
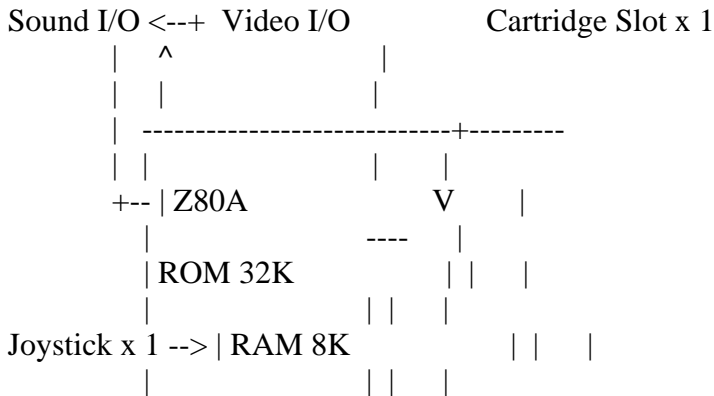
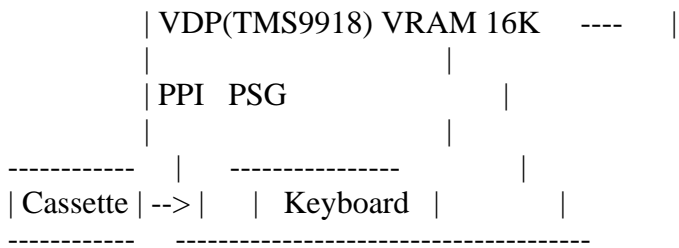


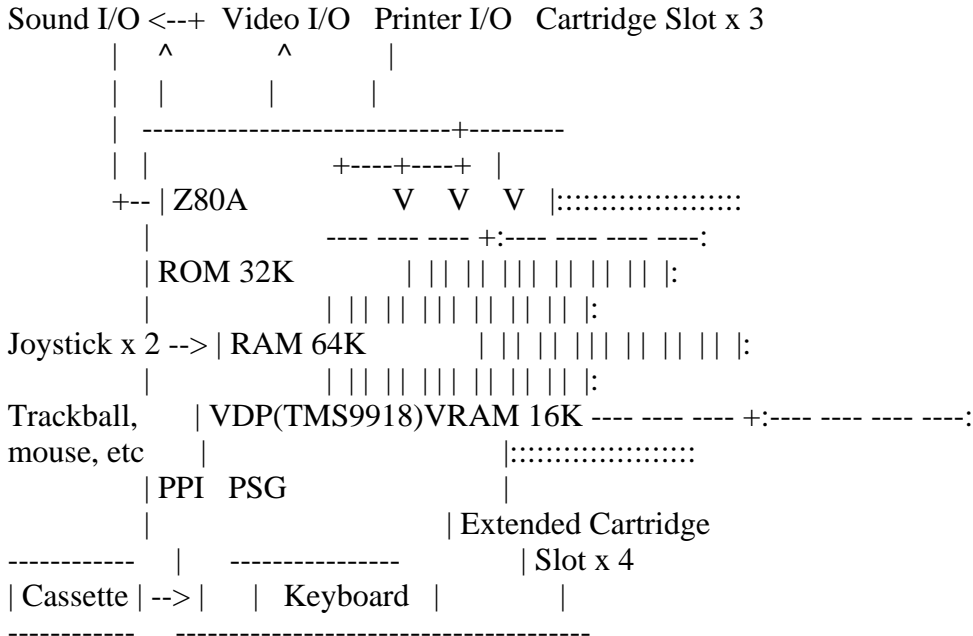
Figure 1.2 MSX1 system configuration

1. Minimum configuration





## 2. Software support range



### \* MSX-BASIC

BASIC has also been extended from version 1.0 to version 2.0 in order to support a new VDP, backup RAM, CLOCK-IC, and so on. Compatibility with MSX1 is maintained. When using the newly extended screen mode, be careful when specifying range, since ranges are slightly different in MSX2.

MSX2 has three types of memory, ROM, RAM, VRAM, which are described below.

### ROM

Standard ROM size is 48K bytes. The MSX ROM uses only 32K bytes. The extra 16K bytes portion of the MSX2 contains routines supporting the extended features.

The "MAIN-ROM" consists of 32K bytes and contains the BASIC interpreter, and the "extended ROM" or "SUB-ROM" consists of 16K bytes and contains routines for the extended features.

### RAM

Standard RAM size is 64K bytes, which is large enough so that MSX-DOS can be executed. The RAM size of MSX1 varied from 8K to 64K bytes, so in some cases large programs could not be executed without expanding RAM. MSX2 does not

have this problem.

## VRAM

A minimum of 64K bytes are required for VRAM in order to execute the added features of the screen display. VRAM is thus four times larger than in MSX1, which had only 16 K bytes VRAM. But many machines actually use a VRAM size of 128K bytes, which is eight times larger. Machines with 128K bytes VRAM can display 256 colours at the same time.

MSX machines which have 64K bytes VRAM but cannot be expanded to 128K bytes are marked "VRAM64K" on their catalogue or packaging.

## \* VDP

The MSX series computers use a video display processor (VDP) type LSI chip for controlling the screen output. The VDP used for MSX1 was the TMS9918, but the MSX2 uses the V9938 (MSX-VIDEO), which has upper and full compatibility with the TMS9918 and can execute software for TMS9918 without any modification.

Table 1.2 shows the VDP standard and Table 1.3 shows each screen mode. V9938 is an excellent LSI chip with digitising, superimposing, and hardware scrolling features. Chapter 4 of this manual describes it in detail.

Table 1.2 VDP specifications

	V9938	TMS9918
Screen mode	10 (see table 1.3)	4
Number of dots (horizontal x vertical)	512 x 212 maximum 424 dots for vertical can be achieved by interlace feature	256 x 192 maximum
Number of colours to specify	512 maximum	16 maximum
Colour		
Number of colours to display at the same time	256 maximum	16 maximum
Character set	alphanumeric + graphic symbols 256 characters 8 x 8 dots	
Sprite colour	16 maximum per sprite	1 per sprite
Palette feature	Yes	No

Table 1.3 V9938 screen mode

Mode	Number of characters	Dots	Colours	Palette	Sprite
* Text 1	40 x 24		2 from 512	Yes	No
Text 2	80 x 24		4 from 512	Yes	No
* Multi-colour	64 x 48		16 from 512	Yes	Mode 1
* Graphic 1	32 x 24		16 from 512	Yes	Mode 1
* Graphic 2	256 x 192		16 from 512	Yes	Mode 1
Graphic 3	256 x 192		16 from 512	Yes	Mode 2
Graphic 4	256 x 212		16 from 512	Yes	Mode 2
Graphic 5	512 x 212		4 from 512	Yes	Mode 2
Graphic 6	512 x 212		16 from 512	Yes	Mode 2
Graphic 7	256 x 212	256 from 256		No	Mode 2

(\*) Feature modes available from TMS9918 (however, palette feature only from V9938).

#### \* Battery-powered Clock-IC

Battery-powered RAM is connected to the I/O port and is used for storage of setup information and for keeping track of the date and time. Setup information specifies the screen colour and mode at reset. This allows the user to set up the desired environment when the system is booted.

The CLOCK-IC works independently of the main power supply. After being set once new time settings are no longer required.

#### \* RAM Disk Feature

When using BASIC on MSX1 machines which had 64K bytes RAM, only 32K bytes of RAM were used; the other 32K bytes were unused since the BASIC interpreter occupied the address space. On MSX2 machines this unused RAM can be used as a RAMDISK. For users who do not have a disk drive, this feature is very useful when loading or saving BASIC programs temporarily.

## 2. MSX2 SYSTEM OVERVIEW

This section gives a simple overview of the MSX2 software and hardware systems. To help you understand the concepts, diagrams which would be useful when developing software, such as VRAM map, the I/O map, and the interface standard, are found in the APPENDIX of this manual.

### 2.1 Hardware overview

First of all, look at the block diagram in Figure 1.3 to understand the hardware configuration of the MSX2 as a whole.

Figure 1.3 MSX2 block diagram



-----  
CPU Z80A

|  
| -----  
+--| |--| ROM 48K (MSX-BASIC ver 2.0) |

| | | -----  
| | S | -----  
| | |--| MAIN RAM 64K |

| | L | -----  
| | ::::::::::: :::::::::::  
| | O |::: MEMORY MAPPER ::: RAM 64K to 4M :

| | | ::::::::::: :::::::::::  
| | T | :::::::::::  
| | |::: CARTRIDGE : I/O Cartridge (Disk, RS-232C)  
| | | ::::::::::: RAM Cartridge  
| ----- ROM Cartridge (Game, Application)  
| Slot Expansion Box, Etc.

| ----- --- Joystick Input

+--| PSG AY-3 8910 |---|  
| ----- --- Audio Output

| ::::::::::: :::::::::::  
|::: MSX-AUDIO (FM sound) ::: Audio Memory Maximum of 256K :

| -----  
+--| CASSETTE INTERFACE |::: Cassette

| -----  
+--| PRINTER INTERFACE |::: Printer

| -----  
+--| BATTERY BACKUP RAM + CLOCK IC |

| -----  
+--| PPI 8255A |-----+----| Keyboard |

| -----  
| | -----  
| | +----| Slot Holder |  
| -----

-----  
| MSX-VIDEO |----- RGB/Video/RF Output  
-----

| -----  
+--| VRAM 64K | VRAM 64K : Expansion RAM :

| :::::::::::  
|--: SUMPERIMPOSE :-----+---- Video Input

```

| ..... |
+--: DIGITISE :-----+
| ..... |

```

Note: The dotted lines represent optional features.

### 2.1.1 Address map

#### \* Memory map

The MSX2 has three kinds of memory: MAIN-ROM, SUB-ROM, and RAM. Each memory resides in an independent 64K address space and is allocated as shown in Figure 1.4 (1) (each 64K space is called a "slot", which consists of four 16K areas called "pages"). Figures 1.3 (2) and (3) show memory usage when using BASIC and MSX-DOS, respectively.

For each class of memory, Figure 1.5 shows the memory map of Figure 1.4 (1)(a), Figure 1.6 for Figure 1.4 (1)(b), and Figure 1.7 for Figure 1.4 (1)(c). There is also a VRAM map and I/O map whose standards are defined. They are found in the APPENDIX.

Figure 1.4 MSX2 standard memory

#### (1) Physical allocation of standard memories

	(a)	(b)	(c)
0000H	-----		
Page 0	(1)	(3)	(4)
	MAIN-ROM	SUB-ROM	RAM
4000H	-----	-----	-----
Page 1	(2)	not	(5)
	MAIN-ROM	used	RAM
8000H	-----	-----	-----
Page 2	not	not	(6)
	used	used	RAM
C000H	-----	-----	-----
Page 3	not	not	(7)
	used	used	RAM
	-----		
	MAIN-ROM	SUB-ROM	64K-RAM
	SLOT	SLOT	SLOT

#### (2) CPU memory space when using BASIC      (3) CPU memory space when using MSX-DOS

	(2)	(3)
0000H	-----	
Page 0	(1)	(3)
	MAIN-ROM	SUB-ROM
4000H	-----	-----
Page 1	(2)	1 and 3 are
	MAIN-ROM	switched
8000H	-----	under certain
Page 2	(6)	circumstances
	RAM	RAM
C000H	-----	-----
Page 3	(6)	(7)

```

|   RAM   |           |   RAM   |
-----

```

Note: Four pages (4 to 7) of 64K RAM are not always in the same slot.

Figure 1.5 MAIN-ROM memory map

```

0000H  -----
|  BIOS  |
|  Entry  |
015CH  |-----|
| Additional |
| BIOS Entry |
017AH  |-----|
| Empty   |
01B6H  |-----|
| BASIC   |
| Interpreter|
7FFDH  |-----|
| BDOS    |
7FFFH  | Entry |
8000H  |-----|
|         |

```

Figure 1.6 SUB-ROM memory map

```

0000h  -----
|  BIOS  |
|  Entry  |
01FDH  |-----|
| SLOTS  |
| Management |
| Control  |
0336H  |-----|
| BASIC   |
| Interpreter|
3FFFH  | and BIOS |
-----

```

Figure 1.7 MAIN-RAM memory map

```

0000H  -----
|         |
| RAM Disk |
| Area    |
|         |
8000H  |-----|
|         |
| User    |
| Area    |
|         |
F380H  |-----|
| System  |

```

```

    | Work Area |
FD9AH  |-----|
    | RAM Hook |
    | Area   |
FFCAH  |-----|
    | Expanded |
    | BIOS call |
    | Entry   |
FFCFH  |-----|
    | Interrupt |
    | Control |--> Note: Used for the disk
    | Hook Area |      and RS-232 interface
FFD9H  |-----|
    | Interrupt |
    | Control |--> Note: Used for the RS-232
    | Program |      interface
    | Area   |
FFE7H  |-----|
    | New VDP |
    | Register |
    | Subroutine|
    | Area   |
FFF7H  |-----|
    | Main ROM |
    | Slot   |
    | Address |
FFF8H  |-----|
    | Reserved |
FFFCH  |-----|
    | Slot   |
    | Selection |
FFFFH  | Register |
    -----

```

## 2.1.2 Interfacing with peripherals

MSX2 interfacing with peripherals is standardised in detail.

The following is a list of standardised interfaces:

- Display interface
- Audio interface
- Cassette interface
- General-purpose input/output interface
- Printer interface

The printer interface was optional on the MSX1 but is standard on the MSX2.

The disk drive interface is still an option but may be considered part of the standard specification because the MSX2 has 64K bytes of RAM.

For detailed information about the cartridge specifications, see the appendix.

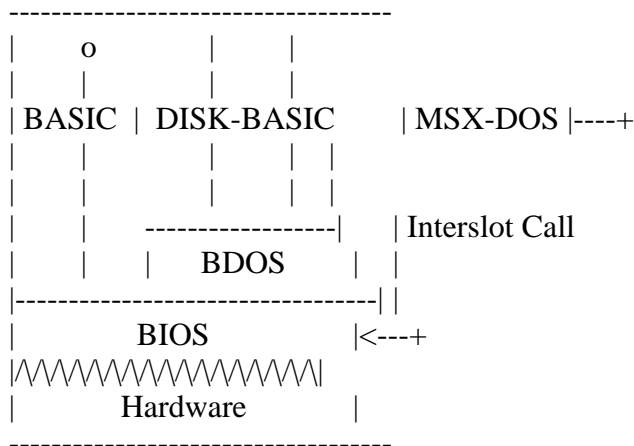
## 2.2 Software Overview

The MSX has two software environments: BASIC mode and DOS mode. BASIC mode enables easy development and execution of MSX-BASIC program and is the mode most often used by most users. A major reason why the use of personal computers has grown is that BASIC is easy to use.

The DOS mode enables various languages, utilities, and applications using MSX-DOS. Most programs in DOS can be executed on different machines. The computers automatically compensate for any differences in hardware. This allows the user to use accumulated software resources efficiently. MSX-DOS uses the same disk format as MS-DOS, which is popular on 16-bit machines. You should also note that software for CP/M, which has a great deal of applications available for 8-bit machines, can be executed only by doing file conversions.

A remarkable point is that BASIC and DOS use the same disk format in the MSX machines. This enables the sharing of resources. Both are, as shown in figure 1.8, on the united software environment which has BIOS (Basic I/O System) as a common basis. BDOS (Basic Disk Operating System), which is the basis of the disk operation, is also constructed on this BIOS. MSX offers the same programming environment to BASIC and DOS through common BDOS and BIOS.

Figure 1.8 Software hierarchy of MSX1 and MSX2



## MSX2 TECHNICAL HANDBOOK

Edited by: ASCII Systems Division  
 Published by: ASCII Corporation - JAPAN  
 First edition: March 1987

Text files typed by: Nestor Soriano (Konami Man) - SPAIN  
 March 1997

### Changes from the original:

- In description of REM statement, [<comment>] field has been added.
- In description of SGN function, "Examines the sign and returns..." has been substituted for "Examines the sign of <expression> and returns..."
- Descriptions for MSX DISK-BASIC statements DSKI\$ and DSKO\$ have been added.

- Descriptions for new commands on MSX DISK-BASIC version 2 have been added.
- In Table 2.20 (List of intermediate codes), the code "FC" is shown as assigned to "\" as it is actually, and not to "\$" as in the original text.
- In List 2.3 (Changing error handling routine), the third line of "command initialize", which is "LD HL,CMDHDAT" in the original, is corrected and substituted by "LD HL,HDAT".
- In section 5, "Notes on Software Development", subsection "BASIC version number", the part "and so on" has been added in point 1.
- In error code list, description of errors 72 to 75 have been added.

-----

## CHAPTER 2 - BASIC

The BASIC of MSX2 has been upgraded: the new version is called MSX BASIC version 2.0. And, when using a disk system, MSX DISK-BASIC can be used, which consists of MSX BASIC version 2.0 and additional instructions for disk operations. The following sections describe these two versions of BASIC.

### 1. LIST OF INSTRUCTIONS

First of all, the sentence and function for each instruction of BASIC are listed. Each instruction is listed in the format shown in Figure 2.1.

Figure 2.1 Instruction list format

Instruction format	
Instruction type	Function or action of instruction

#### (a) Syntax of instructions

If there is an "\*" followed by a keyword, it indicates that the syntax or function of the instruction has just been modified after version 1.0, or that the instruction has been added to version 2.0.

Descriptions of sentences use the following notational conventions.

- \* [item] ..... the item is optional
- \* [, item ... ] ..... more items having the same form may appear
- \* [item1 | item2] ..... choose item1 or item2

And <filename>, which is used in the sentence, is a string specifying I/O devices or files for input/output in the format listed below. <Filename> for a cassette files is a string consisting of any combination of up to 6 characters. <filename> for disk or RAM disk is a string, whose form is "<filename (up to 8 characters)> + <filename extension (up to 3

characters)>". <drive> is one of characters from A to H (depending on the number of drives connected).

"CAS: <filename>" ..... Cassette file  
"MEM: <filename>" ..... RAM disk  
"CRT:" ..... Text screen  
"GRP:" ..... Graphic screen  
"LPT:" ..... Printer  
"<drive>:<filename>" .. Disk file

## (b) Instruction type

There are four types of instructions:

- \* Function ..... Returns a certain value depending on the given parameter(s).
- \* System variable ..... Variables available from BASIC. Generally, assignment is allowed.
- \* Statement ..... Takes a certain action.
- \* Command ..... Gives an instruction to BASIC interpreter itself.

## (c) Function or action of instruction

The following list gives a brief description of the action for each instruction. More detailed descriptions about instructions which have been modified or added at version 2.0 are given in section 2.

### 1.1 Instructions of MSX BASIC version 2.0

--- A ---

**ABS (<expression>)**

Function Returns absolute value of <expression>.

**ASC (<string>)**

Function Returns the code of the first character of <string>.

**ATN (<expression>)**

Function Returns arc tangent of <expression> in radians.

**AUTO [<linenumber>[, <increment>]]**

Command Produces line numbers automatically.

--- B ---

\* **BASE (<expression>)**

System variable Contains the table address of the screen assigned on VRAM.

**BEEP**

Statement Produces beep to the audio terminal.

**BIN\$ (<expression>)**

**Function** Converts the value of <expression> to a string of binary expression, then returns its result.

**BLOAD "<filename>"[,R[,offset]]**

**Command** Loads an assembly language program.

**BSAVE "<filename>",<start address>,<end address>[,<execution address>]**

**Command** Saves an assembly language program.

--- C ---

**CALL <extended statement name>[(<argument>[,<argument>...])]**

**Statement** Calls the extended statements by inserting various cartridges.

**\* CALL MEMINI [(<upper limitation of RAM disk>)]**

**Statement** Specifies the upper limit of memory for Ram disk.

**\* CALL MFILES**

**Statement** Lists file names in RAM disk.

**\* CALL MKILL ("<filename>")**

**Statement** Deletes a file in RAM disk.

**\* CALL MNAME ("<old filename>" AS "<new filename>")**

**Statement** Renames a file in RAM disk.

**CDBL (<expression>)**

**Function** Converts the value of <expression> to a double precision real value and returns its result.

**CHR\$ (<expression>)** Returns a character which has the code of

**Function** <expression> value.

**CINT (<expression>)**

**Function** Converts the value of <expression> to an integer value and returns its result.

**\* CIRCLE {(X,Y) | STEP(X,Y)},<radius>[, <colour>[, <start angle>[, <end angle>[, <proportion>]]]]**

**Statement** Draws a circle whose center is at (X,Y) and whose size depends on <radius>.

**CLEAR [<size of string area>[, <upper limitation of memory>]]**

**Statement** Initialises variables and sets the size of memory area.

**CLOAD ["<filename>"]**

**Command** Loads a program from cassette.

**CLOAD? ["<filename>"]**

**Command** Compares a program on cassette with the one in memory.

**CLOSE [[#]<filename>[, [#]<filename>...]]**



Command Closes a file represented by <filename>.

## CLS

Statement Clears screen.

\* COLOR [<foreground colour>[, <background colour>[, <border colour>]]]

Statement Specifies the colours of each part of the screen.

\* COLOR [=NEW]

Statement Initialises the palette.

\* COLOR = (<palette number>, <red brightness>, <green brightness>, <blue brightness>)

Statement Sets the palette colour.

\* COLOR = RESTORE

Statement Puts the contents of the colour palette storage table into the palette register.

\* COLOR SPRITE (<sprite plane number>)=<colour>

Statement Sets the colour to the sprite of <sprite plane number> to the specified colour.

\* COLOR SPRITE\$ (<sprite plane number>)=<string expression>

Statement Sets the colour of each horizontal line of the sprite using <string expression>.

## CONT

Command Resumes the execution of the program which has been stopped.

\* COPY <source> TO <destination>

Statement Transfers the screen data among the screen, array, and disk file.

\* COPY SCREEN [<mode>]

Statement Writes colour bus data into VRAM (optional).

COS (<expression>)

Function Returns the cosine value of <expression> (in radians).

CSAVE "<filename>"[, <baud rate>]

Command Saves a program to cassette.

CSGN (<expression>)

Function Converts the value of <expression> to a single precision real value, and returns its result.

## CSRLIN

System variable Contains the vertical screen location of the cursor. No assignment is allowed.

--- D ---

DATA <constant>[, <constant>...]

Statement      Prepares data to be read by READ statement.

DEF FN <name> [(<argument>[, <argument>...])]=<function-definitive  
expression>

Statement      Defines a user-defined function.

DEFINT <character range>[, <character range>...]

Statement      Declares the specified variable(s) as integer type.

DEFSNG <character range>[, <character range>...]

Statement      Declares the specified variable(s) as single  
precision real type.

DEFDBL <character range>[, <character range>...]

Statement      Declares the specified variable(s) as double  
precision real type.

DEFSTR <character range>[, <character range>...]

Statement      Declares the specified variable(s) as character type.

DEF USR [<number>]=<start address>

Statement      Defines the starting address for the execution of  
assembly language routine, called by USR function.

DELETE {[<start linenumber>-<end linenumber>] | <linenumber> | -<end  
linenumber>}

Command          Deletes the specified portion of the program.

DIM <variable name> (<maximum subscript value>[, <maximum subscript  
value>...])

Statement      Defines an array variable and allocates it into  
memory.

DRAW <string expression>

Statement      Draws a line or lines on the screen according to  
<string expression (DRAW macro)>.

--- E ---

END

Statement      Ens the program, close all files, and returns to  
the command level.

EOF (<filenumber>)

Function      Checks if the file is finished and returns -1 if at  
the end of file.

ERASE <array variable name>[, <array variable name>...]

Statement      Deletes the array variable(s).

ERL

System variable    Contains the error code for the preceding error.  
No assignment is allowed.

ERR

System variable    Contains the line number of the previous error.

No assignment is allowed.

**ERROR** <error code>

Statement Puts the program into the error condition.

**EXP** (<expression>)

Function Returns the exponent (power) of the natural exponential form of <expression>.

--- F ---

**FIX** (<expression>)

Function Returns the value of <expression>, without any decimal fractions.

**FOR** <variable name> = <initial value> TO <end value> [STEP <increment>]

Statement Repeats the execution from FOR statement to NEXT statement for the specified times.

**FRE** ({<expression> | <string expression>})

Function Returns the size of unused user's area or unused character area.

--- G ---

\* **GET DATE** <string variable name>[, A]

Statement Assigns date into a string variable.

\* **GET TIME** <string variable name>[, A]

Statement Assigns time into a string variable.

**GOSUB** <linenumber>

Statement Calls the subroutine at <linenumber>.

**GOTO** <linenumber>

Statement Jumps to <linenumber>.

--- H ---

**HEX\$** (<expression>)

Function Converts the value of <expression> to a string of hexadecimal expression, then returns its result.

--- I ---

**IF** <condition> THEN {<statement> | <linenumber>} [ELSE {<statement> | <linenumber>}]

Statement Judges the condition. If <condition> is not zero, it is true.

**IF** <condition> GOTO <linenumber> [ELSE {<statement> | <linenumber>}]

Statement Judges the condition. If <condition> is not zero, it is true.

## INKEY\$

Function Returns a character when a key is being pressed, or when not, returns null string.

## INP (<port number>)

Function Reads the port specified by <port number> and returns its result.

## INPUT ["<prompt statement>";]<variable name>[, <variable name>...]

Statement Assigns data input from keyboard into the specified variable(s).

## INPUT #<filename>, <variable name>[, <variable name>...]

Statement Reads data from the file and assigns the data into the specified variable(s).

## INPUT\$ (<number of characters>[, [#]<filename>])

Function Reads the specified size of string from the keyboard or file.

## INSTR ([<expression>,<string expression 1>,<string expression 2>)

Function Searches <string expression 2> from the left of <string expression 1>, and returns its location if found, otherwise zero. <Expression> is the character location to start searching.

## INT (<expression>)

Function Returns the largest integer less than <expression>.

## INTERVAL {ON | OFF | STOP}

Statement Allows, suppresses, or suspends the timer interrupt.

--- K ---

## KEY <key number>,<string>

Command Redefines a function key.

## KEY LIST

Command Displays the contents of function keys.

## KEY (<key number>){ON | OFF | STOP}

Statement Allows, suppresses, or suspends the function key interrupt.

## KEY {ON | OFF}

Statement Specifies whether to display the contents of function keys at the bottom of the screen.

--- L ---

## LEFT\$ (<string expression>,<expression>)

function Gets <expression> characters from the left of <string expression>.

## LEN (<string expression>)

Function Returns the number of characters of <string

expression>.

[LET] <variable name> = <expression>

Statement Assigns the value of <expression> to the variable.

\* LINE [{(X1,Y1) | STEP(X1,Y1)}] - {(X2,Y2) | STEP(X2,Y2)}[, <colour>  
[, {B|BF}], <logical operation>]]]

Statement Draws a line or a rectangle on the screen.

LINE INPUT ["<prompt statement>";]<string variable name>

Statement Assigns a whole line of string data from the keyboard into the string variable.

LINE INPUT# <filename>, <string variable name>

Statement Reads data in lines from the file and assigns the data into the string variable.

LIST [[<linenumber>] - [<linenumber>]]

Command Displays the program in memory on the screen.

LLIST [[<linenumber>] - [<linenumber>]]

Command Sends the program in memory to the printer.

LOAD "<filename>" [,R]

Command Loads a program saved in ASCII format.

\* LOCATE [<X-coordinate>[, <Y-coordinate>[, <cursor switch>]]]

Statement Locates the cursor on the text screen.

LOG (<expression>)

Function Returns the natural logarithm of <expression>.

LPOS (<expression>)

System variable Contains the location of the printer head.  
No assignment is allowed.

LPRINT [<expression>[{: | ,}<expression>]...]

Statement Outputs characters or numerical values to the printer.

LPRINT USING <form>; <expression>[{: | ,}<expression>]...]

Statement Outputs characters or numerical values through the printer according to <form>.

--- M ---

MAXFILES = <number of files>

Statement Sets the number of files to be opened.

MERGE "<filename>"

Command Merges the program in memory with the program saved in ASCII format (in external storage device).

MID\$ (<string expression>, <expression 1>[, <expression 2>])

Function Returns <expression 2> character(s) starting from the <expression 1>th position of <string expression>.

MID\$ (<string variable name>, <expression 1>[, <expression 2>])  
= <string expression>  
Statement Defines <string expression> using <expression 2>  
character(s) from the <expression 1>th position  
of <string variable name>.

MOTOR [{ON | OFF}]  
Statement Turns the motor of cassette ON and OFF.

--- N ---

NEW  
Command Deletes the program in memory and clears variables.

NEXT [<variable name>[, <variable name>...]]  
Statement Indicates the end of FOR statement.

--- O ---

OCT\$ (<expression>)  
Function Converts the value of <expression> to the string of  
octal expression and returns its result.

ON ERROR GOTO <linenumber>  
Statement Defines the line to begin the error handling routine.

ON <expression> GOSUB <linenumber>[, <linenumber>...]  
Statement Executes the subroutine at <linenumber> according to  
<expression>.

ON <expression> GOTO <linenumber>[, <linenumber>...]  
Statement Jumps to <linenumber> according to <expression>.

ON INTERVAL = <time> GOSUB <linenumber>  
Statement Defines the timer interrupt interval and the line to  
begin the interrupt handling routine.

ON KEY GOSUB <linenumber>[, <linenumber>...]  
Statement Defines the line to begin the function key interrupt  
handling routine.

ON SPRITE GOSUB <linenumber>  
Statement Defines the line to begin the piled-sprite interrupt  
handling routine.

ON STOP GOSUB <linenumber>  
Statement Defines the line to begin the CTRL+STOP key interrupt  
handling routine.

ON STRING GOSUB <linenumber>[, <linenumber>...]  
Statement Defines the line to begin the trigger button  
interrupt handling routine.

OPEN "<filename>" [FOR <mode>] AS #<filenumber>  
Statement Opens the file in the specified mode.

OUT <port number>,<expression>

Statement Sends data to the output port specified by <port number>.

--- P ---

\* PAD (<expression>)

Function Examines the state of tablet, mouse, light pen, or track ball specified by <expression>, then returns its result.

\* PAINT {(X,Y) | STEP(X,Y)}[, <colour>[, <border colour>]]

Statement Paints the area surrounded by specified <border colour> using <colour>.

PDL (<paddle number>)

Function Returns the state of the paddle which has the specified number.

PEEK (<address>)

Function Returns the contents of one byte of the memory specified by <address>.

PLAY <string expression 1>[, <string expression 2>[, <string expression 3>]]

Statement Plays the music by <string expression (music macro)>.

PLAY (<voice channel>)

Function Examines whether the music is being played and returns its result (if in play, -1 is returned).

POINT (X,Y)

Function Returns the colour of the dot specified by coordinate (X,Y).

POKE <address>,<data>

Statement Writes one byte of <data> into the memory specified by <address>.

POS (<expression>)

System variable Contains the horizontal location of the cursor on the text screen. No assignment is allowed.

\* PRESET {(X,Y) | STEP(X,Y)}[, <colour>[, <logical operation>]]

Statement Erases the dot specified by coordinate (X,Y) on the graphic screen

PRINT [<expression> [{; | ,}<expression>...]

Statement Displays characters of numbers on the screen.

PRINT USING <form>; <expression> [{; | ,}<expression>...]

Statement Displays characters or numbers on the screen according to <form>.

PRINT #<filename>, [<expression> [{; | ,}<expression>...]

Statement Writes characters or numbers to the file specified by <file number>.

PRINT #<filenumber>, USING <form>; <expression>[{:|,}<expression>...]

Statement Writes characters or numbers to the file specified by <file number> according to <form>.

PSET {(X,Y) | STEP(X,Y)}[, <colour>[, <logical operation>]]

Statement Draws the dot in the coordinate specified by (X,Y) on the graphic screen.

\* PUT KANJI [(X,Y),<JIS kanji code>[, <colour>[, <logical operation>[, <mode>]]]]

Statement Displays the kanji on the screen (KANJI ROM is required).

\* PUT SPRITE <sprite plane number>[, {(X,Y) | STEP(X,Y)}[, <colour>[, <sprite pattern number>]]]

Statement Displays the sprite pattern.

--- R ---

READ <variable name>[, <variable name>...]

Statement Reads data from DATA statement(s) and assigns the data to the variable(s).

REM [<comment>]

Statement Puts the comment in the program.

RENUM [<new linenumber>[, <old linenumber>[, <increment>]]]

Command Renumbers the line numbers.

RESTORE [<linenumber>]

Statement Specifies the line to begin reading DATA by READ statement.

RESUME {[0] | NEXT | <linenumber>}

Statement Ends the error recovery routine and resumes execution of the program.

RETURN [<linenumber>]

Statement Returns from a subroutine.

RIGHT\$ (<string expression>, <expression>)

Function Gets <expression> characters from the right of <string expression>.

RND [(<expression>)]

Function Returns a random number between 0 and 1.

RUN [<linenumber>]

Command Executes the program from <linenumber>.

--- S ---

SAVE "<filename>"

Command Saves the program in ASCII format.



\* SCREEN <screen mode>[, <sprite size>[, <key click switch>[, <cassette baud rate>[, <printer option>[, <interlace mode>]]]]

Statement Sets the screen mode and so on.

\* SET ADJUST (<X-coordinate offset>, <Y-coordinate offset>)  
statement Changes the display location of the screen. Ranges from -7 to 8.

\* SET BEEP <timbre>, <volume>  
Statement Selects the BEEP tone. Ranges from 1 to 4.

\* SET DATE <string expression>[, A]  
Statement Sets a date. "A" is the specification of alarm.

\* SET PAGE <display page>, <active page>  
Statement Specifies the page to display and the page to read and write data to.

\* SET PASSWORD <string expression>  
Statement Sets a password.

\* SET PROMPT <string expression>  
Statement Sets a prompt (up to 8 characters).

\* SET SCREEN  
Statement Reserves the parameters of the current settings of SCREEN statement.

\* SET TIME <string expression>[, A]  
Statement Sets time. "A" is the alarm specification.

\* SET VIDEO [<mode>[, <Ym>[, <CB>[, <sync>[, <voice>[, <video input>[, <AV control>]]]]]]  
Statement Sets superimposing and other modes (optional).

SGN (<expression>)  
Function Examines the sign of <expression> and returns its result (positive=1, zero=0, negative=-1).

SIN (<expression>)  
Function Returns the sine of <expression> in radians.

SOUND <register number>,<data>  
Statement Writes data to the register of PSG.

SPACE\$ (<expression>)  
Function Returns a string containing <expression> spaces.

SPC (<expression>)  
Function Produces <expression> spaces; used in the instructions of PRINT family.

SPRITE {ON | OFF | STOP}  
Statement Allows, supresses, or suspends the piled-sprite interrupt.

SPRITE\$ (<sprite pattern number>)

System variable    Contains the sprite pattern.

SQR (<expression>)

Function        Returns the square root of <expression>.

STICK (<joystick number>)

Function        Examines the direction of the joystick and returns its result.

STOP

Statement      Stops the execution of the program.

STRIG (<joystick number>)

Function        Examines the state of the trigger button and returns its result.

STRIG (<joystick number>) {ON | OFF | STOP}

Statement      Allows, supresses, or suspends interrupts from the trigger button.

STR\$ (<expression>)

Function        Converts the value of <expression> to a string decimal expression and returns its result.

STRING\$ (<expression 1>, {<string expression> | <expression 2>})

Function        Converts the leading character of <string expression> or the character containing the code <expression 2> to a string whose length is <expression 1>, and returns the string.

SWAP <variable name>, <variable name>

Statement      Exchanges the value of two variables.

--- T ---

TAB (<expression>)

Function        Produces the specified spaces in PRINT instructions.

TAN (<expression>)

Function        Returns the tangent of <expression> in radians.

TIME

System variable    Contains the value of the interval timer.

TRON

Command        Keeps displaying the line numbers of the program currently being executed.

TROFF

Command        Cancels TRON and stops displaying the line numbers.

--- U ---

USR [<number>](<argument>)

Function        Calls the assembly language routine.

--- V ---

VAL (<string expression>)

Function Converts <string expression> to a numerical value and returns its result.

VARPTR (<variable name>)

Function Returns the address containing the variable.

VARPTR (#<filename>)

Function Returns the starting address of the file control block.

\* VDP (<register number>)

System variable Writes/reads data to/from the VDP registers.

\* VPEEK (<address>)

Function Reads data from <address> in VRAM.

\* VPOKE (<address>)

Statement Writes data to <address> in VRAM.

--- W ---

WAIT <port number>, <expression 1>[, <expression 2>]

Statement Stops the execution until data of the input port grows to the specified value.

\* WIDTH <number>

Statement Specifies the number of characters per line in the display screen.

## 1.2 Instructions of MSX DISK-BASIC

Note: Instructions marked with "\*\*\*" have been added to version 2 of MSX DISK-BASIC and are not available in version 1.

--- B ---

\* BLOAD "<filename>"[ { [, R] | [, S] } ], <offset>]

Command Loads the assembly language program or screen data from a file.

\* BSAVE "<filename>", <start address>, <end address>[, { <execution address> | S }]

Command Saves the assembly language program or screen data in a file.

--- C ---

CLOSE [[#]<filename>[, [#]<filename>...]]

Statement Closes the file specified by <filename>.

\*\* CALL CHDRV ("<drive name>:")

Command Sets the drive specified by <drive name> as the default drive.

\*\* CALL CHDIR ("<directory path>")

Command Changes to the directory specified by <directory path>.

CALL FORMAT

Command Formats the floppy disk.

\*\* CALL MKDIR ("<directory name>")

Command Creates the directory with the name specified in <directory name> in the current directory.

\*\* CALL RAMDISK (<size in kilobytes>[, <variable name>])

Command Tries to create the DOS 2 RAM disk of the specified size, and returns in the variable (if specified) the actual size of the RAM disk created.

\*\* CALL RMDIR ("<directory name>")

Command Deletes the directory specified in <directory name>. If the directory is not empty, "File already exists" error will be returned.

CALL SYSTEM

Command Returns to MSX-DOS.

\*\* CALL SYSTEM [("<filename>")]

Command Returns to MSX-DOS and executes the DOS command <filename> if it is specified.

COPY "<filename 1>"[ TO "<filename 2>"]

Command Copies the contents of <filename 1> to the file specified by <filename 2>.

CVD (<8-byte string>)

Function Converts the string to the double precision real value and returns its result.

CVI (<2-byte string>)

Function Converts the string to the integer value and returns its result.

CVS (<4-byte string>)

Function Converts the string to the single precision real value and returns its result.

--- D ---

DSKF (<drive number>)

Function Returns the unused portions of the disk in clusters.

DSKI\$ (<drive number>, <sector number>)

Function Reads the specified sector of the specified drive to the memory area indicated by address &HF351, and

returns a null string.

DSKOS (<drive number>, <sector number>)

Statement Writes 512 bytes starting from address indicated by &HF351 to the specified sector of the specified drive.

--- E ---

EOF (<filename>)

Function Checks if the file has ended and returns -1 if at the end of file.

--- F ---

FIELD [#]<filename>, <field width> AS <string variable name>[, <field width> AS <string variable name>...]

Statement Assigns the string variable name to the random input/output buffer.

FILES ["<filename>"]

Command Displays the name of the file matched with <filename> on the screen.

\*\* FILES ["<filename>"][,L]

Command Displays the name of the file matched with <filename> on the screen, and also the attributes and the size of the file if "L" is specified.

--- G ---

GET[#]<filename>[, <record number>]

Statement Reads one record from the random file to the random input/output buffer.

--- I ---

INPUT #<filename>, <variable name>[, <variable name>...]

Statement Reads data from the file.

INPUT\$ (<the number of characters>[, [#]<filename>])

Function Gets the string of the specified length from the file.

--- K ---

KILL "<filename>"

Command Deletes the file specified by <filename>.

--- L ---

LFILES ["<filename>"]

Command           Sends the name of the file matched with <filename> to the printer.

\*\* LFILES ["<filename>"][,L]

Command           Sends the name of the file matched with <filename> to the printer, and also the attributes and the size of the file if "L" is specified.

LINE INPUT #<file number>, <string variable name>

Statement        Reads lines of data from the file to the string variable.

LOAD "<filename>"[, R]

Command           Loads the program into memory.

LOC (<file number>)

Function         Returns the record number of the most recently accessed location of the file.

LOF (<file number>)

Function         Returns the size of the specified file in bytes.

LSET <string variable name>=<string expression>

Statement        Stores data padded on the left in the random input/output buffer.

--- M ---

MAXFILES = <the number of files>

Statement        Declares the maximum number of files that can be opened.

MERGE "<filename>"

Command           Merges the program in memory with the program saved in ASCII format.

MKD\$ (<double precision real value>)

Function         Converts the double precision real value to the character code corresponding to the internal expression.

MKI\$ (<integer value>)

Function         Converts the integer value to the character code corresponding to the internal expression.

MKS\$ (<single precision real value>)

Function         Converts the single precision real value to the character code corresponding to the internal expression.

--- N ---

NAME "<filename 1>" AS "<filename 2>"

Command           Renames the name of a file.

--- O ---

OPEN "<filename>"[FOR <mode>] AS #<filenumber>[LEN = <record length>]  
Statement Opens the file.

--- P ---

PRINT #<filenumber>, [<expression>[; | ,]<expression>...]  
Statement Sends data to the sequential file.

PRINT #<filenumber>, USING <form>; <expression>[; | ,]<expression>...]  
Statement Sends data to the sequential file according to the form.

PUT [#]<filenumber>[, <record number>]  
Statement Sends data of the random input/output buffer to the random file.

--- R ---

RSET <string varibale name>=<string expression>  
Statement Stores data padded on the right in the random input/output buffer.

RUN "<filename>"[, R]  
Command Loads a program from the disk and executes it.

--- S ---

SAVE "<filename>"[, A]  
Command Saves a program. The program is saved in ASCII format when "A" is specified.

--- V ---

VARPTR (#<filenumber>)  
Function Returns the starting address of the file control block.

## 2. DIFFERENCES IN MSX BASIC VERSION 2.0

A great deal of functions in MSX BASIC version 2.0 have been added or modified when compared with MSX BASIC version 1.0. They are either the functions that are added or modified with the version-up of VDP (Video Display Processor) or the functions that are added or modified because of the various hardware features such as RAM disk, clock, or memory switch; especially, the alternation of VDP affects, most of the statement for the screen display.

This section picks up these statements and indicates the additions or the modifications. In the following descriptions, "MSX1" means MSX BASIC version

1.0 and "MSX2" for MSX BASIC version 2.0.

## 2.1 Additions or Modifications to Screen Mode

\* SCREEN <screen mode>[, <sprite size>[, <key click switch>[, <cassette baud rate>[, <printer option>[, <interlace mode>]]]]

<Screen mode> and <interlace mode> have been modified.

<Screen mode> may be specified from 0 to 8. Modes from 0 to 3 are the same as MSX1 and the rest have been added. When specifying a screen mode, in BASIC it is called "SCREEN MODE", which is somewhat different from "screen mode" which is used by VDP internally. Table 2.1 shows these correspondences and meanings. The difference between screen modes 2 and 4 is only in the sprite display functions.

Table 2.1 Correspondances of BASIC screen (SCREEN) modes and VDP screen modess

BASIC mode	VDP mode	Meaning	Screen
	characters	Dots or at a time	Display colours format
SCREEN 0 (1)	TEXT 1	40 x 24 chars	2 from 512   Text
SCREEN 0 (2)	TEXT 2	80 x 24 chars	2 from 512   Text
SCREEN 1	GRAPHIC 1	32 x 24 chars	16 from 512   Text
SCREEN 2	GRAPHIC 2	256 x 192 dots graphics	16 from 512   High res.
SCREEN 3	MULTICOLOUR	64 x 48 dots graphics	16 from 512   Low res.
SCREEN 4	GRAPHIC 3	256 x 192 dots graphics	16 from 512   High res.
SCREEN 5	GRAPHIC 4	256 x 212 dots graphics	16 from 512   Bit map
SCREEN 6	GRAPHIC 5	512 x 212 dots graphics	4 from 512   Bit map
SCREEN 7	GRAPHIC 6	512 x 212 dots graphics	16 from 512   Bit map
SCREEN 8	GRAPHIC 7	256 x 212 dots graphics	256   Bit map

Specifying <interlace mode> enables to set the interlace functions of VDP



(see Table 2.2). In the alternate screen display mode, the display page specified in "SET PAGE" must be odd. In this case the display page and the page of which the number is smaller by one is displayed alternately.

Table 2.2 Differences of display function in the interlace mode

Interlace mode	Display function
0	Normal non-interlaced display (default)
1	Interlaced display
2	Non interlaced, Even/Odd alternate display
3	Interlaced, Even/Odd alternate display

\* SET PAGE <display page>, <active page>

This statement is new. It allows users to set the page to display and the page to read and write data to. This is valid when the screen mode is between 5 and 8, and the value specified depends on the VRAM capacity and the screen mode (see Table 2.3).

Table 2.3 Page values to be specified depending on the screen mode and the VRAM capacity

Screen mode	VRAM 64K	VRAM 128K
SCREEN 5	0 to 1	0 to 3
SCREEN 6	0 to 1	0 to 3
SCREEN 7	Unusable	0 to 1
SCREEN 8	Unusable	0 to 1

See the VRAM map in the APPENDIX for the page assignment on VRAM.

## 2.2 Additions or Modifications for the Colour Specification

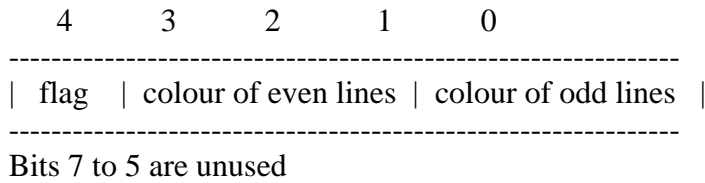
\* COLOR [<foreground colour>[, <background colour>[, <border colour>]]]

In MSX2, with its colour palette feature, the ranges and meanings of values specifying colours in the screen mode are different (see Table 2.4). The <background colour> except that of the text display changes when the CLS statement is executed. If the display mode is 0, specification of a <border colour> is ignored.

The "border colour" in screen mode 6 has special meanings. Figure 2.2 shows the bitwise meanings of <border colour> in the mode. In this mode, by changing the flag (bit 4), the colour of vertical lines at odd X-coordinates and the colour of those at even coordinates can be specified differently.

When the flag is 0 (the value of border colour is one of the values from 0 to 15), different colours cannot be specified and the border colour is set as the colour of vertical odd lines. When the flag is 1 (the value of border colour is one of the values from 16 to 31), the border colours are set as the colour of vertical odd lines and that of vertical even lines; when these two colours are different, the screen shows a vertically-striped pattern.

Figure 2.2 Bitwise meanings for the border colour on screen mode 6



\* COLOR = (<palette number>, <red brightness>, <green brightness>, <blue brightness>)

This statement sets the colour of the specified palette. See Table 2.4 for the specification of <palette number>. Note that nothing happens and no error occurs when the screen mode is 8, which has no palette feature. Though palette number 0 is ordinarily fixed to a transparent colour (that is, border space is seen transparently), it can be dealt in the same way as other palettes by changing the register of VDP:

```

VDP(9)=VDP(9) OR &H20    (when dealing as with other palettes)
VDP(9)=VDP(9) AND &HDF  (when fixing it to a transparent colour)

```

Table 2.4 Colour specifications for the screen mode.

Screen mode	Colour specification	Range of number
SCREEN 0	Palette number	0 to 15
SCREEN 1	Palette number	0 to 15
SCREEN 2	Palette number	0 to 15
SCREEN 3	Palette number	0 to 15
SCREEN 4	Palette number	0 to 15
SCREEN 5	Palette number	0 to 15
SCREEN 6	Palette number	0 to 3
SCREEN 7	Palette number	0 to 15
SCREEN 8	Colour number	0 to 255

Brightness of each colour can be set to one of eight steps from 0 to 7 and combining them enables to display 512 colours; 8 (red) x 8 (green) x 8 (blue).

\* COLOR=RESTORE

This statement resets the colour palette register according to the contents of the colour palette storage table (see APPENDIX VRAM MAP). For example, if image data written under unusual colour palette settings is BSAVED, the

original images cannot be reproduced because BLOADing the data does not change the colour palettes. Therefore, the image data should be BSAVED with the colour palette storage table. To obtain the colours of the original images, BLOAD the data and reset the palettes with the COLOR=RESTORE instruction.

\* COLOR [=NEW]

This statement initialises the colour palette to the same state as when the power of the computer is turned on (see Table 2.5). It is a good idea to place this statement at the beginning and the end of the program.

Table 2.5 Initial colours of colour palettes and palette setting values

Palette number	Colour	Brightness of red	Brightness of blue	Brightness of green
0	transparent	0	0	0
1	black	0	0	0
2	bright green	1	1	6
3	light green	3	3	7
4	deep blue	1	7	1
5	bright blue	2	7	3
6	deep red	5	1	1
7	light blue	2	7	6
8	bright red	7	1	1
9	light red	7	3	3
10	bright yellow	6	1	6
11	pale 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

### 2.3 Additions or Modifications for the Character Display

\* LOCATE [<X-coordinate>[, <Y-coordinate>[, <cursor switch>]]]

This statement specifies the location to display a character in the text display screen.

Since an 80-character display feature has been added to the screen mode 0, the X-coordinate value can be specified up to 79.

### 2.4 Additions or Modifications for the Graphics Display

\* LINE [{(X1,Y1) | STEP(X1,Y1)} - {(X2,Y2) | STEP(X2,Y2)}[, <colour>[, {B|BF}[, <logical operation>]]]

\* PSET {(X,Y) | STEP(X,Y)}[, <colour>[, <logical operation>]]

\* PRESET {(X,Y) | STEP(X,Y)}[, <colour>[, <logical operation>]]

The specifiable coordinate range of these statements varies according to the screen mode (see Table 2.6).

Table 2.6 Range of coordinates for each screen mode

Screen mode	X-coordinate	Y-coordinate
SCREEN 2	0 to 255	0 to 191
SCREEN 3	0 to 255	0 to 191
SCREEN 4	0 to 255	0 to 191
SCREEN 5	0 to 255	0 to 211
SCREEN 6	0 to 511	0 to 211
SCREEN 7	0 to 511	0 to 211
SCREEN 8	0 to 255	0 to 211

The logical operation feature is new. When <logical operation> is specified, a logical operation is done between the specified <colour> and the original colour, and the colour of its result will be used to draw. Logical operation types are listed in Table 2.7. <Colour> is specified by the palette number, except for screen mode 8.

Table 2.7 Logical operation

Logical operation	Function to draw
PSET (default) ,TPSET	Use "specified colour"
PRESET ,TPRESET	Use "NOT (specified colour)"
XOR ,TXOR	Use "(background colour) XOR (specified colour)"
OR ,TOR	Use "(background colour) OR (specified colour)"
AND ,AND	Use "(background colour) AND (specified colour)"

Note: The list above assumes that <colour> is (specified colour) and that the original colour of the place to be drawn is (background colour). Specifying a logical operation preceded by "T" causes nothing to be done when <colour> is transparent (colour 0).

\* CIRCLE {(X,Y) | STEP(X,Y)},<radius>[, <colour>[, <start angle>[, <end angle>[, <proportion>]]]]

The coordinate range to be specified depends on the screen mode (see Table 2.6). <colour> is specified by the palette number, except for screen mode 8.

\* PAINT {(X,Y) | STEP(X,Y)}[, <colour>[, <border colour>]]

The coordinate range to be specified depends on the screen mode (see Table 2.6). <Colour> is specified by the palette number, except for screen mode 8.

The specification of <border color> is invalid in screen modes 2 and 4.

## 2.5 Additions or modifications for VDP access

### \* BASE (<expression>)

This system variable contains the starting address of each table assigned to VRAM. The contents of <expression> and the screen mode tables correspond as listed in Table 2.8.

The starting address of the table can be read for each <expression>, but can be written only when <expression> is a value from 0 to 19 (that is, from screen mode 0 to screen mode 3).

Note that the table of screen mode 4 changes as you change the table address of screen mode 2.

Address returned for screen mode from 5 to 8 is the offset value from the starting address of the active page.

Table 2.8 Correspondences between BASE set values and VRAM table

Expression	Screen mode	Table
0	0	Pattern name table
1	0	N/A
2	0	Pattern generator table
3	0	N/A
4	0	N/A
5	1	Pattern name table
6	1	Colour table
7	1	Pattern generator table
8	1	Sprite attribute table
9	1	Sprite generator table
10	2	Pattern name table
11	2	Colour table
12	2	Pattern generator table
.	.	.
.	.	.
.	.	.
43	8	Sprite attribute table
44	8	Sprite generator table

### \* VDP (<n>)

This allows the value of VDP register to be read and written. <n> is slightly different from the actual VDP register number. Their correspondances are listed in Table 2.9.

Table 2.9 Correspondances with VDP register

n	VDP register number	Access mode
0 to 7	0 to 7 (same as MSX1)	Read/write
8	Status register 0	Read only
9 to 24	8 to 23	Read/write
33 to 47	32 to 46	Write only
-1 to -9	Status register 1 to 9	Read only

- \* VPEEK (<address>)
- \* VPOKE <address>, <data>

When the screen mode is from 5 to 8, the offset value from the starting address of the active page should be set for <address>. Valid range for the <address> value is from 0 to 65535 and the valid range for the data value is from 0 to 255.

- \* BSAVE <filename>, <start address>, <end address>, S
- \* BLOAD <filename>, S

These are statements of DISK BASIC, used to save/load the contents of VRAM to/from disk files. Both can be used in any screen mode, note, however, that only the active pages are valid when the screen mode is from 5 to 8. No cassette tapes can be used. Valid value range of <address> is from -32768 to -2, or from 0 to 65534 (&HFFFE).

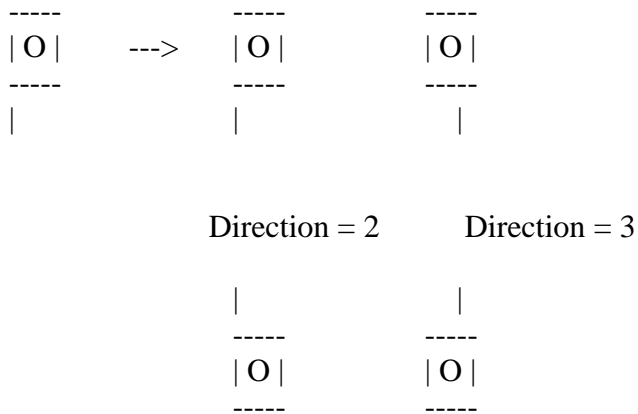
- \* COPY (X1,Y1) - (X2,Y2)[, <source page>] TO (X3,Y3)[, <destination page>][, <logical operation>]]
- \* COPY (X1,Y1) - (X2,Y2)[, <source page>] TO {<array variable name> | <filename>}
- \* COPY {<array variable name> | <filename>}[, <direction>] TO (X3,Y3)[, <destination page>[, <logical operation>]]
- \* COPY <filename> TO <array variable name>
- \* COPY <array variable name> TO <filename>

The COPY statements transfer screen data and are valid when the screen mode is from 5 to 8. VRAM, array variables, and disk files can be used with these statements, and data can be transferred among these at will.

(X1,Y1) - (X2,Y2) means that the rectangular area, with a diagonal formed by these two coordinates is to be transferred. <Source page> and <destination page> indicate the page to be transferred from and the page to be transferred to, respectively, and if these pages are omitted, the active pages are assumed. <Direction> indicates the direction for writing the screen data to the screen, and is specified by a number from 0 to 3 (see Figure 2.3).

Figure 2.3 Directions for writing the screen data

Original Screen      Direction = 0      Direction = 1



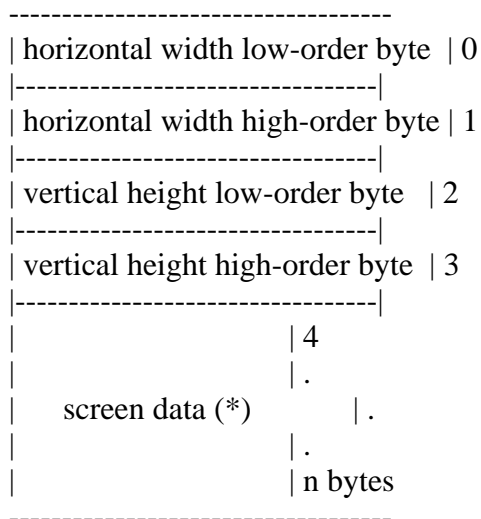
<Array variable> is of the integer type, or single precision real type, or double precision real type. It should be prepared with enough area to get the screen data. Its size can be calculated by expression 1 as shown below.

<Pixel size> is the number of bits to be used to express one dot on the screen. It is 4 when the screen mode is 5 or 7, 2 for mode 6, and 8 for mode 8. Screen data is stored in the format shown in figure 2.4.

Expression 1

$$\text{INT} ((\langle\text{pixel size}\rangle * (\text{ABS}(\text{X2}-\text{X1})+1) * (\text{ABS}(\text{Y2}-\text{Y1})+1)+7)/8)+4 \text{ bytes}$$

Figure 2.4 Screen data format



(\*) If the length of data cannot be divided by byte, excess bits are to be 0.

<Logical operation> specifies a logical operation between the data which resides on the destination and the data to be transferred. See table 2.7 for the parameters to specify.

When operations preceded by "T" are specified, the transparent portions of the source will not be transferred.

## 2.7 Additions or Modifications for Sprite

The sprites used in screen mode 4-8 of MSX2 are called sprite mode 2, which has upgraded a great deal as compared with MSX1. On MSX1, for example, one sprite could treat only one colour, while in this mode of MSX2 different colours can be specified for each horizontal line and so multi-coloured characters can be realised with one sprite. Additionally, it is a good idea to combine two sprites as though they were one sprite to paint each dot with different colours. And, on MSX1, when more than five sprites are arrayed on a horizontal line, the sprites after the fifth one were not displayed, but on MSX2 up to eight sprites can be displayed, so a higher flexibility is offered.

Colours which can be specified for sprites are shown in Table 2.4 (colour statement) except for screen mode 8. The sprite in screen mode 8, not capable of using the palette, uses the colour number for the specification, and only 16 colours can be used (see Table 2.10).

Table 2.10 Sprite colours in screen mode 8

0: Black	1: Deep Blue	2: Deep Red	3: Deep Purple
4: Deep Green	5: Turquoise	6: Olive	7: Grey
8: Light Orange	9: Blue	10: Red	11: Purple
12: Green	13: Light Blue	14: Yellow	15: White

\* PUT SPRITE <sprite plane number>[, {(X,Y) | STEP(X,Y)}[, <colour>[, <sprite pattern number>]]]

In screen modes 1 through 3, Y-coordinate was 209 for erasing the display of the specified sprite and was 208 for erasing the displays of the specified sprite and all sprites following it, but in screen modes 4 through 8, where the limit of Y-coordinate has been increased to 212 dots, the values to be specified are now 217 and 216, respectively.

\* COLOR SPRITE\$ (<sprite plane number>) = <string expression>

This statement specifies a colour for each horizontal line (see Figure 2.5).

<String expression> consists of one to sixteen characters. Bits 0 through 3 of the character's ASCII code are used for the colour specification, and bits 4 through 7 are used to specify each function of the sprite (see Table 2.11).

These specifications are valid only for screen modes 4 through 8.

COLOR SPRITE\$ = CHR\$ (colour of the first line) + CHR\$ (colour of the second line) + ..... + CHR\$ (colour of the eight line)

Figure 2.5 Relation of the sprite and <string expression>

-----  
Line 1 --> | \* | | | \* | \* | | \* |



```

|-----+-----+-----+-----+-----+-----+-----+-----|
Line 2 --> | * | * | | | | | * | * |
|-----+-----+-----+-----+-----+-----+-----+-----|
| | | | | | | | | |
|-----+-----+-----+-----+-----+-----+-----+-----|
| * | * | | | | | * | * |
|-----+-----+-----+-----+-----+-----+-----+-----|
| | | * | | | * | | |
|-----+-----+-----+-----+-----+-----+-----+-----|
| * | | | * | * | | | * |
|-----+-----+-----+-----+-----+-----+-----+-----|
| * | * | | | | | * | * |
|-----+-----+-----+-----+-----+-----+-----+-----|
Line 8 --> | | | * | * | * | * | | | The colour for each line
----- can be set.

```

Table 2.11 Bitwise meanings of string expression

b7	If 1, move the sprite to left by 32 dots.
b6	If 1, move the sprites of the successive planes together. The priority and conflict of sprites are ignored, and when sprites are piled up, they are displayed in the colour which is OR-ed with their colour numbers. *
b5	If 1, the conflict of sprites are ignored.
b4	Unused.
b0 to b3	Palette number.

\* For example, assuming that bit 6 of sprite plane 1 is "0" and bit 6 of sprite plane 2 is "1", only by moving sprite plane 1, will sprite plane 2 be displayed displayed to be piled at the same location.

\* COLOR SPRITE (<sprite plane number>) = <expression>

This statement sets the whole sprite of the specified plane to the <expression>, this uses <expression> for colour specification. The format of the colour specification is the same as shown in Table 2.11, but the specification for b7 is disabled. These are valid for screen modes 4 through 8.

## 2.8 Additions for Optional Features

\* SET VIDEO [<mode>[, <Ym>[, <CB>[, <sync>[, <voice>[, <video input>[, <AV control>]]]]]]]

This statement is for the superimposer or the digitiser which are optional, so it can be used only for machines which have these features.

<Mode> sets the superimposing mode and can be set to the value listed in

Table 2.12.

When <Ym> is 1, the brightness of the television is halved.

When <CB> is 1, the colour bus of VDP is prepared for input, and, when 0, it is prepared for output.

When <sync> is 1, "external sync" is selected, and, when 0, "internal sync" is selected.

<Voice> specifies whether to mix external signal for output, and values are listed in Table 2.13.

<Video input> is used to alternate the input of external video signals. When it is 0, the RGB multiconnector is selected; when it is 1, external video signal connector is selected.

<AV control> specifies AV control terminal output of the RGB multiconnector. When it is 0, the output is OFF; when it is 1, the output is ON.

Table 2.12 Input values for SET VIDEO <mode>.

Mode	S1	S2	TP	Display screen
0	0	0	0	Computer
1	0	1	1	Computer
2	0	1	0	Superimpose
3	1	0	0	Television

Note: In the case of mode 0, external sync cannot be used. In other modes the compoalte output of VDP is not available. S1, S0, and TP are the names of flags in the VDP register.

Table 2.13 Input values for SET VIDEO <voice>

Voice	Function for external voice signal
0	No mixing
1	Right channel mixed
2	Left channel mixed
3	Both channels mixed

\* COPY SCREEN [<mode>]

This statement is used for writing data from the colour bus to VRAM, for example, after digitising. This is valid for screen modes 5 to 8.

In mode 0, one field of signals is digitised and written to the display page; in mode 1, two successive fields (that is, one frame) of signals are written to (display page - 1)th page and the display page, so the display page should

be an odd page when the mode is 1. The default mode is 0.

## 2.9 Additions for Timer Features

\* GET DATE <string variable name> [,A]

This statement is for reading the date from the timer and assigning it to the string variable. The format of date to be read is as follows:

YY/MM/DD (YY = lower two digits of year, MM = month, DD = day)

e.g.) 85/03/23 (March 23, 1985)

When option A is specified, the alarm date is read.

\* SET DATE <string expression>[, A]

This statement sets date to timer. The form of parameter and option is the same as "GET DATE"

e.g.) SET DATE "85/03/23"

\* GET TIME <string variable>[, A]

This statement is for reading time from the timer and assigning it to a string variable. The form of time to be read is as follows:

HH:MM:SS (HH = hour, MM = minute, SS = second)

e.g.) 22:15:00 (22 hours 15 minutes 0 seconds)

When A is specified, the time for the alarm is read.

\* SET TIME <string expression>[, A]

This statement sets the time to the timer. The form of parameter and option is the same as "GET TIME".

e.g.) SET TIME "22:15:00"

\* The Alarm

Since the alarm feature is optional, the action taken at the specified time depends on the machine (ordinarily nothing happens).

When the alarm is to be set in both "SET DATE" and "SET TIME", "SET TIME" should be done first (when "SET TIME" is done, date of the alarm set by "SET DATE" will be erased).

The minimum setting for alarm is in minutes (setting in seconds is ignored).

## 2.10 Additions for Memory Switch

Using "SET" instructions, various settings described below can be stored to the battery-powered RAM in CLOCK-IC. Settings based on these are done automatically at system startup (when the system is powered or reset). "SET TITLE", "SET PROMPT", and "SET PASSWORD" use the same RAM, so only the most recent instruction is valid.

\* SET ADJUST (<X-coordinate offset>, <Y-coordinate offset>)

This statement sets the location to display on the screen. The coordinate offset is from -7 to 8.

\* SET BEEP <timbre>, <volume>

This statement sets BEEP sound. <Timbre> and <volume> are from 1 to 4.

Table 2.14 shows the correspondance of <timbre> and to the actual sound.

Table 2.14 Input values for <timbre> of SET BEEP

Timbre	Sound
1	High tone beep (same as MSX1)
2	Low tone beep
3	2 - tone beep
4	3 - tone beep

\* SET TITLE <string expression>[, <title colour>]

This statement specifies the title and the colour of the initial screen at system startup. <Title> is set by a string of up to 6 characters and <colour> is one of the values on Table 2.15. When <title> is 6 characters, keyboard input is awaited just after the title screen is displayed.

Table 2.15 Available colours in SET TITLE

Color	1	2	3	4
Screen color	Blue	Green	Red	Orange

\* SET PROMPT <prompt>

This statement sets the prompt. <Prompt> can have up to 6 characters.

\* SET PASSWORD <password>

This statement sets a system password. <Password> is a string expression up to 255 characters. Once this statement is done, input of the password is requested for invoking the system. When the correct password is given, the system is normally invoked; otherwise, correct password input is requested. When the system is invoked by pressing both graphic key and stop key, no password input is requested (in this case, the password setting has been done by the key cartridge; however, password input is always required for system startup). The password is disabled by specifying a null character in SET TITLE.

#### \* SET SCREEN

This statement records the current parameters of the "SCREEN" statement. At the system startup, they are automatically set. Items to be recorded are the following:

Screen number of text mode	Key click switch
Screen width of text mode	Printer option
Foreground, background, and border colours	Cassette baud rate
Function key switch	Display mode

#### 2.11 Additions for RAM Disk

On MSX1 RAM from 0000H to 7FFFH was used only by DOS. On MSX2, however, this portion can be used as a RAM disk of up to 32K bytes. The format of the file name for RAM disk is described below, where <filename> is a string which consists of 1 to 8 characters and <extension> is one which consists of 1 to 3 characters. Note that ";" (colon), "." (period), control characters of character codes 00H-1FH, and graphic symbols consisting of two bytes cannot be used.

MEM: <filename>[.<extension>]

The following are executable operations for the RAM disk:

1. Load/save a BASIC program (always saved in ASCII format)  
SAVE, LOAD, RUN, MERGE

When any of the above commands is executed from the program, control returns to the command level.

2. Read/write a sequential file  
OPEN, CLOSE  
PRINT #, PRINT USING #  
INPUT #, LINE INPUT #, INPUT\$  
EOF, LOC, LOF

The RAM disk does not support the following instructions:

1. Random file Read/Write
2. BLOAD, BSAVE
3. COPY

\* CALL MEMINI [(<size>)]

This statement specifies the amount of memory to be used as a RAM disk, initialises the RAM disk, and deletes all files. When the RAM disk is to be used, this statement should always be executed.

<Size> is "the amount of memory to be used as RAM disk minus 1". By default, the maximum size is allocated for RAM disk. "CALL MEMINI(0)" causes the RAM disk feature to be disabled.

\* CALL MFILES

This statement displays file names on the RAM disk.

\* CALL MKILL ("<filename>")

This statement deletes the specified file.

\* CALL MNAME ("<old filename>" AS "<new filename>")

This statement renames the specified file.

## 2.12 Other Additions

\* PAD (<expression>)

This function returns status to touch pad (touch panel), light pen, mouse, or track ball.

When <expression> is 0 to 7, it returns the status to touch pad as on MSX1, and, when <expression> is 8 to 11, it returns the status to light pen. Since the coordinates and the value of the switch are read when "PAD(8)" is executed, other data should be read after confirming that the value of PAD(8) is -1 (see Table 2.16).

Table 2.16 <Expression> returning status to light pen

Expression	The value returned
8	-1 when data of light open is valid; otherwise, 0
9	X - coordinate of light pen
10	Y - coordinate of light pen
11	-1 when switch of light pen is pressed; otherwise, 0

This statement returns the status of the mouse or the track ball connected to port 1 when <expression> is 12 to 15 or connected to port 2 when it is 16 to 19 (see Table 2.17). The mouse and track ball are automatically distinguished from each other.

Table 2.17 <Expression> returning status to mouse or track ball

Expression	The value returned
12, 16	- 1; for input request
13, 17	X - coordinate
14, 18	Y - coordinate
15, 19	0 (unused)

Coordinate data is read when PAD(12) or PAD(16) is examined. Coordinate data should be obtained after examining these. The STRIG function is used with the joystick to input the status of the trigger button.

### 3. INTERNAL STRUCTURE OF BASIC

Knowledge of how the BASIC interpreter controls and executes programs is necessary for more advanced use of BASIC. The internal structure of BASIC is discussed next.

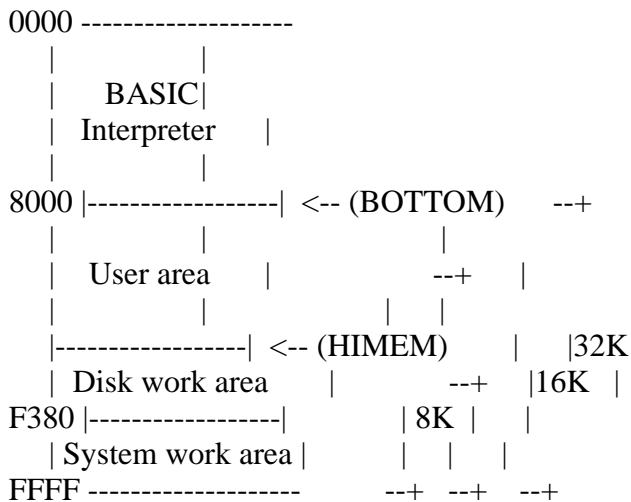
#### 3.1 User's Area

The lowest address of the user's area was different in the MSX1 machine whose amount of RAM was 8K, 16K, 32K, or 64K; in MSX2, it is always 8000H, because MSX2 machines have at least 64K of RAM. It can be obtained from the content of BOTTOM (FC48H).

The highest address of the user's area when no disk drives are connected is F380H; when disk drives are connected (using DISK-BASIC), it depends on the number of disk drives or on the disk capacity. It can be obtained from the content of HIMEM (FC4AH) after reset and before executing CLEAR statement.

Figure 2.6 shows the state of memory when MSX is invoked.

Figure 2.6 State of memory for BASIC mode



Note: Though the machine has more than 32K bytes of RAM, only 32K bytes are

used for BASIC. On MSX2, however, another 32K bytes can be used as a RAM disk by BASIC.

When developing a program on MSX2, we recommend you create it at addresses 8000H to DE3FH as if to install a 2DD-2 drive whose highest address of the user's area is the lowest. The work area of the disk can grow even larger, therefore, HIMEM of the application program should be checked to prevent disasters even in the worst situation. The following are ways to prevent this:

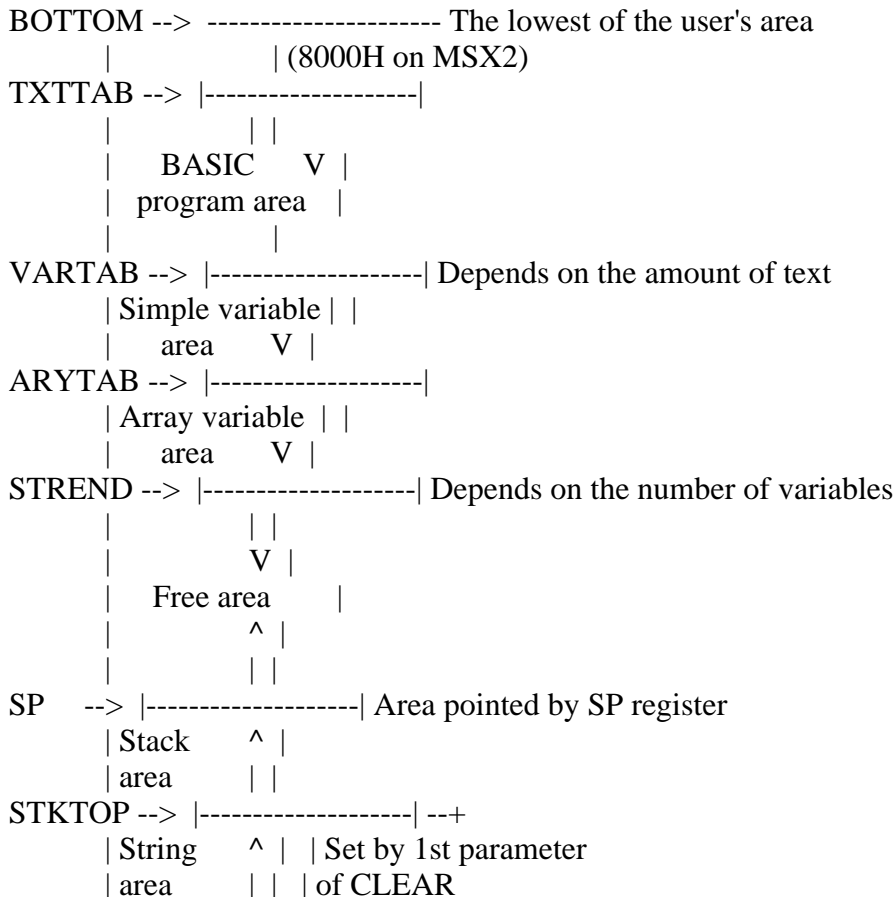
1. Make the work area relocatable
2. Get the work area from BOTTOM
3. Stop after instructing to reduce the number of drives

On MSX, even when disks are mounted, they can be cut off by resetting while pressing the SHIFT key. When only one drive is mounted, the normal invocation causes the work area for two drives to be allocated (mainly for 2 drive simulator): in such a case, invoking the work area for only one drive is possible by resetting while pressing the CTRL key. If these steps are taken, more user's area can be allocated.

### 3.2 Detailed View of the User's Area

Figure 2.7 shows how the user's area will be used in BASIC, and Table 2.18 shows the work area with information about where these areas start. This work area is read-only (the initialising routine sets it when reset), so actions when it is changed are not guaranteed.

Figure 2.7 State of the user's area





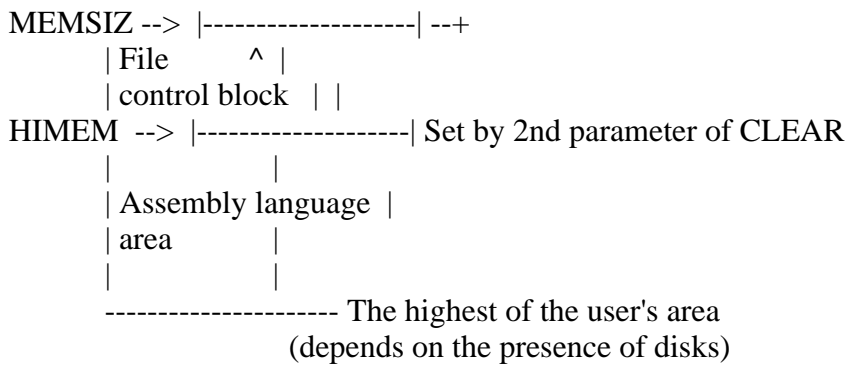


Table 2.18 Work areas with start and end addresses of each area

Area name	Start address	End address
User's area	[BOTTOM (FC48H)]	([HIMEM (FC4AH)] when reset) - 1
Program area	[TXTTAB (F676H)]	[VARTAB (F6C2H)] - 1
Simple variable area	[VARTAB (F6C2H)]	[ARYTAB (F6C4H)] - 1
Array variable area	[ARYTAB (F6C4H)]	[STREND (F6C6H)] - 1
Free area	[STREND (F6C6H)]	[SP register] - 1
Stack area	[SP register]	[STKTOP (F674H)] - 1
String area	[STKTOP (F674H)]	[MEMSIZ (F672H)] - 1
(start of unused area)	[FRETOP (F69BH)]	
File control block	[MEMSIZ (F672H)]	[HIMEM (FC4AH)] - 1
Assembly language area	[HIMEM (FC4AH)]	to the end of the user's area

Roles of each user's area are described below.

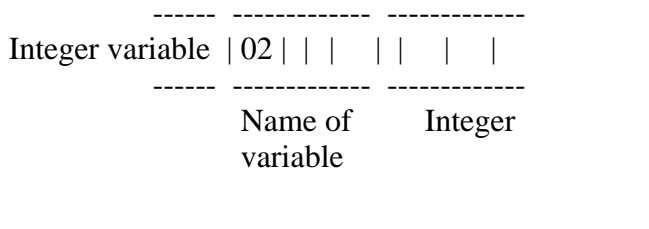
**\* BASIC program area**

A program written in BASIC is stored from the lowest address (8000H on MSX2) of the user's area and its size depends on the amount of the program.

**\* Variable area**

The variable area is located just after the BASIC program area. It is secured to store the name and the value of the variables used when executing the program. The variables storage formats are shown in Figure 2.8 (simple variables) and Figure 2.9 (array variables). Using array variables without declaring in the DIM statement causes the area to be allocated as an array with ten indexes. However, arrays which are more than four dimensional must be declared.

Figure 2.8 Storage format of simple variables



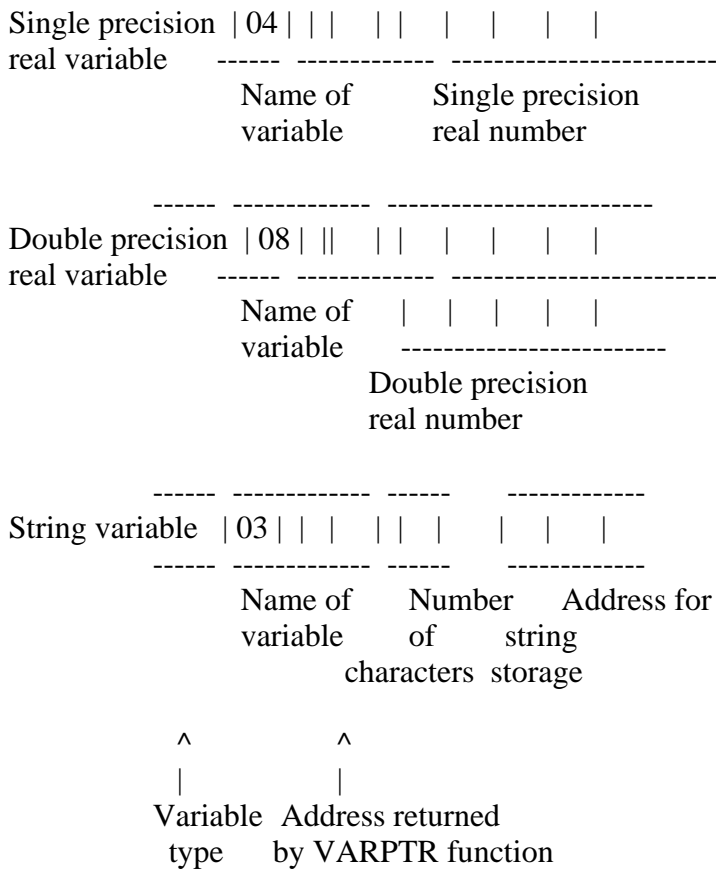
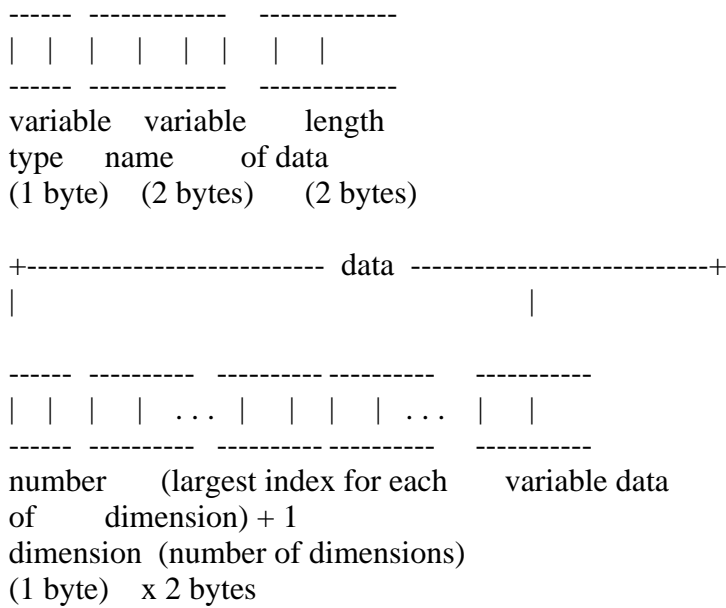
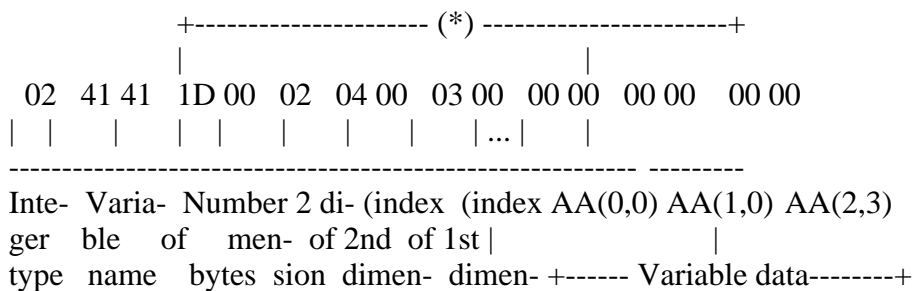


Figure 2.9 Storage format of array variables



e.g.) DEFINT A : DIM AA (2,3)



"AA" of (\*) array sion)+1 sion)+1

Note: variable data format is the same as the storage format of simple variables. The lower of the 2-byte value is stored first, and the higher byte last.

\* Free area

If the program area or the variable area grows too large or a lot of data is stacked and the free area runs out, an "OUT OF MEMORY" error occurs. The amount of free area can be checked by examining PRINT FRE(0) using the FRE function in BASIC.

\* Stack area

This is the stack area used by BASIC. It is used in order from high-order address when executing GOSUB or FOR.

\* String area

This area is used to reserve the contents of string variables and used from high-order address. The space in this area can be specified by the first parameter of the CLEAR statement in BASIC. The default is 200 bytes. Exhausting the space in this area causes a "OUT OF STRING SPACE" error. The amount of unused area can be checked by examining PRINT FRE("") using the FRE function in BASIC

\* File control block

File information is stored in this area with 10BH (267) bytes allocated for each file. The amount of space for files can be specified by the MAXFILES statement of BASIC. At reset, the area for one file (MAXFILES = 1) is allocated. Another space is always allocated for SAVE and LOAD instructions, so actually area for two files is allocated. Table 2.19 shows the format of file control block.

Table 2.19 File control block (FCB) format

Offset	Label	Meaning
+ 0	FL.MOD	Mode of the file opened
+ 1	FL.FCA	Pointer (low) to FCB for BDOS
+ 2	FL.LCA	Pointer (high) to FCB for BDOS
+ 3	FL.LSA	Backup character
+ 4	FL.DSK	Device number
+ 5	FL.SLB	Internal use for the interpreter
+ 6	FL.BPS	FL.BUF location
+ 7	FL.FLG	Flag containing various information
+ 8	FL.OPS	Virtual head information
+ 9...	FL.BUF	File buffer (256 bytes)

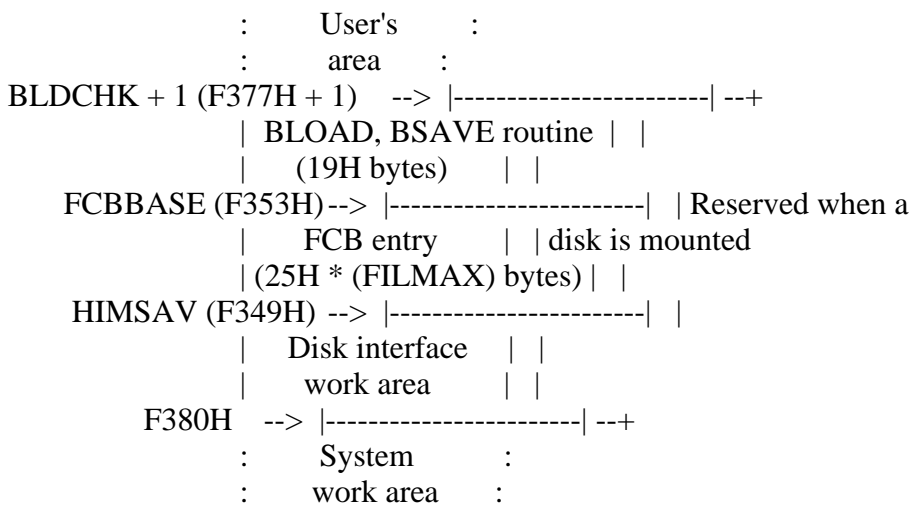
\* Assembly language area

Use this area to write programs in assembly language or to operate from memory directly. To do these, this area should be reserved by CLEAR statement.

\* Work area for disk

Figure 2.10 shows the work area allocated when a disk is mounted. Note that this area does not exist when no disk is mounted. Labels to the right of this figure shows the address information which resides there.

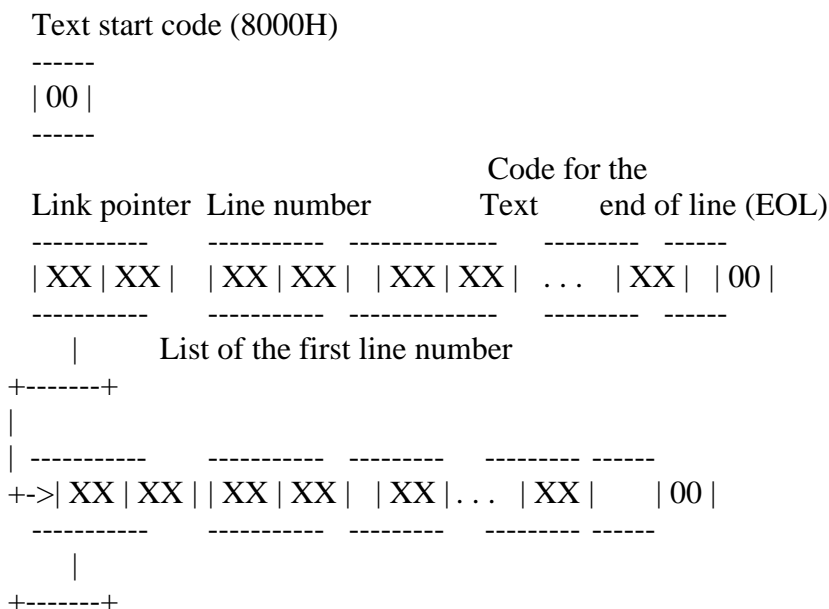
Figure 2.10 Work area for disk



3.3 Storage format of BASIC programs

Programs are stored in memory as shown in Figure 2.11 and the meaning of its contents are described below.

Figure 2.11 Text storage format



```

|
| . . .
+> . . . Line number
|
|
+-----+
|
| -----
+>| XX | XX | | XX | XX | | XX | . . . | XX | | 00 |
|
| -----
| List of the last line number
+-----+
|
| -----
+>| 00 | 00 | Code for the end of text (EOT)
|
| -----

```

Note: Link pointers and line numbers are stored with their low bytes first and high bytes last.

\* Link pointer

The text pointer to the next line is given in the form of an absolute address.

\* Line number

This stores the line number of the program, normally the values from 0 to 65529 (from 0000H to FFF9H). It is possible to make line numbers of 65530 or more, but LIST command does not list them.

\* Text

The program body is stored here in the intermediate code format. Reserved words (keywords), operators, numeric values are converted to the intermediate codes, and others (such as variable names or string constants) are stored as character codes. Table 2.20 lists the intermediate codes and Figure 2.12 shows the numeric formats in text.

See the appendix at the end of this book for character codes. Graphic characters are stored in 2 bytes (2 characters) of "CHR\$(1) + (graphic character code + 64)", so be careful when defining graphic characters.

Table 2.20 List of intermediate codes

>	EE	ERR	E2	PAINT	BF
=	EF	ERROR	A6	PDL	FF A4
<	F0	EXP	FF 8B	PEEK	FF 97
+	F1	FIELD	B1	PLAY	C1

-	F2	FILES	B7	POINT	ED
*	F3	FIX	FF A1	POKE	98
/	F4	FN	DE	POS	FF 91
^	F5	FOR	82	PRESET	C3
\	FC	FPOS	FF A7	PRINT	91
ABS	FF 86	FRE	FF 8F	PSET	C2
AND	F6	GET	B2	PUT	B3
ASC	FF 95	GOSUB	8D	READ	87
ATN	FF 8E	GOTO	89	REM	3A 8F
ATTR\$	E9	HEX\$	FF 9B	RENUM	AA
AUTO	A9	IF	8B	RESTORE	8C
BASE	C9	IMP	FA	RESUME	A7
BEEP	C0	INKEY\$	EC	RETURN	8E
BIN\$	FF 9D	INP	FF 90	RIGHT\$	FF 82
BLOAD	CF	INPUT	85	RND	FF 88
BSAVE	D0	INSTR	E5	RSET	B9
CALL	CA	INT	FF 85	RUN	8A
CDBL	FF A0	IPL	D5	SAVE	BA
CHR\$	FF 96	KEY	CC	SCREEN	C5
CINT	FF 9E	KILL	D4	SET	D2
CIRCLE	BC	LEFT\$	FF 81	SGN	FF 84
CLEAR	92	LEN	FF 92	SIN	FF 89
CLOAD	9B	LET	88	SOUND	C4
CLOSE	B4	LFILES	BB	SPACE\$	FF 99
CLS	9F	LINE	AF	SPC(	DF
CMD	D7	LIST	93	SPRITE	C7
COLOR	BD	LLIST	9E	SQR	FF 87
CONT	99	LOAD	B5	STEP	DC
COPY	D6	LOC	FF AC	STICK	FF A2
COS	FF 8C	LOCATE	D8	STOP	90
CSAVE	9A	LOF	FF AD	STR\$	FF 93
CSNG	FF 9F	LOG	FF 8A	STRIG	FF A3
CSRLIN	E8	LPOS	FF 9C	STRING\$	E3
CVD	FF AA	LPRINT	9D	SWAP	A4
CVI	FF A8	LSET	B8	TAB(	DB
CVS	FF A9	MAX	CD	TAN	FF 8D
DATA	84	MERGE	B6	THEN	DA
DEF	97	MID\$	FF 83	TIME	CB
DEFDBL	AE	MKD\$	FF B0	TO	D9
DEFINT	AC	MKI\$	FF AE	TROFF	A3
DEFSNG	AD	MKS\$	FF AF	TRON	A2
DEFSTR	AB	MOD	FB	USING	E4
DELETE	A8	MOTOR	CE	USR	DD
DIM	86	NAME	D3	VAL	FF 94
DRAW	BE	NEW	94	VARPTR	E7
DSKF	FF A6	NEXT	83	VDP	C8
DSKI\$	EA	NOT	E0	VPEEK	FF 98
DSKO\$	D1	OCT\$	FF 9A	VPOKE	C6
ELSE	3A A1	OFF	EB	WAIT	96
END	81	ON	95	WIDTH	A0
EOF	FF AB	OPEN	B0	XOR	F8
EQV	F9	OR	F7	-----	
ERASE	A5	OUT	9C		
ERL	E1	PAD	FF A5		

Figure 2.12 Numeral formats in text

----- Octal number (&O)   0B   XX : XX   ----- -----	
----- Hexadecimal number (&H)   0C   XX : XX   ----- -----	
----- Absolute address of the Line number (after RUN)   0D   XX : XX   destination line for the ----- branch instruction in memory.	
----- Line number (before RUN)   0E   XX : XX   After RUN, identification ----- code is made 0DH and the line number is changed to the absolute address.	Destination line number ----- for the branch instruction.
----- Integer from 10 to 255 (%)   0F : XX   ----- -----	
----- Integer from 0 to 9 (%)     11 to 1A ----- -----	
----- Integer from 256 to 32767 (%)   1C   XX : XX   ----- -----	
----- Single precision real (!)   1D   XX : XX : XX : XX   ----- -----	
----- Double precision real (#)   1F   XX : XX : XX : XX : ----- : XX : XX : XX : XX   ----- -----	
----- Binary (&B)   "&"   "B"   . . . Characters of "0" or "1" ----- following "&B"	

Numbers called "identification codes" are assigned numeric values to distinguish them from reserved words and variable names, and by referring to them the following values can be recognised.

The high and low bytes of a 2-byte numeric value are stored in reverse. Signed numeric values have only the intermediate codes + or - preceding the identifying codes, numeral values themselves are always stored as positive values. Floating-point notations are almost the same as the descriptions of Math-Pack (Mathematical Package) in the APPENDIX, note that numerical values are always stored as positive. Binary numbers (&B) do not have identifying codes and are stored as ASCII codes.

#### 4. LINKING WITH ASSEMBLY LANGUAGE PROGRAMS

As described so far, MSX BASIC version 2.0 has powerful features, but, if you wish to save execution time even more or to make full use of MSX2 hardware, you should use assembly language. The following sections show how to call assembly language programs from BASIC and gives the information you will need.

#### 4.1 USR Function

To call the assembly language routine from BASIC, follow the steps described below. The value in parenthesis of the USR function is passed to the assembly language routine as an argument. The argument may be either an expression or a string expression.

1. Specify the starting address of the assembly language program for the execution, using DEF USR statement.
2. Call the assembly language program by USR function.
3. Execute RET (C9H) when returning from the assembly language routine to BASIC.

e.g.) To call the assembly language program whose starting address is C000H:

```
DEFUSR=&HC000
A=USR(0)
```

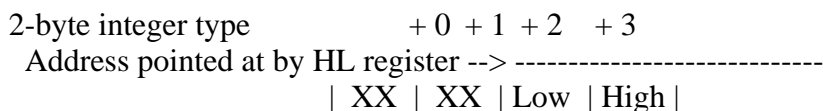
#### 4.2 Data Exchange by the Argument and Return Value of USR Function

When the argument is passed from BASIC to the assembly language program, its type can be checked by examining the contents of register A in the assembly language program (see Table 2.21). Since the object value is stored in the form as shown in Figure 2.13 according to the argument type, you can get the value according to the format. As an example, List 2.1 shows a program which receives an argument of the string type.

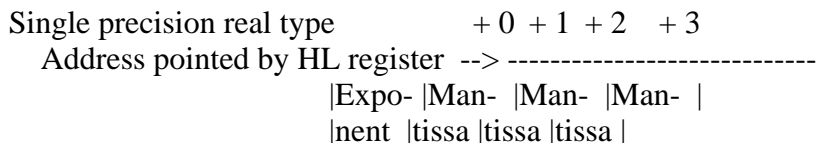
Table 2.21 Argument types assigned to register A

2	2-byte integer type
3	String type
4	Single precision real type
8	Double precision real type

Figure 2.13 How values are passed as arguments



Note: "XX" may be anything





```

-----
Double precision real type      + 0 + 1      + 7
Address pointed by HL register --> -----
|Expo- |Man- | . . . |Man- |
|nent |tissa | . . . |tissa |
-----

```

```

String type                    + 0 + 1 + 2 These three
Address pointed by DE register --> ----- bytes are
|   | Low | High | called the
----- string
^ |   | descriptor.
Number of characters --+ +-----+
Points to the address
of the string

```

List 2.1 Example of the argument of string type

=====

```

;*****
; List 2.1 print string with USR function
; to use, do DEF USR=&HB000 : A$=USR("STRING")
;*****
;
CHPUT EQU 00A2H ;character output

ORG 0B000H

RDARG: CP 3
RETNZ ;parameter is not string

PUSH DE
POP IX ;IX := string descriptor
LD A,(IX+0) ;get string length
LD L,(IX+1) ;get string pointer (low)
LD H,(IX+2) ;get string pointer (high)
OR A
RETZ ;if length = 0

RD1: PUSH AF
LD A,(HL) ;get a characetr
CALL CHPUT ;put a character
POP AF
DEC A
RETZ
INC HL
JR RD1

END

```

=====

On the other hand, these values passed as arguments can be passed to BASIC as USR function values by changing them in the assembly language program. In

this case the type of return value can also be changed to types other than that of the argument from BASIC by changing VALTYP (F663H). Note that the amount of characters for a string cannot be changed.

### 4.3 Making New Commands

In MSX the reserved words "CMD" and "IPL" are currently unused and by changing the pointers to these words (FE0DH and FE03H) to jump to your own assembly language routine, new commands can be built. List 2.2 shows a simple example.

List 2.2 Making CMD command

```

=====
;*****
; List 2.2  make CMD command ( turn on/off the CAPS LOCK )
; to initialize command:      DEF USR=&HB000 : A=USR(0)
; to use command:            CMD
;*****
;
CHGCAP  EQU    0132H      ;CAPS LAMP on/off
CAPST   EQU    0FCABH    ;CAPS LOCK status
HCMD    EQU    0FE9DH    ;CMD HOOK

      ORG     0B000H

;----- CMD initialize ----- Note: Executing this section adds the
                          CMD command

      LD  BC,5      ;NEW HOOK SET
      LD  DE,HCMD
      LD  HL,HDAT
      LDIR
      RET

;----- new HOOK data ----- Note: 5-byte data to be written into
                          hook (FE0DH)

HDAT:  POP AF
      JP  CAPKEY
      NOP

;----- executed by CMD ----- Note: Actual CMD command

CAPKEY: CALL  CHGCAP
      LD  A,(CAPST)
      CPL
      LD  (CAPST),A
      RET

      END
=====

```

The first "POP AF" written to the pointer in this case, discards the error handling addresses stacked at "CMD" execution. Without this, the "RET" command would jump to the error handling routine instead of returning to BASIC. It is a way to use this address for printing errors inside of user routine.

These pointers are reserved for future expansion, so should not be used with application programs on the market.

#### 4.4 Expansion of CMD command

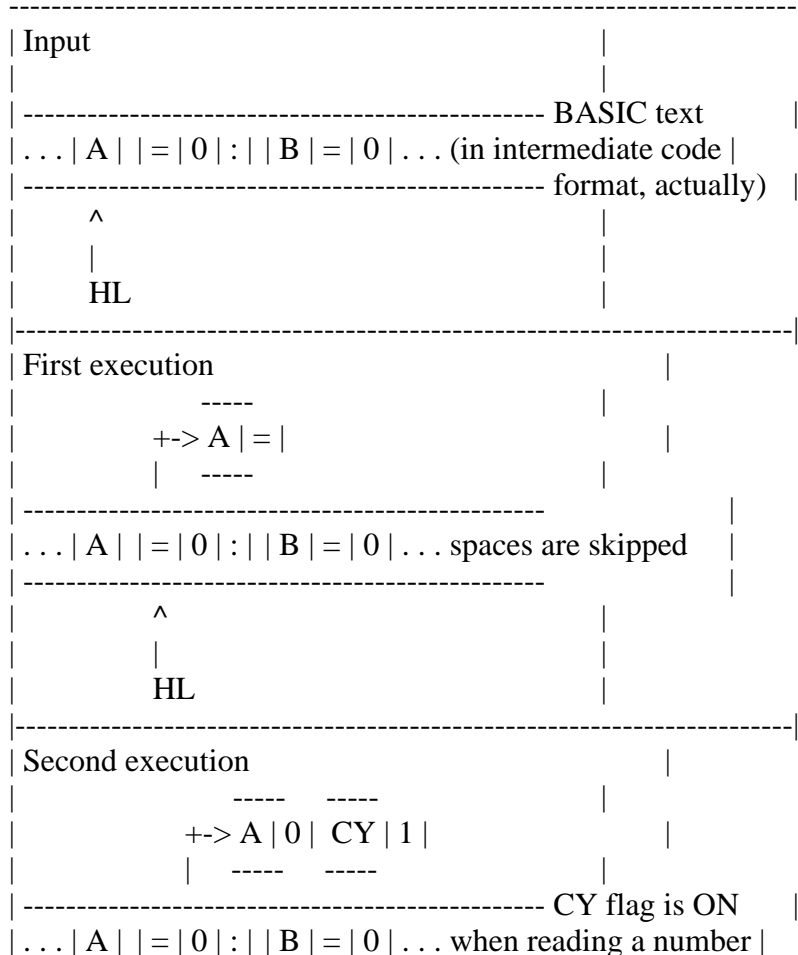
For more sophisticated expansions of statements it is useful if arguments can be passed to the CMD command. As the HL register points to the next location after "CMD" in the BASIC text when the assembly language routine is called, it can be done by appreciating the successive string. The following is a list of internal routines, useful for these.

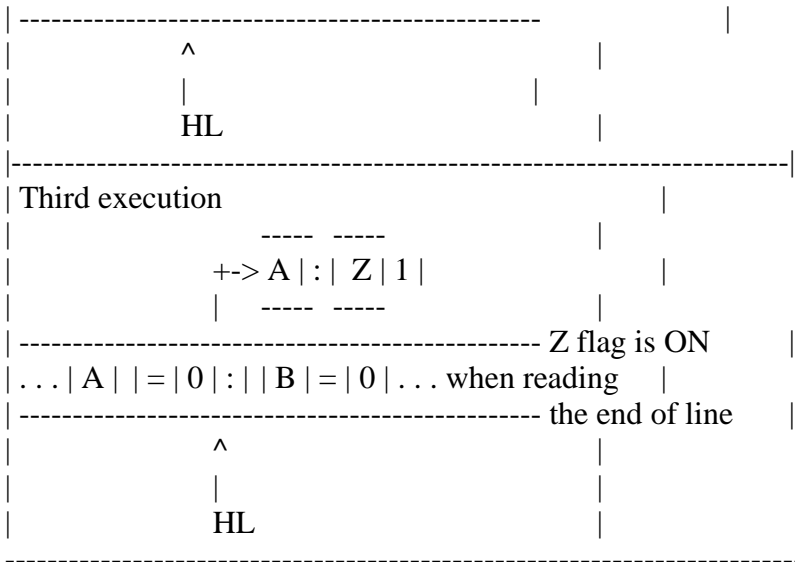
\* CHRGT (4666H/MAIN) ---- Extract one character from text (see Figure 2.14)

Input: HL <-- Address pointing to text  
 Output: HL <-- Address of the extracted character  
 A <-- Extracted character  
 Z flag <-- ON at the end of line (: or 00H)  
 CY flag <-- ON if 0 to 9

Purpose: Extract one character from the text at (HL + 1). Spaces are skipped.

Figure 2.14 Input/output state of CHRGT





\* FRMEVL (4C64/MAIN) ---- Evaluate an expression in text (see Figure 2.15)

Input: HL <-- Starting address of the expression in text

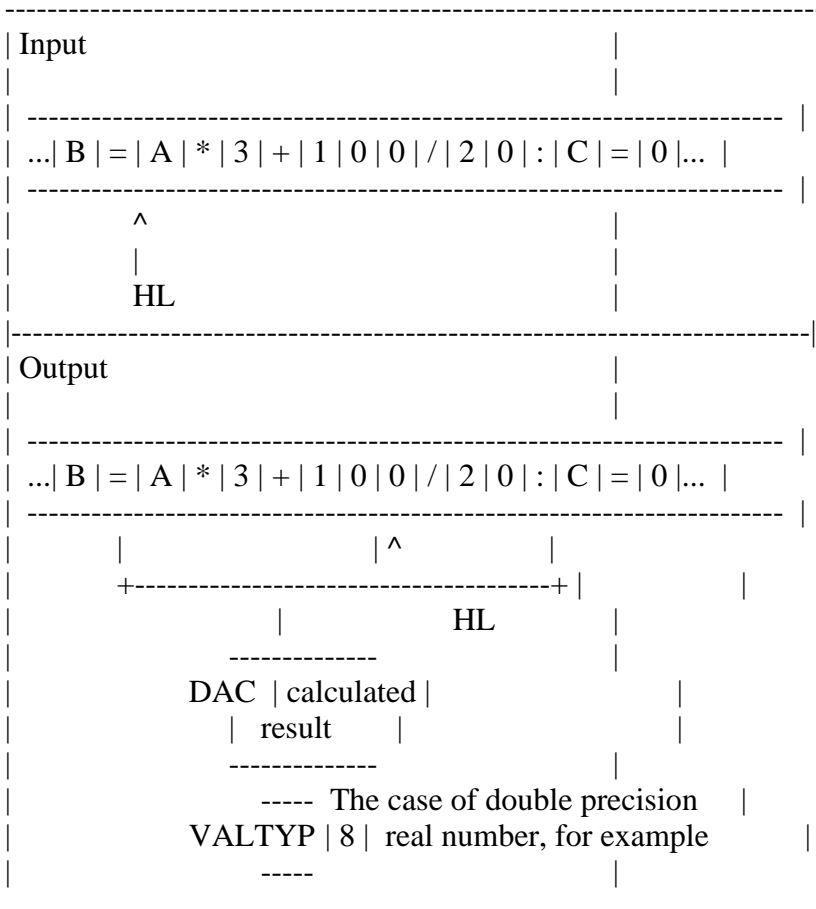
Output: HL <-- Address after the expression

[VALTYP (F663H)] <-- Value 2, 3, 4 or 8 according to the expression

[DAC (F7F6H)] <-- Result of the evaluation of the expression

Purpose: Evaluate an expression and make output according to its type.

Figure 2.15 Input/output state of FRMEVL



\* FRMQNT (542F/MAIN) ---- Evaluate an expression un 2-byte integer type.

Input: HL <-- Starting address of the expression in text  
Output: HL <-- Address after the expression  
DE <-- Result of evaluation of the expression  
Purpose: Evaluate an expression and make output in integer type (INT). When the result is beyond the range of 2-byte integer type, an "Ovwrflow" error occurs and the system returns to the BASIC command level.

\* GETBYT (521C/MAIN) ---- Evaluate an expression in 1-byte integer type.

Input: HL <-- Starting address of the expression in text  
Output: HL <-- Next address of expression  
A, E <-- Result of expression evaluation  
(A and E contains the same value.)

Purpose: Evaluate an expression and make 1-byte integer output. When the result is beyond the range of 1-byte integer type, an "Illegal function call" error occurs and the execution returns to BASIC command level.

\* FRESTR (67D0/MAIN) ---- Register a string.

Input: [VALTYP (F663H)] <-- Type (if not string type, an error occurs)  
[DAC (F7F6H)] <-- Pointer to string descriptor

Output: HL <-- Pointer to string descriptor

Purpose: Register the result of the string type obtained by FRMEVL and obtain its string descriptor. When evaluating a string, this is generally combined with FRMEVL described above to use as follows:

```

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

```

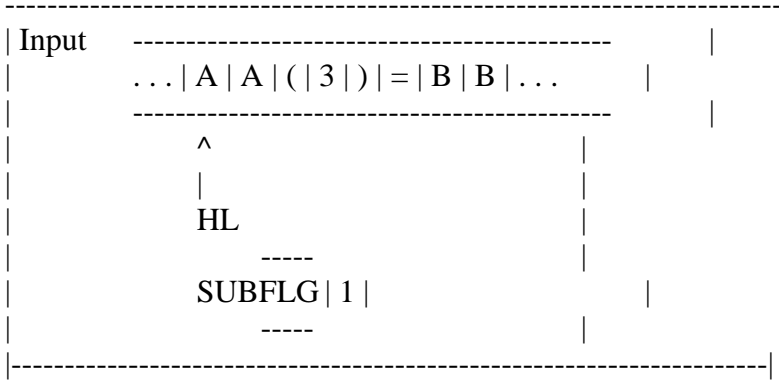
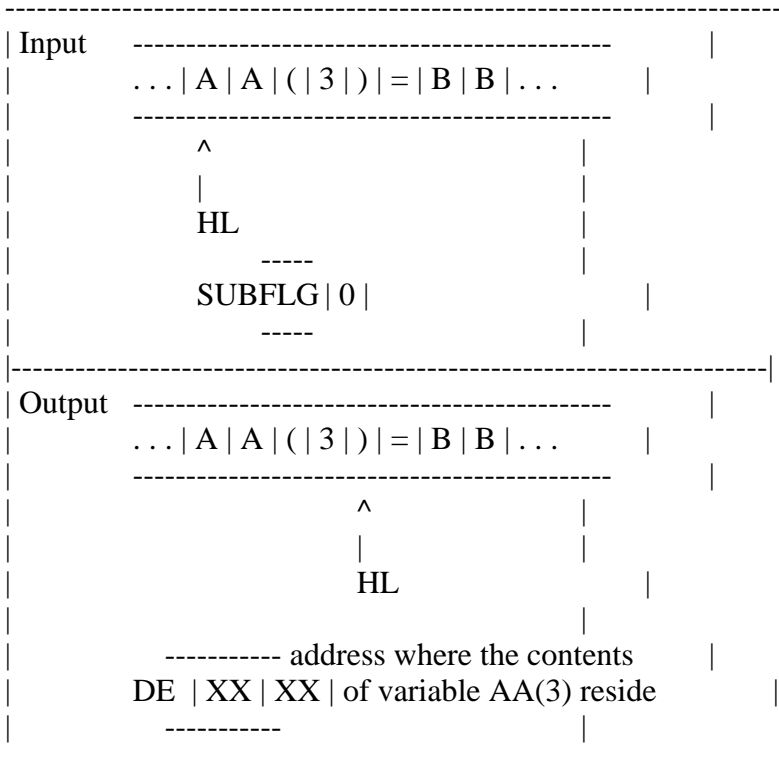
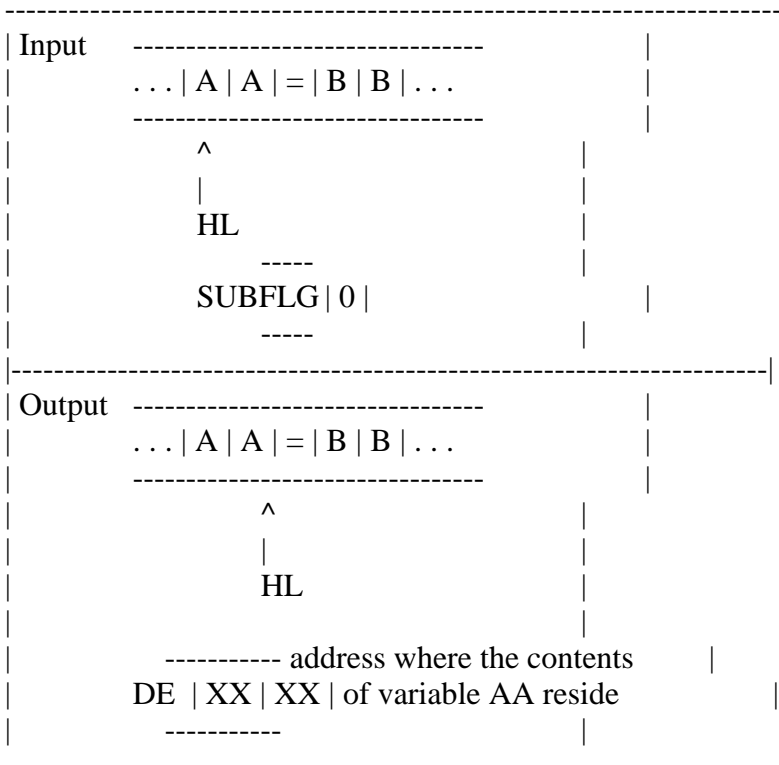
\* PTRGET (5EA4/MAIN) ---- Obtain the address for the storage of a variable  
(see Figure 2.16).

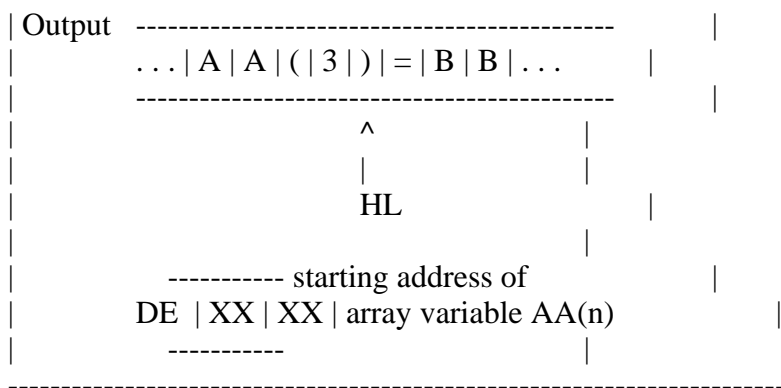
Input: HL <-- Starting address of the variable name in text  
[SUBFLG (F6A5H)] <-- 0: Simple variable,  
other than 0: array variable

Output: HL <-- Address after the variable name  
DE <-- Address where the contents of the objective variable  
is stored

Purpose: Obtain the address for the storage of a variable (or an array variable). Allocation of the area is also done when the area for the objective variable has not been allocated. When the value of [SUBFLG] is set to other than 0, the starting address of the array is obtained, other than individual elements of the array.

Figure 2.16 Input/output state of PTRGET





\* NEWSTT (4601H/MAIN) ---- Execute a text

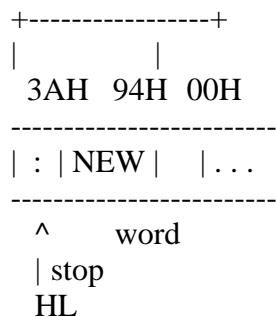
Input: HL <-- Address of the text to be executed

Output: ----

Purpose: Execute a text. The state of the text is necessary to be as same as shown in Figure 2.17.

Figure 2.17 Memory setting for NEWSTT

Intermediate codes of BASIC are contained here.



Since these internal routines are for BASIC texts, the same error handling as BASIC is done when an error occurs. In this case, by changing H.ERROR (FFB1H), the user can handle the error (the E register contains the error number) (see List 2.3).

### List 2.3 Changing error handling routine

=====

```

;*****
; List 2.3 Your own error
; To use, do DEF USR=&HB000 : A=USR(0)
;*****
;
HERR EQU 0FFB1H ;error hook
SYNERREQ 2 ;syntax error code
CHPUT EQU 0A2H ;character output
NEWSTT EQU 4601H ;run
READYR EQU 409BH

```

ORG 0B000H

Note:

;----- command initialize ----- When this portion is executed, the error

handling routine is changed.

```
LD BC,5      ;SET NEW HOOK
LD DE,HERR
LD HL,HDAT
LDIR
RET
```

```
HDAT: JP ERROR
      NOP
      NOP
```

Note:

;----- error routine ----- Error handling body

```
ERROR: LD A,E      ;when in error, E holds error code
      CP SYNERR    ;syntax error ?
      RETNZ        ;no
```

```
      LD HL,DATA1  ;yes
LOOP: LD A,(HL)    ;put new error message
      CP "$"
      JR Z,EXIT
      PUSH HL
      CALL CHPUT
      POP HL
      INC HL
      JR LOOP
```

```
EXIT: JP READYR    ;BASIC hot start
```

```
DATA1: DEFM OOHPS!! ;new error message
      DB 07H,07H,07H,"$"
```

```
END
```

---

## 4.5 Interrupt usage

The Z80 CPU has INT and NMI interrupt terminals. The MSX, however, uses only INT. The INT terminal gets 60 [Hz] signals, so timer interrupts are executed 60 times per 1 second. As the interrupt mode of Z80 is set to 1, 38H is called when an interrupt occurs and then the system control jumps to the timer interrupt routine, where various operations such as key input are done.

The timer interrupt routine jumps to hook H.TIMI (FD9FH) in mid course. Using this hook enables the user to add a function to this timer interrupt routine. Though there is ordinarily only a RET command, be careful when peripherals such as disks are connected and this hook is already in use. In this case, careless modifications causes peripherals to be disabled, so prearrangement is necessary to make machines to execute that normally. List 2.4 is an example of this handling and the interrupt usage.

### List 2.4 Correct usage of timer interrupt hook

---



```

;*****
; List 2.4 How to use HOOK safety
; This routine uses TIMER INTERRUPT HOOK
; and turn on/off CAPS LOCK
; To start, do DEF USR=&HB000 : A=USR(0)
; To end, do DEF USR=&HB030 : A=USR(0)
;*****

```

```

CHGCAP EQU 0132H ;CAPS LAMP on/off
CAPST EQU 0FCABH ;CAPS LOCK status
TIMI EQU 0FD9FH ;timer interrupt hook
JPCODE EQU 0C3H
TIMER EQU 020H

```

```
ORG 0B000H
```

```
;----- interrupt on ----- Note: restore the former hook
when changing the hook
```

```
INTON: DI
LD HL,TIMI ;OLD HOOK SAVE
LD DE,HKSAVE
LD BC,5
LDIR

LD A,JPCODE ;NEW HOOK SET
LD (TIMI),A
LD HL,INT
LD (TIMI+1),HL
EI
RET
```

```
ORG 0B030H
```

```
;----- interrupt off ----- Note: restore the reserved hook and exit
```

```
INTOFF: DI
LD HL,HKSAVE
LD DE,TIMI
LD BC,5
LDIR
EI
RET
```

```
;----- interrupt routine -----
```

```
INT:PUSH AF
LD A,(CAPST)
OR A
JR Z,CAPON
```

```
CAPOFF: LD A,(COUNT1)
DEC A
LD (COUNT1),A
JR NZ,FIN
LD A,TIMER
LD (COUNT1),A
```

```

XOR  A
LD  (CAPST),A
LD  A,0FFH
CALL CHGCAP
JR  FIN

```

```

CAPON: LD  A,(COUNT2)
      DEC  A
      LD  (COUNT2),A
      JR  NZ,FIN
      LD  A,TIMER
      LD  (COUNT2),A
      LD  A,0FFH
      LD  (CAPST),A
      XOR  A
      CALL CHGCAP

```

```

FIN:  POP  AF
      CALL HKSAVE      ;old HOOK call

      RET

```

```

COUNT1: DEFB  TIMER
COUNT2: DEFB  TIMER

```

```

HKSAVE: NOP          ;old HOOK save area
      NOP
      NOP
      NOP
      RET

      END

```

## 5. NOTES ON SOFTWARE DEVELOPMENT

There are some matters, when developing the software for MSX, that should be followed so as to make the software work without any problems on any MSX machines. The following describes these matters and introduces information that will help you develop software.

### \* BIOS

The purpose of BIOS is to separate the hardware and the software and to make the software still valid if the hardware changes. Applications for sale which manage input and output should use BIOS (except for VDP).

BIOS is called through the jump table which begins at 0000H of MAIN-ROM. Though MSX2 has a jump table on SUB-ROM, it is used for calling the extended functions. The branch destination of the jump table or, the contents of BIOS may be modified for the hardware modification or the extension of the function, so applications should not call them directly. Though this book has some examples that call addresses other than the BIOS jump table, you should consider them for information only (see BIOS list in APPENDIX). Applications

can call Math-Pack and internal routines for the extended statements described above. These will not be changed in the future.

#### \* Work area

F380H to FFFFH of MAIN-RAM cannot be used, as it is a work area for BIOS and BASIC interpreter. Free space in the work area cannot be used, because it is reserved for the future use. See "3.1 User's area" for the work area of the disk.

#### \* Initialisation of RAM and stack pointer

The contents of RAM are unpredictable when the machine is powered and areas other than system work areas are not initialised. Applications should initialise the work area. There was once an application which expected the contents of RAM to be 00H and was unusable.

The value of the stack pointer when the INIT routine (see Section 7 of Chapter 5) in the ROM cartridge is called is unpredictable and the value when disk interface has been initialised is smaller than when not. For these reasons some programs which did not initialise the stack pointer had unpredictable results. Programs which are invoked by the INIT routine and continue processing (that is, programs which do not need to use peripherals such as disks or BASIC interpreter) should initialise the stack pointer.

#### \* Work area of extended BIOS

When using extended BIOS calls, a stack should be placed above C000H so that CPU can refer to the work area even if the slot is switched over. For the same reason, FCB of RS-232C should be above 8000H.

#### \* Work area of device drivers, etc.

Special attention should be paid for the allocation of the work area of programs which reside in memory with another program at the same time, programs such as the device driver or a subroutine called from BASIC.

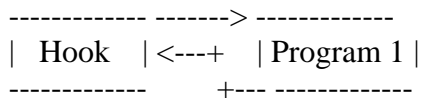
The INIT routine of the cartridge changes BOTTOM (FC48H), reserves the area between the old BOTTOM and new BOTTOM as its work area, and records the address of the work area to 2-byte area SLTWRK (FD09H) allocated for each slot. For more details, see Section 7 of Chapter 5.

#### \* Hook

When using the RS-232C cartridge, change the hook for an interrupt. For example, if another cartridge uses an interrupt hook, the RS-232C cartridge cannot use the same hook. To prevent this, the previous contents of the hook (inter-slot call command for the interrupt handling routine of RS-232C cartridge, in the example above) should be copied to another location, and, when called by the hook, it should be called so that all cartridges intending to use the hook can receive control (see Figure 2.18). For more details, see Section 7 of Chapter 5.

Figure 2.18 Initialisation of the hook

Initialisation of the hook of program 1



Initialisation of the hook of program 2



\* VRAM capacity

The capacity of VRAM can be found by evaluating bits 1 and 2 of MODE (FAFCH) (see Table 2.22).

Table 2.22 Getting the information about the VRAM capacity

[FAFCH]			VRAM Capacity
Bit 2	Bit 1		
0	0	16K (MSX1)	
0	1	64K (MSX2)	
1	0	128K (MSX2)	

\* BASIC version number

The following methods can be used for applications to find out the version number of BASIC.

1. Read the contents of 2DH of MAIN-ROM (00H = version 1.0, 01H = version 2.0, and so on).
2. In version 2.0 or later versions, EXBRSA (FAF8H) contains the slot address of SUB-ROM. When it has none (00H), the version is version 1.0.

\* International MSX

There are different kinds of MSX for various countries. The following items are different by country:

- Keyboard arrangement, character set, PRINT USING format
- Timer interrupt frequency

The version of machine can be found by reading the ID byte information in ROM (see Figure 2.19) and the correspondence for MSX of each country will be

accomplished (see Table 2.23).

Figure 2.19 Contents of ID byte

```

-----
| 2BH | b0 --+
----- | character generator
  b1 |
      | 0: Japan      1: United States, etc.
  b2 | 2: USSR
      |
  b3 --+

  b4 --+
      | date format
  b5 |
      | 0:Y/M/D   1:M/D/Y   2:D/M/Y
  b6 --+

  b7  interrupt period (VSYNC)
      0:60Hz 1:50Hz

```

```

-----
| 2CH | b0 --+
----- | keyboard
  b1 |
      | 0:Japan 1:United States, etc.
  b2 | 2:France  3:United Kingdom
      | 4:Germany 5:USSR      6:Spain
  b3 --+

  b4 --+
      |
  b5 | information about PRINT USING
      | or others
  b6 |
      |
  b7 --+

```

Table 2.23 MSX format for each country

-----						
			PRINT USING			
Country	TV set	Date format	Initial screen	String length	Re- place cy	Curren- symbol
		mode	specif.	char.		
-----+-----+-----+-----+-----+-----+-----						
Japan	NTSC (60Hz)	YY/MM/DD	Screen 1	&	@	(yen)
UK	PAL (50Hz)	DD/MM/YY	Screen 0	\	&	(pound)
Internat.	PAL (50Hz)	MM/DD/YY	Screen 0	\	&	\$(dollar)
US	NTSC (60Hz)	MM/DD/YY	Screen 0	\	&	\$(dollar)
France	SECAM (50Hz)	DD/MM/YY	Screen 0	\	&	\$(dollar)
Germany	PAL (50Hz)	DD/MM/YY	Screen 0	\	&	\$(dollar)

| USSR | NTSC (60Hz) | MM/DD/YY | Screen 0 | \ | & |\$(dollar)|  
| Spain | PAL (50Hz) | MM/DD/YY | Screen 0 | \ | & |\$(dollar)|

---

#### \* Escape sequence

MSX has the escape sequence feature (see Appendix), which can be used in the PRINT statement of BASIC, and in console output of BIOS or BDOS call (MSX-DOS). The escape sequence feature is a subset of DEC VT52 terminal and Heathkit H19 terminal.

#### \* Returning to BASIC

##### - Warm start

After selecting a slot of MAIN-ROM, jump to 409BH of MAIN-ROM. If the work area of BASIC has not been destroyed, the BASIC prompt will be displayed. The contents of register and stack at the jump are ignored.

Another way is to execute the next command in internal routine NEWSTT (see 4.4 of Chapter 4) (see Figure 2.20)

Figure 2.20 Input setting of NEWSTT for the warm start

```
3AH 81H 00H
-----
| : | END | | = (:END)
-----
^ word
| stop
HL
```

#### \* Auto start

In the case of simple game cartridges which do not use the BIOS or BASIC work areas, the program can be invoked by writing a starting address for the program to "INIT" in ROM header. But using this method prevents the initial settings of another cartridge, so disk drives cannot be used.

To prevent this, the hook "H.STKE" is at FEDAH; write the inter-slot call command in the program to be invoked at the execution on "INIT" routine of the cartridge, and return to the system by RET command. Then after initialising all cartridges and after preparing the DISK BASIC environment if there is a disk, the hook is called, so the objective program can be invoked. This method is also effective when there is no disk (see APPENDIX).

#### Error code list

1. NEXT without FOR There is no FOR statement corresponding with the NEXT statement.
2. Syntax error There is an error in syntax.

3. RETURN without GOSUB      The RETURN statement does not correspond to the GOSUB statement.
4. Out of DATA                There is no data to be READ by the READ statement.
5. Illegal function call      There is an error in the function or numeric value specification.
6. Overflow                    The numeric value has overflow.
7. Out of memory              The free area has been exhausted.
8. Undefined line number      There is no such a line number in the program.
9. Subscript out of range    The subscript value of the array variable exceeds the declared range.
10. Redimensioned array      The array is declared twice.
11. Division by zero          The attempt to divide by zero is made. The negative exponent of zero is done.
12. Illegal direct             The statement which cannot be executed in the direct mode is carried out directly.
13. Type mismatch            There is a conflict in the data types.
14. Out of string space      The string space is exhausted.
15. String too long          The length of the string is longer than 255 characters.
16. String formula too complex    The specified string is too complex.
17. Can't CONTINUE            The CONT command cannot be executed.
18. Undefined user function    An attempt was made to use the user-defined function which has not been defined by DEF FN statement.
19. Device I/O error          An error occurred in input/output of device.
20. Verify error              The program on cassette and the one in memory are not the same.
21. No RESUME                There is no RESUME statement in the error handling routine.
22. RESUME without error      The RESUME statement is used other than in the error handling routine.
23. Undefined.
24. Missing operand            Necessary parameters are not specified.

25. Line buffer overflow There are too many characters for the input data.
- 26 to 49. Undefined.
50. FIELD overflow The field size defined in FIELD statement exceeds 256 bytes.
51. Internal error An error occurred inside BASIC.
52. Bad file number File number which has not been OPENed is specified. The specified file number exceeds the number specified in MAXFILES statement.
53. File not found The specified file is not found.
54. File already open The file has already been OPENed.
55. Input past end The attempt to read the file is made after reading the end of it.
56. Bad file name There is an error in the specification of the file name.
57. Direct statement Data other than the program is found while loading the ASCII format program.
58. Sequential I/O only Random access to the sequential file is made.
59. File not OPEN The specified file has not been OPENed yet.
60. Bad FAT Unusual disk format.
61. Bad file mode An incorrect input/output operation is made in the OPENed mode.
62. Bad drive name There is an error in the drive name specification.
63. Bad sector number There is an error in the sector number.
64. File still open The file has not been closed.
65. File already exists The file name specified in NAME statement already exists on the disk.
66. Disk full The free area of the disk has been exhausted.
67. Too many files The number of files exceeds 112 (the directory space has been exhausted).
68. Disk write protected The disk is protected from writing.
69. Disk I/O error Some trouble occurred in the disk input/output.
70. Disk offline The diskette is not in.



- 71. Rename across disk      NAME statement is done across different disks.
- 72. File write protected    The file has the read-only attribute set.
- 73. Directory already exists    The directory name specified in CALL MKDIR statement already exists.
- 74. Directory not found    The specified directory is not found.
- 75. RAM disk already exists    Attempt to create the DOS 2 RAM disk when it already exists is made.
- 76 to 255. Undefined.

\* Note: Errors with codes 72 to 75 are added from version 2 of MSX DISK-BASIC. In version 1 they are undefined.

Use larger numbers first for user error definition.

## MSX2 TECHNICAL HANDBOOK

-----

Edited by:            ASCII Systems Division  
 Published by:        ASCII Corporation - JAPAN  
 First edition:        March 1987

Text files typed by: Nestor Soriano (Konami Man) - SPAIN  
 March 1997

### Changes from the original:

- Remarks (1) and (2) about the FCB format in version 2 of MSX-DOS have been added.
- Description of function call 06H is modified. The name of this function in the original text is "String output", and the setup description is "E register <-- starting address of string to send to the console. When 0FF is specified, the character will be sent to the console as character code."
- In description of function calls 13H (Deleting files) and 23H (File size acquisition), the original text has "DE register <-- starting address of opened FCB" in setup field. Instead of this, the correct setup explanation is set.
- In description of function 26H (Random writing to the disk 2), the correct term "FCB" is set instead of "DMA" in DE register setup explanation.
- In description of function 27H (Random readout 2), the phrase "When this number is almost one, the data which has been read is set in the area indicated by DMA" has been added.

-----

## CHAPTER 3 - MSX-DOS

Large capacity storage devices with high-speed access are necessary for business applications. That is why a disk operating system was added to the MSX machine. The DOS (disk operating system) is also required to handle the large amount of data on the disk effectively. MSX-DOS is derived from MS-DOS which is used widely on 16-bit machines. Thus, it represents the most powerful DOS environment for Z-80 based machines. Chapter 3 describes the basic operations of MSX-DOS and the use of the system calls.

### 1. OVERVIEW

What kind of software is MSX-DOS? What does it offer to users? The following sections describe and introduce the features, functions, and software configurations of MSX-DOS.

#### 1.1 Features of MSX-DOS

##### \* Consolidation of disk operating environment

MSX-DOS is the disk operating system for MSX computers. It works with any version of MSX and can be operated on both the MSX1 and MSX2 without any problem. Disk operation on MSX is always done via MSX-DOS. This is also true concerning MSX DISK-BASIC, which uses BDOS calls for disk input/output. MSX-DOS and DISK-BASIC use the same disk format so that file conversion between BASIC and DOS is not necessary. This greatly increases operating efficiency and allows more effective use of file resources when MSX-DOS is used as the software development environment.

##### \* Compatibility with MS-DOS

MSX-DOS, created on the basis of MSX-DOS (ver 1.25) which is a disk operating system for 16-bit personal computers, uses the same file format as MS-DOS. It is compatible with MS-DOS at the file level so that MSX-DOS can read and write files written on MS-DOS disks. In turn MS-DOS can read and write files created by MSX-DOS. Both disk operating systems use similar commands, so users who are familiar with MSX-DOS can easily use MS-DOS when upgrading to 16-bit machines.

##### \* Using CP/M applications

MSX-DOS has system call compatibility with CP/M and can execute most programs created on CP/M without any modification. Most CP/M applications can thus be easily used with MSX-DOS. This opens up a large library of existing software which can be run on the MSX machines.

#### 1.2 MSX-DOS Environment

##### \* System requirements

To use MSX-DOS, a minimum configuration of 64K bytes RAM, a CRT, one disk drive, and a disk interface ROM is required. If less than 64K bytes RAM is installed, MSX-DOS cannot be used. MSX computers can only use MSX-DOS if they have 64K bytes RAM or more. Since MSX2 computers always have 64K bytes or more of RAM, they can always run MSX-DOS. A limited disk basic is used on those machines with less than 64K bytes RAM. Disk interface ROM is always supplied with the disk drive, and, on MSX machines with an internal disk drive, it resides inside the machine. For those machines using disk cartridges, it is in the cartridge.

\* System supported

MSX-DOS supports up to eight disk drives. On a one-drive system, it has a 2-drive simulation feature (it uses one drive as two drives by replacing diskettes temporarily). It supports keyboard input, screen output, and printer output.

\* Media supported

MSX-DOS, which has a flexible file manager that does not depend on the physical structure of the disk, supports various media and uses 3.5 inch double density disks as standard. Either a one-sided disk called 1DD or two-sided disk called 2DD is used. Each of them uses either an 8-sector track format so four kinds of media can be used. The Microsoft formats for these four types are shown below.

Table 3.1 Media supported by MSX-DOS

	1DD, 9 sectors	2DD, 9 sectors	1DD, 8 sectors	2DD, 8 sectors
media ID	0F8H	0F9H	0FAH	0FBH
number of sides	1	2	1	2
tracks per side	80	80	80	80
sectors per track	9	9	8	8
bytes per sector	512	512	512	512
cluster size (in sectors)	2	2	2	2
FAT size (in sectors)	2	3	1	2
number of FATs	2	2	2	2
number of recordable files	112	112	112	112

Note: See chapter 3 for the meanings of the above words.

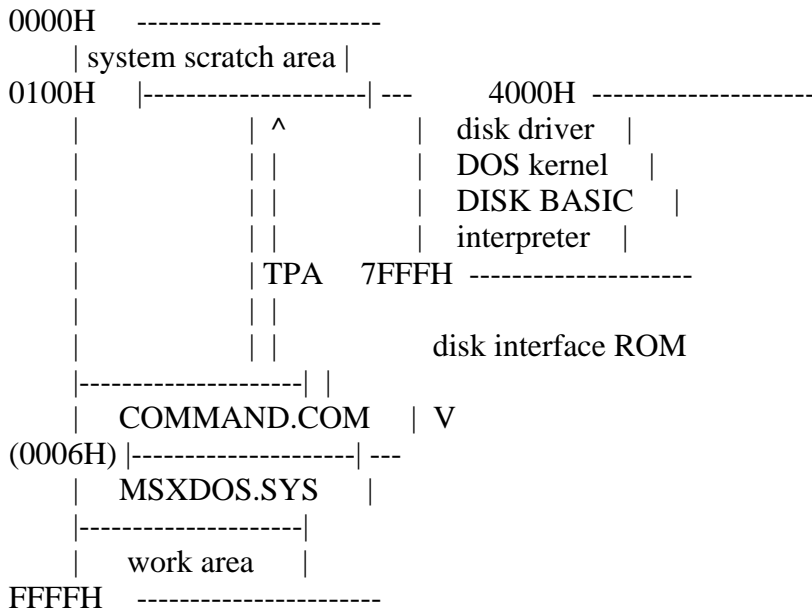
### 1.3 MSX-DOS System Resources

\* Memory map

MSX-DOS consists of the following modules: COMMAND.COM, MSXDOS.SYS, and a disk interface ROM. It resides in memory as shown in Figure 3.1 when MSX-DOS is active. COMMAND.COM and MSXDOS.SYS are disk files until MSX-DOS is booted

and then read into RAM after that. Disk interface ROM includes a disk driver, DOS kernel, and DISK-BASIC interpreter.

Figure 3.1 MSX-DOS memory map



The area 00H to FFH of RAM is called the system scratch area, which is used by MSX-DOS for exchanging data with other programs. This area is important when using system calls, which are described later. The area which begins at 0100H and ends where the contents of 0006H of RAM indicates is called the TPA (Transient Program Area). This area is accessible by the user. MSXDOS.SYS always resides at a higher address than TPA (when destroyed, the result is unpredictable), and COMMAND.COM is placed in TPA.

\* COMMAND.COM

The main operation of MSX-DOS is to accept typing commands from the keyboard and execute them. In this case the program COMMAND.COM is responsible for the process from getting a string to interpreting and executing it, or accepting commands from the user interface. Programs executed by COMMAND.COM consists of internal commands, batch commands, and external commands.

Internal commands are inside COMMAND.COM and on RAM. Typing an internal command causes COMMAND.COM to call and execute it immediately.

For the external command, COMMAND.COM loads the routine from disk to TPA and executes it (the execution of external commands always begins at 100H). In this case COMMAND.COM frees its own area for the external command. That is, COMMAND.COM might erase itself and writes the external command onto it, when the external command is small enough and does not use the high-end of TPA, COMMAND.COM would not be destroyed. When the external command ends with "RET", MSXDOS.SYS examines whether COMMAND.COM has been destroyed (by using checksum) and, if so, re-loads COMMAND.COM onto RAM and passes the control to COMMAND.COM.

Batch commands are carried out by getting command line input from a batch file instead of from the keyboard. Each step of the batch file can execute

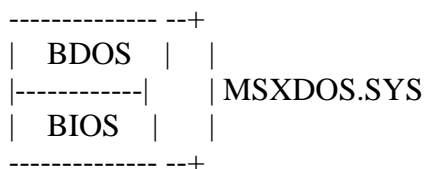
any internal command or external command. It is possible that the batch command executes another batch command, but the control will not return to the caller after the called batch command is done.

#### \* MSXDOS.SYS

MSXDOS.SYS, core of MSX-DOS, controls disk access and communications with peripherals. These MSXDOS.SYS functions are opened as "BDOS (Basic Disk Operating System)" so that the user can use them. Each routine opened is called a "system call", which is useful in developing software for managing the disk (see chapter 4). Each execution is, however, not done by MSXDOS.SYS itself but DOS kernel. MSXDOS.SYS is an intermediation which arranges input/output requests from COMMAND.COM or external commands and passes them to the DOS kernel.

MSXDOS.SYS includes a portion called BIOS other than BDOS, as shown in Figure 3.2. BIOS, which has been prepared to be compatible with CP/M, is not normally used.

Figure 3.2 MSXDOS.SYS



#### \* DOS kernel

The DOS kernel is the fundamental input/output routine which resides in the disk interface ROM and executes BDOS functions of MSXDOS.SYS. Actually, any system call function can be executed using the DOS kernel. DISK-BASIC executes system calls by calling the DOS kernel directly.

#### \* Procedure for invoking MSX-DOS

MSX-DOS is invoked by the following procedure:

1. Resetting MSX causes all the slots to be examined first, and when two bytes, 41H and 42H, are written in the top of the examined slot, the slot is interpreted as connected to a certain ROM. When connected with ROM, the INIT (initialize) routine whose address is set to the header portion of ROM is carried out. In the case of the INIT routine of the disk interface ROM, the work area for the drive connected to the interface is allocated first.
2. When all slots have been examined, FEDAH (H.STKE) is then referred to. Unless the contents of this address is C9H (unless a certain routine is set to the hook of H.STKE during INIT routine), the environment for DISK-BASIC is prepared and execution jumps to H.STKE.
3. When the contents of H.STKE is C9H in the examination above, the cartridge with TEXT entry is searched in each slot and, if found, the environment for DISK-BASIC is prepared, and then the BASIC program at the cartridge is

carried out.

4. Then, the contents of the boot sector (logical sector #0) is transferred to C000H to C0FFH. At this time, when "DRIVE NOT READY" or "READ ERROR" occurs, or when the top of the transferred sector is neither EBH nor E9H, DISK-BASIC is invoked.

5. The routine at C01EH is called with CY flag reset. Normally, since code "RET NC" is written to this address, nothing is carried and the execution returns. Any boot program written here in assembly language is invoked automatically.

6. RAM capacity is examined (contents of RAM will not be destroyed). Less than 64K bytes causes DISK-BASIC to be invoked.

7. The environment for MSX-DOS is prepared and C01EH is called with a CY flag set. MSXDOS.SYS is loaded from 100H, and the execution jumps to 100H. After this, MSX-DOS transfers itself to a high order address. If MSXDOS.SYS does not exist, DISK-BASIC is invoked.

8. MSXDOS.SYS loads COMMAND.COM from 100H and jumps to its start address. COMMAND.COM also transfers itself to a high order address and then begins to execute. If COMMAND.COM does not exist, the message "INSERT A DISKETTE" appears and the execution waits for the correct diskette to be inserted in the drive.

9. At the first boot for MSX-DOS, when a file named "AUTOEXEC.BAT" exists, it is carried out as a batch file. When MSX-DOS is not invoked and DISK-BASIC starts, if a BASIC program named "AUTOEXEC.BAS" exists, it will be carried out.

## 2. OPERATION

This section describes how to type command line input from the keyboard. This is the basis of MSX-DOS operations. Several examples of actual use and their explanations will be given for the commands used in MSX-DOS.

### 2.1 Basic Operations

#### \* Message at startup

When MSX-DOS is invoked, the following message appears on the screen:

Figure 3.3 Screen at atartup

```
-----  
|                               |  
| MSX-DOS version 1.03      |  
| Copyright 1984 by Microsoft |  
|                               |  
| COMMAND version 1.11     |  
|                               |  
-----
```

The upper two lines show the version of MSXDOS.SYS and its copyright. The last line shows the version of COMMAND.COM.

#### \* Prompt

Then, a prompt (input request symbol) appears under the version description. The prompt for MSX-DOS consists of two characters: the default drive name plus ">".

#### \* Default drive

The term "default drive" as the first character of the prompt is the drive to be accessed automatically when the drive name is omitted. When the default drive is A, for example, referring to a file "BEE" on drive "B" needs to be typed as "B:BEE". A file "ACE" on drive A, however, can be typed simply as "ACE" omitting the drive name.

ex.1) A>DIR B:BEE (<-- referring to "BEE" on drive B)

ex.2) A>DIR ACE (<-- referring to "ACE" on drive A)

#### \* Changing default drive

When using systems with more than one drive, typing "B" causes the default drive to be changed to B. When changing the default drive to C to H, "C" or the appropriate letter is needed. Specification of a drive which does not exist causes an error.

ex.1) A>B:  
B> (<-- Default drive has been changed to B)

ex.2) A>K:  
Invalid Drive Specification  
A> (<-- Drive K does not exist.  
Default drive is not changed.)

#### \* Command input

When a prompt is displayed it indicates that MSX-DOS requests a command to be input. By typing in a command, MSX-DOS can get an instruction.

Three forms of commands exist as shown in Table 3.2. The COMMAND.COM program interprets and executes these commands. MSX-DOS operations are repeats of the actions "give a command - make COMMAND.COM execute it".

Table 3.2 Three forms of commands

(1) Internal	Command inside COMMAND.COM. Assembly routine on RAM.	
command	Thirteen commands are prepared as described later.	
(2) External	Assembly routine on disk. It is loaded from disk at	
command	execution. Its file name has an extension "COM".	

```

-----|
| (3) Batch | Text file containing one or more commands. Commands |
| command | are executed orderly (batch operation). File names |
| | have the extension "BAT". |
-----|

```

\* File name convention

Files handled by MSX-DOS are expressed by a "file spec" which is described below:

- (1) File spec is expressed in the form "<drive>:<file name>".
- (2) <drive> is a character from A to H. When specifying the default drive, it can be omitted as well as the colon ":" following it.
- (3) <file name> is expressed in the form of "<filename>.<extension>".
- (4) <filename> is a string containing one or more (up to 8) characters. When more than 8 characters are specified, the ninth and subsequent characters are ignored.
- (5) <extension> is a string containing up to 3 (including zero) characters. When more than 3 characters are specified, 4th and subsequent characters are ignored.
- (6) <extension> can be omitted as well as a preceding period ".".
- (7) Characters which are available in <filename> and <extension> are shown in Table 3.3.
- (8) Cases are not sensitive. Capital letters and small letters have the same meaning.

Table 3.3 Available characters for file name

```

-----|
| Available | A to Z    0 to 9 $ & # % ( ) - @ ^ { } ' ` ! |
| characters | characters corresponding to character codes 80H to FEH |
|-----+-----|
| Unavailable | ~ * + , . / : ; = ? [ ] |
| characters | characters corresponding to character codes 00H to 20H |
| | and 7FH, FFH |
|-----|

```

\* Wildcards

Using a special character called a "wildcard" in the description of <filename> and <extension> of the file specification causes files with common characters to be specified. Wildcards are "?" and "\*".

- (1) "?" is a substitution for one character.  
ex) "TEXT", "TEST", "TENT" <-- "TE?T"  
"F1-2.COM", "F2-6.COM" <-- "F?-?.COM"



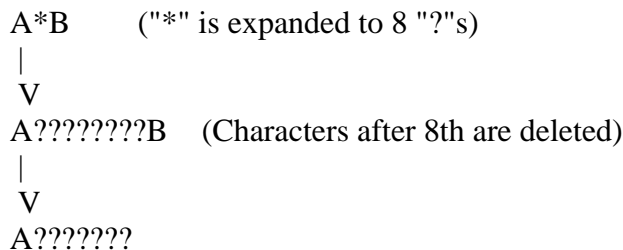
- (2) "\*" is a substitution for a string with any length.  
 ex) "A", "AB", "ABC" <-- "A\*"  
 "files with an extension .COM" <-- "\*.COM"  
 "all files" <-- "\*.\*"

When comparing existing file names and file names with wildcards, the portion less than 8 characters of <filename> and the portion less than 3 characters of <extension> are considered to be padded with spaces (" "). Thus, a specification "A???.??" is not expanded to "ABCDE.123" but to "AZ.9", as shown in Figure 3.4.

Figure 3.4 Wildcard expansion



An asterisk (\*) is interpreted as either 8 question marks or 3 question marks (?) depending on if it is in the file name position or file extension position. For example, a file name "A\*B" is not interpreted as "any strings which begin with A and end with B". It is interpreted as "any strings which begin with A", as shown below.



\* Device name

MSX-DOS does not need special commands for data input/output with peripherals. This means that it considers each objective device as a certain file (device file) and input/output actions are done by reading or writing to or from this file. This enables MSX-DOS users to treat input/output devices in the same way as files on a disk. Five devices are supported as standard by

MSX-DOS as shown in Table 3.4 and are specified with proper names. For this reason, these names can not be used to specify disk files. These device names with drive specifications or extensions are also treated as simple device names.

Table 3.4 Device names

Device name	Input/output device to be specified
AUX	Reserved name for input/output expansion which normally has the same effect as NUL
CON	Console (keyboard for input, screen for OUTPUT)
LST	Printer (ouput only; cannot be used for input)
PRN	Printer (same as LST)
NUL	Special device used as a dummy when the result is not desired to be displayed on the screen or put in a file. When used for input, always EOF.

\* Input functions using a template

A "template" is a character buffer area and can be used for command input. The template contains the previous command line most recently entered. It is possible to use the template for easier command entry. By taking advantage of this template feature, it is easy to execute previous commands again or to execute the command partially modified. The keys listed in Table 3.6 are used for the template operation.

\* Other special keys

In addition to the template operation keys, the following control keys are also available. These special key functions also support some other system calls described later.

Table 3.5 Special key functions

	Function
^C	stops command currently executed
^S	pauses screen output until any key is pressed

^P	send characters to the printer at the same time
	they appear on the screen
^N	resets ^P and send characters only to the screen
^J	feeds a line on the screen and continue input

Table 3.6 Template functions

Name	Keys used	Functions
COPY1	RIGHT, ^\	Gets one character from the template and displays it in the command line
COPYUP	SELECT, ^X	Gets characters before the character to be typed next (by keyboard) from the template and displays them on the command line
COPYALL	DOWN, ^_	Gets all characters from the location which the template is currently referring to the end of the line and displays them on the command line
SKIP1	DEL	Skips one character of the template
SKIPUP	CLS, ^L	Skips template characters before the character to be typed next (by keyboard)
VOID	UP, ESC, ^^, ^U, ^[	Discards current line input not changing the template
BS	LEFT, BS, ^H, ^]	Discards one character input and returns the location referred by the template by one character
INSERT	INS, ^R	Switches insert mode/normal input mode, in insert mode, displays keyboard input on the command line with fixing the location referred by the template
NEWLINE	HOME, ^K	Transfers the contents of current command line to the template
Return key		Feeds a line on screen but continues getting input. Transfers the contents of current command line to the template and executes it
Keys other than above		Displays a character corresponding to the key on the command line and skips one character of the template

Table 3.7 Template operation examples

Keyboard input	Command line display	Contents of template ("-" referred to)	indicates location currently referred to)
DIR ABCDE	A>DIR ABCDE	-----	
RETURN	A>	DIR ABCDE	
DOWN	A>DIR ABCDE	DIR ABCDE	
LEFT LEFT LEFT	A>DIR AB	DIR ABCDE	
INS XYZ	A>DIR ABXYZ	DIR ABCDE	
RIGHT RIGHT RIGHT	A>DIR ABXYZCDE	DIR ABCDE	
UP	A>	DIR ABCDE	
DOWN	A>DIR ABCDE	DIR ABCDE	
UP	A>	DIR ABCDE	
XXX	A>XXX	DIR ABCDE	
DOWN	A>XXX ABCDE	DIR ABCDE	
HOME	A>XXX ABCDE	XXX ABCDE	

#### \* Disk errors

When an error occurs during disk access, MSX-DOS retries sometimes. Still more errors cause MSX-DOS to display the following message and inquire what to do with them. Press one of the keys A, R, or I.

Figure 3.5 Error display

<error type>	error	<action>	drive	<drive name>
Write protect	Reading	A to H		
Not ready	Writing	-----		
Disk	-----			
Abort, Retry, Ignore?				
Abort: stops the disk access and ends the command execution				

| Retry: tries again |  
| Ignore: stops the disk access and continues the command execution |  
| |

---

The following error might occur other than listed above. It indicates that the pointer in FAT is pointing to a cluster which does not exist. When this error occurs, the diskette will be unusable.

### Bad FAT

## 2.2 Internal commands

Internal commands are assembly language programs grouped together in COMMAND.COM. It is not necessary to read them from the disk so they are executed fast. Following are 13 internal commands. This section describes their use.

BASIC ..... jumps to MSX DISK-BASIC  
COPY ..... copies a file  
DATE ..... displays or modifies date  
DEL ..... deletes a files  
DIR ..... displays a list of files  
FORMAT ..... formats a disk  
MODE ..... modifies number of characters to be displayed  
          in one line  
PAUSE ..... pauses a batch command operation  
REM ..... puts a comment line in a batch command  
REN ..... renames a file name  
TIME ..... displays or modifies time  
TYPE ..... prints the contents of a file  
VERIFY ..... turns on/off the verify mode

### \* BASIC

form: BASIC [<file spec>]

Starts DISK-BASIC. This is not done by loading BASIC onto RAM but by selecting BASIC-ROM in 0000H to 7FFFH by switching the slot, so it starts immediately. When <file spec> is specified, the corresponding BASIC program is automatically read and executed. To return to the MSX-DOS environment from BASIC, execute "CALL SYSTEM".

### \* COPY

This command copies the contents of one file to another. Specifying parameters enables various options.

#### (1) File duplication

form COPY <file spec 1> <file spec 2>

Duplicates the file specified by <file spec 1> into a file specified by <file

spec 2>. Files having the same names cannot be created on the same disk. On different disks, specifying the same names is possible.

examples:

```
A>COPY ABC XYZ <-- copies file "ABC" and makes a file "XYZ".
```

```
A>COPY B:ABC XYZ <-- copies a file "ABC" on drive B and makes
a file "XYZ".
```

```
A>COPY B:ABC C:XYZ <-- copies a file "ABC" on drive B and makes
a file "XYZ" on drive C.
```

When copying files, either ASCII or binary mode may be selected. The "/A" switch specifies ASCII mode and the "/B" switch specifies binary mode. If no mode is specified, binary mode is selected by default (except when combining files, described in (4) below, when ASCII is the default mode). Table 3.8 shows the differences between the ASCII and the binary modes.

Table 3.8 ASCII mode and binary mode

	Read from source file	Write to destination file
ASCII mode	ignore after 1AH (file end mark)	add one byte 1AH to end
Binary mode	read as long as physical file size	write without modification

examples:

```
A>COPY/A ABC XYZ <-- ABC to XYZ (both files are in ASCII mode)
```

```
A>COPY ABC/A XYZ/B <-- reads ABC in ASCII mode and writes it to XYZ
in binary mode
```

## (2) File duplication to another disk drive

```
formCOPY <file spec> [<destination drive>:]
```

Copies a file specified by <file spec> to <destination drive> under the same file name. When <destination drive> is omitted, it is copied to the default drive. The drive name included in the <file spec> must not be the same as the <destination drive>.

More than one file can be copied by using wildcards in the <file spec>. In this case, the file name is displayed on the screen each time the file is copied.

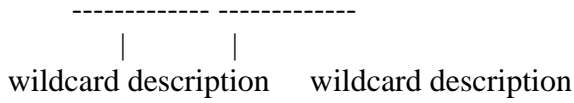
examples:

```
A>COPY *.COM B: <-- copies any files with extension "COM"
on default drive to drive B
```

A>COPY B:ABC <-- copies a file ABC to default drive

### (3) Simultaneous duplication of many files

form COPY <file spec 1> <file spec 2>

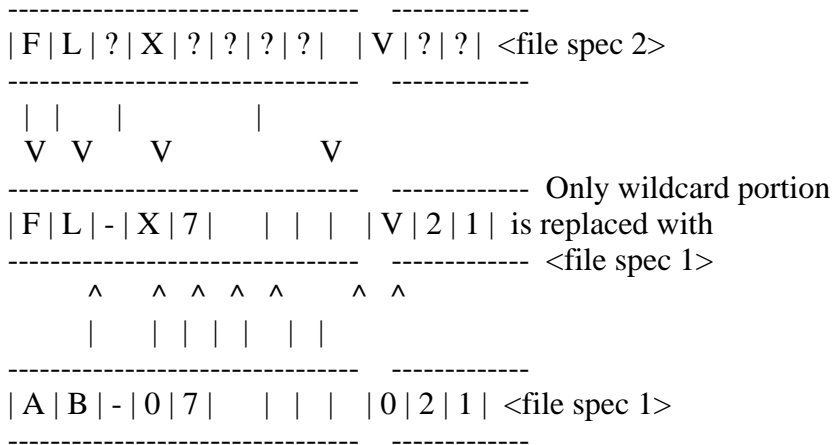


When <file spec 2>, the destination, is described using wildcards, the portions corresponding to wildcards are replaced with corresponding characters in <file spec 1>. For example, when

COPY AB-07.021 FL?X\*.V??

is executed, it is interpreted as shown in figure 3.6 and a file "FL-X7.V21" is created.

Figure 3.6 Wildcard specification of destination file



Using wildcards in the specification of <file spec 1> enables the duplication of many files at the same time.

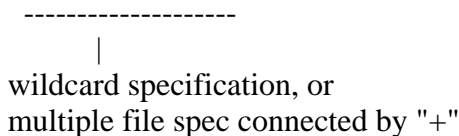
examples:

A>COPY \*.ASM \*.MAC <-- makes files with extension "MAC" from any files with extension "ASM"

A>COPY A\*.\* B:Z\*.\* <-- Any files beginning with the character A are copied to files beginning with the character Z on drive B

### (4) File concatenation

form COPY <multiple file spec> <file name>



When one destination file receives more than one source file, the contents of all source files are concatenated and stored to the specified destination file. When specifying more than one source file, wildcards are available, and file specs can also be copied by using the plus sign.

When files are concatenated, ASCII mode is selected by default and 1AH is considered the file end mark. Thus, concatenating binary files including data 1AH by the COPY command causes data after 1AH to be discarded. To prevent this, specify /B switch and use COPY command in binary mode.

If more than one wildcard appears in the specification of source files, the second wildcard and after are expanded referring to original file names, as in paragraph (3) above. This permits concatenation of similar files at the same time.

examples:

```
A>COPY X+Y+Z XYZ      <-- concatenates X, Y, AND Z and stores
                        in a file XYZ
```

```
A>COPY *.LST ALL      <-- concatenates any files with extension
                        "LST" and stores in a file ALL
```

```
A>COPY /B *.DAT ALL   <-- concatenates any ".DAT" files in
                        binary mode
```

```
A>COPY ASC/A+BIN/B AB/B <-- concatenates an ASCII file ASC and a
                        binary file BIN and stores in a
                        file AB
```

```
A>COPY *.LST+*.REF *.PRN <-- concatenates files named same with
                        extension "LST" and extension "REF"
                        and makes a file with extension "PRN"
```

\* DATE

formDATE [<month>-<day>-<year>]

```
  - -
  | |
  -----
  |
```

"/" and "." are also allowed.

Sets the date in the internal CLOCK-IC. For MSX machines without a CLOCK-IC, it is written to the specific work area. Creations or modifications of files on MSX-DOS cause this date information to be recorded for each file.

When the DATE command is executed without specifying <month>/<day>/<year>, the date currently set is displayed with a request for a new date as shown below. Pressing only the RETURN key here leaves the date unchanged.

```
Current date is <day of week> <month>-<day>-<year>
Enter new date:
```

The format of the date to be set by the DATE command has three fields: <year>, <month>, and <day>. Each field is separated by "-", "/", or ".". Each



field can have the following numerical values:

<year>: 1980 to 2079  
0 to 79 (considered as 2000 to 2079)  
80 to 99 (considered as 1980 to 1999)  
<month>: 1 to 12  
<day>: 1 to 31

Foreign versions of MSX-DOS have different date formats: <month>-<day>-<year>  
or <day>-<month>-<year>.

## \* DEL

formDEL <file spec>  
ERASE is also allowed

Deletes the specified file. Wildcards can be used to specify more than one files.

Since "DEL \*.\*" causes all files on the diskette to be deleted, in this case, an acknowledgement is required.

```
A>DEL *.*  
Are you sure (Y/N)?
```

Pressing "Y" or "y" causes all files to be deleted.

"ERASE" may be used the same way as the DEL command.

## \* DIR

formDIR [<file spec>] [/W] [/P]

The following information about the specified at <file spec> is listed from the left side in one line.

<file name> <file size> <date> <time>

The fields <date> and <time> show when the file was created or last modified. When this information is longer than one line, items displayed near the right side are omitted.

In addition to the usual wildcards, the following abbreviations for <file spec> can be used.

Abbreviation		Formal notation
DIR	=	DIR *.*
DIR <drive>:	=	DIR <drive>: *.*
DIR <filename>	=	DIR <filename>.*
DIR .<extension>	=	DIR *.*.<extension>

When the /W switch is specified, only <filename>s are padded to one line.  
When the /P switch is specified, the listing is stopped after each display page to wait for any key input.

examples:

A>DIR <-- displays information for all files on drive A

A>DIR B: <-- displays information for all files on drive B

A>DIR TEST <-- displays information for all files  
having <filename> "TEST"

A>DIR /W <-- displays all file names of drive A

## \* FORMAT

formFORMAT

Formats a diskette in MSX-DOS format. In other words, directories and FAT are initialised and any files are erased. Since MSX-DOS has the same disk format as MS-DOS, the formatted diskette is also read or written by MS-DOS.

When executing the FORMAT command, an inquiry

Drive name? (A,B)(<-- Depends on number of drives)

is made for the name of the drive containing a disk to be formatted.

Answering "A" or "B" causes the menu to be displayed when a drive that can select one-sided and two-sided formats is being used. After specifying the type of format,

Strike a key when ready

is displayed to wait for a key input. Pressing any key starts formatting. See the disk drive manual for the format menu.

## \* MODE

formMODE <characters per line>

Sets the number of characters to be displayed in one line on the screen. <characters per line> can have a value from 1 to 80 and the screen mode depends on that value:

<characters per line>	Screen mode
1 to 32	GRAPHIC 1 (SCREEN 1)
33 to 40	TEXT 1 (SCREEN 0:WIDTH 40)
41 to 80	TEXT 2 (SCREEN 0:WIDTH 80)

## \* PAUSE

formPAUSE [<comment>]

MSX-DOS has a "batch operation" feature which automatically executes a series of commands written in a text file. During the batch operation, you may want to stop command execution temporarily. One example would be for the user to exchange disks. PAUSE can be used in such cases.

When this command is executed,

Strike a key when ready...

is displayed and a key input is expected. Pressing any key other than Ctrl-C here ends the PAUSE command and proceeds to the next one. Pressing CTRL-C abandons the batch operation. Any kind of comments can follow "PAUSE". This makes it possible to display the purpose of the request for the key input.

#### \* REM

formREM [<comment>]

REM is used to write a comment in the batch command. It does nothing as a command. A space between "REM" and <comment> is required.

#### \* REN

formREN <file spec> <file name>  
RENAME is also allowed

REN changes the file name specified by <file spec>. Wildcards can be used in both <file spec> and <file name>. Specifying wildcards for <file name> causes these wildcards to be replaced with corresponding characters of the <file spec> (see COPY command).

Any attempt to change a file name to a name already in use will cause an error.

examples:

A>REN ABC XYZ <-- changes the file name "ABC" to "XYZ"

A>REN B:ABC XYZ <-- changes the file name "ABC" on drive B to "XYZ"

A>REN \*.BIN \*.COM <-- changes any files with the extension "BIN" to "COM"

#### \* TIME

formTIME [<hour>[:<minute>[:<second>]]]

TIME sets the time for the internal CLOCK-IC. Nothing happens to machines that do not have a CLOCK-IC. When a file is created on MSX-DOS, time information set here is recorded for each file.

Executing the TIME command without specifying the time causes the current time setting to be displayed as shown below. Then there is an input request for a new time. Pressing only the RETURN key does not change the time.

Current time is <hour>:<minute>:<second>:<second/100><p or a>  
Enter new time:

The punctuation mark ":" separates the three TIME command fields of <hour>, <minute>, and <second>. Fields after <minute> or <second> may be omitted or

considered to be 0. Each field can have the following values:

<hour>: 0 to 23  
12A (represents midnight)  
0A to 11A (represents midnight to 11 o'clock  
in the morning)  
12P (represents noon)  
1P to 11P (represents 1 o'clock to 11 o'clock  
in the evening)  
<minute>: 0 to 59  
<second>: 0 to 59

examples:

A>TIME 12 <-- sets time to 12:00:00

A>TIME 1:16P <-- sets time to 13:16:00

\* TYPE

formTYPE <file spec>

The command TYPE displays the contents of a file specified by <file spec>. Using wildcards in <file spec> causes the first of the corresponding files to be displayed. This command is for ASCII files, and displaying binary files causes unreadable control characters to be sent to the screen.

\* VERIFY

formVERIFY [ON|OFF]

VERIFY sets/resets the verify mode. When the verify mode is turned ON, after data is written to the disk, it is always read to ensure that it was written correctly. This is why disk access takes longer. "VERIFY OFF" is set by default.

## 2.3 Batch Command Usage

MSX-DOS has a batch feature that allows a series of commands listed in the order of operation to be executed automatically. The file containing this procedure is called a "batch file" and the series of operations defined by a batch file is called a "batch command".

A batch file uses the extension ".BAT". Typing only the file name (the extension ".bat" is not typed) at the command line prompt causes MSX-DOS to execute the commands in the file line by line.

For example, let us consider the following operation:

1. Copy all files on drive A with the extension ".COM" onto drive B.
2. List all "COM" files on drive B.
3. Delete all "COM" files on drive A.

This operation could be achieved by issuing the following commands to

MSX-DOS:

```
A>COPY A:*.COM B:
A>DIR B:*.COM /W
A>DEL A:*.COM
```

If these three lines are combined into a batch file called "MV.BAT", the command line input "MV" will automatically execute the operation shown above. The following list illustrates this.

```
A>COPY CON MV.BAT -+
COPY A:*.COM B: | creates "MV.BAT"
DIR B:*.COM /W |
DEL A:*.COM -+
^Z          Ctrl-Z + RETURN key input
```

```
A>TYPE MV.BAT -+
COPY A:*.COM B: | to confirm the contents of "MV.BAT"
DIR B:*.COM /W |
DEL A:*.COM -+
```

```
A>MV          invokes the batch command "MV"
A>COPY A:*.COM B: reads the first line automatically and executes it
.
.
.
A>DIR B:*.COM /W reads the second line automatically and executes it
.
.
.
A>DEL A:*.COM reads the third line automatically and executes it
.
.
.
```

A batch operation may be interrupted by pressing Ctrl-C. When Ctrl-C is entered during batch operations, the request shown in Figure 3.7 is displayed on the screen.

Figure 3.7 Interrupt of the batch operation

```
-----
|                               |
| Terminate batch file (Y/N)? |
|                               |
-----
```

Selecting "Y" here terminates the batch command and returns to MSX-DOS. Selecting "N" reads the next line of the batch file and continues the execution of the batch command.

#### \* Batch variables

For more flexible use of the batch command, any string can be passed as parameters from the command line to the batch command. Parameters passed are

referred to with the symbols "%n" where n is any number from 0 to 9. These "%n" symbols are called batch variables.

Batch variables %1, %2, ... correspond to parameters specified in the command line from left to right, and %0 is for the name of the batch command itself.

Figure 3.8 Examples for batch variables usage

```
-----  
|  
| A>COPY CON TEST.BAT ..... creates a batch command |  
| REM %0 %1 %2 %3 |  
| ^Z |  
| 1 file copied |  
| A>TYPE TEST.BAT |  
| REM %0 %1 %2 %3 ..... a batch command to display 3 arguments |  
|  
| A>TEST ONE TWO THREE FOUR ..... executes the batch command, |  
| A>REM TEST ONE TWO THREE giving arguments to it |  
| A> |  
|  
-----
```

#### \* AUTOEXEC.BAT

The batch file named "AUTOEXEC.BAT" is used as a special autostart program at MSX-DOS startup. When MSX-DOS is invoked, COMMAND.COM examines whether AUTOEXEC.BAT exists and, if so, executes it.

### 2.4 External Commands

External commands exist on the diskette as files with the extension ".COM", and typing the external command name (except for the extension) causes the command to be executed in the following manner.

1. loads an external command after 100H
2. calls 100H

#### \* Developing external commands

Assembly language routines created to work in memory at location 100H and saved under file names with the extension ".COM" are called external commands and can be executed from MSX-DOS.

For example, consider a program to produce a control code "0CH" by using one-character output routine (see system calls) and clear the screen. This is an 8-byte program as shown below.

#### List 3.1 Contents of CLS.COM

```
=====
```

```
1E 0C      LD  E,0CH      ; E := control-code of CLS
```

```
0F 02      LD  C,02H      ; C := function No. of CONSOLE OUTPUT
CD 05 00CALL 0005H      ; call BDOS
C9      RET
```

---

Writing these 8 bytes to a file named CLS.COM produces the external command "CLS" to clear the screen. The following sample program uses the sequential file access feature of BASIC to make this command. After this program is run, the CLS command is created on the diskette. Confirm that the command actually works after returning to MSX-DOS.

### List 3.2 Creating CLS.COM

---

```
100 '***** This program makes "CLS.COM" *****
110 '
120 OPEN "CLS.COM" FOR OUTPUT AS #1
130 '
140 FOR I=1 TO 8
150  READ D$
160  PRINT #1,CHR$(VAL("&H"+D$));
170 NEXT
180 '
190 DATA 1E,0C,0E,02,CD,05,00,C9
```

---

#### \* Passing arguments to an external command

When creating an external command, there are two ways to pass arguments from the command line to the external command. First, when passing the file names to the command line as arguments, use 5CH and 6CH in the system scratch area. COMMAND.COM, which always considers the first and second parameters as file names when external commands are executed, expands them to a drive number (1 byte) + file name (8 bytes) + extension (3 bytes) and stores them in 5CH and 6CH. These are in the same format as the first 12 bytes of FCB, so setting these address as first addresses of FCB permits various operations.

However, since in this method only 16 bytes differ from the starting addresses of two FCBs, either 5CH or 6CH (only) can be used as a complete FCB. Next, when passing arguments other than file names (numbers, for instance) or creating an external command handling more than three file names, COMMAND.COM stores the whole command line, which invoked the external command, except for the command line itself in the form of number of bytes (1 byte) + command line body, so it can be used by interpreting it in the external command properly. See list 3.3 of section 4 for an example of passing arguments using this DMA area.

### 3. STRUCTURE OF DISK FILES

Information about the structure of data on the disk and how it is controlled

is important when accessing the disk using system calls. This section begins with a description about "logical sectors" which are the basic units for exchanging data with the disk on MSX-DOS, and proceeds to the method of handling data with "files" which is more familiar to programmers.

### 3.1 Data units on the disk

#### \* Sectors

MSX-DOS can access most types of disk drives including the 3.5 inch 2DD and hard disks. For handling different media in the same way, the system call consider "logical sector" as the basic units of data on the disk. A logical sector is specified by numbers starting from 0.

#### \* Clusters

As long as system calls are used, a sector may be considered the basic unit of data as considered above. In fact, however, data on the disk is controlled in units of "clusters" which consists of multiple sectors. As described later in the FAT section, a cluster is specified by a serial number from 2 and the top of the data area corresponds to the location of cluster #2. For getting information about how many sectors a cluster has, use the system call function 1BH (acquiring disk information).

#### \* Conversion from clusters to sectors

In a part of the directory or FCB, described later, the data location on the disk is indicated by the cluster. To use system calls to access data indicated by cluster, the relation of the correspondence between the cluster and the sector needs to be calculated. Since cluster #2 and the top sector of the data area reside in the same location, this can be done as follows:

1. Assume the given cluster number is C.
2. Examine the top sector of the data area (by reading DPB) and assume it is S0.
3. Examine the number of sectors equivalent to one cluster (using function 1BH) and assume it is n.
4. Use the formula  $S = S0 + (C-2) * n$  to calculate sector numbers.

In MSX-DOS, sectors in the disk are divided into four areas, as shown in Table 3.9. The file data body written to the disk is recorded in the "data area" portion. Information for handling data is written in three areas. Figure 3.9 shows the relation of the locations of these areas. The boot sector is always in sector #0, but the top sectors (FAT, directory, and data area) differ by media, so DPB should be referred to.

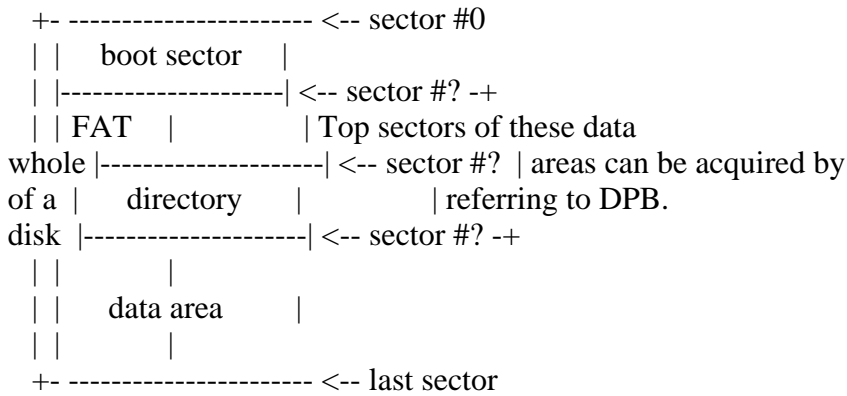
Table 3.9 Disk contents

---

boot sector	MSX-DOS startup program and information proper to the disk
FAT	physical control information of data on the disk
directory	control information of files on the disk
data area	actual file data



Figure 3.9 Relation of locations of elements in the disk

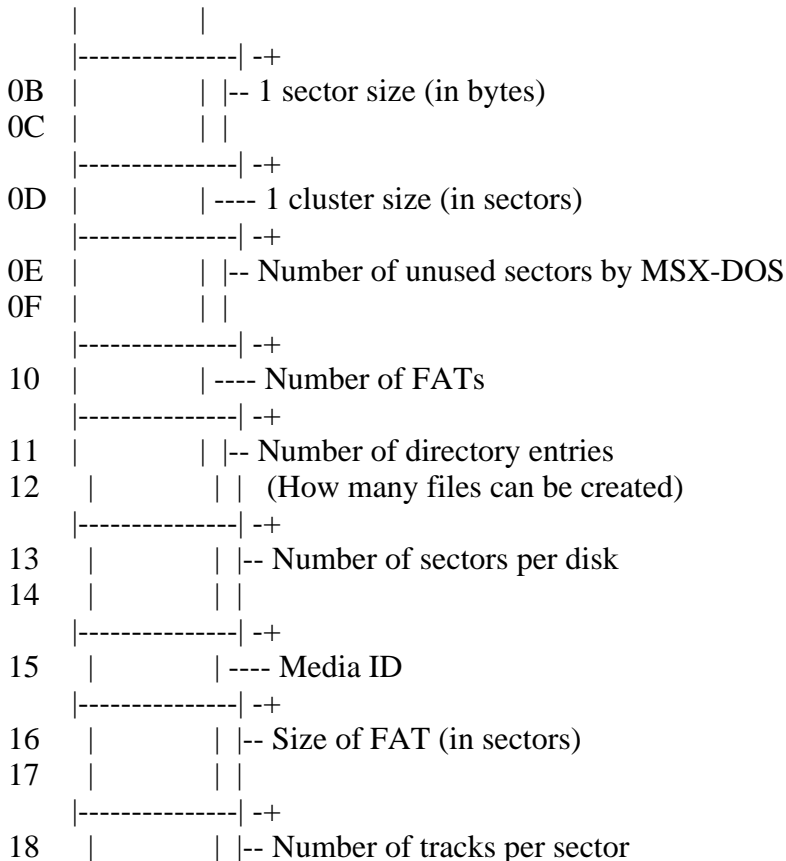


\* DPB (drive parameter block) and boot sector

On MSX-DOS, the area "DPB" is allocated in the work area of memory for each connected drive, and information proper to each drive is recorded there. MSX-DOS can handle most types of disk drives, because the differences between media can be compensated for by the process corresponding to each drive.

Information written on DPB, which is originally on the boot sector (sector #0) of the disk, is read at MSX-DOS startup. Note that the differences between the contents of the boot sector and DPB, as shown in Figures 3.10 and 3.11. Data is arranged differently in the boot sector and the DPB.

Figure 3.10 Information of the boot sector



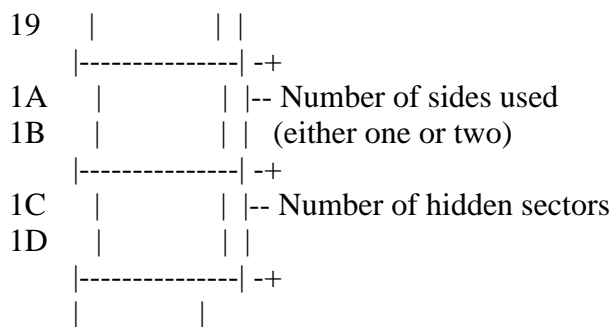
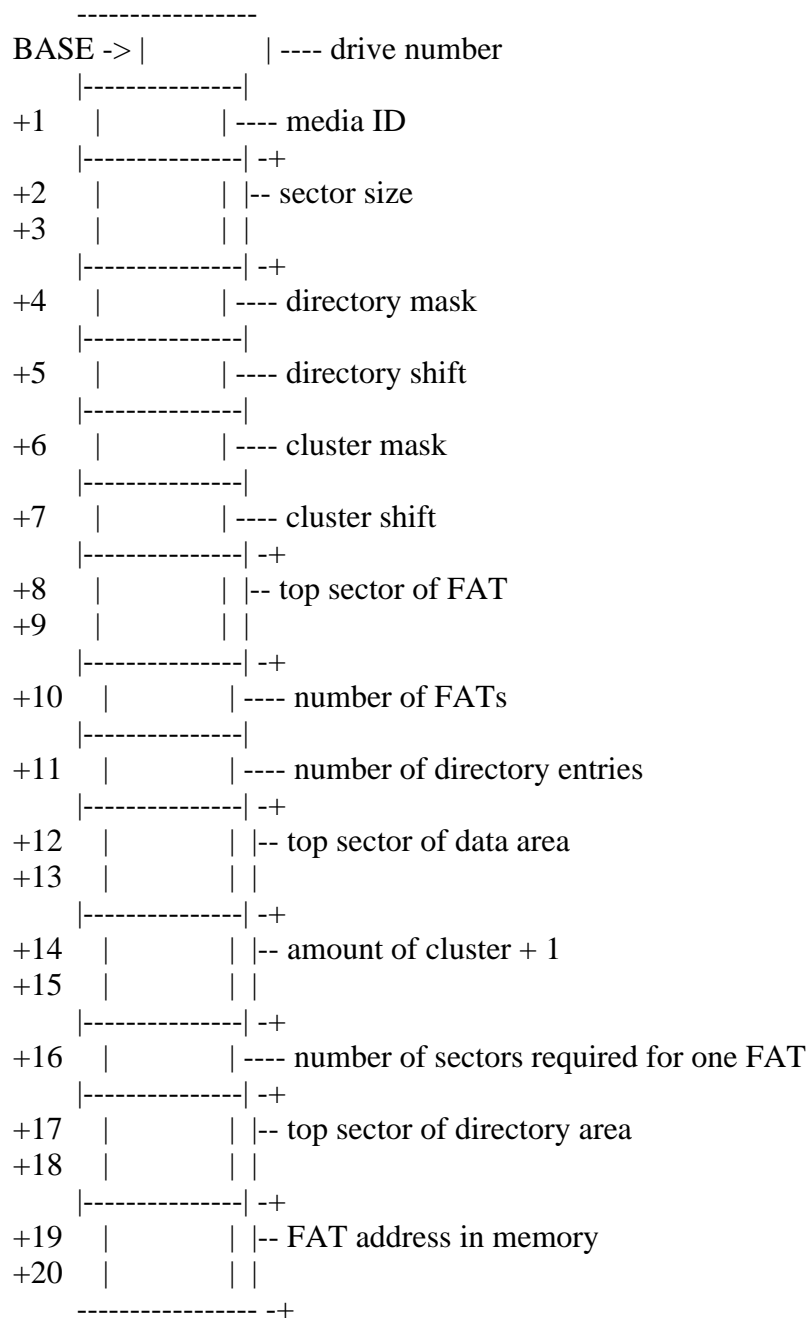


Figure 3.11 DPB structure



Use the system call Function 1BH (disk information acquisition) to access the DPB. This system call returns the DPB address in memory and other information for each drive written on the boot sector (see section 4 "System call usage" for the detailed usage).

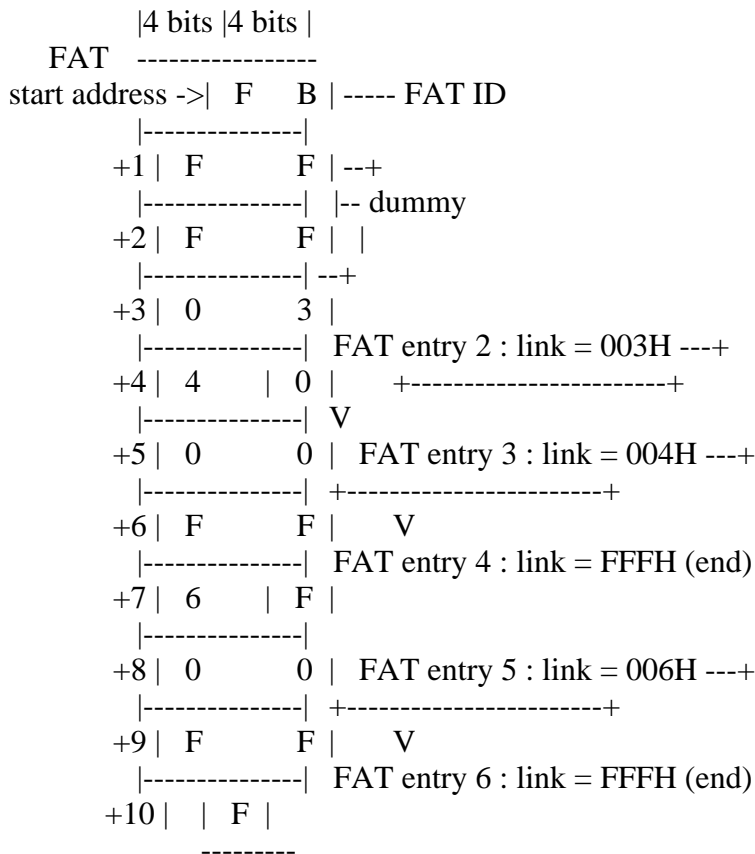
\* FAT (file allocation table)

In MSX-DOS, a "cluster" is the data unit for writing to the disk. Files larger than a cluster are written across multiple clusters. But in this case adjacent clusters are not always used. In particular, after creating and deleting files many times, clusters which are no longer used are scattered at random across the disk. When a large file is created for such cases, the file is broken down into several clusters and these clusters are stored where space is available. The linkage information is kept at the beginning so that the file can be recreated. This is the main function of the FAT.

When a bad cluster is found, FAT is also used to record that location, so access will not be made there any more. Linkage information of clusters and information concerning bad clusters is necessary for managing disk files. Without this information, the whole disk will be unusable. For this reason, more than one FAT is always prepared in case of accidental erasure.

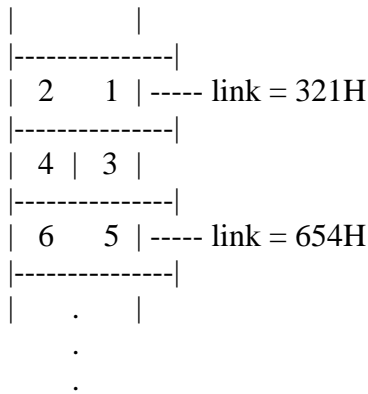
Figure 3.2 shows an example of a FAT. The first byte is called the "FAT ID" which contains the value indicating the type of media (the same value as media ID in Table 3.2). The next two bytes contains meaningless dummy values. From the fourth byte (start address + 3), actual linkage information is recorded in an irregular format of 12 bits per cluster. Each 12-bit area containing linkage information is called a FAT entry. Note that the FAT entry begins with number 2. The number of the FAT entry is also the number of the cluster corresponding to it. Read the 12-bit linkage information recorded in the FAT entry in the way shown in Figure 3.13.

Figure 3.12 FAT example



The linkage information is the value indicating the next cluster number. FFFH means that the file ends with that cluster. The example of Figure 3.12 shows a file of three clusters, (cluster #2 -> cluster #3 -> cluster #4), and a file of two clusters, (cluster #5 -> cluster #6). The linkage from the cluster with the smaller number is only for easy comprehension. In actual practice, numbers are not necessarily ordered.

Figure 3.13 Reading FAT



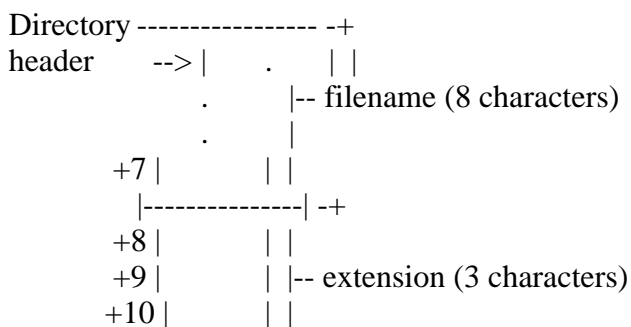
\* Directory

The FAT as described above, relates the physical location of data on the disk and does not include information about the contents of data written there. Thus, an information resource other than FAT is required to know what kind of data is in a file. This resource is called a "directory". A directory entry is composed of 32 bytes and records file names, file attributes, date created, time created, number of the top cluster of the file, and file size, as shown in Figure 3.14.

"File attributes" in the directory are used for specifying the invisibility attribute in a file. Specifying "1" in the second bit from the lowest of this byte prevents files specified in the directory from being accessed by the system call (see Figure 3.15). MS-DOS also has a file attribute byte which permits a write-prohibit attribute using another bit, but MSX-DOS does not support this feature.

The date and time are recorded so that two bytes of each are divided into three bitfields, as shown in Figure 3.16 and Figure 3.17. Since only 5 bits are prepared for the "time" bitfield, the minimum unit for time is two seconds. The year (1980 to 2079) is specified by using 0 to 99 in 7 bits.

Figure 3.14 Directory construction



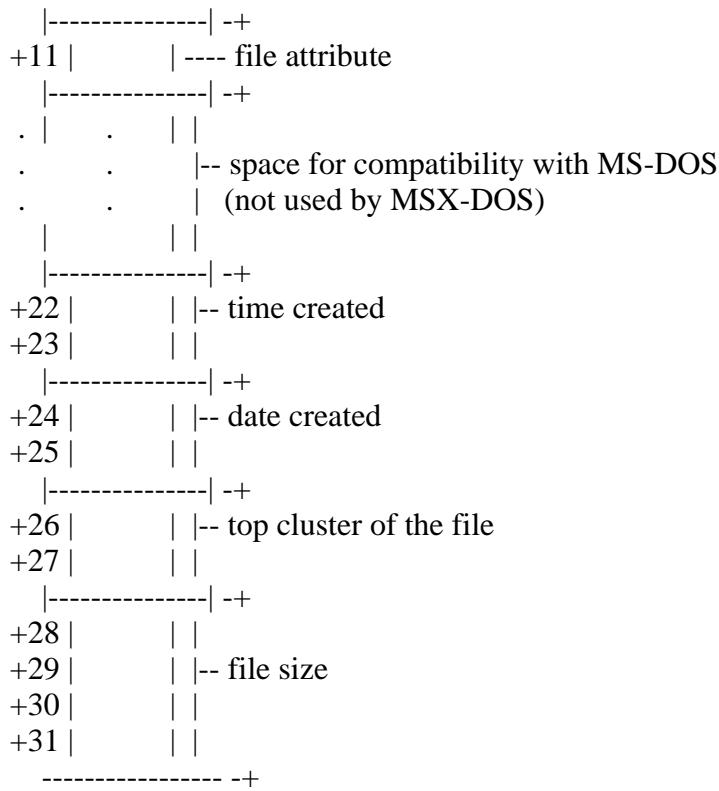


Figure 3.15 Invisibility attribute of the file

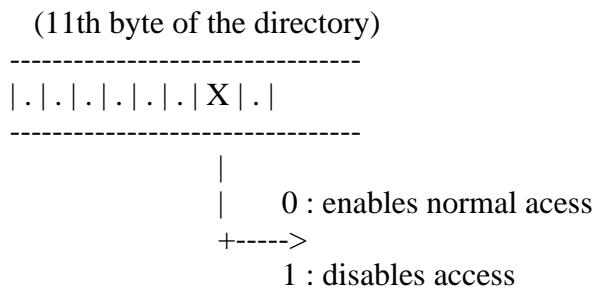


Figure 3.16 Bitfield representing time

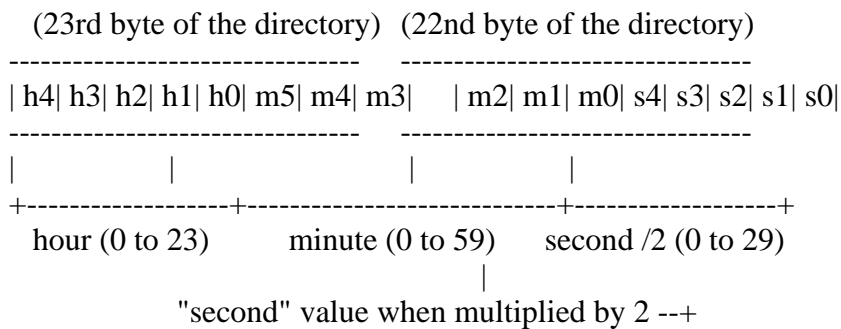
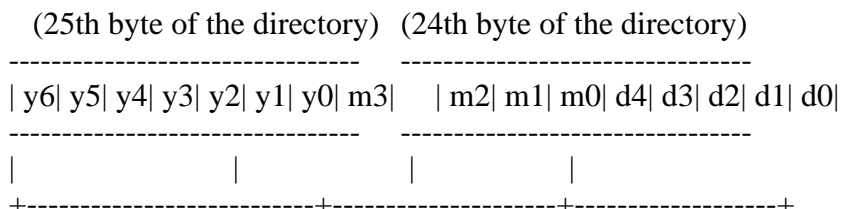


Figure 3.17 Bitfield representing date

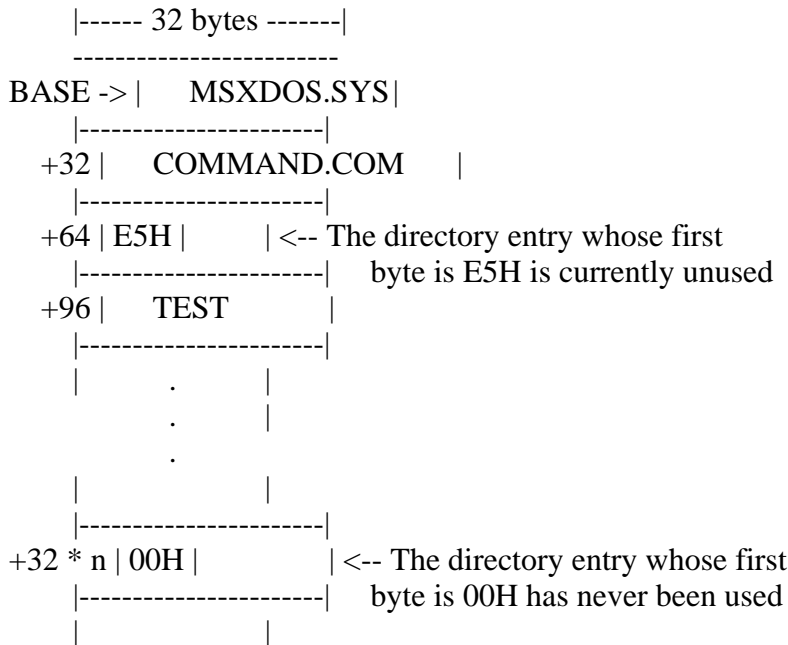


year (0 to 99)      month (1 to 12)      day (1 to 31)

|  
+-- corresponds to 1980 to 2079

The place where this directory information is actually recorded is the directory area on the disk (see Figure 3.9). The location (top sector) is recorded in the DPB. Directory entries (locations of directory storage) are arranged every 32 bytes in the directory area, as shown in Figure 3.18. When a file is created, the directory is created at the lowest value of unused directory entries. Deleting a file causes E5H to be written to the first byte of the corresponding directory entry, which is empty. After all directory entries are exhausted, new files cannot be created even if there is a lot of unused space on the disk. The number of directory entries, that is, the number of files which can be created on one disk is also recorded in the DPB.

Figure 3.18 Organisation of directory area



### 3.2 File Access

\* FCB (file control block)

Using information recorded in the directory area allows data to be treated as a "file". The advantage of this method is that the data location is not represented by an absolute number such as sector number or cluster number; instead, the file can be specified with a "name". The programmer need only specify the file name and the system will do all the work concerned with accessing the requested file. In other words, the programmer need not understand the details of which sectors the file occupies. In this case, FCB plays an important role for directories.

FCB is the area for storing information needed to handle files using system calls. Handling one file requires 37 bytes of memory each, as shown in Figure 3.19. Although the FCB can be located anywhere in memory, the address 005CH is normally used to utilize MSX-DOS features.

Figure 3.19 Organization of FCB

```

FCB -----
bytes      | 0 | drive number
from|----|
top  | 1 | file name
      |   | filename ..... 8 bytes
V    | 11 | extension .... 3 bytes
      |----|
      | 12 | current block
      | 13 | number of blocks from the top of the file to the
      |----| current block
      | 14 | record size
      | 15 | 1 to 65535
      |----|
      | 16 | file size
      |   | 1 to 4294967296
      | 19 |
+---|----|
      | 20 | date
      | 21 | same form as directory
(1) |----|
      | 22 | time
      | 23 | same form as directory
+---|----|
      | 24 | device ID
      |----|
      | 25 | directory location
      |----|
      | 26 | top cluster number of the file
(2) | 27 |
      |----|
      | 28 | last cluster number accessed
      | 29 |
      |----|
      | 30 | relative location from top cluster of the file
      | 31 | number of clusters from top of the file
+---|----| to the last cluster accessed
      | 32 | current record
      |----|
      | 33 | random record
      |   | record order from the top of the file
      | 36 | usually stores the last record made random access
-----

```

Notes: FCB usages differ, depending on whether they use CP/M compatible system calls or additional system calls. See the description below for details.

(1) When using version 2 of MSX-DOS, here is stored the volume-id of the disk, and should not be modified by the program.

(2) When using version 2 of MSX-DOS, here is stored internal information relative to the physical location of the file on the disk. The format of this information is different from shown in figure 3.19, and should not be modified by the program.

\* drive number (00H)

Indicates the disk drive containing the file.

(0 -> default drive, 1 -> A:, 2 -> B:...)

\* filename (01H to 08H)

A filename can have up to 8 characters. When it has less than 8, the rest are filled in by spaces (20H).

\* extension (09H to 0BH)

A extension can have up to 8 characters. When it has less than 3, the rest are filled in by spaces (20H).

\* current block (0CH to 0DH)

Indicates the block number currently being referred to by sequential access (see function 14H, 15H in section 4).

\* record size (0EH to 0FH)

Specifies the size of data unit (record) to be read or written at one access, in bytes (see function 14H, 15H, 21H, 27H, 28H).

\* file size (10H to 13H)

Indicates the size of the file in bytes.

\* date (14H to 15H)

Indicates date when a file was last written. The format is the same as the one recorded in the directory.

\* time (16H to 17H)

Indicates time when a file was last written. The format is the same as the one recorded in the directory.

\* device ID (18H)

When a peripheral is opened as a file, the value shown in Table 3.10 is specified for this device ID field. For normal disk files, the value of this field is 40H + drive number. For example, the device ID for drive A is 40H (for future expansion, application programs should not use the ID byte).

Table 3.10 Device ID

```

-----
| Device name  | Device ID |
|-----+-----|
| CON (Console) | 0FFH  |
| PRN (Printer) | 0FBH  |
| LST (List=Printer) | 0FCH  |
| AUX (Auxiliary) | 0FEH  |
| NUL (Null)    | 0FDH  |
-----

```

\* directory location (19H)

Indicates the order of the directory entries of a file in the directory area.

\* top cluster (1AH to 1BH)

Indicates the top cluster of the file in the disk.



\* last cluster accessed (1C to 1DH)

Indicates the last cluster accessed.

\* relative location from top cluster of last cluster accessed (1EH to 1FH)

Indicates the relative location from the top cluster of the last cluster accessed.

\* current record (20H)

Indicates the record currently being referred to by sequential access (see function 14H, 15H).

\* random record (21H to 24H)

Specifies a record to be accessed by random access or random block access.

Specifying a value from 1 to 63 for the record size field described above causes all four bytes from 21H to 24H to be used, where only three bytes from 21H to 23H have meaning when the record size is greater than 63 (see function 14H, 15H, 21H, 22H, 27H, 28H).

### \* Opening a file

A special procedure is required to open a file when using FCB. "Opening a file" means, at the system call level, transforming the incomplete FCB whose file name field is only defined for the complete FCB, by using information written in the directory area. Figure 3.20 shows the differences between "unopened FCB" and "opened FCB".

Figure 3.20 Before/after opening FCB

before the open	after the open
-----	-----
0   S   drive number	0   S   default drive (00H) is converted
1   S   -----	1   S   to real drive (01H to 06H)
2   S   ^	2   S
3   S	3   S
4   S	4   S
5   S	5   S
6   S   file name	6   S
7   S	7   S
8   S	8   S
9   S	9   S
10   S   V	10   S
11   S   -----	11   S
12	12     \current block
13	13     /
14	14     \record size
15	15     /
16	16   S   ---+
17	17   S     file
18	18   S     size
19	19   S   ---+
20	20   S   \ date
21	21   S   /
22	22   S   \ time
23	23   S   /

24			24		S		device ID
25			25		S		directory location
26			26		S		\ top cluster number number of the file
27			27		S		/
28			28		S		\ last cluster number accessed
29			29		S		/
30			30		S		\ relative location from top
31			31		S		/ cluster of the file
32			32				current record
33			33				--+
34			34				random
35			35				record
36			36				--+
-----			-----				

\* Closing a file

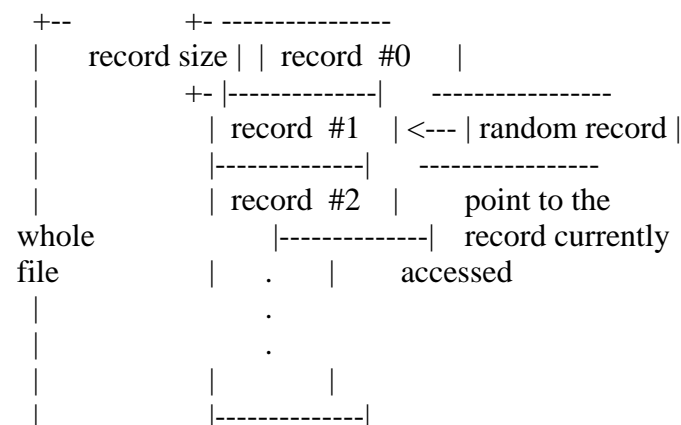
When a file is opened and written to, the contents of each field of FCB, such as size, is also modified. Unless the updated FCB information is returned to the directory area, directory information and the actual contents of the file might be different at the next file access. This operation to return the updated FCB information to the directory corresponds to closing a file at the system call level.

\* Random block access (file management by records)

MSX-DOS has two system calls dealing with random access, "RANDOM BLOCK READ" and "RANDOM BLOCK WRITE". With these system calls, a file can be divided into data units of any size, which can be handled by numbers, such as 0, 1, 2, ..., from the top. This data unit is called a "record". Record size can be any value of more than one byte. So, treating a whole file as one record (extreme sequential access), treating data with one byte as one record (extreme random access), or treating 128 bytes as one record (the CP/M way) are all possible.

In this case, the FCB fields, "record size" and "random record" are used to specify the record. The value of the record size field is the number of bytes in one record. Random record fields can have any record number to be accessed (for more detailed usage, see descriptions of each system call).

Figure 3.21 File and record



```

|           | record #n |
+---+-----+

```

\* Sequential access (file management by fixed-length record + current record + current block)

MSX-DOS can also access files the same way as CP/M for purposes of compatibility. One way is the sequential file which is managed by "current record" and "current block". This uses a 128-byte fixed-length record as the basic unit of data. File access is always done from the top sequentially and the number of records which was accessed is counted at the current record field of FCB. The value of the current record field is reset to 0 when it reaches 128, and the carry is counted in the current block field.

\* Random access (file management by fixed-length record + random record)

A second method included to keep compatibility with CP/M is a random access method using random record fields. It can access the record of any location but the record size is fixed at 128 bytes.

#### 4. SYSTEM CALL USAGE

The system calls are a collection of general-purpose subroutines which handle the basic input/output operations of MSX-DOS. Having these system calls gathered into BIOS in a predefined manner permits the basic functions of the MSX disk system to be easily accessed.

There are two purposes of system calls; first, to reduce programming time by preprogramming basic functions; second, to increase portability by the fact that all programs share the same basic functions. Utilizing system calls shortens program development time and makes the developed program highly portable.

To execute a system call, enter the defined function number in the C register of the Z80 CPU and call one of the following addresses:

```

0005H ..... MSX-DOS
F37DH (&HF37D) ..... MSX DISK-BASIC

```

For example, when the function number is 01FH and the system call requires 00H to be set in the A register, the following assembler code can be used with MSX-DOS:

```

LD  A,00H
LD  C,01FH
CALL 0005H
.
.
.

```

The CALL statement is also used in operations that return values or restore registers from memory. System calls can also be used from DISK-BASIC by using the entry address of F37DH. For this case, store the machine codes in the area allocated by the CLEAR statement and call its start address using the

USR function.

\* System call format

This section introduces system call usages in the following notation:

```
-----  
| Function:   function number           |  
| Setup:    value needed to be set in register or memory by programmer |  
| Return value: value set in register by system call           |  
-----
```

Function:

The function number is used to identify the system call. When using a system call, set the function number in the C register.

Setup:

In this section, "setup:" indicates the value to be set in the named register or memory location before executing system calls.

Return value:

The result obtained by a system call is normally set in a register or memory location. This is called output in this section and "return value:" indicates where and how this output is set.

Is important to note that when using system calls, the contents of registers other than those specified are sometimes destroyed. So, before using system calls, store the contents of registers whose value you do not want to change in an appropriate place (stack, for example) before executing system calls.

There are forty-two MSX system calls. These are listed in Table 3.11, and are described in this section. There are four categories:

- \* Peripheral I/O
- \* Environment setting
- \* Absolute READ/WRITE (direct access to sector)
- \* File access using FCB

Table 3.11 List of System Calls

Function no.	Function
00H	system reset
01H	get one character from console (input wait, echo back, control code check)
02H	send one character to console
03H	get one character from auxiliary device
04H	send one character to auxiliary device
05H	send one character to printer

06H	get one character from console (no input wait, no echo back, no control code check)/ one character output
07H	get one character from console (input wait, no echo back, no control code check)
08H	get one character from console (input wait, no echo back, control code check)
09H	send string
0AH	get string
0BH	check input from console
0CH	get version number
0DH	disk reset
0EH	select default drive
0FH	open file
10H	close file
11H	search the first file matched with wildcard
12H	search the second and after the second file matched wildcard
13H	delete file
14H	read sequential file
15H	write sequential file
16H	create file
17H	rename file
18H	get login vector
19H	get default drive name
1AH	set DMA address
1BH	get disk information
1CH-20H	no function
21H	write random file
22H	read random file
23H	get file size
24H	set random record field
25H	no function
26H	write random block
27H	read random block
28H	write random file (00H is set to unused portion)
29H	no function
2AH	get date
2BH	set date
2CH	get time
2DH	set time
2EH	set verify flag
2FH	read logical sector
30H	write logical sector

\* Note

System call function numbers are from 00H to 30H; the following seven numbers are blank:

1CH to 20H, 25H, 29H

Calling these blank function system calls do nothing except setting 00H in the A register. System calls after function 31H are undefined. Using them may cause unpredictable results (not advisable).

### List 3.3 Utility routines

---

```
*****
;
; List 3.3 utility.mac
;
; these routines are used in other programs
;
; GETARG, STOHEX, PUTHEX, PUTCHR, DUMP8B
;
*****
;
; PUBLIC GETARG          Note: Five utility routines included in
; PUBLIC STOHEX          this program list will be used in
; PUBLIC PUTHEX          sample programs later.
; PUBLIC PUTCHR
; PUBLIC DUMP8B

BDOS: EQU    0005H
DMA:  EQU    0080H

;----- DE := address of arg(A)'s copy -----

GETARG: PUSH  AF          Note: Nth parameter (N is specified by
; PUSH  BC              A register) of the command line
; PUSH  HL              stored in default DMA area
;                       (0080H to ) is loaded in memory and
; LD  C,A              its starting address is returned in
; LD  HL,DMA           DE register.
; LD  B,(HL)
; INC HL
; INC B

SKPARG: DEC  B
; JR  NZ,NOARG
SKP1:  LD  A,(HL)
; INC HL
; CALL TERMCHK
; JR  NZ,SKP1
SKP2:  LD  A,(HL)
; INC HL
; CALL TRMCHK
; JR  Z,SKP2
; DEC  HL
; DEC  C
; JR  NZ,SKPARG

CPYARG: LDDE,BUFMEM
CPY1:  LD  A,(HL)
; LD  (DE),A
; INC HL
; INC DE
; CALL TRMCHK
; JR  NZ,CPY1

; DEC  DE
```

```
LD A,"$"
LD (DE),A
LD DE,BUFMEM
JR EXIT
```

```
NOARG: LD DE,BUFMEM
LD A,"$"
LD (DE),A
```

```
EXIT: POP HL
POP BC
POP AF
RET
```

```
TRMCHK: CP 09H
RETZ
CP 0DH
RETZ
CP " "
RETZ
CP ";"
RET
```

;----- HL := hexadecimal value of [DE] -----

```
SOTHEX: PUSH AF
PUSH DE
LD HL,0000H
CALL STOHI
POP DE
POP AF
RET
```

Note: Hexadecimal string indicated by DE register is converted into two-byte integer and stored in HL register.

```
STOHI: LD A,(DE)
INC DE
SUB"0"
RETC
CP 10
JR C,STOH2
SUB"A"- "0"
RETC
CP 6
RETNC
ADD A,10
```

```
STOH2: ADD HL,HL
ADD HL,HL
ADD HL,HL
ADD HL,HL
OR L
LD L,A
JR STOHI
```

;----- print A-reg, in hexadecimal form (00-FF) -----

```
PUTHEX: PUSH AF
RR A
```

Note: Contents of A register is displayed using two hexadecimal digits.

```

RR A
RR A
RR A
CALL PUTHX1
POP AF
PUTHX1: PUSH AF
AND 0FH
CP 10
JR C,PUTHX2
ADD A,"A"-10-"0"
PUTHX2: ADD A,"0"
CALL PUTCHR
POP AF
RET

```

;----- put character -----

```

PUTCHR: PUSH AF
PUSH BC
PUSH DE
PUSH HL
LD E,A
LD C,02H
CALL BDOS
POP HL
POP DE
POP BC
POP AF
RET

```

;----- dumps 8bytes of [HL] to [HL+7] in hexa & ASCII form -----

```

DUMP8B: PUSH HL      Note: Contents of eight bytes after the
LD B,8              address indicated in HL register
DUMP1: LD A,(HL)    are dumped in both hexadecimal
INC HL              and character codes.
CALL PUTHX1
LD A," "
CALL PUTCHR
DJNZ DUMP1
POP HL
LD B,8
DUMP2: LD A,(HL)
INC HL
CP 20H
JR C,DUMP3
CP 7FH
JR NZ,DUMP4
DUMP3: LD A,"."
DUMP4: CALL PUTCHR
DJNZ DUMP2
LD A,0DH
CALL PUTCHR
LD A,0AH
CALL PUTCHR
RET

```



;----- work area -----

BUFMEM: DS 256

END

---

## 4.1 Peripheral I/O

The following system calls are intended for input/output operations. Some examples include console I/O (screen/keyboard), auxiliary I/O (external input/output), and printer I/O. Since subroutines such as getting information from the keyboard or controlling printers are necessary for most programs, you will find the system calls described in this section useful for general programming.

### \* Console input

Function: 01H

Setup: none

Return value: A register <-- one character from console

When there is no input (no key pressed and input buffer empty), an input is wait for. Input characters are echoed back to the console. The following control character input is allowed: Ctrl-C causes program execution to be halted and a return to the MSX-DOS command level; Ctrl-P causes any successive input to also echoed to the printer until Ctrl-N is accepted; Ctrl-S causes the display to stop until any key is pressed.

Ctrl-C ..... system reset

Ctrl-P ..... printer echo

Ctrl-N ..... halt printer echo

Ctrl-S ..... pause display

### \* Console output

Function: 02H

Setup: E register <-- character code to be sent out

Return value: none

This system call displays the character specified by the E register on the screen. It also checks the four control characters, listed above.

### \* External input

Function: 03H

Setup: none

Return value: A register <-- one character from AUX device

This system call checks four control characters.

\* External output

Function: 04H

Setup: E register <-- character code to send to AUX device

Return value: none

This system call checks four control characters.

\* Printer output

Function: 05H

Setup: A register <-- one character from console

This system call does not echo back. It treats control characters in the same way as function 01H.

\* Direct console input/output

Function: 06H

Setup: E register <-- character code to be send to the console  
When 0FFH is specified, the character will be input from the console.

Return value: When the E register is set to 0FFH (input), the result of input is in the A register. The value set in the A register is the character code of the key, if it was pressed; otherwise, the value is 00H. When the E register is set to a value other than 0FFH (output), there is no return value.

This system call does not support control characters and does not echo back input. This system call checks four control characters.

\* Direct console input - 1

Function: 07H

Setup: none

Return value: A register <-- one character from console

This system call does not support control characters, nor echo back.

\* Direct console input - 2

Function: 08H

Setup: none

Return value: A register <-- one character from console

This system call does not echo back. It treats control characters in the same way as function 01H.

\* String output

Function: 09H

Setup: DE register <-- starting address of string, prepared on

memory, to be sent to the console.

Return value: none

24H ("\$\$") is appended to the end of the string as the end symbol. This system call checks and performs four control character functions, as listed previously.

\* String input

Function: 0AH

Setup: The address of memory where the maximum number of input characters (1 to 0FFH) is set should be set in the DE register.

Return value: Number of characters actually sent from console is set in the address, one added to the address indicated by the DE register; string sent from console is set in the area from the address, two added to the address indicated by the DE register.

Return key input is considered as the end of console input. However, when the number of input characters exceeds the specified number of characters (contents indicated by DE register, 1 to 255), characters within the specified number of characters will be treated as an input string and set in memory, and the operation ends. The rest of characters including the return key are ignored. Editing with the template is available to string input using this system call. This system call checks and performs four control character function, as listed previously.

\* Console status check

Function: 0BH

Setup: none

Return value: 0FFH is set in the A register when the keyboard is being pressed; otherwise, 00H is set.

This system call checks and performs four control character function, as listed previously.

## 4.2 Environment Setting and Readout

The following system calls set the MSX system environment; for example, changing the default drive, or setting various default values of the system

\* System reset

Function: 00H

Setup: none

Return value: none

When this is called in MSX-DOS, the system is reset by jumping to 0000H. When MSX DISK-BASIC call this, it is "warm started". That is, it returns to BASIC command level without destroying programs currently loaded.

\* Version number acquisition

Function: 0CH

Setup: none

Return value: HL register <-- 0022H

This system call is for acquiring various CP/M version numbers, on MSX-DOS, however, 0022H is always returned.

\* Disk reset

Function: 0DH

Setup: none

Return value: none

If there is a sector which has been changed but not written to the disk, this system call writes it to the disk, then it sets the default drive to drive A and sets DMA to 0080H.

\* Default drive setting

Function: 0EH

Setup: E register <-- default drive number (A = 00H, B = 01H, ...)

Return value: none

Disk access by the system calls are made to the drive indicated by the default drive number, unless otherwise specified. Note that, when the drive number, which is set in the FCB specified upon calling the system call, is other than 00H, the default drive setting made by this system call is ignored.

\* Login vector acquisition

Function: 18H

Setup: none

Return value: HL register <-- online drive information

The online drive is the drive connected to MSX normally when the disk system is booted up. Executing this system call causes each drive to be examined whether it is online, and the result is returned in the HL register as shown in Figure 3.22. When the bit is "1", the corresponding drive is online; otherwise it is not.

Figure 3.22 Login vector

register name	H								L							
bit number	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
drive name	meaningless on MSX-DOS								H	G	F	E	D	C	B	A
online/offline	1 is set for online and 0 for offline in each bit															

-----

\* Default drive acquisition

Function: 19H

Setup: none

Return value: A register <-- default drive number (A = 00H, B = 01H, ...)

\* Setting of address to be transferred to

Function: 1AH

Setup: DE register <-- address setting to be transferred to  
(DMA address)

Return value: none

Though DMA address is initialized to 0080H at system reset, it can be reset to any address by using this system call.

\* Disk information acquisition

Function: 1BH

Setup: E register <-- number of the objective drive  
(default drive = 00H, A = 01H, B = 02H, ...)

Return value: A register <-- number of logical sectors per one cluster  
(FFH if E register is set inappropriate)

BC register <-- logical sector size

DE register <-- amount of clusters

IX register <-- DPB starting address

IY register <-- FAT starting address on memory

This system call gets the information about the disk in the specified drive. Specifying 00H for the drive number specifies the default drive. For other than that, specify 01H for drive A, 02H for drive B, and so on.

This system call has been created for MSX-DOS and is not compatible with CP/M.

\* Date acquisition

Function: 2AH

Setup: none

Return value: HL register <-- year

D register <-- month

E register <-- day of month

A register <-- day of week

This system call has been created for MSX-DOS and is not compatible with CP/M.

\* Date setting

Function: 2BH

Setup: HL register <-- year  
D register <-- month  
E register <-- day of month

Return value: A indicates whether the system call has done successfully. If successful, the A register is set to 00H; otherwise, 0FFH.

This system call has been created for MSX-DOS and is not compatible with CP/M.

#### \* Time acquisition

Function: 2CH

Setup: none

Return value: H register <-- hour  
L register <-- minute  
D register <-- second  
E register <-- 1/100 second

This system call has been created for MSX-DOS and is not compatible with CP/M.

#### \* Time setting

Function: 2DH

Setup: H register <-- hour  
L register <-- minute  
D register <-- second  
E register <-- 1/100 second

Return value: If successful, the A register is set to 00H; otherwise, 0FFH

This system call has been created for MSX-DOS and is not compatible with CP/M.

#### \* Verify flag setting

Function: 2EH

Setup: E register <-- 00H, when resetting verify flag  
E register <-- value other than 00H, when setting  
the verify flag

Return value: none

Setting the verify flag causes successive writing to the disk to be done in mode "verify on". That is, by reading the contents written on the disk, the check is made to compare them with the contents to be written.

This system call has been created for MSX-DOS and is not compatible with CP/M.

### 4.3 Absolute READ/WRITE (direct access to sectors)

MSX manages the disk with the "logical sector" as a minimum unit. The logical sector is defined independent of the physical sectors of the disk, and is numbered from 0 to the maximum logical sector (maximum number depends on the

kind of the disks).

Logical sectors enable users of MSX-DOS or MSX DISK-BASIC to access the disk without being concerned about the number of physical sectors per track, where that number depends on the media type of the disk. In fact, by utilizing system calls which use FCB (file control block), the file can be easily handled in detail even without considering logical sectors, so the user does not even need to use logical sectors. But for some purposes, access using logical sectors is desirable, so MSX-DOS and MSX DISK-BASIC offer system calls which can access logical sectors.

This section describes the system calls which access the disk by use of logical sectors.

#### \* Reading from the disk using logical sectors

Function: 2FH

Setup: The logical sector number to be read from (for more than one logical sector, the starting logical sector number) should be set in the DE register. The number of logical sectors to be read should be set in the H register, and the drive number (00H for drive A, 01H for drive B, and so on. The same follows for function 30H below) to be used to read should be set in the L register.

Return value: The contents read are set in the DMA buffer.

This system call reads out a specified number of continuous logical sectors from the specified logical sector of the specified drive and stores the contents in memory after DMA. It then stores the contents of what it has read in memory after DMA. Function 1AH (which specifies the address it is to be transferred to) assures that there is enough available space in memory.

This system call has been created for MSX-DOS and is not compatible with CP/M.

#### \* Writing to the disk using logical sectors

Function: 30H

Setup: Contents to be written should be set in memory area after the address indicated by DMA. The logical sector number from where the writing begins should be set in the DE register. The number of logical sectors to be written should be set in the H register. The drive number to be written to should be set in the L register.

Return value: none

This system call has been created for MSX-DOS and is not compatible with CP/M.

### List 3.4 Cluster dump

```
=====
;*****
;
;
```

```

; List 3.4    cluster dump
;
;           this program must link List 3.3
;
;*****
;
EXTRN GETARG      Note: The first 128 bytes of an arbitrary
EXTRN STOHEX      cluster specified in the command
EXTRN PUTHHEX     line are dumped.
EXTRN PUTCHR
EXTRN DUMP8B

```

```
BDOS EQU 0005H
```

```
;----- program start -----
```

```

LD A,1
CALL GETARG      ;[DE] := 1st argument of command line
CALL STOHEX     ;HL := evaluate [DE] as hexadecimal
                ; this is the target cluster No.
PUSH HL
LD E,00H        ;requests the default drive
LD C,1BH        ;get disk information
CALL BDOS
POP HL
CP 0FFH         ;fail ?
JR NZ,L2        ;if not fail, A := sector/cluster and goto L2

LD DE,ERMSG1    ;[DE] := 'Cannot get Disk information'
LD C,09H        ;string putput function
CALL BDOS
RET             ;error return

L2: LD E,(IX+12) ;DE := 1st sector of data area
LD D,(IX+13)
DEC HL
DEC HL          ;HL := Cluster No. - 2
LD B,H
LD C,L         ;BC := Cluster No. - 2
LOOP: DEC A    ;Count N times
JR Z,RESULT
ADD HL,BC
JR LOOP

RESULT: ADD HL,DE ;HL := sector of target cluster
PUSH HL      ;save target sector
LD DE,NEWDMA ;we reserved 1024 bytes area for DMA
LD C,1AH     ;Set DMA address function
CALL BDOS
LD C,19H
CALL BDOS    ;default drive ?
LD L,A
POP DE      ;DE := target sector
LD H,1     ;H := 1 (read 1 sector only)
LD C,2FH   ;absolute read function
CALL BDOS  ;data will be set into DMA

DUMP: LD HL,NEWDMA ;HL := DMA address

```



```

LD DE,0000H ;DE := relative address from cluster top
LD B,16 ;dump 16 lines
DLOOP: PUSH BC
LD A,D
CALL PUTHEx
LD A,E
CALL PUTHEx
LD A," "
CALL PUTCHR
PUSH HL
LD HL,8
ADD HL,DE
EX DE,HL ;DE := DE+8
POP HL
CALL DUMP8B ;8 bytes dump subroutine (in another file)
POP BC
DJNZ DLOOP
RET ;all work have done.

```

;----- work area -----

```

NEWDMA: DS 1024 ;Private DMA area
ADRS: DS 2

```

```

ERMSG1: DB "Cannot get Disk information.$"
ERMSG2: DB "Cannot read that cluster.$"

```

END

=====

#### 4.4 File Access Using FCB

Since accessing a file is difficult when using the system calls described in the previous section (which read and write logical sectors directly), system calls using FCB are needed to easier access the disk by specifying files.

There are three categories of system calls using FCB. First is sequential file access and second is random file access, both are offered to maintain CP/M compatibility. The third is what gives MSX-DOS its power: random block access. This method is not available in CP/M. Random block access has the following features:

- \* Any record size can be specified
- \* Random access can be made to multiple records
- \* File size can be controlled in bytes

This section describes system calls for file access using FCB, including random block access. Note that the following three functions do not work correctly when FCB is in the address range 4000H to 7FFFH:

1. Function call 11H
2. Function call 12H
3. Input/output for devices (CON, PRN, NUL, AUX)

## \* Opening files

Function: 0FH

Setup: DE register <-- starting address of FCB which is not opened

Return value: 00H is set in the A register when a file is opened successfully; otherwise 0FFH is set. When a file is opened successfully, each field of the FCB is set.

When 00H is specified for a drive number, the default drive set by function 0EH (default drive setting) is used. To open a file on another drive, specify 01H for drive A, 02H for drive B and so on.

When a file is opened by this system call, all FCB fields except record size, current block, current record, and random record are set using information from the directory area on the disk. Fields which are not set should be set by the user after executing this system call, if needed. The state that each field of FCB is set is "the state that file is opened" when using system calls using FCB, and, in this case, system calls which access the file using FCB, described below, can be used.

## \* Closing files

Function: 10H

Setup: DE register <-- starting address of opened FCB

Return value: 00H is set in the A register when file is closed successfully; otherwise, 0FFH is set.

By writing the current contents of FCB in memory to the corresponding directory area on the disk, file information can be kept current. When the file is only read, it does not need to be closed by using this system call.

## \* File search - 1

Function: 11H

Setup: DE register <-- starting address of FCB which is not opened

Return value: 00H is set in the A register when the file is found; otherwise 0FFH is set. When the file is found, the directory entry (32 bytes) of the file on the disk is set in the area indicated by DMA, and FCB drive number is set (thus, 33 bytes are used).

Wildcard characters can be used in the name of the file. For example, a specification "????????.c" causes any file name with an extension of "c" to be searched for, and the directory information of the file first found is written in after DMA. To find all matching files or to see whether there is only one matching file, use function 12H described below.

## \* File search - 2

Function: 12H

Setup: none

Return value: 00H is set in the A register when the file is found; otherwise 0FFH is set. When the file is found, the directory entry (32 bytes) of the file on the disk is set in the area

indicated by DMA, and the FCB drive number is set (thus, 33 bytes are used).

This system call should be used to search for multiple files meeting the file name specification by wildcard characters in function 11H. So this function should not be used by itself.

This system call allows the directory information of files meeting the specifications in function 11H to be listed in order, one by one.

#### \* Deleting files

Function: 13H

Setup: DE register <-- starting address of FCB which is not opened

Return value: 00H is set in the A register when file is successfully deleted., otherwise 0FFH is set.

Using wildcard characters for the file name may cause more than one file to be deleted. Exercise caution when using wildcards to delete files.

#### \* Sequential readout

Function: 14H

Setup: DE register <-- starting address of opened FCB

FCB current block <-- starting block for readout

FCB current record <-- starting record for readout

Return value: 00H is set in the A register when readout is successful; otherwise 01H is set. When successful, one record which has been read is set in the area indicated by DMA.

The FCB current block and record will be updated automatically after the readout. That is, in successive readouts, the current block and record do not need to be set. The record size for readout is fixed at 128 bytes.

#### \* Sequential writing to the disk

Function: 15H

Setup: DE register <-- starting address of opened FCB

FCB current block <-- starting block for writing

FCB current record <-- starting record for writing

128 bytes starting from DMA <-- data to be written

Return value: 00H is set in the A register when writing is successful; otherwise 01H is set.

The FCB current block and record will be updated automatically after the readout.

#### \* Creating files

Function: 16H

Setup: DE register <-- starting address of FCB which is not opened

Return value: 00H is set in the A register when the file is created successfully; otherwise 0FFH is set.

The record size, current block and record, and the random record of the FCB should be set after executing this system call.

#### \* Renaming files

Function: 17H

Setup: New file name should be set within 11 bytes after the 18th byte of the FCB (2nd byte of file size field of FCB = 16 bytes after old file name) corresponding to old file name (that is, it should be set in 18th to 28th byte), the FCB address should be set in the DE register.

Return value: 00H is set in the A register when the file name is renamed successfully; otherwise 0FFH is set.

Wildcard characters can be used for both the new and old file names. For example, specifying "?????????.o" for the old file name and "?????????.obj" for the new file name causes the extension of all files having ".o" to be changed to ".obj".

#### \* Random reading from the disk

Function: 21H

Setup: DE register <-- starting address of opened FCB  
random record in FCB <-- record for readout

Return value: 00H is set in the A register when readout is successful; otherwise 01H is set. When successful, the contents of one record which has been read are set in the area indicated by DMA.

The length of the record is fixed to 128 bytes.

#### \* Random writing to the disk

Function: 22H

Setup: DE register <-- starting address of opened FCB  
random record in FCB <-- record to be written to  
128 bytes starting from DMA <-- data to be written

Return value: 00H is set in the A register when writing is successful; otherwise 01H is set.

The length of the record is fixed to 128 bytes.

#### \* File size acquisition

Function: 23H

Setup: DE register <-- starting address of FCB which is not opened  
Return value: 00H is set in the A register when the function is successful; otherwise 00H is set. When successful, the size of the specified file is set in increments of 128 bytes, in the first three bytes of the random record field.

The file size is calculated in increments of 128 bytes. That is, 2 would be set for files ranging in size from 129 bytes to 256 bytes. Thus a file with

257 bytes would return a value of 3.

\* Random record field setting

Function: 24H

Setup: DE register <-- starting address of opened FCB  
FCB current block <-- objective block  
FCB current record <-- objective record

Return value: Current record position, calculated from the current block and record fields of specified FCB, is set in the random record field.

\* Random writing to the disk - 2 (random block access)

Function: 26H

Setup: DE register <-- the starting address from the FCB  
FCB record size <-- size of record to be written  
FCB random record <-- the record ID number  
HL register <-- the number of records to be written  
DMA memory buffer <-- the data to be written

Return value: 00H is set in the A register when writing is successful; otherwise 01H is set.

After writing to the disk, the value of the random record field is automatically updated and points to the next record. The size of one record can be set to any value from 1 byte to 65535 bytes by setting the desired value in the FCB record size field. When 0 records are to be written, the file length is calculated at the record size multiplied by the record number. The rest is discarded.

This system call has been created for MSX-DOS and is not compatible with CP/M.

\* Random readout - 2 (random block access)

Function: 27H

Setup: DE register <-- starting address of opened FCB  
FCB record size <-- record size to be read  
FCB random record <-- record to start reading  
HL register <-- number of records to be read

Return value: 00H is set in the A register when data is read successfully; otherwise 01H is read. The number of records actually read is set back in the HL register. When this number is almost one, the data which has been read is set in the area indicated by DMA.

After readout, the random record field is automatically updated. After executing this system call, the total number of records actually read is set in the HL register. That is, if the end of file is reached before the specified number of records have been read, the actual number of records read will be returned in the HL register.

This system call has been created for MSX-DOS and is not compatible with CP/M.

\* Random writing - 3

Function: 28H

Setup: DE register <-- starting address of opened FCB  
FCB random record <-- record to be written  
128 bytes in DMA buffer <-- data to be written

Return value: 00H is set in the A register when writing is successful;  
otherwise, 01H is set.

The length of records is fixed at 128 bytes.

This system call is the same as 22H except for one point. When the file becomes large, 00H is written to the added records coming before the specified record.

List 3.5 File dump

=====

```
*****  
;  
;  
; List 3.5 file dump  
;  
; this program must link List 3.3  
;  
*****  
;  
EXTRN GETARG Note: gets the dump list of the file  
EXTRN STOHEX specified at the command line  
EXTRN PUTCHR  
EXTRN PUTHEx  
EXTRN DUMP8B  
  
BDOS: EQU 0005H Note: The file name specified as the first  
FCB: EQU 005CH parameter of the command line is  
stored in the default FCB area  
from (005CH)  
  
;----- program start -----  
  
LD DE,FCB ;DE := default FCB address  
LD C,0FH ;open file function  
CALL BDOS  
OR A ;success ?  
JR Z,READ ;if so, goto READ  
  
LD DE,ERMSG1 ;[DE] := 'Cannot open that file'  
LD C,09H ;string output function  
CALL BDOS  
RET ;error return  
  
READ: LD A,2  
CALL GETARG ;get 2nd argument of command line  
CALL STOHEX ;HL := value of the argument  
LD (ADRS),HL ;set address counter
```

```

LD DE,NEWDMA
LD C,1AH      ;set DMA address function
CALL  BDOS

LD HL,8
LD (FCB+14),HL ;record size := 8

LD HL,0
LD (FCB+33),HL
LD (FCB+35),HL ;random record := 0

RD1:  LD HL,NEWDMA  ;clear DMA area
      LD B,8
RD2:  LD (HL)," "
      INC HL
      DJNZ  RD2

LD HL,1      ;read 1 record
LD DE,FCB
LD C,27H     ;random block read function
CALL  BDOS
OR  A       ;success ?
JR  Z,DUMP  ;if so, goto DUMP

LD DE,ERMSG2 ;[DE] := 'Ok.'
LD C,09H     ;string output function
CALL  BDOS
RET

DUMP: LD HL,(ADRS)
      LD A,H
      CALL  PUTHEx
      LD A,L
      CALL  PUTHEx
      LD A," "
      CALL  PUTCHR
      LD DE,8
      ADD  HL,DE
      LD (ADRS),HL

LD HL,NEWDMA
CALL  DUMP8B      ;dump 8 bytes

JR  RD1

;----- work area -----

ADRS:  DS  2
NEWDMA: DS  8

;----- error message -----

ERMSG1: DB "Cannot open that file.$"
ERMSG2: DB "Ok.$"

END

```

=====

## MSX2 TECHNICAL HANDBOOK

-----

Edited by: ASCII Systems Division  
Published by: ASCII Corporation - JAPAN  
First edition: March 1987

Text files typed by: Nestor Soriano (Konami Man) - SPAIN  
March 1997

### Changes from the original:

- In Figure 4.3, "Port#17" indication is corrected to "R#17".
- In Figure 4.4, "00" field in R#17 is corrected to "10".
- In section 3.2.2, subsection "Pattern name table", text "12 low order bits o the address (A9 to A0)" is corrected to "12 low order bits of the address (A11 to A0)"
- In Figure 4.17, the numerations of the two last rows in the Screen correspondence table, originally "22" and "23", are corrected to "25" and "26" respectively.
- In section 3.2.2, subsection "Blink table", the text "the 9 low order bits of the address (A9 to A0)" is corrected to "the 8 low order bits of the address (A8 to A0)".
- In Figure 4.25, indication "Specifies the value of the screen (0 to 15)" is changed to "Specifies the border colour (0-15)".
- In Figure 4.34, in the screen correspondance table, the three stages of the screen are named "Upper stage of screen" in the original. This is corrected, and the stages are named "Upper", "Middle" and "Lower".
- The title of section 3.6.3 is "Screen colour mode specification" in the original. The word "mode" is erased.
- In section 3.8.2, the text "by writing the 2 high order bits" is corrected to "by writing the high order bit".
- The title of Figure 4.63 is "Judging the conflict (sprite mode 2)" in the original. This is corrected to "Judging the conflict (sprite mode 1)".
- In Figure 4.68, indication "Color code = 8 or 4 or 12" is changed to "Color code = 8 or 4 = 12".

-----

## CHAPTER 4 - VDP AND DISPLAY SCREEN (Parts 1 to 5)

The MSX2 machines uses an advanced VDP (video display processor) for its display screen, the V9938 (MSX-VIDEO). This LSI chip allows for several new graphics features to be accessed by the MSX2 video display. It is also fully



compatible with the TMS9918A used in the MSX1.

Chapter 4 describes how to use this video display processor. It describes functions not accessible by BASIC. For mode details (e.g. hardware specifications, see V9938 MSX-VIDEO Technical Data Book (ASCII)).

## 1. MSX-VIDEO CONFIGURATION

The following features of the MSX-VIDEO give it a better display capabilities than the TMS9918A:

- \* 512 colours with a 9-bit colour palette
- \* Max. 512 x 424 dot resolution (when using the interlace)
- \* Max. 256 colours at the same time
- \* Full bitmap mode which makes graphic operations easy
- \* Text display mode of 80 characters per line
- \* LINE, SEARCH, AREA-MOVE executable by hardware
- \* Up to 8 sprites on the same horizontal line
- \* Different colours can be specified for each line in a sprite
- \* Video signal digitizing feature built-in
- \* Superimpose feature built-in

### 1.1 Registers

MSX-VIDEO uses 49 internal registers for its screen operations. These registers are referred to as "VDP registers" in this book. VDP registers are classified by function into three groups as described below. The control register group and status register group can be referred to using VDP(n) system variables from BASIC.

#### (1) Control register group (R#0 to R#23, R#32 to R#46)

This is a read-only 8-bit register group controlling MSX-VIDEO actions. Registers are expressed using the notation R#n. R#0 to R#23 are used to set the screen mode. R#32 to R#46 are used to execute VDP commands. These VDP commands will be described in detail in section 5. Control registers R#24 to R#31 do not exist. The roles of the different control registers are listed in Table 4.1.

Table 4.1 Control register list

R#n	Corresponding VDP(n)	Function
R#0	VDP(0)	mode register #0
R#1	VDP(1)	mode register #1
R#2	VDP(2)	pattern name table
R#3	VDP(3)	colour table (LOW)
R#4	VDP(4)	pattern generator table
R#5	VDP(5)	sprite attribute table (LOW)
R#6	VDP(6)	sprite pattern generator table

R#7	VDP(7)	border colour/character colour at text mode
R#8	VDP(9)	mode register #2
R#9	VDP(10)	mode register #3
R#10	VDP(11)	colour table (HIGH)
R#11	VDP(12)	sprite attribute table (HIGH)
R#12	VDP(13)	character colour at text blinks
R#13	VDP(14)	blinking period
R#14	VDP(15)	VRAM access address (HIGH)
R#15	VDP(16)	indirect specification of S#n
R#16	VDP(17)	indirect specification of P#n
R#17	VDP(18)	indirect specification of R#n
R#18	VDP(19)	screen location adjustment (ADJUST)
R#19	VDP(20)	scanning line number when the interrupt occurs
R#20	VDP(21)	colour burst signal 1
R#21	VDP(22)	colour burst signal 2
R#22	VDP(23)	colour burst signal 3
R#23	VDP(24)	screen hard scroll
-----		
R#32	VDP(33)	SX: X-coordinate to be transferred (LOW)
R#33	VDP(34)	SX: X-coordinate to be transferred (HIGH)
R#34	VDP(35)	SY: Y-coordinate to be transferred (LOW)
R#35	VDP(36)	SY: Y-coordinate to be transferred (HIGH)
R#36	VDP(37)	DX: X-coordinate to be transferred to (LOW)
R#37	VDP(38)	DX: X-coordinate to be transferred to (HIGH)
R#38	VDP(39)	DY: Y-coordinate to be transferred to (LOW)
R#39	VDP(40)	DY: Y-coordinate to be transferred to (HIGH)
R#40	VDP(41)	NX: num. of dots to be transferred in X direction (LOW)
R#41	VDP(42)	NX: num. of dots to be transferred in X direction (HIGH)
R#42	VDP(43)	NY: num. of dots to be transferred in Y direction (LOW)
R#43	VDP(44)	NY: num. of dots to be transferred in Y direction (HIGH)
R#44	VDP(45)	CLR: for transferring data to CPU
R#45	VDP(46)	ARG: bank switching between VRAM and expanded VRAM
R#46	VDP(47)	CMR: send VDP command

## (2) Status register (S#0 to S#9)

This is a read-only 8-bit register group which reads data from MSX-VIDEO. Registers are expressed using the notation S#n. The functions of the registers are listed in Table 4.2.

Table 4.2 Status register list

S#n	Corresponding VDP(n)	Function
S#0	VDP(8)	interrupt information
S#1	VDP(-1)	interrupt information
S#2	VDP(-2)	DP command control information/etc.
S#3	VDP(-3)	coordinate detected (LOW)
S#4	VDP(-4)	coordinate detected (HIGH)
S#5	VDP(-5)	coordinate detected (LOW)
S#6	VDP(-6)	coordinate detected (HIGH)

S#7	VDP(-7)	data obtained by VDP command	
S#8	VDP(-8)	X-coordinate obtained by search command (LOW)	
S#9	VDP(-9)	X-coordinate obtained by search command (HIGH)	

---

### (3) Colour palette register group (P#0 to P#15)

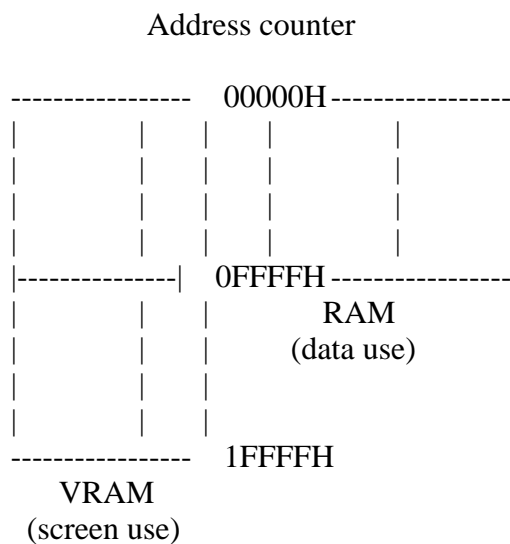
These registers are used to set the colour palette. Registers are expressed using the notation P#n where 'n' is the palette number which represents one of 512 colours. Each palette register has 9 bits allowing three bits to be used for each RGB colour (red, green, and blue).

## 1.2 VRAM

MSX-VIDEO can be connected with 128K bytes VRAM (Video RAM) and 64K bytes expanded RAM. MSX-VIDEO has a 17-bit counter for accessing this 128K bytes address area. Note that this memory is controlled by MSX-VIDEO and cannot be directly accessed by the CPU.

Expanded RAM memory cannot be directly displayed to the screen as can that of VRAM. However, it can be manipulated the same as VRAM when using the video processor commands. This large work area is very useful when processing screen data. Note that the MSX standard does not include instructions regarding expanded RAM, so taking advantage of this in program design could result in compatibility problems with other MSX machines.

Figure 4.1 VRAM and expanded RAM



## 1.3 I/O ports

MSX-VIDEO has four I/O ports that send data back and forth the CPU. The functions of these ports are listed in Table 4.3. The ports are accessed by the CPU through its I/O addresses in the table below, addresses expressed as n, n' are stored at address locations 6 and 7 in MAIN-ROM. Although n = n' = 98H normally, this can be different on some machines, so port addresses should be obtained from these addresses for reliable results.

It is generally recommended that BIOS be used for I/O operations for purposes of compatibility. However, the screen display often requires high speed, so these I/O ports are capable of accessing MSX-VIDEO directly.

Table 4.3 MSX-VIDEO ports

Port	Address	Function
port #0 (READ)	n	read data from VRAM
port #0 (WRITE)	n'	write data to VRAM
port #1 (READ)	n + 1	read status register
port #1 (WRITE)	n' + 1	write to control register
port #2 (WRITE)	n' + 2	write to palette register
port #3 (WRITE)	n' + 3	write to indirectly specified register

Note: The value of n should be obtained by referring to address 6 in MAIN-ROM  
The value of n' should be obtained by referring to address 7 in MAIN-ROM

## 2. ACCESS TO MSX-VIDEO

MSX-VIDEO can be accessed directly through the I/O ports without going through BIOS. This chapter describes how to do this.

### 2.1 Access to Registers

#### 2.1.1 Writing data to control registers

The control registers are write-only registers. As described above, the partial contents of control registers (R#0 to R#23) can be obtained by referring to VDP(n) from BASIC. This only reads the value which has been written in the work area of RAM (F3DFH to F3E6H, FFE7H to FFF6H) used for writing to registers.

There are three ways, described below, to write data to control registers. Since MSX accesses MSX-VIDEO inside the timer interrupt routine to examine the occurrence of sprite conflicts, note that access procedure will not inhibiting the interrupt when the registers are accessed in the proper way as described below.

#### (1) Direct access

The first way is to directly specify the data and where it is to be written to. Figure 4.2 illustrates the procedure. The data is first written to port#1 and then the destination register number is written to port#1 using the five least significant bits. The most significant bit is set to 1 and the second bit is set to 0. Thus the value would be 10XXXXXB in binary notation where XXXXX is the destination register number.

Figure 4.2 Direct access to R#n

MSB 7 6 5 4 3 2 1 0 LSB

-----  
 Port #1 | : : : Data : : : | 1.Puts data to port #1.  
 -----

-----  
 Port #1 | 1 | 0 | R5 | R4 | R3 | R2 | R1 | R0 | 2. Then puts register  
 ----- number with two  
 | | | | | | | | high bits set to  
 +-----+ "10" to port #1.  
 fixed register number (0 to 46)  
 at "10"

Port#1 is also used to set VRAM addresses and is described in section 2.2. The most significant bit of the second byte sent to this port is the address/register flag and determines the operation to take place. When the bit is set to "1", writing data to a control register as described here will take place.

(2) Indirect Access (non-autoincrement mode)

The second way is to write data to the register specified as the objective register (R#17 contains the objective pointer). To begin with, store the register number to be accessed in R#17 by direct access. The most significant bit is set to 1 and the second bit to 0. Thus the value would be 10XXXXXB in binary notation where XXXXX is the objective register number. After this is done, data can be written to the objective register by sending data to port#3. This method is used for sending data to the same register continuously. An example would be for the execution of VDP commands.

Figure 4.3 Indirect access to R#n (non-autoincrement mode)

First byte

MSB 7 6 5 4 3 2 1 0 LSB

-----  
 R#17 | 1 | 0 | R5 | R4 | R3 | R2 | R1 | R0 | 1.Set register number n  
 ----- to R#17, with two high  
 | | | | | | | | order bits set to "10".  
 +-----+  
 fixed n (0 to 46)  
 at "10"

-----  
 Port#3 | : : : Data : : : | 2.Send data to port#3.  
 ----- The data is stored  
 in register R#n.

Following bytes

-----  
 Port#3 | : : : Data : : : | 3.After these are done,  
 ----- data can be written to  
 register R#n only by

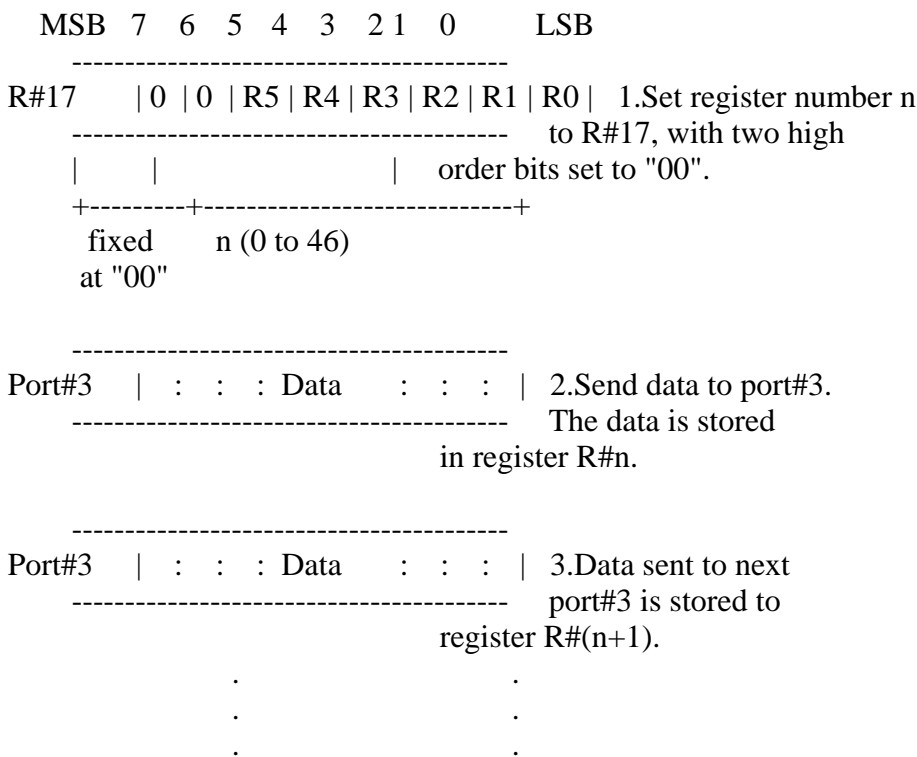
sending to port #3.

### (3) Indirect Access (autoincrement mode)

The third way is to write data to the register indicated by R#17. R#17 is incremented each time data is sent to port#3. To begin with, store the beginning register number to be accessed in R#17 by direct access. The two most significant bits are set to 0. Thus the value would be 00XXXXXB in binary notation where XXXXX is the beginning register number.

Since this method allows writing data to continuous control registers effectively, it is useful when several continuous registers are to be changed at once. One example would be when the screen mode is changed.

Figure 4.4 Indirect access to R#n (autoincrement mode)

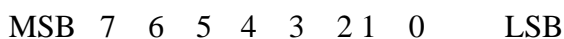


#### 2.1.2 Setting a palette

To set data in the MSX-VIDEO palette registers (P#0 to P#15), specify the palette register number in the four lowest significant bits of R#16 (color palette pointer), and then send the data to port#2. Since palette registers have a length of 9 bits, data must be sent twice; red brightness and blue brightness first, then green brightness. Brightness is specified in the lower three bits of a four bit segment. Refer to Figure 4.5 for details.

After data is sent to port#2 twice, R#16 is automatically incremented. This feature makes it easy to initialize all the palettes.

Figure 4.5 Setting a colour palette register



```

-----
R#16  | 0 | 0 | 0 | 0 | R3 | R2 | R1 | R0 | 1.Set palette number n
-----
                    to R#16, with four
|           |           | high order bits set
+-----+-----+ to "0000".
fixed at "0000"   n (0 to 15)

```

```

-----
Port#2 | | Red bright. | | Blue bright. | 2.Send red and blue
-----
                    brightness to port#2.
|           | |           |
+-----+-----+
0 to 7      0 to 7

```

```

-----
Port#2 | | | | | Green bright. | 2.Send green brightness
-----
                    to port#2. (*)
|           |
+-----+
0 to 7

```

(\*) Since R#16 is incremented at this point, setting next palette can be done by sending data to port#2 continuously.

### 2.1.3 Reading status registers

Status registers are read-only registers. Their contents can be read from port#1 by setting the status register number in the least significant four bits of R#15 (status register pointer) as shown in Figure 4.6. The four most significant bits are set to 0. Thus the value would be 0000XXXXB in binary notation where XXXX is the status register number. Interrupts should be inhibited before the status register is accessed. After the desired task is completed, R#15 should be set to 0 and the interrupts released.

Figure 4.6 Accessing status registers

```

MSB 7 6 5 4 3 2 1 0      LSB
-----
R#15  | 0 | 0 | 0 | 0 | R3 | R2 | R1 | R0 | 1.Set register number n
-----
                    to R#15, with four
|           |           | high order bits set
+-----+-----+ to "0000".
fixed at "0000"   n (0 to 9)

-----
Port#1 | : : : Data : : : | 2.Read data from port#1.
-----

```

### 2.2 VRAM Access From the CPU

When a VRAM address is to be accessed from the CPU, follow the procedure described below.

### (1) Bank switching

The first 64K bytes of VRAM (00000H to 0FFFFH) and the 64K bytes of expanded RAM both reside at the same address space as viewed by MSX-VIDEO. Bank switching is used so that they can both be online at the same time. Since MSX2 does not use expanded VRAM, always select the VRAM bank. Bank switching is controlled by bit 6 of R#45.

Figure 4.7 VRAM/expanded RAM bank switching

MSB	7	6	5	4	3	2	1	0	LSB
-----									
R#45		.		X		.		.	
-----									
									When bit 6 of R#45 is "0"
									VRAM is selected; when it
0:VRAM									is 1, expanded RAM is
+-->									selected.
1:Expanded RAM									

### (2) Setting the VRAM page (three high order bits)

The 17-bit address for accessing the 128K bytes of VRAM is set in the address counter (A16 to A0). R#14 contains the three high order bits (A16 to A14). So this register can be viewed as switching between eight 16K byte pages of VRAM.

Figure 4.8 Setting the VRAM page (3 high order bits)

MSB	7	6	5	4	3	2	1	0	LSB
-----									
R#14		0		0		0		0	
									A16  A15  A14
									Set 3 high order bits of
									address counter in the
									field from bit 2 to bit 0
									on R#14.

### (3) Setting the VRAM address (14 low order bits)

The 14 low order bits of the address should be sent to port#1 in two bytes. Figure 4.9 shows the details. Make sure that the most significant bit of the second byte sent is set to 0. This sets the address/register flag to address mode. The second most significant bit sets the read/write flag. 1 signifies writing to VRAM and 2 signifies reading from VRAM.

Figure 4.9 Setting 14 low order bits

MSB	7	6	5	4	3	2	1	0	LSB
-----									
Port#1		A7		A6		A5		A4	
									A3   A2   A1   A0   1.Send A7 to A0 to
									port#1.
-----									
Port#1		0		A13		A12		A11	
									A10  A9   A8   2.Send A13 to A8 to
									port#1, continuously.



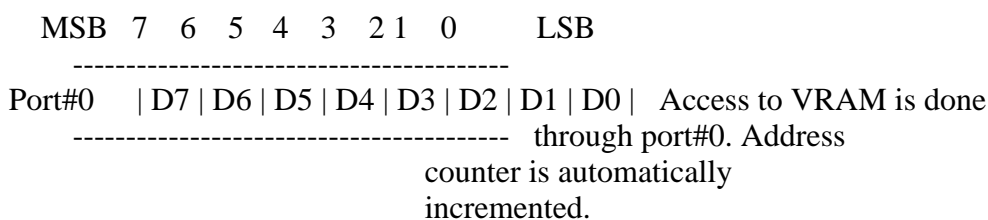
		0:reading VRAM	Bit 7 must be set to
"0"	+->		"0". Bit 6 determines
		1:writing VRAM	reading/writing data.

#### (4) Reading/writing VRAM

After setting the value in the address counter, read or write data through port#0. The read/write flag is set the same time as A13 to A8 of the address counter, as described above.

The address counter is automatically incremented each time a byte of data is read or written to port #0. This feature allows for easy access of continuous memory in VRAM.

Figure 4.10 Access to VRAM through port#0



### 3. SCREEN MODES OF THE MSX2

The MSX2 has ten different modes as shown in Table 4.4. Six screen modes marked with "\*" in the table below (TEXT 2 and GRAPHIC 3 to GRAPHIC 7) have been introduced for the MSX2. The other modes have been improved due to the change from TMS9918A to MSX-VIDEO. Fetures of these ten screen modes and how to use them are described below.

Table 4.4 Screen modes listing of MSX2

Mode Name	SCREEN mode	Description
TEXT 1	SCREEN 0	40 characters per line of text, one colour (width=40) for each character
* TEXT 2	SCREEN 0	80 characters per line of text, (width=80) character blinkable selection
MULTI-COLOR	SCREEN 3	pseudo-graphic, one character divided into four block
GRAPHIC 1	SCREEN 1	32 characters per one line of text, the COLOURed character available
GRAPHIC 2	SCREEN 2	256 x 192, the colour is specified for each 8 dots
* GRAPHIC 3	SCREEN 4	GRAPHIC 2 which can use sprite mode 2

* GRAPHIC 4   SCREEN 5	256 x 212; 16 colours are available for each dot
* GRAPHIC 5   SCREEN 6	512 x 212; 4 colours are available for each dot
* GRAPHIC 6   SCREEN 7	512 x 212; 16 colours are available for each dot
* GRAPHIC 7   SCREEN 8	256 x 212; 256 colours are available for each dot

### 3.1 TEXT 1 Mode

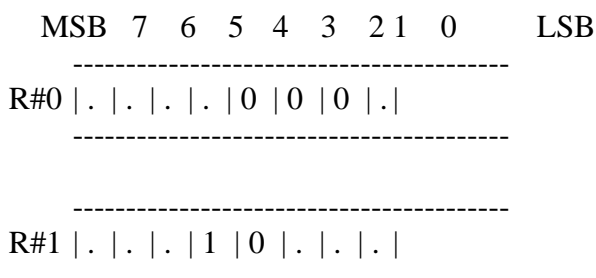
TEXT 1 screen mode has the following features:

screen:	40 (horizontal) x 24 (vertical)	background/character colours can be selected from 512 colours
character:	256 characters available	character size: 6 (horizontal) x 8 (vertical)
memory requirements:	for character font ... 2048 bytes (8 bytes x 256 characters)	for display ..... 960 bytes (40 characters x 24 lines)
BASIC:	compatible with SCREEN 0 (WIDTH 40)	

#### 3.1.1 Setting TEXT 1 mode

MSX-VIDEO screen modes are set by using 5 bits of R#0 and R#1. Figure 4.11 shows the details. The 3-bit mask in R#0 is 000B and the 2-bit mask in R#1 is 10B when using the TEXT 1 mode.

Figure 4.11 Setting TEXT1 mode



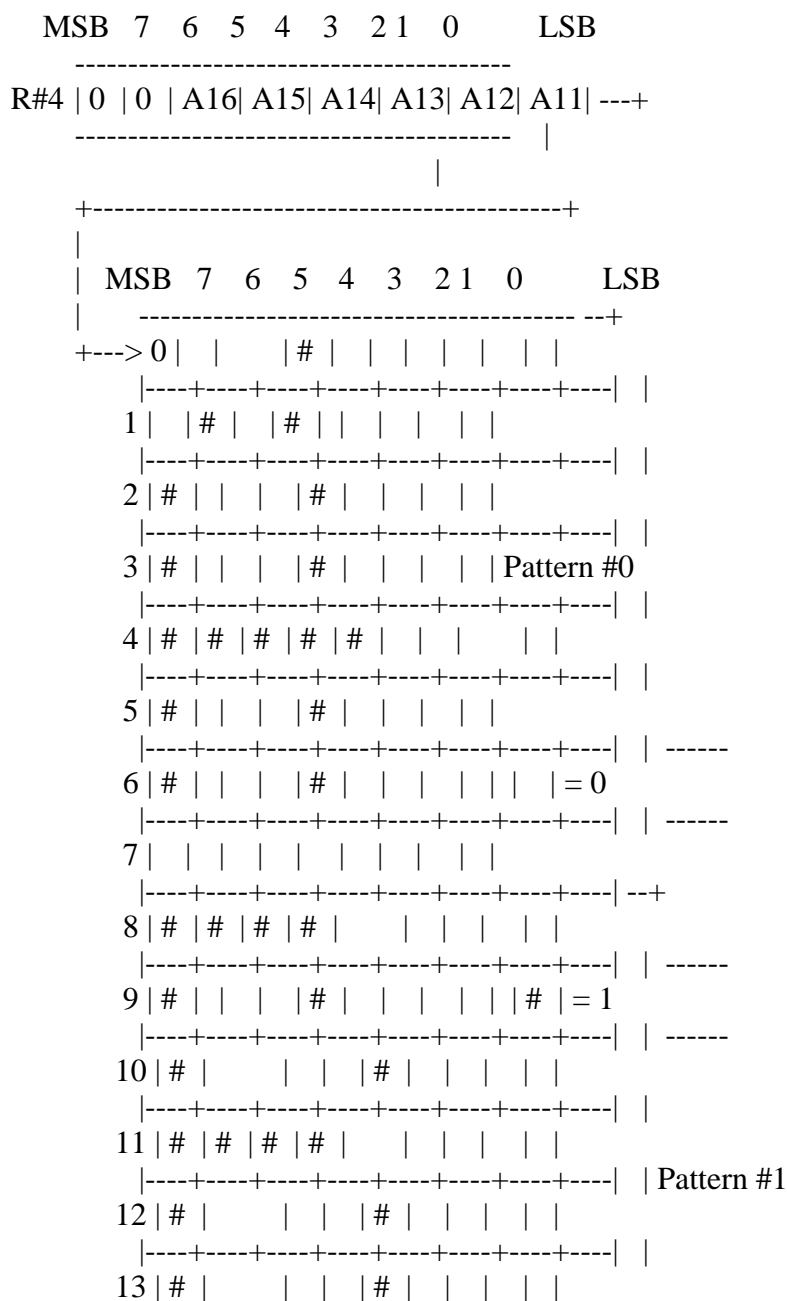
#### 3.1.2 Screen structure of TEXT 1 mode

\* Pattern generator table

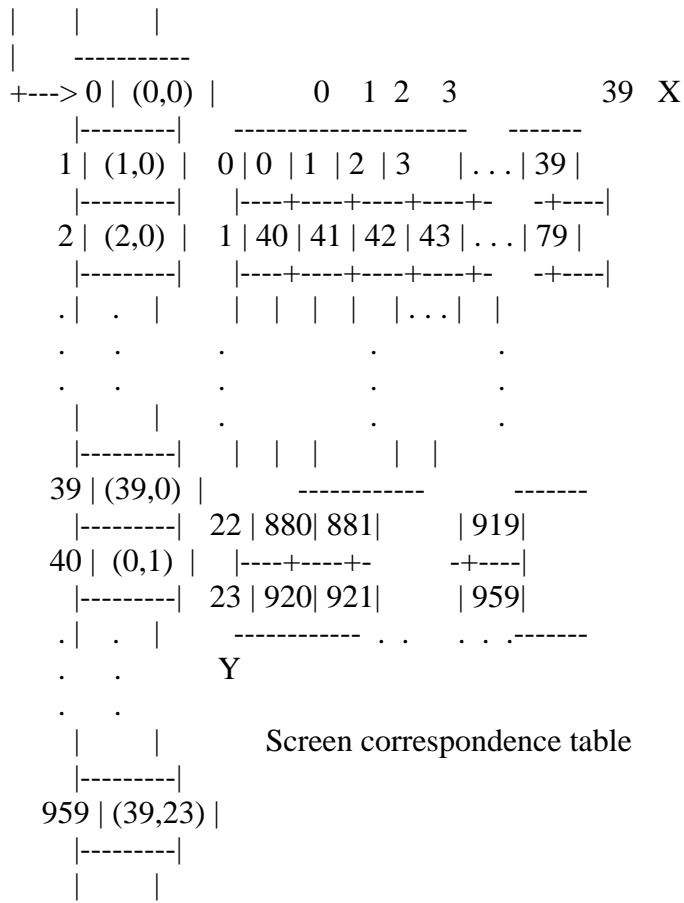
The area in which character fonts are stored is called the pattern generator table. This table is located in VRAM, and, although the font is defined by using 8 bytes for each character from the top of the table, the 2 low order bits of each byte representing the right two columns are not displayed on the screen. Thus, the size of one character is 6 x 8 pixels. Each character font set contains 256 different characters numbered from 0 to 255. Use this code to specify which character should be displayed on the screen.

Specify the location of the pattern generator table in R#4. Note that the 6 high order bits of the address (A16 to A11) are specified and the 11 low order bits of the address (A10 to A0) are always 0 ("00000000000B"). So the address in which the pattern generator table can be set always begins at a multiple of 2K bytes from 00000H. This address can be found using the system variable BASE(2) from BASIC. Figure 4.12 shows the structure of the pattern generator table.

Figure 4.12 Structure of the pattern generator table







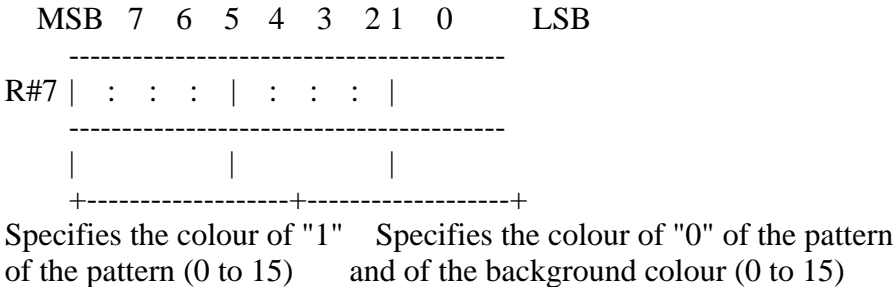
Screen correspondence table

Pattern Name Table

3.1.3 Specifying screen colour

The screen colour is specified by R#7. The background colour is the palette specified by the 4 low-order bits of R#7; the 4 high-order bits specify the foreground colour (see Figure 4.14). A "0" in the font pattern is displayed in the background colour and a "1" is displayed in the foreground colour. Note that in TEXT 1 the border colour of the screen cannot be set and it is the same as the background colour.

Figure 4.14 Colour specification in TEXT 1



3.2 TEXT 2 Mode

The screen mode TEXT 2 has the following features:

-----

```

|
| screen:      80 (horizontal) x 24 (vertical) or 26.5 (vertical) |
|              background colour/character colour can be selected |
|              from 512 colours                                     |
| character:   256 characters available                             |
|              character size: 6 (horizontal) x 8 (vertical)      |
|              each character blinkable                             |
| memory requirements: 24 lines                                    |
|                  for character font ... 2048 bytes               |
|                  (8 bytes x 256 characters)                       |
|                  for display ..... 1920 bytes                   |
|                  (80 characters x 24 lines)                       |
|                  for blinking ..... 240 bytes (= 1920 bits)    |
| 26.5 lines                                         |
|                  for character font ... 2048 bytes               |
|                  (8 bytes x 256 characters)                       |
|                  for display ..... 2160 bytes                   |
|                  (80 characters x 27 lines)                       |
|                  for blinking ..... 270 bytes (= 2160 bits)    |
| BASIC:      compatible with SCREEN 0 (WIDTH 80)                 |
|
|-----

```

### 3.2.1 Setting TEXT 2 mode

Set TEXT2 mode as shown in Figure 4.15.

Figure 4.15 Setting TEXT2 mode

```

      MSB 7 6 5 4 3 2 1 0      LSB
-----
R#0 | . | . | . | . | 0 | 1 | 0 | . |
-----

-----
R#1 | . | . | . | 1 | 0 | . | . | . |
-----

```

\* Setting number of lines (24 lines/26.5 lines)

TEXT2 mode can switch the screen to 24 lines or 26.5 lines depending on the value of bit 7 in R#9. Note that, when the screen is set to 26.5 lines, only the upper half of the characters at the bottom of the screen are displayed. This mode is not supported by BASIC.

Figure 4.16 Switching number of lines

```

      MSB 7 6 5 4 3 2 1 0      LSB
-----
R#9 | LN | . | . | . | . | . | . |
-----
| 0:24 lines
+-->

```

3.2.2 Screen structure of TEXT 2

\* Pattern generator table

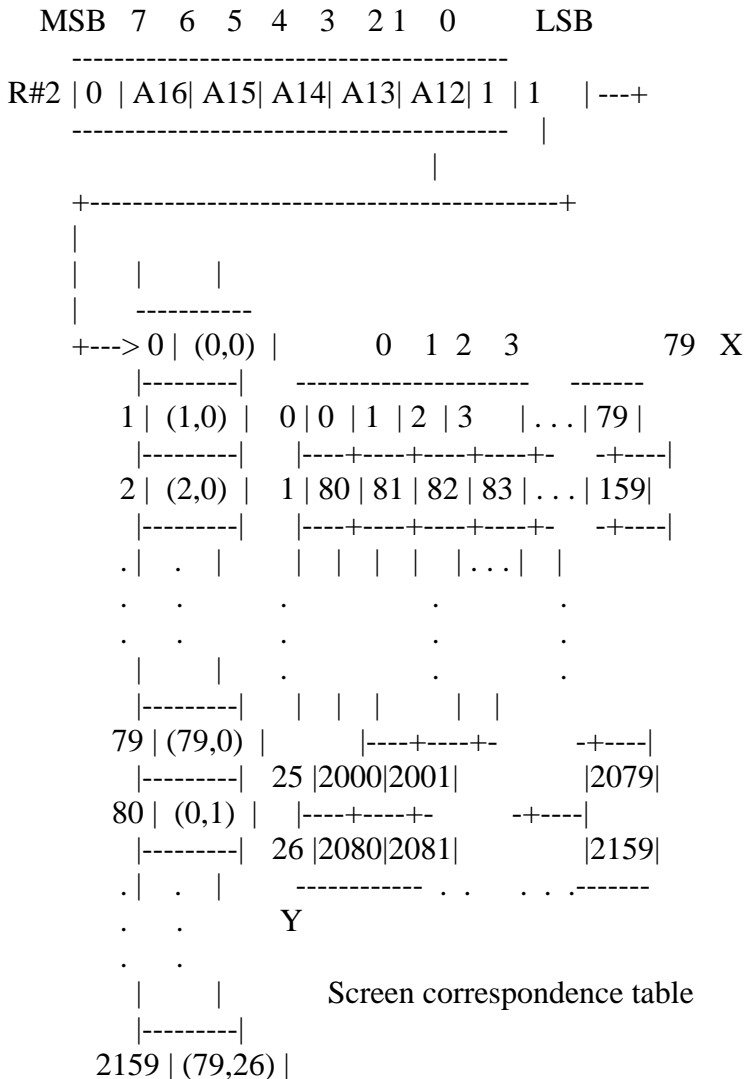
The pattern generator table has the same structure and function as the one of TEXT1. See the descriptions for TEXT1.

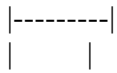
\* Pattern name table

Since the number of characters to be displayed in the screen has been increased to 2160 (80 x 27) characters maximum, the maximum area occupied by the pattern name table is 2160 bytes.

Specify the location of the pattern name table in R#2. The 5 high order bits of the address (A16 to A12) are specified and the 12 low order bits of the address (A11 to A0) are always 0 ("000000000000B"). So the address in which the pattern name table can be set always begins at a multiple of 4K bytes from 00000H.

Figure 4.17 Structure of TEXT2 pattern name table



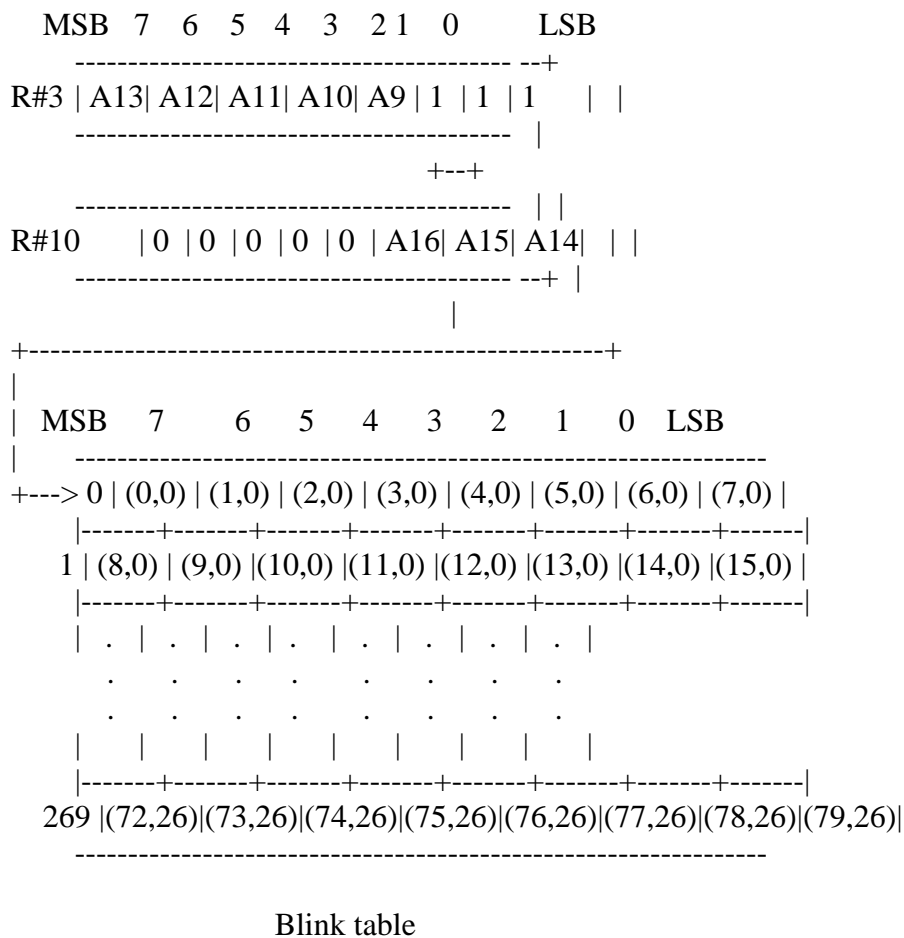


### Pattern Name Table

#### \* Blink table

In TEXT2 mode, it is possible to set the blink attribute for each character. The blink table stores the information of the screen location of the characters blinked. One bit of the blink table corresponds to one character on the screen (that is, on the pattern name table). When the bit is set to "1" blinking is enabled for the corresponding character; when the bit is "0" blinking is disabled.

Figure 4.18 Blink table structure of TEXT2



Specify the starting address of the blink table by setting the 8 high order bits (A16 to A9) in R#3 and R#10. The location of the blink table is set by writing the 8 high order bits of the address (A16 to A9) in R#3 and R#10. The 9 low order bits of the address (A8 to A0) are always 0 ("00000000B"). So the address in which the blink table can be set always begins at a multiple of 512 bytes from 00000H.

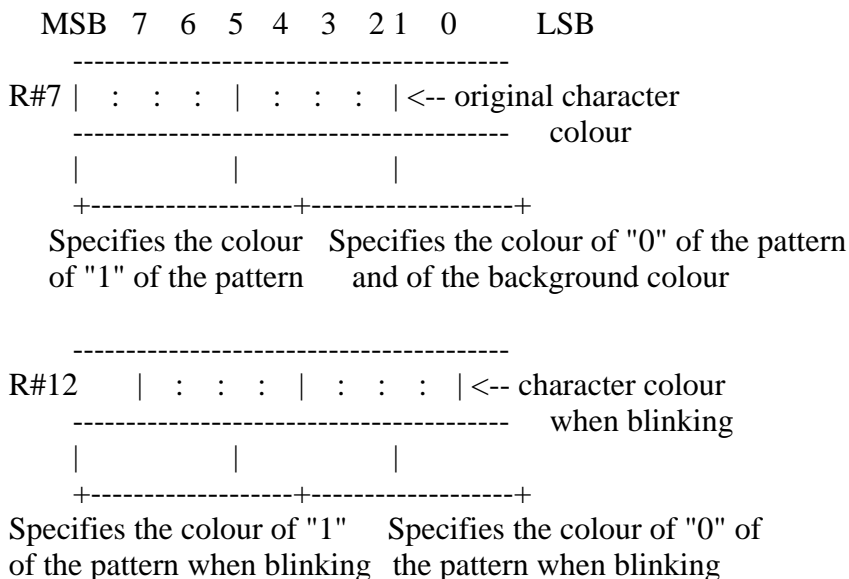
#### 3.2.3 Screen colour and character blink specification

The foreground colour is specified by the 4 high order bits of R#7 and the



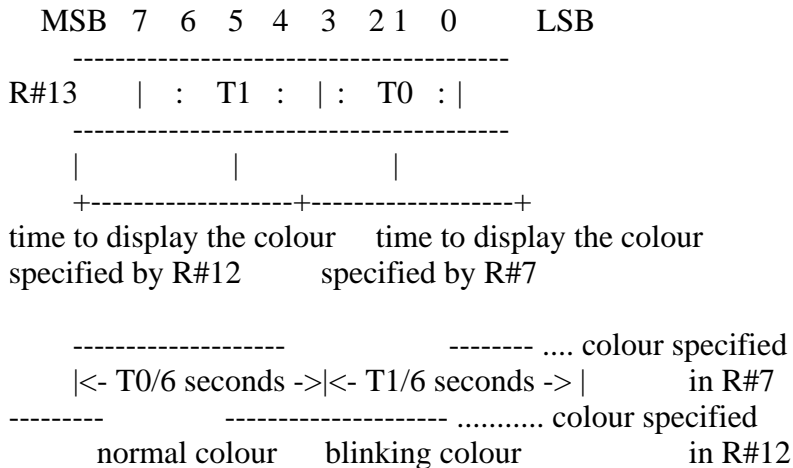
background colour by the 4 low order bits of R#7. Characters with a blink attribute of 1 defined by the blink table alternate between the blink colour and the colour specified in R#12.

Figure 4.19 Setting screen colour and blink colour



The blinking rate is set in R#13. The 4 high order bits define the display time in the original colour, and the 4 low order bits define the display time in the blink colour. The period of time is defined in units of 1/6 seconds.

Figure 4.20 Setting blink rate



List 4.1 Blink example

```

=====
1000 '*****
1010 ' LIST 4.1 BLINK SAMPLE
1020 '*****
1030 '
1040 SCREEN 0 : WIDTH 80          'TEXT 2 mode
1050 ADR=BASE(1)                'TAKE COLOR TABLE ADDRESS
1060 '

```

```

1070 FOR I=0 TO 2048/8
1080 VPOKE ADR+I,0          'reset blink mode
1090 NEXT
1100 '
1110 VDP(7) = &HF1          'text color=15, back color=1
1120 VDP(13) = &H1F        'text color=1, back color=15
1130 VDP(14) = &H22        'set interval and start blink
1140 '
1150 PRINT "Input any character : ";
1160 '
1170 K$ = INPUT$(1)
1180 IF K$ < CHR$(28) THEN 1230
1190 IF K$ > " " THEN GOSUB 1280
1200 PRINT K$;
1210 GOTO 1170
1220 '
1230 VDP(14) = 0           'stop blink
1240 END
1250 '
1260 '----- set blink mode -----
1270 '
1280 X = POS(0) : Y = CSRLIN
1290 A = (Y*80 + X) \ 8
1300 B = X MOD 8
1310 M = VAL("&B" + MID$("0000000100000000", 8 - B, 8))
1320 VPOKE ADR + A, VPEEK(ADR + A) XOR M
1330 RETURN
=====

```

### 3.3 MULTI COLOUR Mode

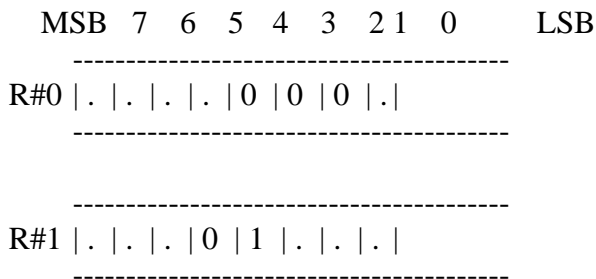
The MULTI COLOUR mode is described below:

screen:	64 (horizontal) x 48 (vertical) blocks	
	16 colours from 512 colours can be displayed at the same time	
block:	block size is 4 (horizontal) x 4 (vertical) dots	
	colour can be specified to each block	
memory requirements:	for setting colours ..... 2048 bytes	
	for specifying locations ..... 768 bytes	
sprite:	sprite mode 1	
BASIC:	compatible to SCREEN 3	

#### 3.3.1 Setting MULTI COLOUR mode

Set MULTI COLOUR mode as shown in Figure 4.21.

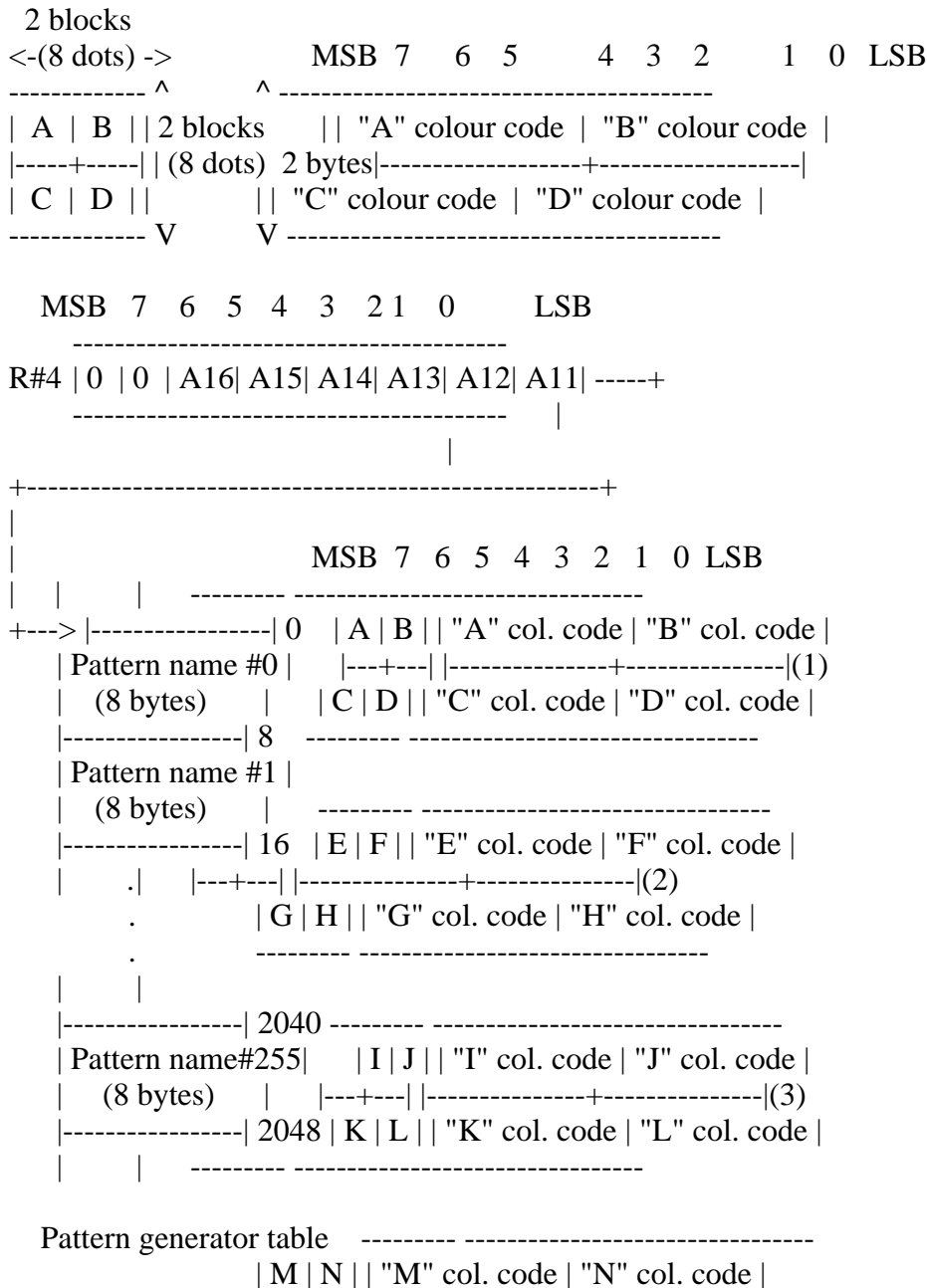
Figure 4.21 Setting MULTI COLOUR mode



\* Pattern generator table

In this mode, patterns are constructed as 2 x 2 blocks and one pattern name corresponds to four patterns. The starting address on this table is specified in R#4. Since only the 6 high order bits (A16 to A11) of the address is specified, the pattern generator table can be located at intervals of 2K bytes from 00000H (see Figure 4.22).

Figure 4.22 Pattern generator table structure of MULTI COLOUR



```

|----+----| |-----+-----| (4)
| O | P | | "O" col. code | "P" col. code |
-----

```

- (1) This table is in effect when Y is 0, 4, 8, 12, 16, or 20
- (2) This table is in effect when Y is 1, 5, 9, 13, 17, or 21
- (3) This table is in effect when Y is 2, 8, 10, 14, 18, or 22
- (4) This table is in effect when Y is 3, 7, 11, 15, 19, or 23

\* Pattern name table

This is the table for displaying specified patterns at desired locations on the screen. One of four patterns in a pattern name is displayed at its Y-coordinate value. BASIC sets the contents of this table as shown in Figure 4.23. The starting address of the pattern name table is specified by R#2. Since only the 7 high order bits of the address (A16 to A10) are specified, the address at which this table can be set is at increments of 1K bytes from 00000H (see Figure 4.24).

Figure 4.23 Setting BASIC pattern name table

```

Pattern 0
  X 0 1 2 3 4 5 ... 26 27 28 29 30 31
----- Y -----
| | | 0 | 0 | 1 | 2 | 3 | 4 | 5 | ... | 26 | 27 | 28 | 29 | 30 | 31 |
|----+----| |-----+-----|
| | | 1 | 0 | 1 | 2 | 3 | 4 |          | 27 | 28 | 29 | 30 | 31 |
|----+----| ----> |-----+-----|
| | | 2 | 0 | 1 | 2 | 3 |          | 28 | 29 | 30 | 31 |
|----+----| |-----+-----|
| | | 3 | 0 | 1 | 2 | 3 | ..... | 28 | 29 | 30 | 31 |
|----+----| |-----+-----|
| | | 4 | 32 | 33 | 34 | 35 |          | 60 | 61 | 62 | 63 |
|----+----| |-----+-----|
| | | 5 | 32 | 33 | .          | . | 62 | 63 |
|----+----| |-----+ . |-----+
| | | 6 | 32 | .          | . | 63 |
|----+----| |----+ .          | . |----+
| | | 7 | 32 | .          | . | 63 |
----- |----+ .          | . |----+
      8 | 64 | .      (64 x 64 blocks) | 95 |
Pattern 0 |----+ .          | . |----+
appears   | 64 | .          | . | 95 |
here as a |----+ .          | . |----+
result    | .          | .          |
          | .          | .          |
          | . | | | 1 data unit | .          |
          | . |----+ | corresponds | . |----+
          | . | 128 | . | | | to a 2 x 2 block | . | 159 |
          |----+ .          | . |----+
20 | 160 | .          | . | 191 |
          |-----+ .          | . |-----+
21 | 160 | 161 | .          | . | 190 | 191 |
          |-----+ |-----+
22 | 160 | 161 | 162 | 163 | ..... | 188 | 189 | 190 | 191 |

```



of the screen (0 to 15)

### 3.4 GRAPHIC 1 Mode

GRAPHIC 1 Mode is the screen mode as shown below:

```

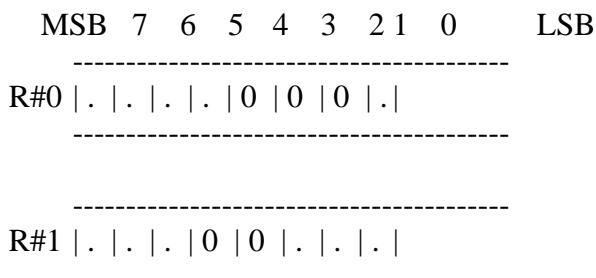
-----
| screen:      32 (horizontal) x 24 (vertical) patterns |
|              16 from 512 colours can be displayed   |
|              at the same time                       |
| pattern:    256 kinds of patterns are available     |
|              pattern size is 8 (horizontal) x 8 (vertical) dots |
|              any Figure can be defined for each pattern |
|              different colour for each 8 pattern can be set |
| memory requirements: for pattern font ..... 2048 bytes |
|              for colour table ..... 32 bytes        |
| sprite:     sprite mode 1                          |
| BASIC:      compatible with SCREEN 1                |
-----

```

#### 3.4.1 Setting GRAPHIC 1 mode

GRAPHIC 1 mode can be set as shown in Figure 4.26.

Figure 4.26 Setting GRAPHIC 1 mode

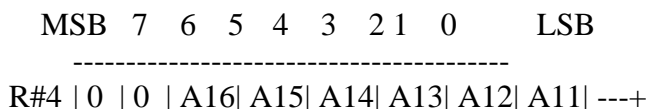


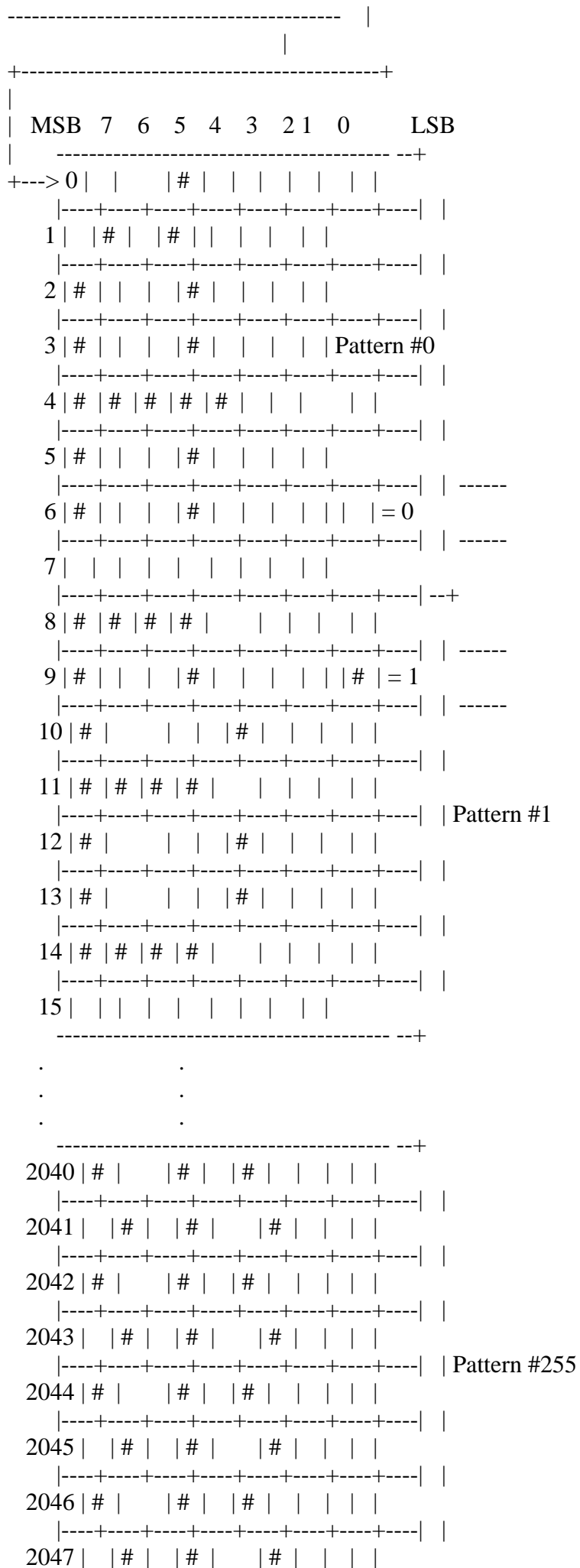
#### 3.4.2 Screen structure of GRAPHIC 1 mode

\* Pattern generator table

In this mode, 256 kinds of patterns, corresponding to codes 0 to 255, can be displayed on the screen. Fonts of each pattern are defined in the pattern generator table (see Figure 4.27). The starting address of the pattern generator table is specified by R#4. Note that only the 6 high order bits of the address (A16 to A11) are specified.

Figure 4.27 Pattern generator table of GRAPHIC 1 mode



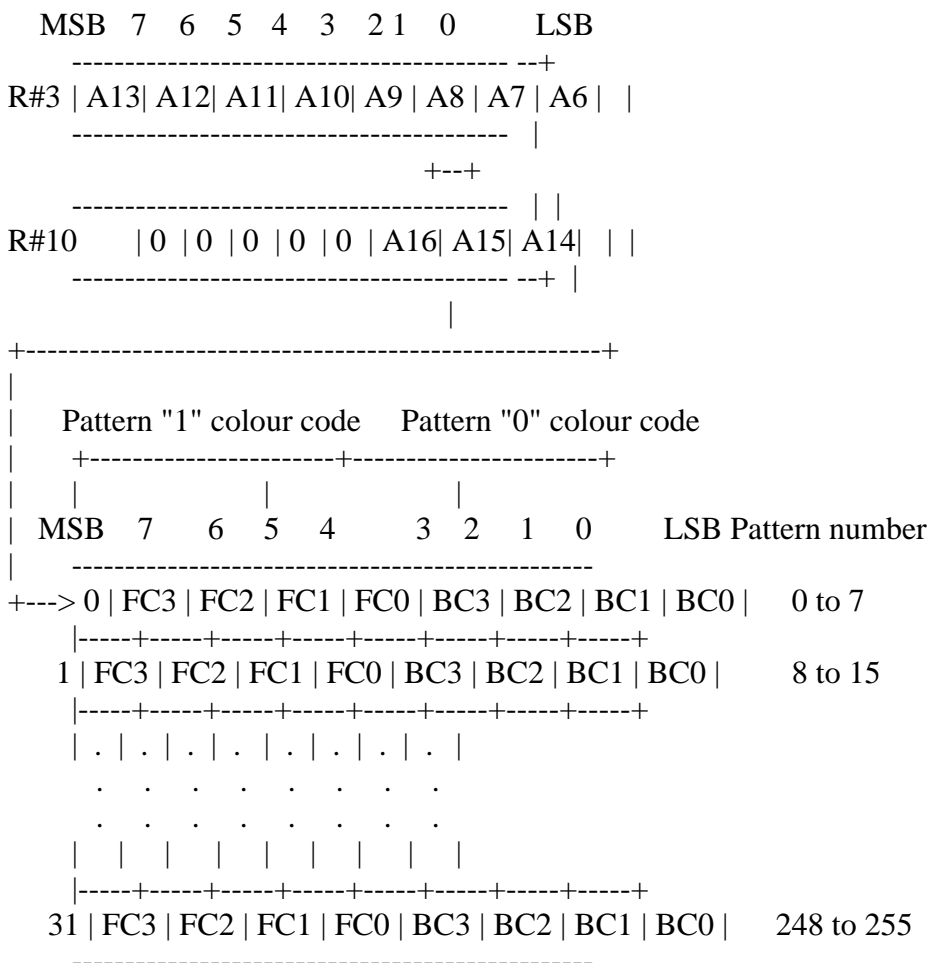


-----+  
 Pattern generator table

\* Colour table

The colour specification for each of the 8 patterns are done by the colour table. Colours for "0" and "1" of the bit of each pattern can be specified (see Figure 4.28). The starting address of the colour table is specified by R#3 and R#10. Note that only the 11 high order bits of the address (A16 to A6) are specified.

Figure 4.28 Colour table structure of GRAPHIC 1 mode



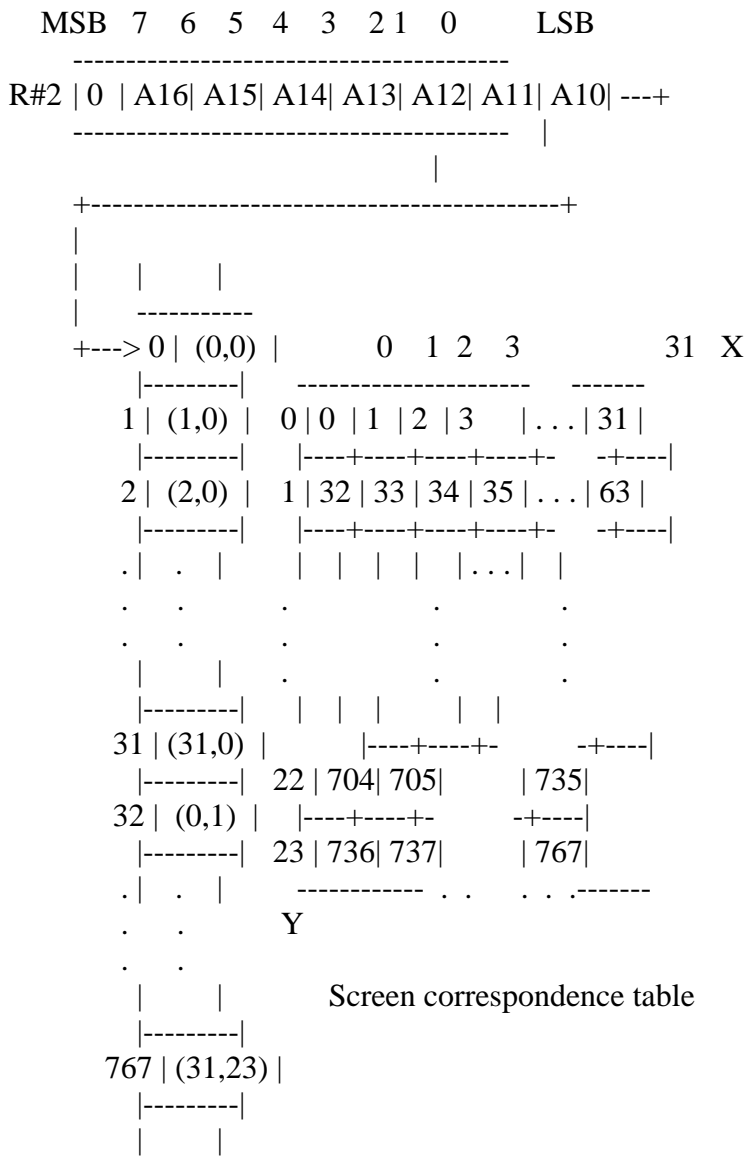
Colour table

\* Pattern name table

The size of the pattern name table is 768 bytes and the table corresponds to the pattern on the screen, one by one (see Figure 4.29). The starting address of the pattern name table is specified by R#2. Note that only the 7 high order bits of the address (A16 to A10) are specified.

Figure 4.29 Pattern name table structure of GRAPHIC 1 mode



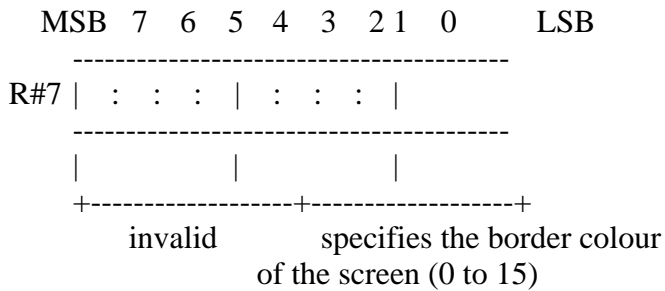


Pattern Name Table

### 3.4.3 Specifying the screen colour

The border colour of the screen can be specified by R#7 (see Figure 4.30).

Figure 4.30 Screen colour specification of GRAPHIC 1 mode



### 3.5 GRAPHIC 2, GRAPHIC 3 modes

GRAPHIC 2 and GRAPHIC 3 modes are the screen modes as described below:

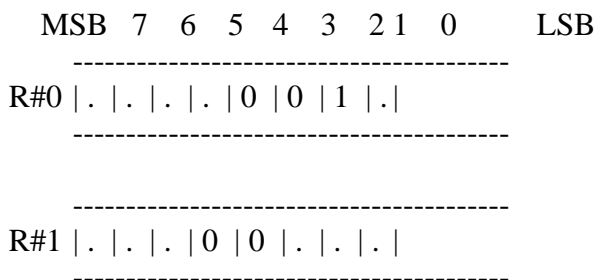
screen:	32 (horizontal) x 24 (vertical) patterns 16 from 512 colours can be displayed at the same time
pattern:	768 kinds of patterns are available pattern size is 8 (horizontal) x 8 (vertical) dots any Figure can be defined for each pattern only two colours can be used in horizontal 8 dots
memory requirements:	for pattern font ..... 6144 bytes for colour table ..... 6144 bytes
sprite:	sprite mode 1 for GRAPHIC 2 sprite mode 2 for GRAPHIC 3
BASIC:	compatible to SCREEN 2 for GRAPHIC 2 compatible to SCREEN 4 for GRAPHIC 3

### 3.5.1 Setting GRAPHIC 2, GRAPHIC 3 modes

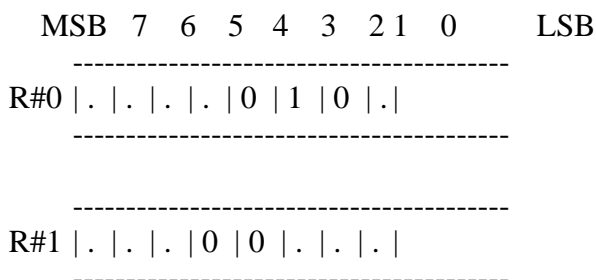
GRAPHIC 2, and GRAPHIC 3 modes are set as Figure 4.3.1.

Figure 4.3.1 Setting GRAPHIC 2, GRAPHIC 3 modes

#### GRAPHIC 2 mode setting



#### GRAPHIC 3 mode setting



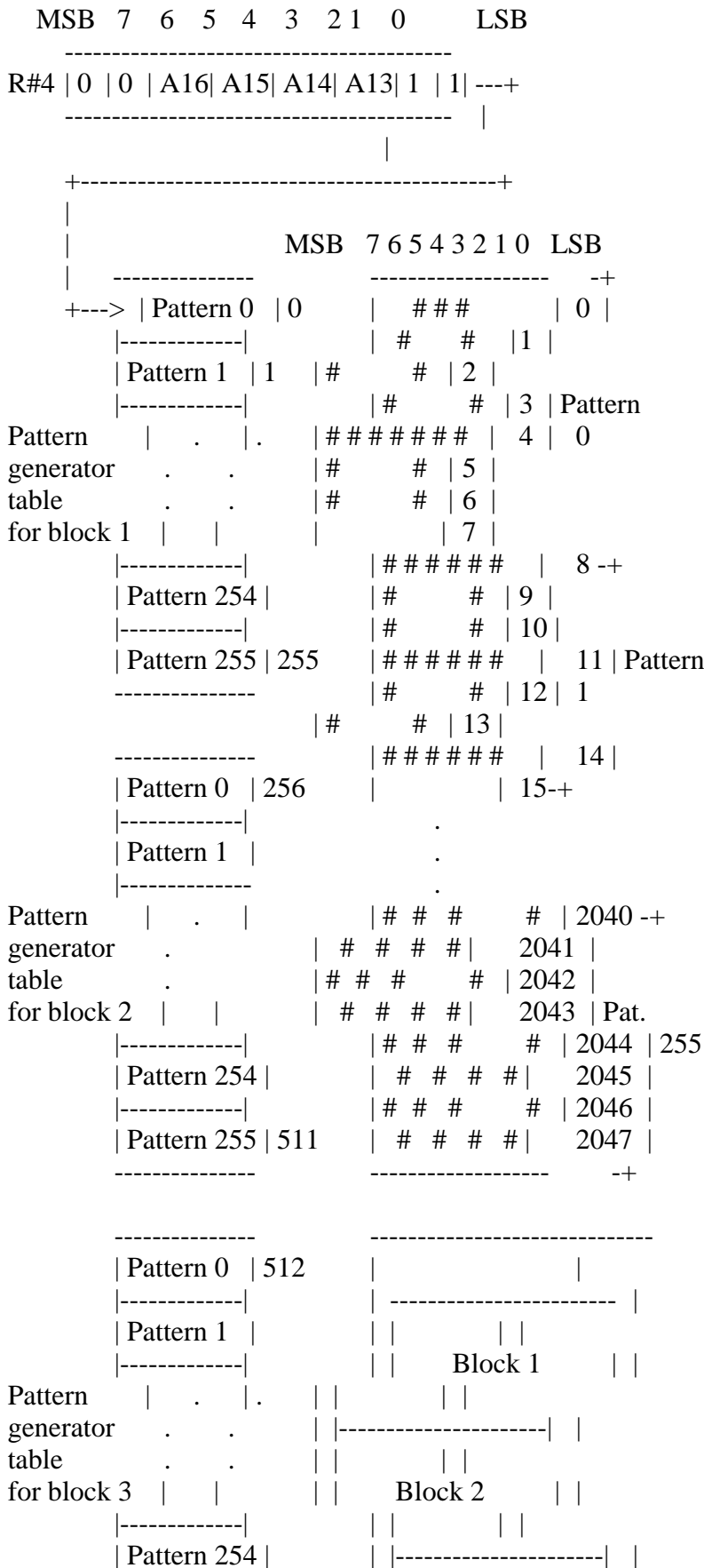
### 3.5.2 Screen structure of GRAPHIC 2, GRAPHIC 3 modes

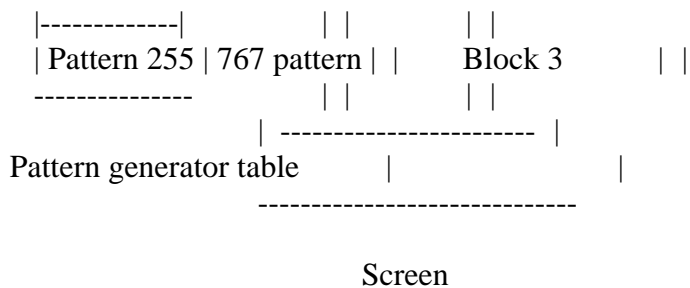
#### \* Pattern generator table

In this mode, there are three pattern generator tables which are compatible with GRAPHIC 1 and 768 patterns can be displayed. It cannot display patterns which are overlapped on the screen, and operating the pattern generator table in this case causes the 256 x 192 dot graphics display to be simulated. The

starting address of the pattern generator table is specified by R#4. Note that only 4 bits of the address (A16 to A13) are specified, so the address which can be set is located at interval steps of 8K bytes from 00000H (see Figure 4.32).

Figure 4.32 Pattern generator table structure of GRAPHIC 2, GRAPHIC 3

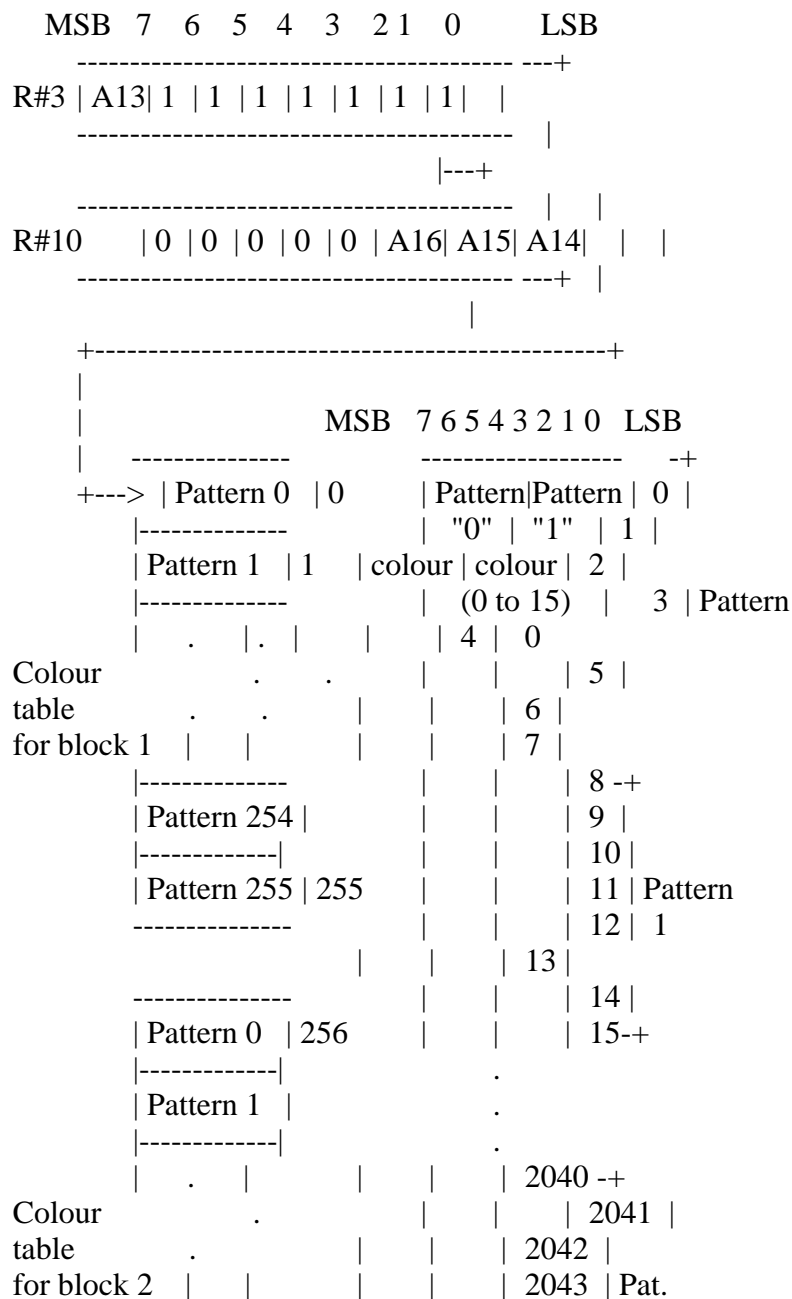


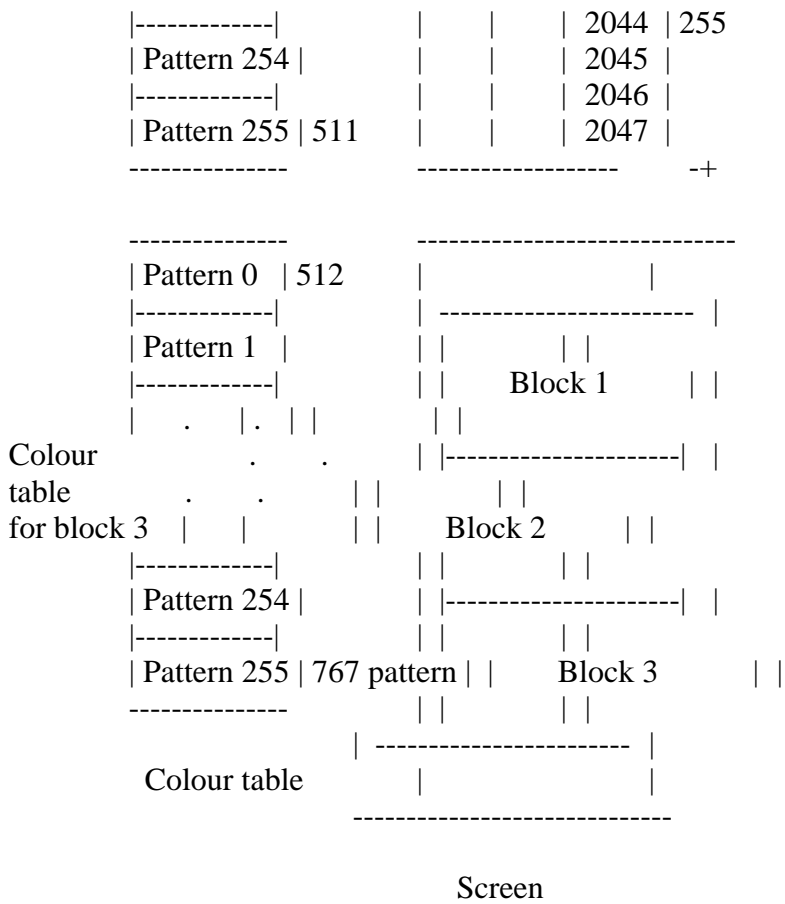


\* Colour table

The size of the colour table is the same as that of the pattern generator table and colours for "0" and "1" bits of each horizontal line of each pattern can be specified (see Figure 4.33). The starting address of the colour table is specified by R#3 and R#10. Note that only the 4 high order bits of the address (A16 to A13) is specified.

Figure 4.33 Colour table structure of GRAPHIC 2, GRAPHIC 3 modes

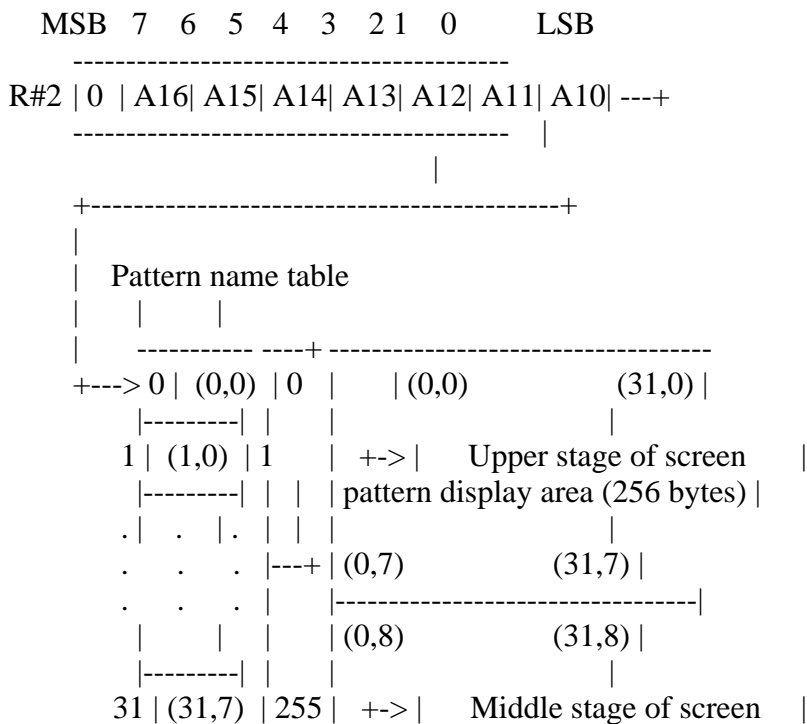


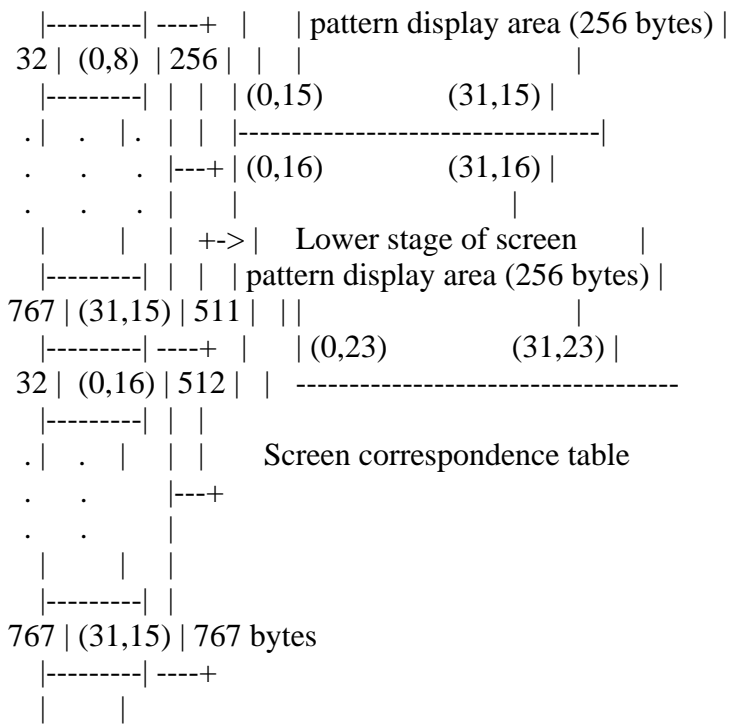


\* Pattern name table

The pattern name table is divided into three stages - upper, middle, and lower; each displays the pattern by referring to 256 bytes of the pattern generator (see Figure 4.34). This method enables each of 768 bytes on the pattern name table to display a different pattern font.

Figure 4.34 Pattern name table for GRAPHIC modes 2 and 3





Actual contents of fixed pattern name table

X axis		0		8		16		240		248		255	
Y axis	0	&H0	&H8						&HF0	&HF8			
	&H1	&H9							&HF1	&HF9			
	&H2	&HA							&HF2	&HFA			
	&H3	&HB	....						&HF3	&HFB			
	&H4	&HC							&HF4	&HFC			
	&H5	&HD							&HF5	&HFD			
	&H6	&HE							&HF6	&HFE			
	&H7	&HF							&HF7	&HFF			
8	&H100	&H108							&H1F0	&H1F8			
	&H101	&H109							&H1F1	&H1F9			
	&H102	&H10A							&H1F2	&H1FA			
	&H103	&H10B							&H1F3	&H1FB			
	&H104	&H10C	....						&H1F4	&H1FC			
	&H105	&H10D							&H1F5	&H1FD			
	&H106	&H10E							&H1F6	&H1FE			
	&H107	&H10F							&H1F7	&H1FF			
16	.	.							.	.			
.	.	.							.	.			
.	.	.	(256 x 192 dots)	.	.				.	.			
.	.	.							.	.			
184	&H1700	&H1708							&H17F0	&H17F8			
	&H1701	&H1709							&H17F1	&H17F9			
	&H1702	&H170A							&H17F2	&H17FA			
	&H1703	&H170B	....						&H17F3	&H17FB			
	&H1704	&H170C							&H17F4	&H17FC			
	&H1705	&H170D							&H17F5	&H17FD			

```

| &H1706 | &H170E | | &H17F6 | &H17FE |
191 | &H1707 | &H170F | | &H17F7 | &H17FF |
----- . . . -----

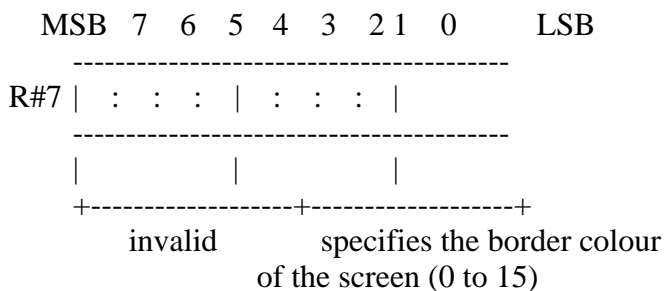
```

Note: The values are offset from the base address of the pattern generator table.

### 3.5.3 Screen colour specification

The border colour of the screen can be specified by R#7 (see Figure 4.35).

Figure 4.35 Screen colour specification of GRAPHIC 2, GRAPHIC 3 modes



## 3.6 GRAPHIC 4 Mode

GRAPHIC 4 mode is described below:

```

-----
| screen:      256 (horizontal) x 212 (vertical) dots |
|              (or, 192 vertical)                    |
|              16 colours can be displayed at the same time |
|              each of 16 colours can be selected from 512 colours|
| command:    high speed graphic by VDP command available |
| sprite:     mode 2 sprite function available          |
| memory requirements: for 192 dots                    |
|               bitmap screen ..... 24K bytes (6000H bytes) |
|               (4 bits x 256 x 192)                   |
|               for 212 dots                            |
|               bitmap screen ..... 26.5K bytes (6A00H bytes) |
|               (4 bits x 256 x 212)                   |
| BASIC:      compatible to SCREEN 5                  |
-----

```

### 3.6.1 Setting GRAPHIC 4 mode

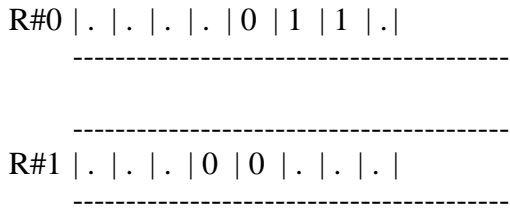
Set GRAPHIC 4 mode as shown in Figure 4.36.

Figure 4.36 GRAPHIC 4 mode setting

```

MSB 7 6 5 4 3 2 1 0 LSB
-----

```

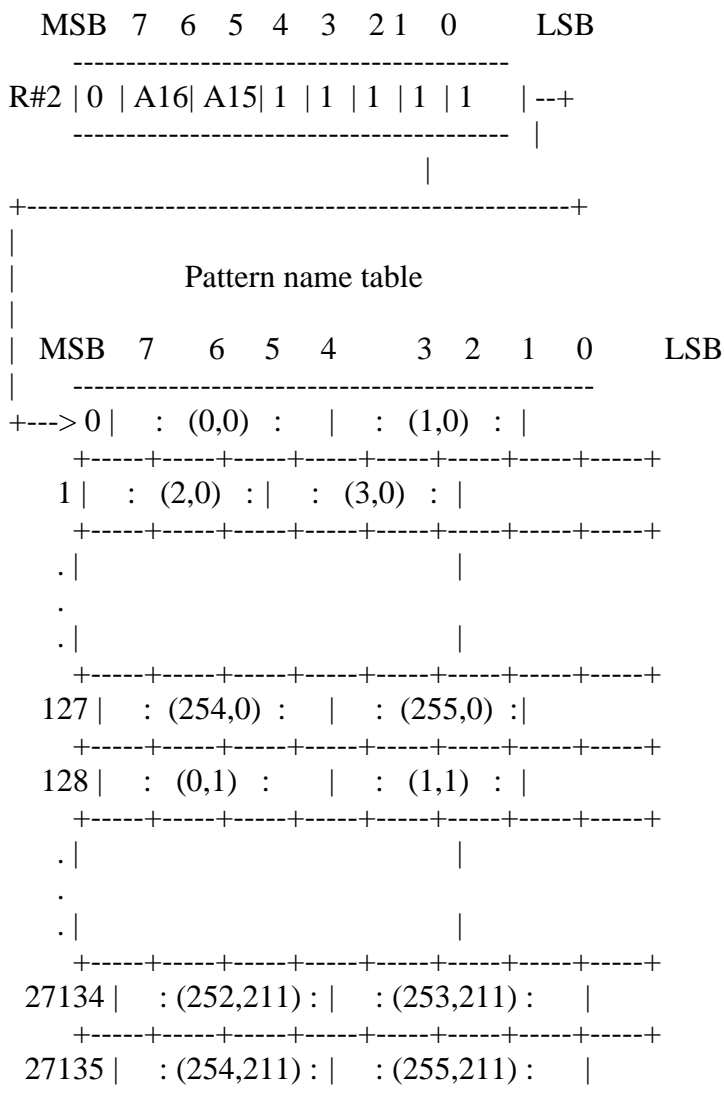


### 3.6.2 Screen structure of GRAPHIC 4 mode

#### \* Pattern name table

In GRAPHIC 4 mode, one byte of the pattern name table corresponds with 2 dots on the screen. The colour information of each dot is represented by 4 bits and 16 colours can be specified (see Figure 4.37). The starting address of the pattern name table is specified by R#2. Only the 2 high order bits of the address (A16 to A15) are specified and the 15 low order bits are considered as "0". Thus, the four addresses at which the pattern name can be set are 00000H, 08000H, 10000H, and 18000H.

Figure 4.37 Pattern name table structure of GRAPHIC 4 mode





This table shows how colour codes are set for each dot. (0 to 15)

	+-----> X			
		.....		
V		0,0   1,0		254,0   255,0
		+-----+		+-----+
Y		0,1		255,1
		+-----+		+-----+
		.		.
		.	X,Y	.
		.		.
		+-----+		+-----+
		0,191   LN = 0		255,191
		+-----+		+-----+
		0,211   LN = 1		255,211
		+-----+		+-----+

Screen correspondence table

The dot at (X,Y) coordinate on the screen can be accessed by using Expression 4.1. The program of List 4.2 illustrates the use of Expression 4.1.

Expression 4.1 The expression for accessing the dot at (X,Y) coordinate

$$\text{ADR} = X/2 + Y * 128 + \text{base address}$$

(The colour of the dot is represented by 4 high order bits in the case that X is even and by 4 low order bits in the case that X is odd.)

List 4.2 PSET for GRAPHIC 4 mode written in BASIC

```

=====
100 '*****
110 ' LIST 4.2 dot access of GRAPHIC 4 mode
120 '*****
130 '
140 SCREEN 5
150 BA=0
160 FOR I=0 TO 255
170 X=I:Y=I\2
180 COL=15
190 GOSUB 1000
200 NEXT
210 END
220 '
1000 '*****
1010 ' PSET (X,Y),COL

```

```

1020 ' COL:color BA:graphics Base Address
1030 '*****
1040 '
1050 ADR=X\2+Y*128+BA
1060 IF X AND 1 THEN BIT=&HF0:C=COL ELSE BIT=&HF:C=COL*16
1070 D=VPEEK(ADR)
1080 D=(D AND BIT) OR C
1090 VPOKE ADR,D
1100 RETURN

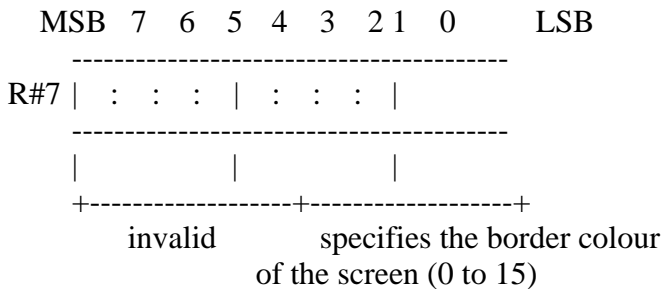
```

---

### 3.6.3 Screen colour specification

The border colour of the screen can be specified by R#7 (see Figure 4.38).

Figure 4.38 Screen colour specification in GRAPHIC 4 mode



### 3.7 GRAPHIC 5 Mode

GRAPHIC 5 mode is described as follows:

---

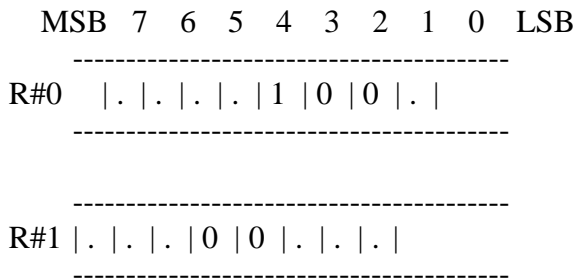
screen:	512 (horizontal) x 212 (vertical) dots (or, 192 vertical)	
	4 colours can be displayed at the same time each of 4 colours can be selected from 512 colours	
command:	graphic command by hardware available	
sprite:	mode 2 sprite function available	
memory requirements:	for 192 dots bitmap screen ..... 24K bytes (6000H bytes) (2 bits x 512 x 192)	
	for 212 dots bitmap screen ..... 26.5K bytes (6A00H bytes) (2 bits x 512 x 212)	
BASIC:	compatible to SCREEN 6	

---

#### 3.7.1 Setting GRAPHIC 5 mode

Set GRAPHIC 5 mode as shown in Figure 4.39.

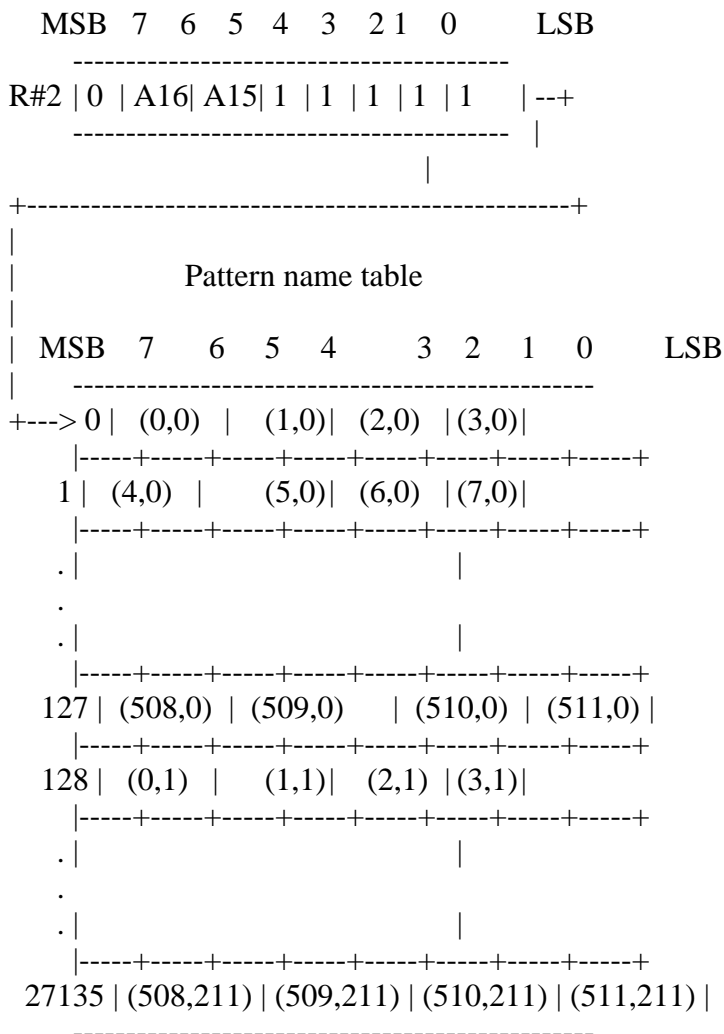
Figure 4.39 GRAPHIC 5 mode setting



### 3.7.2 Pattern name table

In GRAPHIC 5 mode, one byte of the pattern name table corresponds with 4 dots on the screen. The colour information of each dot is represented by 2 bits and 4 colours can be specified. As with GRAPHIC 4 mode, the pattern name table is set by writing 2 high order bits of the address in R#2. The addresses can be set at either 00000H, 08000H, 10000H, or 18000H (see Figure 4.40).

Figure 4.40 Pattern name table structure of GRAPHIC 5 mode



This table shows how colour codes are set for each dot. (0 to 3)

```

+-----> X
|
| .....
| | 0,0 | 1,0 | | 510,0 | 511,0 |
V |-----+-----+-----+-----|
| | 0,1 | | | 511,1 | |
Y |-----+-----+-----+-----|
| | . | | | . | |
| | . | | X,Y | | . |
| | . | | | . |
| |-----+-----+-----+-----|
| | 0,191 | LN = 0 | 511,191 |
|-----+-----+-----+-----|
| | | |
|-----+-----+-----+-----|
| | 0,211 | LN = 1 | 511,211 |
|-----+-----+-----+-----|

```

Screen correspondence table

The dot at (X,Y) coordinate on the screen can be accessed by using Expression 4.2. The program of List 4.3 confirms Expression 4.2.

Expression 4.2 The expression for accessing the dot at (X,Y) coordinate

$$| \text{ADR} = X/4 + Y * 128 + \text{base address} |$$

(The colour of the dot is represented by bit 7 and 6, or 5 and 4, or 3 and 2, or 1 and 0, when X MOD 4 is 0, or 1, or 2, or 3, respectively.)

List 4.3 PSET for GRAPHIC 5 mode written in BASIC

```

=====
100 '*****
110 ' LIST 4.3 dot access of GRAPHIC 5 mode
120 '*****
130 '
140 SCREEN 6
150 BA=0
160 FOR I=0 TO 511
170 X=I : Y=I\2
180 COL=3
190 GOSUB 1000
200 NEXT
210 END
220 '
1000 '*****
1010 ' PSET(X,Y)
1020 ' COL:colour BA:graphic Base Address
1030 '*****

```

```

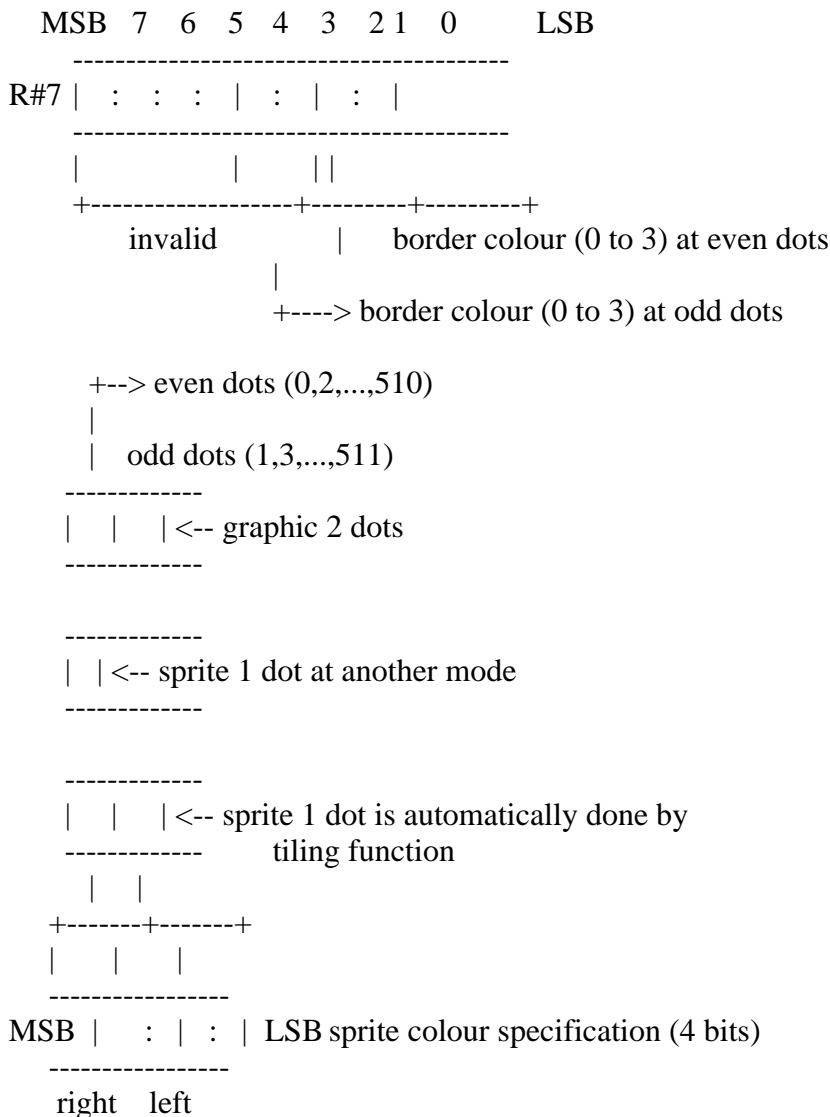
1040 '
1050 ADR=X\4+Y*128+BA
1060 LP=X MOD 4
1070 IF LP=0 THEN BIT=&H3F:C=COL*&H40
1080 IF LP=1 THEN BIT=&HCF:C=COL*&H10
1090 IF LP=2 THEN BIT=&HF3:C=COL*&H4
1100 IF LP=3 THEN BIT=&HFC:C=COL
1110 D=VPEEK(ADR)
1120 D=(D AND BIT) OR C
1130 VPOKE ADR,D
1140 RETURN

```

### 3.7.3 Setting the screen colour

In GRAPHIC 5 mode, hardware tiling is done for the border colour of the screen and sprites. As with the other modes, these colours are specified by 4 bits; 2 high order bits of 4 bits represents the dot colour at even locations, and 2 low order bits for the dot colour at odd locations (see Figure 4.41).

Figure 4.41 Screen colour specification in GRAPHIC 5 mode



side side  
 colour colour  
 (0 to 3) (0 to 3)

### 3.8 GRAPHIC 6 Mode

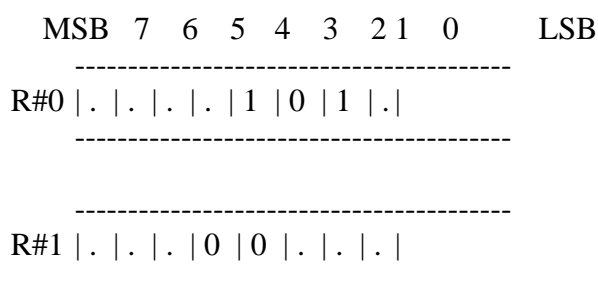
GRAPHIC 6 mode is described as follows:

screen:	512 (horizontal) x 212 (vertical) dots (or, 192 vertical)
	16 colours can be displayed at the same time each of 16 colours can be selected from 512 colours
command:	graphic command by hardware available
sprite:	mode 2 sprite function available
memory requirements:	for 192 dots bitmap screen ..... 48K bytes (C000H bytes) (4 bits x 512 x 192)
	for 212 dots bitmap screen ..... 53K bytes (D400H bytes) (4 bits x 512 x 212)
	Note that this mode cannot be used at all with 64K byte VRAM because of the hardware
BASIC:	compatible to SCREEN 7

#### 3.8.1 Setting GRAPHIC 6 mode

Set GRAPHIC 6 mode as shown in Figure 4.42.

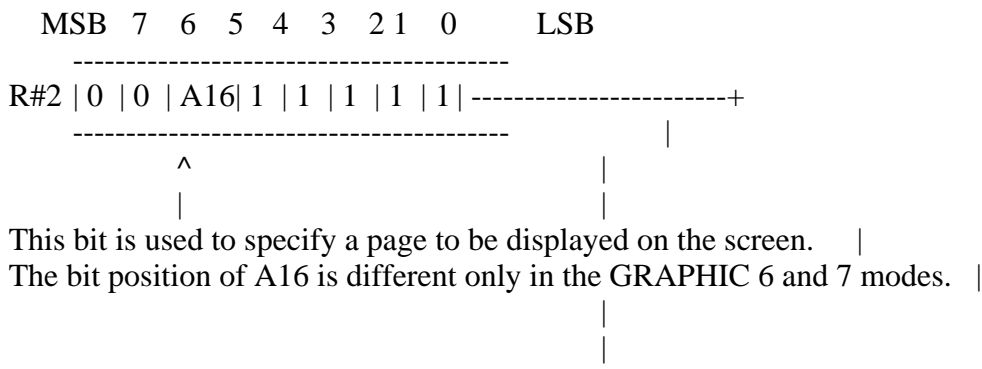
Figure 4.42 GRAPHIC 6 mode setting



#### 3.8.2 Pattern name table

In GRAPHIC 6 mode, one byte of the pattern name table corresponds with 2 dots on the screen. The colour information of each dot is represented by 4 bits and 16 colours can be specified (see Figure 4.43). The starting address of the pattern name table is set by writing the high order bit of the address in R#2. The two addresses at which the pattern name table can be set are either 00000H or 10000H. The dot at (X,Y) coordinate on the screen can be accessed by using Expression 4.3. The program of List 4.4 illustrates this.

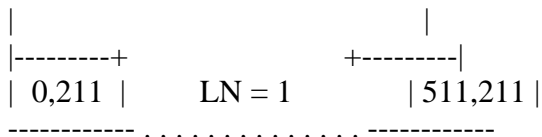
Figure 4.43 Pattern name table structure of GRAPHIC 6 mode



Pattern name table									
MSB	7	6	5	4	3	2	1	0	LSB
0	:	(0,0)	:	:	(1,0)	:	:	:	
1	:	(2,0)	:	:	(3,0)	:	:	:	
.	:	.	:	.	.	:	.	.	
255	:	(510,0)	:	:	(511,0)	:	:	:	
256	:	(0,1)	:	:	(1,1)	:	:	:	
.	:	.	:	.	.	:	.	.	
54270	:	(508,211)	:	:	(509,211)	:	:	:	
54271	:	(510,211)	:	:	(511,211)	:	:	:	

This table shows how colour codes are set for each dot. (0 to 15)

+-----> X									
	0,0	1,0			510,0	511,0			
V	+-----+				+-----+				
	0,1				511,1				
Y	+-----+				+-----+				
	.				.				
	.	X,Y			.				
	.				.				
	+-----+				+-----+				
	0,191	LN = 0			511,191				



Screen correspondence table

The dot at (X,Y) coordinate on the screen can be accessed by using Expression 4.1. The program of List 4.2 illustrates the use of Expression 4.1.

Expression 4.3 The expression for the access to the dot at (X,Y) coordinate

$$| \text{ADR} = X/2 + Y * 256 + \text{base address} |$$

(The colour of the dot is represented by 4 high order bits in the case that X is even and by 4 low order bits in the case that X is odd.)

List 4.4 PSET for GRAPHIC 6 mode written in BASIC

```

=====
100 '*****
110 ' LIST 4.4 dot access of GRAPHIC 6 mode
120 '*****
130 '
140 SCREEN 7
150 BA=0
160 FOR I=0 TO 511
170 X=I: Y=I\2: COL=15: GOSUB 1000
180 NEXT
190 END
200 '
1000 '*****
1010 ' PSET (X,Y)
1020 ' COL:color BA:graphic Base Address
1030 '*****
1040 '
1050 ADR=X\2+Y*256+BA
1060 IF X AND 1 THEN BIT=&HF: C=COL ELSE BIT=&HF0: C=COL*16
1070 VPOKE ADR,(VPEEK(ADR) AND BIT) OR C
1080 RETURN
=====

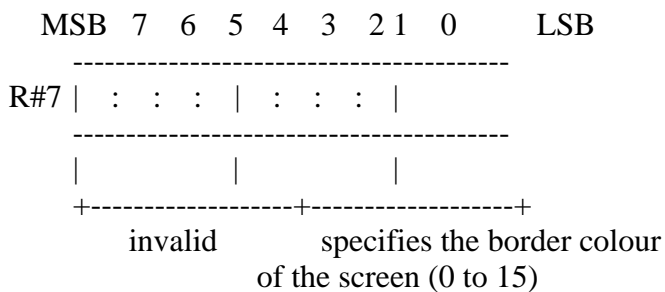
```

### 3.8.3 Setting screen colour

The border colour of the screen can be specified by R#7 (see Figure 4.44).

Figure 4.44 Screen colour specification in GRAPHIC 6 mode





### 3.9 GRAPHIC 7 Mode Use

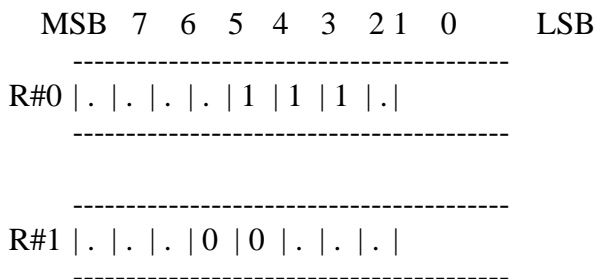
GRAPHIC 7 mode is described as follows:

screen:	256 (horizontal) x 212 (vertical) dots (or, 192 vertical)
	256 colours can be displayed at the same time
command:	graphic command by hardware available
sprite:	mode 2 sprite function available
memory requirements:	for 192 dots
	bitmap screen ..... 48K bytes (C000H bytes) (8 bits x 256 x 192)
	for 212 dots
	bitmap screen ..... 53K bytes (D400H bytes) (8 bits x 256 x 212)
	Note that this mode cannot be used with 64K byte VRAM machines, as in the case of GRAPHIC 6
BASIC:	compatible to SCREEN 8

#### 3.9.1 Setting GRAPHIC 7 mode

Set GRAPHIC 7 mode as shown in Figure 4.45.

Figure 4.45 GRAPHIC 4 mode setting



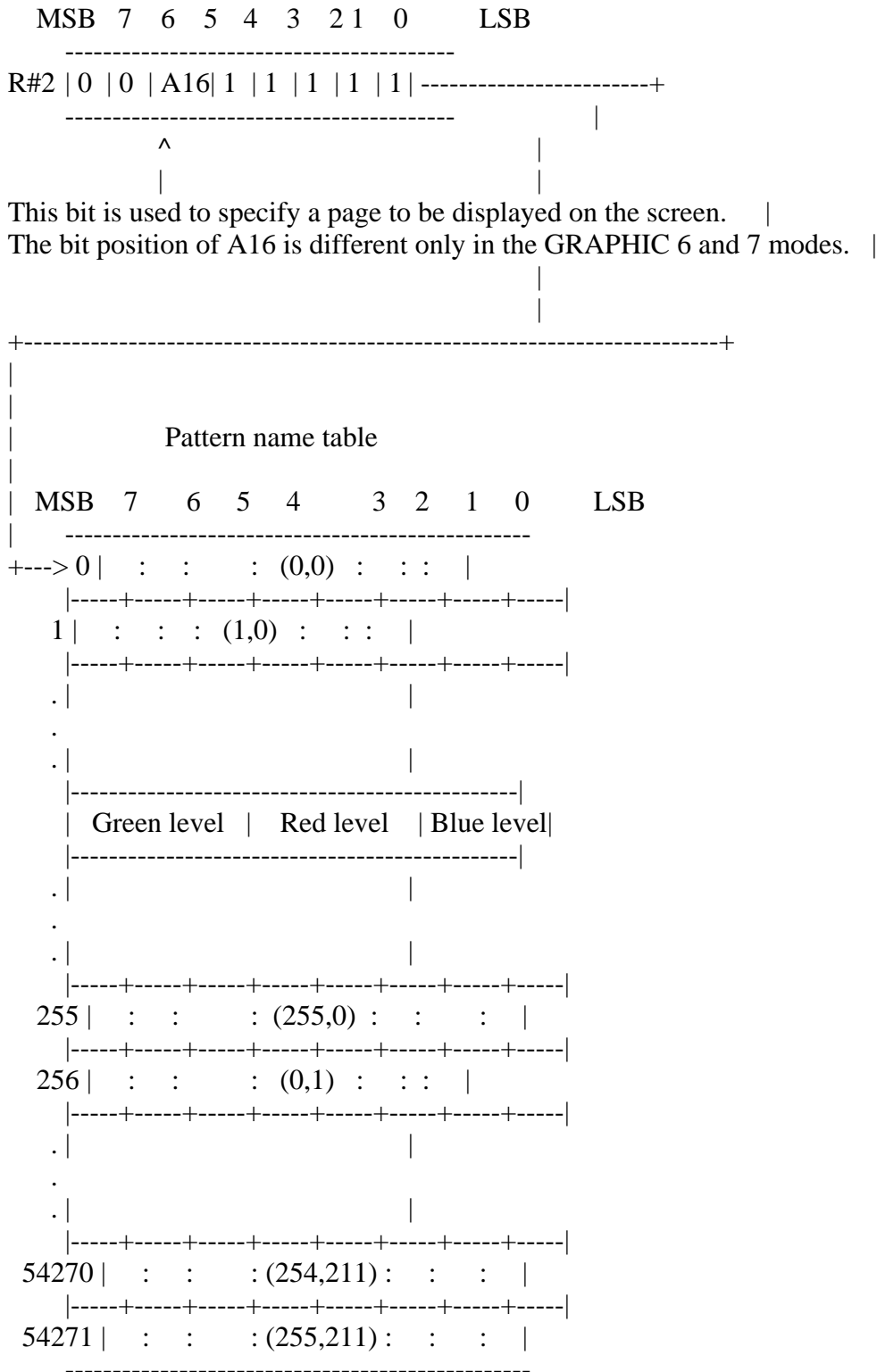
#### 3.9.2 Pattern name table

Configuration of GRAPHIC 7 mode is the simplest of all modes; one dot on the screen corresponds with one byte in the pattern name table. The value of one byte written in the table represents 256 kinds of colours. The starting address of the pattern name table is set by R#2. The two addresses at which

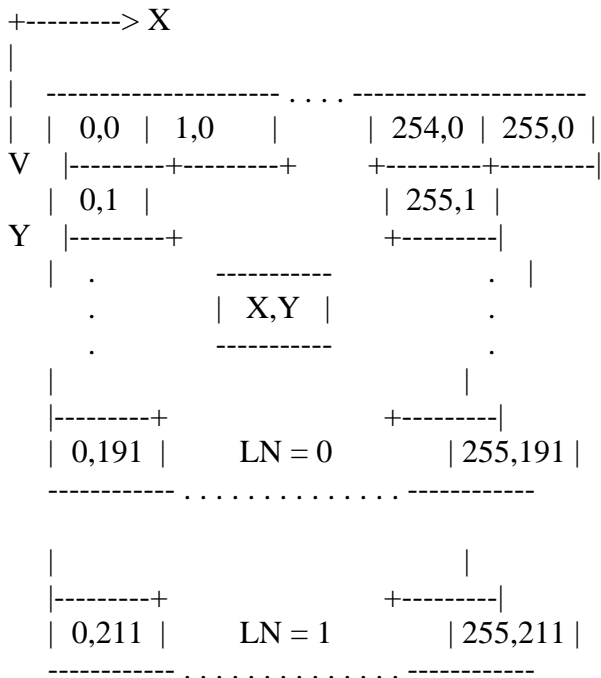
the pattern name table can be set are either 00000H or 10000H (see Figure 4.46).

One byte of data represents the intensity of 3 bits for green, 3 bits for red, and 2 bits for blue, as shown in Figure 4.47. The dot at (X,Y) coordinate on the screen can be accessed by using Expression 4.4.

Figure 4.46 Pattern name table structure of GRAPHIC 7 mode

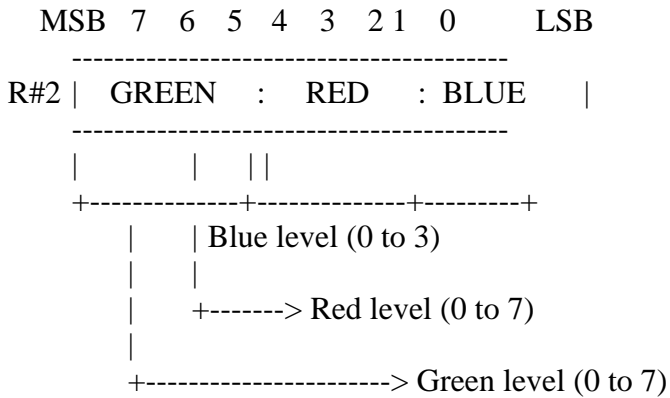


This table shows how colour codes are set for each dot. (0 to 255)



Screen correspondence table

Figure 4.47 RGB brightness information



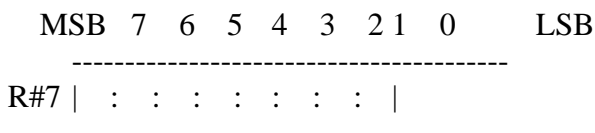
Expression 4.4 The expression for accessing to the dot at (X,Y) coordinate

ADR = X + Y * 256 + base address	
----------------------------------	--

### 3.9.3 Setting the screen colour

The border colour of the screen can be specified by R#7 (see Figure 4.48).

Figure 4.48 Screen colour specification in GRAPHIC 7 mode



| |  
 +-----+  
 specifies the border colour of the screen (0 to 255)

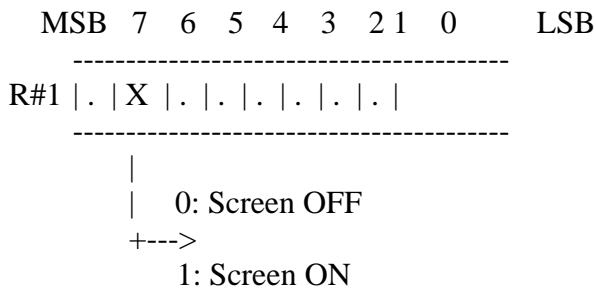
#### 4. MISCELLANEOUS FUNCTIONS FOR THE SCREEN DISPLAY

Detailed settings for the screen display are available in MSX-VIDEO. These include screen ON/OFF and specification of the display location. These MSX-VIDEO functions are described in this function.

##### \* Screen ON/OFF

The screen ON/OFF function is controlled by bit 6 of R#1 (see Figure 4.49). When set OFF, the entire screen changes to the colour specified by the 4 low order bits of R#7 (8 bits in GRAPHIC 7 mode). Drawing with the VDP commands is faster when the screen is set OFF.

Figure 4.49 Screen ON/OFF



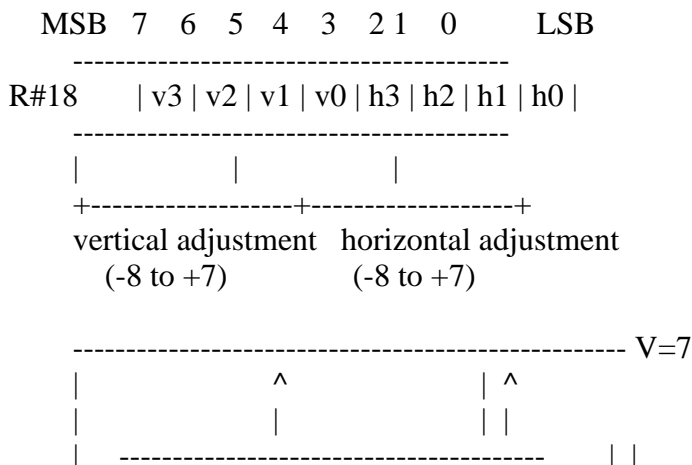
BASIC program lines:

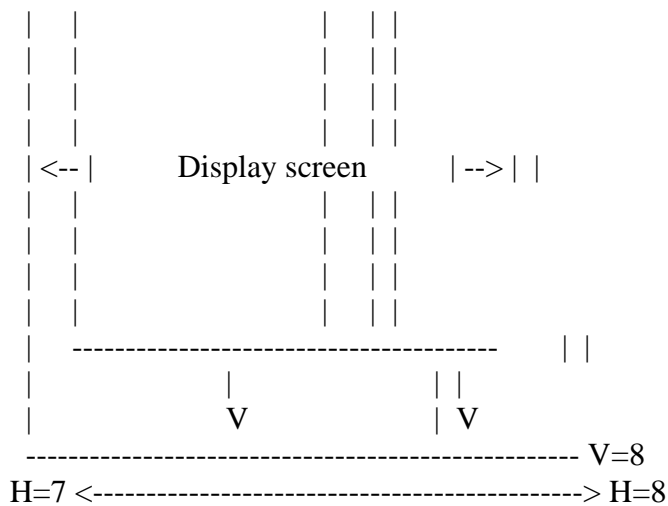
```
VDP(1)=VDP(1) AND &B10111111 <-- Screen OFF
VDP(1)=VDP(1) OR &B01000000 <-- Screen ON
```

##### \* Adjustment of the display location on the screen

R#18 is used for adjusting the display location on the screen (see Figure 4.50). This corresponds with the "SET ADJUST" instruction of BASIC.

Figure 4.50 Adjustment of the screen display

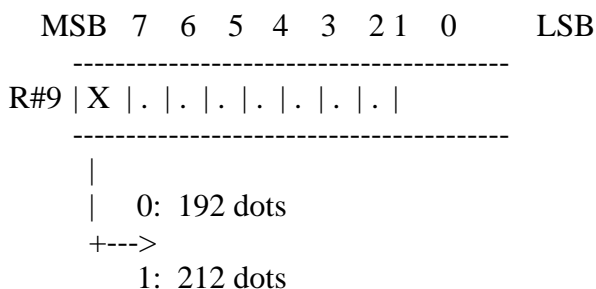




\* Switching the number of pixels in the Y direction

The number of dots displayed in the Y direction on the screen can be switched to either 192 dots or 212 dots by setting bit 7 of R#9 to 0 or 1. This function is only valid for five screen modes, TEXT 2, and GRAPHIC 4 to GRAPHIC 7 modes. When 212 dots are set in TEXT 2 mode, the number of text lines is 26.5 (=212/8) and on the 27th line only the upper halves of characters are displayed.

Figure 4.51 Switching the number of dots in the vertical direction



BASIC program lines:

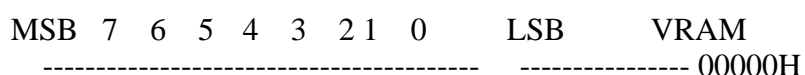
```
VDP(10)=VDP(10) AND &B01111111 <-- 192 dots
VDP(10)=VDP(10) OR &B10000000 <-- 212 dots
```

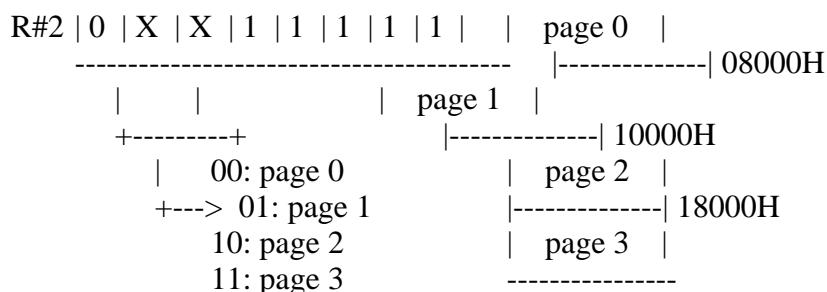
\* Switching the display page

In GRAPHIC modes 4 to 7, the display pages can be easily switched by setting the starting address of the pattern name table using R#2. In fact, the second parameter of the "SET PAGE" BASIC instruction switches the display page this way.

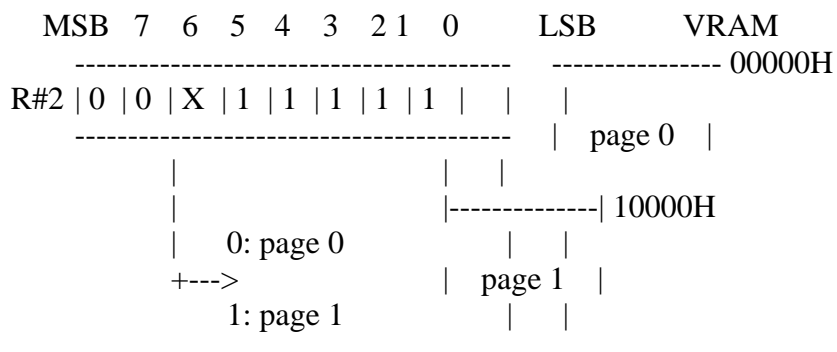
Figure 4.52 Switching pages

\* GRAPHIC modes 4 and 5





\* GRAPHIC modes 6 and 7

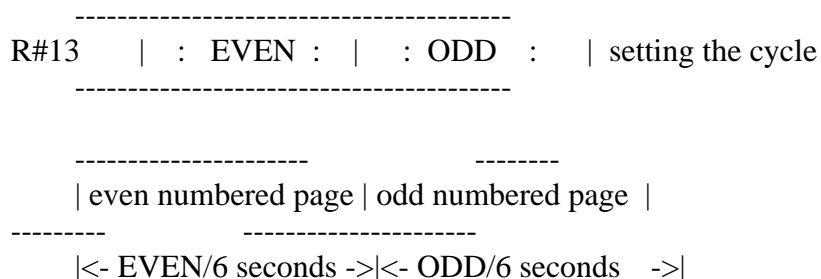


\* Automatic alternate screen display

In GRAPHIC modes 4 to 7, two pages can be displayed alternately by using the following method. Either page 0 and page 1, or page 2 and page 3 can be displayed alternately.

To begin the alternate display, select the odd-numbered page (1 or 3) using R#2 and set the screen alternation rate in R#13. The 4 high order bits of R#13 represent the time for displaying the even page and the 4 low order bits represent the time for displaying the odd page. The time is set in 1/6 seconds interval. Setting 0 for both time periods causes only the odd page to be displayed.

Figure 4.53 Setting the rate of the screen alternation



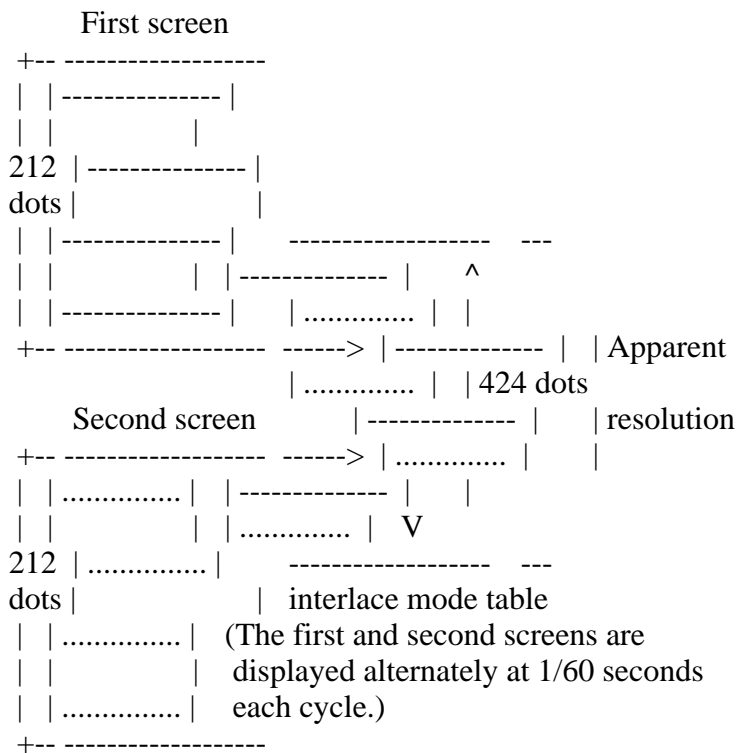
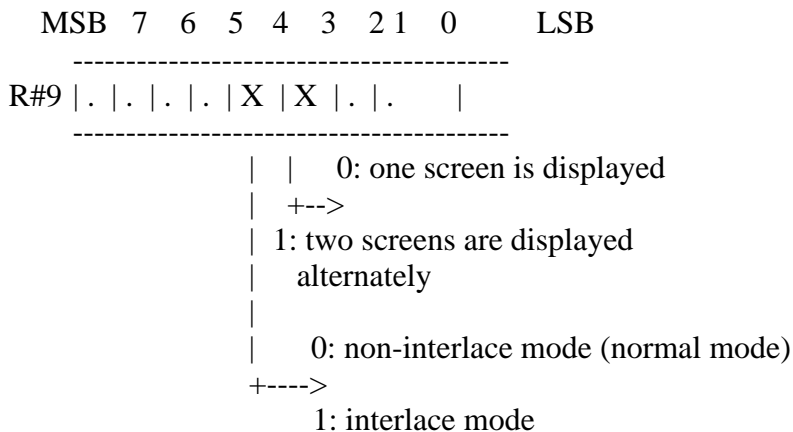
\* Setting the interlaced mode

The interlaced mode allows an apparent screen resolution in the Y direction of double the normal mode. A resolution of up to 424 dots in the Y direction can be achieved using this mode. This is done by alternating at high speed the normal screen and a screen whose scanning lines are offset vertically by half a line. In MSX-VIDEO the interlaced mode is specified by setting bit 3

of R#9 to "1". The two screens are switched 60 times a second.

When the odd page is selected in GRAPHIC 4 to GRAPHIC 7 screen modes and the alternate screen display mode is selected, the screen is normally switched at slow rates specified in units of 1/6 seconds. However, combining this function and the interlaced function can make the number of the vertical dots of the display screen seem double.

Figure 4.54 Setting the interlaced mode



List 4.5 Interlaced mode example

```

=====
1000 '*****
1010 ' List 4.5 interlace mode
1020 '*****
1030 '
1040 COLOR 15,0,0 : SCREEN 5,,,,,0 'noninterlace mode
1050 '
  
```

```

1060 SET PAGE 0,0 : CLS
1070 LINE (32,0)-(64,120),15,BF
1080 SET PAGE 1,1 : CLS
1090 LINE (192,91)-(224,211),15,BF
1100 '
1110 VDP(10)=VDP(10) OR &B00001100  'interlace mode!!!
1120 '
1130 FOR I=32 TO 192
1140  SET PAGE 1,0
1150  LINE (I,0)-STEP(0,120),0
1160  LINE (I+33,0)-STEP(0,120),15
1170  SET PAGE 1,1
1180  LINE (256-I,91)-STEP(0,120),0
1190  LINE (221-I,91)-STEP(0,120),15
1200 NEXT I
1210 '
1220 VDP(10)=VDP(10) AND &B11110011  'interlace off

```

---

\* Vertical scroll of the screen

R#23 is used to set the line at which display begins on the screen. Changing this register enables vertical scrolling of the screen. Note that, since the scroll is done every 256 lines, the sprite tables should be moved to another page. List 4.6 shows an example.

List 4.6 Vertical scroll example

---

```

1000 '*****
1010 ' List 4.6 Hardware scroll
1020 '*****
1030 '
1040 SCREEN 5,2: COLOR 15,0,0: CLS
1050 COPY (0,0)-(255,43) TO (0,212),,PSET 'erase (212,0)-(255,255)
1060 '
1070 FOR I=1 TO 8: D(I)=VAL(MID$("00022220",I,1))-1: NEXT
1080 '
1090 OPEN "GRP:" AS #1
1100 FOR I=0 TO 3
1110  PRESET (64,I*64): PRINT #1,"Hit CURSOR Key"
1120 NEXT
1130 '
1140 J=STICK(0)
1150  P=(P+D(J)) AND &HFF
1160  VDP(24)=P
1170 GOTO 1140

```

---

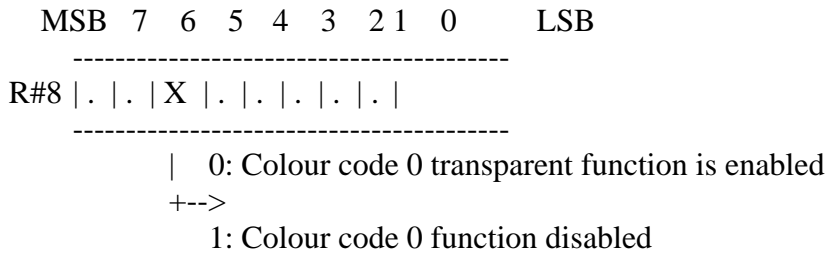
\* Specifying the colour code 0 function

Among the 16 colour codes, only code 0 can be made as a "transparent" colour



(the border colour of the screen can be set transparently), the colour set in palette P#0. Setting bit 5 of R#8 to "1" disables this function and the colour code 0 changes to the colour defined by the palette P#0.

Figure 4.55 Colour code 0 function

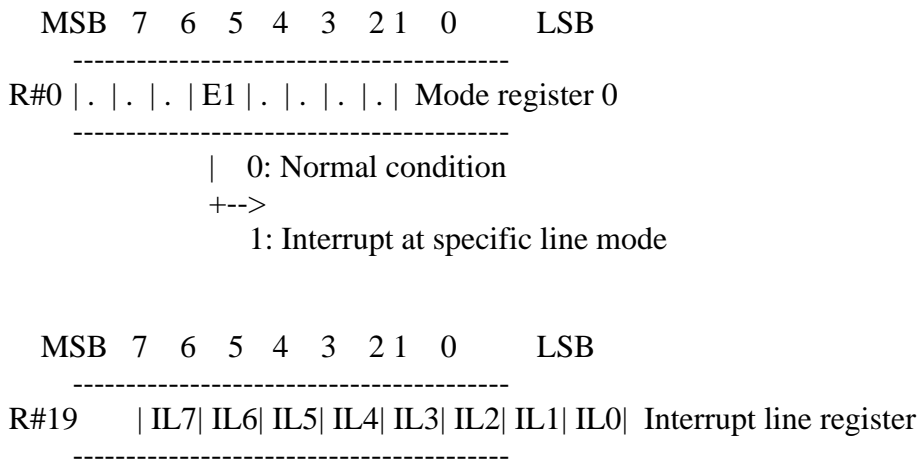


- When the TB bit is "0", colour code 0 becomes transparent.
- When the TB bit is "1", colour code 0 changes to the colour defined by palette P#0.

\* Generating interrupts by the scanning line location

In MSX-VIDEO an interrupt can be generated just after the CRT finishes displaying a specific scanning line. Set in R#19 the number of the scanning line at which the interrupt should be generated, and set bit 4 of R#0 to "1" (see Figure 4.56).

Figure 4.56 Generating the scanning line interrupt



## 5. SPRITES

Sprites are used to display movable character patterns of 8 x 8 or 16 x 16 dots on the screen. This function is especially useful in the programming of games.

The parameters specified are the X and Y coordinates, the character number, and the colour code. The sprite is displayed by writing this data to the preset sprite attribute table.

There are two modes for MSX2 sprites. Mode 1 is compatible to the TMS9918 used in the MSX1 machines. Mode 2 includes several improved functions and has



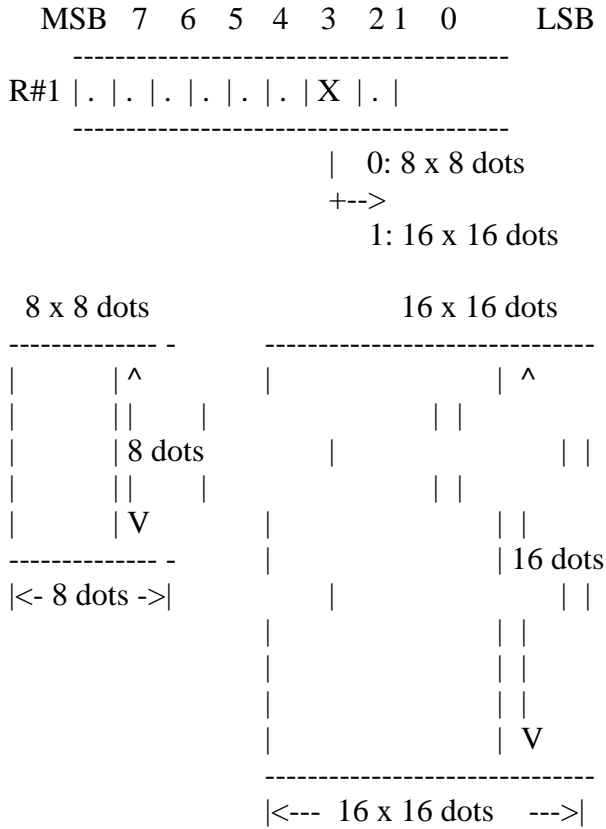
### 5.2.2 Sprite display settings

The following descriptions are settings to display the sprite.

#### \* Setting the size of the sprite

8 x 8 dots or 16 x 16 dots can be set (see Figure 4.58). By default, 8 x 8 dots is selected.

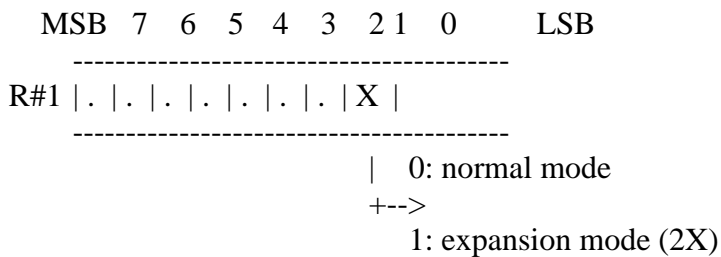
Figure 4.58 Setting the size of the sprite



#### \* Expanding the sprite

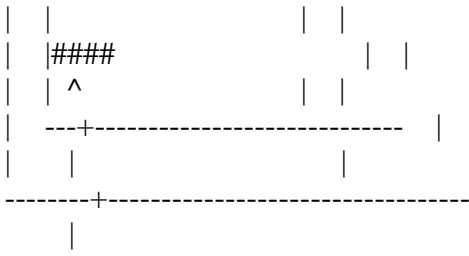
Figure 4.59 shows how to select whether one dot of the sprite corresponds to one dot of the screen or whether it is expanded double in both the horizontal and vertical directions. By default, the one dot to one dot size is selected.

Figure 4.59 Expanding the sprite

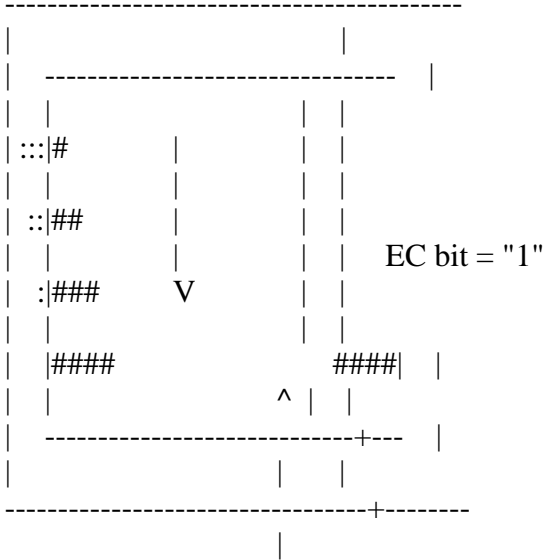








The sprite will display from here (X coordinate = 0)

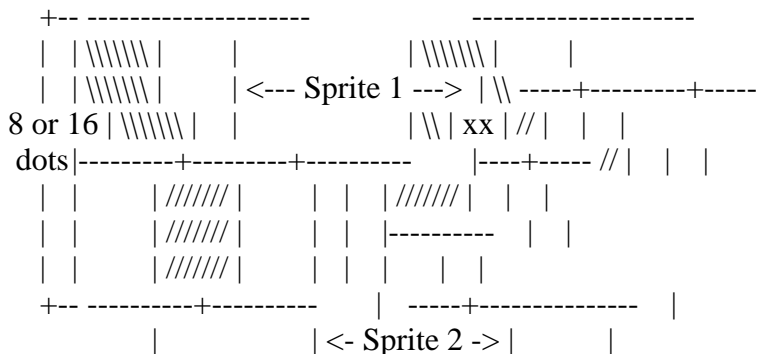
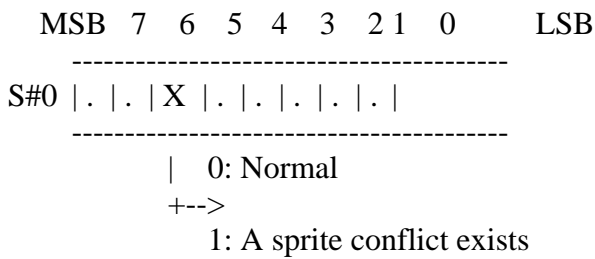


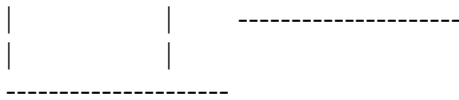
The sprite will display until here (X coordinate = 255)

### 5.2.3 Judging the sprite conflicts

When two sprites conflict, bit 5 of S#0 becomes "1" to inform of the conflict. A "conflict" means that bits "1" in the sprite pattern whose colour is not "transparent" occupy the same coordinate (see Figure 4.62)

Figure 4.62 Conflict of sprites (sprite mode 1)





These two sprites do not conflict      These two sprites conflict

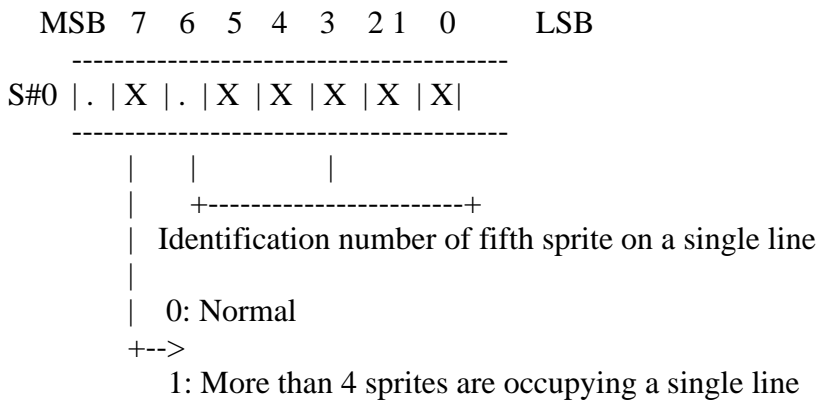
```

-----
|\| | \| | --> This pattern bit has one part
-----

```

When more than 5 sprites are placed on the same line, bit 6 of S#0 becomes "1" and the identifying number of the 5th sprite (the portion which cannot be displayed) is set in the 5 low order bits of S#0.

Figure 4.63 Judging the conflict (sprite mode 1)



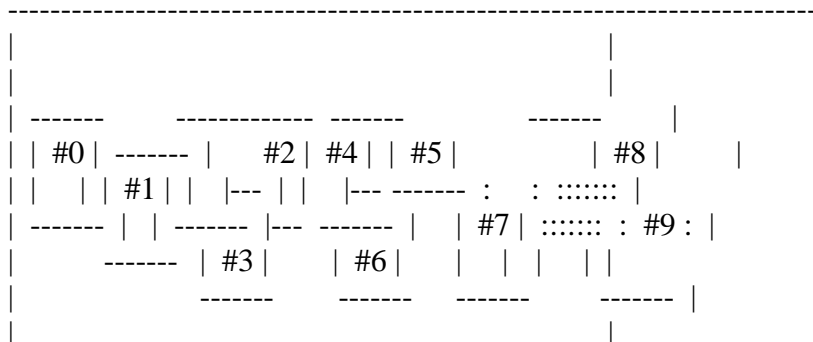
### 5.3 Sprite Mode 2

Sprite mode 2 is the newly added mode for MSX-VIDEO. It is not compatible with TMS9918 and cannot be used with MSX1 machines.

#### 5.3.1 Number of sprites to be displayed

The number of sprites which can be displayed on one screen is also 32, but up to eight sprites can be displayed on a given horizontal line of the screen. The priorities are the same as in mode 1 with the lower numbers having highest priority.

Figure 4.64 Number of sprites to be displayed (sprite mode 2)

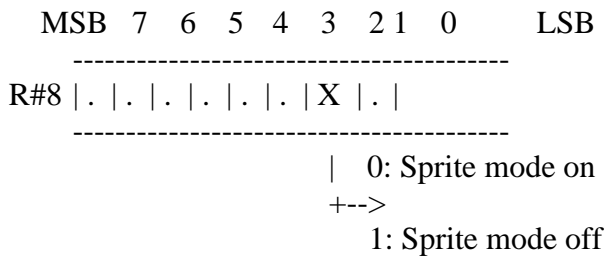


5.3.2 Sprite display settings

- \* Sprite size ..... same as sprite mode 1
- \* Expanding sprite ..... same as sprite mode 1
- \* Sprite display ON/OFF

In sprite mode 2, the sprite display can be turned ON/OFF by bit 1 of R#8. When this bit is set to 1, no sprites will appear on the screen.

Figure 4.65 Sprite display specification



- \* Setting the pattern generator table ..... same as sprite mode 1
- \* Sprite attribute table

In sprite mode 2, since different colours can be set for each horizontal line of the sprite, the colour information is stored in a sprite colour table as described below, which is independent of the sprite attribute table. Three kinds of information are stored in the sprite attribute table (see Figure 4.66).

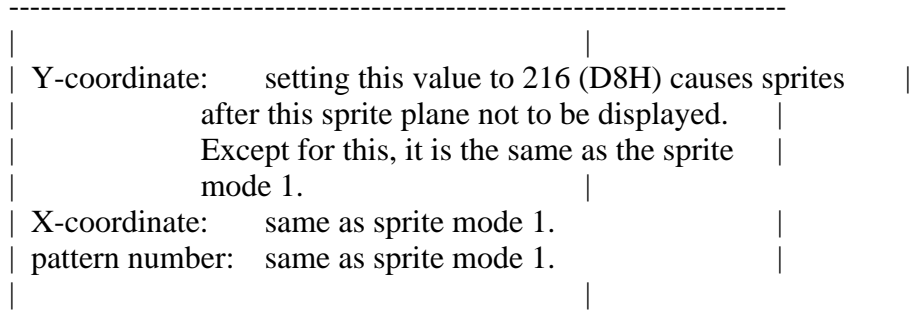


Figure 4.66 Structure of the sprite attribute table (sprite mode 2)

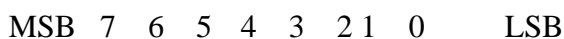






Figure 4.67 Structure of the sprite colour table (sprite mode 2)

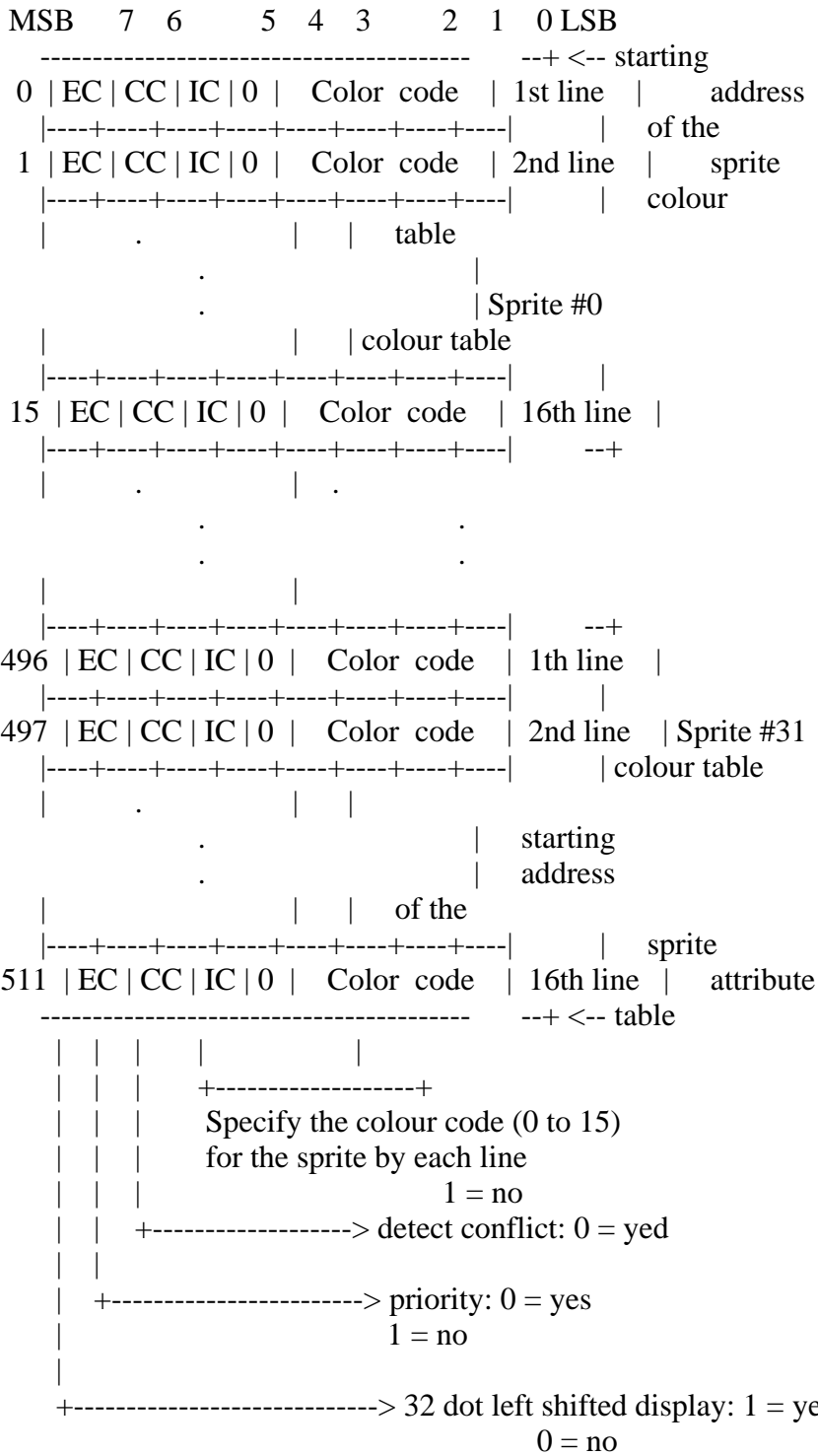
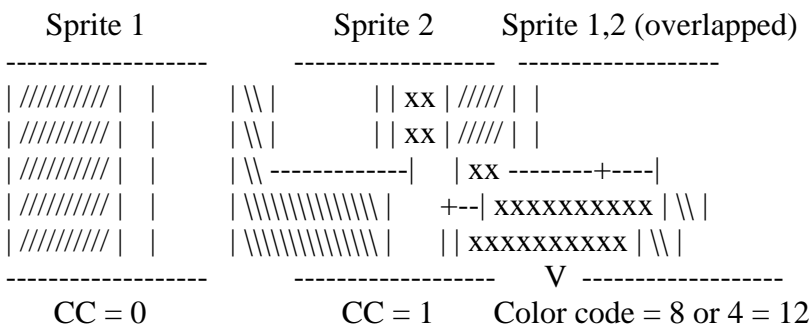


Figure 4.68 CC bit detection

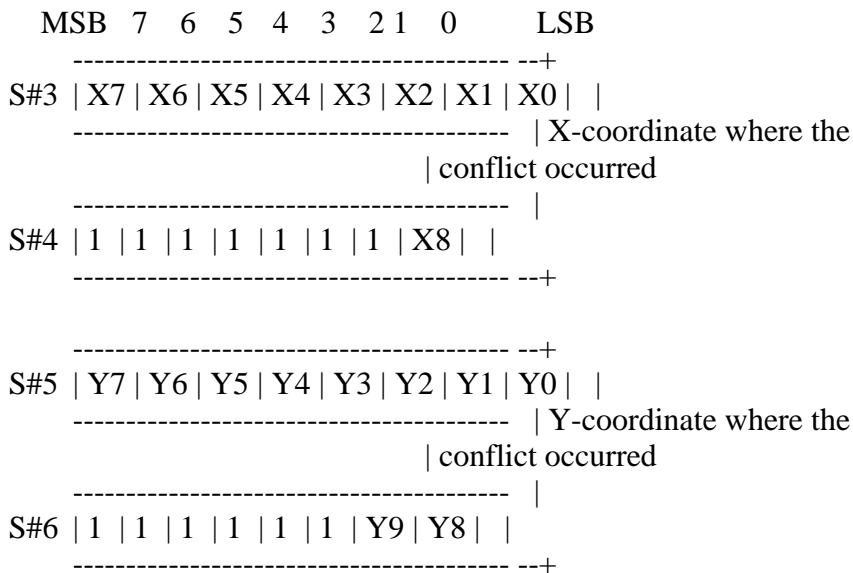




```
| |0|0| : : : :|
```

```
-----
| |
+-----+
CC and IC bits are both "0"
```

Figure 4.70 Readout of the conflict coordinate



Expression 4.5 Calculating the actual conflict coordinate

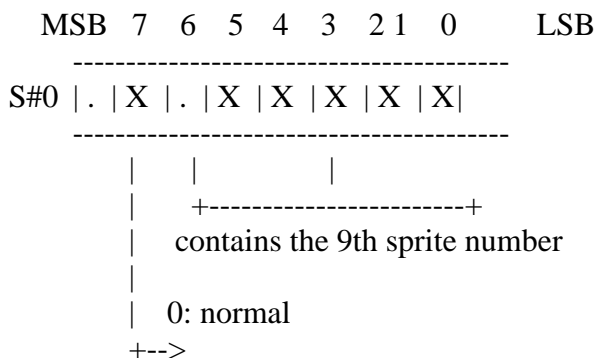
```

-----
|
| (X-coordinate where the conflict occurred) = |
| (X-coordinate of S#3 and S#4) - 12          |
|
| (Y-coordinate where the conflict occurred) = |
| (Y-coordinate of S#5 and S#6) - 8          |
|
|
|
-----

```

When more than nine sprites are placed on the same horizontal line, bit 6 of S#0 becomes "1" and the number of the sprite plane whose order of priority is 9 is entered to the 5 low order bits of S#0 (see Figure 4.71).

Figure 4.71 Conflict of the sprite (sprite mode 2)



1: more than 9 sprites are occupying the same line

## MSX2 TECHNICAL HANDBOOK

-----  
Edited by: ASCII Systems Division  
Published by: ASCII Corporation - JAPAN  
First edition: March 1987

Text file typed by: Nestor Soriano (Konami Man) - SPAIN  
March 1997

### Changes from the original:

- In Figure 4.72, last "10000H" is corrected to "1FFFFH".
- In Table 4.6, in TEOR line, "else DC+..." is corrected to "else DC=..."
- In Figure 4.76, in R#45 figure, DIX and DIY bits have been placed correctly (they were inverted in the original).
- In Figure 4.79, in R#42 and R#43 explanation, "NY -> of dots..." has been changed to "NY -> number of dots..."
- In List 4.9, in the line with the comment "YMMM command", 11010000 bitfield has been corrected to 11100000.
- In Figure 4.84, "\*" mark removed from the explanation of NX.
- In Figure 4.85, in R#45 explanation, "select source memory" text has been corrected to "select destination memory".
- In List 4.13, labels beginning with "LMMC" have been corrected to "LMCM".
- In List 4.15, in the line with the comment "NY", the "OUT (C),H" instruction has been corrected to "OUT (C),L".
- In section 6.5.9, the explanation of usage of the LINE command were mixed with other text. It has been corrected.
- In Figure 4.94, a line explaining the meaning of R#44 has been added.
- In Figure 4.97, BX9 bit has been suppressed in S#9 figure.
- In Figure 4.99, a line explaining the meaning of R#44 has been added.
- In Table 4.7, "CLR L" has been corrected to "CMR L".

-----  

## CHAPTER 4 - VDP AND DISPLAY SCREEN (Part 6)

### 6. VDP COMMAND USAGE

MSX-VIDEO can execute basic graphic operations, which are called VDP commands. These are done by accessing special hardware and are available in the GRAPHIC 4 to GRAPHIC 7 modes. These graphic commands have been made easy to implement, requiring only that the necessary parameters be set in the proper registers before invoking them. This section describes these VDP commands.

## 6.1 Coordinate System of VDP Commands

When VDP commands are executed, the location of the source and destination points are represented as (X, Y) coordinates as shown in Figure 4.72. When commands are executed, there is no page division and the entire 128K bytes VRAM is placed in a large coordinate system.

Figure 4.72 Coordinate system of VRAM

GRAPHIC 4 (SCREEN 5)		GRAPHIC 5 (SCREEN 6)	
----- 00000H -----		----- 00000H -----	
(0,0) (255,0)	(0,0) (511,0)		
Page 0	Page 0		
(0,255) (255,255)	(0,255) (511,255)		
-----	08000H  -----		
(0,256) (255,256)	(0,256) (511,256)		
Page 1	Page 1		
(0,511) (255,511)	(0,511) (511,511)		
-----	10000H  -----		
(0,512) (255,512)	(0,512) (511,512)		
Page 2	Page 2		
(0,767) (255,767)	(0,767) (511,767)		
-----	18000H  -----		
(0,768) (255,768)	(0,768) (511,768)		
Page 3	Page 3		
(0,1023) (255,1023)	(0,1023) (511,1023)		
-----	1FFFFH -----		
----- 00000H -----		----- 00000H -----	
GRAPHIC 7 (SCREEN 8)		GRAPHIC 6 (SCREEN 7)	
(0,0) (255,0)	(0,0) (511,0)		
Page 0	Page 0		
(0,255) (255,255)	(0,255) (511,255)		
-----	10000H  -----		
(0,256) (255,256)	(0,256) (511,256)		
Page 1	Page 1		
(0,511) (255,511)	(0,511) (511,511)		
-----	1FFFFH -----		

## 6.2 VDP Commands

There are 12 types of VDP commands which can be executed by MSX-VIDEO. These are shown in Table 4.5.

Table 4.5 List of VDP commands

Command name	Destination	Source	Units	Mnemonic	R#46 (4 hi ord)
	VRAM	CPU	bytes	HMMC	1 1 1 1
High speed move	VRAM	VRAM	bytes	YMMM	1 1 1 0
	VRAM	VRAM	bytes	HMMM	1 1 0 1
	VRAM	VDP	bytes	HMMV	1 1 0 0
	VRAM	CPU	dots	LMMC	1 0 1 1
Logical move	CPU	VRAM	dots	LMCM	1 0 1 0
	VRAM	VRAM	dots	LMMM	1 0 0 1
	VRAM	VDP	dots	LMMV	1 0 0 0
Line	VRAM	VDP	dots	LINE	0 1 1 1
Search	VRAM	VDP	dots	SRCH	0 1 1 0
Pset	VRAM	VDP	dots	PSET	0 1 0 1
Point	VDP	VRAM	dots	POINT	0 1 0 0
Reserved	----	----	----	----	0 0 1 1
	----	----	----	----	0 0 1 0
	----	----	----	----	0 0 0 1
Stop	----	----	----	----	0 0 0 0

\* When data is written in R#46 (Command register), MSX-VIDEO begins to execute the command after setting 1 to bit 0 (CE/Command Execute) of the status register S#2. Necessary parameters should be set in register R#32 to R#45 before the command is executed.

\* When the execution of the command ends, CE becomes 0.

\* To stop the execution of the command, execute STOP command.

\* Actions of the commands are guaranteed only in the bitmap modes (GRAPHIC 4 to GRAPHIC 7).

### 6.3 Logical Operations

When commands are executed, various logical operations can be done between data in VRAM and the specified data. Each operation will be done according to the rules listed in Table 4.6.

In the table, SC represents the source color and DC represents the destination colour. IMP, AND, OR, EOR and NOT write the result of each operation to the destination. In operations whose names are preceded by "T", dots which correspond with SC=0 are not the objects of the operations and remains as DC. Using these operations enables only colour portions of two figures to be overlapped, so they are especially effective for animations.

List 4.7 shows an example of these operations.

Table 4.6 List of logical operations

Logical name		L03	L02	L01	L00
IMP	DC=SC			0	0 0 0 0
AND	DC=SCxDC			0	0 0 0 1
OR	DC=SC+DC			0	0 0 1 0
EOR	$\overline{DC=SCxDC+SCxDC}$			0	0 0 1 1
NOT	$\overline{DC=SC}$			0	1 0 0 0
----		0	1	0	1
----		0	1	1	0
----		0	1	1	1
TIMP	if SC=0 then DC=DC else DC=SC			1	0 0 0 0
TAND	if SC=0 then DC=DC else DC=SCxDC			1	0 0 0 1
TOR	if SC=0 then DC=DC else DC=SC+DC			1	0 0 1 0
TEOR	if SC=0 then $\overline{DC=DC}$ else $\overline{DC=SCxDC+SCxDC}$			1	0 0 1 1
TNOT	if SC=0 then $\overline{DC=DC}$ else DC=SC			1	1 0 0 0
----		1	1	0	1
----		1	1	1	0
----		1	1	1	1

- \* SC = Source colour code
- \* DC = Destination colour code
- \* EOR = Exclusive OR

List 4.7 Example of the logical operation with T

```

=====
1000 '*****
1010 ' List 4.7 logical operation with T
1020 '*****
1030 '
1040 SCREEN8 : COLOR 15,0,0 : CLS
1050 DIM A%(3587)
1060 '

```



```

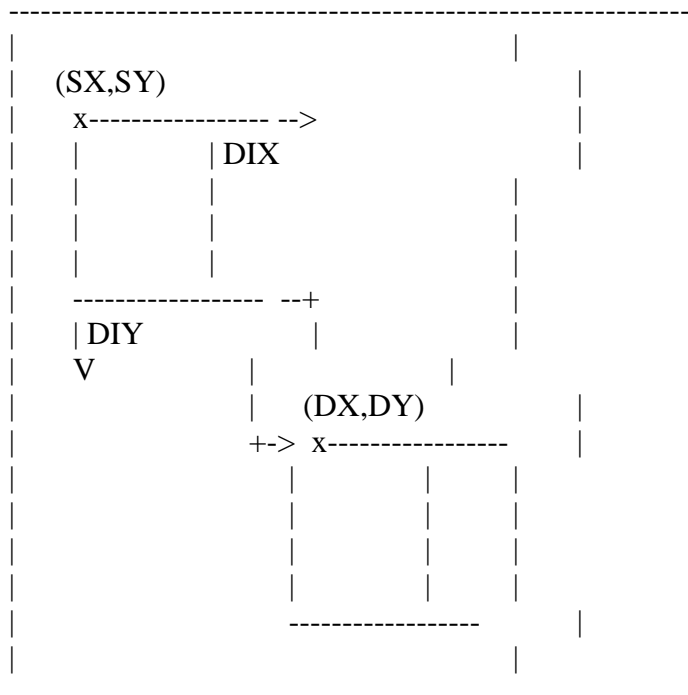
1070 LINE (50,50)-(60,100),48,8 : PAINT (51,51),156,48
1080 CIRCLE (55,30),30,255 : PAINT (55,30),240,255
1090 COPY(20,0)-(90,100) TO A%
1100 CLS
1110 '
1120 R=RND(-TIME)
1130 FOR Y=0 TO 100 STEP 3
1140 X=INT(RND(1)*186)
1150 COPY A% TO (X,Y),,TPSET
1160 NEXT
1170 '
1180 GOTO 1180

```

## 6.4 Area Specification

AREA-MOVE commands are for transferring screen data inside areas surrounded by a rectangle. The area to be transferred is specified by one vertex and the length of each side of the rectangle as shown in Figure 4.73. SX and SY represent the basic point of the rectangle to be transferred and NX and NY represent the length of each side in dots. The two bits, DIX and DIY, are for the direction of transferring data (the meaning of DIX and DIY depends on the type of command). The point where the area is to be transferred is specified in DX and DY.

Figure 4.73 Area specification



## 6.5 Use of Each Command

Commands are classified into three types, high-speed transfer commands, logical transfer commands, and drawing commands. This section describes the commands and their use.

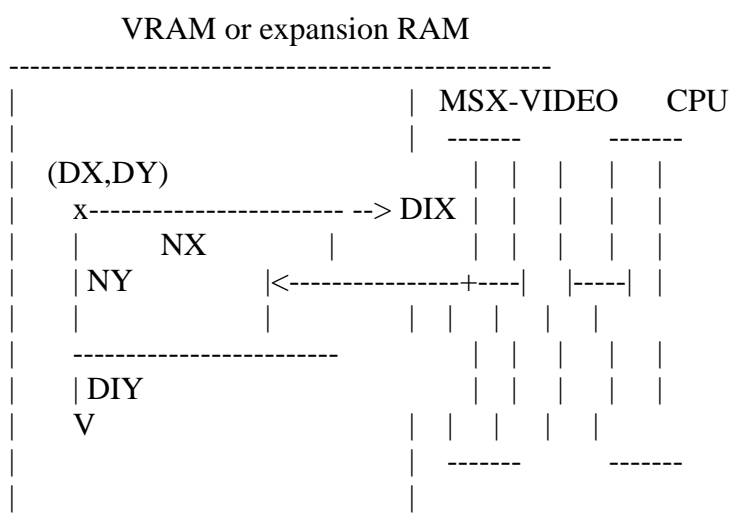
### 6.5.1 HMMC (CPU -> VRAM high-speed transfer)

Data is transferred into the specified area of VRAM from the CPU (see Figure 4.74). Logical operations cannot be specified. Data is transferred in bytes in high-speed transfer commands such as HMMC. Note that the low order bit of the X-coordinate is not referred to in GRAPHIC 4, or 6 modes. The two low order bits are not referred to in GRAPHIC 5 mode (see Figure 4.75).

Set the parameters as shown in Figure 4.76 to the appropriate registers. At this point, write only the first byte of data to be transferred from the CPU in R#44. Writing the command code F0H in R#46 causes the command to be executed, and UMSX-VIDEO receives data from R#44 and writes it to VRAM, then waits for data from the CPU.

The CPU writes data after the second byte in R#44. Note that data should be transferred after MSX-VIDEO can receive data (in the case that TR bit is "1"), referring to TR bit of S#2. When the CE bit of S#2 is "0", this means that all data has been transferred (see figure 4.77). List 4.8 shows an example of using HMMC.

Figure 4.74 Action of HMMC command



MXD: select the destination memory 0 = VRAM, 1 = expansion RAM

NX: number of dots to be transferred in X direction (0 to 511)\*

NY: number of dots to be transferred in Y direction (0 to 1023)

DIX: direction of NX from the origin 0 = right, 1 = left

DIY: direction of NY from the origin 0 = below, 1 = above

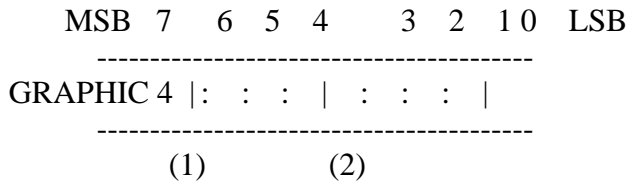
DX: destination origin X-coordinate (0 to 511)\*

DY: destination origin Y-coordinate (0 to 1023)

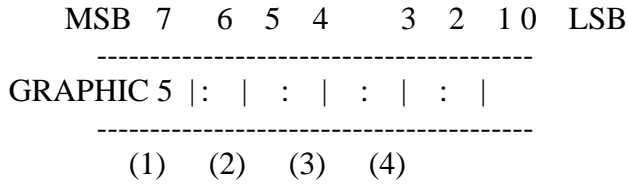
CLR (R#44:Colour register): 1st byte of data to be transferred

\* The one low-order bit for GRAPHIC 4 and 6 modes, or two low-order bits for GRAPHIC 5 mode of the DX and NX registers are ignored.

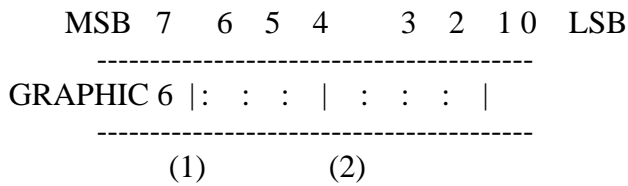
Figure 4.75 Dots not to be referred to



Since 1 VRAM byte represents 2 dots, 1 low order bit of X-coordinate is not referred to.



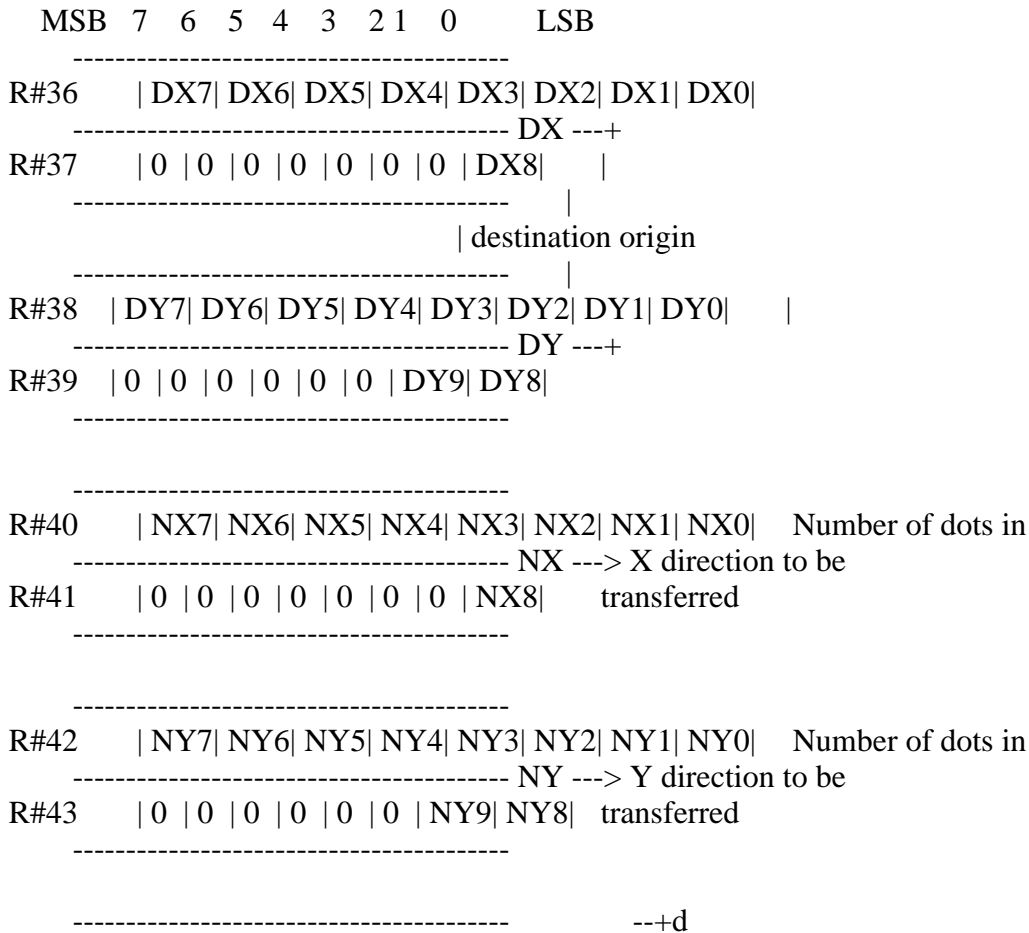
Since 1 VRAM byte represents 4 dots, 2 low order bits of X-coordinate are not referred to.

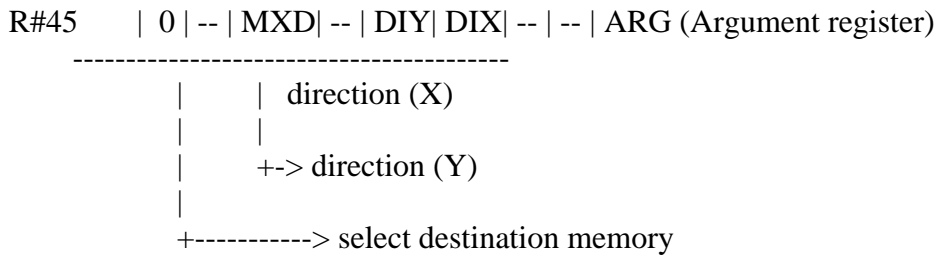
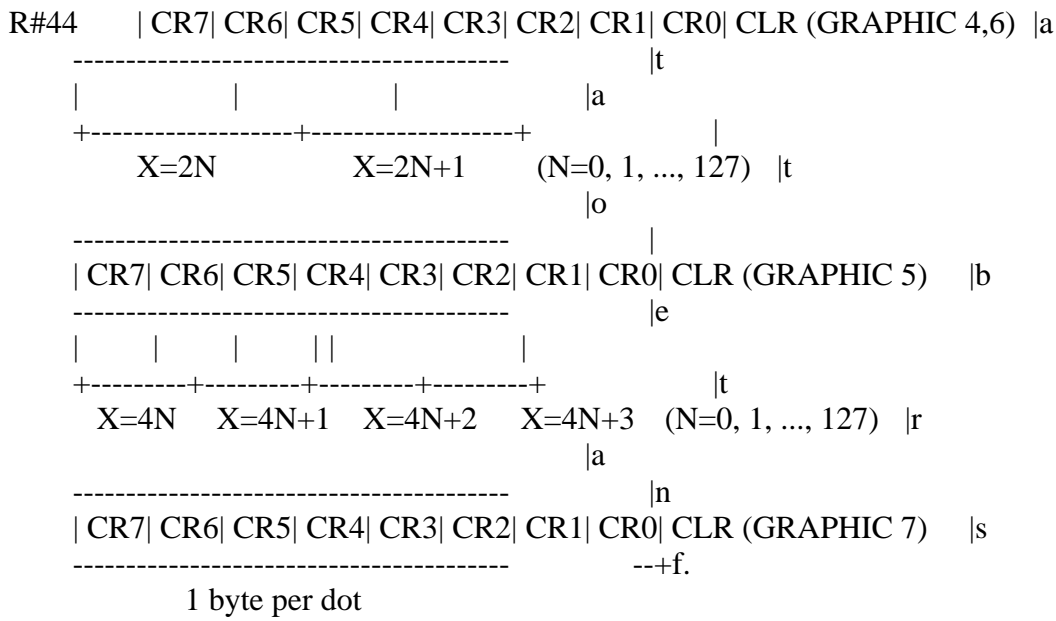


Since 1 VRAM byte represents 2 dots, 1 low order bit of X-coordinate is not referred to.

Figure 4.76 Register settings of HMMC command

> HMMC register setup





> HMMC command execution

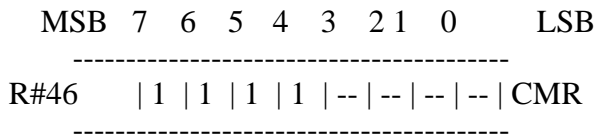
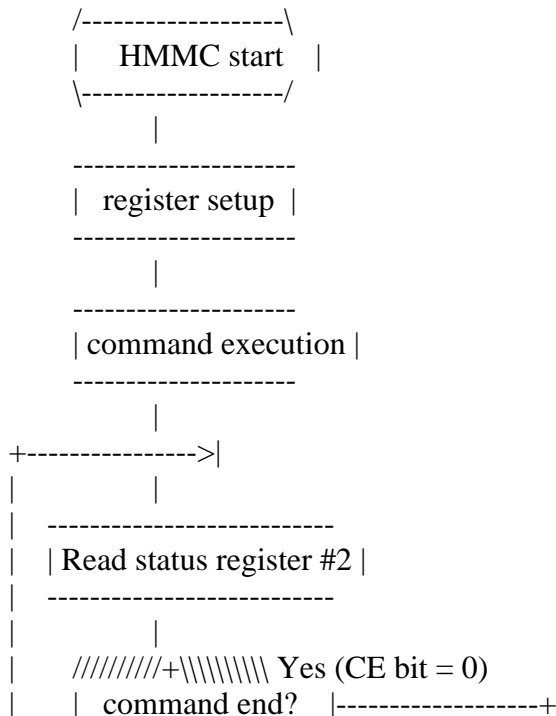
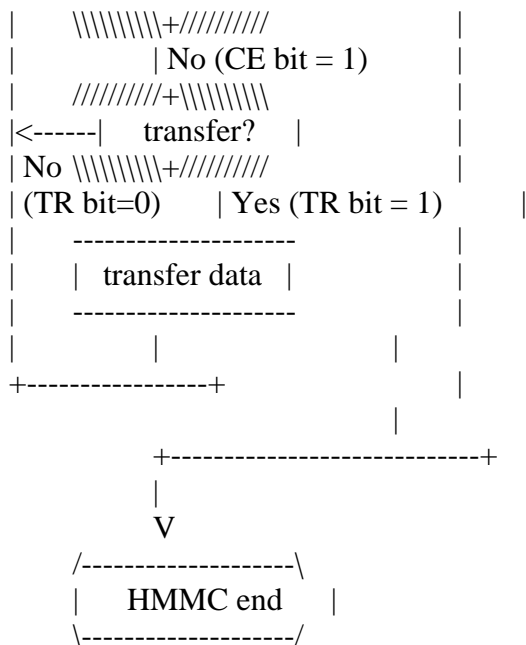


Figure 4.77 HMMC command execution flow chart





List 4.8 Example of HMMC command execution

```

;*****
; List 4.8 HMMC sample
; to use, set H, L, D, E, IX and go
; RAM (IX) ---> VRAM (H,L)-(D,E)
;*****
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

HMMC: DI ;disable interrupt
CALL WAIT.VDP ;wait end of command

LD A,(WRVDP)
LD C,A
INC C ;C := PORT#1's address
LD A,36
OUT (C),A
LD A,17+80H
OUT (C),A ;R#17 := 36

INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),H ;DX
OUT (C),A
OUT (C),L ;DY
OUT (C),A

LD A,H ;make NX and DIX
SUBA
LD D,00000100B
JR NC,HMMC1

```

```

LD D,00000000B
NEG
HMMC1:LD H,A          ;H := NX , D := DIX

LD A,L
SUBA
LD E,00001000B
JR NC,HMMC2
LD E,00000000B
NEG
HMMC2:LD L,A          ;L := NY , E := DIY

XOR A
OUT (C),H             ;NX
OUT (C),A
OUT (C),L             ;NY
OUT (C),A
LD H,(IX+0)
OUT (C),H             ;first DATA
LD A,D
OR E
OUT (C),A             ;DIX and DIY
LD A,0F0H
OUT (C),A             ;HMMC command

LD A,(WRVDP)
LD C,A                ;C := PORT#1's address
INC C
LD A,44+80H
OUT (C),A
LD A,17+80H
OUT (C),A
INC C
INC C

LOOP: LD A,2
CALL GET.STATUS
BIT 0,A               ;check CE bit
JR Z,EXIT
BIT 7,A               ;check TR bit
JR Z,LOOP
INC IX
LD A,(IX+0)
OUT (C),A
JR LOOP

EXIT: LD A,0
CALL GET.STATUS       ;when exit, you must select S#0
EI
RET

GET.STATUS:           ;read status register specified by A
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH

```

```

OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

```

WAIT.VDP:                ;wait VDP ready
LD A,2
CALL GET.STATUS
AND 1
JR NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

END

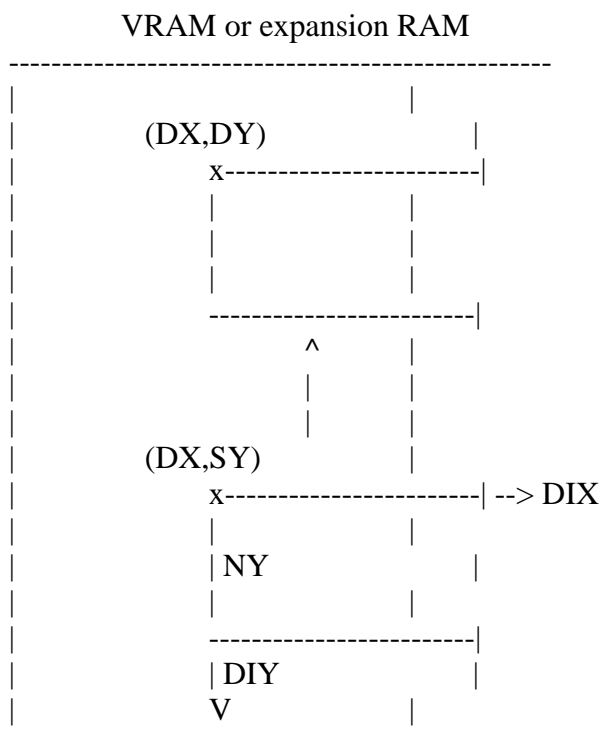
```

### 6.5.2 YMMM (high-speed transfer between VRAM in Y direction)

Data from a specified VRAM area is transferred into another area in VRAM. Note that transfers using this command can only be done in the Y direction (see Figure 4.78).

After setting the data as shown in Figure 4.79 in the proper registers, writing command code E0H in R#46 causes the command to be executed. When the CE bit of S#2 is "1", it indicates that the command is currently being executed. List 4.9 shows an example of using YMMM.

Figure 4.78 Actions of YMMM command



MXD: select the destination memory 0 = VRAM, 1 = expansion RAM

SY: source origin Y-coordinate (0 to 1023)

NY: number of dots to be transferred in Y direction (0 to 1023)

DIX: set which to be transferred, to the right end or to the left end of the screen from the source origin 0 = right, 1 = left

DIY: direction of NY from the origin 0 = below, 1 = above

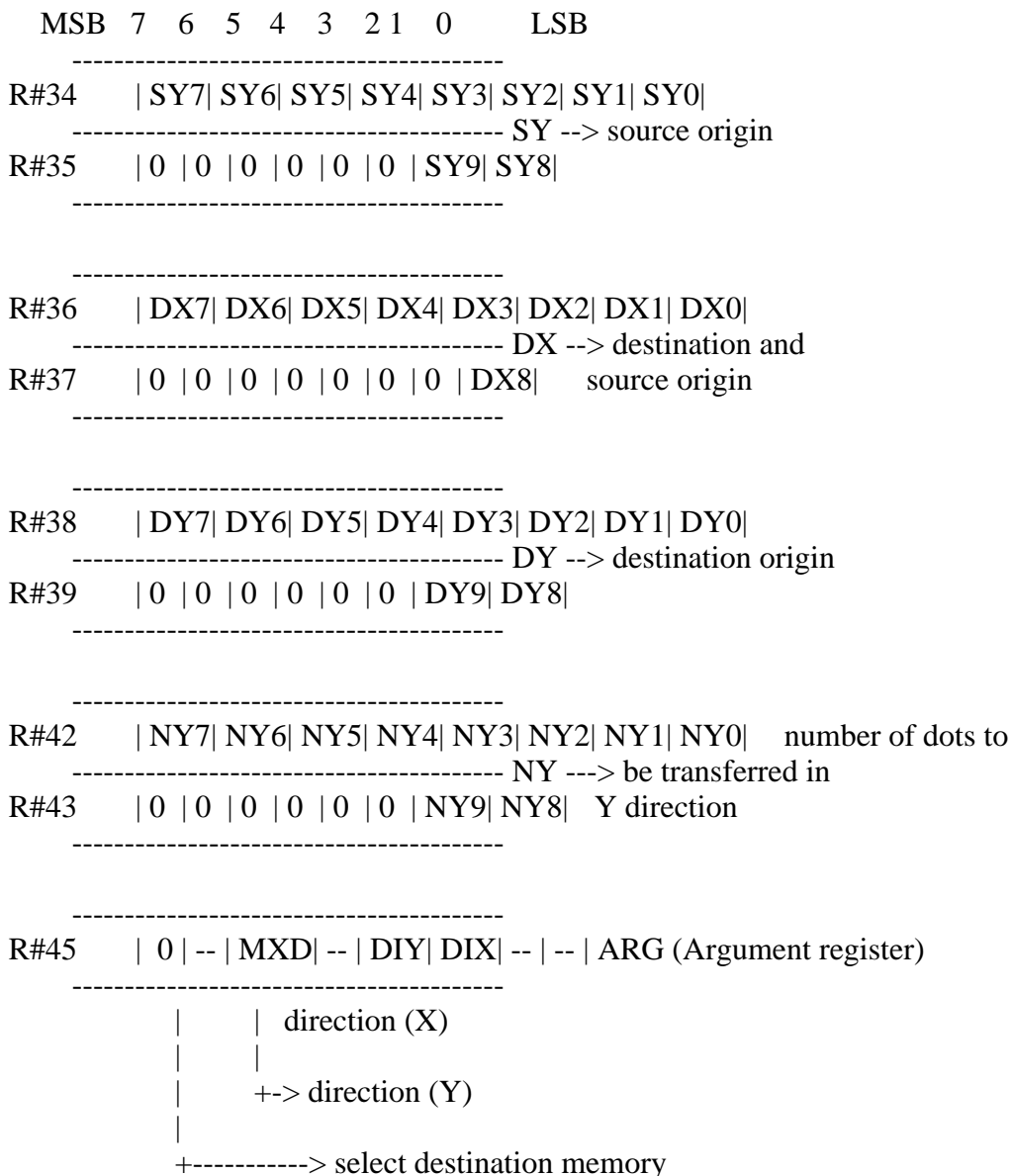
DX: destination origin X-coordinate (0 to 511)\*

DY: destination origin Y-coordinate (0 to 1023)

\* The one low-order bit for GRAPHIC 4 and 6 modes, or two low-order bits for GRAPHIC 5 mode of the DX register are ignored.

Figure 4.79 Register settings of YMMM command

> YMMM register setup



> YMMM command execution



MSB 7 6 5 4 3 2 1 0 LSB

R#46 | 1 | 1 | 1 | 0 | -- | -- | -- | -- | CMR

List 4.9 Example of YMMM command execution

```
*****
; List 4.9 YMMM sample
; to use, set L, E, B, C, D(bit 2) and go
; VRAM (B,L)-(*,E) ---> VRAM (B,C)
; DIX must be set in D(bit 2)
*****
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

YMMM: DI ;disable interrupt
PUSH BC ;save destination
CALL WAIT.VDP ;wait end of command

LD A,(WRVDP)
LD C,A
INC C ;C := PORT#1's address
LD A,34
OUT (C),A
LD A,17+80H
OUT (C),A ;R#17 := 34

INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),L ;SY
OUT (C),A

LD A,L ;make NY and DIY
SUBA
LD E,00001000B
JP NC,YMMM1
LD E,00000000B
NEG
YMMM1: LD L,A ;L := NY , D := DIY

LD A,D
OR E

POP DE ;restore DX,DY
PUSH AF ;save DIX,DIY
XOR A
OUT (C),D ;DX
OUT (C),A
OUT (C),E ;DY
OUT (C),A
```

```

OUT    (C),A           ;dummy
OUT    (C),A           ;dummy
OUT    (C),L           ;NY
OUT    (C),A
OUT    (C),A           ;dummy
POP AF
OUT    (C),A           ;DIX and DIY
LD  A,11100000B       ;YMMM command
OUT    (C),A

```

```

EI
RET

```

GET.STATUS:

```

PUSH  BC
LD  BC,(WRVDP)
INC  C
OUT  (C),A
LD  A,8FH
OUT  (C),A
LD  BC,(RDVDP)
INC  C
IN  A,(C)
POP  BC
RET

```

WAIT.VDP:

```

LD  A,2
CALL GET.STATUS
AND  1
JP  NZ,WAIT.VDP
XOR  A
CALL GET.STATUS
RET

```

END

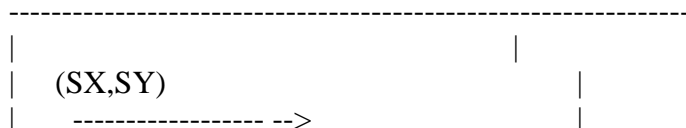
### 6.5.3 HMMM (high-speed transfer between VRAM)

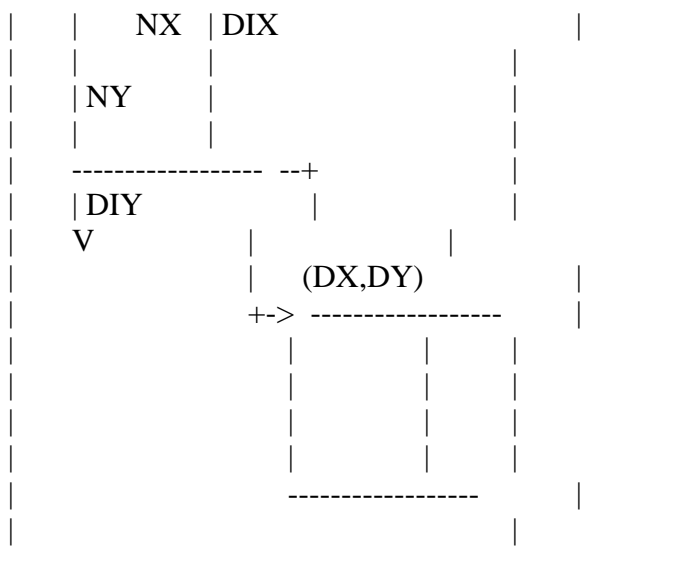
Data of specified VRAM area is transferred into another area in VRAM (see Figure 4.80).

After setting the parameters as shown in Figure 4.81, writing D0H in R#46 causes the command to be executed. While the command is being executed, CE bit of S#2 is "1". List 4.10 shows an example of using HMMM.

Figure 4.80 Actions of HMMM command

VRAM or expansion RAM





MXS: select the source memory      0 = VRAM, 1 = expansion RAM  
 MXD: select the destination memory   0 = VRAM, 1 = expansion RAM

SX: source origin X-coordinate (0 to 511)\*  
 SY: source origin Y-coordinate (0 to 1023)

NX: number of dots to be transferred in X direction (0 to 511)\*  
 NY: number of dots to be transferred in Y direction (0 to 1023)

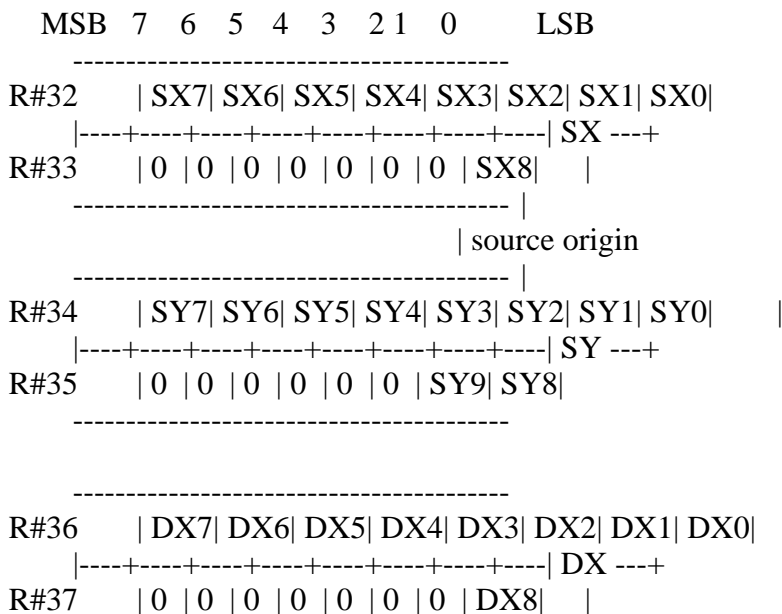
DIX: direction of NX from the origin   0 = right, 1 = left  
 DIY: direction of NY from the origin   0 = below, 1 = above

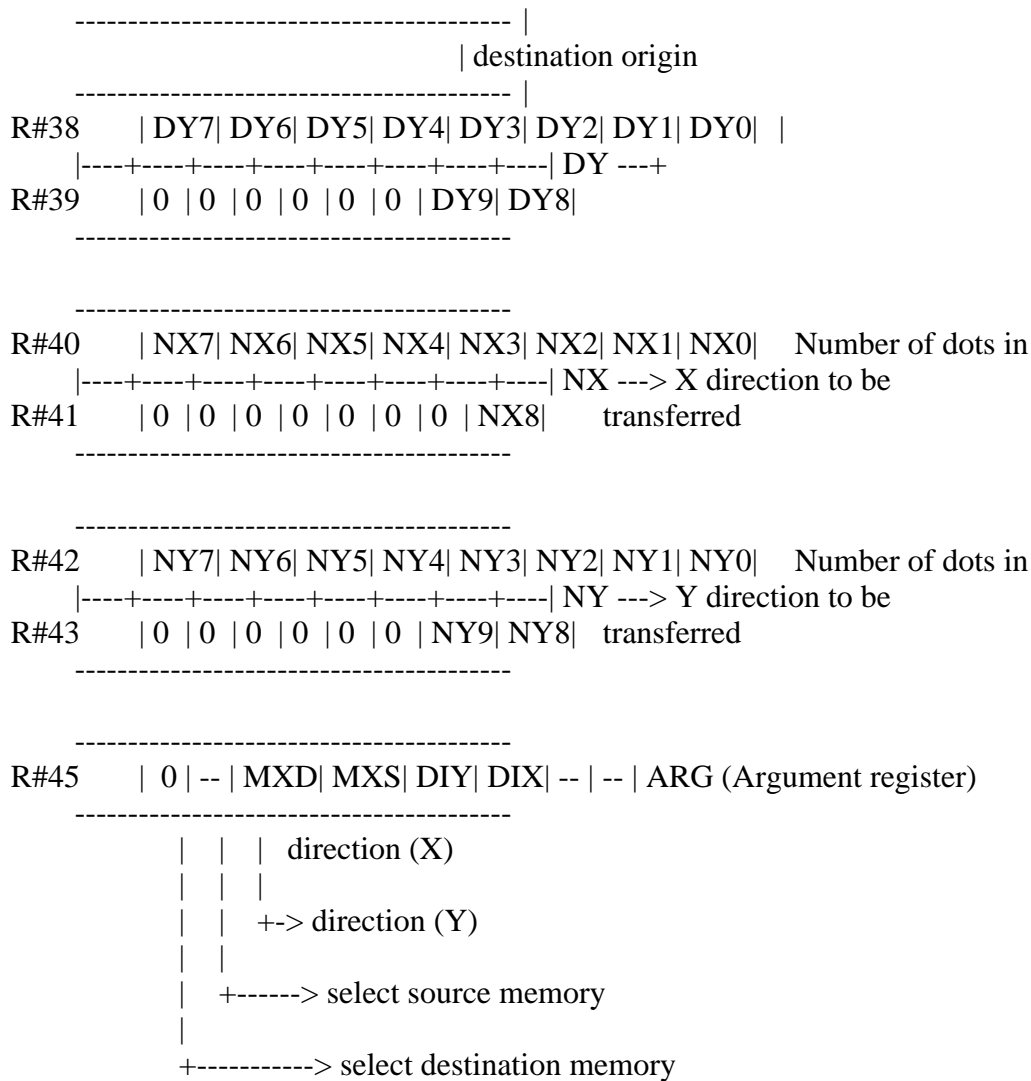
DX: destination origin X-coordinate (0 to 511)\*  
 DY: destination origin Y-coordinate (0 to 1023)

\* The one low-order bit for GRAPHIC 4 and 6 modes, or two low-order bits for GRAPHIC 5 mode of the SX, DX, and NX register are ignored.

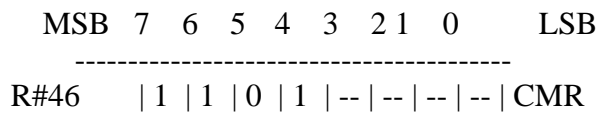
Figure 4.81 Register settings of HMMM command

> HMMM register setup





> HMMM command execution



List 4.10 Example of HMMM command execution

```

=====
;*****
;
; List 4.10 HMMM sample
;
; to use, set H, L, D, E, B, C and go
; VRAM (H,L)-(D,E) ---> VRAM (B,C)
; DIX must be set in D(bit 2)
;*****
;
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

HMMM: DI ;disable interrupt
      PUSH BC ;save destination
      CALL WAIT.VDP ;wait end of command

```

```

LD A,(WRVDP)
LD C,A
INC C          ;C := PORT#1's address
LD A,32
OUT (C),A
LD A,80H+17
OUT (C),A          ;R#17 := 32

INC C
INC C          ;C := PORT#3's address
XOR A
OUT (C),H      ;SX
OUT (C),A
OUT (C),L      ;SY
OUT (C),A

LD A,H          ;make NX and DIX
SUBA
LD D,00000100B
JP NC,HMMM1
LD D,00000000B
NEG
HMMM1: LD H,A          ;H := NX , D := DIX

LD A,L          ;make NY and DIY
SUBA
LD E,00001000B
JP NC,HMMM2
LD E,00000000B
NEG
HMMM2: LD L,A          ;L := NY , E := DIY

LD A,D
OR E
POP DE          ;restore DX,DY
PUSH AF          ;save DIX,DIY
XOR A
OUT (C),D      ;DX
OUT (C),A
OUT (C),E      ;DY
OUT (C),A
OUT (C),H      ;NX
OUT (C),A
OUT (C),L      ;NY
OUT (C),A
OUT (C),A      ;dummy
POP AF
OUT (C),A      ;DIX and DIY

LD A,11010000B ;HMMM command
OUT (C),A

EI
RET

```

```

GET.STATUS:
PUSH BC

```

```

LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

```

WAIT.VDP:
LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

END

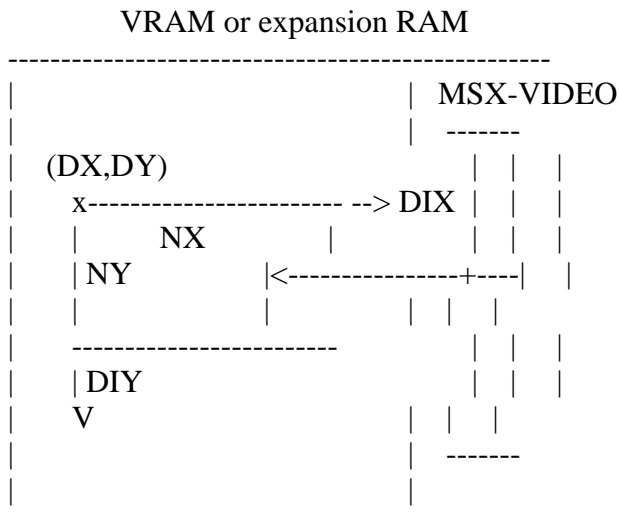
```

#### 6.5.4 HMMV (painting the rectangle in high speed)

Each byte of data in the specified VRAM area is painted by the specified colour code (see Figure 4.82)

After setting the parameters as shown in Figure 4.83, writing COH in R#46 causes the command to be executed. While the command is being executed, the CE bit of S#2 is 1. List 4.11 shows an example of using HMMV.

Figure 4.82 Actions of HMMC command



MXD: select memory                    0 = VRAM, 1 = expansion RAM

NX: number of dots to be painted in X direction (0 to 511)\*

NY: number of dots to be painted in Y direction (0 to 1023)

DIX: direction of NX from the origin 0 = right, 1 = left

DIY: direction of NY from the origin 0 = below, 1 = above

DX: origin X-coordinate (0 to 511)\*

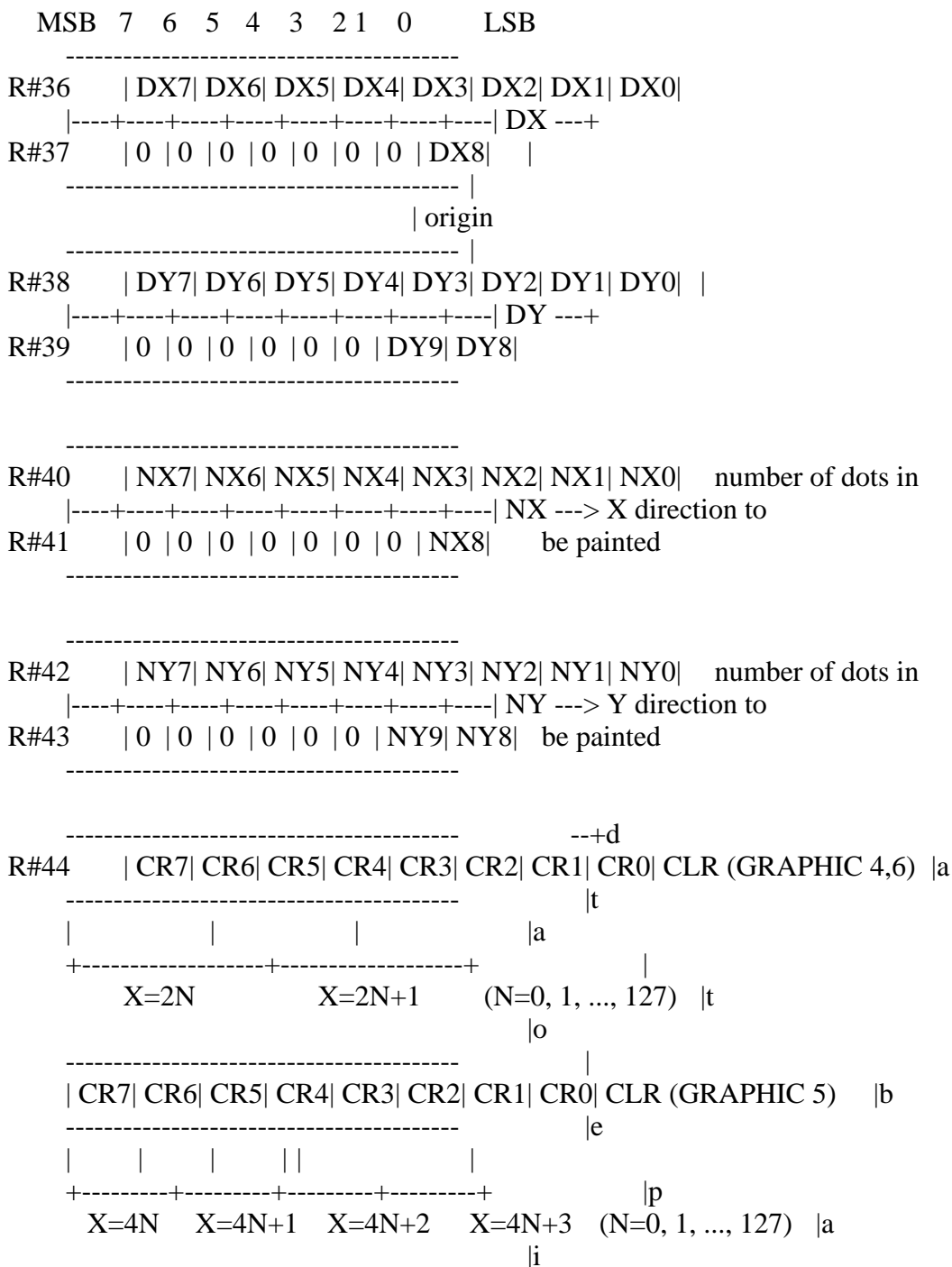
DY: origin Y-coordinate (0 to 1023)

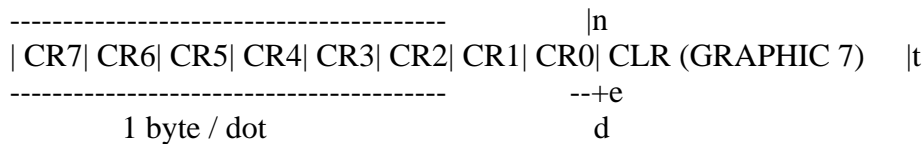
CLR (R#44:Colour register): Painted data

\* The one low-order bit for GRAPHIC 4 and 6 modes, or two low-order bits for GRAPHIC 5 mode of the DX and NX registers are ignored.

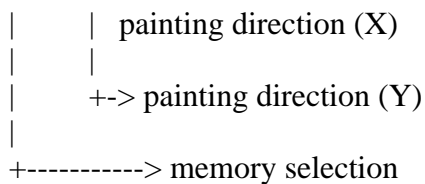
Figure 4.83 Register settings of HMMV command

> HMMV register setup





R#45 | 0 | -- | MXD | -- | DIY | DIX | -- | -- | ARG (Argument register)



> HMMV command execution

MSB 7 6 5 4 3 2 1 0 LSB

R#46 | 1 | 1 | 0 | 0 | -- | -- | -- | -- | CMR

#### List 4.11 Example of HMMV command execution

```

;*****
; List 4.11 HMMV sample
; to use, set H, L, D, E, B and go
; B ---> VRAM (H,L)-(D,E) fill
;*****
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

HMMV: DI ;disable interrupt
CALL WAIT.VDP ;wait end of command

LD A,(WRVDP)
LD C,A
INC C ;C := PORT#1's address
LD A,36
OUT (C),A
LD A,80H+17
OUT (C),A ;R#17 := 36

INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),H ;DX
OUT (C),A
OUT (C),L ;DY
OUT (C),A

LD A,H ;make NX and DIX
SUBA

```



```

LD D,00000100B
JP NC,HMMV1
LD D,00000000B
NEG
HMMV1: LD H,A ;H := NX

LD A,L ;make NY and DIY
SUBA
LD E,00001000B
JP NC,HMMV2
LD E,00000000B
NEG
HMMV2: OUT (C),H
LD H,A ;H := NY

XOR A
OUT (C),A
OUT (C),H
OUT (C),A
OUT (C),B ;fill data
XOR A
OR D
OR E
OUT (C),A ;DIX and DIY

LD A,11000000B ;HMMV command
OUT (C),A

EI
RET

```

```

GET.STATUS:
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

```

WAIT.VDP:
LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

```

```

END

```

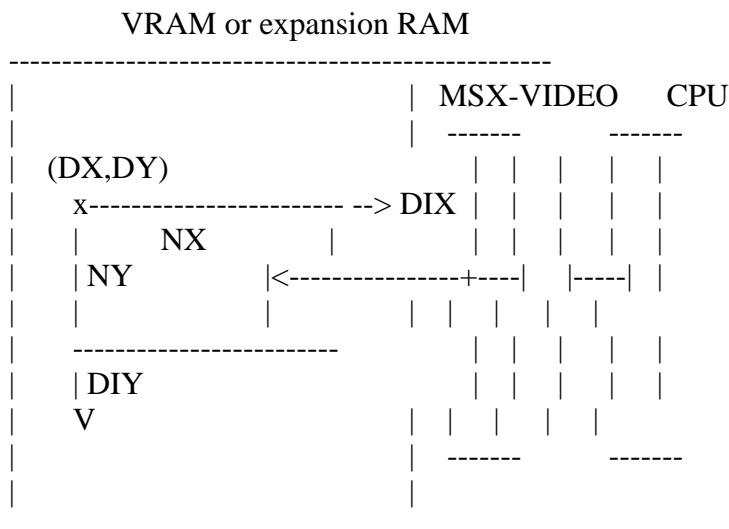
---

### 6.5.5 LMMC (CPU -> VRAM logical transfer)

Data is transferred from the CPU to the specified VRAM area in dots (see Figure 4.84). Logical operations with the source can be specified. In the logical transfer commands, such as LMMC, data is transferred in dots and one byte is required for the information of one pixel in all screen modes.

After setting the data as shown in Figure 4.85, write command code B0H in R#46. At this point, logical operations can be specified by using the 4 low order bits of the command register. Data is transferred with reference to the TR and CE bit of S#2, as in HMMC (see Figure 4.86). List 4.12 shows an example of using LMMC.

Figure 4.84 Action of LMMC command



MXD: select destination memory      0 = VRAM, 1 = expansion RAM

NX: number of dots to be transferred in X direction (0 to 511)

NY: number of dots to be transferred in Y direction (0 to 1023)

DIX: direction of NX from the origin    0 = right, 1 = left

DIY: direction of NY from the origin    0 = below, 1 = above

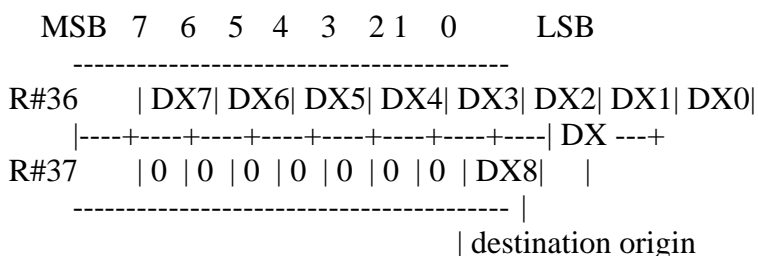
DX: destination origin X-coordinate (0 to 511)

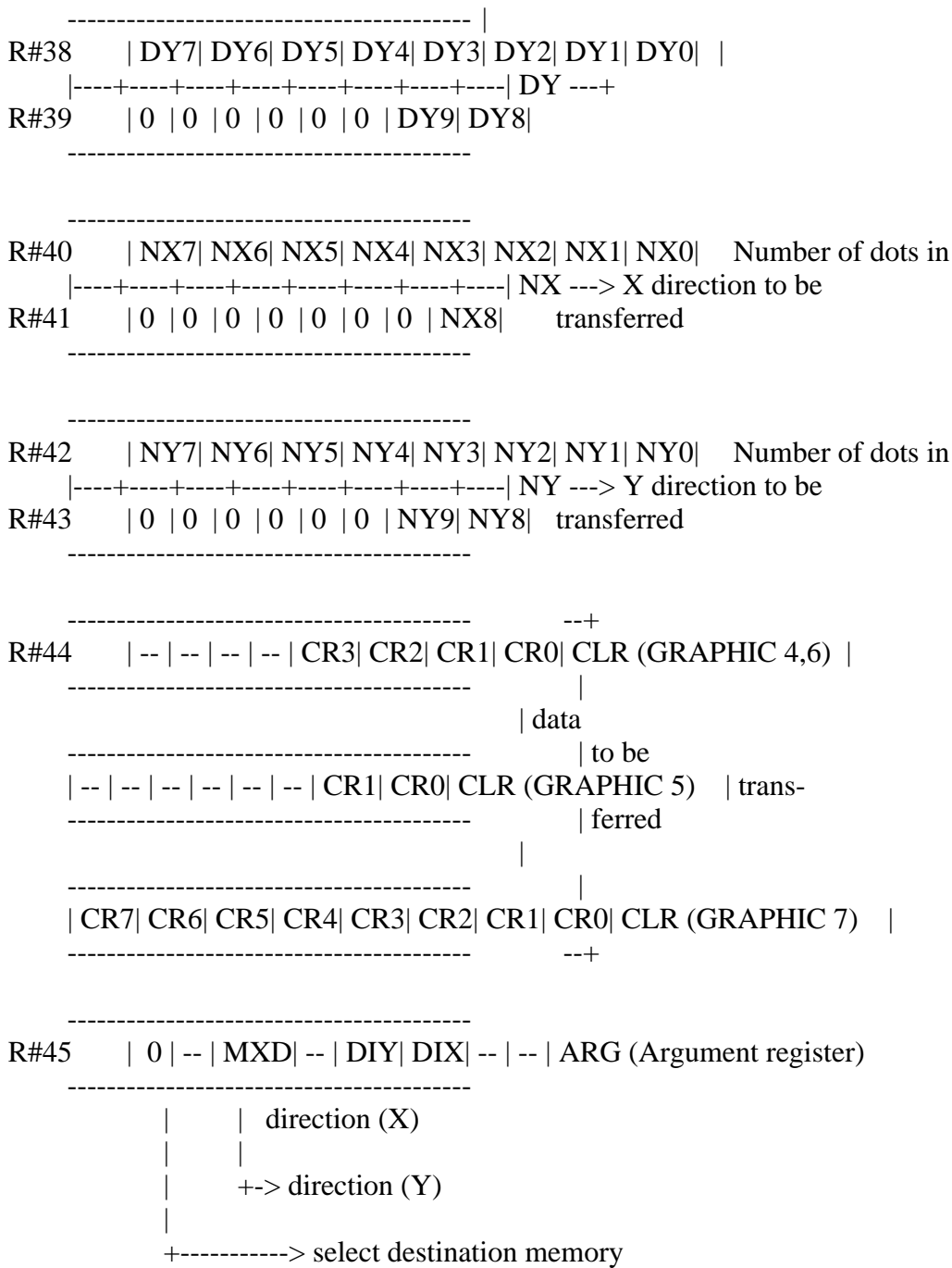
DY: destination origin Y-coordinate (0 to 1023)

CLR (R#44:Colour register): 1st byte of data to be transferred

Figure 4.85 Register settings of LMMC command

> LMMC register setup





> LMMC command execution

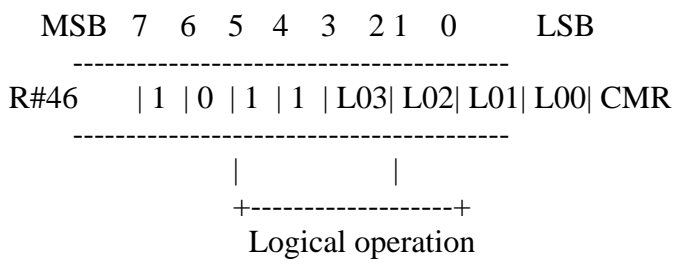
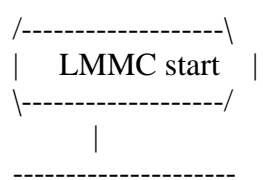
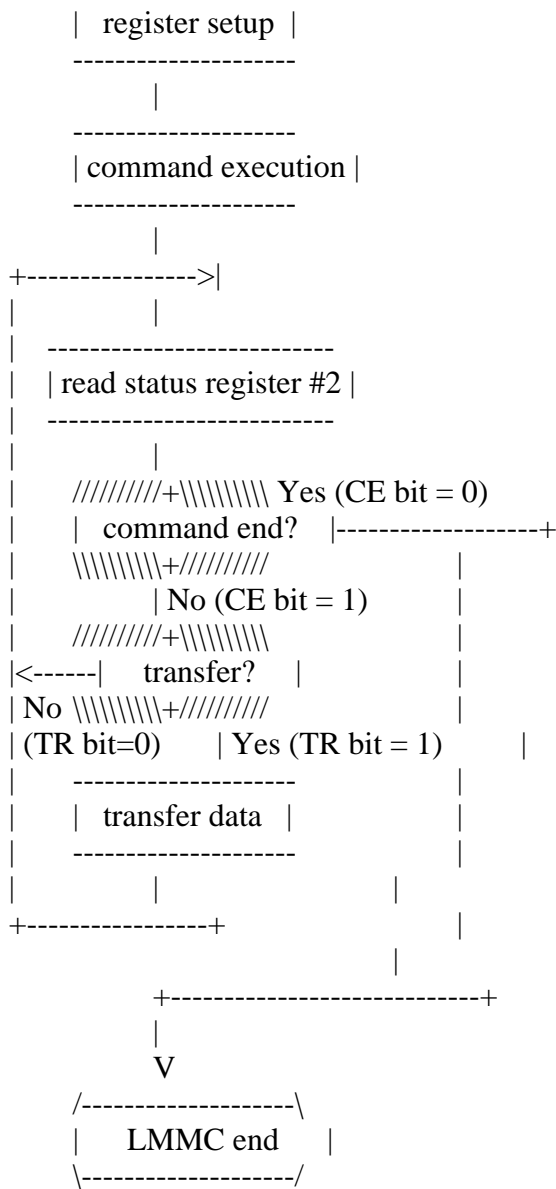


Figure 4.86 LMMC command execution flow chart





List 4.12 Example of LMMC command execution

=====

```

;*****
; List 4.12    LMMC sample
;             to use, set H, L, D, E, IX, A and go
;             RAM (IX) ---> VRAM (H,L)-(D,E) (logi-OP : A)
;*****
;
RDVDP: EQU    0006H
WRVDP: EQU    0007H

;----- program start -----

LMMC: DI                ;disable interrupt
      LD  B,A           ;B := LOGICAL OPERATION
      CALL WAIT.VDP     ;wait end of command

      LD  A,(WRVDP)
      LD  C,A
      INC C             ;C := PORT#1's address
      LD  A,36

```

```
OUT (C),A
LD A,80H+17
OUT (C),A ;R#17 := 36
```

```
INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),H ;DX
OUT (C),A
OUT (C),L ;DY
OUT (C),A
```

```
LD A,H ;make NX and DIX
SUBA
LD D,00000100B
JR NC,LMMC1
LD D,00000000B
NEG
LMMC1: LD H,A ;H := NX , D := DIX
```

```
LD A,L
SUBA
LD E,00001000B
JR NC,LMMC2
LD E,00000000B
NEG
LMMC2: LD L,A ;L := NY , E := DIY
```

```
XOR A
OUT (C),H ;NX
OUT (C),A
OUT (C),L ;NY
OUT (C),A
LD A,(IX+0)
OUT (C),A ;first DATA
LD A,D
OR E
OUT (C),A ;DIX and DIY
```

```
LD A,B ;A := LOGICAL OPERATION
OR 10110000B ;LMMC command
OUT (C),A
```

```
DEC C
DEC C
```

```
LOOP: LD A,2
CALL GET.STATUS
BIT 0,A ;check CE bit
JP Z,EXIT
BIT 7,A ;check TR bit
JP Z,LOOP
INC IX
LD A,(IX+0)
OUT (C),A
JR LOOP
```

```
EXIT: LD A,0
      CALL GET.STATUS
```

```
EI
RET
```

```
GET.STATUS:
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET
```

```
WAIT.VDP:
LD A,2
CALL GET.STATUS
AND 1
JR NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET
```

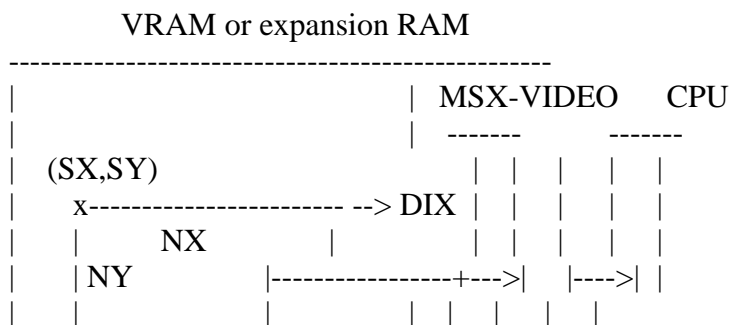
```
END
```

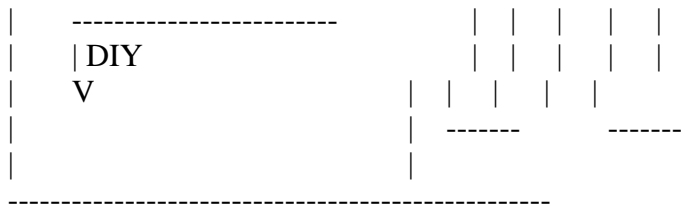
### 6.5.6 LMCM (VRAM - CPU logical transfer)

Data is transferred from the specified VRAM area to CPU in dots (see Figure 4.87)

After setting the parameters as shown in Figure 4.88, writing command code A0H in R#46 causes the command to be executed and data to be transferred from MSX-VIDEO. The CPU refers to the TR bit of S#2 and, since data of MSX-VIDEO has been prepared if this bit is "1", the CPU reads data from S#7. When CE bit of S#2 is "0", data comes to the end (see Figure 4.89). List 4.13 shows an example of using LMCM.

Figure 4.87 Action of LMCM command





**MXS:** select source memory      0 = VRAM, 1 = expansion RAM

**SX:** source origin X-coordinate (0 to 511)

**SY:** source origin Y-coordinate (0 to 1023)

**NX:** number of dots to be transferred in X direction (0 to 511)

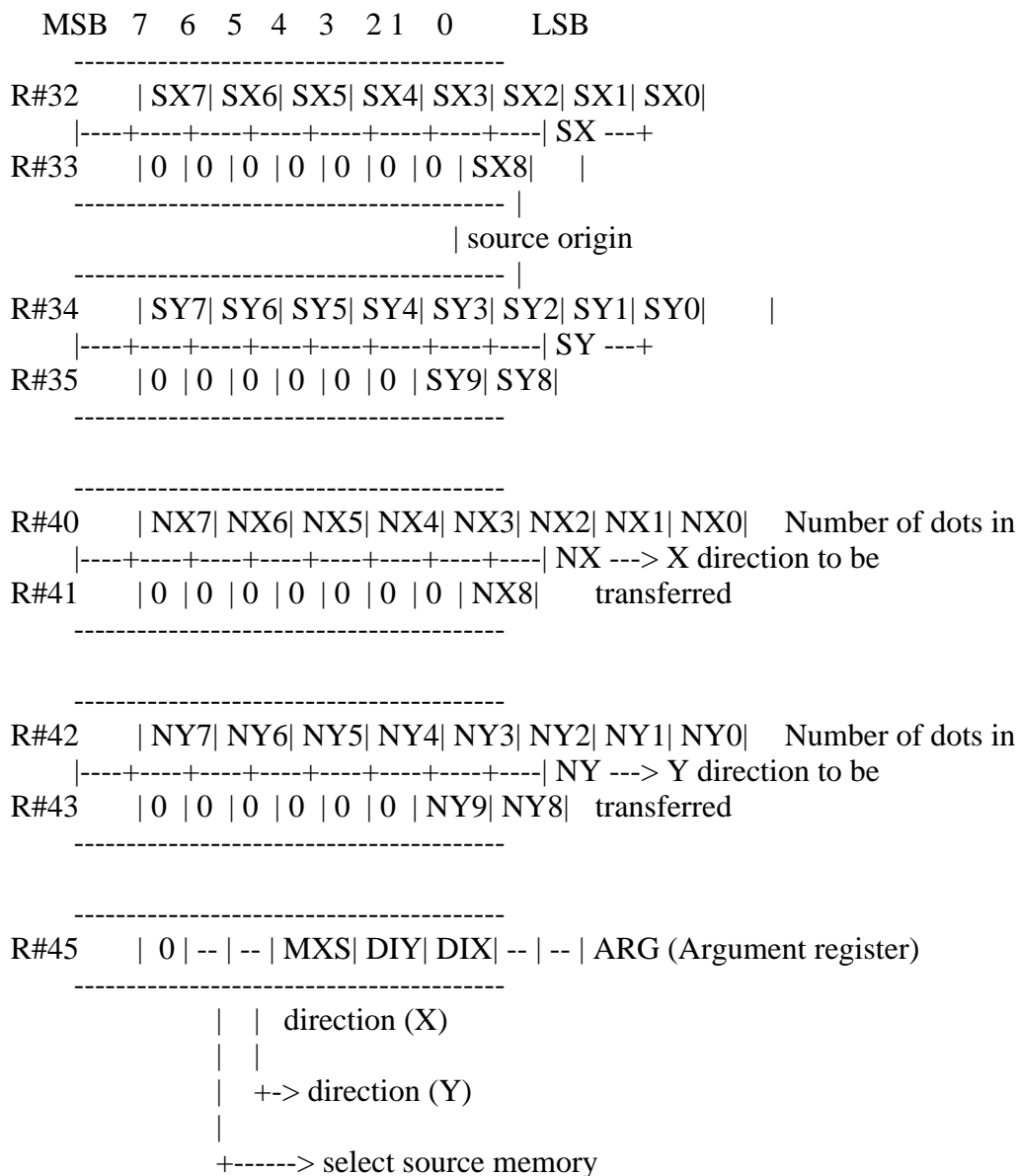
**NY:** number of dots to be transferred in Y direction (0 to 1023)

**DIX:** direction of NX from the origin    0 = right, 1 = left

**DIY:** direction of NY from the origin    0 = below, 1 = above

Figure 4.88 Register settings of LCMCM command

> LCMCM register setup



> LMCM command execution

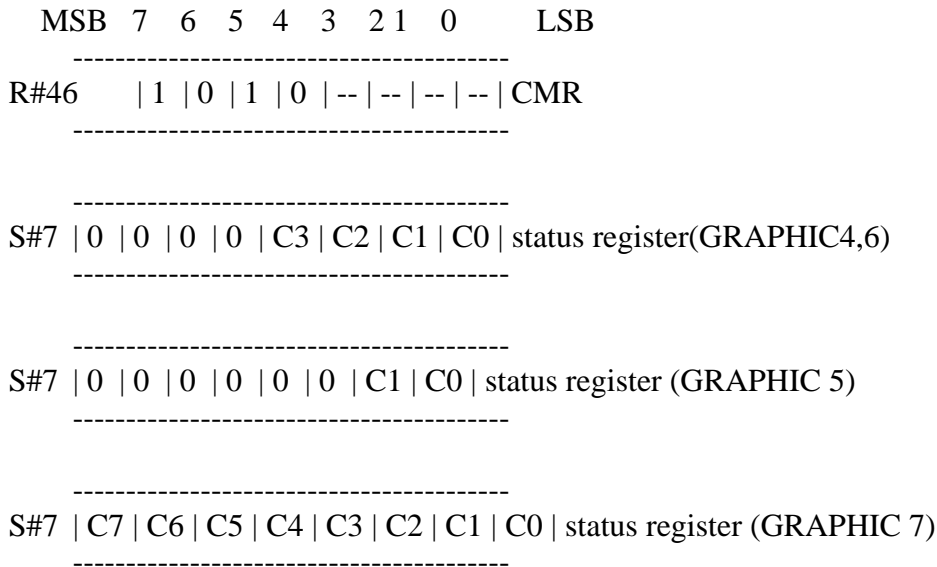
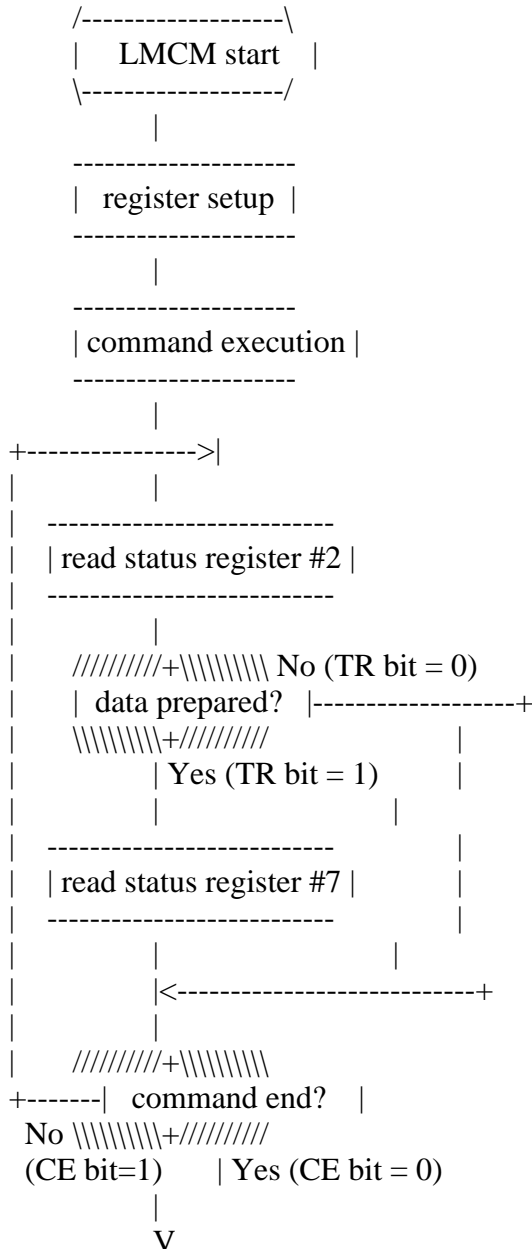


Figure 4.89 LMCM command execution flow chart





```

/-----\
|   LMCM end   |
\-----/

```

\* Note 1: Read status register #7 in "register setup", since TR bit should be reset before the command execution.

\* Note 2: Though last data was set in register #7 and TR bit was 1, the command would end inside of the MSX-VIDEO and CE would be zero.

#### List 4.13 Example of LMCM command execution

```

=====
;*****
; List 4.13   LMCM sample
;           to use, set H, L, D, E, IX, A and go
;           VRAM (H,L)-(D,E) ---> RAM (IX)
;*****
;
RDVDP: EQU   0006H
WRVDP: EQU   0007H

;----- program start -----

LMCM: DI           ;disable interrupt
      LD B,A       ;B := LOGICAL OPERATION
      CALL WAIT.VDP ;wait end of command

      LD A,(WRVDP)
      LD C,A
      INC C         ;C := PORT#1's address
      LD A,32
      OUT (C),A
      LD A,80H+17
      OUT (C),A     ;R#17 := 32
      INC C
      INC C         ;C := PORT#3's address
      XOR A
      OUT (C),H     ;SX
      OUT (C),A
      OUT (C),L     ;SY
      OUT (C),A
      OUT (C),A     ;dummy
      OUT (C),A     ;dummy
      OUT (C),A     ;dummy
      OUT (C),A     ;dummy
      LD A,H        ;make NX and DIX
      SUBA
      LD D,00000100B
      JR NC,LMCM1
      LD D,00000000B
      NEG
LMCM1: LD H,A       ;H := NX , D := DIX

      LD A,L
      SUBA

```

```

LD E,00001000B
JR NC,LMCM2
LD E,00000000B
NEG
LMCM2: LD L,A          ;L := NY , E := DIY

XOR A
OUT (C),H            ;NX
OUT (C),A
OUT (C),L            ;NY
OUT (C),A
LD A,(IX+0)
OUT (C),A            ;dummy
LD A,D
OR E
OUT (C),A            ;DIX and DIY
LD A,7
CALL GET.STATUS
LD A,B                ;A := LOGICAL OPERATION
OR 10100000B         ;LMCM command
OUT (C),A
LD A,(RDVDP)
LD C,A                ;C := PORT#1's address
LOOP: LD A,2
CALL GET.STATUS
BIT 0,A                ;check CE bit
JP Z,EXIT
BIT 7,A                ;check TR bit
JP Z,LOOP
LD A,7
CALL GET.STATUS
LD (IX+0),A
INC IX
JR LOOP

EXIT: LD A,0
CALL GET.STATUS
EI
RET

GET.STATUS:
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

WAIT.VDP:
LD A,2
CALL GET.STATUS
AND 1

```

```

JR  NZ, WAIT.VDP
XOR  A
CALL GET.STATUS
RET

END

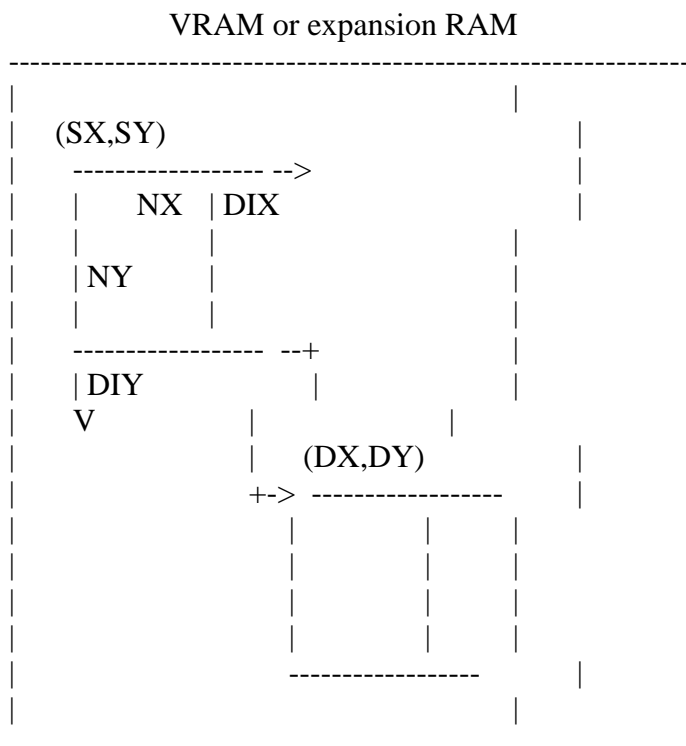
```

### 6.5.7. LMMM (VRAM->VRAM logical transfer)

Data of the specified VRAM area is transferred into another VRAM area in dots (see figure 4.9)

After setting the parameters as shown in Figure 4.91, writing command code 9XH (X means a logical operation) in R#46 causes the command to be executed. While the CE bit of S#2 is "1", the command is being executed. List 4.14 shows an example of using LMMM.

Figure 4.90 Actions of LMMM command



**MXS**: select the source memory      0 = VRAM, 1 = expansion RAM

**MXD**: select the destination memory      0 = VRAM, 1 = expansion RAM

**SX**: source origin X-coordinate (0 to 511)

**SY**: source origin Y-coordinate (0 to 1023)

**NX**: number of dots to be transferred in X direction (0 to 511)

**NY**: number of dots to be transferred in Y direction (0 to 1023)

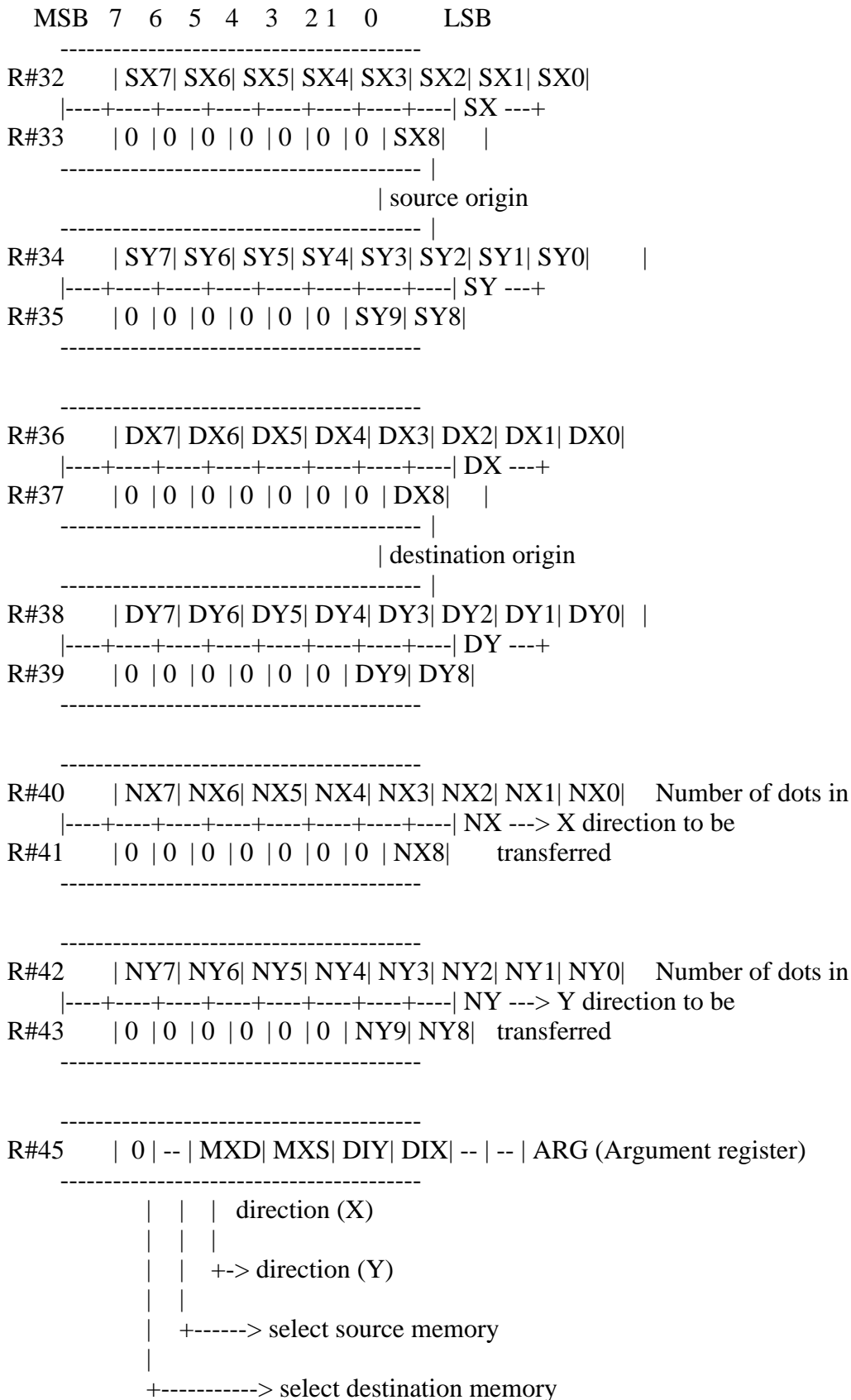
**DIX**: direction of NX from the origin      0 = right, 1 = left

**DIY**: direction of NY from the origin      0 = below, 1 = above

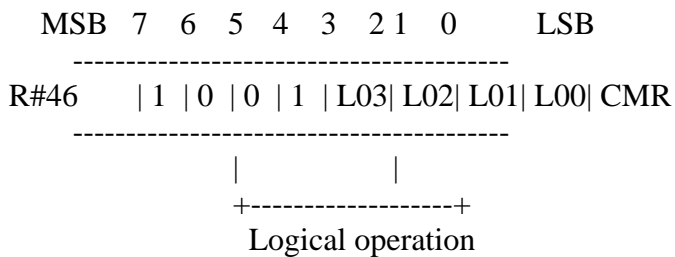
DX: destination origin X-coordinate (0 to 511)  
 DY: destination origin Y-coordinate (0 to 1023)

Figure 4.91 Register settings of LMMM command

> LMMM register setup



> LMMM command execution



List 4.14 Example of LMMM command execution

```

*****
; List 4.14 LMMM sample
; to use, set H, L, D, E, B, C, A and go
; VRAM (H,L)-(D,E) ---> VRAM (B,C) (logi-OP : A)
*****
;
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

LMMM: DI ;disable interrupt
PUSH AF ;save LOGICAL OPERATION
PUSH BC ;save DESTINATION
CALL WAIT.VDP ;wait end of command

LD A,(WRVDP)
LD C,A
INC C ;C := PORT#1's address
LD A,32
OUT (C),A
LD A,80H+17
OUT (C),A ;R#17 := 32

INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),H ;SX
OUT (C),A
OUT (C),L ;SY
OUT (C),A

LD A,H ;make NX and DIX
SUBA
LD D,00000100B
JP NC,LMMM1
LD D,00000000B
NEG
LMMM1: LD H,A ;H := NX , D := DIX

LD A,L ;make NY and DIY
SUBA
LD E,00001000B
JP NC,LMMM2
LD E,00000000B

```

```

NEG
LMMM2:  LD L,A          ;L := NY , E := DIY

```

```

LD A,D
OR E
POP DE          ;restore DX,DY
PUSH AF        ;save DIX,DIY
XOR A
OUT (C),D      ;DX
OUT (C),A
OUT (C),E      ;DY
OUT (C),A
OUT (C),H      ;NX
OUT (C),A
OUT (C),L      ;NY
OUT (C),A
OUT (C),A      ;dummy
POP AF
OUT (C),A      ;DIX and DIY

```

```

POP AF          ;A := LOGICAL OPERATION
OR 10010000B   ;LMMM command
OUT (C),A

```

```

EI
RET

```

GET.STATUS:

```

.  PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

WAIT.VDP:

```

LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

```

```

END

```

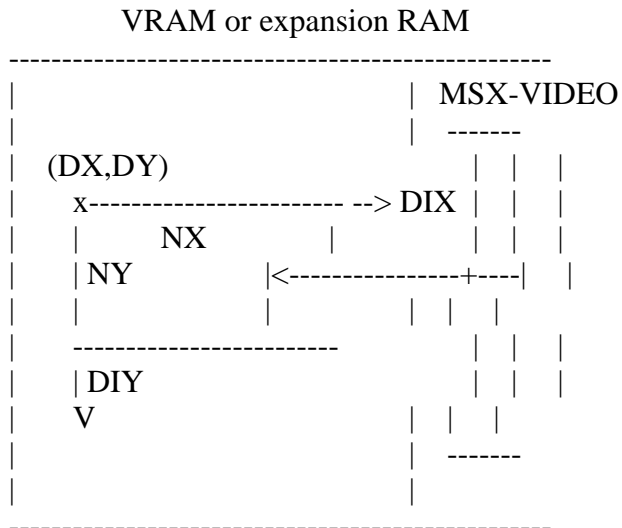
### 6.5.8 LMMV (VRAM logical paint)

The specified VRAM area is painted by the colour code in dots (see Figure

4.92). Logical operations between data in VRAM and the specified data are allowed.

After setting the parameters as shown in Figure 4.93, writing command code 8Xh (X means a logical operation) in R#46 causes the command to be executed. While the CE bit of S#2 is "1", the command is being executed. List 4.15 shows an example of using LMMV.

Figure 4.92 Actions of LMMV command



MXD: select memory                    0 = VRAM, 1 = expansion RAM

NX: number of dots to be painted in X direction (0 to 511)

NY: number of dots to be painted in Y direction (0 to 1023)

DIX: direction of NX from the origin   0 = right, 1 = left

DIY: direction of NY from the origin   0 = below, 1 = above

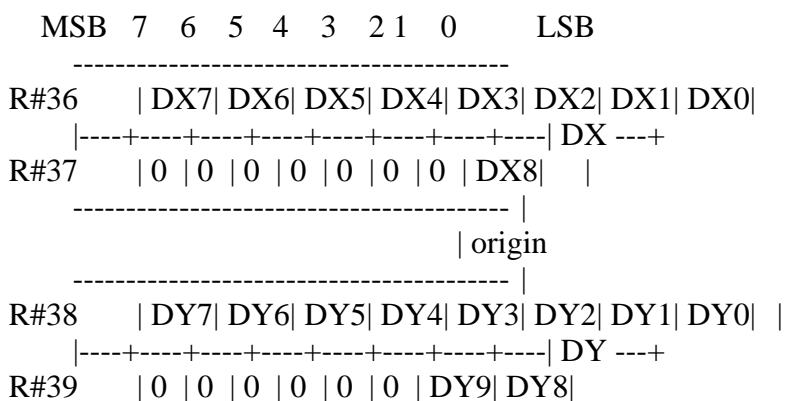
DX: origin X-coordinate (0 to 511)

DY: origin Y-coordinate (0 to 1023)

CLR (R#44:Colour register): Painted data

Figure 4.93 Register settings of LMMV command

> LMMV register setup



```

-----
R#40   | NX7| NX6| NX5| NX4| NX3| NX2| NX1| NX0|  number of dots in
|-----+-----+-----+-----+-----+-----+-----+-----| NX ---> X direction to
R#41   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | NX8|  be painted
-----

```

```

-----
R#42   | NY7| NY6| NY5| NY4| NY3| NY2| NY1| NY0|  number of dots in
|-----+-----+-----+-----+-----+-----+-----+-----| NY ---> Y direction to
R#43   | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | NY9| NY8|  be painted
-----

```

```

-----
R#44   | 0 | 0 | 0 | 0 | CR3| CR2| CR1| CR0| CLR (GRAPHIC 4,6)  |data
-----
|
|to
|
| 0 | 0 | 0 | 0 | 0 | 0 | CR1| CR0| CLR (GRAPHIC 5) |be
-----
|
|tran
|
| CR7| CR6| CR5| CR4| CR3| CR2| CR1| CR0| CLR (GRAPHIC 7) |rred
-----
--+
```

```

-----
R#45   | 0 | -- | MXD| -- | DIY| DIX| -- | -- | ARG (Argument register)
-----
|
| painting direction (X)
|
| +-> painting direction (Y)
|
+-----> memory selection

```

> LMMV command execution

```

MSB 7 6 5 4 3 2 1 0  LSB
-----
R#46   | 1 | 0 | 0 | 0 | L03| L03| L01| L00| CMR
-----
|
|
+-----+
Logical operation

```

List 4.15 Example of LMMV command execution

```

*****
;
; List 4.15 LMMV sample
;
; to use, set H, L, D, E, B, A and go
;
; data B ---> fill VRAM (H,L)-(D,E) (logi-op : A)
*****
;
RDVDP: EQU    0006H
WRVDP: EQU    0007H

```



;----- program start -----

```
LMMV: DI          ;disable interrupt
      PUSH  AF     ;save LOGICAL OPERATION
      PUSH  BC     ;save FILL DATA
      CALL  WAIT.VDP ;wait end of command
```

```
LD  A,(WRVDP)
LD  C,A
INC C          ;C := PORT#1's address
LD  A,36
OUT  (C),A
LD  A,80H+17
OUT  (C),A    ;R#17 := 36
```

```
INC C
INC C          ;C := PORT#3's address
XOR  A
OUT  (C),H    ;DX
OUT  (C),A
OUT  (C),L    ;DY
OUT  (C),A
```

```
LD  A,H      ;make NX and DIX
SUBA
LD  D,00000100B
JP  NC,LMMV1
LD  D,00000000B
NEG
```

```
LMMV1: LD  H,A      ;H := NX , D := DIX
```

```
LD  A,L      ;make NY and DIY
SUBA
LD  E,00001000B
JP  NC,LMMV2
LD  E,00000000B
NEG
```

```
LMMV2: LD  L,A      ;L := NY , E := DIY
```

```
XOR  A
OUT  (C),H    ;NX
OUT  (C),A
OUT  (C),L    ;NY
OUT  (C),A
POP AF
OUT  (C),A    ;FILL DATA
LD  A,D
OR  E
OUT  (C),A    ;DIX and DIY
```

```
POP AF          ;restore LOGICAL OPERATION
OR  A,10000000B ;LMMV command
OUT  (C),A
```

```
EI
RET
```

```

GET.STATUS:
  PUSH  BC
  LD  BC,(WRVDP)
  INC  C
  OUT  (C),A
  LD  A,8FH
  OUT  (C),A
  LD  BC,(RDVDP)
  INC  C
  IN  A,(C)
  POP  BC
  RET

```

```

WAIT.VDP:
  LD  A,2
  CALL GET.STATUS
  AND  1
  JP  NZ,WAIT.VDP
  XOR  A
  CALL GET.STATUS
  RET

  END

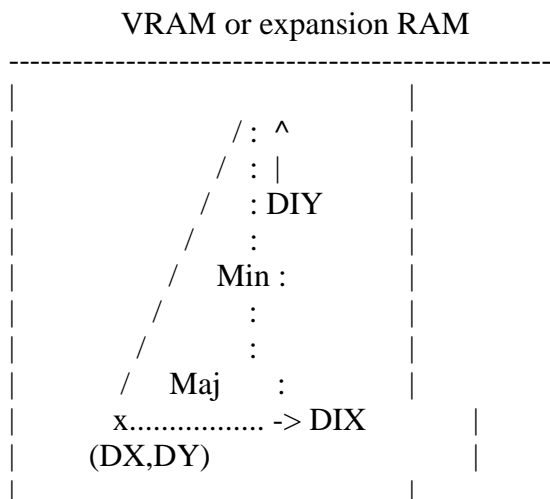
```

### 6.5.9 LINE (drawing a line)

Lines can be drawn between any coordinates in VRAM. The parameters to be specified include the (X,Y) coordinates of the starting point and the X and Y lengths in units to the ending point (see Figure 4.94). Logical operations between data in VRAM and the specified data are allowed.

After setting the parameters as shown in Figure 4.94, writing command code 7XH (X means a logical operation) in R#46 causes the command to be executed. While the CE bit of S#2 is "1", the command is being executed. List 4.16 shows an example of using LINE.

Figure 4.94 Actions of LINE command



-----  
MXD: select memory                    0 = VRAM, 1 = expansion RAM

Maj: number of dots of major side (0 to 1023)

Min: number of dots of minor side (0 to 512)

MAJ: 0 = The major side is parallel to X axis

MAJ: 1 = The major side is parallel to Y axis,  
      or the major side = the minor side

DIX: direction of the end from the origin    0 = right, 1 = left

DIY: direction of the end from the origin    0 = below, 1 = above

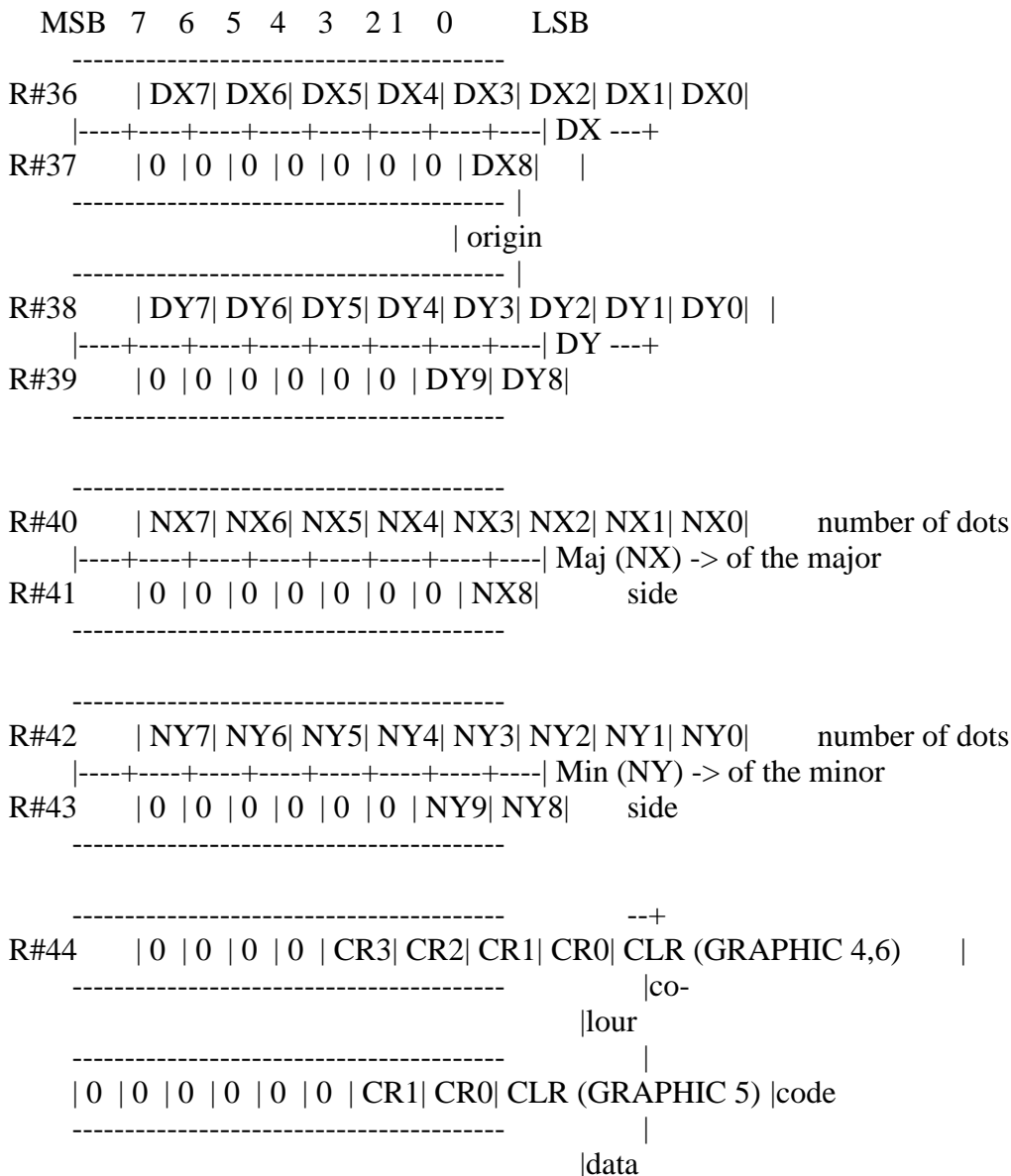
DX: origin X-coordinate (0 to 511)

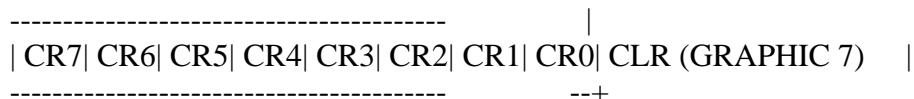
DY: origin Y-coordinate (0 to 1023)

CLR (R#44:Colour register): Line colour data

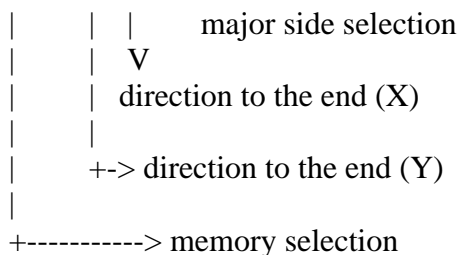
Figure 4.95 Register settings of LINE command

> LINE register setup

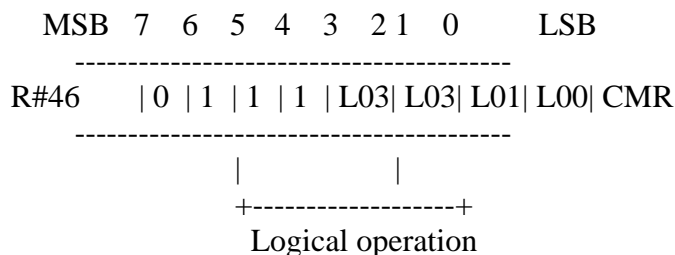




R#45 | 0 | -- | MXD | -- | DIY | DIX | -- | MAJ | ARG (Argument register)



> LINE command execution



List 4.16 Example of LINE command execution

```

=====
;*****
; List 4.16 LINE sample
; to use, set H, L, D, E, B, A and go
; draw LINE (H,L)-(D,E) with color B, log-op A
;*****
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

LINE: DI ;disable interrupt
PUSH AF ;save LOGICAL OPERATION
PUSH BC ;save COLOR
CALL WAIT.VDP ;wait end of command
LD A,(WRVDP)
LD C,A
INC C ;C := PORT#1's address
LD A,36
OUT (C),A
LD A,80H+17
OUT (C),A ;R#17 := 36

INC C
INC C ;C := PORT#3's address
XOR A
OUT (C),H ;DX
OUT (C),A

```

```
OUT (C),L ;DY
OUT (C),A
```

```
LD A,H ;make DX and DIX
SUBD
LD D,00000100B
JP NC,LINE1
LD D,00000000B
NEG
```

```
LINE1: LD H,A ;H := DX , D := DIX
```

```
LD A,L ;make DY and DIY
SUBE
LD E,00001000B
JP NC,LINE2
LD E,00000000B
NEG
```

```
LINE2: LD L,A ;L := DY , E := DIY
```

```
CP H ;make Maj and Min
JP C,LINE3
XOR A
OUT (C),L ;long side
OUT (C),A
OUT (C),H ;short side
OUT (C),A
LD A,00000001B ;MAJ := 1
JP LINE4
```

```
LINE3: XOR A
OUT (C),H ;NX
OUT (C),A
OUT (C),L ;NY
OUT (C),A
LD A,00000000B ;MAJ := 0
```

```
LINE4: OR D
OR E ;A := DIX , DIY , MAJ
POPHL ;H := COLOR
OUT (C),H
OUT (C),A
POP AF ;A := LOGICAL OPERATION
OR 01110000B
OUT (C),A
LD A,8FH
OUT (C),A
EI
RET
```

```
GET.STATUS:
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
```

```

INC C
IN  A,(C)
POP BC
RET

```

```

WAIT.VDP:
LD  A,2
CALL GET.STATUS
AND  1
JP  NZ,WAIT.VDP
XOR  A
CALL GET.STATUS
RET

END

```

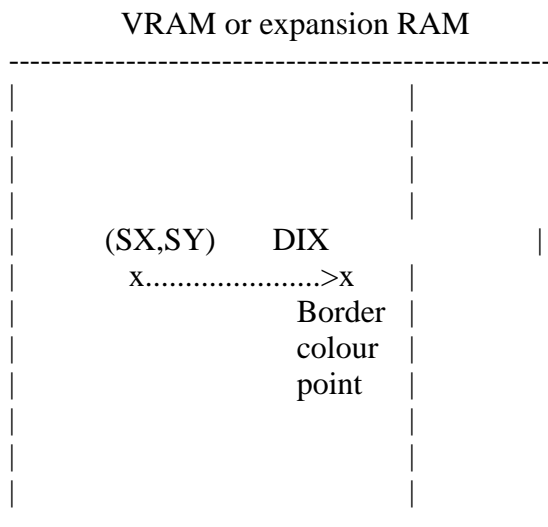
### 6.5.10 SRCH (colour code search)

SRCH searches for the existence of the specified colour from any coordinate on VRAM to the right or the left (see figure 4.96). This is very useful for paint routines.

After setting the parameters as shown in Figure 4.97, writing 60H in R#46 causes the command to be executed. The command terminates when the objective colour is found or when it cannot be found after searching for it to the screen edge. While the CE bit of S#2 is "1", the command is being executed (see Figure 4.98).

After the command ends, the objective colour code is stored in S#8 and S#9. List 4.17 shows an example of using SRCH.

Figure 4.96 Actions of SRCH command



MXD: memory selection for the search 0 = VRAM, 1 = expansion RAM

SX: search origin X-coordinate (0 to 511)

SY: search origin Y-coordinate (0 to 1023)

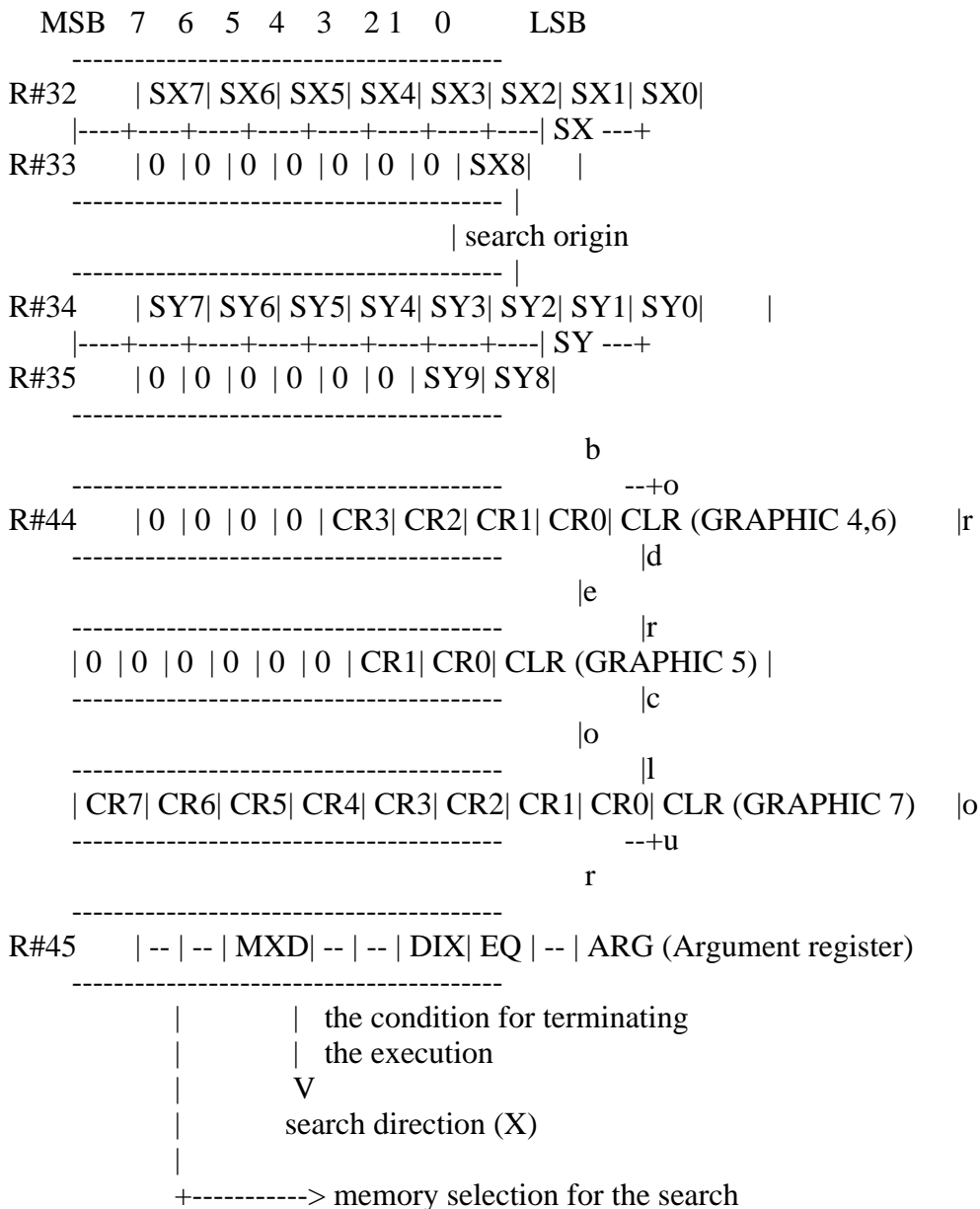
DIX: direction for the search from the origin 0 = right, 1 = left

EQ: 0 = ends the execution when the border colour is found  
1 = ends the execution when the colour is found other than the border colour

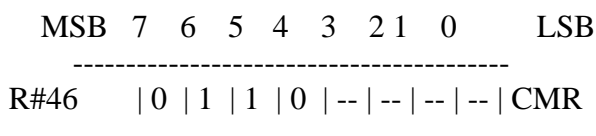
CLR (R#44:Colour register): border colour

Figure 4.97 Register settings of SRCH command

> SRCH register setup



> SRCH command execution



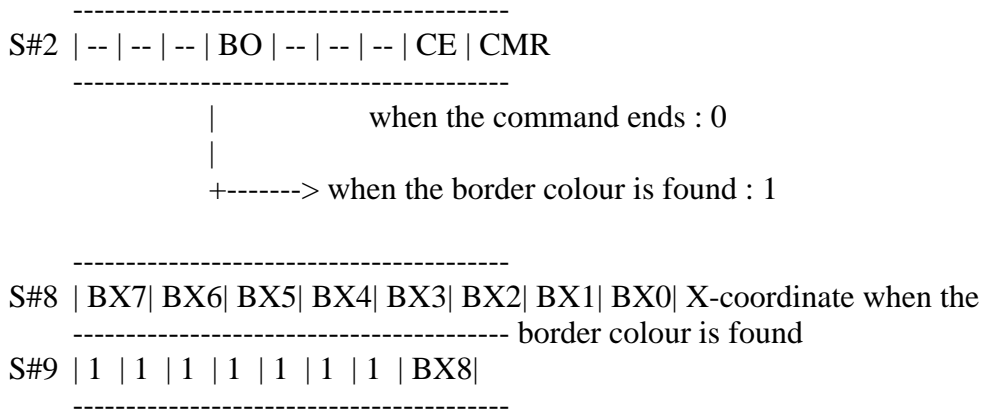
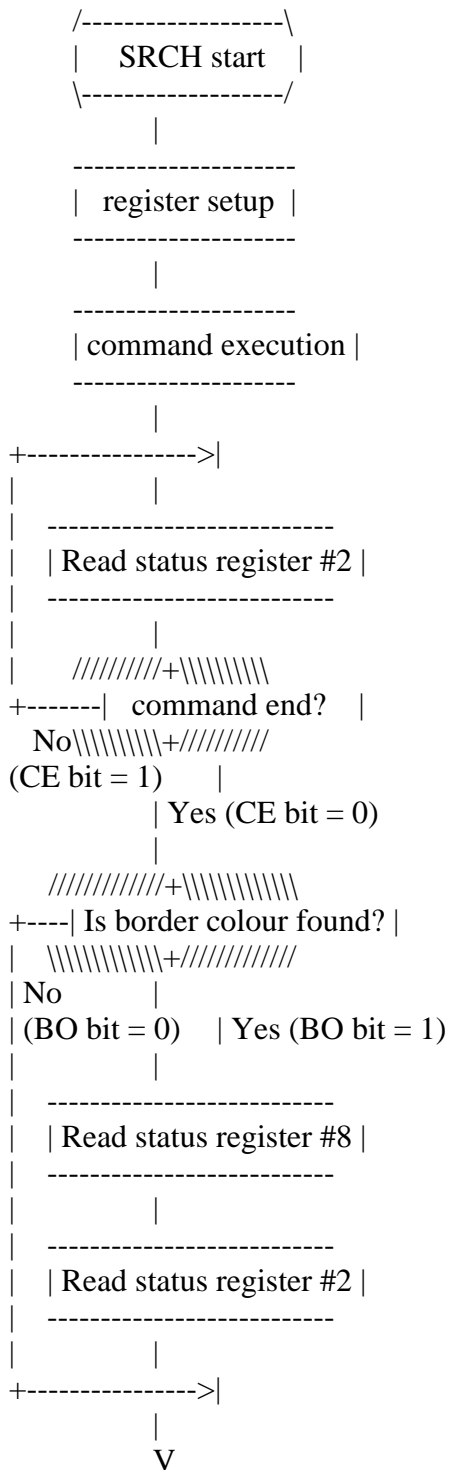


Figure 4.98 SRCH command execution flowchart





```

/-----\
|  SRCH end  |
\-----/

```

#### List 4.17 Example of SRCH command execution

---

```

;*****
; List 4.17 SRCH sample
; to use, set H, L, E, A as follows
; srch (x:H, y:L, color:E, arg(reg#45) : A)
; returns: Z (not found)
; NZ (A := X)
;*****
;
RDVDP: EQU 0006H
WRVDP: EQU 0007H

;----- program start -----

SRCH: DI ;disable interrupt
      PUSH AF ;save arg
      CALL WAIT.VDP

      LD A,(WRVDP)
      LD C,A
      INC C ;C := PORT#1's address

      LD D,0
      LD A,32+80H
      OUT (C),H
      OUT (C),A ;R#32 := H
      INC A
      OUT (C),D
      OUT (C),A ;R#33 := 0
      INC A
      OUT (C),L
      OUT (C),A ;R#34 := L
      INC A
      OUT (C),D
      OUT (C),A ;R#35 := 0
      LD A,44+80H
      OUT (C),E
      OUT (C),A ;R#44 := E
      INC A
      LD E,A
      POP AF ;A := ARG
      OUT (C),A
      OUT (C),E ;R#45 := A

      LD A,01100000B
      OUT (C),A
      INC E
      OUT (C),E ;R#46 := SRCH command

LOOP: LD A,2

```

```

CALL GET.STATUS
BIT 0,A
JP NZ,LOOP
LD E,A
LD A,8
CALL GET.STATUS
LD D,A
LD A,9
CALL GET.STATUS
LD A,D
BIT 4,E

EI
RET

```

```

GET.STATUS:
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

```

WAIT.VDP:
LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

END

```

---

List 4.18 Simple PAINT routine using SRCH and LINE

---

```

;*****
; List 4.18 SRCH and LINE sample
; search color to right and left,
; then draw line between the two points
;*****
;
EXTRN SRCH
EXTRN LINE

Y EQU 0A800H
X EQU 0A801H
COL EQU 0A802H

```

```
ARG EQU 0A803H
PCOL EQU 0A804H
```

```
;----- program start -----
```

```
MAIN: LD (STK),SP
      LD SP,AREA
      LD HL,(Y)
      LD A,(COL)
      LD E,A
      LD A,(ARG)
      PUSH HL
      PUSH DE
      SET 2,A
      CALL SRCH
      POP DE
      POP HL
      JP NZ,S1
      LD A,(X)
      DEC A
S1: INC A
     PUSH AF
     LD A,(ARG)
     RES 2,A
     CALL SRCH
     JP NZ,S2
     LD A,(X)
     INC A
S2: DEC A
     LD D,A
     POP AF
     LD H,A
     LD A,(Y)
     LD L,A
     LD E,A
     LD A,(PCOL)
     LD B,A
     LD A,0 ;PSET
     CALL LINE
     LD SP,(STK)
     RET
```

```
;----- work area -----
```

```
STK: DS 2
      DS 200
AREA: $
```

```
END
```

```
=====  
List 4.19 Example of the use of simple PAINT routine  
=====
```

```
1000 '*****'
```

```

1010 ' list 4.19  SRCH and LINE sample
1020 ' Operate cursor while holding down the space bar.
1030 '*****
1040 '
1050 SCREEN 5
1060 FOR I=0 TO 50:LINE -(RND(1)*255,RND(1)*211),15:NEXT
1070 I=&HA000 :DEF USR=I
1080 READ A$
1090 IF A$="END" THEN 1130
1100 POKE I,VAL("&H"+A$):I=I+1
1110 READ A$
1120 GOTO 1090
1130 X=128:Y=100:COL=15:PCOL=2:ARG=0
1140 CURS=0
1150 A=STICK(0)
1160 CURS=(CURS+1) AND 1
1170 LINE (X-5,I)-(X+5,I),15,,XOR
1180 LINE (X,Y-5)-(X,Y+5),15,,XOR
1190 IF CURS=1 THEN 1290
1200 IF A=1 THEN Y=Y-1
1210 IF A=2 THEN Y=Y-1:X=X+1
1220 IF A=3 THEN X=X+1
1230 IF A=4 THEN X=X+1:Y=Y+1
1240 IF A=5 THEN Y=Y+1
1250 IF A=6 THEN Y=Y+1:X=X-1
1260 IF A=7 THEN X=X-1
1270 IF A=8 THEN X=X-1:Y=Y-1
1280 IF STRIG(9) THEN GOSUB 1300
1290 GOTO 1150
1300 POKE &HA800,Y
1310 POKE &HA801,X
1320 POKE &HA802,COL
1330 POKE &HA803,ARG
1340 POKE &HA804,PCOL
1350 A=USR(0)
1360 RETURN
1370 DATA ED,73,80,A8,31,4A,A9,2A,00,A8,3A,02
1380 DATA A8,5F,3A,03,A8,E5,D5,CB,D7,CD,AD
1390 DATA A0,D1,E1,C2,21,A0,3A,01,A8
1400 DATA 3D,3C,F5,3A,03,A8,CB,97,CD,AD,A0,C2
1410 DATA 32,A0,3A,01,AB,3C,3D,57,F1,67,3A
1420 DATA 00,A8,6F,5F,3A,04,A8,47,3E
1430 DATA 00,CD,49,A0,ED,7B,80,A8,C9,F3,F5,CD
1440 DATA 0D,A1,C5,3A,06,00,4F,0C,3E,24,ED
1450 DATA 79,3E,91,ED,79,0C,0C,AF,ED
1460 DATA 61,ED,79,ED,69,ED,79,7C,92,16,04,D2
1470 DATA 72,A0,16,00,ED,44,67,7D,93,1E,08
1480 DATA D2,7E,A0,1E,00,ED,44,BC,DA
1490 DATA 90,A0,ED,79,AF,ED,79,ED,61,ED,79,26
1500 DATA 01,C3,9C,A0,ED,61,67,AF,ED,79,ED
1510 DATA 61,ED,79,26,00,7C,B2,B3,E1
1520 DATA ED,61,ED,79,F1,E6,0F,F6,70,ED,79,FB
1530 DATA C9,F5,F3,CD,0D,A1,ED,4B,06,00,0C
1540 DATA 3E,A0,16,00,ED,61,ED,79,3C
1550 DATA ED,51,ED,79,3C,ED,69,ED,79,3C,ED,51
1560 DATA ED,79,3E,AC,ED,59,ED,79,3C,5F,F1
1570 DATA ED,79,ED,59,3E,60,ED,79,1C

```

```

1580 DATA ED,59,3E,02,CD,FD,A0,CB,47,C2,E2,A0
1590 DATA 5F,3E,08,CD,FD,A0,57,3E,00,CD,FD
1600 DATA A0,7A,CB,63,FB,C9,C5,ED,4B
1610 DATA 06,00,0C,ED,79,3E,8F,ED,79,ED,78,C1
1620 DATA C9,3E,02,CD,FD,A0,E6,01,C2,0D,A1
1630 DATA AF,CD,FD,A0,C9,END

```

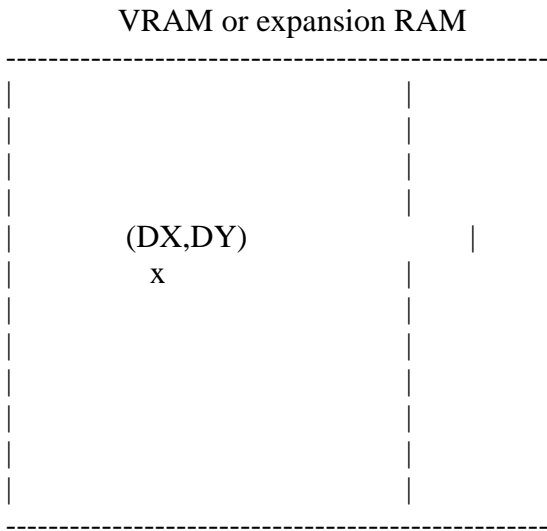
=====

### 6.5.11 PSET (drawing a point)

A point is drawn at any coordinate in VRAM (see figure 4.99).

After setting the parameters as shown in Figure 4.100, writing 5XH (X means a logical operation) in R#46 causes the command to be executed. While the CE bit of S#2 is "1", the command is being executed. List 4.20 shows an example of using PSET.

Figure 4.99 Actions of PSET command



MXD: memory selection            0 = VRAM, 1 = expansion RAM

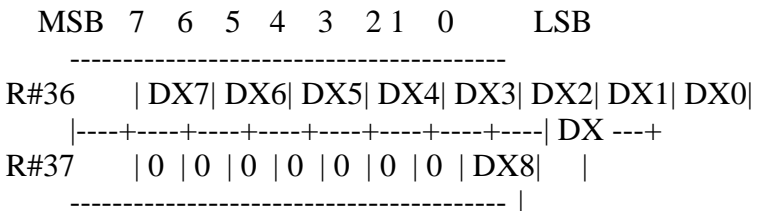
DX: origin X-coordinate (0 to 511)

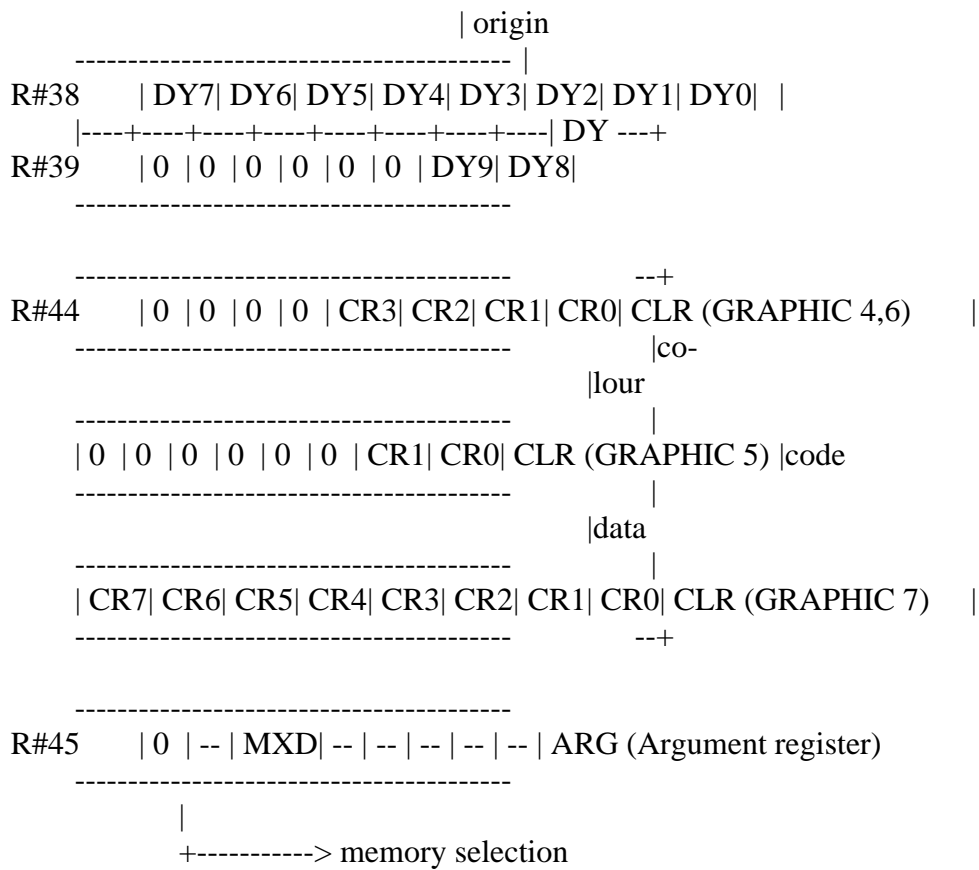
DY: origin Y-coordinate (0 to 1023)

CLR (R#44:Colour register): point colour

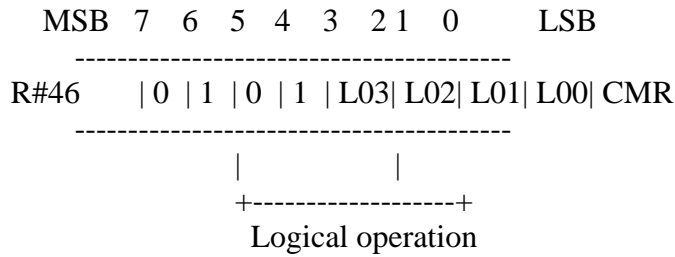
Figure 4.100 Register settings of PSET command

> PSET register setup





> PSET command execution



List 4.20 Example of PSET command execution

```

=====
;*****
; List 4.20 PSET sample
; to use, set H, L, E, A as follows
; pset (x:H, y:L), color:E, logi-OP:A
;*****
;
PUBLIC PSET

RDVDP: EQU    0006H
WRVDP: EQU    0007H

;----- program start -----

PSET:  DI
        PUSH  AF
        CALL  WAIT.VDP
        LD   BC,(WRVDP)

```

```
INC C
LD A,36
OUT (C),A
LD A,80H+17
OUT (C),A
```

```
PUSH BC
INC C
INC C
XOR A
OUT (C),H
OUT (C),A
OUT (C),L
OUT (C),A
POP BC
```

```
LD A,44
OUT (C),A
LD A,80H+17
OUT (C),A
```

```
INC C
INC C
OUT (C),E
XOR A
OUT (C),A
```

```
LD E,01010000B
POP AF
OR E
OUT (C),A
```

```
EI
RET
```

GET.STATUS:

```
PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET
```

WAIT.VDP:

```
LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET
```

END

---

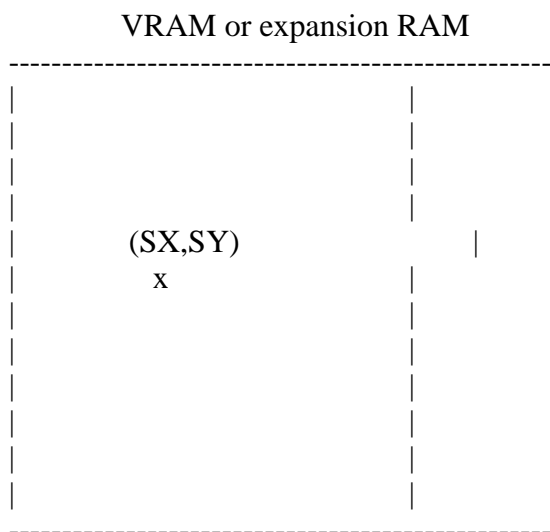
---

### 6.5.12 POINT (reading a colour code)

POINT reads the colour code in any coordinate of VRAM (see Figure 4.101).

After setting the parameters as shown in Figure 4.102, writing 40H in R#46 causes the command to be executed. While the CE bit of S#2 is "1", the command is being executed. After the command terminates, the colour code of the specified coordinate is set in S#7. List 4.21 shows an example of using POINT.

Figure 4.101 Actions of POINT command



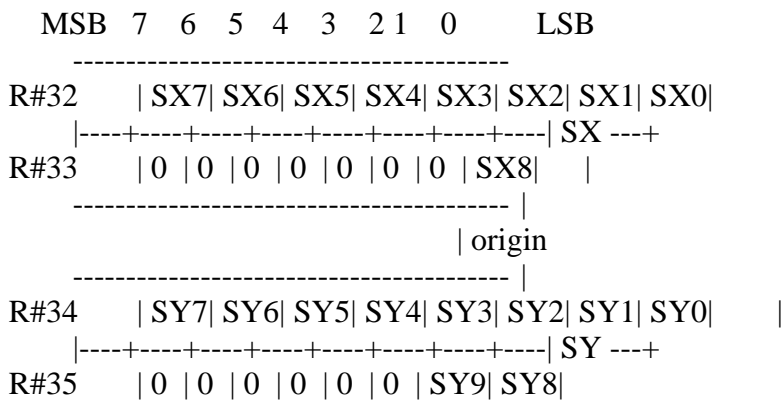
MXD: memory selection            0 = VRAM, 1 = expansion RAM

SX: origin X-coordinate (0 to 511)

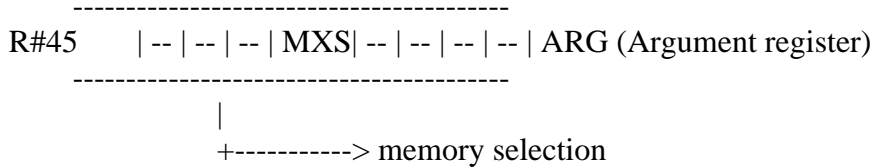
SY: origin Y-coordinate (0 to 1023)

Figure 4.102 Register settings of POINT command

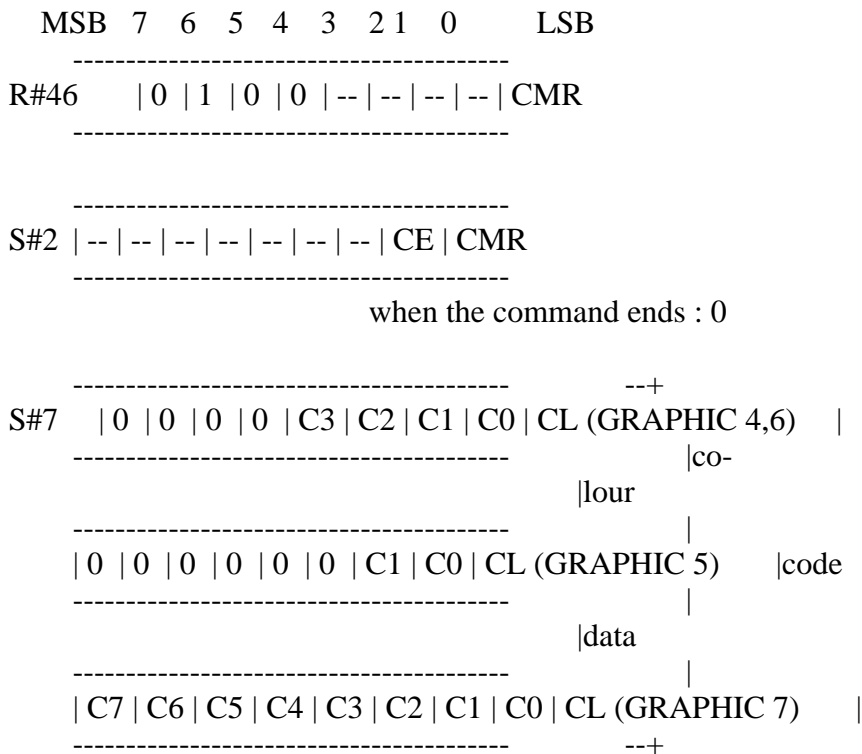
> POINT register setup







> POINT command execution



List 4.21 Example of POINT command execution

```

=====
;*****
; List 4.21 POINT sample
; to use, set H, L as follows
; POINT ( x:H, y:L )
; returns: A := COLOR CODE
;*****
;
PUBLIC POINT

RDVDP: EQU    0006H
WRVDP: EQU    0007H

;----- program start -----

POINT: DI
      CALL    WAIT.VDP

      LD     A,(WRVDP)
      LD     C,A

      INC   C
      LD   A,32
      OUT  (C),A

```

```

LD A,80H+17
OUT (C),A

INC C
INC C
XOR A
OUT (C),H
OUT (C),A
OUT (C),L
OUT (C),A

DEC C
DEC C
OUT (C),A
LD A,80H+45
OUT (C),A
LD A,01000000B
OUT (C),A
LD A,80H+46
OUT (C),A
CALL WAIT.VDP
LD A,7
CALL GET.STATUS
PUSH AF
XOR A
CALL GET.STATUS
POP AF

EI
RET

```

GET.STATUS:

```

PUSH BC
LD BC,(WRVDP)
INC C
OUT (C),A
LD A,8FH
OUT (C),A
LD BC,(RDVDP)
INC C
IN A,(C)
POP BC
RET

```

WAIT.VDP:

```

LD A,2
CALL GET.STATUS
AND 1
JP NZ,WAIT.VDP
XOR A
CALL GET.STATUS
RET

```

END

=====

List 4.22 PAINT routine using PSET and POINT

```

=====
;*****
; List 4.22 paint routine using PSET and POINT
; ENTRY: X:H, Y:L, BORDER COLOR:D, PAINT COLOR:E
;*****
;
; EXTRN PSET
; EXTRN POINT

Q.LENGTH EQU 256*2*2
MAX.Y EQU 211

;----- paint main routine -----

PAINT: CALL POINT
        CP D
        RETZ
        CALL INIT.Q
        LD (COL),DE
        CALL PUT.Q
        LD A,(COL)
        LD E,A
        XOR A ;logi-OP : PSET
        CALL PSET
PAINT0: CALL GET.Q
        RETC
        INC H
        CALL NZ,PAINT.SUB
        DEC H
        JP Z,PAINT1
        DEC H
        CALL PAINT.SUB
        INC H
PAINT1: DECL
        LD A,-1
        CP L
        CALL NZ,PAINT.SUB
        INC L
        INC L
        LD A,MAX.Y
        CP L
        CALL NC,PAINT.SUB
        JP PAINT0

;----- check point and pset -----

PAINT.SUB:
        CALL POINT
        LD D,A
        LD A,(BORD)
        CP D
        RETZ
        LD A,(COL)
        CP D
        RETZ

```

```
LD E,A
XOR  A
CALL PSET
CALL PUT.Q
RET
```

;----- init Q.BUFFER pointer -----

```
INIT.Q:
PUSH HL
LD HL,Q.BUF
LD (Q.TOP),HL
LD (Q.BTM),HL
POP HL
RET
```

;----- put point to Q.BUF (X:H , Y:L) -----

```
PUT.Q:
EX DE,HL
LD HL,(Q.TOP)
LD BC,Q.BUF+Q.LENGTH+1
OR A ;clear CARRY
PUSH HL
SBC HL,BC
POP HL
JP C,PUT.Q1
LD HL,Q.BUF
```

```
PUT.Q1:
LD (HL),D
INC HL
LD (HL),E
INC HL
LD (Q.TOP),HL
EX DE,HL
RET
```

;----- take point data to D, E -----

```
; returns: NC H:x, L:y
; C buffer empty
```

```
GET.Q: LD HL,(Q.BTM)
LD BC,(Q.TOP)
OR A
SBC HL,BC
JP NZ,GET.Q0
SCF
RET
```

```
GET.Q0: LD HL,(Q.BTM)
LD BC,Q.BUF+Q.LENGTH+1
OR A
PUSH HL
SBC HL,BC
POP HL
JP C,GET.Q1
LD HL,Q.BUF
```

```

GET.Q1: LD  D,(HL)
        INC HL
        LD  E,(HL)
        INC HL
        LD  (Q.BTM),HL
        OR  A
        EX  DE,HL
        RET

```

;----- work area -----

```

COL     DS  1
BORD    DS  1
Q.TOP   DS  2
Q.BTM   DS  2
Q.BUF   DS  Q.LENGTH

```

END

---

#### List 4.23 Example of using the PAINT routine

---

```

1000 '*****
1010 ' list 4.23  paint routine using POINT and PSET
1020 ' Position cursor at beginnig of paint area and press the space bar.
1030 '*****
1040 '
1050 SCREEN 5
1060 FOR I=0 TO 50
1070 LINE -(RND(1)*255,RND(1)*211),15
1080 NEXT
1090 I=&HA000 :DEFUSR=I
1100 READ A$
1110 IF A$="END" THEN 1150
1120 POKE I,VAL("&H"+A$):I=I+1
1130 READ A$
1140 GOTO 1110
1150 X=128:Y=100:COL=15:PCOL=2
1160 CURS=0
1170 A=STICK(0)
1180 CURS=(CURS+1) AND 1
1190 LINE (X-5,I)-(X+5,I),15,,XOR
1200 LINE (X,Y-5)-(X,Y+5),15,,XOR
1210 IF CURS=1 THEN 1310
1220 IF A=1 THEN Y=Y-1
1230 IF A=2 THEN Y=Y-1:X=X+1
1240 IF A=3 THEN X=X+1
1250 IF A=4 THEN X=X+1:Y=Y+1
1260 IF A=5 THEN Y=Y+1
1270 IF A=6 THEN Y=Y+1:X=X-1
1280 IF A=7 THEN X=X-1
1290 IF A=8 THEN X=X-1:Y=Y-1
1300 IF STRIG(9) THEN GOSUB 1320
1310 GOTO 1170

```

```

1320 POKE &HA8CA,Y
1330 POKE &HA8CB,X
1340 POKE &HA8CD,COL
1350 POKE &HA8CC,PCOL
1360 A=USR(0)
1370 RETURN
1380 DATA ED,73,00,A8,31,CA,A8,2A,CA,A8,ED,5B,CC,A8,CD,67
1390 DATA A0,ED,7B,00,A8,C9,E5,21,D4,A8,22,D0,A8,22,D2,A8
1400 DATA E1,C9,EB,2A,D0,A8,01,D5,AC,B7,E5,ED,42,E1,DA,34
1410 DATA A0,21,D4,A8,72,23,73,23,22,D0,A8,EB,C9,2A,D2,A8
1420 DATA ED,4B,D0,A8,B7,ED,42,C2,4C,A0,37,C9,2A,D2,A8,01
1430 DATA D5,AC,B7,E5,ED,42,E1,DA,5D,A0,21,D4,A8,56,23,5E
1440 DATA 23,22,D2,A8,B7,EB,C9,CD,B8,A0,BA,C8,CD,16,A0,ED
1450 DATA 53,CE,A8,CD,22,A0,3A,CE,A8,5F,AF,CD,F4,A0,CD,3D
1460 DATA A0,D8,24,C4,A1,A0,25,CA,8F,A0,25,CD,A1,A0,24,2D
1470 DATA 3E,FF,BD,C4,A1,A0,2C,2C,3E,D3,BD,D4,A1,A0,C3,7E
1480 DATA A0,CD,B8,A0,57,3A,CF,A8,BA,C8,3A,CE,A8,BA,C8,5F
1490 DATA AF,CD,F4,A0,CD,22,A0,C9,F3,CD,3A,A1,ED,4B,06,00
1500 DATA 0C,3E,20,ED,79,3E,91,ED,79,0C,0C,AF,ED,61,ED,79
1510 DATA ED,69,ED,79,0D,0D,ED,79,3E,AD,ED,79,3E,40,ED,79
1520 DATA 3E,AE,ED,79,CD,3A,A1,3E,07,CD,2A,A1,F5,AF,CD,2A
1530 DATA A1,F1,FB,C9,F3,F5,CD,3A,A1,ED,4B,06,00,0C,3E,24
1540 DATA ED,79,3E,91,ED,79,C5,0C,0C,AF,ED,61,ED,79,ED,69
1550 DATA ED,79,C1,3E,2C,ED,79,3E,91,ED,79,0C,0C,ED,59,AF
1560 DATA ED,79,1E,50,F1,B3,ED,79,FB,C9,C5,ED,4B,06,00,0C
1570 DATA ED,79,3E,8F,ED,79,ED,78,C1,C9,3E,02,CD,2A,A1,E6
1580 DATA 01,C2,3A,A1,AF,CD,2A,A1,C9
1590 DATA END

```

=====

## 6.6 Speeding Up Commands

MSX-VIDEO performs various screen management duties in addition to executing the specified commands. Sometimes the command execution speed seems to be a bit slow because of this. Thus, by discarding these operations, the speed of the command executions can be made faster. This can be done using the following method.

### 1. Sprite display inhibition

This method is useful since speedup can be realised while the screen remains displayed. Set "1" to bit 1 of R#8.

### 2. Screen display inhibition

This method cannot be used frequently except in the case of initialising the screen, since the screen fades out in this mode. Set "1" to bit 6 of R#1.

## 6.7 Register Status at Command Termination

Table 4.7 shows the register status at the command termination for each command.

When the number of dots to be executed in Y direction assumes N, the values of SY\*, DY\*, and NYB can be calculated as follows:

SY\*=SY+N, DY\*=DY+N ..... when DIY bit is 0  
 SY\*=SY-N, DY\*=DY-N ..... when DIY bit is 1  
 NYB=NY-N

Note: when MAJ bit is 0 in LINE, N = N - 1.

Table 4.7 Register status at command termination

command name	SX	SY	DX	DY	NX	NY	CLR	CMR H	CMR L	ARG
HMMC	---	---	---	.	---	#	---	0	---	---
YMMM	---	.	---	.	---	#	---	0	---	---
HMMM	---	.	---	.	---	#	---	0	---	---
HMMV	---	---	---	.	---	#	---	0	---	---
LMMC	---	---	---	.	---	#	---	0	---	---
LMCM	---	.	---	---	---	#	.	0	---	---
LMMM	---	.	---	.	---	#	---	0	---	---
LMMV	---	---	---	.	---	#	---	0	---	---
LINE	---	---	---	.	---	---	---	0	---	---
SRCH	---	---	---	---	---	---	---	0	---	---
PSET	---	---	---	---	---	---	---	0	---	---
POINT	---	---	---	---	---	.	---	0	---	---

--- : no change

. : coordinate (SY\*, DY\*) and the colour code at the command termination

# : the number of counts (NYB), when the screen edge is fetched

## MSX2 TECHNICAL HANDBOOK

Edited by: ASCII Systems Division  
 Published by: ASCII Corporation - JAPAN  
 First edition: March 1987

Text file typed by: Nestor Soriano (Konami Man) - SPAIN  
 October 1997

Changes from the original:

- In Figure 5.2, unused bits are marked as "x", and inverted signals are marked with "\*", for easiest readability.
- Figure 5.17B was added.
- In List 5.4, the last line before the work area, "JR START", has been corrected to "JR SCAN".
- In Figure 5.18, the addresses for GETPNT y PUTPNT were swapped. They have been corrected.
- In description of BIOS routines PINLIN and INLIN, "BUF" address has been corrected from F55DH to F55EH.
- In Figure 5.22 (B), "Arabaic mode display" has been changed to "Arabic or kana mode display".
- In description of BIOS routine GTTRIG, the input needed for reading B buttons has been added in the "Input" field.
- In Table 5.5, in the Note 4, "the trigger button of the mouse or the trigger button" has been changed to "the trigger button of the mouse or the trigger button of the track ball".
- In Figure 5.29, "1200 or 2400 hours" indication has been corrected to "12 or 24 hours".
- In Figure 5.32, "Register 3 #11" indication has been corrected to "Register #11".
- In Figure 5.33, "Adjust Y (8 to +7)" has been corrected to "Adjust Y (-8 to +7)".
- In description of BIOS routine WRTCLK, the input needed in the A register has been added in the "Input" field.

-----

## CHAPTER 5 - ACCESS TO PERIPHERALS THROUGH BIOS (Parts 1 to 6)

The basic philosophy of MSX is to have a standard interface, independent of machines or versions, to access peripherals through BIOS. Thus, the user should get to know about using BIOS first. In chapter 5, accessing peripherals using BIOS and the structure used for each peripheral are described.

### 1. PSG AND SOUND OUTPUT

MSX has the following three kinds of sound output functions, but function (3) is not installed in the standard MSX, so it is not described in this manual. This section describes functions (1) and (2).

- (1) PSG sound output (3 channels, 8 octaves)
- (2) Sound output by 1 bit I/O port



(3) Sound output by MSX-AUDIO (FM sound generator) ..... not described  
in this manual

### 1.1. PSG functions

An AY-3-8910 compatible LSI is used for the MSX music play function and for BEEP tone generation. This LSI is referred to as the PSG (Programmable Sound Generator), and can generate complex music and various tones. It has the following features:

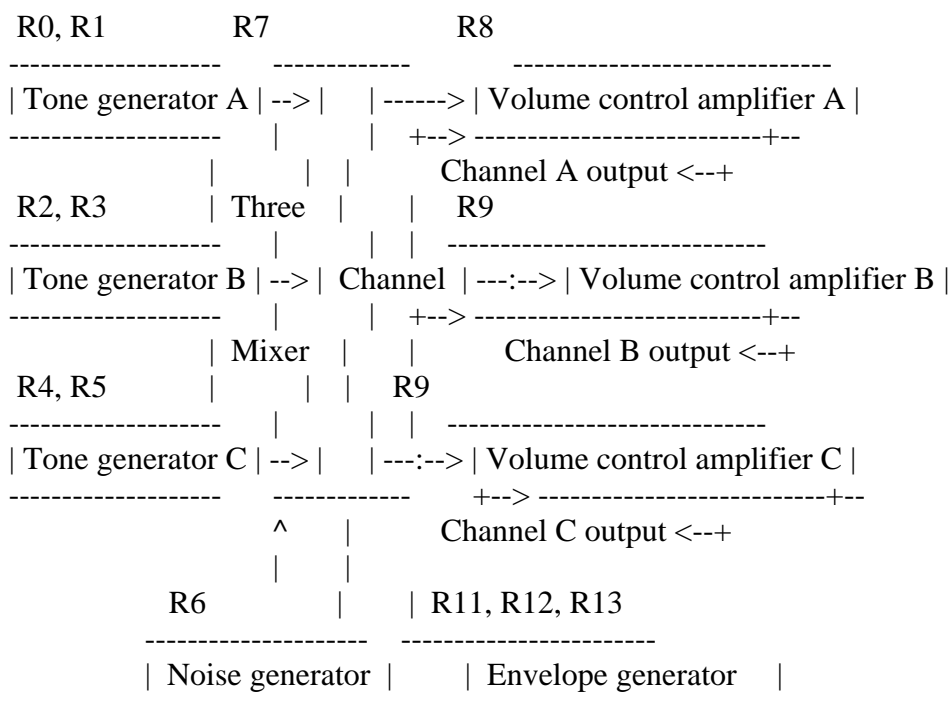
\* There are three tone generators, each of which can independently specify 4096 scales (equivalent to 8 octaves) and 16 volume levels.

\* It can generate piano and organ tones by using envelope patterns. Note that, since there is only one envelope generator, the tone of only one channel can be modified fundamentally.

\* With the noise generator inside, tones such as the wind or waves can easily be generated. Note that since there is only one noise generator, only one channel can generate the noise.

\* Any necessary frequency, such as the tone or the envelope, is obtained by dividing the input clock (in MSX, it is defined that  $f_c = 1.7897725$  MHz). So there is no unsteady pitch or rhythm.

Figure 5.1 PSG block diagram



The PSG has two additional I/O (input/output) ports used for other than tone generating functions, which are omitted in the block diagram above. MSX uses them as general-purpose I/O ports to connect to I/O devices such as joystick, a touch pad, a paddle, or a mouse. These general-purpose I/O ports are described in section 5.

\* PSG registers

Since the PSG generates tones, the CPU simply notifies PSG when the tone is to be changed. This is done by writing values in 16 8-bit registers inside the PSG as shown in Figure 5.2.

Roles and uses of these registers are described below.

\* Setting the tone frequency (R0 to R5)

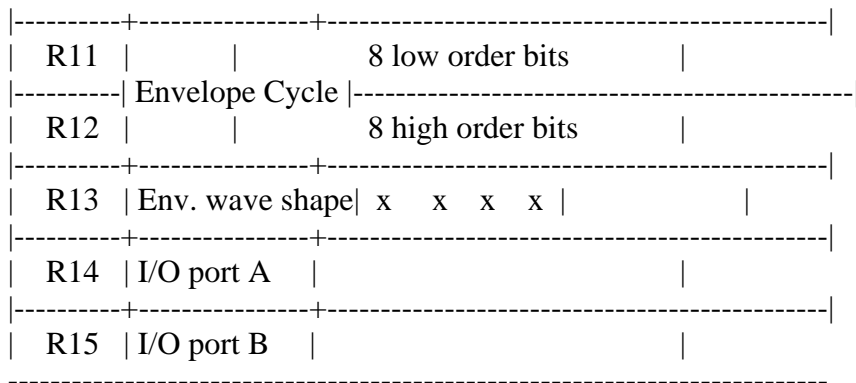
Each tone frequency of channel A, B, and C is set by R0 to R5. The input clock frequency ( $f_c = 1.7897725$  MHz) is divided by 16 and the result is the standard frequency. Each channel divides the standard frequency by the 12-bit data assigned for each, and the objective pitch is obtained. The following relation exists between 12-bit data (TP) and the tone frequency to be generated (ft).

$$\begin{aligned} ft &= f_c / (16 * TP) \\ &= 0.11186078125 / TP \text{ [MHz]} \\ &= 111860.78125 / TP \text{ [Hz]} \end{aligned}$$

A 12-bit data TP is specified for each channel by 4 high order bit coarse tune CT and 8 low order bit fine tune value FT, as shown in Figure 5.3. Table 5.1 shows the register settings to make the scales.

Figure 5.2 PSG register structure

Register	Bit	B7	B6	B5	B4	B3	B2	B1	B0
R0	Channel A note	8 low order bits							
R1	Dividing rate	x	x	x	x	4 high order bits			
R2	Channel B note	8 low order bits							
R3	Dividing rate	x	x	x	x	4 high order bits			
R4	Channel C note	8 low order bits							
R5	Dividing rate	x	x	x	x	4 high order bits			
R6	Noise div. rate	x	x	x					
R7	Enable*	IN*/OUT		NOISE*			TONE*		
		IOB	IOA	C	B	A	C	B	A
R8	Chan. A volume	x	x	x	M				
R9	Chan. B volume	x	x	x	M				
R10	Chan. C volume	x	x	x	M				



NOTE: x = unused bit  
 \* = inverted signal

Figure 5.3 Setting the pitch

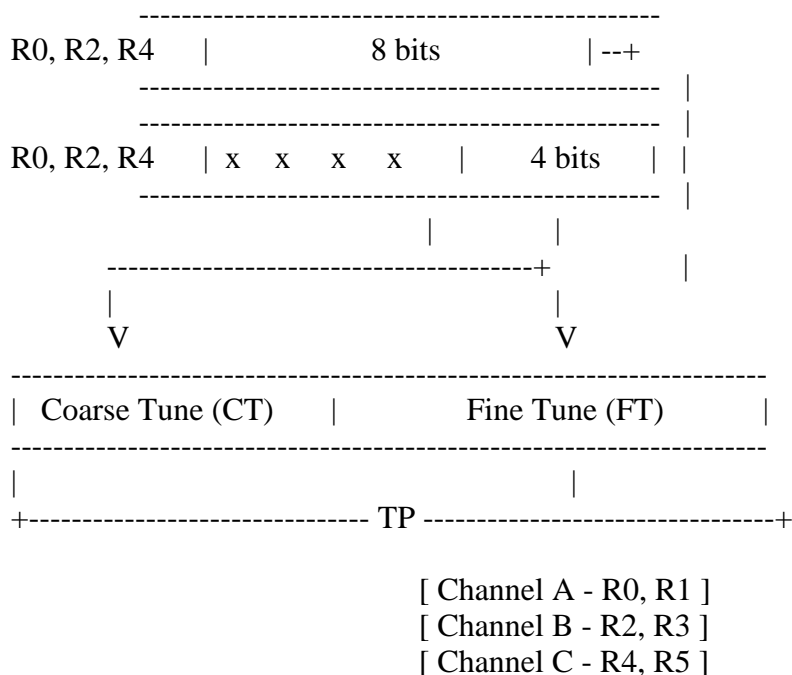


Table 5.1 Setting the tone frequency (scale data)

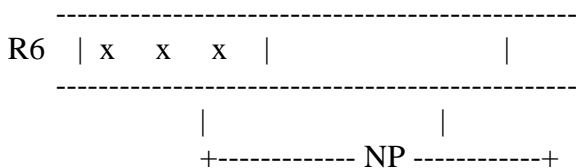
Octave	1	2	3	4	5	6	7	8
Note								
C	D5D	6AF	357	1AC	D6	6B	35	1B
C#	C9C	64E	327	194	CA	65	32	19
D	BE7	5F4	2FA	17D	BE	5F	30	18
D#	B3C	59E	2CF	168	84	5A	2D	16
E	A9B	54E	2A7	153	AA	55	2A	15
F	A02	501	281	140	A0	50	28	14

F#	973	4BA	25D	12E	97	4C	26	13
G	8EB	476	23B	11D	8F	47	24	12
G#	88B	436	21B	10D	87	43	22	11
A	7F2	3F9	1FD	FE	7F	40	20	10
A#	780	3C0	1E0	F0	78	3C	1E	F
B	714	38A	1C5	E3	71	39	1C	E

\* Setting the noise frequency (R6)

The noise generator is used for synthesizing explosion sounds or wave sounds. The PSG can send the noise output by the noise generator to channels A to C. Since there is only one noise generator, the same noise is sent to all channels. By changing the average frequency, various noise effects can be obtained and this is done by R6 register settings. The 5 low order bit data (NP) of this register is divides into the standard frequency (fc/16) and this determines the average frequency of the noise (fn).

Figure 5.4 Setting the noise frequency



The following relation exists between NP and fn.

$$\begin{aligned}
 fn &= fc / (16 * NP) \\
 &= 0.11186078125 / NP \text{ [MHz]} \\
 &= 111860.78125 / NP \text{ [Hz]}
 \end{aligned}$$

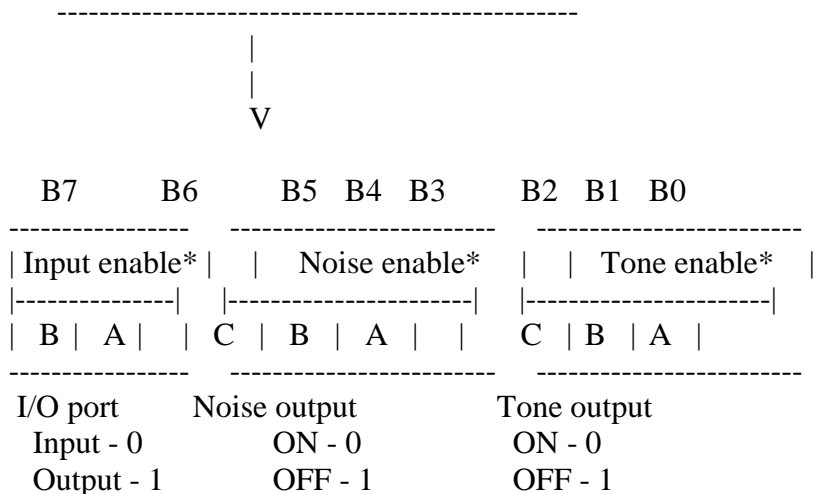
Since the value of NP is from 1 to 31, the average frequency of the noise can be set from 3.6kHz to 111.9kHz.

\* Mixing the sound (R7)

R7 is used to select the output of the tone and noise generator, or a mixture of both. As shown in Figure 5.5, the 3 low order bits (B0 to B2) of R7 control the tone output and the next 3 bits (B3 to B5) control the noise output. In both cases, when the corresponding bit is 0, the output is ON and, when 1, it is OFF.

Figure 5.5 Output selection for each channel



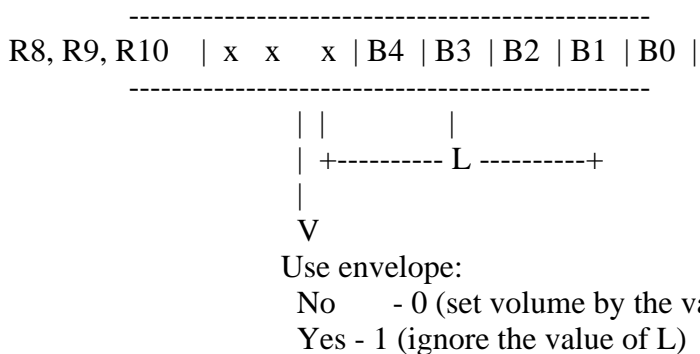


The 2 high order bits of R7 do not affect sound output. These are used to determine the direction of the data of two I/O ports which PSG has. When the corresponding bit is 0, the input mode is selected and, when 1, the output mode is selected. In MSX, port A is used for the input and port B for the output, so it should always be set so that bit 6 = "0" and bit 7 = "1".

\* Setting the volume (R8 to R10)

R8 to R10 are used to specify the volume of each channel. Two ways can be selected by these registers: specifying the fixed volume by 4-bit data (0 to 15) and generating sound effects such as vibrato or fade-out by using the envelope.

Figure 5.6 Setting the volume

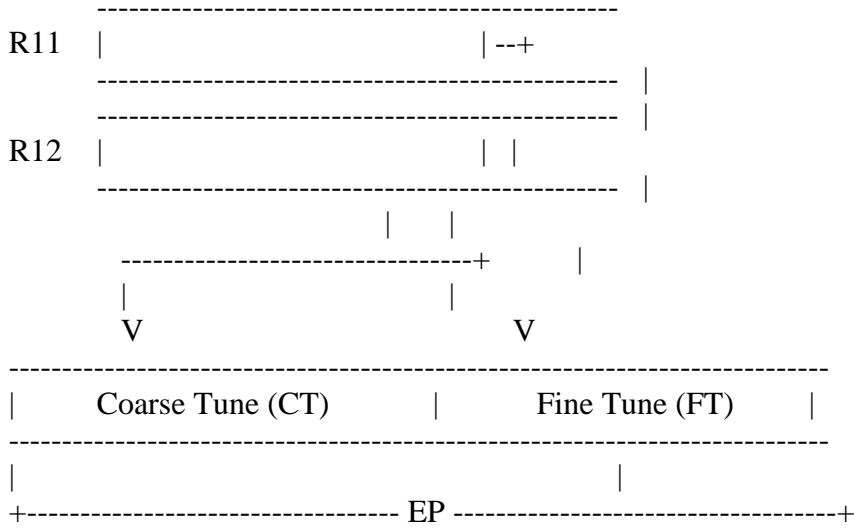


When bit 4 of these registers is "0", the envelope is not used and the 4 low order bit value L (0 to 15) of the registers specify the volume. When bit 4 is "1", the volume depends on the envelope signals and the value L is ignored.

\* Setting the envelope cycle (R11, R12)

R11 and R12 specify the envelope cycle in 16-bit data. The 8 high order bits are set in R12 and the 8 low order bits are set in R11.

Figure 5.7 Setting the envelope cycle



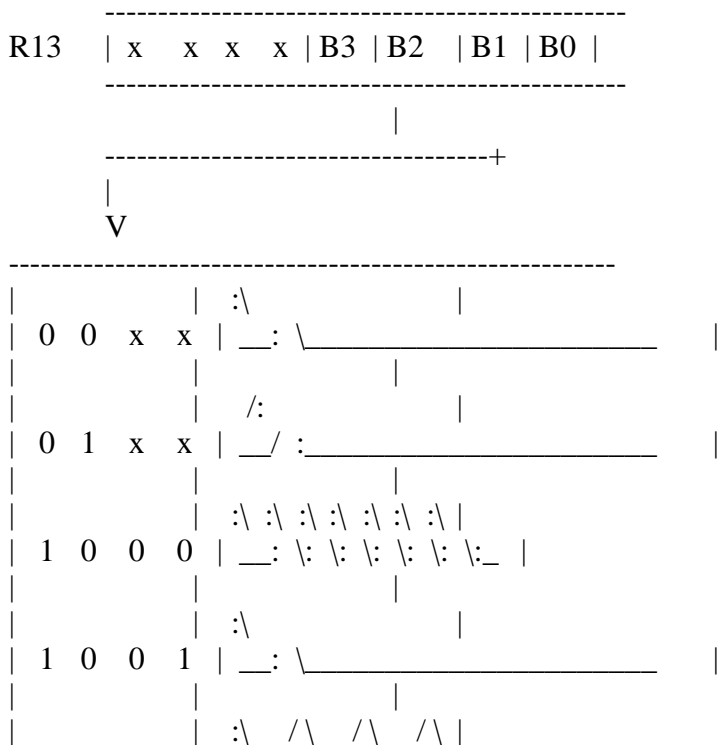
The following relation exists between the envelope cycle T and 16-bit data EP.

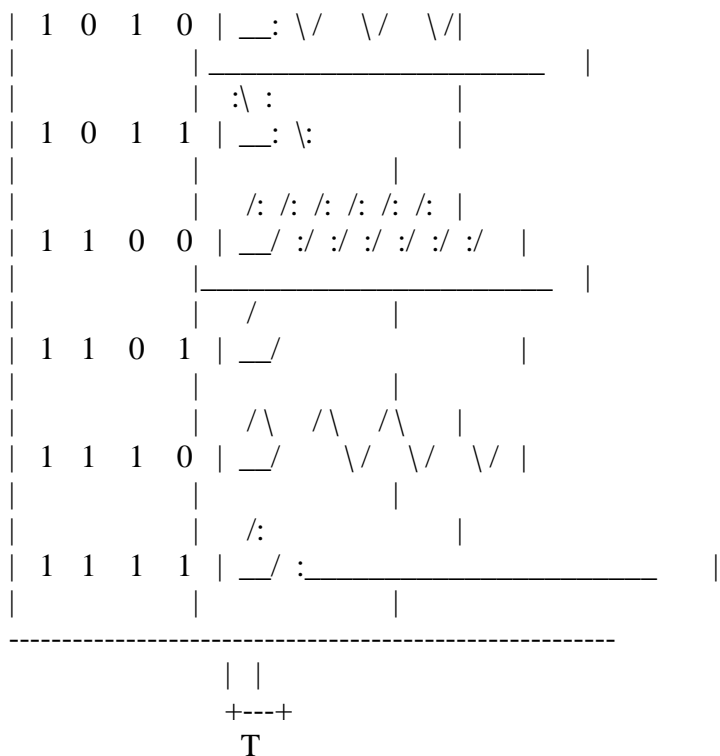
$$\begin{aligned}
 T &= (256 * EP) / f_c \\
 &= (256 * EP) / 1.787725 \text{ [MHz]} \\
 &= 143.03493 * EP \text{ [micro second]}
 \end{aligned}$$

\* Setting the envelope pattern (R13)

R13 sets the envelope pattern by the 4 low order bit data as shown in Figure 5.8. The intervals of T specified in the figure correspond to the envelope cycle specified by R11 and R12.

Figure 5.8 Setting the wave forms of the envelopes





\* I/O port (R14, R15)

R14 and R15 are the ports to send and receive 8-bit data in parallel. MSX uses these as the general-purpose I/O interface. For more information, see section 5.

## 1.2 Access to the PSG

For access the PSG from assembly language programs, several BIOS routines described below are available.

\* GICINI (0090H/MAIN) ..... PSG initialization

Input: ---

Output: ---

Function: initializes PSG registers and does the initial settings of the work area in which PLAY statement of BASIC is executed. Each register of PSG is set to the value as shown in Figure 5.9.

Figure 5.9 Initial values of PSG registers

		Bit							
		7	6	5	4	3	2	1	0
Register									
R0	Channel A	0	1	0	1	0	1	0	1
R1	frequency	0	0	0	0	0	0	0	0

R2	Channel B	0	0	0	0	0	0	0	0
R3	frequency	0	0	0	0	0	0	0	0
R4	Channel C	0	0	0	0	0	0	0	0
R5	frequency	0	0	0	0	0	0	0	0
R6	Noise frequency	0	0	0	0	0	0	0	0
R7	Channel setting	1	0	1	1	1	0	0	0
R8	Chan. A volume	0	0	0	0	0	0	0	0
R9	Chan. B volume	0	0	0	0	0	0	0	0
R10	Chan. C volume	0	0	0	0	0	0	0	0
R11	Envelope Cycle	0	0	0	0	1	0	1	1
R12		0	0	0	0	0	0	0	0
R13	Env. pattern	0	0	0	0	0	0	0	0
R14	I/O port A								
R15	I/O port B								

\* WRTPSG (0093H/MAIN) ..... writing data in PSG registers

Input: A <-- PSG register number  
E <-- data to be written

Output: ---

Function: writes the contents of the E register in the PSG register whose number is specified by the A register.

\* RDPSG (0096H/MAIN) ..... reading PSG register data

Input: A <-- PSG register number

Output: A <-- contents of the specified register

Function: reads the contents of PSG register whose number is specified by the A register and stores the value in the A register.

\* STRTMS (0099H/MAIN) ..... starting the music

Input: (QUEUE) <-- MML which is translated into the intermediate language

Output: ---

Function: examines whether the music is played as the background task, and plays the music which is set in the queue, if the music has not yet been played.



## List 5.1 Single tone generation

```

=====
;*****
;
; List 5.1 440 Hz tone
;
;*****
;
;
WRTPSG EQU 0093H

ORG 0B000H

;----- program start -----

LD A,7 ;Select Channel
LD E,00111110B ;Channel A Tone := On
CALL WRTPSG

LD A,8 ;Set Volume
LD E,10
CALL WRTPSG

LD A,0 ;Set Fine Tune Channel A
LD E,0FEH ;Data 0FEH
CALL WRTPSG

LD A,1 ;Set Coarse Tune Channel A
LD E,0 ;Data 0H
CALL WRTPSG

RET

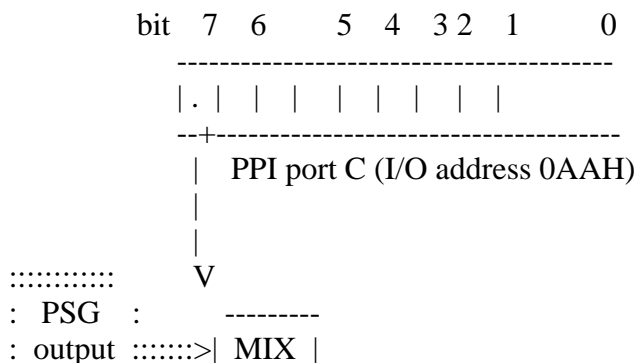
END
=====

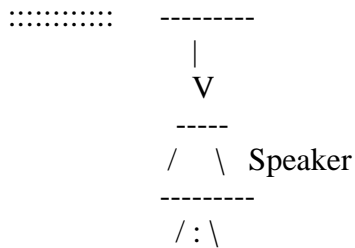
```

### 1.3 Tone Generation by 1-bit Sound Port

MSX has another sound generator in addition to the PSG. This is a simple one that generates sound by turning ON/OFF the 1-bit I/O port output repeatedly using software.

Figure 5.10 1-bit sound port





#### 1.4 Access to 1-bit Sound Port

To access to the 1-bit sound port, the following BIOS routine is offered.

##### \* CHGSND (0135H/MAIN)

Input: A <-- specification of ON/OFF (0 = OFF, others = ON)

Output:

Function: calling this routine with setting 0 in the A register turns the bit of the sound port OFF; calling it with another value turns it ON.

#### List 5.2 Reading from cassette tape

```

=====
;*****
;
;
; List 5.2  Read from cassette tape
;
;      Set music tape into tape-recorder
;      and run this program.
;      Then your MSX will replay it.
;
;*****
;
;
CHGSNG  EQU    0135H
STMOTR  EQU    00F3H
RDPSG  EQU    0096H
BREAKX  EQU    00B7H

```

```

ORG    0B000H

```

```

;----- program start ----- Note: Play tape using 1-bit sound port.

```

```

START: LD  A,1      ;motor on
      CALL STMOTR

LBL01: LD  A,14     ;register 14
      CALL RDPSG    ;read PSG

      AND  80H      ;check CSAR
      CALL CHGSNG   ;change SOUND PORT

      CALL BREAKX   ;check Ctrl-STOP
      JR  NC,LBL01

```

```

XOR    A          ;stop cassette motor
CALL   STMOTR
RET

END

```

---

## 2. CASSETTE INTERFACE

Cassette tape recorders are the least expensive external storage devices available for the MSX. Knowledge of the cassette interface is required to treat information in cassette tapes within assembly language programs. This section offers the necessary information.

### 2.1 Baud Rate

The following two baud rates can be used by the MSX cassette interface (see Table 5.2). When BASIC is invoked, 1200bps is set by default.

Table 5.2 MSX baud rate

Baud rate	Characteristics
1200 bps	Low speed / high reliability
2400 bps	High speed / low reliability

The baud rate is specified by the fourth parameter of the SCREEN instruction or the second parameter of the CSAVE instruction. Once the baud rate is set, it stays at that value.

```

SCREEN    ,,<baud rate>
CSAVE     "filename",<baud rate>
          (<baud rate> is 1 for 1200bps, 2 for 2400 bps)

```

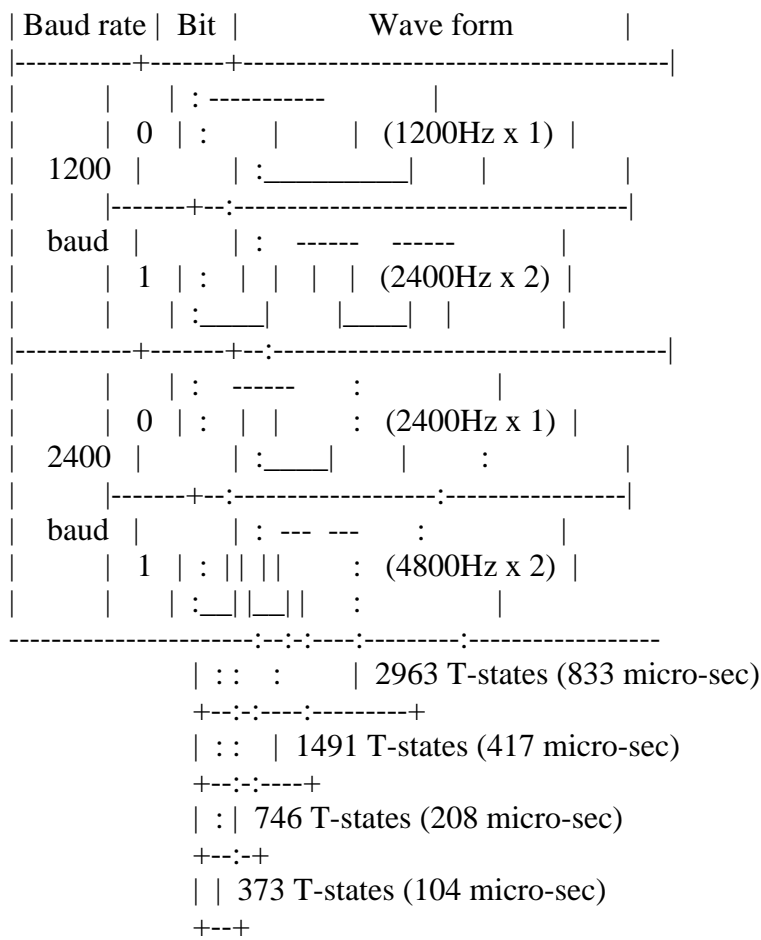
### 2.2 One bit composition

One bit data, the basis of I/O, is recorded as shown in Figure 5.11. The pulse width is determined by counting the T-STATE of the CPU, so, while the cassette interface is active, any interrupt is inhibited.

The bit data from the cassette can be read through the seventh bit of port B of the general-purpose I/O interface (register 15 of the PSG). This function was used in the program example of List 5.3, section 1 of chapter 5.

Figure 5.11 One bit composition

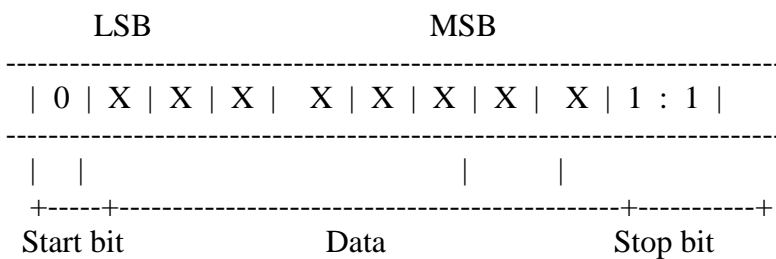
---



### 2.3 One byte composition

One byte data is recorded in the array of bits as shown in Figure 5.12. There is one "0" bit as the start bit, followed by the 8-bit data body from LSB to MSB and by two "1" bit as the stop bits, so 11 bits are used.

Figure 5.12 One byte composition



### 2.4 Header Composition

The header is the portion where the signal of the specific frequency is recorded on the tape for a certain period. This allows the cassette tape speed to stabilize after it is started, or divides two files. There is a long header and a short header. The long header is used to wait until the motor is stabilized. The baud rate at reading the tape is determined by reading the long header. The short header is used to divide file bodies. Table 5.3 shows the compositions of both.

Table 5.3 Header composition

Baud rate	Header	Header composition
1200 baud	Long header	2400 Hz x 16000 (about 6.7 sec)
	Short header	2400 Hz x 4000 (about 1.7 sec)
2400 baud	Long header	4800 Hz x 32000 (about 6.7 sec)
	Short header	4800 Hz x 8000 (about 1.7 sec)

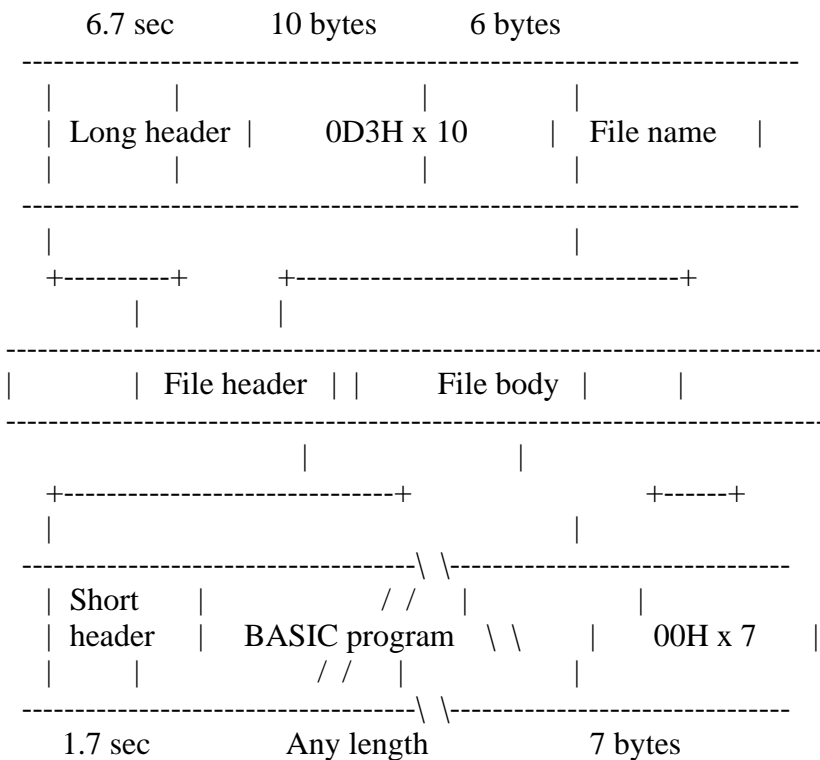
## 2.5 File Formats

MSX BASIC supports the following three kinds of cassette format files.

### (1) BASIC text file

BASIC programs saved with the CSAVE command are recorded in this format. The file is divided into the preceding file header and the succeeding the body.

Figure 5.13 Binary file format

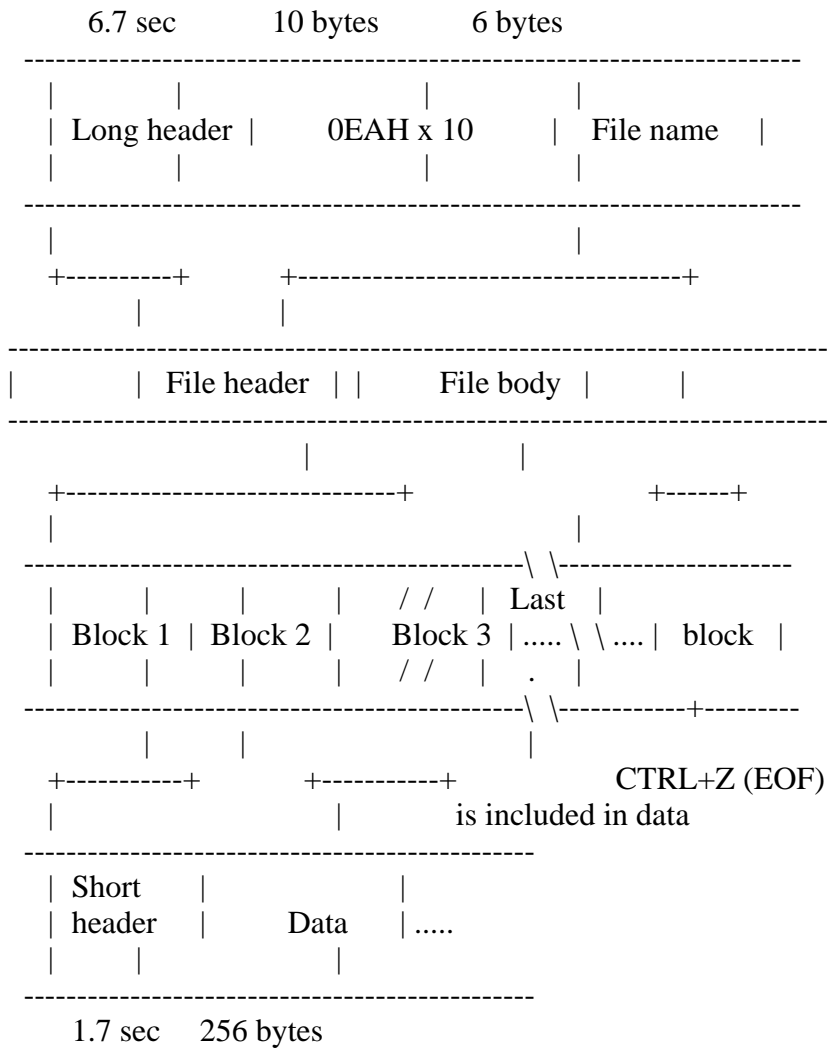


In the file header, ten bytes each of the value 0D3H follow after the long header and six bytes containing the file name are placed after them. In the file body, program body follows the short header and the end of the file is indicated by seven bytes of 00H.

(2) ASCII text file

BASIC programs saved in ASCII format by the SAVE command and data files created by the OPEN command are recorded in this format.

Figure 5.14 ASCII file format



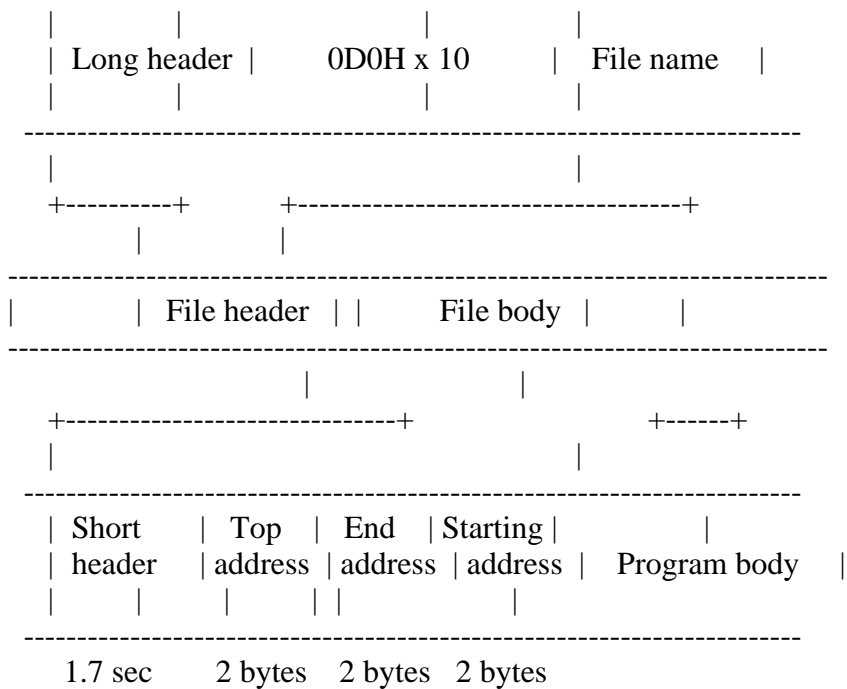
(3) Machine code file

Machine code files saved by the BSAVE command are recorded in the following format. In the file header, 10 bytes each of the value 0D0H follow after the long header and 6 bytes containing the file name are placed after them.

In the file body, the starting address, the end address, and the entry address are recorded in order after the short header, and the machine codes follow after them. Since the amount of data can be calculated from the starting and ending addresses, there is no special mark for the end of the file. The entry address is the address where the program is executed when the R option of the BLOAD command is used.

Figure 5.15 Machine code file format





## 2.6 Access to cassette files

The following BIOS routines are offered to access cassette files.

\* TAPION (00E1H/MAIN) ..... OPEN for read

Input: ---

Output: CY flag = ON at abnormal terminations

Function: starts the motor of the tape recorder and reads the long header or the short headet. At the same time, the baud rate in which the file is recorded is detected and the work area is set according to it. Interrupts are inhibited.

\* TAPIN (00E4H/MAIN) ..... read one byte

Input: ---

Output: A <-- data which has been read  
CY flag = ON at abnormal terminations

Function: reads one byte of data from the tape and stores it in the A register.

\* TAPIOF (00E7H/MAIN) ..... CLOSE for read

Input: ---

Output: ---

Function: ends reading from the tape. At this point, interrupts are allowed.

\* TAPOON (00EAH/MAIN) ..... OPEN for write

Input: A <-- type of header (0 = short header, others = long header)

Output: CY flag = ON at abnormal terminations

Function: starts the motor of the tape recorder and writes the header of the type specified in the A register to the tape.  
Interrupts are inhibited.

\* TAPOUT (00EDH/MAIN) ..... write one byte

Input: A <-- data to be written  
Output: CY flag = ON at abnormal terminations  
Function: writes the contents of the A register to the tape.

\* TAPOOF (00F0H/MAIN) ..... CLOSE writing

Input: ---  
Output: ---  
Function: ends writing the tape. At this point, interrupts are allowed.

\* STMOTR (00F3H/MAIN) ..... specify the actions of the motor

Input: A <-- action (0 = stop, 1 = start, 255 = reverse the current status)  
Output: ---  
Function: sets the status of the motor according to the value specified in the A register.

When READ/WRITE routines for the cassette files are created using these BIOS calls, only READ or WRITE, without any other action, should be done. For example, reading data from the tape and displaying it on the CRT might cause a READ error.

List 5.3 is a sample program which uses BIOS routines.

List 5.3 Listing names of files saved in the cassette

=====

```
*****  
;  
;  
; List 5.3 Cassette files  
;  
; Set cassette tape into recorder and run this program.  
; Then all the names and attributes of the programs  
; in that tape will be listed.  
;  
*****  
;  
CHPUT EQU 00A2H  
TAPION EQU 00E1H  
TAPIN EQU 00E4H  
TAPIOF EQU 00E7H  
  
ORG 0C000H  
  
;----- program start ----- Note: View program names on cassette tape.
```



START: CALL TAPION ;motor on and read header

```
LD B,16
LD HL,WORK ;work area address
LBL01: PUSH HL
PUSH BC
CALL TAPIN ;read a byte of data from tape
POP BC
POP HL
JR C,ERROR ;set carry flag if read error
LD (HL),A
INC HL
DJNZ LBL01
```

```
LD HL,FILNAM ;write file name
CALL PUTSTR
LD HL,WORK+10
CALL PUTSTR
CALL CRLF
```

```
LD A,(WORK) ;check file attributes
```

```
LD HL,BINFIL
CP 0D3H ;check binary file
JR Z,LBL03
```

```
LD HL,ASCFIL
CP 0EAH ;check ascii file
JR Z,LBL03
```

```
LD HL,MACFIL
CP 0D0H ;check machine code file
JR Z,LBL03
```

ERROR: LD HL,ERRSTR

```
LBL03: CALL PUTSTR
CALL TAPIOF
RET
```

;----- put CRLF -----

```
CRLF: LD HL,STCRLF
CALL PUTSTR
RET
```

;----- put string -----

```
PUTSTR: LD A,(HL) ;get a character from strings
CP '$' ;check end of strings
RETZ
CALL CHPUT ;write a character to CRT
INC HL
JR PUTSTR
```

;----- strings data -----

```

FILNAM: DB 'FILE NAME :$'
ASCFIL: DB 'ASCII FILE',0DH,0AH,$'
BINFIL: DB 'BINARY FILE',0DH,0AH,$'
MACFIL: DB 'BSAVE FILE',0DH,0AH,$'
ERRSTR: DB 'TAPE READ ERROR',0DH,0AH,$'
STCRLF: DB 0DH,0AH,$'

```

;----- WORK AREA -----

```

WORK: DS 16,0
      DB '$' ;end of strings

      END

```

### 3. KEYBOARD INTERFACE

Although the MSX2 keyboard has the same design as that of the MSX1, it is more convenient to use because of the Romand-to-kana translation available for kana input. This chapter describes the keyboard interface of the MSX2.

Descriptions of the key arrangement are based on the Japanese keyboard standard; note that data is slightly different for the international MSX versions.

#### 3.1 Key Scanning

MSX uses the key matrices as shown in Figure 5.16, Figure 5.17 and Figure 5.17B. The key status can be obtained in real time by examining this key matrix and is available for reading input.

Scanning the key matrix is done by the following BIOS routine.

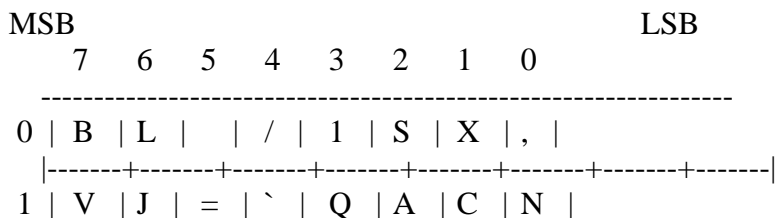
\* SNSMAT (0141H/MAIN) ..... reads the specified line of the key matrix

Input: A <-- key matrix line to be read (0 to 10)

Output: A <-- status of the specified line of the key matrix  
(when pressed, the bit of the key is 0)

Function: specifies a line of the key matrix shown in Figure 5.16, Figure 5.17 or Figure 5.17B and stores its status in the A register. The bit corresponding with the key being pressed is "0", and "1" for the key not being pressed.

Figure 5.16 MSX USA version key matrix



```

-----|
2 | G | 8 | 0 | ] | W | F | Z | M |
-----|
3 | T | I | ~ | ; | 2 | D | U | \ |
-----|
4 | 6 | K | P | ' | 3 | R | 7 | H |
-----|
5 | 5 | 0 | 9 | [ | 4 | E | Y | . |
-----|
6 | F3 | F2 | F1 | CODE | CAPS | GRAPH | CTRL | SHIFT |
-----|
7 | RETURN | SELECT | BS | STOP | TAB | ESC | F5 | F4 |
-----|
8 | RIGHT | DOWN | UP | LEFT | DEL | INS | HOME | SPACE |
-----|

```

[TEN KEY]

```

-----|
9 | 4 | 3 | 2 | 1 | 0 | option | option | option |
-----|
10 | . | , | - | 9 | 8 | 7 | 6 | 5 |
-----|

```

Figure 5.17 MSX International version key matrix

```

MSB                                     LSB
  7   6   5   4   3   2   1   0
-----|
0 | B | L | deadkey | / | 1 | S | X | , |
-----|
1 | V | J | ^ | ] | Q | A | C | N |
-----|
2 | G | 8 | 0 | [ | W | F | Z | M |
-----|
3 | T | I | ~ | ; | 2 | D | U | \ |
-----|
4 | 6 | K | P | : | 3 | R | 7 | H |
-----|
5 | 5 | 0 | 9 | @ | 4 | E | Y | . |
-----|
6 | F3 | F2 | F1 | CODE | CAPS | GRAPH | CTRL | SHIFT |
-----|
7 | RETURN | SELECT | BS | STOP | TAB | ESC | F5 | F4 |
-----|
8 | RIGHT | DOWN | UP | LEFT | DEL | INS | HOME | SPACE |
-----|

```

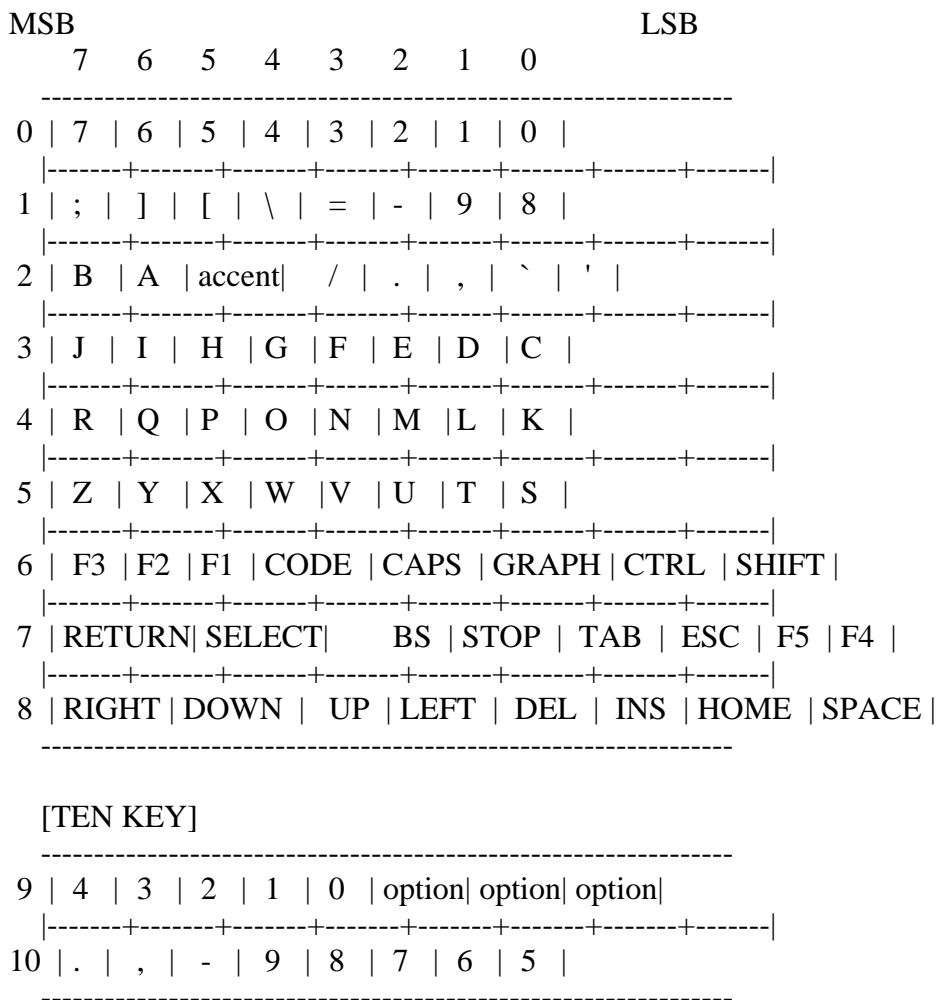
[TEN KEY]

```

-----|
9 | 4 | 3 | 2 | 1 | 0 | option | option | option |
-----|
10 | . | , | - | 9 | 8 | 7 | 6 | 5 |
-----|

```

Figure 5.17B MSX European version key matrix



List 5.4 Use of the key scanning routine

```

;*****
;
;
; List 5.4 scan key-matrix and display it
;
;*****
;
CHPUT EQU 00A2H
BREAKX EQU 00B7H
POSIT EQU 00C6H
SNSMAT EQU 0141H

ORG 0B000H

;----- program start ----- Note: read key matrix and display key
; pattern.

SCAN: LD C,0 ;C := line of key matrix

SC1: LD A,C
CALL SNSMAT ;Read key matrix

LD B,8
LD HL,BUF ;HL := buffer address
SC2: LD D,!'
```

```

        RLA                ;Check bit
        JR  C,SC3
        LD  D,'#'

SC3:    LD  (HL),D        ;store '.' or '#' to buffer
        INC HL
        DJNZ SC2

        LD  H,05H        ;x := 5
        LD  L,C          ;y := C+1
        INC L
        CALL POSIT      ;set cursor position

        LD  B,8          ;put out bit patterns to CRT
        LD  HL,BUF

SC4:    LD  A,(HL)
        CALL CHPUT
        INC HL
        DJNZ SC4

        CALL BREAKX     ;check Ctrl-STOP
        RETC

        INC C           ;line No. increment
        LD  A,C
        CP  09
        JR  NZ,SC1
        JR  SCAN

;----- work area -----

BUF:    DS  8

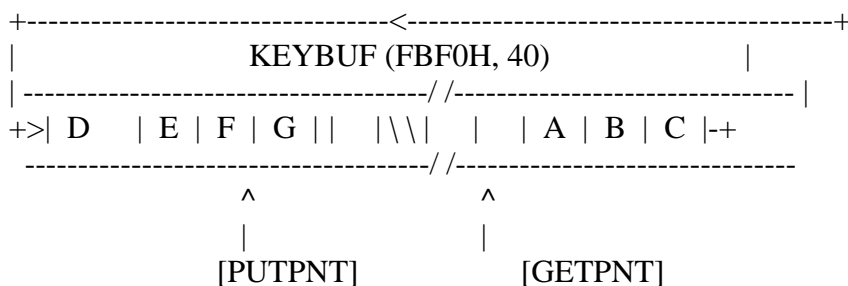
        END

```

### 3.2 Character Input

MSX scans the key matrix every 1/60 second using the timer interrupt and, when a key is pressed, stores the character code in the keyboard buffer as shown in Figure 5.18. Key input to MSX is generally done by reading this keyboard buffer.

Figure 5.18 Keyboard ring buffer



GETPNT (F3FAH, 2) points to the next character to be obtained in CHGET routine.

PUTPNT (F3F8H, 2) points to the next location for the character to be put when the keyboard is pressed next time.

BIOS routines having functions for key input using this keyboard buffer and functions related to it are described below. Inhibiting the timer interrupt renders them useless, of course.

\* CHSNS (009CH/MAIN) ..... checks the keyboard buffer

Input: ---

Output: Z flag = ON when the buffer is empty

Function: examines whether any characters remain in the keyboard buffer and sets the Z flag when the buffer is empty.

\* CHGET (009FH/MAIN) ..... one character input from the keyboard buffer

Input: ---

Output: A <-- character code

Function: reads one character from the keyboard buffer and stores it in the A register. When the buffer is empty, it displays the cursor and waits for a key input. While a key input is waited for, the CAP lock, KANA lock, and Roman-to-kana translation lock are valid. The related work area is listed below. In the list, since SCNCNT and REPCNT are initialised after the execution of CHGET routine, this area should be set at each CHGET call to change the interval of the auto-repeat.

Work area

CLIKSW (F3DBH, 1) key click sound (0 = OFF, others = ON)

SCNCNT (F3F6H, 1) key scanning interval (1, normally)

REPCNT (F3F7H, 1) delay until beginning auto-repeat  
(50, normally)

CSTYLE (FCAAH, 1) figure of the cursor  
(0 = block, others = underline)

CAPST (FCABH, 1) CAPS lock (0 = OFF, others = ON)

DEADST (FCACH, 1) dead key lock  
0 = on preceding dead key  
1 = dead key  
2 = shifted dead key  
3 = code dead key  
4 = code shift dead key

\* KILBUF (0156H/MAIN) ..... empty the keyboard buffer

Input: ---

Output: ---

Function: empties the keyboard buffer.

## List 5.5 Use of one character input routine

```

=====
;*****
;
; List 5.5    get key code
;
;           this routine doesn't wait for key hit
;
;*****
;
CHSNS EQU    009CH    ;check keyboard buffer
CHGET EQU    009FH    ;get a character from buffer
CHPUT EQU    00A2H    ;put a character to screen
BREAKX EQU    00B7H    ;check Ctrl-STOP
KILBUF EQU    0156H    ;clear keyboard buffer
REPCNT EQU    0F3F7H    ;time interval until key-repeat
KEYBUF EQU    0FBF0H    ;keyboard buffer address

    ORG    0B000H

;----- program start ----- Note: Real-time input using CHGET

KEY:  CALL  CHSNS    ;check keyboard buffer
      JR   C,KEY1

      LD  A,1
      LD  (REPCNT),A ;not to wait until repeat
      CALL CHGET    ;get a character (if exists)
      JR  KEY2

KEY1: LD  A,'-'    ;A := '-'

KEY2: CALL  CHPUT    ;put the character
      CALL  KILBUF   ;clear keyboard buffer
      CALL  BREAKX   ;check Ctrl-STOP
      JR   NC,KEY

      END
=====

```

\* CNVRCHR (00AB/MAIN) ..... graphic character operation

Input: A <-- character code

Output: A <-- translated graphic character

(normal characters are not translated)

CY flag = OFF (input was the graphic header byte 01H)

CY flag = ON, Z flag = ON (input was the graphic character  
and was translated)

CY flag = ON, Z flag = OFF (input was the normal character  
and was not translated)

Function: executing CNVCHR after CHGET causes the graphic character  
to be translated to one byte code as shown in Figure 5.19  
and causes other character not to be translated and to be  
returned. Since the graphic character is represented by

irregular 2-byte code with the graphic header byte (01H), annoying procedures are required for the character operations; this routine makes it somewhat easy.

Figure 5.19 Graphic character translation chart

Before conversion	After conversion	Before conversion	After conversion
		0150H --> 50H	
0141H --> 41H		0151H --> 51H	
0142H --> 42H		0152H --> 52H	
0143H --> 43H		0153H --> 53H	
0144H --> 44H		0154H --> 54H	
0145H --> 45H		0155H --> 55H	
0146H --> 46H		0156H --> 56H	
0147H --> 47H		0157H --> 57H	
0148H --> 48H		0158H --> 58H	
0149H --> 49H		0159H --> 59H	
014AH --> 4AH		015AH --> 5AH	
014BH --> 4BH		015BH --> 5BH	
014CH --> 4CH		015CH --> 5CH	
014DH --> 4DH		015DH --> 5DH	
014EH --> 4EH		015EH --> 5EH	
014FH --> 4FH		015FH --> 5FH	

\* PINLIN (00AEH/MAIN) ..... one line input

Input: ---

Output: HL <-- F55DH

[F55EH] <-- input string (the end of the line is represented by 00H)

CY flag <-- terminated by STOP=ON, terminated by RETURN=OFF

function: stores input string in the line buffer BUF (F55EH). All functions of the screen editing are available at the string input. Pressing RETURN or STOP causes the input to be finished. The work area is listed below.

Work area

BUF (F55EH, 258) the line buffer where the string is stored

LINTTB (FBB2H, 24) 00H when the one physical line is the succession of the line above

\* INLIN (00B1H/MAIN) ..... one line input (prompt available)

Input: ---

Output: same as PINLIN

Function: stores input string in the line buffer BUF (F55EH), as PINLIN routine. Note that the portion before the cursor location at the time when the routine begins to execute is not received. List 5.6 shows the difference between PINLIN and INLIN.



## List 5.6 Difference between INLIN and PINLIN

```
=====
;*****
;
; List 5.6  INLIN and PINLIN
;
;*****
;
CHPUT EQU    00A2H
INLIN EQU    00B1H
PINLIN EQU   00AEH
KILBUF EQU   0156H

BUFEQU    F55EH

    ORG    0B000H

;----- program start -----

    LD HL,PRMPT1
    CALL PUTMSG          ;put prompt message
    CALL INLIN          ;use INLIN routine
    LD HL,BUF
    CALL PUTMSG

    LD HL,PRMPT2
    CALL PUTMSG          ;put prompt message
    CALL PINLIN         ;use PINLIN routine
    LD HL,BUF
    CALL PUTMSG

    RET

;----- put a string -----

PUTMSG: LDA,(HL)
        CP '$'
        RETZ
        CALL CHPUT
        INC HL
        JR PUTMSG

;----- string data -----

PRMPT1: DB 0DH,0AH,'INLIN:$'
PRMPT2: DB 0DH,0AH,'PINLIN:$'

    END
=====
```

### 3.3 Function Keys

MSX has ten function keys, which can be defined by the user at will. A 16

byte work area is allocated for the definition of each key. The following list shows their addresses.

FNKSTR (F87FH, 16) ..... F1 key definition address  
+ 10H (F88FH, 16) ..... F2 key definition address  
+ 20H (F89FH, 16) ..... F3 key definition address  
+ 30H (F8AFH, 16) ..... F4 key definition address  
+ 40H (F8BFH, 16) ..... F5 key definition address  
+ 50H (F8CFH, 16) ..... F6 key definition address  
+ 60H (F8DFH, 16) ..... F7 key definition address  
+ 70H (F8EFH, 16) ..... F8 key definition address  
+ 80H (F8FFH, 16) ..... F9 key definition address  
+ 90H (F90FH, 16) ..... F10 key definition address

Pressing a function key causes the string defined in that key to be stored in [KEYBUF]. The end of the string is indicated by 00H and a maximum of 15 keystrokes can be defined for one function key (definitions longer than 16 keystrokes are defined over more than one function key definition area). To restore the initial settings of the function keys, use the following BIOS routine.

\* INIFNK (003EH/MAIN) ..... initialize function keys

Input: ---

Output: ---

Function: restores the function key definition to the setting when BASIC starts.

### 3.4 STOP Key During Interrupts

CHGET, the one-character input routine described in 3.3, determines the pressed key in the timer interrupt routine. Thus, when the timer interrupt is inhibited, such as during cassette data I/O, pressed keys cannot be detected. By using the BIOS routine described below, the CTRL key + STOP key combination can be detected even when interrupts are inhibited.

\* BREAKX (00B7H/MAIN) ..... CTRL + STOP detection

Input: ---

Output: CY flag = ON, when CTRL + STOP is pressed

Function: scans keys and decides whether CTRL key and STOP key are pressed at the same time. When both are pressed, this routine sets "1" to the CY flag and returns. Otherwise, it resets "0" to the CY flag and returns. This routine is available while interrupts are inhibited.

## 4. PRINTER INTERFACE

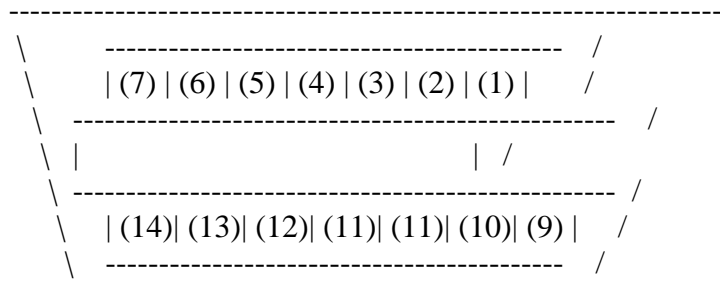
This section describes how to access the MSX printer interface from assembly language. The information described here is helpful if the printer is going to be used to print bit image graphics.

## 4.1 Print Interface Overview

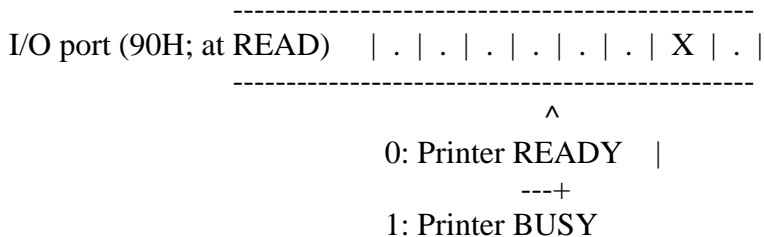
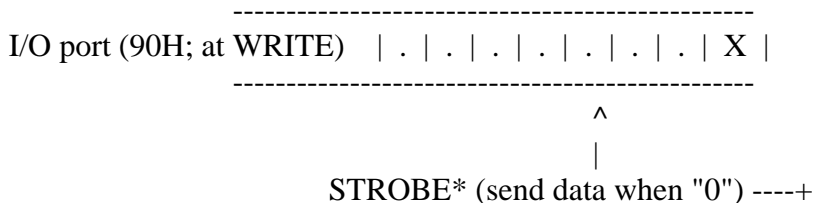
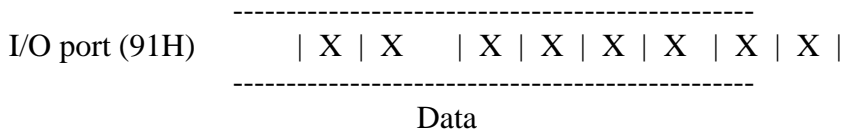
The printer interface is supported by BIOS and BASIC. MSX drives the printer through an 8-bit parallel output port and uses a handshaking method with BUSY and STROBE signals. The standard connector is also defined (Amphenol 14-pin, female side to the machine). Figure 5.20 shows the signal lines.

Figure 5.20 Printer interface

### Printer interface pin connections



- (1) ..... STROBE\*
- (2) to (9) .... Data (b0 to b7)
- (11) ..... BUSY
- (14) ..... BGND



## 4.2 Output to the MSX Standard Printer

If data is sent from MSX to the printer, the action depends on whether the printer receiving the data is of the MSX standard. The use of MSX standard printers is described in this section. Descriptions about other printers are in the next section.

An MSX standard printer can print any character that can be displayed on the screen. Special graphic characters corresponding to character codes  $n = 01H$

to 1FH can be also printed by sending the code 40H + n after the graphic character header (01H). In addition to these, the control codes shown in Table 5.4 can be used with MSX standard printers (see the manual of the printer for controlling a printer which has other functions such as printing Chinese characters).

To feed lines in MSX standard printers, send 0DH and 0AH successively. To print the bit image, send nnnn bytes data, where nnnn means four decimal figures, after the escape sequence ESC + "Snnnn". Note that, MSX has a function to transform the tab code (09H) to the adequate number of space codes (20H) for printers not having a tab function. This transformation is normally done. To print a bit image which includes the value 09H correctly, change the following work area.

\* RAWPRT (F418H, 1) ..... replaces a tab by spaces when the contents are 00H, otherwise not.

Table 5.4 Control codes of the printer

code	function
0AH	line feed
0CH	form feed
0DH	carriage return
ESC + "A"	normal line spacing (spaces between lines; characters are read easily)
ESC + "B"	line spacing for graphics (no space between lines)
ESC + "Snnnn"	bit image printing

### 4.3 Access to the printer

To send output to the printer, the following BIOS routines are offered.

#### \* LPTOUT (00A5H/MAIN)

Input: A register <-- character code  
 Output: CY flag = ON at abnormal termination  
 Function: sends a character specified by the A register to the printer.

#### \* LPTSTT (00A8/MAIN)

Input: ---  
 Output: A register <-- printer status  
 Function: examines the current printer status. After calling this routine, the printer can be used when the A register is 255

and the Z flag is 0; when the A register is 0 and the Z flag is 1, the printer cannot be used.

\* OUTDLP (014DH,MAIN)

Input: A register <-- character code

Output: CY flag = ON at abnormal termination

Function: sends a character specified by the A register to the printer.

Differences between this routine and LPTOUT routine is as following:

- \* prints corresponding number of spaces for TAB code
- \* transforms hiragana to katakana for printers other than MSX standard
- \* returns Device I/O error at abnormal termination

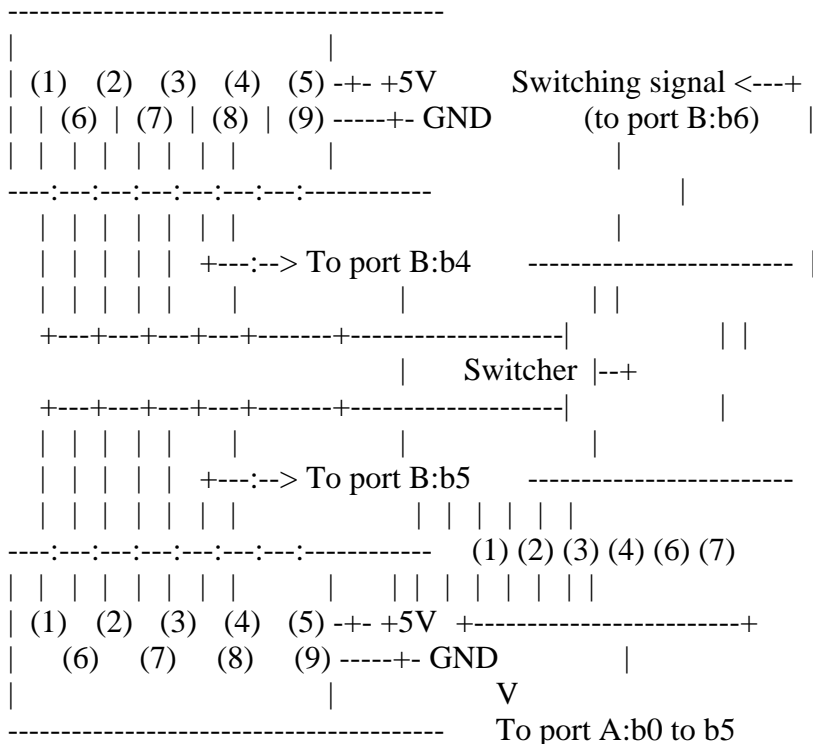
## 5. UNIVERSAL I/O INTERFACE

As described in section 1, the PSG used by MSX has two 8-bit I/O ports, port A and port B, in addition to the sound output function. In MSX, these two ports are connected to the universal I/O interface (joystick port) and are used to exchange data with the joystick or the paddle (see Figure 5.21). Various devices to be connected to this universal I/O interface have the necessary BIOS routine in ROM, so they are easily accessible.

In this section, the function of each I/O device and the method for accessing with BIOS routines are described.

Figure 5.21 Universal I/O interface

### Universal input/output interface -1



### Universal input/output interface -2

## 5.1 Functions of the Ports

Two I/O ports of PSG are used as shown in Figure 5.22.

Figure 5.22 (A) Functions of PSG port A

Port A (PSG#14)

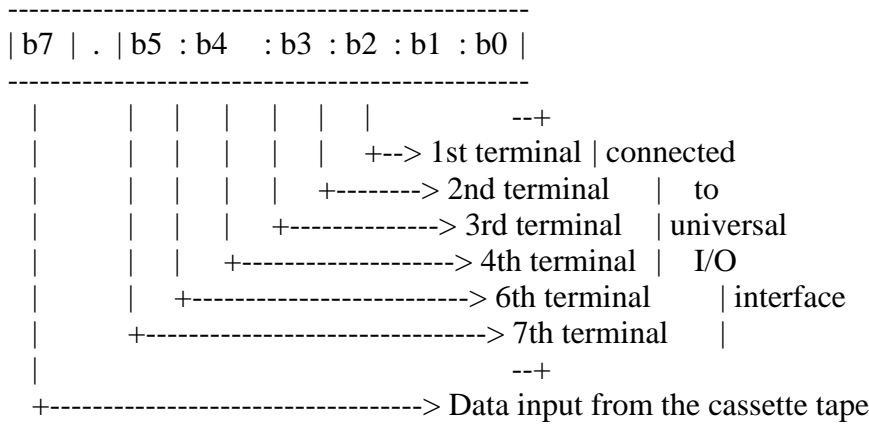
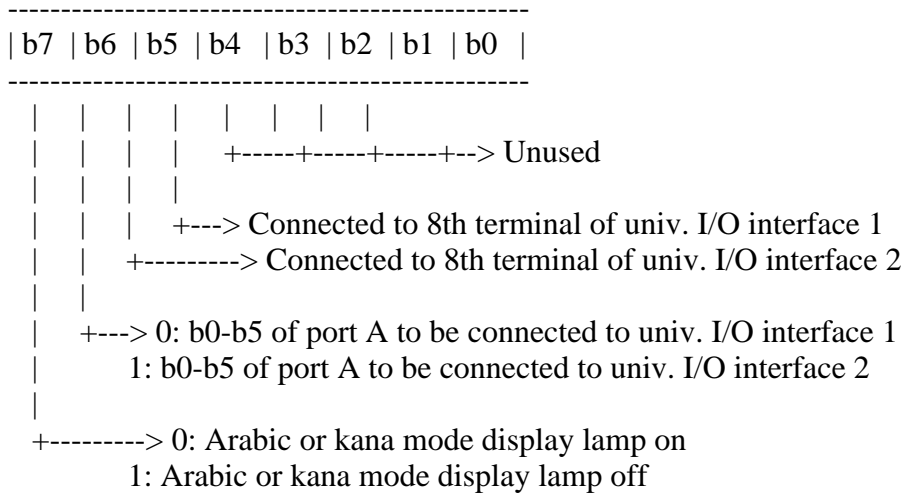


Figure 5.22 (B) Functions of PSG port B

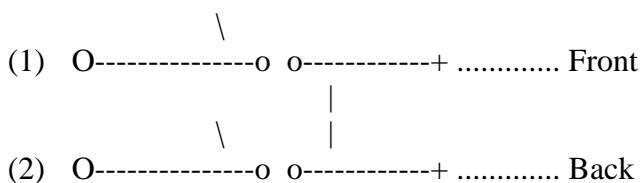
Port B (PSG#15)

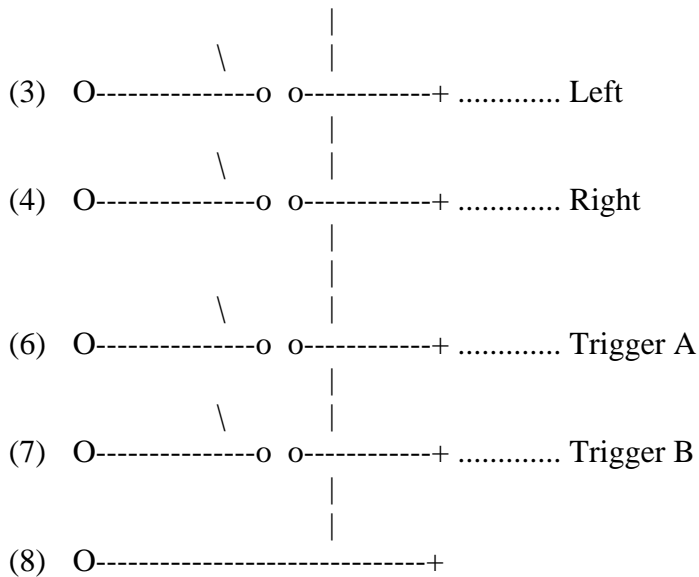


## 5.2 Joystick Use

Figure 5.23 shows the joystick circuit. As the circuit shows, sending "0" to the 8th terminal and reading the 1st to 4th and 6th to 7th terminals enable information about the stick and the trigger buttons to be obtained. However, it is advisable to use BIOS for accessing the joystick, in order to give portability to the program.

figure 5.23 Joystick circuit





The following BIOS routines are offered for accessing the joystick. These routines have similar functions to the STICK function and STRIG function of BASIC. The status of the cursor keys or the space bar, in addition to the joystick, can be read in real time.

\* GTSTCK (00D5H/MAIN) ..... read joystick

Input: A <-- joystick number (0 = cursor key, 1 and 2 = joystick)  
 Output: A <-- direction of joystick or cursor key  
 Function: returns the current status of the joystick or the cursor keys in the A register. The value is the same as the STICK function in BASIC.

\* GTTRIG (00D8H/MAIN) ..... read trigger button

Input: A <-- trigger button number (0 = space bar, 1 and 2 = trigger button A, 3 and 4 = trigger button B)  
 Output: A <-- status of trigger button or space bar (0FFH = pressed, 00H = released)  
 Function: returns the current status of the trigger buttons or the space bar in the A register. The value is 0FFH when the trigger is pressed, otherwise it is 0.

List 5.7 Joystick use

=====

```

;*****
;
; List 5.7 Joystick and trigger access
;
;*****
;
;
CHPUT EQU 00A2H
BREAKX EQU 00B7H
GTSTCK EQU 00D5H
GTTRIG EQU 00D8H

```

ORG 0D00H

;----- program start ----- Note: display joystick status

```
STICK: LD A,1 ;choose joystick 1
      CALL GTSTCK ;read joystick status
      LD (WK1),A
      LD A,1 ;choose joystick 1
      CALL GTTRIG ;read trigger status

      OR A
      JR Z,STCK1
      LD HL,WDON ;trigger ON
      JR STCK2
STCK1: LD HL,WDOFF ;trigger OFF
STCK2: CALL PUTSTR
      LD A,(WK1)
      OR A
      JR Z,BRKCH0 ;do not use joystick
      LD C,0
STCK3: DEC A
      JR NZ,STCK4
      INC C
      JR STCK3

STCK4: SLAC ;C := C*16
      SLAC
      SLAC
      SLAC
      LD B,0 ;Accounting Strings data address
      LD HL,WSTK
      ADD HL,BC
      CALL PUTSTR
```

```
BRKCH0: LD A,0DH ;put carriage return
        CALL CHPUT ;code := 0DH
```

```
BRKCHK: CALL BREAKX ;break check
        RETC
        JR STICK
```

;----- put strings to screen -----

```
PUTSTR: LD A,(HL)
        CP '$'
        RETZ
        INC HL
        CALL CHPUT
        JR PUTSTR
```

;----- string area -----

```
WDON: DB 'Trigger ON: $'
WDOFF: DB 'Trigger OFF: $'
WDSTK: DB 'UP only ',0DH,0AH,$'
        DB 'Up and Right ',0DH,0AH,$'
        DB 'Right only ',0DH,0AH,$'
```



```

DB 'Right & Down ',0DH,0AH,'$'
DB 'Down only  ',0DH,0AH,'$'
DB 'Down and Left',0DH,0AH,'$'
DB 'Left only  ',0DH,0AH,'$'
DB 'Left and Up ',0DH,0AH,'$'

```

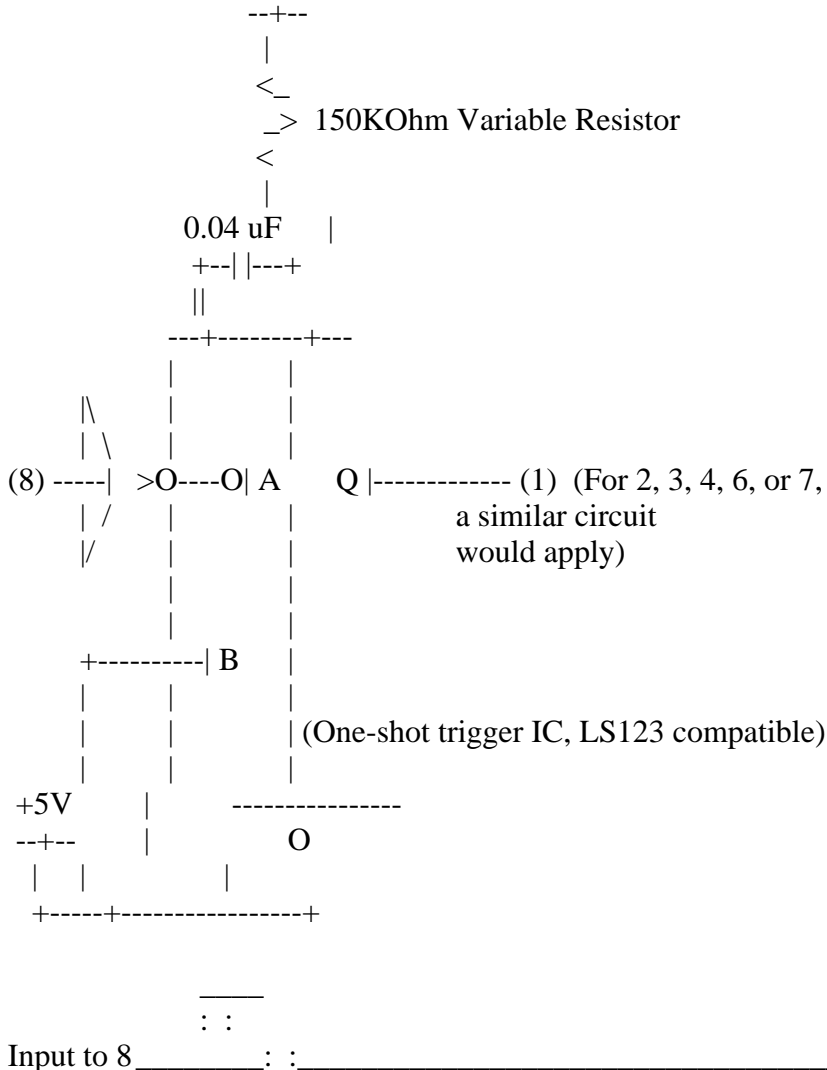
WK1: DW 0

END

### 5.3 Paddle Use

Figure 5.24 shows the paddle circuit. Sending a pulse to the 8th terminal causes the single stable multi-vibrator to generate a pulse with a specified interval. This interval depends on the value of the variable register which can range from 10 to 3000 microseconds (0.01 to 3.00 ms). Measuring the pulse length enables the value in the variable register and the turning angle to be obtained.

Figure 5.24 Paddle circuit



Output to 1 : \_\_\_\_\_ : \_\_\_\_\_  
 |<----- 10 us to 3 ms ----->|

BIOS routines for accessing the paddle are described below.

\* GTPDL (00DEH/MAIN) ..... read paddle information

Input: A <-- paddle number (1 to 12)  
 Output: A <-- turning angle (0 to 255)  
 Function: examines the status of the paddle specified in the A register  
 and returns the result in the A register.

#### 5.4 Use of Touch Panel, Light Pen, Mouse, and Track Ball

The touch panel, light pen, mouse, and track ball (cat) are accessible using the same BIOS routine. This routine is described below.

\* GTPAD (00DBH/MAIN) ..... access to various I/O devices

Input: A <-- device ID (0 to 19)  
 Output: A <-- objective information  
 Function: obtains various information as shown in Table 5.5 according to the value specified in the A register. This is the same as the PAD function of BASIC. "XXX1" in the table means the "XXX" device connected to the universal I/O interface 1; "XXX2" means the one connected to the universal I/O interface #2.

Table 5.5 GTPAD BIOS Function

Device ID	Device specified	Information returned
0		0FFH when touching panel surface, 00H when not
1	Touch panel 1	X-coordinate (0 to 255)
2		Y-coordinate (0 to 255)
3		0FFH when button is pressed, 00H when not
4		
5	Touch panel 2	Same as above
6		
7		

8		0FFH: valid data, 00H: invalid data
9	Light pen	X-coordinate (0 to 255)
10		Y-coordinate (0 to 255)
11		0FFH when switch is pressed, 00H when not
12		Always 0FFH (used to request for input)
13	Mouse 1 or track ball 1	X-coordinate (0 to 255)
14		Y-coordinate (0 to 255)
15		Always 00H (no meaning)
16		
17	Mouse 2 or track ball 2	Same as above
18		
19		

Note 1: Though information of the coordinate of the light pen (A = 9, 10) and the switch (A = 11) are read at the same time when BIOS is called with A = 8, other values are valid only when the result is 0FFH. In the case that the result of BIOS which is called with A = 8 is 00H, the coordinate values and the status of the switch contained after that are meaningless.

Note 2: Mouse and track ball are automatically distinguished.

Note 3: To obtain the coordinate value of the mouse or the track ball, do the input request call (A = 12 or A = 16), then execute the call to obtain the coordinate value actually. In this case, the interval of these two calls must be minimized as possible. Too much interval between the input request and the coordinate input causes the obtained data to be unreliable.

Note 4: To obtain the status of the trigger button of the mouse or the trigger button of the track ball, use GTTRIG (00D8H/MAIN), not GTPAD routine.

### List 5.8 Touch panel use

```

=====
;*****
;
; List 5.8 touch pad access
;
;*****
;
BREAKX EQU 00B7H

```

```
GTPAD EQU 00D8H
WRTVRM EQU 004DH
```

```
ORG 0B000H
```

```
;----- program start ----- Note: Displays "*" at position specified
by touch pad.
```

```
PAD: XOR A ;check sense
CALL GTPAD
OR A
JR NZ,PAD1
LD A,3
CALL GTPAD ;break check
OR A
RETNZ
JR PAD
```

```
PAD1: LD A,1 ;get X axis
CALL GTPAD
SRL A ;A := A/8
SRL A
SRL A
LD (WORK),A ;reserve X axis
LD A,2 ;get Y axis
CALL GTPAD
LD L,A ;HL := Y data (0-255)
LD H,0
LD C,A
LD B,0
ADD HL,BC ;HL := HL*3 (HL := 0-767)
ADD HL,BC
LD A,L
AND 11100000B
LD L,A
LD A,(WORK)
ADD A,L
LD L,A
LD BC,1800H ;VRAM start address
ADD HL,BC
LD A,2AH
CALL WRTVRM ;write VRAM
LD A,3
CALL GTPAD ;break check
OR A
RETNZ
JR PAD
```

```
;----- work area -----
```

```
WORK: DW 0 ;work
```

```
END
```

```
=====
```

List 5.9 Mouse and track ball use

=====

```

;*****
;
; List 5.9 mouse and track ball access
;
;*****

```

```

GTPAD EQU 00DBH
WRTVRM EQU 004DH
RDVRM EQU 004AH
BREAKX EQU 00B7H

```

```

ORG 0D000H

```

```

;----- program start ----- Note: Displays "*" at position specified
;                               by mouse or track ball.

```

```

TEST: CALL VADR      ;Put old data
      LD A,(WKOLD)
      CALL WRTVRM
      LD A,12
      CALL GTPAD     ;Request mouse/track ball data
      LD A,13
      CALL GTPAD     ;Read X val.
      LD (WKXVAL),A
      LD A,14
      CALL GTPAD     ;Read Y val.
      LD (WKYVAL),A

      LD A,(WKX)
      LD B,A
      LD A,(WKXVAL)
      ADD A,B
      CP 245        ;X<0?
      JR C,TEST01
      XOR A         ;X=0
      JR TEST02

```

```

TEST01: CP 32      ;X>31?
        JR C,TEST02
        LD A,31

```

```

TEST02: LD (WKX),A

```

```

      LD A,(WKY)
      LD B,A
      LD A,(WKYVAL)
      ADD A,B
      CP 245        ;Y<0?
      JR C,TEST03
      XOR A         ;Y=0
      JR TEST04

```

```

TEST03: CP 24      ;Y>23?
        JR C,TEST04

```

LD A,23

TEST04: LD (WKY),A

CALL VADR  
CALL RDVRM ;Read old data  
LD (WKOLD),A

CALL VADR  
LD A,2AH  
CALL WRTVRM ;Put cursor ("\*").

CALL BREAKX ;Break check  
RETC

CALL WAIT

JR TEST

VADR: LD A,(WKY) ;Make SCREEN Address:  
LD H,A ; From X,Y axis on WORK AREA  
LD L,0 ; To Hl reg.  
SRL H  
RR L  
SRL H  
RR L  
SRL H  
RR L  
LD A,(WKX)  
ADD A,L ; Y=32+X  
LD L,A  
LD BC,1800H ; VRAM start address  
ADD HL,BC  
RET

WAIT: LD A,0 ;WAIT routine

WLP1: INC A  
LD B,(IX+0)  
LD B,(IX+0)  
LD B,(IX+0)  
JR NZ,WLP1  
RET

;----- data -----

WKX: DB 10 ;X axis  
WKY: DB 10 ;Y axis  
WKOLD:DB 0 ;Character code on (X,Y)  
WKXVAL: DB 0 ;X variable  
WKYVAL: DB 0 ;Y variable

END

=====

## 6. CLOCK AND BATTERY-POWERED MEMORY

MSX2 uses a CLOCK-IC to for its timer function. Since this IC is battery-powered, it remains active even after MSX2 is turned off. MSX2 uses a small amount of RAM inside to set the PASSWORD or to set the screen mode at startup automatically, in addition to the CLOCK functions.

## 6.1 CLOCK-IC Functions

This IC has the following three functions:

### \* CLOCK function

- set/read the settings of "year, month, day, day of week, hour, minute, second"
- for the expression of time, 24-hour clock/12-hour clock available
- for months, months of 31 days and of 30 days are distinguished (leap years are also recognised)

### \* Alarm function

- when the time for alarm is set, CLOCK generates signals at that time.
- the time for alarm is set as "XXday XXhour XXminute".

### \* Battery-powered memory function

- has 26 sets of 4-bit memory, and can be battery-powered.
- MSX2 stores the following data in this memory:

1. adjustment value of CRT display width and height
2. initial values of SCREEN, WIDTH, colour
3. BEEP tone and volume
4. title screen colour
5. country code
6. password        --+
7. BASIC prompt   | (one of 6 to 8)
8. title caption   --+

## 6.2 Structure of the CLOCK-IC

The CLOCK-IC has four blocks inside as shown in Figure 5.25. Each block consists of 13 sets of 4-bit registers, which are specified by addresses from 0 to 12. In addition, it has three 4-bit registers for selecting the block or controlling functions; they are specified by the addresses from 13 to 15.

The registers inside the block (#0 to #12) and the MODE register (#13) can be read from and written to. The TEST register (#14) and RESET register (#15) can only be written to.

Figure 5.25 Clock IC structure

BLOCK 0        BLOCK 1        BLOCK 2        BLOCK 2

	(CLOCK)	(ALARM)	(RAM-1)	(RAM-2)
0	Seconds (the 1st decimal place)			
1	Seconds (the 2nd decimal place)			
.	.		Any data	Any data
.	.			
.	.			
.	.			
12	Year (the 2nd decimal place)			
	:<-- 4 bits -->:	:<-- 4 bits -->:	:<-- 4 bits -->:	:<-- 4 bits -->:

13	MODE	
14	TEST	
15	RESET	
	:<-- 4 bits -->:	

### 6.3 MODE Register Functions

The MODE register has the following 3 functions:

#### \* Selecting block

To read from or write to registers from #0 to #12, select the block to be used and then access the objective address. The 2 low order bits of the MODE register are used to select the block.

Registers from #13 to #15 are accessible whichever block is selected.

#### \* Alarm output ON/OFF

To switch the alarm input ON/OFF, use bit 2 of the MODE register. Since the standard MSX2 does not support the alarm, modifying this bit causes nothing to happen in general.

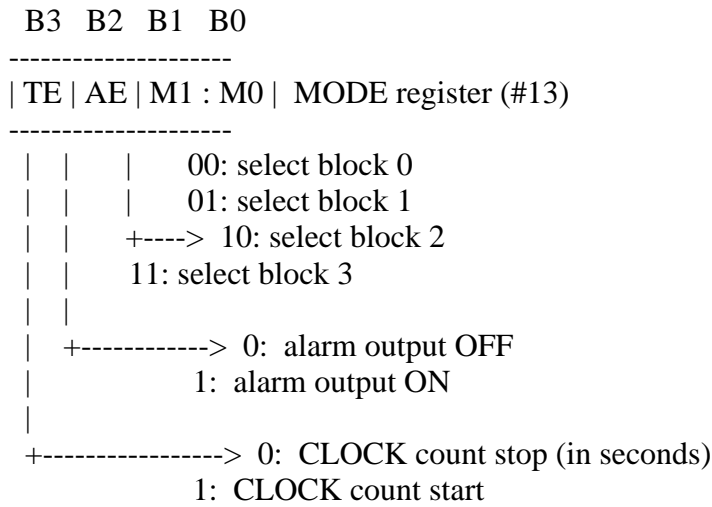
#### \* Terminating CLOCK count

By writing "0" in bit 3 of the MODE register, the count in seconds is stopped



(the stages before the seconds are not stopped) and the clock function is terminated. By writing "1" in bit 3, the count is resumed.

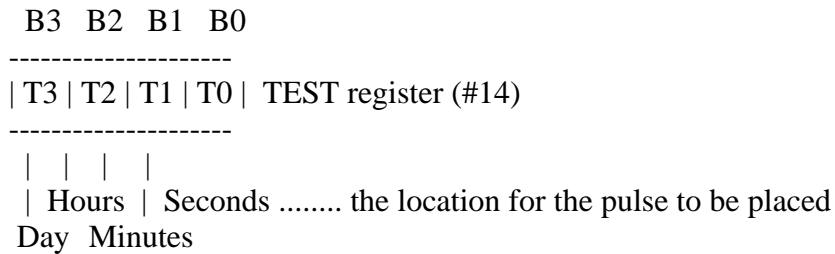
Figure 5.26 MODE register functions



#### 6.4 TEST Register functions

The TEST register (#14) is used to increment the upper counter quickly and to confirm that date and time carries are done correctly. Setting "1" in each bit of the register, the pulse of  $2^{14}$  (=16384)[Hz] is directly set in day, hour, minute, and second counters.

Figure 5.27 TEST register functions



#### 6.5 RESET Register Functions

The RESET register (#15) has the following functions:

\* Resetting the alarm

Setting "1" in bit 0 causes all alarm registers to be reset to 0.

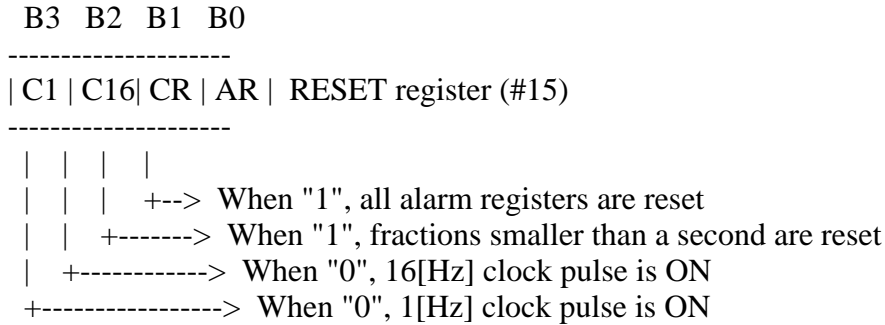
\* Setting the seconds

Setting "1" in bit 1 causes the stage before the seconds to be reset. Use this function to set the seconds correctly.

\* Clock pulse ON/OFF

Setting "1" in bit 2 turns the 16Hz clock pulse output ON, and setting "0" in bit 3 turns the 1Hz clock pulse output ON. Note that both are not supported by the MSX2 standard.

Figure 5.28 RESET register function



## 6.6 Setting the Clock and Alarm

\* Setting date and time

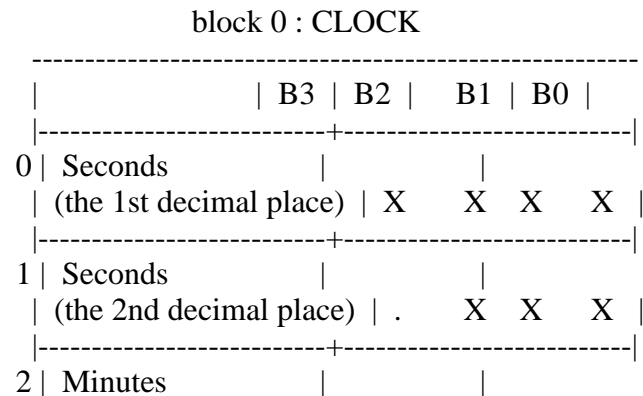
Block 0 is used to set the clock. Selecting block 0 in the MODE register and writing data in the objective register causes the date and the time to be set. The current time is acquired by reading the contents of the register. See Figure 5.29 for the meaning of the register and its address.

Block 1 is used to set the alarm. Note that the time of the alarm can be set only in days, hours, and minutes. Nothing happens, in general, when the time of the clock meets the time of the alarm.

In the clock, the year is represented by 2 digits (registers #11 and #12). In MSX-BASIC, the 2 low order digits of the year is represented by adding the offset 80 to this value. For example, after setting register #11 to 0 and register #12 to 0, the year would be 80, as "80/XX/XX", when the date is read by using the GET DATE instruction of BASIC.

The day of the week is represented by 0 to 6. This is only a mod 7 counter which is renewed along with the date, and the correspondence between the actual day of the week and the number value 0 to 6 is not defined.

Figure 5.29 Setting the CLOCK and ALARM



	(the 1st decimal place)	X	X	X	X
3	Minutes (the 2nd decimal place)	.	X	X	X
4	Hours (the 1st decimal place)	X	X	X	X
5	Hours (the 2nd decimal place)	.	.	X	X
6	Day of the week	.	X	X	X
7	Day (the 1st decimal place)	X	X	X	X
8	Day (the 2nd decimal place)	.	.	X	X
9	Month (the 1st decimal place)	X	X	X	X
10	Month (the 2nd decimal place)	.	.	.	X
11	Year (the 1st decimal place)	X	X	X	X
12	Year (the 2nd decimal place)	X	X	X	X

block 1 : ALARM

		B3	B2	B1	B0
0	_____	.	.	.	.
1	_____	.	.	.	.
2	Minutes (the 1st decimal place)	X	X	X	X
3	Minutes (the 2nd decimal place)	.	X	X	X
4	Hours (the 1st decimal place)	X	X	X	X
5	Hours (the 2nd decimal place)	.	.	X	X
6	Day of the week	.	X	X	X

7	Day (the 1st decimal place)	X	X	X	X
8	Day (the 2nd decimal place)	.	.	X	X
9		.	.	.	.
10	12 or 24 hours	.	.	.	X
11	Leap year counter	.	.	X	X
12		.	.	.	.

Bits indicated by a "." are always 0 and cannot be modified.

\* Selecting 12-hour clock/24-hour clock

Two clocks can be selected; one is a 24-hour clock which represents one o'clock in the afternoon as 13 o'clock, and the other is a 12-hour clock which represents it as 1 p.m. Register #10 is used to select between them. As shown in Figure 5.30, the 12-hour clock is selected when B0 is "0" and the 24-hour clock when B0 is "1".

Figure 5.30 Selecting 12-hour clock/24-hour clock

B3	B2	B1	B0
.	.	.	B0

Register #10 (block 1)

0: 12-hour clock  
1: 24-hour clock

Figure 5.31 Morning/afternoon flag for 12-hour clock

B3	B2	B1	B0
.	.	B1	X

register #5 (block 0)

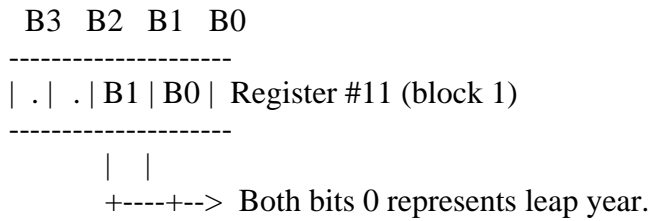
0: before noon  
1: after noon

\* Leap year counter

Register #11 of block 1 is a mod 4 counter which is renewed along with the

count of the year. When the 2 low order bits of this register are 00H, that is considered as a leap year and 29 days are counted in February.

Figure 5.32 Leap year determination

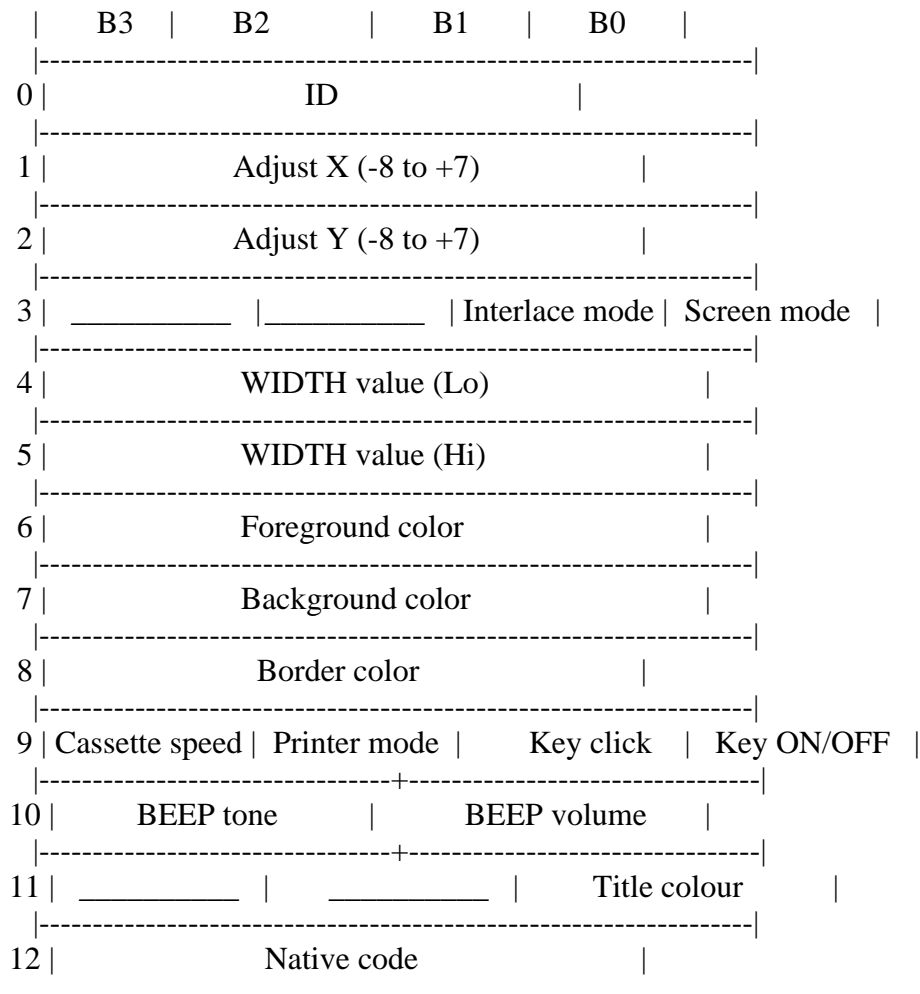


### 6.7 Contents of the Battery-powered Memory

Blocks 2 and 3 of the CLOCK-IC are used as the battery-powered 4-bit x 13 memory blocks. MSX2 uses this area as shown below.

\* Contents of block 2

Figure 5.33 Contents of block 2

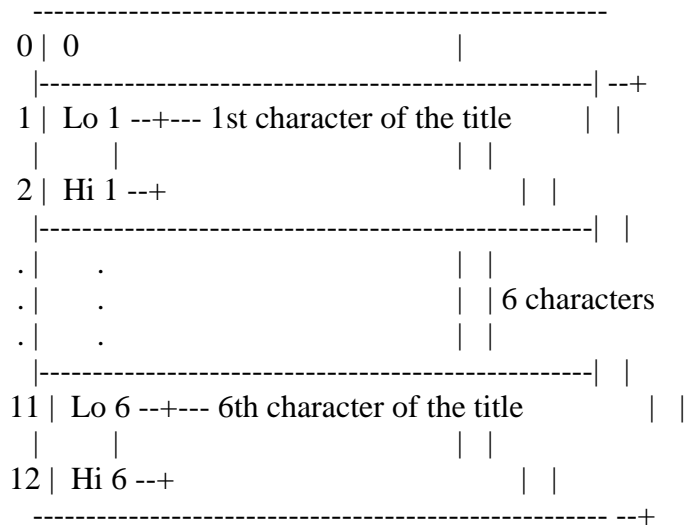


\* Contents of block 3

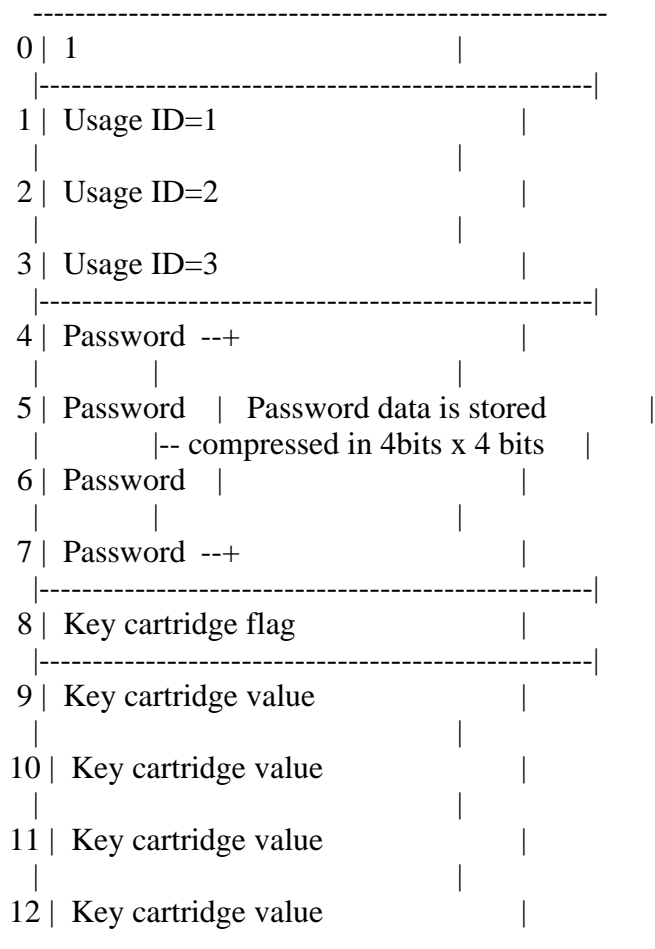
Block 3 has three functions, depending on the contents of the ID value (register #0). Figure 5.34 shows the functions.

Figure 5.34 Contents of block 3

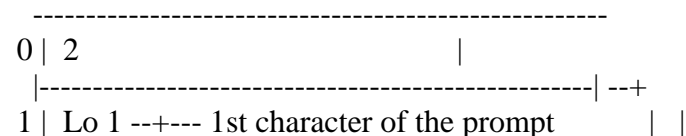
ID=0: displays the title (within 6 characters) on the initial screen

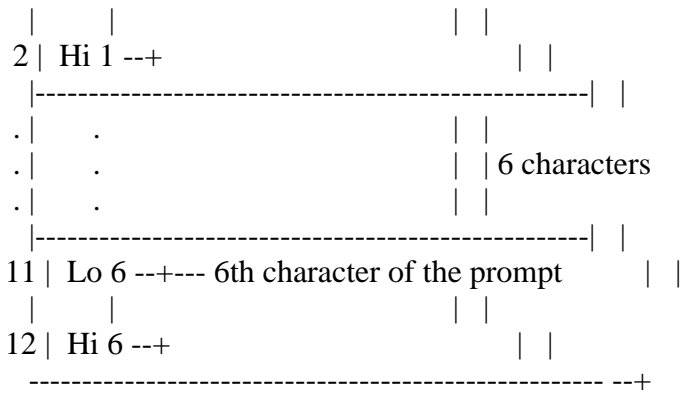


ID=1: sets the password



ID=2: sets the prompt on BASIC





## 6.8 Access to the CLOCK-IC

The following BIOS routines are offered to access the clock and the battery-powered memory. Since these routines reside in SUB-ROM, they are called by using the inter-slot call.

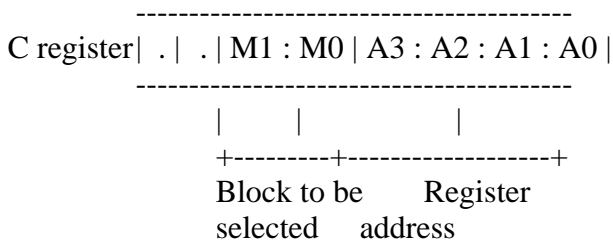
\* REDCLK (015FH/SUB) ..... read CLOCK-IC data

Input: C <-- CLOCK-IC address (see Figure 5.35)

Output: A <-- data obtained (only 4 low order bits valid)

Function: reads CLOCK-IC register in the address specified by the C register and stores in the A register. Since the address specification includes the block selection information as shown in Figure 5.35, it is not necessary to set the MODE register and then read the objective register.

Figure 5.35 CLOCK-IC register specification method



\* WRTCLK (01F9H/SUB) ..... write CLOCK-IC data

Input: C <-- CLOCK-IC address (see Figure 5.35)

A <-- data to be written (4 low order bits)

Output: ---

Function: write the contents of the A register in the CLOCK-IC at the address specified by the C register. The address is specified in the format shown in Figure 5.35 as REDCLK.

List 5.10 shows an example of this BIOS routine.

List 5.10 Setting the prompt

=====

```

;*****
;
;
; List 5.10 set prompt message
;
;*****
;
;
WRTCLK: EQU 01F9H
EXTROM: EQU 015FH

```

```

ORG 0B000H

```

```

;----- program start ----- ;Note: Set prompt message for BASIC.

```

```

START: LD C,00110000B ;address data
LD A,2 ;ID := prompt mode
CALL WRTRAM ;write to back-up RAM

LD B,6 ;loop counter
LD HL,STRING ;prompt data
L01: LD A,(HL) ;read string data
AND 0FH ;A := hi 4 bit
INC C ;increment address
CALL WRTRAM ;write data to back-up RAM
LD A,(HL)
RRCA
RRCA
RRCA
RRCA
AND 0FH
INC C ;increment address
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 ;use interslot call
POP BC
POP HL
RET

```

```

;----- string data -----

```

```

STRING: DB 'Ready?'

```

```

END

```

```

=====

```

```

MSX2 TECHNICAL HANDBOOK

```

```

-----

```



Edited by: ASCII Systems Division  
Published by: ASCII Corporation - JAPAN  
First edition: March 1987

Text file typed by: Nestor Soriano (Konami Man) - SPAIN  
October 1997

Changes from the original:

- In description of SUBROM routine, comment "see page 352" has been changed to "see appendix 2..."

- In description of SLTATR and SLTWRK work areas, expressions for calculate the concrete work area for a given slot and page have been added.

- In the first line after beginning of section 7.2.3, "The following routines..." has been corrected to "The following addresses..."

- In Figure 5.52, indication of F380H address was placed in the middle of the user's area. It has been moved to the beginning of system work.

-----

## CHAPTER 5 - ACCESS TO PERIPHERALS THROUGH BIOS (Part 7)

### 7. SLOTS AND CARTRIDGES

The CPU (Z80) used in the MSX can access an address space of only 64K bytes (0000H to FFFFH). MSX is set up to access an effective space of 1M bytes. This is accomplished by using "slots", which allocate more than one memory byte or device to the same address.

This chapter introduces the use of the slot and information necessary to connect the cartridge software or the new device to MSX via the slot.

#### 7.1 Slots

A slot is an interface to effectively use a large address space, and to interface any memory or devices connected to the MSX address bus installed via the slot. The BASIC ROM inside the machine or RAM at MSX-DOS mode are not exceptions. The place at which the cartridge software is installed is also one of the slots. The following descriptions introduce how the software and the devices are connected to the slot.

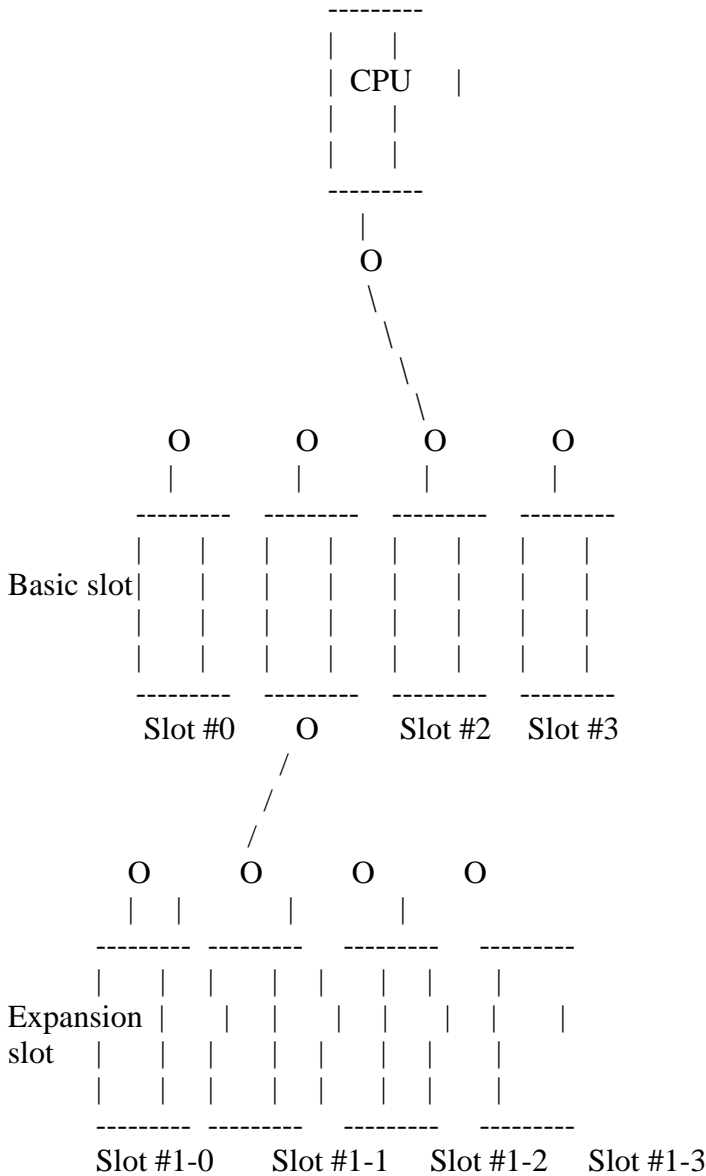
##### 7.1.1 Basic slot and expansion slot

The slot is either a basic slot or a expansion slot. A "basic slot" is a slot directly connected to the CPU address bus, as shown in Figure 5.36. The standard MSX machine can have up to four basic slots. The basic slot can be expanded up to four slots by connecting a slot expansion box (in some cases, the expansion is already done inside the machine), and is called "expansion slots". When each of four basic slots is expanded to four expansion slots, the maximum number of slots is 16. If you multiply 16 slots x 64K bytes you

will get 1M bytes of accessible address space.

Note that the system itself cannot be started when expansion slot boxes are connected to the expansion slot. So the user should only connect expansion slot boxes to the basic slots. (Though the slot for the standard MSX cartridge is always a basic slot, in some cases the connector for the optional hardware of each machine might be connected to the expansion slot).

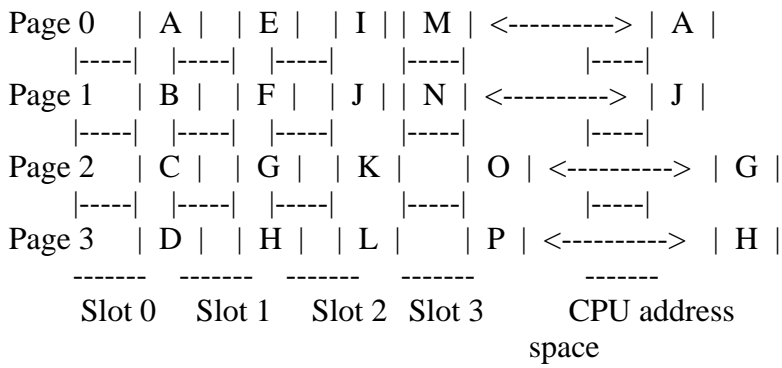
Figure 5.36 Basic slot and expansion slot



Each slot has 64K bytes from 0000H to FFFFH of address space and MSX manages it by dividing it into four "pages" of 16K bytes each. The CPU can select and access any slot for each page, and, as shown in Figure 5.37, it is possible to select and combine portions of several slots. Note that a pages with a given page number cannot be assigned to a page with a different page number (that is, page n of each slot is also page n to the CPU).

Figure 5.37 Example of the page selection

-----



### 7.1.2 Selecting slots

Selecting slots is different for the basic slot than for the expansion slot. For basic slots, it is done through the I/O port at A8H (see Figure 5.38), and for expansion slots, it is done through the "expansion slot selection register (FFFFH)" of the installed expansion slot (see Figure 5.39). It is not recommended to change them directly, so the user should not switch the slots without careful planning. When the program switches the slot of the page in which it resides, the action is not always predictable. To call the program in another slot, use the inter-slot call described in the next section.

Figure 5.38 Selecting the basic slot

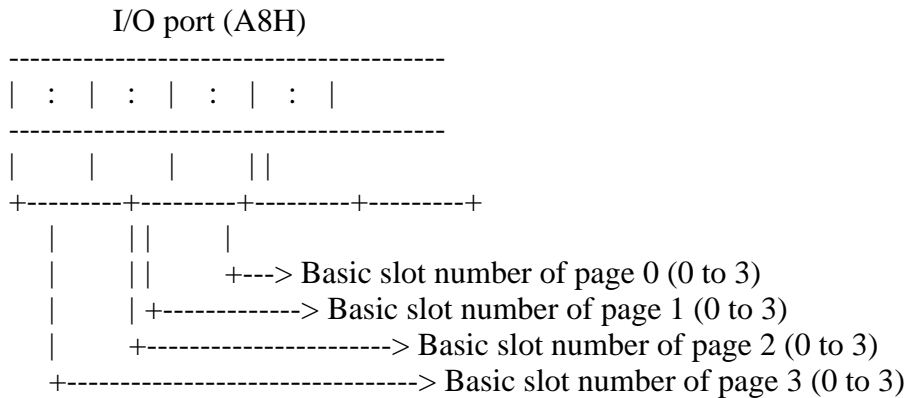
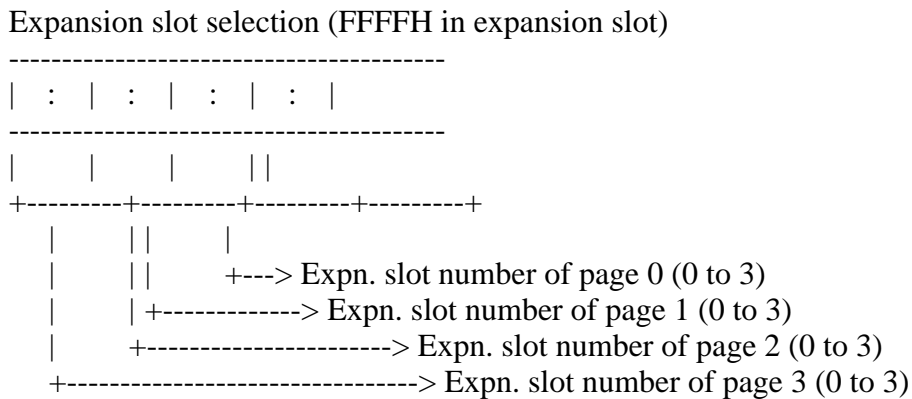


Figure 5.39 Selecting the expansion slot



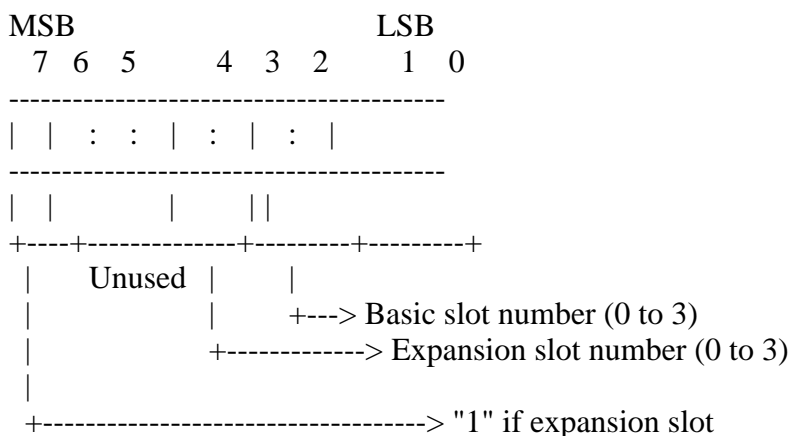
Note: to identify the kind of slot, in the case of the expansion slot, the value written is read as the reversed value. The value of this register is the same inside the basic slot.

The slot where MAIN-ROM or RAM is installed and the slot number of the slot for the cartridge depend on the machine. Refer to the appropriate manual to see how slots are used on your MSX. But the MSX standard guarantees the normal operation no matter what is in the slots, so it is not necessary to worry about the slot use, as long as you are following the standard.

In some cases, however, it is required to know the slot number of the specified software. For example, in the previous version, BASIC MAIN-ROM was placed in basic slot #0 or in expansion slot #0-0 when basic slot #0 was expanded. So when MSX1 is upgraded to have the MSX2 functions by installing MSX-VIDEO and BASIC ver 2.0 ROM, the MAIN ROM should be placed somewhere other than slot #0 or slot #0-0. The slot where MSX2 SUB-ROM resides depends on the machine, but the information about the slot where BASIC interpreter ROM resides can be obtained by referring to the work area described below (the slot information can be obtained in the format shown in Figure 5.40). When calling BIOS from DOS, examine the slot of MAIN-ROM in this way.

- \* EXPTBL (FCC1H, 1) the slot of MAIN-ROM
- \* EXBRSA (FAF8H, 1) the slot of SUB-ROM (0 for MSX1)

Figure 5.40 Format for the slot



When a given routine resides over page 1 and page 2 (4000H to BFFFH), the same slot for page 2 as the one for page 1 should be selected when the jump from page 1 to page 2 occurs within this routine. To do this, you need to examine the slot, in page 1, where the program resides and then to switch page 2 to that slot. To obtain information about the slot where the program currently is, execute the program shown in List 5.11.

List 5.11 Program to know the current slot

```

=====
;*****
;
; List 5.11 to know where you are
;*****
;
; Suppose your program cartridge is 32K bytes
; long (4000h..0BFFFH). You set the ID at 4000H
; and 4001H and the execution start address within
; page 1 (4000h..7FFFH), MSX passes control
; to this address so the part which resides in
; page 2 is not yet enabled at this point. You

```

```

;   have to know where you are (in what primary
;   slot, in what secondary slot) and enable the
;   part at page 2. Below is the sample program
;   to do this.
;
ENASLT EQU    0024H    ;enable slot
RSLREG EQU    0138H    ;read primary slot select register
EXPTBL EQU    0FCC1H   ;slot is expanded or not

```

```

;----- program start -----

```

```

ENAP2:
    CALL  RSLREG    ;read primary slot #
    RRCA                ;move it to bit 0,1 of [Acc]
    RRCA
    AND    00000011B
    LD  C,A
    LD  B,0
    LD  HL,EXPTBL ;see if this slot is expanded or not
    ADD  HL,BC
    LD  C,A    ;save primary slot #
    LD  A,(HL) ;See if the slot is expanded or not
    AND  80H
    OR  C    ;set MSB if so
    LD  C,A    ;save it to [C]
    INC HL    ;Point to SLTTBL entry
    INC HL
    INC HL
    INC HL
    LD  A,(HL) ;Get what is currently output
                ;to expansion slot register
    AND  00001100B
    OR  C    ;Finally form slot address
    LD  H,80H
    JP  ENASLT ;enable page 2

    END

```

## 7.2 Inter-slot Calls (calls between slots)

As described above, programs reside in different slots, so a program not in the current slot might be needed in some cases. The most common cases are listed below:

- (1) calling BIOS in MAIN-ROM from MSX-DOS level
- (2) calling BIOS in SUB-ROM from BASIC level (only the case of MSX2)
- (3) calling BIOS in MAIN-ROM or SUB-ROM from cartridge software

In doing such calls, to switch slots easily and safely, there is a group of BIOS routines called the inter-slot calls, which can be called from the routine in any slot. This section describes the use of the inter-slot calls.

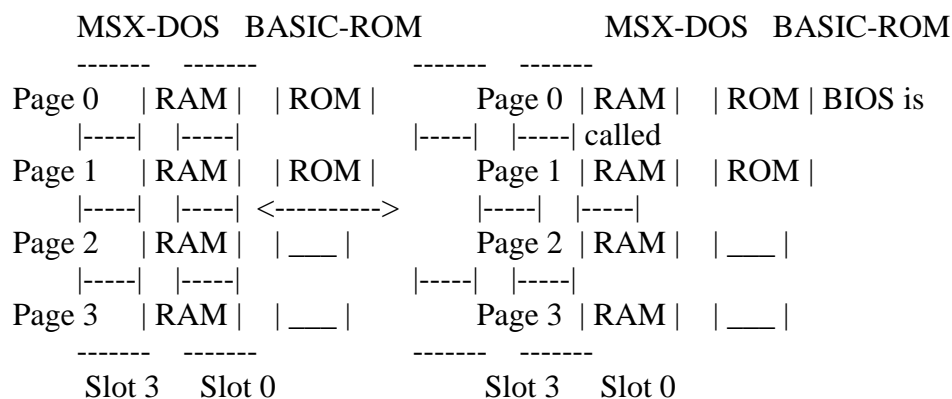
### 7.2.1 Inter-slot call operation

When calling BIOS in MAIN-ROM from MSX-DOS, the state of slots changes as described below.

- (1) Since, in the initial MSX-DOS mode, RAM is selected all over 64K address space, BASIC-ROM cannot be accessible in this state (see Figure 5.41-a).
- (2) To call BIOS in ROM, switch page 0 to MAIN-ROM or BASIC to access. Then, call BIOS (see Figure 5.41-b).
- (3) Restore the original status after BIOS operations and return to the initial address.

Figure 5.41 Inter-slot call

a) When using MSX-DOS



This is easily done when the program resides in other than page 0, but there can be some problem when the calling program resides in page 0 which is the same page as BIOS of the called program. Care is needed to prevent the program from disappearing itself and generating unpredictable results when it is switched to page 0. The inter-slot call settles this problem by jumping to page 3 temporarily and then switching slots.

### 7.2.2 Inter-slot call use

There are several ways to execute inter-slot calls, as described below. They are included in MAIN-ROM as BIOS calls. Some of them are offered in MSX-DOS, so inter-slot calls can be executed under MSX-DOS.

#### (1) Inter-slot call routines in BIOS

\* RDSLT (000CH/MAIN) ..... read value at specified address of specified slot

Input: A register <-- slot specification

HL register <-- address to be read at

Output: A register <-- value which has been read out

Use: AF, BC, DE

Function: reads the value at the specified address of the specified slot and stores it in the A register. The slot specification is done using the A register in the form shown in Figure

5.40. At this point, when the objective slot is the basic slot, set "0" to the 6 high order bits and define the slot #0 to #3 using the 2 low order bits. When specifying the expansion slot, specify the basic slot by bit 0 and bit 1 and the expansion slot by bit 2 and bit 3 and set bit 7 to "1".

\* WRSLT (0014H/MAIN) ..... write value at specified address of specified slot

Input: A register <-- slot specification (same format as in Figure 5.40)

HL register <-- address to be written in

E register <-- data to be written

Output: ---

Use: AF, BC, D

Function: writes E register value in the address specified by HL register of the slot specified by the A register (the specification format is the same as in Figure 5.40).

\* CALSLT (001CH/MAIN) ..... call specified address of specified slot

Input: 8 high order bits of IY register <-- slot (same format as in Figure 5.40)

IX register <-- address to be called

Output: depends on the result of the called program

Use: depends on the result of the called program

Function: calls the routine at the address specified by IX register of the slot specified by the 8 high order bits of IY register (the specification format is the same as in Figure 5.40).

\* ENASLT (0024H/MAIN) ..... swith slots

Input: A register <-- slot (same format as in Figure 5.40)

HL register <-- specifies the page to switch the slot by 2 high order bits

Output: ---

Use: all

Function: switches the page specified by the 2 high order bits of the HL register to the slot specified by the A register.

\* CALLF (0030H/MAIN) ..... call specified address of specified slot

Input: specifies the slot and the address in the inline parameter format

Output: depends on the result of the called program

Use: depends on the result of the called program

Function: calls the specified address of the specified slot, but, different from CALSLT described above, the slot and the address is specified in the inline parameter format, as described below. That is, parameters are passet by one byte (same format as RDSLT) to specify that the slot is placed

just after the instruction which calls CALLF and the next two bytes to specify the address are placed. "CALL 0030H" may be replaced by the RST (restart) instruction, "RST 30H". In this case, the inter-slot call is done in 4 bytes.

Figure 5.42 Example of the inter-slot call execution

```

-----
|          RST 30H      ;interslot call          |
|          DB 00000000B ;select slot#0         |
|          DW 006CH     ;call address = 006CH   |
|-----|

```

\* RSLREG (0138H/MAIN) ..... read the basic slot selection register

Input: ---  
 Output: A register <-- value which has been read  
 Use: ---  
 Function: reads the contents of the basic slot selection register and stores it in the A register.

\* WSLREG (013BH/MAIN) ..... write in the basic slot selection register

Input: A register <-- value to be written  
 Output: ---  
 Use: ---  
 Function: writes the A register value in the basic slot selection register and selects the slot.

\* SUBROM (015CH/MAIN) ..... call specified address in SUB-ROM

Input: IX register <-- address to be called, PUSH IX  
 (see Appendix 2, SUB-ROM list)  
 Output: depends on the result of the called program  
 Use: background register and IX, IY registers are reserved  
 Function: This is the routine to call BASIC SUB-ROM especially. The slot where SUB-ROM resides is automatically examined. Normally, EXTROM, described below, is used.

\* EXTROM (015FH/MAIN) ..... call specified address in SUB-ROM

Input: IX register <-- address to be called  
 Output: depends on the result of the called program  
 Use: background register and IY register are reserved  
 Function: This is the routine to call BASIC SUB-ROM. The difference between this and SUB-ROM above is the point whether the IX register value is pushed.



## (2) Inter-slot call in MSX-DOS

In MSX-DOS, five kinds of inter-slot calls are offered and their entry addresses are defined at jump vectors of MSX-DOS. These are the same as ones in BIOS, so refer to BIOS above for their functions or use. Note that these routines should not be used when calling routines in SUB-ROM from MSX-DOS.

- \* RDSLT (000CH) ..... read value at specified address of specified slot
- \* WRSLT (0014H) ..... write value at specified address of specified slot
- \* CALSLT (001CH) ..... call specified address of specified slot
- \* ENASLT (0024H) ..... make specified slot available
- \* CALLF (0030H) ..... call specified address of specified slot

### List 5.12 Calling BIOS from MSX-DOS

```
=====
;*****
; List 5.12 How to use BIOS from MSX-DOS.
;*****
;
CALSLT EQU    001CH    ;Inter slot call
EXBRSAEQU    0FAF0H    ;Slot address of BIOS (main) ROM
EXPTBLEQU    0FCC1H    ;Slot address of extended ROM
;
; LD IY,(EXPTBL-1) ;Load slot address of the BIOS ROM
; ;in high byte of IY
; LD IX,address of the BIOS jump table
; CALL CALSLT
;
;----- Sample program to set text mode -----
INITXT EQU    006CH
LINL40 EQU    0F3AEH
;
TOTEXT: LD B,40
LD A,(EXBRSA) ;slot address of SUB-ROM
OR A ;0 if MSX1
JR Z,TO40
LD B,80

TO40: LD (LINL40),B ;set width into work area
LD IX,INITXT
LD IY,(EXPTBL-1) ;get expanded slot status to IYH
CALL CALSLT ;perform an inter-slot call
EI ;because CALSLT do DI
RET

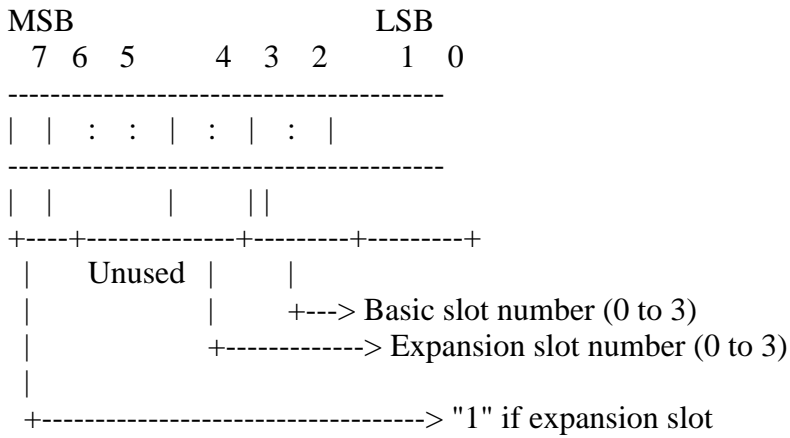
END
=====
```

### 7.2.3 Work area to obtain the slot status

The following addresses involve the slot work area.

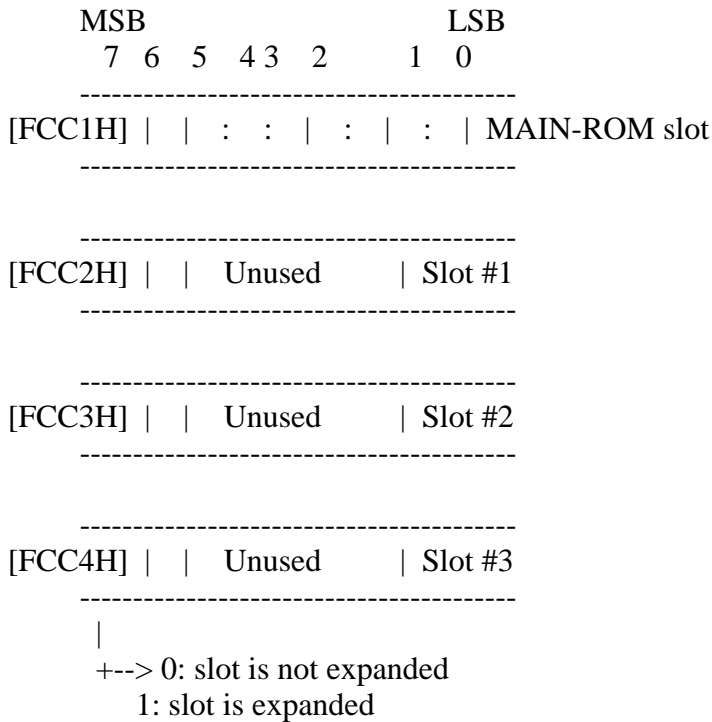
\* EXBRSA (FAF8H, 1) ..... SUB-ROM slot

Figure 5.43 SUB-ROM slot



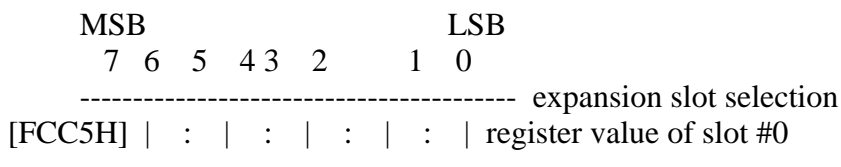
\* EXPTBL (FCC1H, 4) ..... whether the basic slot is expanded or not

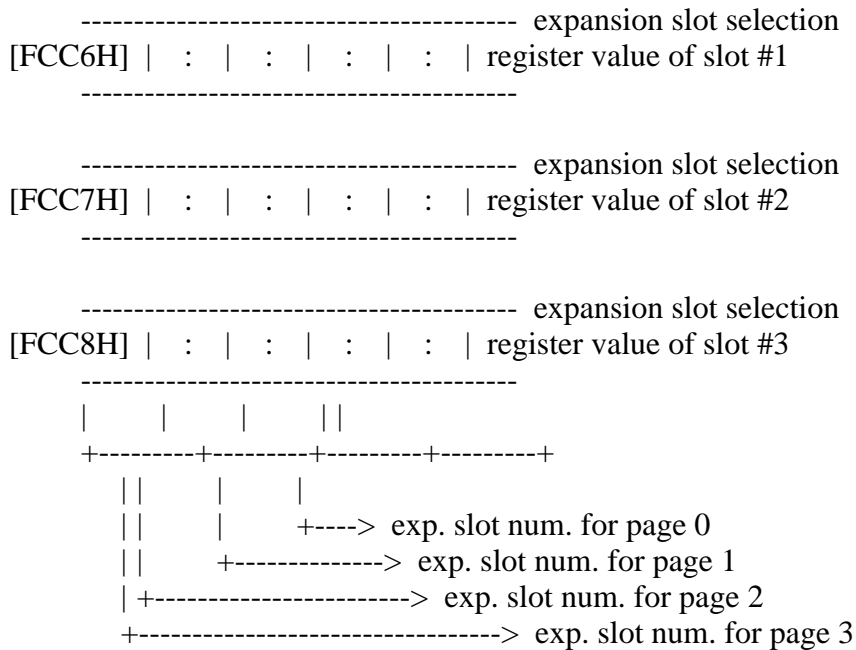
Figure 5.44 Selecting the basic slot



\* SLTTBL (FCC5H, 4) ..... preservation area for the expansion slot selection register value

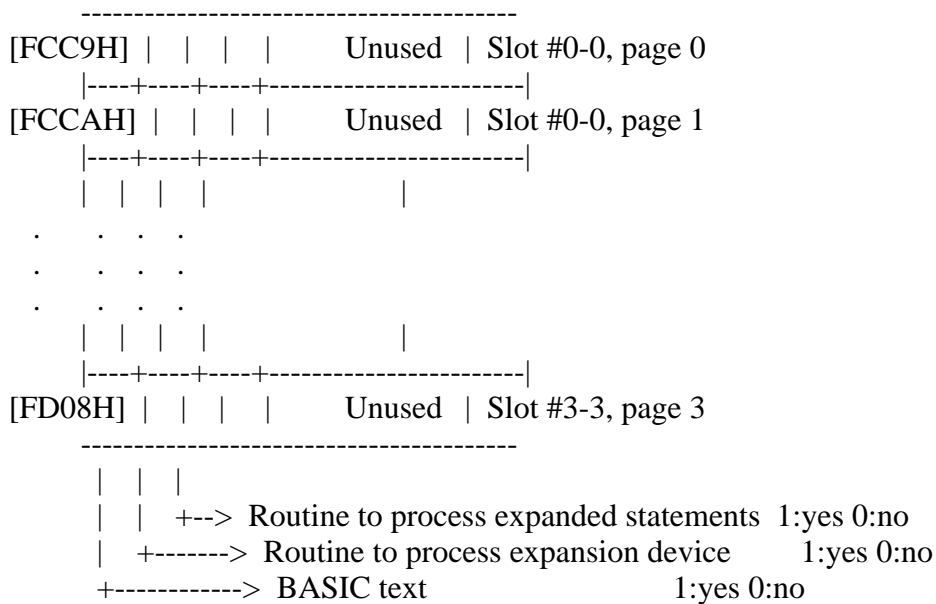
Figure 5.45 Selecting the expansion slot





\* SLTATR (FCC9H, 64) ..... test for existence of application in each slot/page

Figure 5.46 Test for existence of application



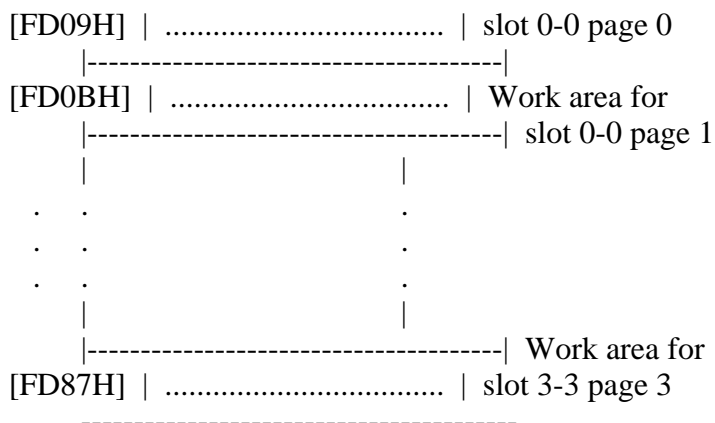
The concrete work area for a given slot and page can be obtained with the following expression:

$$\text{SLTATR address} = \text{FCC9H} + 16 * \text{basic slot} + 4 * \text{expansion slot} + \text{page}$$

\* SLTWRK (FD09H, 128) ..... work area for application

Figure 5.47 Work area for application

----- Work area for



The concrete work area for a given slot and page can be obtained with the following expression:

$$\text{SLTWRK address} = \text{FD09H} + 32 * \text{basic slot} + 8 * \text{expansion slot} + 2 * \text{page}$$

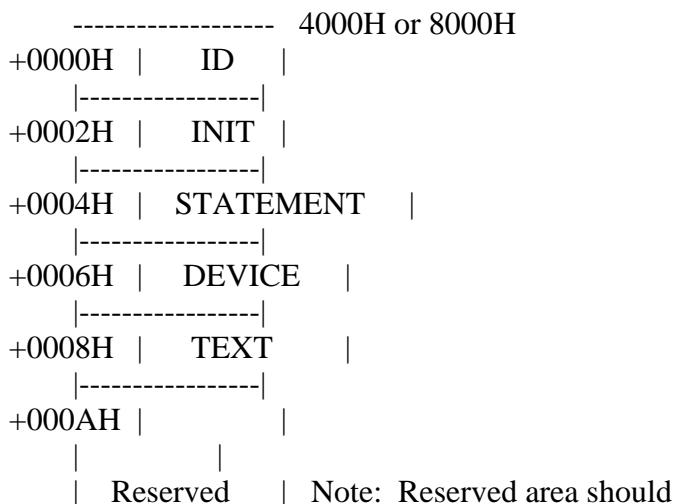
### 7.3 Developing Cartridge Software

MSX machines usually have at least one external slot and the hardware to be put there is called a "cartridge". There are cartridges such as the ROM cartridges for application programs or games, input-output device cartridges for a disk or RS-232 interface, RAM expansion cartridges for expanding RAM, and slot expansion cartridges for expanding slots. These cartridges make the MSX easy to upgrade. BASIC and assembly language programs can also be stored in ROM cartridge easily. This section describes how to develop cartridge software.

#### 7.3.1 Cartridge header

MSX cartridges have a 16-byte common header and, when the system is reset, the cartridge is initialised by the information written in this header. For ROM cartridges of BASIC or assembly language programs, they can be automatically started by using the information written in the header. Figure 5.48 shows the cartridge header configuration.

Figure 5.48 Program cartridge header



be filled with 00H.

+0010H -----

\* ID

In the case of ROM cartridges, these two bytes have codes "AB" (41H, 42H). For SUB-ROM cartridges, the ID is "CD".

\* INIT

When the cartridge is made to initialise the work area or I/O, these two bytes are the addresses for the initialization routine; otherwise 0000H is assumed. After instructions such as getting the work area in the initialization routine are placed, end with "RET". All registers except the SP register may be destroyed. For assembly language programs, such as games, which loop within the cartridge, it is possible to execute the object program from here.

\* STATEMENT

When the cartridge is made to expand the CALL statement, these two bytes are the address for the statement expansion routine; otherwise 0000H is assumed. If so, the statement expansion routine should reside at 4000H to 7FFFH.

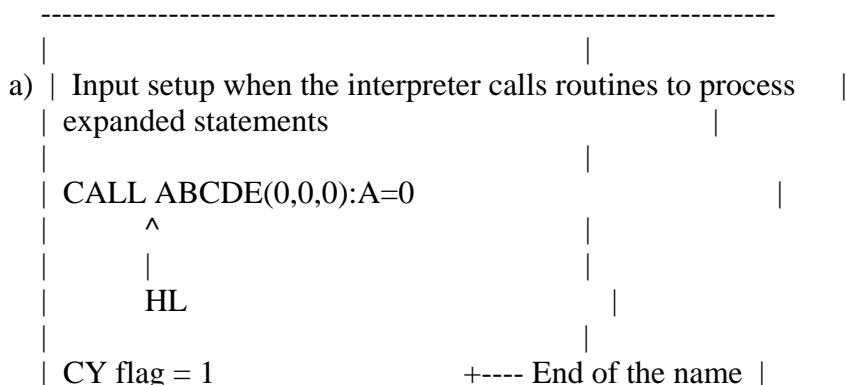
The CALL statement is described in the following format:

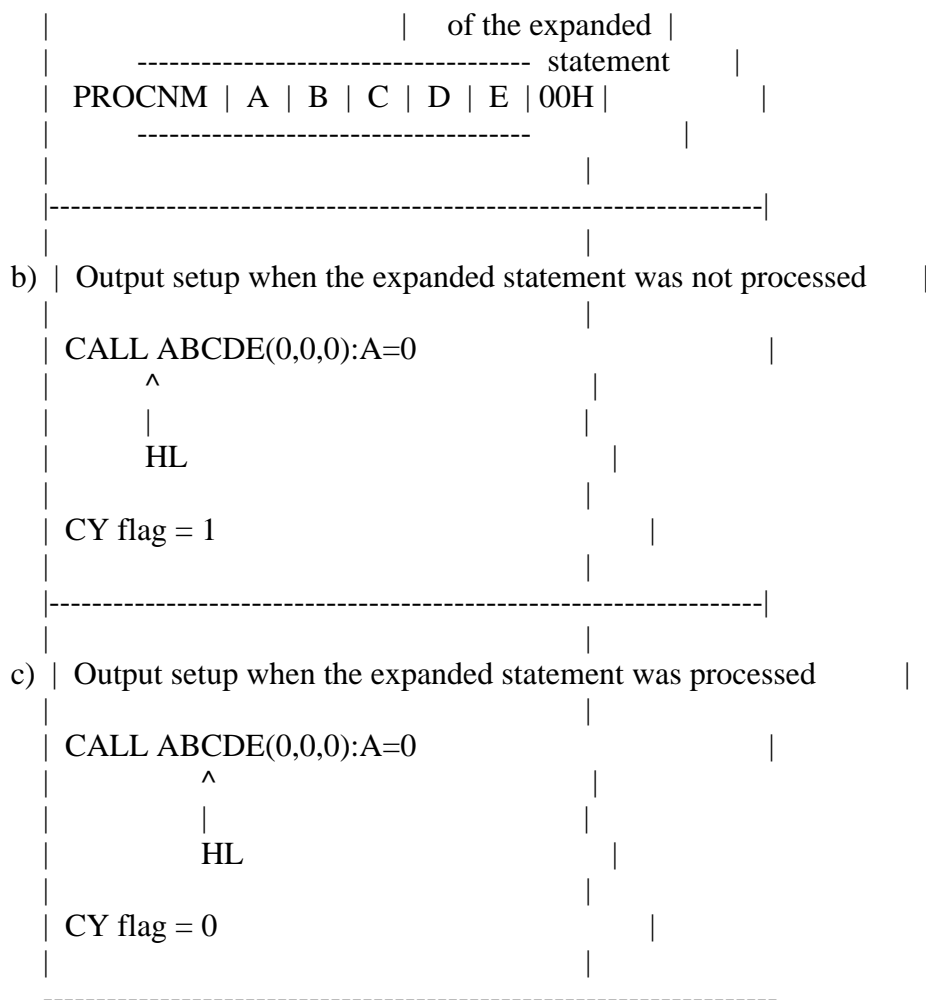
CALL <expression statement name> [(<argument>[, <argument>...])]

The expression statement name can have up to 15 characters. As an abbreviation for CALL, "\_" (underscore) is available.

When the BASIC interpreter finds a CALL statement, it puts the expansion statement name in PROCNM (FD89H, 16) in the work area and passes the control to cartridges, whose contents of STATEMENT of the header is other than 0, in the order from the one with the smaller slot number. The HL register points to the text address next to the expansion statement name at this point (see Figure 5.49a).

Figure 5.49 Input-output of the operation routine of the expansion statement





To develop the statement expansion routine, recognise the name of the expansion statement written in PROCNM first, then return with setting "1" to the carry flag without modifying the HL register if the statement is not to be handled (see Figure 5.49b); otherwise, handle it properly and set the HL register (text pointer) to the next handled statement (where 00H or 3AH is placed usually), then return after setting "0" to the carry flag (see Figure 5.49c).

The BASIC interpreter determines the status of the carry flag whether a CALL statement has been executed, and, if not, calls the next cartridge. When all cartridges have not executed the statement (when the carry flag has been "1" all the time), it displays "SYNTAX ERROR". To test arguments of the statement, it is convenient to use "internal routines for the statement expansion" in section 4.4 of chapter 2.

#### \* DEVICE

These two bytes are the addresses of the device expansion routine, when the cartridge does the device expansion (the input-output device expansion); otherwise 0000H is used. When doing the device expansion, the device expansion routine should be in 4000H-7FFFH. One cartridge can have up to 4 devices. The name of the expansion device should be less than 16 characters.

When the BASIC interpreter finds an undefined device, it stores that in PROCNM (FD89H, 16) and put FFH in the A register and passes control to the cartridge whose contents is not 0 in the order from the one with the smaller

slot number (see Figure 5.50a).

When creating device expansion routines, identify the file descriptor of PROCNM first, and, when it is not for the device to be processed, return with setting 1 to the carry flag (see Figure 5.50b). Otherwise, process it and set the device ID (0-3) in the A register, then return with setting 0 to the carry flag (see Figure 5.50c).

The BASIC interpreter determines by the status of the carry flag whether or not it is processed, and, if not, call the next cartridge. When all cartridges were not processed (that is, when the carry flag was "1" all the time), "Bad file name ERROR" is displayed.

When the actual input-output operations are done, the BASIC interpreter sets the device ID (0-3) in DEVICE (FD99H) and sets the request to the device in the A register (see Table 5.6), then calls the device expansion routine. The device expansion routine should refer to it to handle the request.

Figure 5.50 Input-output to the device expansion routine

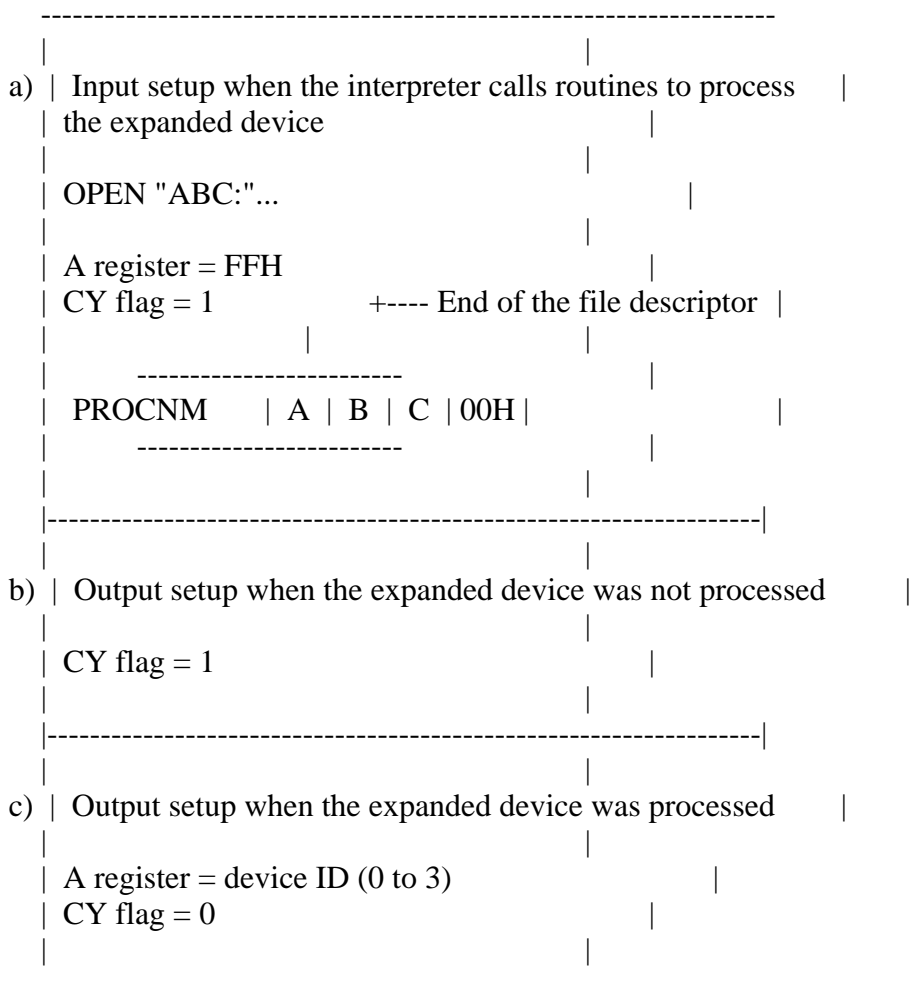


Table 5.6 Requests to the device

Register A	Request
0	OPEN

2	CLOSE
4	Random access
6	Sequential output
8	Sequential input
10	LOC function
12	LOF function
14	EOF function
16	FPOS function
18	Backup character

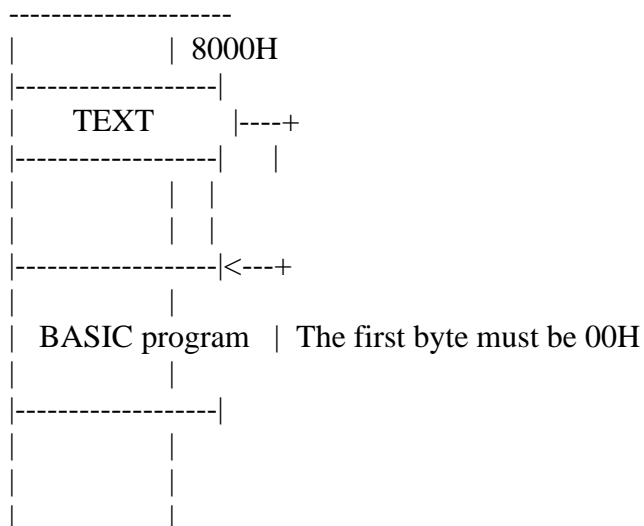
**\* TEXT**

These two bytes are text pointers of the BASIC program, when the BASIC program in the cartridge would be auto-started (executed at reset); otherwise they are 0000H. The size of the program must be under 16K bytes, 8000H to BFFFH.

The BASIC interpreter examines the contents of TEXT of the header after the initialization (INIT) and after the system is started. When they are not 0000H, it begins the execution from the address as BASIC text pointer (see Figure 5.51). BASIC programs should be stored in the form of the intermediate code and the beginning of it (the address pointed by TEXT) must be 00H, which indicates the beginning of the program.

The execution speed of the program will be improved when the objective line number of statements such as GOTO is the absolute address of the objective text pointer.

Figure 5.51 Executing BASIC program cartridge





----- BFFFH

\* How to place BASIC programs in ROM

1. Change the starting address of BASIC text to 8021H.

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

Note: these statements must be in one line

2. Load the objective BASIC program.

```
LOAD "PROGRAM"
```

3. Create ID.

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

4. Put 8000H to BFFFH in ROM.

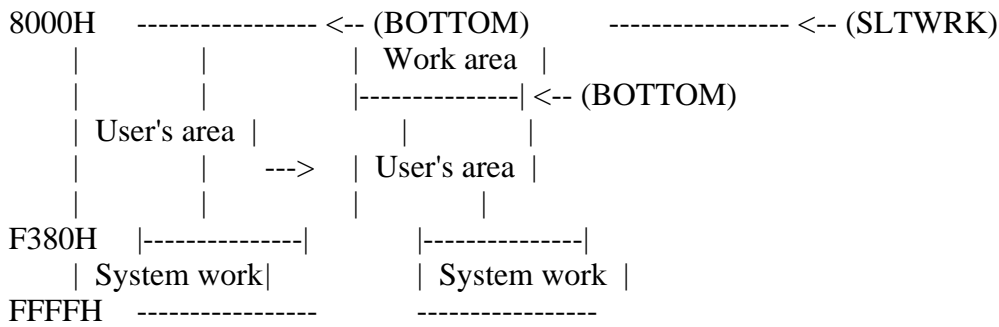
7.3.2 notes on the creation of the cartridge software

In programs not requiring software from other cartridges (stand-alone software such as games), the portion with the smaller address than the work area used by BIOS (F380H) can be used freely.

But in programs which are executed by using BASIC interpreter functions, the same area cannot be shared as the work area. To do this, there are three methods:

- (1) Place RAM on the cartridge itself (the safest and most reliable method).
- (2) When one or two bytes are needed for the work area, use two bytes corresponding to itself in SLTWRK (FD09H to ...) as the work area.
- (3) When more than three bytes are needed for the work area, allocates it from RAM used by BASIC. To do this, put the contents of BOTTOM (FC48H) to the area corresponding to SLTWRK (FD09H to ...), and increase the value of BOTTOM by the needed work area, then allocate it for the work area (see figure 5.52).

Figure 5.52 Allocating the work area



See the following list for the reference of (2) and (3).

### List 5.13 Example of allocating the work area

=====

```

;*****
;
; List 5.13 subroutines to support slot
; for ROM in 1 page
;*****

```

```

RSLGREG EQU 0138H
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)

```

```

; ||+----- secondary 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 tow 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 initialization  
; 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 ;Updae (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

=====

\* Hook

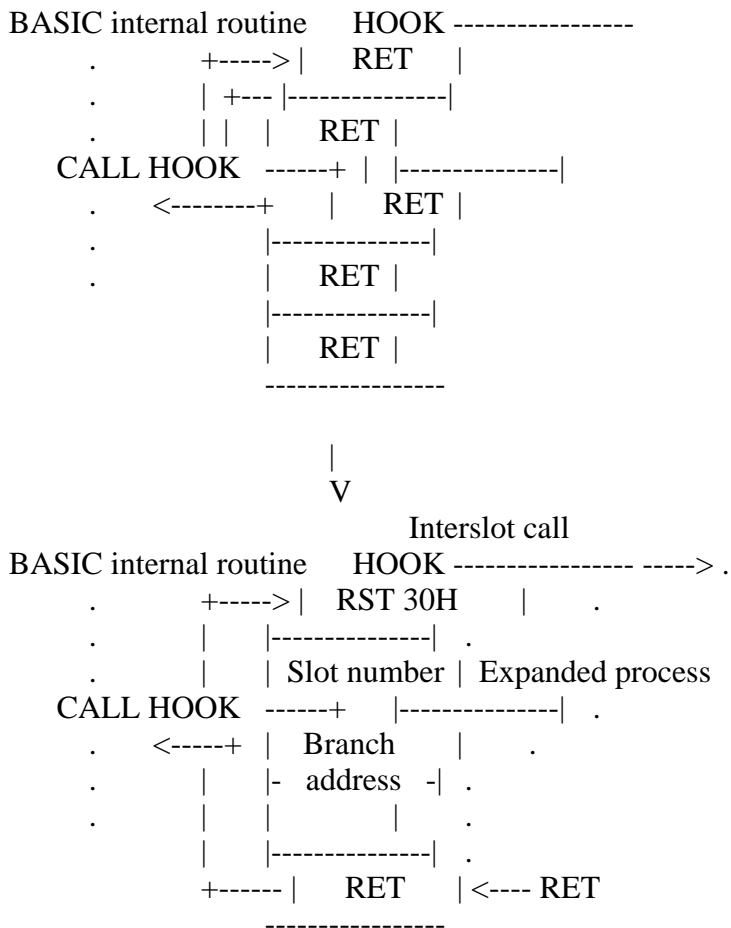
The area called "hook" is used for expanding BASIC functions in FD9AH to FFC9H of the work area used by MSX-BASIC. One hook has five bytes, which are normally "RET".

When MSX-BASIC does a certain operation (such as the one in the description about the hook of the work area), it calls this hook from there once. When the hook is "RET", the control returns immediately; but the function of BASIC can be expanded, when these five bytes were re-written to do the inter-slot

call to the program inside the cartridge by the initialization routine (INIT) (see Figure 5.53).

List 5.14 shows an example of the program that the cartridge uses to hook H.KEYI for the timer interrupt process.

Figure 5.53 Setting the hook



List 5.14 Using the hook

```

=====
;*****
; List 5.14 Sample program to use HOOK
;*****
;
; Start-up initialize entry
; This program will be called when system initializing.
;
H.KEYI EQU 0FD9AH ; interrupt hook
EXPTBL EQU 0FCC1H ; slots expanded or not
PSLTRG EQU 0A8H ; I/O port address of primary slot register
EXTMYINT

INIT:
; <<< Please insert other initialization routine here, if you need. >>>

; Set interrupt entry

```

DI ; start of critical region

; Get old interrupt entry inter-slot call hook

```
LD DE,OLDINT ; get address of old int. hook saved area
LD HL,H.KEYI ; get address of interrupt entry hook
LD BC,5 ; length of hook is 5 bytes
LDIR ; transfer
```

; Which slot address is this cartridge placed?

```
CALL GETMSLT ; get my slot address
```

; Set new inter-slot call of interrupt entry

```
LD (H.KEYI+1),A ; set slot address
LD A,0F7H ; 'RST 30H' inter-slot call operation code
LD (H.KEYI),A ; set new hook op-code
LD HL,INTENT ; get our interrupt entry point
LD (H.KEYI+2),HL ; set new interrupt entry point
LD A,0C9H ; 'RET' operation code
LD (H.KEYI+4),A ; set operation code of 'RET'
EI ; end of critical region
RET
```

;-----

; Which slot address is the cartridge placed?

; Entry: No

; Action: Compute my slot address

; Return: A = slot address

; Modify: Flag

GTMSLT:

```
PUSH BC ; save environment
PUSH HL
IN A,(PSLTRG) ; read primary slot register
RRCA ; move it to bit 0,1 of A
RRCA
AND 00000011B ; get bit 1,0
LD C,A ; set primary slot No.
LD B,0
LD HL,EXPTBL ; see if the slot is expanded or not
ADD HL,BC
OR (HL) ; set MSB if so
LD C,A
INC HL ; point to SLTTBL entry
INC HL
INC HL
INC HL
LD A,(HL) ; get what is currently output to
; expansion slot register

AND 00001100B ; get bits 3,2
OR C ; finally form slot address

POP HL ; restore environment
```

```
POP BC
RET      ; return to main
```

;----- Interrupt entry -----

```
INTENT:
CALL MYINT      ; call interrupt handler
JP  OLDINT      ; go to old interrupt handler
```

;----- HOOK save area -----

```
OLDINT: DS 5
```

```
END
```

=====

### \* Stack pointer initialisation

When MSX has an internal disk, sometimes the disk interface ROM does the initialisation before the cartridge does, depending on the slot location, and pushes down the stack pointer in the direction of the low order address to allocate the work area. In this case, software not using the disk should set the stack pointer again after the cartridge received control; otherwise, the stack area might be exhausted and a system crash might occur. Remember to initialise the stack pointer at the beginning of the program.

### \* Testing the performance of the expansion slot

When general software in the market is put in the expansion slot or when RAM resides in the expansion slot, sometimes the application program do not work. Since most MSX2 machines use the expansion slot inside, problems may sometime result. Software to be sold in the market should be thoroughly tested in both cases that it is put in the expansion slot and that RAM resides in expansion slot.

Since the expansion slot register is placed in FFFFH, do not use it as if it were RAM. For example, setting the stack in FFFFH using "LD SP, 0" in the program causes machines using the expansion slot to go out of control.

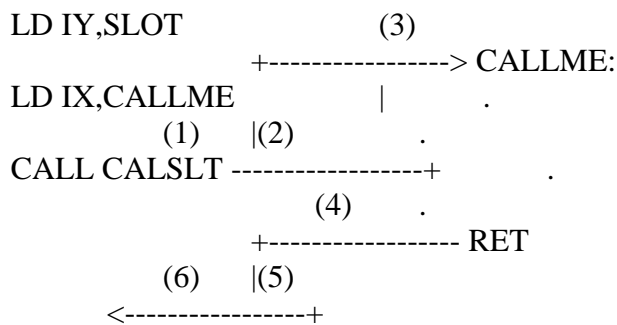
### \* Notes on CALSLT use

Executing the inter-slot call in CALSLT and CALLF destroys the contents of IX, IY, and the background processing register. When returning from this routine, in MSX1 the interrupt is inhibited, but in MSX2 the state before the call is restored.

When using CALSLT or CALLF to execute the inter-slot call, the interrupt is always inhibited when calling the object program (see 2 in the figure below) and when returning to the calling program (see 6 below).

Caller	Interslot call	Called
--------	----------------	--------





In MSX2, the state of the interrupt is reserved before and after the inter-slot call. That is, 3 in the figure is in the same state as 1, and 6 is in the same state as 4. Note when the called program executes "EI" or "DI".

## MSX2 TECHNICAL HANDBOOK

-----

Edited by: ASCII Systems Division  
 Published by: ASCII Corporation - JAPAN  
 First edition: March 1987

Text file typed by: Nestor Soriano (Konami Man) - SPAIN  
 October 1997

### Changes from the original in APPENDIX 1:

- In description of ENASLT, the needed input in HL has been added.
- In description of GETYPR, the Input field has been added.
- In description of INITXT (MAIN), the reference to "INIPLT" has been corrected to "INIPLT".
- In description of SUBROM routine, the mark "\*1" has been erased.
- In description of INITXT (SUB), the needed input in LINL40 has been added.
- Description of PHYDIO routine has been added.

### Changes from the original in APPENDIX 2:

- In the explanation before Figure A.3, the indication about the excess 64 method has been added.
- In Figure A.3, in the third byte, "63rd power of 10" has been corrected to "-63rd power of 10".
- In the explanation before Figure A.3, the indication about the excess 64 method has been added.
- In Figure A.3, in the third byte, "63rd power of 10" has been corrected to "-63rd power of 10".

-----

## APPENDIX 1 - BIOS LISTING

This section lists the 126 BIOS entries available to the user.

There are two kinds of BIOS routines, the ones in MAIN-ROM and the ones in SUB-ROM. They each have different calling sequences which will be described later. The following is the entry notation.

Label name (address) \*n  
Function: descriptions and notes about the function  
Input: parameters used by call  
Output: parameters returned by call  
Registers: registers which will be used (original contents are lost)

The value of \*n has the following meanings.

- \*1 ... same as MSX1
- \*2 ... call SUB-ROM internally in screen modes 5 to 8
- \*3 ... always call SUB-ROM
- \*4 ... do not call SUB-ROM while screen modes 4 to 8 are changed

Routines without "\*n" are appended for MSX2.

### MAIN-ROM

-----

To call routines in MAIN-ROM, the CALL or RTS instruction is used as an ordinary subroutine call.

#### \* RSTs

Among the following RSTs, RST 00H to RST 28H are used by the BASIC interpreter. RST 30H is used for inter-slot calls and RST 38H is used for hardware interrupts.

CHKRAM (0000H) \*1  
Function: tests RAM and sets RAM slot for the system  
Input: none  
Output: none  
Registers: all

SYNCHR (0008H) \*1  
Function: tests whether the character of [HL] is the specified character. If not, it generates SYNTAX ERROR, otherwise it goes to CHRGT (0010H).  
Input: set the character to be tested in [HL] and the character to be compared next to RST instruction which calls this routine (inline parameter).

```

Example: LD HL,LETTER
         RST 08H
         DB "A"
         .
         .
         .
         LETTER: DB "B"

```

Output: HL is increased by one and A receives [HL]. When the tested character is numerical, the CY flag is set; the end of the statement (00H or 3AH) causes the Z flag to be set.

Registers: AF, HL

RDSLTL (000CH) \*1

Function: selects the slot corresponding to the value of A and reads one byte from the memory of the slot. When this routine is called, the interrupt is inhibited and remains inhibited even after execution ends.

Input: A for the slot number.

F000EPPP

- ----

| ||+----- Basic slot number (0 to 3)  
| ++----- Expansion slot number (0 to 3)  
+----- "1" when using expansion slot

HL for the address of memory to be read

Output: the value of memory which has been read in A

Registers: AF, BC, DE

CHRGTR (0010H) \*1

Function: gets a character (or a token) from BASIC text

Input: [HL] for the character to be read

Output: HL is incremented by one and A receives [HL]. When the character is numerical, the CY flag is set; the end of the statement causes the Z flag to be set.

Registers: AF, HL

WRSLTL (0014H) \*1

Function: selects the slot corresponding to the value of A and writes one byte to the memory of the slot. When this routine is called, interrupts are inhibited and remain so even after execution ends.

Input: specifies a slot with A (same as RDSLTL)

Output: none

Registers: AF, BC, D

OUTDO (0018H) \*2

Function: sends the value to current device

Input: A for the value to be sent

sends output to the printer when PTRFLG (F416H) is other than 0

sends output to the file specified by PTRFIL (F864H) when

PTRFIL is other than 0

Output: none

Registers: none

CALSLT (001CH) \*1

Function: calls the routine in another slot (inter-slot call)

Input: specify the slot in the 8 high order bits of the IY register (same as RDSLTL). IX is for the address to be called.

Output: depends on the calling routine

Registers: depends on the calling routine

DCOMPR (0020H) \*1

Function: compares the contents of HL and DE

Input: HL, DE

Output: sets the Z flag for HL = DE, CY flag for HL < DE

Registers: AF

ENASLT (0024H) \*1

Function: selects the slot corresponding to the value of A and enables the slot to be used. When this routine is called, interrupts are inhibited and remain so even after execution ends.

Input: specify the slot by A (same as RDSLTL)  
specify the page to switch the slot by 2 high order bits of HL

Output: none

Registers: all

GETYPR (0028H) \*1

Function: returns the type of DAC (decimal accumulator)

Input: none

Output: S, Z, P/V flags are changed depending on the type of DAC:

integer type	single precision real type
C = 1	C = 1
S = 1 *	S = 0
Z = 0	Z = 0
P/V = 1	P/V = 0 *

string type	double precision real type
C = 1	C = 0 *
S = 0	S = 0
Z = 1 *	Z = 0
P/V = 1	P/V = 1

Types can be recognised by the flag marked by "\*".

Registers: AF

CALLF (0030H) \*1

Function: calls the routine in another slot. The following is the calling sequence:

RST 30H

DB n ;n is the slot number (same as RDSLT)  
DW mn ;nn is the called address

Input: In the method described above

Output: depends on the calling routine

Registers: AF, and other registers depending on the calling routine

KEYINT (0038H) \*1

Function: executes the timer interrupt process routine

Input: none

Output: none

Register: none

\* I/O initialisation

INITIO (003BH) \*1

Function: initialises the device

Input: none

Output: none

Registers: all

INIFNK (003EH) \*1

Function: initialises the contents of function keys

Input: none

Output: none

Registers: all

\* VDP access

DISSCR (0041H) \*1

Function: inhibits the screen display

Input: none

Output: none

Registers: AF, BC

ENASCR (0044H) \*1

Function: displays the screen

Input: none

Output: none

Registers: all

WRTVDP (0047H) \*2

Function: writes data in the VDP register

Input: C for the register number, B for data; the register number  
is 0 to 23 and 32 to 46

Output: none

Registers: AF, BC

#### RDVRM (004AH) \*1

Function: reads the contents of VRAM. This is for TMS9918, so only the 14 low order bits of the VRAM address are valid. To use all bits, call NRDVRM.

Input: HL for VRAM address to be read

Output: A for the value which was read

Registers: AF

#### WRTVRM (004DH) \*1

Function: writes data in VRAM. This is for TMS9918, so only the 14 low order bits of the VRAM address are valid. To use all bits, call NWRVRM.

Input: HL for VRAM address, A for data

Output: none

Registers: AF

#### SETRD (0050H) \*1

Function: sets VRAM address to VDP and enables it to be read. This is used to read data from the sequential VRAM area by using the address auto-increment function of VDP. This enables faster readout than using RDVRM in a loop. This is for TMS9918, so only the 14 low order bits of VRAM address are valid. To use all bits, call NSETRD.

Input: HL for VRAM address

Output: none

Registers: AF

#### SETWRT (0053H) \*1

Function: sets VRAM address to VDP and enables it to be written. The purpose is the same as SETRD. This is for TMS9918, so only the 14 low order bits of VRAM address are valid. To use all bits, call NSETRD.

Input: HL for VRAM address

Output: none

Registers: AF

#### FILVRM (0056H) \*4

Function: fills the specified VRAM area with the same data. This is for TMS9918, so only the 14 low order bits of the VRAM address are valid. To use all bits, see BIGFIL.

Input: HL for VRAM address to begin writing, BC for the length of the area to be written, A for data.

Output: none

Registers: AF, BC

#### LDIRMV (0059H) \*4

Function: block transfer from VRAM to memory

Input: HL for source address (VRAM), DE for destination address (memory), BC for the length. All bits of the VRAM address are valid.

Output: none

Registers: all

#### LDIRVM (005CH) \*4

Function: block transfer from memory to VRAM

Input: HL for source address (memory), DE for destination address (VRAM), BC for the length. All bits of the VRAM address are valid.

Output: none

Registers: all

#### CHGMOD (005FH) \*3

Function: changes the screen mode. The palette is not initialised. To initialise it, see CHGMDP in SUB-ROM.

Input: A for the screen mode (0 to 8)

Output: none

Registers: all

#### CHGCLR (0062H) \*1

Function: changes the screen colour

Input: A for the mode

FORCLR (F3E9H) for foreground color

BAKCLR (F3EAH) for background color

BDRCLR (F3EBH) for border colour

Output: none

Registers: all

#### NMI (0066H) \*1

Function: executes NMI (Non-Maskable Interrupt) handling routine

Input: none

Output: none

Registers: none

#### CLRSPR (0069H) \*3

Function: initialises all sprites. The sprite pattern is cleared to null, the sprite number to the sprite plane number, the sprite colour to the foreground colour. The vertical location of the sprite is set to 209 (mode 0 to 3) or 217 (mode 4 to 8).

Input: SCRMOD (FCAFH) for the screen mode

Output: none

Registers: all

#### INITXT (006CH) \*3

Function: initialises the screen to TEXT1 mode (40 x 24). In this routine, the palette is not initialised. To initialise the palette, call INIPLT in SUB-ROM after this call.

Input: TXTNAM (F3B3H) for the pattern name table

TXTCGP (F3B7H) for the pattern generator table

LINL40 (F3AEH) for the length of one line

Output: none

Registers: all

INIT32 (006FH) \*3

Function: initialises the screen to GRAPHIC1 mode (32x24). In this routine, the palette is not initialised.

Input: T32NAM (F3BDH) for the pattern name table  
T32COL (F3BFH) for the colour table  
T32CGP (F3C1H) for the pattern generator table  
T32ATR (F3C3H) for the sprite attribute table  
T32PAT (F3C5H) for the sprite generator table

Output: none

Registers: all

INIGRP (0072H) \*3

Function: initialises the screen to the high-resolution graphics mode. In this routine, the palette is not initialised.

Input: GRPNAM (F3C7H) for the pattern name table  
GRPCOL (F3C9H) for the colour table  
GRPCGP (F3CBH) for the pattern generator table  
GRPATR (F3CDH) for the sprite attribute table  
GRPPAT (F3CFH) for the sprite generator table

Output: none

Registers: all

INIMLT (0075H) \*3

Function: initialises the screen to MULTI colour mode. In this routine, the palette is not initialised.

Input: MLTNAM (F3D1H) for the pattern name table  
MLTCOL (F3D3H) for the colour table  
MLTCGP (F3D5H) for the pattern generator table  
MLTATR (F3D7H) for the sprite attribute table  
MLTPAT (F3D9H) for the sprite generator table

Output: none

Registers: all

SETTXT (0078H) \*3

Function: set only VDP in TEXT1 mode (40x24)

Input: same as INITXT

Output: none

Registers: all

SETT32 (007BH) \*3

Function: set only VDP in GRAPHIC1 mode (32x24)

Input: same as INIT32

Output: none

Registers: all

SETGRP (007EH) \*3

Function: set only VDP in GRAPHIC2 mode

Input: same as INIGRP

Output: none

Registers: all



SETMLT (0081H) \*3

Function: set only VDP in MULTI colour mode

Input: same as INIMLT

Output: none

Registers: all

CALPAT (0084H) \*1

Function: returns the address of the sprite generator table

Input: A for the sprite number

Output: HL for the address

Registers: AF, DE, HL

CALATR (0087H) \*1

Function: returns the address of the sprite attribute table

Input: A for the sprite number

Output: HL for the address

Registers: AF, DE, HL

GSPSIZ (008AH) \*1

Function: returns the current sprite size

Input: none

Output: A for the sprite size (in bytes). Only when the size is 16 x 16, the CY flag is set; otherwise the CY flag is reset.

Registers: AF

GRPPRT (008DH) \*2

Function: displays a character on the graphic screen

Input: A for the character code. When the screen mode is 0 to 8, set the logical operation code in LOGOPR (FB02H).

Output: none

Registers: none

\* PSG

GICINI (0090H) \*1

Function: initialises PSG and sets the initial value for the PLAY statement

Input: none

Output: none

Registers: all

WRTPSG (0093H) \*1

Function: writes data in the PSG register

Input: A for PSG register number, E for data

Output: none

Registers: none

RDPSG (0096H) \*1

Function: reads the PSG register value  
Input: A for PSG register number  
Output: A for the value which was read  
Registers: none

STRTMS (0099H) \*1

Function: tests whether the PLAY statement is being executed as a background task. If not, begins to execute the PLAY statement  
Input: none  
Output: none  
Registers: all

\* Keyboard, CRT, printer input-output

CHSNS (009CH) \*1

Function: tests the status of the keyboard buffer  
Input: none  
Output: the Z flag is set when the buffer is empty, otherwise the Z flag is reset  
Registers: AF

CHGET (009FH) \*1

Function: one character input (waiting)  
Input: none  
Output: A for the code of the input character  
Registers: AF

CHPUT (00A2H) \*1

Function: displays the character  
Input: A for the character code to be displayed  
Output: none  
Registers: none

LPTOUT (00A5H) \*1

Function: sends one character to the printer  
Input: A for the character code to be sent  
Output: if failed, the CY flag is set  
Registers: F

LPTSTT (00A8H) \*1

Function: tests the printer status  
Input: none  
Output: when A is 255 and the Z flag is reset, the printer is READY.  
when A is 0 and the Z flag is set, the printer is NOT READY.  
Registers: AF

CNVCHR (00ABH) \*1

Function: test for the graphic header and transforms the code

Input: A for the character code  
Output: the CY flag is reset to not the graphic header  
the CY flag and the Z flag are set to the transformed code  
is set in A  
the CY flag is set and the CY flag is reset to the  
untransformed code is set in A  
Registers: AF

PINLIN (00AEH) \*1  
Function: stores in the specified buffer the character codes input  
until the return key or STOP key is pressed.  
Input: none  
Output: HL for the starting address of the buffer minus 1, the CY  
flag is set only when it ends with the STOP key.  
Registers: all

INLIN (00B1H) \*1  
Function: same as PINLIN except that AUTFLG (F6AAH) is set  
Input: none  
Output: HL for the starting address of the buffer minus 1, the CY  
flag is set only when it ends with the STOP key.  
Registers: all

QINLIN (00B4H) \*1  
Function: executes INLIN with displaying "?" and one space  
Input: none  
Output: HL for the starting address of the buffer minus 1, the CY  
flag is set only when it ends with the STOP key.  
Registers: all

BREAKX (00B7H) \*1  
Function: tests Ctrl-STOP key. In this routine, interrupts are  
inhibited.  
Input: none  
Output: the CY flag is set when pressed  
Registers: AF

BEEP (00C0H) \*3  
Function: generates BEEP  
Input: none  
Output: none  
Registers: all

CLS (00C3H) \*3  
Function: clears the screen  
Input: set zero flag  
Output: none  
Registers: AF, BC, DE

POSIT (00C6H) \*1

Function: moves the cursor  
Input: H for the X-coordinate of the cursor, L for the Y-coordinate  
Output: none  
Registers: AF

FNKSB (00C9H) \*1

Function: tests whether the function key display is active (FNKFLG).  
If so, displays them, otherwise erases them.  
Input: FNKFLG (FBCEH)  
Output: none  
Registers: all

ERAFNK (00CCH) \*1

Function: erases the function key display  
Input: none  
Output: none  
Registers: all

DSPFNK (00CFH) \*2

Function: displays the function keys  
Input: none  
Output: none  
Registers: all

TOTEXT (00D2H) \*1

Function: forces the screen to be in the text mode  
Input: none  
Output: none  
Registers: all

\* Game I/O access

GTSTCK (00D5H) \*1

Function: returns the joystick status  
Input: A for the joystick number to be tested  
Output: A for the joystick direction  
Registers: all

GTTRIG (00D8H) \*1

Function: returns the trigger button status  
Input: A for the trigger button number to be tested  
Output: When A is 0, the trigger button is not being pressed.  
When A is FFH, the trigger button is being pressed.  
Registers: AF

GTPAD (00DBH) \*1

Function: returns the touch pad status  
Input: A for the touch pad number to be tested

Output: A for the value  
Registers: all

GTPDL (00DEH) \*2  
Function: returns the paddle value  
Input: A for the paddle number  
Output: A for the value  
Registers: all

\* Cassette input-output routine

TAPION (00E1H) \*1  
Function: reads the header block after turning the cassette motor ON.  
Input: none  
Output: if failed, the CY flag is set  
Registers: all

TAPIN (00E4H) \*1  
Function: reads data from the tape  
Input: none  
Output: A for data. If failed, the CY flag is set.  
Registers: all

TAPIOF (00E7H) \*1  
Function: stops reading the tape  
Input: none  
Output: none  
Registers: none

TAPOON (00EAH) \*1  
Function: writes the header block after turning the cassette motor ON  
Input: A = 0, short header; A <> 0, long header  
Output: if failed, the CY flag is set  
Registers: all

TAPOUT (00EDH) \*1  
Function: writes data on the tape  
Input: A for data  
Output: if failed, the CY flag is set  
Registers: all

TAPOOF (00F0H) \*1  
Function: stops writing to the tape  
Input: A for data  
Output: if failed, the CY flag is set  
Registers: all

STMOTR (00F3H) \*1  
Function: sets the cassette motor action  
Input: A = 0 -> stop  
A = 1 -> start  
A = 0FFH -> reverse the current action  
Output: none  
Registers: AF

\* Miscellaneous

CHGCAP (0132H) \*1  
Function: alternates the CAP lamp status  
Input: A = 0 -> lamp off  
A <> 0 -> lamp on  
Output: none  
Registers: AF

CHGSND (0135H) \*1  
Function: alternates the 1-bit sound port status  
Input: A = 0 -> OFF  
A <> 0 -> ON  
Output: none  
Registers: AF

RSLREG (0138H) \*1  
Function: reads the contents of current output to the basic slot  
register  
Input: none  
Output: A for the value which was read  
Registers: A

WSLREG (013BH) \*1  
Function: writes to the primary slot register  
Input: A for the value to be written  
Output: none  
Registers: none

RDVDP (013EH) \*1  
Function: reads VDP status register  
Input: none  
Output: A for the value which was read  
Registers: A

SNSMAT (0141H) \*1  
Function: reads the value of the specified line from the keyboard  
matrix  
Input: A for the specified line  
Output: A for data (the bit corresponding to the pressed key will  
be 0)

Registers: AF, C

#### PHYDIO (0144H)

Function: Physical input/output for disk devices

Input: A for the drive number (0 = A:, 1 = B:,...)

B for the number of sector to be read from or written to

C for the media ID

DE for the first sector number to be read from or written to

HL for the starting address of the RAM buffer to be read from or written to specified sectors

CY set for sector writing; reset for sector reading

Output: CY set if failed

B for the number of sectors actually read or written

A for the error code (only if CY set):

0 = Write protected

2 = Not ready

4 = Data error

6 = Seek error

8 = Record not found

10 = Write error

12 = Bad parameter

14 = Out of memory

16 = Other error

Registers: all

#### ISFLIO (014AH) \*1

Function: tests whether the device is active

Input: none

Output: A = 0 -> active

A <> 0 -> inactive

Registers: AF

#### OUTDLP (014DH) \*1

Function: printer output. Different from LPTOUT in the following points:

1. TAB is expanded to spaces
2. For non-MSX printers, hiragana is transformed to katakana and graphic characters are transformed to 1-byte characters.
3. If failed, device I/O error occurs.

Input: A for data

Output: none

Registers: F

#### KILBUF (0156H) \*1

Function: clears the keyboard buffer

Input: none

Output: none

Registers: HL

#### CALBAS (0159H) \*1

Function: executes inter-slot call to the routine in BASIC interpreter

Input: IX for the calling address

Output: depends on the called routine  
Registers: depends on the called routine

\* Entries appended for MSX2

#### SUBROM (015CH)

Function: executes inter-slot call to SUB-ROM  
Input: IX for the calling address and, at the same time, pushes IX on the stack  
Output: depends on the called routine  
Registers: background registers and IY are reserved

#### EXTROM (015FH)

Function: executes inter-slot call to SUB-ROM  
Input: IX for the calling address  
Output: depends on the called routine  
Registers: background registers and IY are reserved

#### EOL (0168H)

Function: deletes to the end of the line  
Input: H for X-coordinate of the cursor, L for Y-coordinate  
Output: none  
Registers: all

#### BIGFIL (016BH)

Function: same function as FILVRM. Differences are as follows:  
In FILVRM, it is tested whether the screen mode is 0 to 3.  
If so, it treats VDP as the one which has only 16K bytes VRAM (for the compatibility with MSX1). In BIGFIL, the mode is not tested and actions are carried out by the given parameters.  
Input: same as FILVRM  
Output: same as FILVRM  
Registers: same as FILVRM

#### NSETRD (016EH)

Function: enables VRAM to be read by setting the address  
Input: HL for VRAM address  
Output: none  
Registers: AF

#### NSTWRT (0171H)

Function: enables VRAM to be written by setting the address  
Input: HL for VRAM address  
Output: none  
Registers: AF

#### NRDVRM (0174H)



Function: reads the contents of VRAM  
Input: HL for VRAM address to be read  
Output: A for the value which was read  
Registers: F

#### NWRVRM (0177H)

Function: writes data in VRAM  
Input: HL for VRAM address, A for data  
Output: none  
Registers: AF

#### SUB-ROM

---

The calling sequence of SUB-ROM is as follows:

```
.  
. .  
LD IX, INIPLT  
      ; Set BIOS entry address  
CALL EXTROM  
      ; Returns here  
. . .
```

When the contents of IX should not be destroyed, use the call as shown below.

```
. . .  
INIPAL: PUSH IX  
          ; Save IX  
LD IX, INIPLT  
          ; Set BIOS entry address  
JP SUBROM  
          ;Returns caller of INIPAL  
. . .
```

#### GRPRT (0089H)

Function: one character output to the graphic screen (active only in screen modes 5 to 8)  
Input: A for the character code  
Output: none  
Registers: none

#### NVBXLN (00C9H)

Function: draws a box  
Input: start point: BC for X-coordinate, DE for Y-coordinate

end point: GXPOS (FCB3H) for X-coordinate  
GYPOS (FCB5H) for Y-coordinate  
colour: ATRBYT (F3F3H) for the attribute  
logical operation code: LOGOPR (FB02H)

Output: none

Registers: all

#### NVBXFL (00CDH)

Function: draws a painted box

Input: start point: BC for X-coordinate, DE for Y-coordinate

end point: GXPOS (FCB3H) for X-coordinate

GYPOS (FCB5H) for Y-coordinate

colour: ATRBYT (F3F3H) for the attribute

logical operation code: LOGOPR (FB02H)

Output: none

Registers: all

#### CHGMOD (00D1H)

Function: changes the screen mode

Input: A for the screen mode (0 to 8)

Output: none

Registers: all

#### INITXT (00D5H)

Function: initialises the screen to TEXT1 mode (40 x 24)

Input: TXTNAM (F3B3H) for the pattern name table

TXTCGP (F3B7H) for the pattern generator table

LINL40 (F3AEH) for the length of one line

Output: none

Registers: all

#### INIT32 (00D9H)

Function: initialises the screen to GRAPHIC1 mode (32x24)

Input: T32NAM (F3BDH) for the pattern name table

T32COL (F3BFH) for the colour table

T32CGP (F3C1H) for the pattern generator table

T32ATR (F3C3H) for the sprite attribute table

T32PAT (F3C5H) for the sprite generator table

Output: none

Registers: all

#### INIGRP (00DDH)

Function: initialises the screen to the high-resolution graphics mode

Input: GRPNAM (F3C7H) for the pattern name table

GRPCOL (F3C9H) for the colour table

GRPCGP (F3CBH) for the pattern generator table

GRPATR (F3CDH) for the sprite attribute table

GRPPAT (F3CFH) for the sprite generator table

Output: none

Registers: all

#### INIMLT (00E1H)

Function: initialises the screen to MULTI colour mode

Input: MLTNAM (F3D1H) for the pattern name table  
MLTCOL (F3D3H) for the colour table  
MLTCGP (F3D5H) for the pattern generator table  
MLTATR (F3D7H) for the sprite attribute table  
MLTPAT (F3D9H) for the sprite generator table

Output: none

Registers: all

#### SETTXT (00E5H)

Function: sets VDP in the text mode (40x24)

Input: same as INITXT

Output: none

Registers: all

#### SETT32 (00E9H)

Function: sets VDP in the text mode (32x24)

Input: same as INIT32

Output: none

Registers: all

#### SETGRP (00EDH)

Function: sets VDP in the high-resolution mode

Input: same as INIGRP

Output: none

Registers: all

#### SETMLT (00F1H)

Function: sets VDP in MULTI COLOUR mode

Input: same as INIMLT

Output: none

Registers: all

#### CLRSPR (00F5H)

Function: initialises all sprites. The sprite pattern is set to null, sprite number to sprite plane number, and sprite colour to the foreground colour. The vertical location of the sprite is set to 217.

Input: SCRMOD (FCAFH) for the screen mode

Output: none

Registers: all

#### CALPAT (00F9H)

Function: returns the address of the sprite generator table  
(this routine is the same as CALPAT in MAIN-ROM)

Input: A for the sprite number

Output: HL for the address

Registers: AF, DE, HL

#### CALATR (00FDH)

Function: returns the address of the sprite attribute table  
(this routine is the same as CALATR in MAIN-ROM)

Input: A for the sprite number

Output: HL for the address

Registers: AF, DE, HL

#### GSPSIZ (0101H)

Function: returns the current sprite size  
(this routine is the same as GSPSIZ in MAIN-ROM)

Input: none

Output: A for the sprite size. The CY flag is set only for the size  
16 x 16.

Registers: AF

#### GETPAT (0105H)

Function: returns the character pattern

Input: A for the character code

Output: PATWRK (FC40H) for the character pattern

Registers: all

#### WRTVRM (0109H)

Function: writes data in VRAM

Input: HL for VRAM address (0 TO FFFFH), A for data

Output: none

Registers: AF

#### RDVRM (010DH)

Function: reads the contents of VRAM

Input: HL for VRAM address (0 TO FFFFH) to be read

Output: A for the value which was read

Registers: AF

#### CHGCLR (0111H)

Function: changes the screen colour

Input: A for the mode

FORCLR (F3E9H) for the foreground color

BAKCLR (F3EAH) for the background color

BDRCLR (F3EBH) for the border colour

Output: none

Registers: all

#### CLSSUB (0115H)

Function: clears the screen

Input: none

Output: none

Registers: all

#### DSPFNK (011DH)

Function: displays the function keys

Input: none  
Output: none  
Registers: all

#### WRTVDP (012DH)

Function: writes data in the VDP register  
Input: C for the register number, B for data  
Output: none  
Registers: AF, BC

#### VDPSTA (0131H)

Function: reads the VDP register  
Input: A for the register number (0 to 9)  
Output: A for data  
Registers: F

#### SETPAG (013DH)

Function: switches the page  
Input: DPPAGE (FAF5H) for the display page number  
ACPAGE (FAF6H) for the active page number  
Output: none  
Registers: AF

#### INIPLT (0141H)

Function: initialises the palette(the current palette is saved in VRAM)  
Input: none  
Output: none  
Registers: AF, BC, DE

#### RSTPLT (0145H)

Function: restores the palette from VRAM  
Input: none  
Output: none  
Registers: AF, BC, DE

#### GETPLT (0149H)

Function: obtains the colour code from the palette  
Input: D for the palette number (0 to 15)  
Output: 4 high order bits of B for red code  
4 low order bits of B for blue code  
4 low order bits of C for green code  
Registers: AF, DE

#### SETPLT (014DH)

Function: sets the colour code to the palette  
Input: D for the palette number (0 to 15)  
4 high order bits of A for red code  
4 low order bits of A for blue code  
4 low order bits of E for green code  
Output: none

Registers: AF

#### BEEP (017DH)

Function: generates BEEP

Input: none

Output: none

Registers: all

#### PROMPT (0181H)

Function: displays the prompt

Input: none

Output: none

Registers: all

#### NEWPAD (01ADH)

Function: reads the status of mouse or light pen

Input: call with setting the following data in A;  
descriptions in parenthesis are return values.

8 ..... light pen check (valid at 0FFH)

9 ..... returns X-coordinate

10 ..... returns Y-coordinate

11 ..... returns the light pen switch status  
(0FFH, when pressed)

12 ..... whether the mouse is connected to the  
port 1 (valid at 0FFH)

13 ..... returns the offset in X direction

14 ..... returns the offset in Y direction

15 ..... (always 0)

16 ..... whether the mouse is connected to the  
port 2 (valid at 0FFH)

17 ..... returns the offset in X direction

18 ..... returns the offset in Y direction

19 ..... (always 0)

Output: A

Registers: all

#### CHGMDP (01B5H)

Function: changes VDP mode. The palette is initialised.

Input: A for the screen mode (0 to 8)

Output: none

Registers: all

#### KNJPRT (01BDH)

Function: sends a kanki to the graphic screen (modes 5 to 8)

Input: BC for JIS kanji code, A for the display mode. The display mode has the following meaning, similar to the PUT KANJI command of BASIC.

0 ..... display in 16 x 16 dot

1 ..... display even dots

2 ..... display odd dots

REDCLK (01F5H)

Function: reads the clock data  
Input: C for RAM address of the clock

00MMAAAA

-----

||++++--- Address (0 to 15)

++----- Mode (0 to 3)

Output: A for the data which were read (only 4 low order bits are valid)

Registers: F

WRTCLK (01F9H)

Function: writes the clock data  
Input: A for the data to be written, C for RAM address of the clock

Output: none

Registers: F

=====

Changes from the original in APPENDIX 2:

- In the explanation before Figure A.3, the indication about the excess 64 method has been added.
- In Figure A.3, in the third byte, "63rd power of 10" has been corrected to "-63rd power of 10".
- In the explanation before Figure A.3, the indication about the excess 64 method has been added.
- In Figure A.3, in the third byte, "63rd power of 10" has been corrected to "-63rd power of 10".

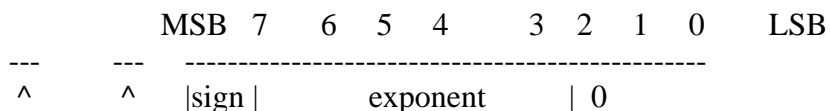
-----

APPENDIX 2 - MATH-PACK

The Math-Pack is the core for the mathematical routines of MSX-BASIC and, by calling these routines from an assembly language program, floating-point operations and trigonometrical functions are available.

Any operations involving real numbers in Math-Pack are done in BCD (Binary Coded Decimal). There are two ways of expressing a real number, "single precision" and "double precision"; a single precision real number (6 digits) is expressed by 4 bytes and a double precision real number (14 digits) by 8 bytes (see Figure A.1 and Figure A.2).

Figure A.1 BCD format for expressing real numbers



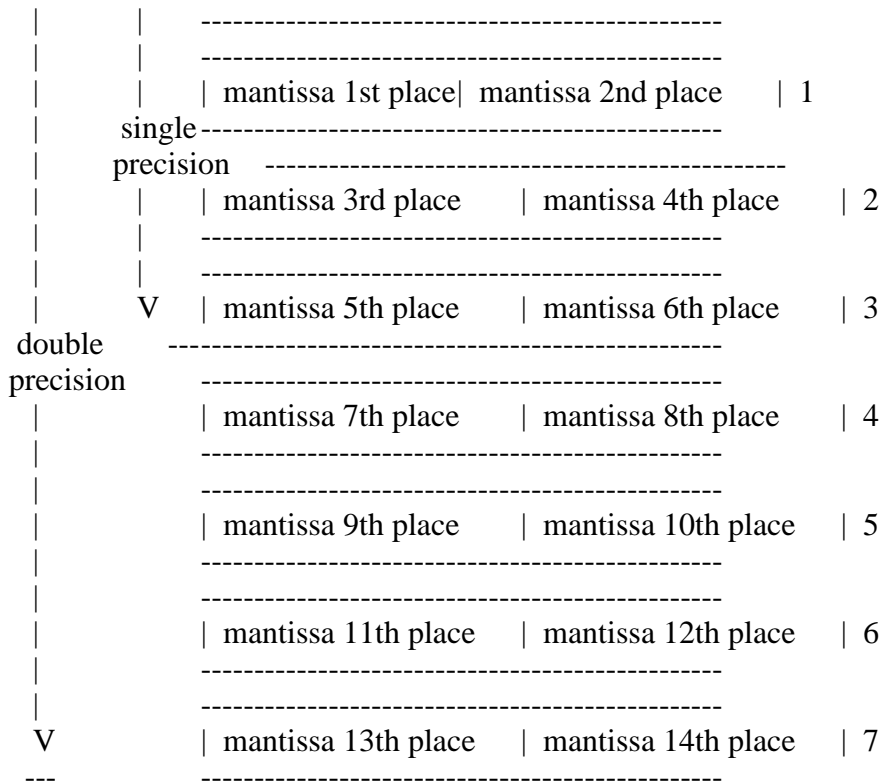
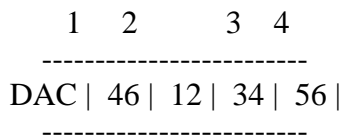


Figure A.2 Examples of expressions for real numbers

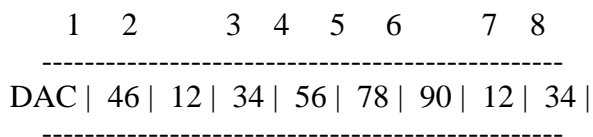
Example of the single precision expression

123456 --> 0.123456 E+6



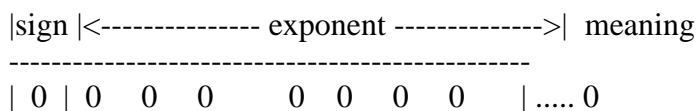
Example of the double precision expression

123456.78901234 --> 0.12345678901234 E+6



A real number consists of a sign, an exponent, and a mantissa. The sign represents the sign of the mantissa; 0 for positive, 1 for negative. The exponent is a binary expression and can be expressed as a power from +63 to -63, with an excess of 64 (see Figure A.3). Figure A.4 shows the valid range of double precision real numbers.

Figure A.3 Exponent format





```

-----
-----
| 1 | 0 0 0 0 0 0 0 0 | ..... undefined (-0?)
-----
-----
| x | 0 0 0 0 0 0 0 1 | ..... -63rd power of 10
-----
-----
| x | 1 0 0 0 0 0 0 0 | ..... 0th power of 10
-----
-----
| x | 1 1 1 1 1 1 1 1 | ..... +63rd power of 10
-----

```

Note: "x" is 1 or 0, both of which are allowed.

Figure A.4 Valid range for double precision real numbers

```

  7 6 5 4 3 2 1 0 (byte)
-----
DAC | FF | 99 | 99 | 99 | 99 | 99 | 99 | 99 | -0.9999999999999999 E+63
-----
      .
      .
      .
-----
| 81 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | -0.1000000000000000 E-63
-----
-----
| 00 | x  | | x | x | x | x  | x | x | 0
-----
-----
| 01 | 10 | 00 | 00 | 00 | 00 | 00 | 00 | +0.1000000000000000 E-63
-----
      .
      .
      .
-----
| 7F | 99 | 99 | 99 | 99 | 99 | 99 | 99 | +0.9999999999999999 E+63
-----

```

In Math-Pack, the memory is predefined for operation. This memory area is called "DAC (Decimal ACumulator (F7F6H))" and the area which reserves the numerical value to be operated is called "ARG (F847H)". For example, in multiplication, the product of the numbers in DAC and ARG is calculated and the result is returned in the DAC.

In the DAC, single precision real numbers, double precision real numbers, and two-byte integers can be stored. In order to distinguish them, "VALTYP (F663H)" is used and its value is 4 for single precision real numbers, 8 for double precision real numbers, and 2 for two-byte integers.

Single and double precision numbers must be stored from the top of the DAC. For two-byte integers, the low and high bytes should be stored in DAC + 2 and DAC + 3.

Since Math-Pack is an internal routine of BASIC, when an error occurs (such as division by 0 or overflow), control automatically jumps to the corresponding error routine, then returns to BASIC command level. To prevent this, change H.ERRO (FFB1H).

\* Math-Pack work area

Label	Address	Size	Meaning
VALTYP	F663H	1	format of the number in DAC
DAC	F7F6H	16	floating point accumulator in BCD format
ARG	F847H	16	argument of DAC

\* Math-Pack entry

Basic operation

Label	Address	Function
DECSUB	268CH	DAC <-- DAC - ARG
DECADD	269AH	DAC <-- DAC + ARG
DECNRM	26FAH	normalises DAC (*1)
DECROU	273CH	rounds DAC
DECMUL	27E6H	DAC <-- DAC * ARG
DECDIV	289FH	DAC <-- DAC / ARG

Note: These operations treat numbers in DAC and ARG as the double precision number. Registers are not preserved.

\*1 Excessive zeros in mantissa are removed. (0.00123 --> 0.123 E-2)

Function 1

Label	Address	Function	Register modified
COS	2993H	DAC <-- COS(DAC)	all
SIN	29ACH	DAC <-- SIN(DAC)	all
TAN	29FBH	DAC <-- TAN(DAC)	all
ATN	2A14H	DAC <-- ATN(DAC)	all
LOG	2A72H	DAC <-- LOG(DAC)	all
SQR	2AFFH	DAC <-- SQR(DAC)	all
EXP	2B4AH	DAC <-- EXP(DAC)	all
RND	2BDFH	DAC <-- RND(DAC)	all

Note: These processing routines all have the same function names as those in BASIC. "All" registers are A, B, C, D, E, H, and L.

## Function 2

Label	Address	Function	Register modified
SIGN	2E71H	A <-- sign of DAC	A
ABSFN	2E82H	DAC <-- ABS(DAC)	all
NEG	2E8DH	DAC <-- NEG(DAC)	A,HL
SGN	2E97H	DAC <-- SGN(DAC)	A,HL

Note: Except for SIGN, these processing routines all have the same function names as those in BASIC. Registers are A, B, C, D, E, H, and L.  
 Note that for SGN, the result is represented as a 2-byte integer.

## Movement

Label	Address	Function	Object	Reg. mod.
MAF	2C4DH	ARG <-- DAC	double prec.	A,B,D,E,H,L
MAM	2C50H	ARG <-- (HL)	double prec.	A,B,D,E,H,L
MOV8DH	2C53H	(DE) <-- (HL)	double prec.	A,B,D,E,H,L
MFA	2C59H	DAC <-- ARG	double prec.	A,B,D,E,H,L
MFM	2C5CH	DAC <-- (HL)	double prec.	A,B,D,E,H,L
MMF	2C67H	(HL) <-- DAC	double prec.	A,B,D,E,H,L
MOV8HD	2C6AH	(HL) <-- (DE)	double prec.	A,B,D,E,H,L
XTF	2C6FH	(SP) <--> DAC	double prec.	A,B,D,E,H,L
PHA	2CC7H	ARG <-- (SP)	double prec.	A,B,D,E,H,L
PHF	2CCCH	DAC <-- (SP)	double prec.	A,B,D,E,H,L
PPA	2CDCH	(SP) <-- ARG	double prec.	A,B,D,E,H,L
PPF	2CE1H	(SP) <-- DAC	double prec.	A,B,D,E,H,L
PUSHF	2EB1H	DAC <-- (SP)	single prec.	D,E
MOVFM	2EBEH	DAC <-- (HL)	single prec.	B,C,D,E,H,L
MOVFR	2EC1H	DAC <-- (CBED)	single prec.	D,E
MOVRF	2ECCH	(CBED) <-- DAC	single prec.	B,C,D,E,H,L
MOVRMI	2ED6H	(CBED) <-- (HL)	single prec.	B,C,D,E,H,L
MOVRM	2EDFH	(BCDE) <-- (HL)	single prec.	B,C,D,E,H,L
MOVMF	2EE8H	(HL) <-- DAC	single prec.	A,B,D,E,H,L
MOVE	2EEBH	(HL) <-- (DE)	single prec.	B,C,D,E,H,L
VMOVAM	2EEFH	ARG <-- (HL)	VALTYP	B,C,D,E,H,L
MOVVFM	2EF2H	(DE) <-- (HL)	VALTYP	B,C,D,E,H,L
VMOVE	2EF3H	(HL) <-- (DE)	VALTYP	B,C,D,E,H,L
VMOVFA	2F05H	DAC <-- ARG	VALTYP	B,C,D,E,H,L
VMOVFM	2F08H	DAC <-- (HL)	VALTYP	B,C,D,E,H,L
VMOVAF	2F0DH	ARG <-- DAC	VALTYP	B,C,D,E,H,L
VMOVFM	2F10H	(HL) <-- DAC	VALTYP	B,C,D,E,H,L

Note: (HL), (DE) means the values in memory pointed to by HL or DE. Four register names in the parentheses are the single precision real numbers which indicate (sign + exponent), (mantissa 1st and 2nd places), (mantissa 3th and 4th places), (mantissa 5th and 6th places) from left to right. Where the object is VALTYP, the movement (2, 4, 8 bytes) is according to the type indicated in VALTYP (F663H).

## Comparison

Label	Address	Object	Left	Right	Reg. mod.
FCOMP	2F21H	single prec. real number	CBED	DAC	HL
ICOMP	2F4DH	2-byte integer	DE	HL	HL
XDCOMP	2F5CH	double prec. real number	ARG	DAC	all

Note: Results will be in A register. Meanings of A register are:

A = 1    --> left < right  
A = 0    --> left = right  
A = -1   --> left > right

In the comparison of single precision real numbers, CBED means that each register has single precision (sign + exponent), (mantissa 1st and 2nd places), (mantissa 3th and 4th places), and (mantissa 5th and 6th places).

## Floating-point input/output

Label	Address	Function
FIN	3299H	Stores a string representing the floating-point number in DAC, converting it in real.
Entry condition	HL <--	Starting address of the string
	A <--	First character of the string
Return condition	DAC <--	Real number
	C <--	FFH: without a decimal point 0: with a decimal point
	B <--	Number of places after the decimal point
	D <--	Number of digits

Label	Address	Function
FOUT	3425H	Converts the real number in DAC to the string (unformatted)
PUFOUT	3426H	Converts the real number in DAC to the string (formatted)
Entry condition	A <--	format
	bit 7	0: unformatted 1: formatted
	bit 6	0: without commas 1: with commas every three digits
	bit 5	0: meaningless 1: leading spaces are padded with "."
	bit 4	0: meaningless 1: "\$" is added before the numerical value
	bit 3	0: meaningless 1: "+" is added even for positive values
	bit 2	0: meaningless 1: the sign comes after the value
	bit 1	unused
	bit 0	0: fixed point 1: floating-point
	B <--	number of digits before and not including the decimal point

C <-- number of digits after and including the decimal point  
 Return condition HL <-- starting address of the string

Label	Address	Function
FOUTB	371AH	Converts 2-byte integer in DAC+2, 3 to a binary expression string.
FOUTO	371EH	Converts 2-byte integer in DAC+2, 3 to an octal expression string.
FOUTH	3722H	Converts 2-byte integer in DAC+2, 3 to a hexadecimal expression string.

Entry condition DAC + 2 <-- 2-byte integer  
 VALTYP <-- 2  
 Return condition HL <-- starting address of the string

Note: no strings are reserved. The starting address of the string in the output routine is normally in FBUFR (from F7C5H). In some cases it may differ slightly. For the integer in DAC + 2, VALTYP (F663H) must be 2, even in cases other than FOUTB, FOUTO and FOUTH.

#### Type conversion

Label	Address	Function
FRCINT	2F8AH	Converts DAC to a 2-byte integer (DAC + 2, 3)
FRCNG	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))

Note: after execution, VALTYP (F663H) will contain the number (2, 4 or 8) representing DAC type. No registers are reserved.

#### Integer operation

Label	Address	Function	Registers modified
UMULT	314AH	DE <-- BC * DE	A, B, C, D, E
ISUB	3167H	HL <-- DE - HL	all
IADD	3172H	HL <-- DE + HL	all
IMULT	3193H	HL <-- DE * HL	all
IDIV	31E6H	HL <-- DE / HL	all
IMOD	323AH	HL <-- DE mod HL (DE <-- DE/HL)	all

#### Power

Label	Address	Function	Base	Exp.	Result
SGNEXP	37C8H	power of single-prec. real	DAC	ARG	DAC
DBLEXP	37D7H	power of double-prec. real	DAC	ARG	DAC
INTEXP	383FH	power of 2-byte integer	DE	HL	DAC

Note: No registers are reserved.

Changes from the original in APPENDIX 3:

none

### APPENDIX 3 - BIT BLOCK TRANSFER

The bit block transfer corresponds to the COPY command in BASIC and is used to transfer data from RAM, VRAM, and the disk. It is easily executed by the routine in expansion ROM and available from the assembly language program. Since it is in expansion ROM, use SUBROM or EXTROM of BIOS for this routine.

#### 1. Transferring in VRAM

\* BLTVV (0191H/SUB)

Function: transfers data in VRAM area

Input: HL register <-- F562H

The following parameters should be set:

- \* SX (F562H, 2) X-coordinate of the source
- \* SY (F564H, 2) Y-coordinate of the source
- \* DX (F566H, 2) X-coordinate of the destination
- \* DY (F568H, 2) Y-coordinate of the destination
- \* NX (F56AH, 2) number of dots in the X direction
- \* NY (F56CH, 2) number of dots in the Y direction
- \* CDUMMY (F56EH, 1) dummy (not required to be set)
- \* ARG (F56FH, 1) selects the direction and expansion  
RAM (same as VDP R#45)
- \* LOGOP (F570H, 1) logical operation code (same as the  
logical operation code of VDP)

Output: the CY flag is reset

Registers: all

#### 2. Transferring data between RAM and VRAM

To use the routines below, the following memory space should be allocated as

graphic area for screen modes.

\* screen mode 6

number of dots in X direction times number of dots in Y direction/4 + 4

\* screen mode 5 or 7

number of dots in X direction times number of dots in Y direction/2 + 4

\* screen mode 8

number of dots in X direction times number of dots in Y direction/2 + 4

Note to raise fractions.

For disk or RAM, data to indicate the size is added as the array data. The first two bytes of data indicate the number of dots in X direction; the next two bytes indicate the number of dots in the Y direction.

\* BLTVM (0195H/SUB)

Function: transfers the array to VRAM

Input: HL register <-- F562H

The following parameters should be set:

- \* DPTR (F562H, 2) source address of memory
- \* DUMMY (F564H, 2) dummy (not required to be set)
- \* DX (F566H, 2) X-coordinate of the destination
- \* DY (F568H, 2) Y-coordinate of the destination
- \* NX (F56AH, 2) number of dots in the X direction  
(not required to be set; this is already in the top of data to be transferred)
- \* NY (F56CH, 2) number of dots in the Y direction  
(not required to be set; this is already in the top of data to be transferred)
- \* CDUMMY (F56EH, 1) dummy (not required to be set)
- \* ARG (F56FH, 1) selects the direction and expansion RAM (same as VDP R#45)
- \* LOGOP (F570H, 1) logical operation code (same as the logical operation code of VDP)

Output: the CY flag is set when the number of data bytes to be transferred is incorrect

Registers: all

\* BLTMV (0199H/SUB)

Function: transfers to the array from VRAM

Input: HL register <-- F562H

The following parameters should be set:

- \* SX (F562H, 2) X-coordinate of the source
- \* SY (F564H, 2) Y-coordinate of the source
- \* DPTR (F566H, 2) destination address of memory
- \* DUMMY (F568H, 2) dummy (not required to be set)
- \* NX (F56AH, 2) number of dots in the X direction
- \* NY (F56CH, 2) number of dots in the Y direction
- \* CDUMMY (F56EH, 1) dummy (not required to be set)
- \* ARG (F56FH, 1) selects the direction and expansion  
RAM (same as VDP R#45)

Output: the CY flag is reset

Registers: all

### 3. Transferring between the disk and RAM or VRAM

The filename should be set first to use the disk (specify the filename as BASIC). The following is an example:

```

.
.
.
LD HL,FNAME      ; Get pointer to file name
LD (FNPTR),HL    ; Set it to parameter area
.
.
.

```

FNAME: DB 22H,"B:TEST.PIC",22H,0 ; "TEST.PIC", end mark

When an error occurs, control jumps to the error handler of the BASIC interpreter. Set the hook to handle the error in the user program or to call this routine from MSX-DOS or a ROM cartridge. This hook is H.ERRO (FFB1H).

#### \* BLTVD (019DH/SUB)

Function: transfers from disk to VRAM

Input: HL register <-- F562H

The following parameters should be set:

- \* FNPTR (F562H, 2) address of the filename
- \* DUMMY (F564H, 2) dummy (not required to be set)
- \* DX (F566H, 2) X-coordinate of the destination
- \* DY (F568H, 2) Y-coordinate of the destination
- \* NX (F56AH, 2) number of dots in the X direction  
(not required to be set; this is  
already in the top of data to be  
transferred)
- \* NY (F56CH, 2) number of dots in the Y direction  
(not required to be set; this is  
already in the top of data to be  
transferred)
- \* CDUMMY (F56EH, 1) dummy (not required to be set)
- \* ARG (F56FH, 1) selects the direction and expansion  
RAM (same as VDP R#45)



\* LOGOP (F570H, 1) logical operation code (same as the logical operation code of VDP)

Output: the CY flag is set when there is an error in the parameter

Registers: all

\* BLTDV (01A1H/SUB)

Function: transfers from VRAM to disk

Input: HL register <-- F562H

The following parameters should be set:

- \* 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 (not required to be set)
- \* NX (F56AH, 2) number of dots in the X direction
- \* NY (F56CH, 2) number of dots in the Y direction
- \* CDUMMY (F56EH, 1) dummy (not required to be set)

Output: the CY flag is reset

Registers: all

\* BLTMD (01A5H/SUB)

Function: loads array data from disk

Input: HL register <-- F562H

The following parameters should be set:

- \* FNPTR (F562H, 2) address of the filename
- \* SY (F564H, 2) dummy (not required to be set)
- \* SPTR (F566H, 2) the starting address for loading
- \* EPTR (F568H, 2) the end address for loading

Output: the CY flag is reset

Registers: all

\* BLTDM (01A9H/SUB)

Function: saves array data to disk

Input: HL register <-- F562H

The following parameters should be set:

- \* SPTR (F562H, 2) the starting address for saving
- \* EPTR (F564H, 2) the end address for saving
- \* FNPTR (F566H, 2) address of the filename

Output: the CY flag is reset

Registers: all

=====  
Changes from the original in APPENDIX 4:

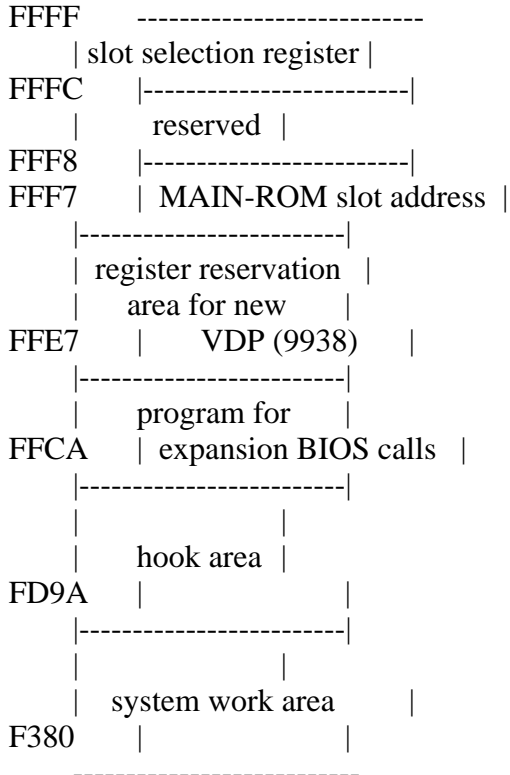
- Address of FLAGS variable is corrected from FB1BH to FB1CH.
- Address of MCLLEN variable is corrected from FB39H to FB3BH.
- Address of H.FIEL hook is corrected from DE2BH to FE2BH.

-----  
**APPENDIX 4 - WORK AREA LISTING**

Figure A.5 shows the map of the MSX2 work area. In this section, the system work area and hook from F380H to FFCAH in the figure are described. The following notation is used. Length is in bytes.

Label name (address, length)  
Initial value, contents, purpose

Figure A.5 Work area



\* Subroutines for read/write calls of the inter-slot

RDPRIM (F380H, 5)  
contents: read from basic slot

WRPRIM (F385H, 7)  
contents: write to basic slot

CLPRIM (F38CH, 14)  
contents: basic slot call

\* Starting address of assembly language program of USR function, text screen

USRTAB (F39AH, 20)  
initial value: FCERR  
contents: starting address of assembly language program of USR function  
(0 to 9); the value before defining assembly language program  
points to FCERR (475AH).

LINL40 (F3AEH, 1)  
initial value: 39  
contents: screen width per line at SCREEN 0 (set by WIDTH statement  
at SCREEN 0)

LINL32 (F3AfH, 1)  
initial value: 32  
contents: screen width per line at SCREEN 1 (set by WIDTH statement  
at SCREEN 1)

LINLEN (F3B0H, 1)  
initial value: 29  
contents: current screen width per line

CRTCNT (F3B1H, 1)  
initial value: 24  
contents: number of lines of current screen

CLMLST (F3B2H, 1)  
initial value: 14  
contents: horizontal location in the case that items are divided by  
commas in PRINT statement

\* Work for initialisation

- SCREEN 0

TXTNAM (F3B3H, 2)  
initial value: 0000H  
contents: pattern name table

TXTCOL (F3B5H, 2)  
contents: unused

TXTCGP (F3B7H, 2)  
initial value: 0800H  
contents: pattern generator table

TXTATR (F3B9H, 2)  
contents: unused

TXTPAT (F3BBH, 2)  
contents: unused

- SCREEN 1

T32NAM (F3BDH, 2)  
initial value: 1800H  
contents: pattern name table

T32COL (F3BFH, 2)  
initial value: 2000H  
contents: colour table

T32CGP (F3C1H, 2)  
initial value: 0000H  
contents: pattern generator table

T32ATR (F3C3H, 2)  
initial value: 1B00H  
contents: sprite attribute table

T32PAT (F3C5H, 2)  
initial value: 3800H  
contents: sprite generator table

- SCREEN 2

GRPNAM (F3C7H, 2)  
initial value: 1800H  
contents: pattern name table

GRPCOL (F3C9H, 2)  
initial value: 2000H  
contents: colour table

GRPCGP (F3CBH, 2)  
initial value: 0000H  
contents: pattern generator table

GRPATR (F3CDH, 2)  
initial value: 1B00H  
contents: sprite attribute table

GRPPAT (F3CFH, 2)  
initial value: 3800H  
contents: sprite generator table

- SCREEN 3

MLTNAM (F3D1H, 2)  
initial value: 0800H  
contents: pattern name table

MLTCOL (F3D3H, 2)  
contents: unused

MLTCGP (F3D5H, 2)  
initial value: 0000H  
contents: pattern generator table

MLTATR (F3D7H, 2)  
initial value: 1B00H  
contents: sprite attribute table

MLTPAT (F3D9H, 2)  
initial value: 3800H  
contents: sprite generator table

\* Other screen settings

CLIKSW (F3DBH, 1)  
initial value: 1  
contents: key click switch (0 = OFF, otherwise = ON), set by  
<key click switch> of SCREEN statement

CSRY (F3DCH, 1)  
initial value: 1  
contents: Y-coordinate of cursor

CSRX (F3DDH, 1)  
initial value: 1  
contents: X-coordinate of cursor

CNSDFG (F3DEH, 1)  
initial value: 0  
contents: function key display switch (0 = display, otherwise = no display), set by KEY ON/OFF statement

\* Area to save VDP registers

RG0SAV (F3DFH, 1)  
initial value: 0

RG1SAV (F3E0H, 1)  
initial value: E0H

RG2SAV (F3E1H, 1)  
initial value: 0

RG3SAV (F3E2H, 1)  
initial value: 0

RG4SAV (F3E3H, 1)  
initial value: 0

RG5SAV (F3E4H, 1)  
initial value: 0

RG6SAV (F3E5H, 1)  
initial value: 0

RG7SAV (F3E6H, 1)  
initial value: 0

STATFL (F3E7H, 1)  
initial value: 0  
contents: stores VDP status (contents of status register 0, in MSX2)

TRGFLG (F3E8H, 1)  
initial value: FFH  
contents: stores trigger button status of joystick

FORCLR (F3E9H, 1)  
initial value: 15  
contents: foreground colour; set by colour statement

BAKCLR (F3EAH, 1)  
initial value: 4  
contents: background colour; set by colour statement

BDRCLR (F3EBH, 1)  
initial value: 7  
contents: border colour; set by colour statement

MAXUPD (F3ECH, 3)  
initial value: JP 0000H (C3H, 00H, 00H)  
contents: used by CIRCLE statement internally

MINUPD (F3EFH, 3)  
initial value: JP 0000H (C3H, 00H, 00H)  
contents: used by CIRCLE statement internally

ATRBYT (F3F2H, 1)  
initial value: 15  
contents: colour code in using graphics

\* Work area for PLAY statement

QUEUES (F3F3H, 2)  
initial value: QUETAB (F959H)  
contents: points to queue table at the execution of PLAY statement

FRCNEW (F3F5H, 1)  
initial value: 255  
contents: used by BASIC interpreter internally

\* Work area for key input

SCNCNT (F3F6H, 1)  
initial value: 1  
contents: interval for the key scan

REPCNT (F3F7H, 1)

initial value: 50

contents: delay until the auto-repeat of the key begins

PUTPNT (F3F8H, 2)

initial value: KEYBUF (FBF0H)

contents: points to address to write in the key buffer

GETPNT (F3FAH, 2)

initial value: KEYBUF (FBF0H)

contents: points to address to read from key buffer

\* Parameters for Cassette

CS120 (F3FCH, 5\*2)

- 1200 baud

contents: 83 (LOW01) ..... Low width representing bit 0  
92 (HIGH01) ..... High width representing bit 0  
38 (LOW11) ..... Low width representing bit 1  
45 (HIGH11) ..... High width representing bit 1  
HEADLEN \* 2/256 ..... High bytes (HEDLEN = 2000)  
of header bits for short  
header

- 2400 baud

contents: 37 (LOW02) ..... Low width representing bit 0  
45 (HIGH02) ..... High width representing bit 0  
14 (LOW12) ..... Low width representing bit 1  
22 (HIGH12) ..... High width representing bit 1  
HEADLEN \* 4/256 ..... High bytes (HEDLEN = 2000)  
of header bits for short  
header

LOW (F406H, 2)

initial value: LOW01, HIGH01 (by default, 1200 baud)

contents: width of LOW and HIGH which represents bit 0 of current baud  
rate; set by <cassette baud rate> of SCREEN statement

HIGH (F408H, 2)

initial value: LOW11, HIGH11 (by default, 1200 baud)

contents: width of LOW and HIGH which represents bit 1 of current baud  
rate; set by <cassette baud rate> of SCREEN statement

HEADER (F40AH, 1)

initial value: HEADLEN \* 2/256 (by default, 1200 baud)

contents: header bit for the short header of current baud rate



(HEADLEN = 2000); set by <cassette baud rate> of SCREEN statement

ASPCT1 (F40BH, 1)

contents: 256/aspect ratio; set by SCREEN statement to use in CIRCLE statement

ASPCT2 (F40DH, 1)

contents: 256 \* aspect ratio; set by SCREEN statement to use in CIRCLE statement

ENDPRG (F40FH, 5)

initial value: ":"

contents: false end of program for RESUME NEXT statement

\* Work used by BASIC internally

ERRFLG (F414H, 1)

contents: area to store the error number

LPTPOS (F415H, 1)

initial value: 0

contents: printer head location

PRTFLG (F416H, 1)

contents: flag whether to send to printer

NTMSXP (F417H, 1)

contents: printer (0 = printer for MSX, otherwise not)

RAWPRT (F418H, 1)

contents: non-zero when printing in raw-mode

VLZADR (F419H, 2)

contents: address of character to be replaced by VAL function

VLZDAT (F41BH, 1)

contents: character to be replaced with 0 by VAL function

CURLIN (F41CH, 2)

contents: currently executing line number of BASIC

KBUF (F41FH, 318)

contents: crunch buffer; translated into intermediate language from  
BUF (F55EH)

BUFMIN (F55DH, 1)

initial value: ", "

contents: used in INPUT statement

BUF (F55EH, 258)

contents: buffer to store characters typed; where direct statements  
are stored in ASCII code

ENDBUF (F660H, 1)

contents: prevents overflow of BUF (F55EH)

TTYPOS (F661H, 1)

contents: virtual cursor location internally retained by BASIC

DIMFLG (F662H, 1)

contents: used by BASIC internally

VALTYP (F663H, 1)

contents: used to identify the type of variable

DORES (F664H, 1)

contents: indicates whether stored word can be crunched

DONUM (F665H, 1)

contents: flag for crunch

CONTXT (F666H, 2)

contents: stores text address used by CHRGET

CONSAV (F668H, 1)

contents: stores token of constant after calling CHRGET

CONTYP (F669H, 1)

contents: type of stored constant

CONLO (F66AH, 8)

contents: value of stored constant

MEMSIZ (F672H, 2)

contents: highest address of memory used by BASIC

STKTOP (F674H, 2)

contents: address used as stack by BASIC; depending on CLEAR statement

TXTTAB (F676H, 2)

contents: starting address of BASIC text area

TEMPPT (F768H, 2)

initial value: TEMPST (F67AH)

contents: starting address of unused area of temporary descriptor

TEMPST (F67AH, 3 \* NUMTMP)

contents: area for NUMTEMP

DSCTMP (F698H, 3)

contents: string descriptor which is the result of string function

FRETOP (F69BH, 2)

contents: starting address of unused area of string area

TEMP3 (F69DH, 2)

contents: used for garbage collection or by USR function

TEMP8 (F69FH, 2)

contents: for garbage collection

ENDFOR (F6A1H, 2)

contents: stores next address of FOR statement (to begin execution from the next of FOR statement at loops)

DATLIN (F6A3H, 2)

contents: line number of DATA statement read by READ statement

SUBFLG (F6A5H, 1)

contents: flag for array for USR function

FLGINP (F6A6H, 1)

contents: flag used in INPUT or READ

TEMP (F6A7H, 2)

contents: location for temporary reservation for statement code; used for variable pointer, text address, and others

PTRFLG (F6A9H, 1)

contents: 0 if there is not a line number to be converted, otherwise not

AUTFLG (F6AAH, 1)

contents: flag for AUTO command validity (non-zero = valid, otherwise invalid)

AUTLIN (F6ABH, 2)

contents: last input line number

AUTINC (F6ADH, 2)

initial value: 10

contents: increment value of line number of AUTO command

SAVTXT (F6AFH, 2)

contents: area to store address of currently executing text; mainly used for error recovery by RESUME statement

ERRLIN (F6B3H, 2)

contents: line number where an error occurred

DOT (F6B5H, 2)

contents: last line number which was displayed in screen or entered

ERRTXT (F6B7H, 2)

contents: text address which caused an error; mainly used for error recovery by RESUME statement

ONELIN (F6B9H, 2)

contents: text address to which control jumps at error; set by ON ERROR GOTO statement

ONEFLG (F6BBH, 1)

contents: flag which indicates error routine execution (non-zero = in execution, otherwise not)

TEMP2 (F6BCH, 2)

contents: for temporary storage

OLDLIN (F6BEH, 2)

contents: line number which was terminated by Ctrl+STOP, STOP instruction, END instruction, or was executed last

OLDTXT (F6C0H, 2)

contents: address to be executed next

VARTAB (F6C2H, 2)

contents: starting address of simple variable; executing NEW statement causes [contents of TXTTAB(F676H) + 2] to be set

ARYTAB (F6C4H, 2)

contents: starting address of array table

STREND (F6C6H, 2)

contents: last address of memory in use as text area or variable area

DATPTR (F6C8H, 2)

contents: text address of data read by executing READ statement

DEFTBL (F6CAH, 26)

contents: area to store type of variable for one alphabetical character; depends on type declaration such as CLEAR, DEFSTR, !, or #

\* Work for user function parameter

PRMSTK (F6E4H, 2)

contents: previous definition block on stack (for garbage collection)

PRMLN (F6E6H, 2)

contents: number of bytes of objective data

PARM1 (F6E8H, PRMSIZ)

contents: objective parameter definition table; PRMSIZ is number of bytes of definition block, initial value is 100

PRMPRV (F74CH, 2)

initial value: PRMSTK

contents: pointer to previous parameter block (for garbage collection)

PRMLN2 (F74EH, 2)

contents: size of parameter block

PARM2 (F750H, 100)

contents: for parameter storage

PRMFLG (F7B4H, 1)

contents: flag to indicate whether PARM1 was searched

ARYTA2 (F7B5H, 2)

contents: end point of search

NOFUNS (F7B7H, 1)

contents: 0 if there is not an objective function

TEMP9 (F7B8H, 2)

contents: location of temporary storage for garbage collection

FUNACT (F7BAH, 2)

contents: number of objective functions

SWPTMP (F7BCH, 8)

contents: location of temporary storage of the value of the first variable of SWAP statement

TRCFLG (F7C4H, 1)

contents: trace flag (non-zero = TRACE ON, 0 = TRACE OFF)

\* Work for Math-Pack

FBUFFR (F7C5H, 43)

contents: used internally by Math-Pack

DECTMP (F7F0H, 2)

contents: used to transform decimal integer to floating-point number

DECTM2 (F7F2H, 2)

contents: used at division routine execution

DECCNT (F7F4H, 2)

contents: used at division routine execution

DAC (F7F6H, 16)

contents: area to set the value to be calculated

HOLD8 (F806H, 48)

contents: register storage area for decimal multiplication

HOLD2 (F836H, 8)

contents: used internally by Math-Pack

HOLD (F83EH, 8)

contents: used internally by Math-Pack

ARG (F847H, 16)

contents: area to set the value to be calculated with DAC (F7F6H)

RNDX (F857H, 8)

contents: stores last random number in double precision real number;  
set by RND function

\* Data area used by BASIC interpreter

MAXFIL (F85FH, 1)

contents: maximum file number; set by MAXFILES statement

FILTAB (F860H, 2)

contents: starting address of file data area

NULBUF (F862H, 2)

contents: points to buffer used in SAVE and LOAD by BASIC interpreter

PTRFIL (F864H, 2)

contents: address of file data of currently accessing file

RUNFLG (F866H, 2)

contents: non-zero value if program was loaded and executed; used  
by R option of LOAD statement

FILNAM (F866H, 11)

contents: area to store filename

FILNM2 (F871H, 11)

contents: area to store filename

NLONLY (F87CH, 1)

contents: non-zero value if program is being loaded

SAVEND (F87DH, 2)

contents: end address of assembly language program to be saved

FNKSTR (F87FH, 160)

contents: area to store function key string (16 character x 10)

CGPNT (F91FH, 3)

contents: address to store character font on ROM

NAMBAS (F922H, 2)

contents: base address of current pattern name table

CGPBAS (F924H, 2)

contents: base address of current pattern generator table

PATBAS (F926H, 2)

contents: base address of current sprite generator table

ATRBAS (F928H, 2)

contents: base address of current sprite attribute table

CLOC (F92AH, 2)

contents: used internally by graphic routine

CMASK (F92CH, 1)

contents: used internally by graphic routine

MINDEL (F92DH, 1)

contents: used internally by graphic routine

MAXDEL (F92FH, 2)

contents: used internally by graphic routine

\* Data area used by CIRCLE statement

ASPECT (F931H, 2)

contents: aspect ratio of the circle; set by <ratio> of CIRCLE statement

CENCNT (F933H, 2)

contents: used internally by CIRCLE statement

CLINEF (F935H, 1)

contents: flag whether a line is drawn toward the center; specified by <angle> of CIRCLE statement

CNPNTS (F936H, 2)

contents: point to be plotted



CPLOTF (F938H, 1)  
contents: used internally by CIRCLE statement

CPCNT (F939H, 2)  
contents: number of one eight of the circle

CPNCNT8 (F93BH, 2)  
contents: used internally by CIRCLE statement

CPCSUM (F93DH, 2)  
contents: used internally by CIRCLE statement

CSTCNT (F93FH, 2)  
contents: used internally by CIRCLE statement

CSCLXY (F941H, 1)  
contents: scale of x and y

CSAVEA (F942H, 2)  
contents: reservation area of ADVGRP

CSAVEM (F944H, 1)  
contents: reservation area of ADVGRP

CXOFF (F945H, 2)  
contents: x offset from the center

CYOFF (F947H, 2)  
contents: y offset from the center

\* Data area used in PAINT statement

LOHMSK (F949H, 1)  
contents: used internally by PAINT statement

LOHDIR (F94AH, 1)  
contents: used internally by PAINT statement

LOHADR (F94BH, 2)  
contents: used internally by PAINT statement

LOHCNT (F94DH, 2)

contents: used internally by PAINT statement

SKPCNT (F94FH, 2)

contents: skip count

MIVCNT (F951H, 2)

contents: movement count

PDIREC (F953H, 1)

contents: direction of the paint

LFPROG (F954H, 1)

contents: used internally by PAINT statement

RTPROG (F955H, 1)

contents: used internally by PAINT statement

\* Data area used in PLAY statement

MCLTAB (F956H, 2)

contents: points to the top of the table of PLAY macro or DRAW macro

MCLFLG (F958H, 1)

contents: assignment of PLAY/DRAW

QUETAB (F959H, 24)

contents: queue table

+0: PUT offset

+1: GET offset

+2: backup character

+3: queue length

+4: queue address

+5: queue address

QUEBAK (F971H, 4)

contents: used in BCKQ

VOICAQ (F975H, 128)

contents: queue of voice 1 (1 = a)

VOICBQ (F9F5H, 128)

contents: queue of voice 2 (2 = b)

VOICCQ (FA75H, 128)

contents: queue of voice 3 (3 = c)

\* Work area added in MSX2

DPPAGE (FAF5H, 1)

contents: display page number

ACPAGE (FAF6H, 1)

contents: active page number

AVCSAV (FAF7H, 1)

contents: reserves AV control port

EXBRSA (FAF8H, 1)

contents: SUB-ROM slot address

CHRCNT (FAF9H, 1)

contents: character counter in the buffer; used in Roman-kana translation (value is  $0 \leq n \leq 2$ )

ROMA (FAFAH, 2)

contents: area to store character in the buffer; used in Roman-kana translation (Japan version only)

MODE (FAFCH, 1)

contents: mode switch for VRAM size

(0000WVV0)

---

|||

|+---- 00 = 16K VRAM

| 01 = 64K VRAM

| 11 = 128K VRAM

|

+----- 1 = mask, 0 = no mask

Flags whether to specify VRAM address

ANDed with 3FFFH in SCREEN 0 to 3;

in SCREEN 4 to 8, never masked

NORUSE (FAFDH, 1)

contents: unused

XSAVE (FAFEH, 2)

contents: [ I OOOOOOO XXXXXXXX ]

YSAVE (FB00H, 2)

contents: [ x OOOOOOO YYYYYYYY ]

I = 1 lightpen interrupt request

OOOOOOO = unsigned offset

XXXXXXX = X-coordinate

YYYYYYY = Y-coordinate

LOGOPR (FB02H, 1)

contents: logical operation code

\* Data area used by RS-232C

RSTMP (FB03H, 50)

contents: work area for RS-232C or disk

TOCNT (FB03H, 1)

contents: used internally by RS-232C routine

RSFCB (FB04H, 2)

contents: FB04H + 0: LOW address of RS-232C

FB04H + 1: HIGH address of RS-232C

RSIQLN (FB06H, 5)

contents: used internally by RS-232C routine

MEXBIH (FB07H, 5)

contents: FB07H +0: RST 30H (0F7H)

FB07H +1: byte data

FB07H +2: (Low)

FB07H +3: (High)

FB07H +4: RET (0C9H)

OLDSTT (FB0CH, 5)

contents: FB0CH +0: RST 30H (0F7H)

FB0CH +1: byte data

FB0CH +2: (Low)

FB0CH +3: (High)

FB0CH +4: RET (0C9H)

OLDINT (FB12H, 5)

contents: FB12H +0: RST 30H (0F7H)

FB12H +1: byte data

FB12H +2: (Low)

FB12H +3: (High)

FB12H +4: RET (0C9H)

DEVNUM (FB17H, 1)

contents: used internally by RS-232C routine

DATCNT (FB18H, 3)

contents: FB18H +0: byte data

FB18H +1: byte pointer

FB12H +2: byte pointer

ERRORS (FB1BH, 1)

contents: used internally by RS-232C routine

FLAGS (FB1CH, 1)

contents: used internally by RS-232C routine

ESTBLS (FB1DH, 1)

contents: used internally by RS-232C routine

COMMSK (FB1EH, 1)

contents: used internally by RS-232C routine

LSTCOM (FB1FH, 1)

contents: used internally by RS-232C routine

LSTMOD (FB20H, 1)

contents: used internally by RS-232C routine

\* Data area used by DOS

reserved (FB21H to FB34H)

contents: used by DOS

\* Data area used by PLAY statement

(the following is the same as with MSX1)

PRSCNT (FB35H, 1)

contents: D1 to D0 string parse

D7 = 0 1 pass

SAVSP (FB36H, 2)

contents: reserves stack pointer in play

VOICEN (FB38H, 1)

contents: current interpreted voice

SAVVOL (FB39H, 2)

contents: reserves volume for the pause

MCLLEN (FB3BH, 1)

contents: used internally by PLAY statement

MCLPTR (FB3CH, 1)

contents: used internally by PLAY statement

QUEUEN (FB3EH, 1)

contents: used internally by PLAY statement

MUSICF (FC3FH, 1)

contents: interrupt flag for playing music

PLYCNT (FB40H, 1)

contents: number of PLAY statements stored in the queue

\* Offset from voice static data area  
(offset is in decimal)

METREX (+0, 2)

contents: timer count down

VCXLEN (+2, 1)

contents: MCLLEN for this voice

VCXPTR (+3, 2)

contents: MCLPTR for this voice

VCXSTP (+5, 2)

contents: reserves the top of the stack pointer

QLENGX (+7, 1)

contents: number of bytes stored in the queue

NTICSX (+8, 2)

contents: new count down

TONPRX (+10, 2)

contents: area to set tone period

AMPPRX (+12, 1)

contents: discrimination of volume and envelope

ENVPRX (+13, 2)

contents: area to set envelope period

OCTAVX (+15, 1)

contents: area to set octave

NOTELX (+16, 1)

contents: area to set tone length

TEMPOX (+17, 1)

contents: area to set tempo

VOLUMX (+18, 1)

contents: area to set volume

ENVLPX (+19, 14)

contents: area to set envelope wave form

MCLSTX (+33, 3)

contents: reservation area of stack

MCLSEX (+36, 1)

contents: initialisation stack

VCBSIZ (+37, 1)

contents: static buffer size

\* Voice static data area

VCBA (FB41H, 37)

contents: static data for voice 0

VCBB (FB66H, 37)

contents: static data for voice 1

VCBC (FB8BH, 37)

contents: static data for voice 2

\* Data area

ENSTOP (FBB0H, 1)

contents: flag to enable warm start by [SHIFT+Ctrl+Kana key]  
(0 = disable, otherwise enable)

BASROM (FBB1H, 1)

contents: indicates BASIC text location (0 = on RAM, otherwise in ROM)

LINTTB (FBB2H, 24)

contents: line terminal table; area to keep information about  
each line of text screen

FSTPOS (FBCAH, 2)

contents: first character location of line from INLIN (00B1H) of BIOS

CODSAV (FBCCH, 1)

contents: area to reserve the character where the cursor is stacked

FNKSW1 (FBCDH, 1)

contents: indicates which function key is displayed at KEY ON  
(1 = F1 to F5 is displayed, 0 = F6 to F10 is displayed)

FNKFLG (FBCEH, 10)

contents: area to allow, inhibit, or stop the execution of the line  
defined in ON KEY GOSUB statement, or to reserve it for each  
function key; set by KEY(n)ON/OFF/STOP statement  
(0 = KEY(n)OFF/STOP, 1= KEY(n)ON)

ONGSBF (FBD8H, 1)

contents: flag to indicate whether event waiting in TRPTBL (FC4CH)  
occurred

CLIKFL (FBD9H, 1)

contents: key click flag

OLDKEY (FBDAH, 11)

contents: key matrix status (old)

NEWKEY (FBE5H, 11)

contents: key matrix status (new)

KEYBUF (FBF0H, 40)



contents: key code buffer

#### LINWRK (FC18H, 40)

contents: temporary reservation location used by screen handler

#### PATWRK (FC40H, 8)

contents: temporary reservation location used by pattern converter

#### BOTTOM (FC48H, 2)

contents: installed RAM starting (low) address; ordinarily 8000H in MSX2

#### HIMEM (FC4AH, 2)

contents: highest address of available memory; set by <memory upper limit> of CLEAR statement

#### TRAPTBL (FC4CH, 78)

contents: trap table used to handle interrupt; one table consists of three bytes, where first byte indicates ON/OFF/STOP status and the rest indicate the text address to be jumped to

FC4CH to FC69H (3 * 10 bytes)	used in ON KEY GOSUB
FC6AH to FC6CH (3 * 1 byte)	used in ON STOP GOSUB
FC6DH to FC6FH (3 * 1 byte)	used in ON SPRITE GOSUB
FC70H to FC7EH (3 * 5 bytes)	used in ON STRIG GOSUB
FC7FH to FC81H (3 * 1 byte)	used in ON INTERVAL GOSUB
FC82H to FC99H	for expansion

#### RTYCNT (FC9AH, 1)

contents: used internally by BASIC

#### INTFLG (FC9BH, 1)

contents: if Ctrl+STOP is pressed, setting 03H here causes a stop

#### PADY (FC9CH, 1)

contents: Y-coordinate of the paddle)

#### PADX (FC9DH, 1)

contents: X-coordinate of the paddle)

#### JIFFY (FC9EH, 2)

contents: used internally by PLAY statement

#### INTVAL (FCA0H, 2)

contents: interval period; set by ON INTERVAL GOSUB statement

INTCNT (FCA2H, 2)

contents: counter for interval

LOWLIM (FCA4H, 1)

contents: used during reading from cassette tape

WINWID (FCA5H, 1)

contents: used during reading from cassette tape

GRPHED (FCA6H, 1)

contents: flag to send graphic character (1 = graphic character,  
0 = normal character)

ESCCNT (FCA7H, 1)

contents: area to count from escape code

INSFLG (FCA8H, 1)

contents: flag to indicate insert mode (0 = normal mode,  
otherwise = insert mode)

CSRSW (FCA9H, 1)

contents: whether cursor is displayed (0 = no, otherwise = yes);  
set by <cursor swich> of LOCATE statement

CSTYLE (FCAAH, 1)

contents: cursor shape (0 = block, otherwise = underline)

CAPST (FCABH, 1)

contents: CAPS key status (0 = CAP OFF, otherwise = CAP ON)

KANAST (FCACH, 1)

contents: kana key status (0 = kaka OFF, otherwise = kana ON)

KANAMD (FCADH, 1)

contents: kana key arrangement status (0 = 50-sound arrangement,  
otherwise = JIS arrangement)

FLBMEM (FCAEH, 1)

contents: 0 when loading BASIC program

SCRMOD (FCAFH, 1)

contents: current screen mode number

OLDSCR (FCB0H, 1)

contents: screen mode reservation area

CASPRV (FCB1H, 1)

contents: character reservation area used by CAS:

BRDATR (FCB2H, 1)

contents: border colour code used by PAINT; set by <border colour>  
in PAINT statement

GXPOS (FCB3H, 2)

contents: X-coordinate

GYPOS (FCB5H, 2)

contents: Y-coordinate

GRPACX (FCB7H, 2)

contents: graphic accumulator (X-coordinate)

GRPACY (FCB9H, 2)

contents: graphic accumulator (Y-coordinate)

DRWFLG (FCBBH, 1)

contents: flag used in DRAW statement

DRWSCL (FCBCH, 1)

contents: DRAW scaling factor (0 = no scaling, otherwise = scaling)

DRWANG (FCBDH, 1)

contents: angle at DRAW

RUNBNF (FCBEH, 1)

contents: flag to indicate BLOAD in progress, BSAVE in progress,  
or neither

SAVENT (FCBFH, 2)

contents: starting address of BSAVE

EXPTBL (FCC1H, 4)

contents: flag table for expansion slot; whether the slot is expanded

SLTTBL (FCC5H, 4)

contents: current slot selection status for each expansion slot  
register

SLTATR (FCC9H, 64)

contents: reserves attribute for each slot

SLTWRK (FD09H, 128)

contents: allocates specific work area for each slot

PROCNM (FD89H, 16)

contents: stores name of expanded statement (after CALL statement) or expansion device (after OPEN); 0 indicates the end

DEVICE (FD99H, 1)

contents: used to identify cartridge device

\* Hooks

H.KEYI (FD9AH)

meaning: beginning of MSXIO interrupt handling  
purpose: adds the interrupt operation such as RS-232C

H.TIMI (FD9FH)

meaning: MSXIO timer interrupt handling  
purpose: adds the timer interrupt handling

H.CHPH (FDA4H)

meaning: beginning of MSXIO CHPUT (one character output)  
purpose: connects other console device

H.DSPC (FDA9H)

meaning: beginning of MSXIO DSPCSR (cursor display)  
purpose: connects other console device

H.ERAC (FDAEH)

meaning: beginning of MSXIO ERACSR (erase cursor)  
purpose: connects other console device

H.DSPF (FDB3H)

meaning: beginning of MSXIO DSPFNK (function key display)  
purpose: connects other console device

H.ERAF (FDB8H)

meaning: beginning of MSXIO ERAFNK (erase function key)  
purpose: connects other console device

H.TOTE (FDBDH)

meaning: beginning of MSXIO TOTEXT (set screen in text mode)  
purpose:connects other console device

#### H.CHGE (FDC2H)

meaning: beginning of MSXIO CHGET (get one character)  
purpose:connects other console device

#### H.INIP (FDC7H)

meaning: beginning of MSXIO INIPAT (character pattern initialisation)  
purpose:uses other character set

#### H.KEYC (FDCCH)

meaning: beginning of MSXIO KEYCOD (key code translation)  
purpose:uses other key arrangement

#### H.KYEA (FDD1H)

meaning: beginning of MSXIO NMI routine (Key Easy)  
purpose:uses other key arrangement

#### H.NMI (FDD6H)

meaning: beginning of MSXIO NMI (non-maskable interrupt)  
purpose:handles NMI

#### H.PINL (FDDBH)

meaning: beginning of MSXIO PINLIN (one line input)  
purpose:uses other console input device or other input method

#### H.QINL (FDE0H)

meaning: beginning of MSXINL QINLIN (one line input displaying "?")  
purpose:uses other console input device or other input method

#### H.INLI (FDE5H)

meaning: beginning of MSXINL INLIN (one line input)  
purpose:uses other console input device or other input method

#### H.ONGO (FDEAH)

meaning: beginning of MSXSTS INGOTP (ON GOTO)  
purpose:uses other interrupt handling device

#### H.DSKO (FDEFH)

meaning: beginning of MSXSTS DSKO\$ (disk output)  
purpose:connects disk device

#### H.SETS (FDF4H)

meaning: beginning of MSXSTS SETS (set attribute)  
purpose:connects disk device

#### H.NAME (FDF9H)

meaning: beginning of MSXSTS NAME (rename)

purpose:connects disk device

#### H.KILL (FDFEH)

meaning: beginning of MSXSTS KILL (delete file)

purpose:connects disk device

#### H.IPL (FE03H)

meaning: beginning of MSXSTS IPL (initial program loading)

purpose:connects disk device

#### H.COPY (FE08H)

meaning: beginning of MSXSTS COPY (file copy)

purpose:connects disk device

#### H.CMD (FE0DH)

meaning: beginning of MSXSTS CMD (expanded command)

purpose:connects disk device

#### H.DSKF (FE12H)

meaning: beginning of MSXSTS DSKF (unusde disk space)

purpose:connects disk device

#### H.DSKI (FE17H)

meaning: beginning of MSXSTS DSKI (disk input)

purpose:connects disk device

#### H.ATTR (FE1CH)

meaning: beginning of MSXSTS ATTR\$ (attribute)

purpose:connects disk device

#### H.LSET (FE21H)

meaning: beginning of MSXSTS LSET (left-padded assignment)

purpose:connects disk device

#### H.RSET (FE26H)

meaning: beginning of MSXSTS RSET (right-padded assignment)

purpose:connects disk device

#### H.FIEL (FE2BH)

meaning: beginning of MSXSTS FIELD (field)

purpose:connects disk device

#### H.MKI\$ (FE30H)

meaning: beginning of MSXSTS MKI\$ (create integer)  
purpose:connects disk device

#### H.MKS\$ (FE35H)

meaning: beginning of MSXSTS MKS\$ (create single precision real)  
purpose:connects disk device

#### H.MKD\$ (FE3AH)

meaning: beginning of MSXSTS MKD\$ (create double precision real)  
purpose:connects disk device

#### H.CVI (FE3FH)

meaning: beginning of MSXSTS CVI (convert integer)  
purpose:connects disk device

#### H.CVS (FE44H)

meaning: beginning of MSXSTS CVS (convert single precision real)  
purpose:connects disk device

#### H.CVD (FE49H)

meaning: beginning of MSXSTS CVS (convert double precision real)  
purpose:connects disk device

#### H.GETP (FE4EH)

meaning: SPDSK GETPTR (get file pointer)  
purpose:connects disk device

#### H.SETF (FE53H)

meaning: SPCDSK SETFIL (set file pointer)  
purpose:connects disk device

#### H.NOFO (FE58H)

meaning: SPDSK NOFOR (OPEN statement without FOR)  
purpose:connects disk device

#### H.NULO (FE5DH)

meaning: SPCDSK NULOPN (open unused file)  
purpose:connects disk device

#### H.NTFL (FE62H)

meaning: SPCDSK NTFLO (file number is not 0)  
purpose:connects disk device

#### H.MERG (FE67H)

meaning: SPCDSK MERGE (program file merge)  
purpose:connects disk device

#### H.SAVE (FE6CH)

meaning: SPCDSK SAVE (save)

purpose:connects disk device

#### H.BINS (FE71H)

meaning: SPCDSK BINSAV (save in binary)

purpose:connects disk device

#### H.BINL (FE76H)

meaning: SPCDSK BINLOD (load in binary)

purpose:connects disk device

#### H.FILE (FD7BH)

meaning: SPCDSK FILES (display filename)

purpose:connects disk device

#### H.DGET (FE80H)

meaning: SPCDSK DGET (disk GET)

purpose:connects disk device

#### H.FILO (FE85H)

meaning: SPCDSK FILOU1 (file output)

purpose:connects disk device

#### H.INDS (FE8AH)

meaning: SPCDSK INDSKC (disk attribute input)

purpose:connects disk device

#### H.RSLF (FE8FH)

meaning: SPCDSK; re-select previous drive

purpose:connects disk device

#### H.SAVD (FE94H)

meaning: SPCDSK; reserve current disk

purpose:connects disk device

#### H.LOC (FE99H)

meaning: SPCDSK LOC function (indicate location)

purpose:connects disk device

#### H.LOF (FE9EH)

meaning: SPCDSK LOC function (file length)

purpose:connects disk device

#### H.EOF (FEA3H)



meaning: SPCDSK EOF function (end of file)  
purpose:connects disk device

#### H.FPOS (FEA8H)

meaning: SPCDSK FPOS function (file location)  
purpose:connects disk device

#### H.BAKU (FEADH)

meaning: SPCDSK BAKUPT (backup)  
purpose:connects disk device

#### H.PARD (FEB2H)

meaning: SPCDEV PARDEV (get peripheral name)  
purpose:expands logical device name

#### H.NODE (FEB7H)

meaning: SPCDEV NODEVN (no device name)  
purpose:sets default device name to other device

#### H.POSD (FEBCH)

meaning: SPCDEV POSDSK  
purpose:connects disk device

#### H.DEVN (FEC1H)

meaning: SPCDEV DEVNAM (process device name)  
purpose:expands logical device name

#### H.GEND (FEC6H)

meaning: SPCDEV GENDSP (FEC6H)  
purpose:expands logical device name

#### H.RUNC (FECBH)

meaning: BIMISC RUNC (clear for RUN)

#### H.CLEAR (FED0H)

meaning: BIMISC CLEARC (clear for CLEAR statement)

#### H.LOPD (FED5H)

meaning: BIMISC LOPDFT (set loop and default value)  
purpose:uses other default value for variable

#### H.STKE (FEDAH)

meaning: BIMISC STKERR (stack error)

#### H.ISFL (FEDFH)

meaning: BIMISC ISFLIO (file input-output or not)

H.OUTD (FEE4H)

meaning: BIO OUTDO (execute OUT)

H.CRDO (FEE9H)

meaning: BIO CRDO (execute CRLF)

H.DSKC (FEEEH)

meaning: BIO DSKCHI (input disk attribute)

H.DOGR (FEF3H)

meaning: GENGRP DOGRPH (execute graphic operation)

H.PRGE (FEF8H)

meaning: BINTRP PRGEND (program end)

H.ERRP (FEFDH)

meaning: BINTRP ERRPTR (error display)

H.ERRF (FF02H)

meaning: BINTRP

H.READ (FF07H)

meaning: BINTRP READY

H.MAIN (FF0CH)

meaning: BINTRP MAIN

H.DIRD (FF11H)

meaning: BINTRP DIRDO (execute direct statement)

H.FINI (FF16H)

meaning: BINTRP

H.FINE (FF1BH)

meaning: BINTRP

H.CRUN (FF20H)

meaning: BINTRP

H.CRUN (FF20H)

meaning: BINTRP

H.CRUS (FF25H)  
meaning: BINTRP

H.ISRE (FF2AH)  
meaning: BINTRP

H.NTFN (FF2FH)  
meaning: BINTRP

H.NOTR (FF34H)  
meaning: BINTRP

H.SNGF (FF39H)  
meaning: BINTRP

H.NEWS (FF3EH)  
meaning: BINTRP

H.GONE (FF43H)  
meaning: BINTRP

H.CHRG (FF48H)  
meaning: BINTRP

H.RETU (FF4DH)  
meaning: BINTRP

H.PRTF (FF52H)  
meaning: BINTRP

H.COMP (FF57H)  
meaning: BINTRP

H.FINP (FF5CH)  
meaning: BINTRP

H.TRMN (FF61H)  
meaning: BINTRP

H.FRME (FF66H)  
meaning: BINTRP

H.NTPL (FF6BH)  
meaning: BINTRP

H.EVAL (FF70H)  
meaning: BINTRP

H.OKNO (FF75H)  
meaning: BINTRP

H.FING (FF7AH)  
meaning: BINTRP

H.ISMI (FF7FH)  
meaning: BINTRP ISMID\$ (MID\$ or not)

H.WIDT (FF84H)  
meaning: BINTRP WIDTHS (WIDTH)

H.LIST (FF89H)  
meaning: BINTRP LIST

H.BUFL (FF8EH)  
meaning: BINTRP BUFLIN (buffer line)

H.FRQI (FF93H)  
meaning: BINTRP FRQINT

H.SCNE (FF98H)  
meaning: BINTRP

H.FRET (FF9DH)  
meaning: BINTRP FRETMP

H.PTRG (FFA2H)  
meaning: BIPTRG PTRGET (get pointer)  
purpose:uses variable other than default value

H.PHYD (FFA7H)  
meaning: MSXIO PHYDIO (physical disk input-output)  
purpose:connects disk device

H.FORM (FFACH)  
meaning: MSXIO FORMAT (format disk)  
purpose:connects disk device

H.ERRO (FFB1H)

meaning: BINTRP ERROR

purpose:error handling for application program

H.LPTO (FFB6H)

meaning: MSXIO LPTOUT (printer output)

purpose:uses printer other than default value

H.LPTS (FFBBH)

meaning: MSXIO LPTSTT (printer status)

purpose:uses printer other than default value

H.SCRE (FFC0H)

meaning: MSXSTS SCREEN statement entry

purpose:expands SCREEN statement

H.PLAY (FFC5H)

meaning: MSXSTS PLAY statement entry

purpose:expands PLAY statement

\* For expanded BIOS

FCALL (FFCAH)

contents: hook used by expanded BIOS

DISINT (FFCFH)

contents: used by DOS

ENAIN (FFD4H)

contents: used by DOS

=====

Changes from the original in APPENDIX 5:

- The original VRAM mapping figures have been converted to simple text tables.

- In SCREEN 0 (WIDTH 80) map, different end addresses for the blink table are indicated for 24 lines mode and 26.5 lines mode.

-----

APPENDIX 5 - VRAM MAP

\* SCREEN 0 (WIDTH 40) / TEXT 1

0000H - 03BFH --> Pattern name table  
0400H - 042FH --> Palette table  
0800H - 0FFFH --> Pattern generator table

\* SCREEN 0 (WIDTH 80) / TEXT 2

0000H - 077FH --> Pattern name table  
0800H - 08EFH --> Blink table (24 lines mode)  
090DH (26.5 lines mode)  
0F00H - 0F2FH --> Palette table  
1000H - 17FFH --> Pattern generator table

\* SCREEN 1 / GRAPHIC 1

0000H - 07FFH --> Pattern generator table  
1800H - 1AFFH --> Pattern name table  
1B00H - 1B7FH --> Sprite attribute table  
2000H - 201FH --> Colour table  
2020H - 204FH --> Palette table  
3800H - 3FFFH --> Sprite generator table

\* SCREEN 2 / GRAPHIC 2

0000H - 07FFH --> Pattern generator table 1  
0800H - 0FFFH --> Pattern generator table 2  
1000H - 17FFH --> Pattern generator table 3  
1800H - 18FFH --> Pattern name table 1  
1900H - 19FFH --> Pattern name table 2  
1A00H - 1AFFH --> Pattern name table 3  
1B00H - 1B7FH --> Sprite attribute table  
1B80H - 1BAFH --> Palette table  
2000H - 27FFH --> Colour table 1  
2800H - 2FFFH --> Colour table 2  
3000H - 37FFH --> Colour table 3  
3800H - 3FFFH --> Sprite generator table

\* SCREEN 3 / MULTI COLOUR

0000H - 07FFH --> Pattern generator table  
0800H - 0AFFH --> Pattern name table  
1B00H - 1B7FH --> Sprite attribute table  
2020H - 204FH --> Palette table  
3800H - 3FFFH --> Sprite generator table

\* SCREEN 4 / GRAPHIC 3

0000H - 07FFH --> Pattern generator table 1  
0800H - 0FFFH --> Pattern generator table 2  
1000H - 17FFH --> Pattern generator table 3  
1800H - 18FFH --> Pattern name table 1

1900H - 19FFH --> Pattern name table 2  
 1A00H - 1AFFH --> Pattern name table 3  
 1B80H - 1BAFH --> Palette table  
 1C00H - 1DFFH --> Sprite colour table  
 1E00H - 1E7FH --> Sprite attribute table  
 2000H - 27FFH --> Colour table 1  
 2800H - 2FFFH --> Colour table 2  
 3000H - 37FFH --> Colour table 3  
 3800H - 3FFFH --> Sprite generator table

\* SCREEN 5, 6 / GRAPHIC 4, 5

0000H - 5FFFH --> Pattern name table (192 lines)  
           69FFH                   (212 lines)  
 7400H - 75FFH --> Sprite colour table  
 7600H - 767FH --> Sprite attribute table  
 7680H - 76AFH --> Palette table  
 7A00H - 7FFFH --> Sprite generator table

\* SCREEN 7, 8 / GRAPHIC 6, 7

0000H - BFFFH --> Pattern name table (192 lines)  
           D3FFH                   (212 lines)  
 F000H - F7FFH --> Sprite generator table  
 F800H - F9FFH --> Sprite colour table  
 FA00H - FA7FH --> Sprite attribute table  
 FA80H - FAAFH --> Palette table

=====

Changes from the original in APPENDIX 6:

none

-----

APPENDIX 6 - I/O MAP

00H to 3FH   user defined

40H to 7FH   reserved

80H to 87H   for RS-232C

- 80H 8251 data
- 81H 8251 status/command
- 82H status read/interrupt mask
- 83H unused
- 84H 8253
- 85H 8253
- 86H 8253
- 87H 8253

88H to 8BH VDP (9938) I/O port for MSX1 adaptor  
This is V9938 I/O for MSX1. To access VDP directly,  
examine 06H and 07H of MAIN-ROM to confirm the port  
address

8CH to 8DH for the modem

8EH to 8FH reserved

90H to 91H printer port

90H bit 0: strobe output (write)

bit 1: status input (read)

91H data to be printed

92H to 97H reserved

98H to 9BH for MSX2 VDP (V9938)

98H VRAM access

99H command register access

9AH palette register access (write only)

9BH register pointer (write only)

9CH to 9FH reserved

A0H to A3H sound generator (AY-3-8910)

A0H address latch

A1H data read

A2H data write

A4H to A7H reserved

A8H to ABH parallel port (8255)

A8H port A

A9H port B

AAH port C

ABH mode set

ACH to AFH MSX engine (one chip MSX I/O)

B0H to B3H expansion memory (SONY specification) (8255)

A8H port A, address (A0 to A7)

A9H port B, address (A8 to A10, A13 to A15), control R/"

AAH port C, address (A11 to A12), data (D0 - D7)

ABH mode set

B4H to B5H CLOCK-IC (RP-5C01)

B4H address latch

B5H data

B6H to B7H reserved

B8H to BBH lightpen control (SANYO specification)

B8H read/write

B9H read/write

BAH read/write

BBH write only



BCH to BFH VHD control (JVC) (8255)

BCH port A

BDH port B

BEH port C

C0H to C1H MSX-Audio

C2H to C7H reserved

C8H to CFH MSX interface

D0H to D7H floppy disk controller (FDC)

The floppy disk controller can be interrupted by an external signal. Interrupt is possible only when the FDC is accessed. Thus, the system can treat different FDC interfaces.

D8 to D9H kanji ROM (TOSHIBA specification)

D8H b5-b0 lower address (write only)

D9H b5-b0 upper address (write)

b7-b0 data (read)

DAH to DBH for future kanji expansion

DCH to F4H reserved

F5H system control (write only)

setting bit to 1 enables available I/O devices

b0 kanji ROM

b1 reserved for kanji

b2 MSX-AUDIO

b3 superimpose

b4 MSX interface

b5 RS-232C

b6 lightpen

b7 CLOCK-IC (only on MSX2)

Bits to void the conflict between internal I/O devices or those connected by cartridge. The bits can disable the internal devices. When BIOS is initialised, internal devices are valid if no external devices are connected. Applications may not write to or read from here.

F8H colour bus I/O

F7H A/V control

b0 audio R mixing ON (write)

b1 audio L mixing OFF (write)

b2 select video input 21p RGB (write)

b3 detect video input no input (read)

b4 AV control TV (write)

b5 Ym control TV (write)

b6 inverse of bit 4 of VDP register 9 (write)

b7 inverse of bit 5 of VDP register 9 (write)

F8H to FBH reserved

FCH to FFH memory mapper

=====  
Changes from the original in APPENDIX 8:

none

-----  
APPENDIX 8 - CONTROL CODES

Code (dec)	Code (hex)	Function	Corresponding key(s)
0	00H		CTRL + @
1	01H	header at input/output of graphic characters	CTRL + A
2	02H	move cursor to the top of the previous word	CTRL + B
3	03H	end the input-waiting state	CTRL + C
4	04H		CTRL + D
5	05H	delete below cursor	CTRL + E
6	06H	move cursor to the top of the next word	CTRL + F
7	07H	speaker output (same as the BEEP statement)	CTRL + G
8	08H	delete a character before cursor	CTRL + H or BS
9	09H	move to next horizontal tab stop	CTRL + I or TAB
10	0AH	line feed	CTRL + J
11	0BH	home cursor	CTRL + K or HOME
12	0CH	clear screen and home cursor	CTRL + L or CLS
13	0DH	carriage return	CTRL + M or RETURN
14	0EH	move cursor to the end of line	CTRL + N
15	0FH		CTRL + O
16	10H		CTRL + P
17	11H		CTRL + Q
18	12H	insert mode ON/OFF	CTRL + R or INS

19	13H		CTRL + S
20	14H		CTRL + T
21	15H	delete one line from screen	CTRL + U
22	16H		CTRL + V
23	17H		CTRL + W
24	18H		CTRL + X or SELECT
25	19H		CTRL + Y
26	1AH		CTRL + Z
27	1BH		CTRL + [ or ESC
28	1CH	move cursor right	CTRL + \ or RIGHT
29	1DH	move cursor left	CTRL + ] or LEFT
30	1EH	move cursor up	CTRL + ^ or UP
31	1FH	move cursor down	CTRL + _ or DOWN
127	7FH	delete character under cursor	DEL

Changes from the original in APPENDIX 10:

none

## APPENDIX 10 - ESCAPE SEQUENCES

### \* Cursor movement

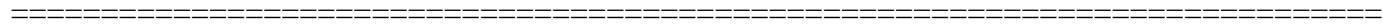
<ESC> A     move cursor up  
 <ESC> B     move cursor down  
 <ESC> C     move cursor right  
 <ESC> D     move cursor left  
 <ESC> H     move cursor home  
 <ESC> Y <Y-coordinate+20H> <X-coordinate+20H>  
           move cursor to (X, Y)

### \* Edit, delete

<ESC> j clear screen  
<ESC> E clear screen  
<ESC> K delete to end of line  
<ESC> J delete to end of screen  
<ESC> L insert one line  
<ESC> M delete one line

\* Miscellaneous

<ESC> x4 set block cursor  
<ESC> x5 hide cursor  
<ESC> y4 set underline cursor  
<ESC> y5 display cursor



APPENDIX 7 - CARTRIDGE HARDWARE

and

APPENDIX 9 - CHARACTER SET

are not available here