MSX Supervised by magazine editorial department

MSX turbo R Technical Hand Book

MSX turbo R Technical handbook

ASCII Publishing Bureau

MSX turbo R Technical Hand Book

This is a translation for this book from japanese to english.

It's made possbile by translating small chunks of texts using Google's Image translation services, to maintain original scanned resolution, and also, to improve translations, going paragraph by paragraph, editing page content to avoid bad translation in some texts because paragraphs spanning two pages.

Original japanese book was download from archive.org, so thanks to the people who scanned and made this book available.

I'm translating it to english to help this knowledge to be of use for a wider audience :-)

htdreams - 2025/06

ASCII Publishing Bureau

- MSX and MSX-DOS are trademarks of ASCII.
- MS-DOS is a trademark of Microsoft Corporation.
- OS-9 is a trademark of Microware Systems, Inc. and Motorola, Inc.
- TEX is a trademark of the American Mathematical Society.
- MicroTEX is a trademark of Addison-Wesley Publishing, USA.
- Other CPU names, system names, product names, etc. used in this manual are generally trademarks of their respective developers.

The TM and R marks are not stated in the text.

This book was typeset using ASCII's "Japanese TEX" except for the title page, colophon, and some illustrations. I would like to thank ishii@cts.dnp.co.jp, the author of msdos.sty, and the Publishing Technology Department. I would also like to thank Melhen Maker, who kindly agreed to do the illustrations on the title page.

In order to avoid making the book too long, I have omitted the description of "Japanese MSX-DOS2" and memory mappers. These will be explained in the "Japanese MSX-DOS2 Technical Handbook (tentative title)" to be released soon.

Introduction

Welcome to the world of MSX turbo R. This book provides detailed information on the internals of the MSX personal computer, which has become incredibly powerful thanks to its high-speed CPU and large-capacity memory, and is necessary to make the most of it.

1. This technique maximizes the performance of the R800, a high-speed CPU with a 16-bit internal configuration that delivers processing speeds more than 10 times faster than previous MSX models.

2. Information and techniques for making full use of the PCM and FM sound sources that come standard with the MSX turbo R.

3. The mechanism of the SLOT mechanism, which is essential for mastering the MSX, and how to use it.

4. The Kanji BASIC mechanism is necessary for developing software that handles Japanese.

5. How to master VDP to bring out your techniques in on-screen displays.

MSX turbo R was the first personal computer to achieve dramatically higher performance by upgrading the CPU to 16-bit without making major changes to the architecture of previous models.

Others changed their architecture completely when they went from 8-bit to 16-bit, which meant that all the software and know-how that had been developed by many people on 8-bit machines was thrown away.

We felt that it was necessary to make the CPU 16-bit to improve the performance of MSX, but that we should not throw away the software and hardware assets developed for MSX, nor the know-how of our users. In order to achieve this, we felt that a CPU that was upwardly compatible with the Z80 was necessary for the new MSX, and so we developed the R800. And to achieve complete compatibility with previous MSXs, we developed the MSX turbo R, which is equipped with the conventional Z80 as well as the newly developed R800.

In this way, MSX turbo R maintains ideal upward compatibility with the conventional MSX. Therefore, users can get many times the performance improvement by simply running the software assets they have accumulated so far on MSX turbo R.[1] In addition, the knowledge required for software development can be used as it is, but by using some of the know-how explained in this book, it will be possible to further extract the machine's performance and realize a system that demonstrates outstanding cost performance.

Ryozo Yamashita, General Manager of the 1st Product Division, Systems Business Division

[1] Commercially available software for MSX may not run as fast as the R800 would be too fast and would not be compatible, so it automatically switches to Z80 instead.

table of contents

1 MSX turbo R **15**

1.1 MSX turbo R hardware . . . 16

1.1.1 Here are the features of the MSX turbo R! . . . 16

1.1.2 MSX turbo R system configuration . . . 16

1.1.3 Elegant CPU switching . . . 18

1.1.4 Everything packed into MSX turbo R ROM configuration . . . 18

1.1.5 System timer to adjust speed . . . 19

1.1.6 MSX turbo R I/O port . . . 20

1.1.7 DRAM mode for maximum speed . . . 22

1.1.8 Here are the features of the R800!! . . . 23

1.1.9 All about the R800 . . . 23

1.2 MSX turbo R How to use . . . 27

1.2.1 Programming that takes advantage of the speed of the R800 . . . 27

1.2.2 Precautions and issues when using the R800 . . . 27

1.2.3 Added BIOS and its function description. . . . 28

1.2.4 What BIOS was changed or removed? . . . 31

1.2.5 Notes on application development . . . 32

1.2.6 Example of a program that switches CPUs . . . 33

1.3 How to use PCM to its limits . . . 37

1.3.1 Basics.....How to use in BASIC . . . 37

1.3.2 PCM-related BASIC instructions . . . 38

1.3.3 Play the BEEP sound with PCM! . . . 39

1.3.4 Advanced edition·····PCM in machine language! . . . 41

2 SLOT **47**

2.1 What is a slot? . . . 48

2.1.1 How is the CPU and memory connected? 48
2.1.2 Exploring the inside of the 8-bit CPU Z80 48
2.1.3 There are various types of memory depending on the function 50
2.1.4 What are MSX slots like? 50
2.1.5 The secret to MSX's expandability lay in its slots 52
2.1.6 The MSX2+ slot has changed 53
2.1.7 Expand your slots 55
2.2 Try switching slots 57
2.2.1 To switch slots 57
2.2.2 How to specify the slot number 57
2.2.3 BIOS functions to operate slots 58
2.2.4 How to know the slot configuration 60
2.2.5 Explore the system work area 61
2.2.6 MSX2+ hardware specifications 64
2.2.7 Device enable to prevent collisions 65
2.3 MSX Trubo R slot configuration 67
2.3.1 Finally, the slot configuration has been unified 67

3 Kanji BASIC 71
3.1 Analyze Kanji BASIC 72
3.1.1 Hardware required for Kanji BASIC 72
3.1.2 What is MSX-JE compatible software? 73
3.1.3 Explaining the operating principle of the Kanji driver 73
3.1.4 JE compatible hardware & software 75
3.1.5 Various screen modes available in Kanji BASIC 76
3.1.6 Kanji text and kanji graphics 77
3.1.7 This is the correct way to use the Kanji Driver 78

4 V9958 VDP 81
4.1 V9958 Register List. 83
4.2 What's new in V9958 85
4.2.1 horizontal scroll 85
4.2.2 Weight 87
4.2.3 command 87
4.2.4 YJK style display 87
4.3 Discontinued functions of V9958 89

4.4 V9958 Hardware Specifications (Changes) 90
4.5 V9958 and MSX2+ 91
4.5.1 There are 12 screen modes in total 91
4.5.2 Controls the VDP register 92
4.5.3 V9958 Register 95
4.5.4 Horizontal scrolling with VDP 95
4.5.5 Whatever you do, don't use any tricks 98
4.6 Dissecting the YJK Method 99
4.6.1 Television broadcasting and the YJK system 99
4.6.2 RGB and YJK data structures 99
4.6.3 color sample program 101
4.6.4 It's a deadly logical operation 103
4.6.5 So-called discoloration 105
4.6.6 What is the difference between SCREEN 10 and 11? 105
4.6.7 To use subtitles in SCREEN 11 106
4.6.8 A trick to display text on SCREEN 12 108
4.6.9 YJK method and VDP register 108
4.7 Studying Scanline Interrupts 110
4.7.1 How does the monitor screen display? 110
4.7.2 Interlaced TV broadcasting 111
4.7.3 Interlaced screen on MSX2 112
4.7.4 Exploring the principles of scan line interruption 113
4.7.5 Here is an example of scan line interruption 114
4.7.6 Let's finally get to the practical part! 116
4.7.7 VDP register used for scan line interrupts 116
4.7.8 How to assemble and how the BASIC part works 119
4.7.9 This is how the assembler part works 121
4.7.10 This is the machine routine for the scan line interrupt 128

5 MSX–MUSIC 129
5.1 What is an FM synthesis source? 130
5.1.1 The history of electronic musical instruments leading to FM synthesis. 130
5.1.2 Let's analyze the sound of the instrument 132
5.1.3 The pitch is not necessarily equal tempered 134
5.1.4 Let's analyze MSX-MUSIC 135
5.1.5 Try making rhythm sounds using FM sound source 137

5.2 Control FM sound source 139
5.2.1 Making sounds with a machine language program 139
5.2.2 Give an overview of the library 141
5.2.3 Let's compile with MSX-C 149
5.3 FM sound source data structure 150
5.3.1 Let's create FM sound data 150
5.3.2 To specify percussion data 152
5.3.3 Let's specify the instrument sound data 154
5.3.4 What the OPLL driver cannot do 156
5.3.5 Adding Sound Data 156
5.3.6 Explaining sample data 158
5.4 Various things related to FM synthesis 160
5.4.1 Correction to the content of the powerful usage method 160
5.4.2 List of MSX-MUSIC tone data 162

A R800 instruction table **165**
A.1 How to use the instruction sheet 166
A.2 8-bit movement instructions 168
A.3 16-bit move instruction 169
A.4 Exchange instruction 171
A.5 stack manipulation instructions 171
A.6 Block transfer instruction 172
A.7 block search instruction 172
A.8 multiplication instruction 172
A.9 addition instruction 173
A.10 subtraction instruction 175
A.11 comparison command 176
A.12 logical operation instructions 177
A.13 bit manipulation instructions 178
A.14 Rotate instruction 179
A.15 shift instruction 181
A.16 branch instruction 182
A.17 call instruction 183
A.18 input/output instructions 185
A.19 CPU control instructions 186

Figure table of contents

1.1 MSX turbo R system configuration 17
1.2 Changes in ROM configuration in MSX turbo R 19
1.3 R800 internal block diagram 25
1.4 Differences in memory access methods between Z80 and R800 26

2.1 Z80 CPU memory 49
2.2 MSX slot configuration (part 1) 51
2.3 MSX slot configuration (Part 2) 52
2.4 Example of MSX2+ slot configuration (when expanding only slot 3) . . . 54
2.5 Example of MSX2+ slot configuration (when expanding slots 0 and 3) . . . 55
2.6 How to specify slot number 58
2.7 device enable 65
2.8 MSX turbo R slot configuration 68

3.1 Kanji driver operating principle 75
3.2 Switching screen mode 78

4.1 Horizontal scrolling (when SP2=0) 85
4.2 Horizontal scrolling (when SP2=1) 86
4.3 List of control register functions added to V9958 96
4.4 Two types of side-scrolling mechanics 97
4.5 Data structure of RGB screen 99
4.6 Data structure of YJK style screen 101
4.7 Data structure of mixed screen 101
4.8 Scanning lines on a television screen 110
4.9 This is what happens in interlaced mode 112
4.10 This is the principle of scan line interruption 114

4.11 Scan line interrupt procedure 115
4.12 VDP register that generates a scan line interrupt 116
4.13 VDP register to detect scan line interrupt 117
4.14 VDP registers that control screen switching 117
4.15 Hardware vertical scrolling mechanism 118

5.1 Exploring the Structure of Four Types of Electronic Musical Instruments 131
5.2 Let's analyze the basic sound 132
5.3 Instrument and Synth Envelopes 134
5.4 Percussion instrument sound data 153
5.5 tone data 157
5.6 List of OPLL registers 161

table of contents

1.1 MSX turbo R I/O map 21
1.2 Comparing the operating speed of the Z80 and R800 24
1.3 List of BIOS and BASIC changes in MSX turbo R 32
1.4 I/O port for PCM 45

2.1 System work area for slots 61
2.2 MSX2+ I/O ports 64

3.1 MSX-JE built-in hardware list 73
3.2 Kanji BASIC screen mode 77
3.3 Hooks used by Kanji drivers 79

4.1 VDP mode and BASIC screen mode 82
4.2 Mode register 83
4.3 command register 84
4.4 status register 84
4.5 Change of terminal of V9958 90
4.6 DC characteristics of V9958 90
4.7 MSX2+ screen mode 91
4.8 VDP I/O Ports 92
4.9 Control register storage location 93
4.10 Other useful system work areas 94
4.11 System work area added and modified in MSX2+ 94
4.12 Details of 0FAFCH Address (MODE) 94
4.13 Logical operation 104

5.1 Comparing the performance of electronic musical instruments 131
5.2 Relationship between scale and frequency 134

5.3 List of temperaments that can be set in MSX-Music 135
5.4 Data structure for 6 instruments + 1 percussion sound 150
5.5 Example of data structure for 6 instruments + 1 percussion sound 151
5.6 9 Instrument sound data structure 152
5.7 Instrument sound data 155
5.8 Example of musical instrument sound data 155
5.9 Tone data list 163

第1章 MSX turbo R



This chapter is a re-edited version of the articles "MSX turbo R Technical Analysis" and "PCM Maximum Usage" from the November 1990 and December 1990 issues of MSX Magazine.

1.1 MSX turbo R hardware

The MSX turbo R has been much talked about, with its newly developed 16-bit CPU "R800", 256KB of main RAM, and MSX-DOS2 with hierarchical directory support as standard equipment. Here is an overview of the system configuration of this notable machine.

1.1.1 These are the features of the MSX turbo R!

- Z80 In addition, by incorporating the high-speed CPU "R800" which is upwardly compatible, it achieves an average speed of 4 to 5 times, and up to 10 times, faster (compared to MSX2+).

- Along with MSX-DOS1, it also comes with Japanese MSX-DOS2 and Kanji drivers, and supports MS-DOS compatible hierarchical directories and environment variables.

- It comes standard with 256KB of main RAM that supports memory mappers. The slot configuration has also been standardized.

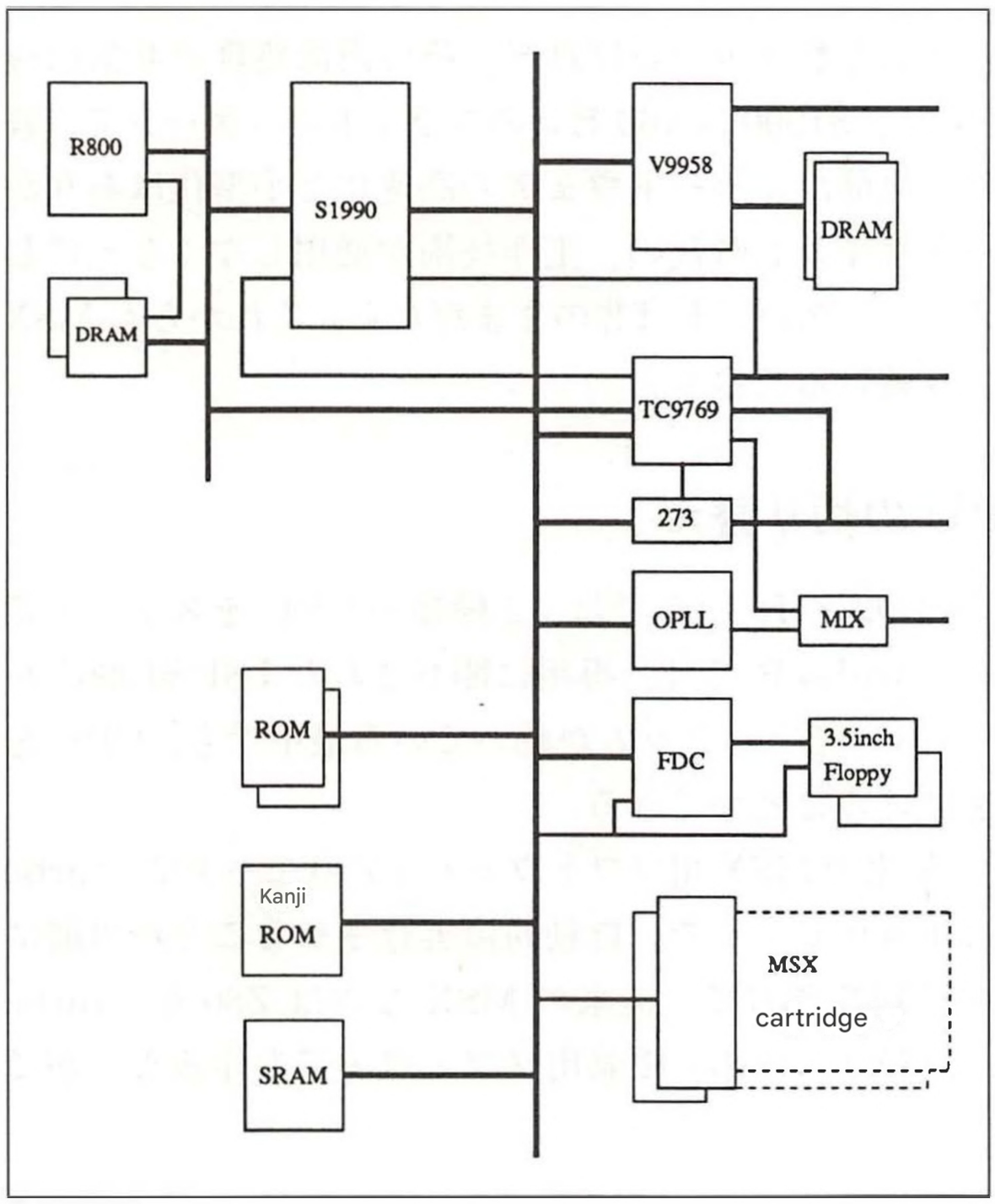
- PCM recording/playback function is standard equipment. MSX-MUSIC, which was previously optional, is now also standard equipment.

1.1.2 MSX turbo R system configuration

The hardware configuration of MSX turbo R (hereafter referred to as turbo R) is shown in Figure 1.1. It includes the same "Z80" compatible CPU as the previous MSX, and the newly developed "R800" CPU. Contrary to the rumors in the industry that "the next MSX will be equipped with Zilog's Z280 or Hitachi's HD64180 (both high-speed Z80-compatible CPUs)," ASCII actually made the CPU.

The performance of these hardware is comparable to older 16-bit machines, and the CPU speed is on par with the V30 (a 16-bit CPU developed by NEC). Also, putting the kanji conversion dictionary in ROM to save RAM and disk space is a traditional MSX design policy. It is also adopted in some recent notebook computers. To evaluate the turbo R hardware in one word, it would be "everyone has been aiming for this."

Figure 1.1: MSX turbo R system configuration



Here is a simplified representation of the turbo R hardware configuration, omitting all the detailed control signal lines. The output of the V9958 is the video signal, the lines connected to the TC9769 (Z80) are the keyboard and joystick, the output of the 273 is the printer port, and the output of MIX is the audio signal.

To explain the hardware configuration in a little more detail, first, the "TC9769" in the diagram is a Toshiba CMOS-LSI (a type of low-power digital LSI) judging from the model number. It is an LSI that includes a Z80-compatible CPU and a PSG sound source, commonly called the "MSX-Engine." In the following, whenever the term "Z80" appears in this book, it refers to this chip.

Below that, "273" is the bus buffer for controlling the printer, "OPLL" is the FM sound source, and "FDC" is the floppy disk controller. Also, "SRAM" is memory that stores the results of the kanji dictionary even when the power is turned off, but this SRAM and the compound phrase conversion dictionary are manufacturer optional features.

By the way, the R800 and main RAM are connected to the bus (the long vertical line in the diagram) through the S1990. For example, when the R800 operates the VDP, the S1990 relays the signal, and if necessary, sends a wait signal to the R800 to synchronize the signal timing with that of the Z80. Conversely, when the Z80 uses the main RAM, the S1990 and R800 relay the signal and handle memory mapping.

The reason why the turbo R has such a complex configuration is to maintain compatibility with existing hardware and software. I think they managed to do this. Bravo.

As you can see, the turbo R has few parts, but its internal processing has become very complicated. Furthermore, the S1990 is a 160-pin flat package, so it is impossible to solder it by hand. Although we are grateful for the faster and smaller hardware, the crafting techniques of the good old days of single-board microcomputers are no longer applicable. However, the MSX cartridge slot remains the same as it was in the past, so the MSX will continue to be a useful teaching material for beginners to hardware.

1.1.3 Elegant CPU switching

JVC's MSX2 machines, the HC-90 and HC-95, used two different CPUs, which could be switched with a switch. However, the turbo R uses a specially developed LSI, the "S1990", to manage the system, so even when the power is on and a program is running, the CPU can be switched and the program can continue to run.

Thanks to this hardware, it is possible to automatically run conventional MSX software in Z80 mode, and turbo R software in the faster R800 mode. It is also possible to create MSX2 / turbo R compatible software that checks the hardware type and selects Z80 for conventional MSX and R800 for turbo R.

1.1.4 Everything packed into MSX turbo R ROM configuration

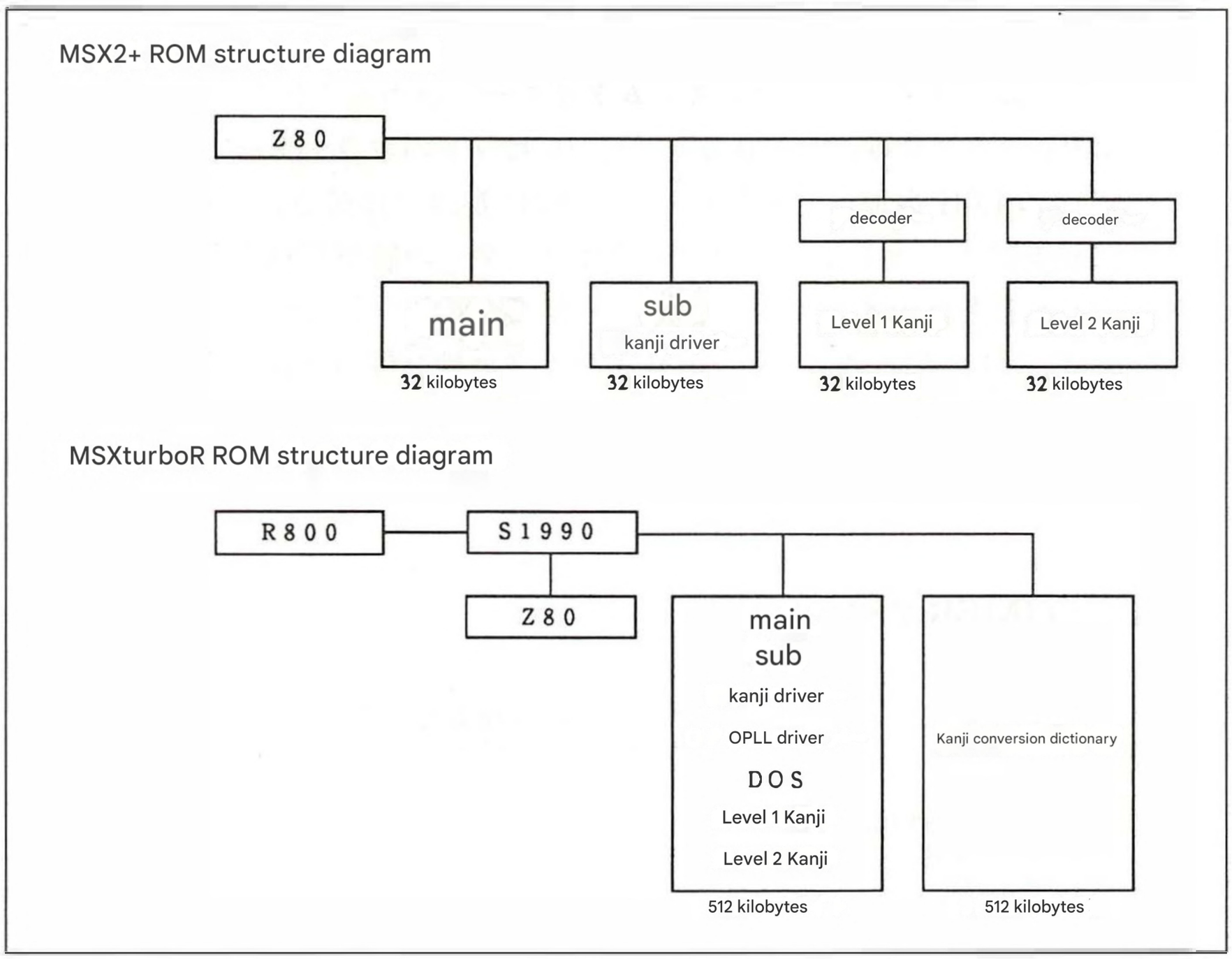
The turbo R should have a lot of ROMs built in, but when you open it up, you'll find that the number of ROMs is surprisingly small. The reason for this is the mega ROM control function of the S1990.

The MSX2+ has a built-in ROM like the one shown in the upper part of Figure 1.2. The main ROM and sub ROM are connected to different slots, and the kanji ROM is connected to an I/O port, so they need to be different ROMs regardless of the total capacity. However, using one 128KB ROM is cheaper than using four 32KB ROMs, and it also reduces the board area and power consumption.

So, in turbo R, the main, sub, OPLL driver, DOS, kanji level 1, and kanji level 2 were all packed into one 512KB ROM, as shown in the lower part of Figure 1.2. However, because of the mega ROM control function of the S1990, which is placed between the CPU and the ROM, from the software's point of view, for example, the kanji level 1 ROM appears to be connected to the I/O port addresses D8H and D9H.

In addition, a total of 64 KB of DOS ROM (16 KB for MSX-DOS and 48 KB for MSX-DOS2) was connected to the 16 KB space in slot 3-2 using a 4-bank switching system.

Figure 1.2: Changes in ROM configuration in MSX turbo R



1.1.5 System timer to adjust speed

When the R800 tries to use the V9958 (the LSI that controls the screen display) at intervals of less than 8 microseconds, the VDP interface circuit built into the S1990 automatically puts the R800 into a wait state. This means that there is no risk of the V9958 malfunctioning due to the CPU processing being too fast.

However, other peripheral LSIs do not have an automatic wait function, so the software itself must adjust the timing.

```
EX    (SP),HL
EX    (SP),HL
```

or

```
PUSH  HL
POP   HL
```

The timing was adjusted by embedding instructions that take time but have no side effects, such as the following, into the program. However, as I will explain later, the execution time of the R800's instructions is uncertain, so it is impossible to achieve timing in this way. Therefore, the turbo R has a "system timer" to adjust the speed.

This is a 16-bit counter that increments every 3.911 microseconds, with the lower byte connected to the I/O port at address E6H and the upper byte connected to address E7H. However, it would be inconvenient if the counter value changed while trying to read the 16-bit value, so it is best to use either the lower byte or the upper byte.

Listing 1.1 shows an example of a program that waits for 3.911 microseconds times the value of the B register. If you rewrite the program to use the high byte of the counter instead of the low byte, you can create a program that waits for 1001.2 microseconds times the value of the B register.

list 1.1 (TIMER.Z80)

```
        .Z80
COUNTLOW        EQU     0E6H    ;       Counter lower 8 bits
COUNTHIGH       EQU     0E7H    ;       Upper 8 bits of counter
;
; B register value 3.911uS wait
; The error is -3.911uS..+OS
; Do not specify 0
; Interrupts must be disabled
; C, A, F are destroyed
;
WAIT:
        IN      A,(COUNTLOW)    ; Get the current value of the counter
        LD      C,A             ; and save it
WAIT_LOOP:
        IN      A,(COUNTLOW)    ; Get the current value of the counter
        SUB     C               ; Calculate the elapsed
        CP      B               ; time Has a specified time elapsed?
        JR      C,WAIT_LOOP     ; Loop if it has not elapsed
        RET
```

1.1.6 MSX turbo R I/O port

The turbo R press release did not include an I/O map, so the editorial team added the information obtained from interviews and hardware analysis to the MSX2+ I/O map to create the I/O map shown in Table 1.1. Items with an "R" note are the I/O ports newly added to the turbo R.

First, the "D/A converter" is an I/O port for controlling PCM recording and playback without going through the BIOS. We'll explain the details later. "Pause key control" is an I/O port for disabling or allowing the pause key to stop a program. It appears to have been provided to prevent situations where a program is interrupted during disk input/output, resulting in the disk being destroyed.

Table 1.1: MSX turbo R I/O map

address	Purpose	Note	
00H～3FH	homemade hardware		
40H～7BH	Manufacturer options		
7CH～7DH	OPLL	+	B
80H～87H	RS-232C	1	B、X
88H～8BH	external VDP	2	XX
90H～93H	printer	2	
98H～9BH	VDP	+	
A0H～A2H	PSG	1	
A4H～A5H	D/A converter	R	
A7H	pause key control	R	
A8H～ABH	8255	1	B
ACH～AFH	MSX-Engine	2	
B0H～B3H	SONY の SRAM	1	XX
B4H～B5H	clock	2	
B8H～BBH	light pen	2	XX
BCH～BFH	VHD control	2	XX
C0H～C1H	MSX-Audio	2	XX
C8H～CCH	MSX-Interface	2	XX
D0H～D7H	floppy disk	2	—、XX
D8H～D9H	No. 1 level Kanji ROM	2	
DAH～DBH	No. 2 level kanji ROM	2	
DCH	Kanji ROM expansion	R	—、X
E3H～E5H	?	R	?
E6H～E7H	system timer	R	
F4H	reset status	+	B
F5H	device enable	2	
F6H～F7H	AV control	2	X
FCH～FFH	memory mapper	2	B

According to a survey by the MSX Magazine editorial team

1 $\mathbf{MSX_1}$ Compatible

2 $\mathbf{MSX_2}$ Compatible

+ $\mathbf{MSX_{2+}}$ Compatible

R **turbo R** Newly established

B Always operate through the BIOS.

— It should not be operated by application programs.

X This is a factory-installed option, but was not implemented in the turbo R prototype we investigated.

XX It was included in the previous model but has been removed from the turbo R model.

? Something is connected, but it's not mentioned in the specs. It looks like there's a register for hardware testing.

"Kanji ROM Expansion" seems to be a reserved function for 24-dot Kanji ROM and JIS Level 3 Kanji ROM that may be created in the future. The "?" below it is not written in any of the documents, but it seems that some hardware works inside the S1990 when reading and writing to the I/O port. I think it is an I/O port for testing turbo R hardware at the factory. And "System Timer" is as I have already explained.

Although this is not mentioned in the table, in order to keep up with the speed of turbo R, you should use the BIOS to operate the PSG, joystick, mouse, printer, keyboard, and clock (battery-backed clock IC).

Next, there is a feature shared with the MSX2+ that I would like to explain in more detail: the "reset status." This is an I/O port that distinguishes between a hardware reset and a restart caused by jumping to address 0 of the main ROM. Specifically, when address 17AH of the main ROM is called, the value of this reset status is read into the A register, and when address 17DH is called, the value of the A register is written into the reset status. For example,

```
CALL  17AH
OR    80H
CALL  17DH
RST   0H
```

By following this procedure, setting bit 7 of the reset status to 1 and then jumping to address 0, you can reliably restart the MSX.

By the way, the reason why you can't use the reset status without going through the BIOS is because the logic of the hardware signal for the reset status is reversed depending on the machine. The BIOS compensates for that difference.

With the turbo R, which comes standard with DOS2, the "memory mapper" has become increasingly important. It requires a somewhat complicated procedure to use the extended BIOS and to operate.

Finally, as a side note, Table 1.1 includes functions that were once put to practical use or prototyped, but are not included in recent MSXs. The author thinks that the recent MSX has become a standard computer, and there are too few cute peripherals, but what do you think?

1.1.7 DRAM mode for maximum speed

Each memory has a minimum time interval between reads and writes, called the "access time." If the CPU speed is too fast, a "wait" must be inserted to match the CPU speed to the memory. This access time varies by type, and the faster the memory, the more expensive it is. In general, RAM has a shorter access time than ROM.

To take advantage of the speed of the R800, it is better to store programs in RAM rather than ROM. Therefore, a "DRAM mode" was provided that transfers the contents of the BIOS, BASIC, sub-ROM, and Kanji driver ROMs to DRAM (main RAM) for use.

This separates the last 64KB of main RAM from the memory mapper, transfers the ROM contents, then write-protects it and connects it to the CPU. From the CPU's perspective, it appears that the normal ROM has been replaced with a high-speed ROM. When executing programs written in BASIC, the ROM containing the BIOS and BASIC interpreter is often used, so the speed of the DRAM mode can be utilized.

However, when running machine language programs, especially DOS programs, the time that ROM is used is relatively short, so it may be more advantageous to use the extra memory for a RAM disk or similar, rather than using DRAM mode.

Also, programs on ROM cartridges will run faster if they are transferred to RAM, but with turbo R disk-based software will likely become more mainstream than ever before.

1.1.8 Here are the features of the R800!

- It is object-compatible with the Z80, so Z80 software will also work, except for parts that depend on CPU timing.

- The CPU clock speed is 7.16 MHz. However, since the number of clocks per instruction is significantly reduced compared to the Z80, it is equivalent to 29 MHz in Z80 terms (when there are no wait states).

- Supports multiplication instructions with precision from 16 bits x 16 bits to 32 bits, enabling a significant improvement in calculation processing speed.

- Access to the upper/lower 8 bits of the IX/IY registers, which was undefined on the Z80, is now officially guaranteed.

1.1.9 All about R800

The R800 used as the CPU for turbo R is a high-speed CPU that is software compatible with the conventional Z80. In other words, unless the CPU is too fast, software developed for the Z80 can be run at high speed on the R800 as is.

The features added to the Z80 include a 16-bit multiplication instruction and an instruction for byte access of the IX/IY register, which was considered a "trick" in the Z80. For details, please refer to the R800 instruction table in the appendix of this book.

The clock frequency of the conventional MSX is 3.58 MHz, and that of the turbo R is 7.16 MHz. From this alone, it seems that the speed has only doubled, but in fact, that is not the case. The number of clocks required to execute one instruction is reduced with the R800, and since there is no M1 cycle wait to access RAM, program execution speed is even faster. With the conventional Z80, a clock frequency of about 29 MHz is required to achieve the same processing speed as the R800, so this is a significant speed increase.

Table 1.2: Comparing the operating speeds of Z80 and R800

instruction		MSX2+ (unit μs)	turbo R (unit μs)	magnification
LD	r,s	1.40	0.14	x10.0
LD	r,(HL)	2.23	0.42	x 5.3
LD	r,(IX+n)	5.87	0.70	x 8.4
PUSH	qq	3.35	0.56	x 6.0
LDIR	(BC $\neq$ 0)	6.43	0.98	x 6.6
ADD	A,r	1.40	0.14	x10.0
INC	r	1.40	0.14	x10.0
ADD	HL,ss	3.35	0.14	x24.0
INC	ss	1.96	0.14	x14.0
JP		3.07	0.42	x 7.3
JR		3.63	0.42	x 8.7
DJNZ	(B $\neq$ 0)	3.91	0.42	x 9.3
CALL		5.03	0.84	x 6.0
RET		3.07	0.56	x 5.5
MULTU	A,r	—	1.96	—
MULTUW	HL,rr	—	5.03	—

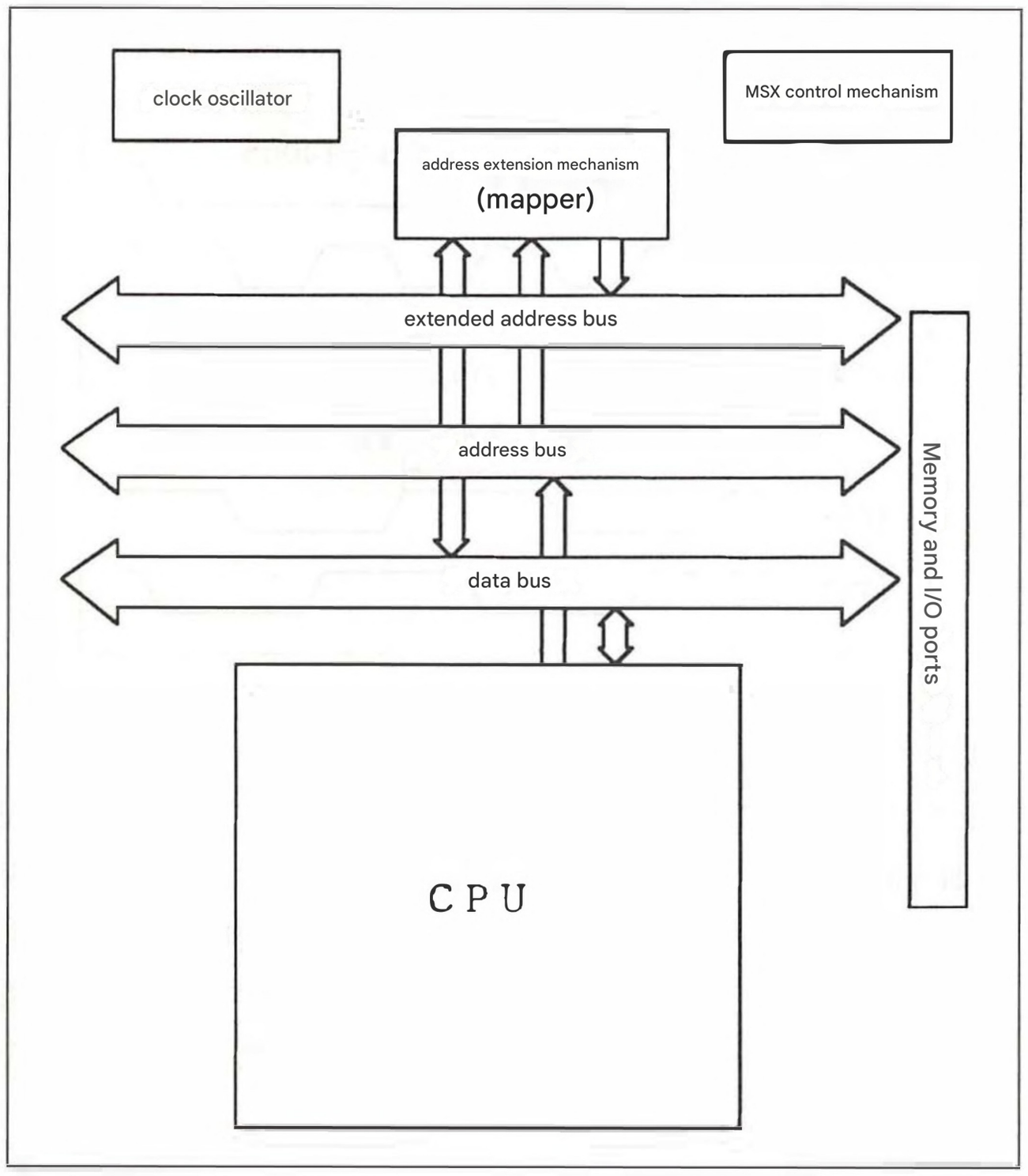
Table 1.2 shows the results of comparing the speed of the Z80 and the R800 for each type of instruction. It is worth noting that data transfer between registers (LD instruction) and addition are 10 times faster. However, the values in this table are measurements of the speed when the R800 operates with no waits. In reality, the speed may decrease due to waits, so be careful. The conditions under which waits occur and how to avoid them will be explained in detail later.

The internal structure of the R800 is shown in Figure 1.3. In the R800, the external data bus is 8 bits, but the data bus inside the CPU is 16 bits, so a 16-bit add instruction is processed in one cycle.

Looking at this hardware configuration, the R800 appears to be closer to 16-bit CPUs with 8-bit external data buses, such as Intel's "8088" or Motorola's "MC68008," than to the 8-bit CPU of the Z80.

At the top of Figure 1.3, there is something called an "address extension mechanism (mapper)", but this seems to have been prepared so that the R800 could be used for things other than MSX. When using it with turbo R, the slot control mechanism and memory mapper built into the S1990, not the R800, will control the system.

figure 1.3: R800 internal block diagram

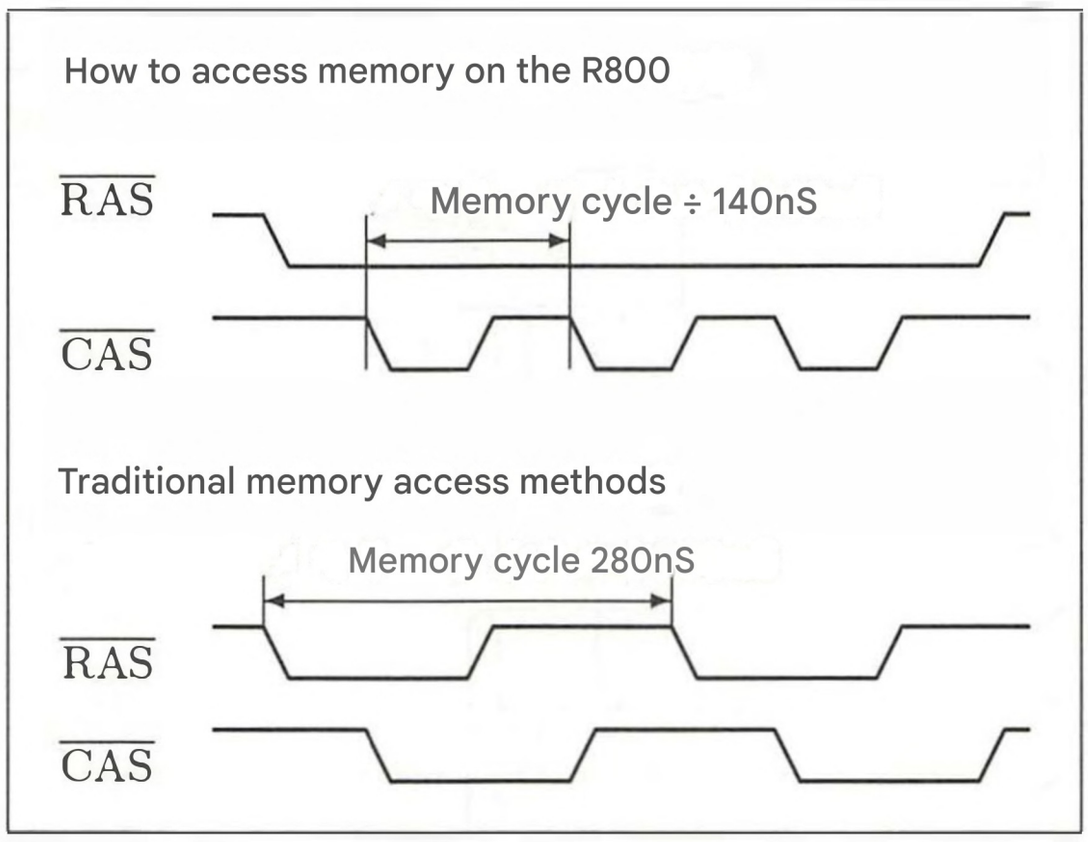


Next, let's explain "DRAM page access" in detail. First, the bottom part of Figure 1.4 shows the previous method of memory access using the Z80. The upper byte of the address (row address) is sent to the DRAM, the RAS (row address strobe) signal is set to LOW, the lower byte of the address (column address) is sent to the DRAM, and then the CAS (column address strobe) signal is set to LOW. This specifies the memory address.

Meanwhile, the upper part of Figure 1.4 shows DRAM page access on the R800. By keeping the upper byte of the address and the RAS signal fixed and varying only the lower byte of the address and the CAS signal, the DRAM is used twice as fast as with the conventional method. In this way, with the R800, when the DRAM is used continuously without changing the upper byte of the address, page access is performed automatically.

Now, the types of DRAM that are easy to connect and use with the R800 include 256kbit (32kB), 1Mbit (128kB), 4Mbit (512kB), etc. Even though the turbo R has a minimum main RAM capacity of 256kB, you can get by with just two 1Mbit DRAMs.

Figure 1.4: Differences in memory access methods between Z80 and R800



The first MSX, developed in 1983, used eight 16-kilobit DRAMs, but the main RAM capacity was only 16 kilobytes. Considering that, today's technology is amazing, as it can achieve a RAM capacity of 256 kilobytes with just two DRAMs. The functionality of the MSX has increased, but the size and power consumption of the hardware has decreased. The application of Japan's latest semiconductor technology can be seen in the much talked about notebook computers and turbo R.

1.2 MSX turbo R How to use

1.2.1 Programming that takes advantage of the speed of the R800

The R800 is certainly fast, but to get the most out of its speed, that is, to avoid waits and utilize the capabilities of the R800, you need to be creative with your programming. Remember that there are three waits for access to an external slot, two waits for access to the internal ROM, and one wait when the internal DRAM cannot be page-accessed.

Ideally, programs should be placed in the same 256-byte range (page-accessible range) of the upper bytes of the internal RAM addresses, and data should be placed in the registers. In this case, no memory access is required for data, and memory access for the CPU to read the program is also performed in page mode, so there is no wait time for the CPU. It is difficult to write all programs like this, but it is a good idea to try to get as close as possible to this condition for at least the subroutines, which require the highest speed.

Now, whether a page can be accessed or not depends on the addresses of the program, data, and stack. do. for example,

```
PUSH    HL
```

The execution time of an instruction is 4 clocks if the high byte of the address where that instruction is located matches the high byte of the stack pointer. If they don't match, it's 5 clocks. There's probably little need to think this through when creating a program, but it's important to remember that the execution time of an instruction varies depending on the situation.

1.2.2 Precautions and issues when using the R800

The Z80 refreshed the DRAM after each instruction, but the R800 takes 280 nanoseconds to refresh the DRAM every 31 microseconds. Note that because of the time it takes to refresh, and the DRAM page accessibility conditions mentioned above, it is not possible to accurately predict the execution time of an R800 program.

So to adjust the speed of the program, we will use something called a "system timer". I will explain how to use this system timer and how to adjust the speed between the CPU and the VDP later, so please wait.

Also, as with any new CPU, one of the problems with the R800 is the lack of development equipment. In particular, it is inconvenient that the "ICE (in-circuit emulator)" which is so useful when developing software cannot be used for debugging.

Therefore, to create software for the turbo R, it is best to first thoroughly debug using the conventional MSX and Z80 ICE, and then rewrite the programs that should work for the turbo R. Create a program that can also be used for the Z80, check that it works, and then rewrite only the parts that use multiplication for the R800. At this point, it is also a good idea to break it down into subroutines and check their operation. Then, if you put the whole thing together and it doesn't work, you'll have to look at the source listing and think about it.

1.2.3 Added BIOS and its function description

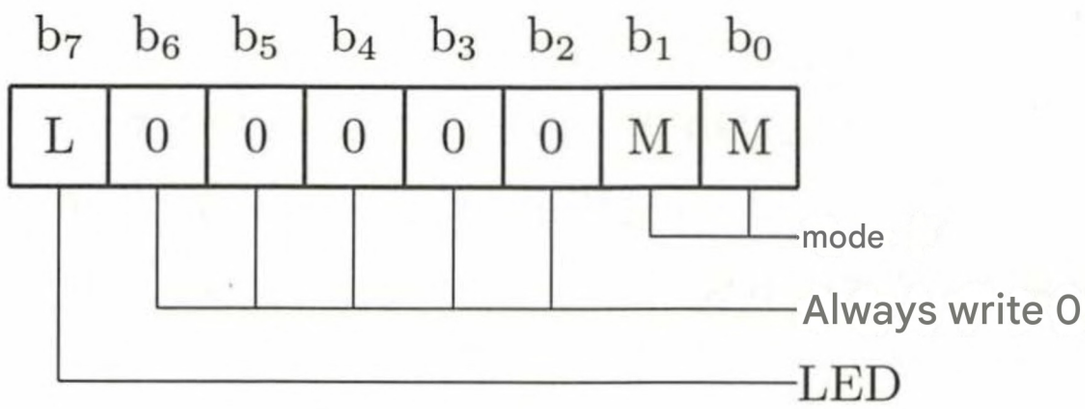
To control the turbo R's new hardware features, a BIOS was added for CPU switching and PCM recording and playback.

Here, we will explain the BIOS name (label), entry address, function, and each register in that order. The symbols used to describe BIOS functions are as follows. First, E is a register that should be set before calling the BIOS. R is a register to which the BIOS returns a value, and M is a register to which the BIOS writes a meaningless value, i.e., the original contents are destroyed. IYH represents the upper byte of the IY register, and the contents of the lower byte are ignored.

CHGCPU 0180H street address

function Switch CPU.

E Bits 1 and 0 of the A register set the mode as follows. "R800 DRAM" is a mode in which the contents of the BIOS ROM are transferred to DRAM.



mode	
00	Z80
01	R800 ROM
10	R800 DRAM

Also, if bit 7 of the A register is 1, the LED indicating which CPU is running will change. Conversely, if bit 7 of the A register is 0, the CPU will be switched, but the LED will not change.

R none

M AF

Note The contents of the registers before switching the CPU are carried over to the new CPU, except for AF and R. Also, interrupts are enabled after switching. Note that you should take note of the following points when switching CPUs, which will be explained in detail later.

GETCPU 0183H address

function Check the running CPU.

E none

R Depending on the CPU being used, the following values will be returned in the A register:

0	Z80
1	R800 ROM
2	R800 DRAM

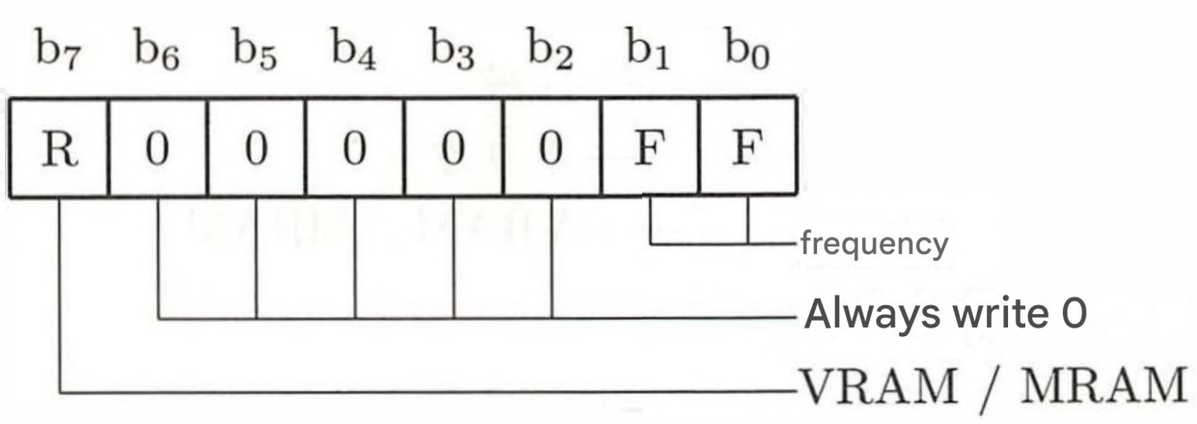
M F

Note You need to make sure your hardware is turbo R as explained later before calling up this BIOS.

PCMPLY 0186H address

function Plays PCM sound.

E A



EHL (data address)

DBC (data length)

If bit 7 of the A register is 1, the PCM audio data is stored in the video RAM, if it is 0, the PCM audio data is stored in the main RAM. Note that the values of the D and E registers are only meaningful if there is data in the video RAM.

Bits 1 and 0 of the A register set the sampling frequency, with 15.75 kHz only available when the turbo R is running in R800 DRAM mode.

00	15.75	kilohertz
01	7.875	kilohertz
10	5.25	kilohertz
11	3.9375	kilohertz

R carry flag

0 Normal termination

1 Abnormal termination

A (Cause of abnormality)

1 Frequency specification error

2 Interruption with STOP key

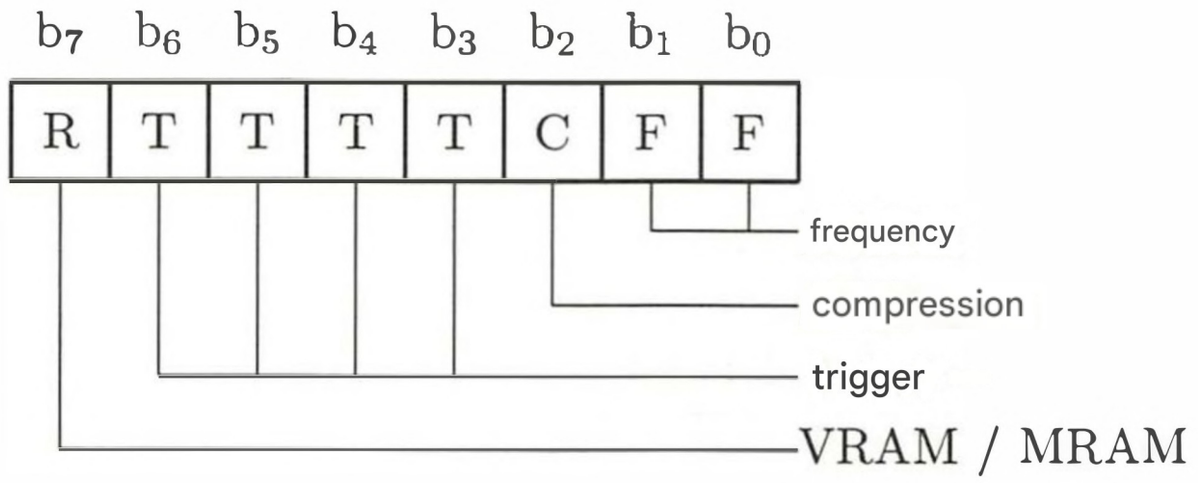
EHL (interrupted address)

M all

PCMREC 0189H address

function Records PCM sound.

E A



EHL (data address)

DBC (data length)

The method for setting bits 7, 1, and 0 of the A register is the same as that explained for PCMPLY. Bits 6 to 3 of the A register are called the "trigger level" and specify the volume of the sound that triggers recording to begin. If this value is 0, recording will begin immediately.

Also, if bit 2 of the A register is 1, the recorded data is compressed. If it is 0, it is not compressed.

R — carry flag

- 0 Normal termination
- 1 Abnormal termination

A (Cause of abnormality)

- 1 Frequency specification error
- 2 Interruption with STOP key

EHL (interrupted address)

M — all

1.2.4 What BIOS was changed or removed?

Table 1.3 lists the BIOS that were changed or removed in turbo R. We will briefly explain each one below.

First, the turbo R no longer has a cassette tape interface, so if you call "TAPION", "TAPIN", "TAPIOF", "TAPOON", "TAPOUT", or "TAPOOF" that were in the previous BIOS, the carry flag will be set and the program will return with an error. Also, "STMOTR" no longer exists, so if you call it, it will return without doing anything.

In addition, the paddle and light pen BIOS were removed to allow new features to be added without changing the capacity of the main ROM. When the BIOS "GTPDL" is called, the A register is always set to 0 and the function returns. Similarly, when "GTPAD" or "NEWPAD" is called with a value of 8 to 11 in the A register to specify the light pen, the A register is always set to 0 and the function returns.

The BIOS that was changed is the "ROM version ID" to know the version of MSX being used. This can be found in the contents of address 002DH of the main ROM, and has been changed to 03H in the case of turbo R. If you are developing a program for turbo R, first make sure that the value of this address is 03H or higher. If it is not, run it as an MSX2 program, or display an error message and abort.

Note that programs that only work when the content of address 002DH is 03H will no longer work when MSX is updated in the future, so they must be written to work on 03H or higher. In general, when it comes to hardware and OS versions, you should program your software to work if it obtains a value equal to or higher than the version number you require.

Table 1.3: List of BIOS and BASIC changes in MSX turbo R

Added BIOS entries	
CHGCPU	0180H
GETCPU	0183H
PCMPLY	0186H
PCMREC	0189H

Modified BIOS entries	
ROM version ID	002DH

Deleted BIOS Entries	
GTPDL	00DEH
TAPION	00E1H
TAPIN	00E4H
TAPIOF	00E7H
TAPOON	00EAH
TAPOUT	00EDH
TAPOOF	00F0H
STMOTR	00F3H
GTPAD	00DBH
NEWPAD	SUB 01ADH

Added statement
CALL PCMREC CALL PCMPLAY CALL PAUSE

modified statement
COPY

deleted statement
CLOAD CSAVE MOTOR

This has actually happened in the past, but due to incorrect checking of the MSX version, some MSX2 programs did not work on MSX2+, and some applications did not work when combined with MSX-JE with learning function. To avoid this, please remember to make sure that the program works on 03H or later.

As with the BIOS, some BASIC features have been added, changed, or removed in the turbo R. For details, see Table 1.3 or the BASIC manual that came with your machine.

1.2.5 Notes on application development

In MSX turbo R, the R800 does not always operate with no waits. There are three waits when accessing an external slot, two waits when accessing internal ROM, and one wait when the internal DRAM causes a page break. Therefore, to speed up a program, you must work while thinking about how to reduce these waits as much as possible. Here are three points to keep in mind for this purpose.

The first is to transfer the program itself to RAM before executing it. Software provided on floppy disks is not a problem because it will inevitably run in RAM, but you need to be careful with programs provided on ROM cartridges in the slots. By transferring only the necessary parts to RAM before executing them, it is possible to significantly increase the speed.

It is also important to code in a way that does not cause a page break. The R800 has a dedicated bus that supports DRAM page access, so make the most of this function. Specifically, it is effective to program memory access so that only the lower 8 bits of the address change, that is, in the range of 256 bytes from ??00H to ??FFH.

By the way, a page break occurs when memory access is performed outside this range, in other words, when the upper 8 bits of the address change.

As I mentioned briefly before, unlike MSX2+, turbo R does not know the exact execution time of instructions at the program coding stage. The reason for this is that DRAM page breaks occur unpredictably, and unlike Z80, DRAM refresh is performed asynchronously with instruction execution.

Also, when creating a program that will run on both turbo R and MSX2+, it is not recommended to use a software loop for timing. Therefore, turbo R now has a new system timer that counts up every 3.911 microseconds. From now on, let's use this system timer for timing.

1.2.6 Example of program to switch CPU

Listing 1.2 is the source listing for “CHGCPU.COM” which switches CPU. When DOS2 is running in turbo R,

```
CHGCPU 0
```

In Z80 mode,

```
CHGCPU 1
```

The ROM mode of the R800 is

```
CHGCPU 2
```

The R800's DRAM mode is selected with these. The program gets the first character of the first parameter of the command from address 5DH of the DOS work area (specifically the default FCB area), sets the value of the A register accordingly, and calls the BIOS "CHGCPU" at address 180H of the main ROM.

To make the program more practical, a process to check the DOS version number was added. Specifically, it first checks that the content of address 2DH in the main ROM is 03H or higher, meaning it is turbo R, and then uses the DOS system call 6FH to check that the DOS kernel version number is 2 or higher.

list 1.2 (CHGCPU.Z80)

```
        .Z80
RDSLT   EQU     0000CH          ; inter slot read
CALSLT  EQU     001CH           ; inter slot call
EXPTBL  EQU     0FCC1H          ; slot # of main ROM
;
        ld      a,(EXPTBL)      ;
        ld      hl,2dh          ; address to read
        call    RDSLT           ; read version
        cp      3
        jr      nc,TURBOR
        ld      de,MSG_NOTR
        ld      c,9             ; _STROUT
        call    5
        rst     0               ; return to DOS
TURBOR:
        ld      c,6fh           ; _DOSVER
        call    5               ;
        ld      a,b             ; version of DOS kernel
        cp      2
        jr      c,NOTDOS2       ;
        ld      a,d             ; version of MSXDOS.SYS
        cp      2
        jr      c,NOTDOS2

        ld      a,(005ch+1)     ; command parameter
        sub     '0'             ; 0:Z80, 1:R800ROM, 2:R800RAM
        ret     c               ; abort if parameter < '0'
        cp      3
        ret     nc              ; abort if '3' <= parameter
        or      80h               set change-LED flag
        ld      ix,180h         ; address of CHGCPU
        ld      iy,(EXPTBL-1)   ; slot of main ROM
        call    CALSLT          ; inter-slot call
        rst     0               ; return to DOS
;
NOTDOS2:
        ld      de,MSG_NOTDOS2
        ld      c,9             ; _STROUT
```

```
        call    5
        rst     0               ; return to DOS
;
MSG_NOTR:
        DB      'not MSX turbo R', 0dh, 0ah, '$'
MSG NOTDOS2:
        DB      'not MSX-DOS 2', 0dh, 0ah, '$'
        END
```

Similarly, the following Listing 1.3 is the source listing for "GAMEBOOT.COM", which tricks MSX2 programs into running in R800 mode. This program is used to start programs on other disks when DOS2 is started and R800 is selected. In other words, it is used to forcibly run programs (such as games) that do not include the DOS2 system in R800 mode.

To briefly explain the program, it first displays a message on the screen and waits for the disk to be swapped. Then it reads the boot sector of the swapped disk and executes it. The environment is the same as when the boot sector is called the second time in the normal way: page 1 is DOS ROM, the other pages are RAM, and the carry flag is set.

In addition, the DOS work area (address F323H) for storing the pointer to the error handling program is set in the HL register, and the address (address F368H) of the program that switches page 1 from RAM to DOS ROM is set in the DE register.

list **1.3 (GAMEBOOT.Z80)**

```
        .z80

_conin          equ     01h
_strout         equ     09h
_setdta         equ     1ah
_rdabs          equ     2fh

dos             equ     0005h
enaslt          equ     0024h

notfirst        equ     0f340h
master          equ     0f348h

        ld      sp,(6)
        ld      de,prompt       ; print prompt message
        ld      c,_strout
        call    dos
        ld      c,_conin        ; wait for key in
        call    dos
        ld      de,0c000h       ; read boot sector at 0c000h
        ld      c,_setdta
        call    dos
        ld      de,0            ; logical sector 0
        ld      l,0             ; drive A:
        ld      h,1             ; read 1 sector
        ld      c,_rdabs
        call    dos
        ld      h,40h
        ld      a,(master)
        call    enaslt
        ld      hl,0f323h
        ld      de,0f368h
        xor     a
        ld      (notfirst),a
        scf
        jp      0c01eh
prompt:
        db      'Insert game disk in drive A:,',0dh,0ah
        db      'and press any key $'

        end
```

1.3 How to use PCM to its limits

A new feature added to turbo R is PCM. It is human nature to want to get the most out of this feature. Here, we will introduce how to use PCM, from BASIC to machine language and special uses that utilize horizontal scan line interrupts.

1.3.1 Basics: How to use in BASIC

Let's start by looking at some basic things using BASIC.

In the first place, PCM converts audio input from a microphone or other device into digital data and stores it in memory. It stores the information in a file and allows you to play it back at will.

In turbo R, PCM data is stored in either main RAM or video RAM. There are four sampling rates to choose from: 15.75 kHz, 7.875 kHz, 5.25 kHz, and 3.9375 kHz. The higher the value, the higher the quality of the sampling.

To use PCM from BASIC, there are two commands you need to remember. Instructions on how to use them are listed below, so please refer to them. Basically, you can record and play PCM just by executing these commands. However, you need to be very careful when setting the start and end addresses where the data will be stored.

First of all, you must reserve memory space for PCM data with the BASIC "CLEAR" command, otherwise the program will definitely go out of control. For example, if you want to use addresses C000H to D000H for PCM data,

```
CLEAR 200,&HC000
```

For now, let's put a simple sample program in List 1.4.

I've prepared this for you, so you can enter it and have a play around with it.

Of course, if you use video RAM for PCM data, you can put the data at any address, so you don't need to worry about the start and end addresses. Also, with video RAM, you can see the PCM data. First,

```
SCREEN 8
```

If you set the screen mode like this and then record PCM, the data will be displayed on the screen in a row, which might be interesting.

If you pay attention to the above, you can record and play back the basic PCM. You can also change the playback sampling rate to play back at four different speeds. However, the problem occurs when playing back PCM. The turbo R is completely occupied with that, so unfortunately you can't do anything while playing back PCM.

list 1.4 (PCM1.BAS)

```
10 CLEAR100,&H9000
20 PRINT " It's time for a new start.";
30 A$=INPUT$(1):PRINT
40 _PCMREC  (@&H9000,&HCFFF,0)
50 PRINT " Restart.";
60 A$=INPUT$(1):PRINT
70 _PCMPLAY (@&H9000,&HCFFF,0)
80 GOTO 20
```

1.3.2 PCM related BASIC instructions

CALL PCMREC

Format

- Record to main RAM or video RAM.

  CALL PCMREC(@start address, end address, sampling rate [, [trigger level], compression switch [, S])

- Recording to an array variable.

  CALL PCMREC(Array variable name, [Length], Sampling rate [, [Trigger level], Compression switch])

Setting the sampling rate	
Specified value	sampling rate
0	15.7500KHz
1	7.8750KHz
2	5.2500KHz
3	3.9375KHz

The trigger level sets the input level when recording begins. The value can be from 0 to 127. Recording will begin when the input level reaches or exceeds this value, and will begin immediately if this is 0 or omitted. The compression switch is set to 1 to compress silent parts, and 0 or omitted to not compress.

CALL PCMPLAY

Format

- Playback from main RAM or video RAM

  CALL PCMPLAY(@start address, end address, sampling rate [, S])

- Playback from array variables

  CALL PCPLAY(Array variable name, [length], sampling rate)

For both PCMREC and PCMPLAY, if the mode is not high-speed, it will temporarily switch to high-speed mode before execution, and will return to the original state when it is finished. Also, if 15.75KHz is specified in the ROM mode of the R800, an error will occur.

If the STOP key is pressed during recording or playback, program execution will be interrupted. In the PCM data format, normal data is from 1 to 255, with 0 being special. The following byte outputs level 0 (127) the number of times specified.

1.3.3 Play BEEP sounds via PCM!

You know that when you press the "CTRL" and "STOP" keys simultaneously while executing a BASIC program to interrupt the program, a "beep" sound is emitted. The same "beep" sound is emitted when you display a list with the "LIST" command and stop the program with "CTRL + (STOP)". You can change the sound by using the "SET BEEP" command in BASIC, but none of the four sounds available are particularly impactful.

So, what happens if you play this BEEP sound with PCM? If you set it to some funny dialogue, the MSX will talk at every opportunity, which might be pretty loud and fun.

So, if you run the program in Listing 1.5, you will be able to play the BEEP sound on the PCM. Of course, this is only for turbo R. It's not a very long program, so please try your best to type it in.

This program is stored in page 1 of the main RAM (addresses 4000H to 60FFH), so after executing the program, you can use the "CLEAR" command to set the upper limit of the user area above address B000H. However, you cannot use memory disks, so be careful not to accidentally use "CALL MEMINI". Also, when using the BASIC "BEEP" command,

```
PRINT CHR$(7)
```

If you do not use it instead, the BEEP sound will not be PCM.

Explain how to use the program.

1 PCM BEEP set

After this, the BEEP sound will be PCM. Once you execute this command, the setting will remain valid until the power is turned off. You can also change the CLEAR statement setting.

2 PCM BEEP reset

BEEP Returns the sound to its original state. Be sure to execute this command when you change to DOS or DOS2 with “CALL SYSTEM”.

3 PCM data playback

Plays the currently set PCM data. Use it for confirmation.

4 PCM data recording

Record PCM data at 15.75 kilohertz. When recording, memory from addresses B000H to CFFFH is used.

5 PCM data LOAD

Reads PCM data saved in BSAVE format with the extension ".PCM".

6 PCM data SAVE

The PCM data recorded using "PCM Data Recording" is written to a disc.

0 END

End the program. You can also use CTRL + STOP.

A simple message will be displayed on the screen for your reference.

list **1.5 (PCM2.BAS)**

```
10 SCREEN0:WIDTH40:DEFINT A-Z
20 CLEAR100,&HB000
30 DEFUSR=&HD800:DEFUSR1=&HD806:DEFUSR2=&HD803
40 FOR I=&HD800 TO &HD87F
50 READ A$:POKE I,VAL("&H"+A$):NEXT
100 PRINT
110 PRINT"1) PCM BEEP set "
120 PRINT"2) PCM BEEP reset"
130 PRINT"3) PCM DATA playback"
140 PRINT"4) PCM DATA recording "
150 PRINT"5) PCM DATA LOAD"
160 PRINT"6) PCM DATA SAVE"
170 PRINT"0) END"
180 PRINT" ....HIT 0-6 KEY=";
190 A$=INPUT$(1):I=ASC(A$)-ASC("0")+1
200 IF I>0 AND I<8 THEN ELSE190
210 ON I GOTO 230,240,310,220,340,390,430
220 PRINTCHR$(7);:GOTO 190
230 GOSUB 470:END
240 GOSUB 470:I=USR1(0):I=USR(0)
250 PRINT"PCM BEEP can be used."
260 PRINT"When using DOS or DOS2, please reset PCM BEEP"
270 PRINT"PCM BEEP / Nearing data page1 (4100H to 60FFH)."
280 PRINT"The CLEAR command can be set to above B000H without any issue,
but commands related to memory disks, such as CALL MEMINI, cannot be
executed."
```

```
290 PRINT"Now, to execute the BEEP command, please use PRINT CHR$(7)."
300 END
310 GOSUB 470:POKE &HFDA4,&HC9
320 PRINT"PCM BEEP has been reset."
330 GOTO 100
340 GOSUB 470
350 PRINT"Starting PCM recording. (HIT ANY KEY!)";
360 A$=INPUT$(1):PRINT:_PCMREC(@&HB000,&HCFFF,0):I=USR(0)
370 PRINT"Recording completed."
380 GOTO 100
390 GOSUB 470
400 PRINT"PCM data LOAD"
410 INPUT" FILE NAME (8 characters)=";A$
420 BLOAD A$+".PCM":I=USR(0):GOTO 100
430 GOSUB 470
440 PRINT"PCM data SAVE"
450 INPUT" FILE NAME (8 characters)=";A$
460 I=USR2(0):BSAVE A$+".PCM",&HB000,&HCFFF:GOTO 100
470 PRINT CHR$(I+47):PRINT:PRINT:RETURN
480 DATA C3,4D,D8,C3,5F,D8,CD,6D
490 DATA D8,3A,42,F3,32,2A,D8,21
500 DATA 2E,D8,11,00,40,01,00,01
510 DATA ED,B0,CD,76,D8,21,29,D8
520 DATA 11,A4,FD,01,05,00,ED,B0
530 DATA C9,F7,00,00,40,C9,FE,07
540 DATA C0,01,00,20,21,00,41,3E
550 DATA 03,D3,A5,F3,DB,A4,D6,01
560 DATA 38,FA,7E,D3,A4,23,0B,79
570 DATA B0,20,F1,FB,C9,CD,6D,D8
580 DATA 21,00,B0,11,00,41,01,00
```

1.3.4 Advanced level: PCM in machine language!

The easiest way to use PCM in machine language is to use the BIOS. The settings for sampling rate, trigger level, etc. are almost the same as those in BASIC, so there should be no problem.

Since this is an advanced topic, we will introduce two programs that can record and play PCM without using the BIOS.

When using the BIOS, you can only choose from four sampling rates: 15.75 kHz, 7.875 kHz, 5.25 kHz, and 3.9375 kHz. This is because the BIOS uses a counter whose value changes every 63.5 microseconds, so you cannot set more than four sampling rates. The program introduced here replaces this counter with a system timer whose value changes every 3.911 microseconds.

First, let's explain how to use the recording program. The HL register is set to the top address of the memory that stores the PCM data, and the BC register is set to the size of the data to be recorded. The E register is set to the number of counts to wait in the system timer. 16 is roughly equivalent to 15.75 kHz.

The playback program is the same. Set the start address of the PCM data to be played in the HL register, the size of the data in the BC register, and the wait count in the E register.

In the middle of the list of PCM recording programs,

```
OEDH,70H
```

There is a strange thing called

```
IN (HL),(C)
```

This is an instruction unique to the R800 that reads a value from the port of the C register and reflects it only in the flags.

Regardless of the principles of the program, try using it. By changing the value of the E register, you can enjoy changing the sound in various ways.

list 1.6 (PCMREC.MAC)

```
PMDAC   EQU 0A4H
PMCNT   EQU 0A4H
PMCNTL  EQU 0A5H
PMSTAT  EQU 0A5H
SYSTML  EQU 0E6H        ; system timer port

REC:
        LD    A,00001100B
        OUT   (PMCNTL),A        ; A/D MODE
        DI
        XOR   A
        OUT   (SYSTML),A        ; reset timer

REC1:
        IN    A,(SYSTML)
        CP    E
        JR    C,REC1            ; wait
        XOR   A
        OUT   (SYSTML),A        ; reset timer

        PUSH  BC
        LD    A,00011100B
        OUT   (PMCNTL),A        ; DATA HOLD
        LD    A,80H
        LD    C,PMSTAT

        OUT   (PMDAC),A         ; BIT CONVERT
```

```
        DEFB  0EDH,70H          ; IN (HL),(C)
        JP    M,RECAD0
        AND   01111111B
RECAD0:
        OR    01000000B

        OUT   (PMDAC),A
        DEFB  0EDH,70H
        JP    M,RECAD1
        AND   10111111B
RECAD1:
        OR    00100000B

        OUT   (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD2
        AND  11011111B
RECAD2:
        OR   00010000B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD3
        AND  11101111B
RECAD3:
        OR   00001000B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD4
        AND  11110111B
RECAD4:
        OR   00000100B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD5
        AND  11111011B
RECAD5:
        OR   00000010B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD6
        AND  11111101B
RECAD6:
        OR   00000001B

        OUT  (PMDAC),A
        DEFB 0EDH,70H
        JP   M,RECAD7
        AND  11111110B
RECAD7:
        OR   00000000B
```

```
        LD    (HL),A
        LD    A,00001100B
        OUT   (PMCNTL),A

        POP   BC
        INC   HL
        DEC   BC
        LD    A,C
        OR    B                    ; end of data ?
        JR    NZ,REC1              ; next data

        LD    A,00000011B
        OUT   (PMCNTL),A   ;D/A MODE
        EI
        RET

        END
```

list 1.7 (PCMPLAY.MAC)

```
PMDAC   EQU 0A4H
PMCNT   EQU 0A4H
PMCNTL  EQU 0A5H
PMSTAT  EQU 0A5H
SYSTML  EQU 0E6H          ; system timier port

PLAY:
        LD    A,00000011B
        OUT   (PMCNTL),A          ; D/A MODE
        DI
        XOR   A
        OUT   (SYSTML),A          ; reset timer

PLAY1:
        IN    A,(SYSTML)
        CP    E
        JR    C,PLAY1             ; wait
        XOR   A
        OUT   (SYSTML),A          ; reset timer

        LD    A,(HL)
        OUT   (PMDAC),A           ; play 1 byte
        INC   HL
        DEC   BC
        LD    A,C
        OR    B                   ; end of data ?
        JR    NZ,PLAY1            ; next data
        EI
        RET

        END
```

Table 1.4: I/O Ports for PCM

street address	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0A5H Write	0	0	0	SMPL	SEL	FILT	MUTE	ADDA
0A5H Read	COMP	0	0	SMPL	SEL	FILT	MUTE	BUFF
0A4H Write	DA7	DA6	DA5	DA4	DA3	DA2	DA1	DA0
0A4H Read	0	0	0	0	0	0	CT1	CT0

- ADDA (BUFF): Buffer mode
  Specifies the output of the D/A converter.
  Set this to 0 (double buffer) for D/A, and 1 (single buffer) for A/D. Note that the output is set to double buffer when reset.

- MUTE: Muting control
  Turns system-wide sound output on or off.

  0: Audio output off (at reset)
  1: Audio output on

- FILT: Selection of sample-and-hold circuit input signal
  Select whether the signal input to the sample-and-hold circuit during A/D conversion is the filter output signal or the reference signal. 0 is the reference signal, 1 is the filter output signal. Reset is 0.

- SEL: Filter input signal selection
  Select whether the signal input to the low pass filter is the output signal of the D/A converter or the output signal of the microphone amplifier. 0 is the D/A converter output signal, 1 is the microphone amplifier output signal sound.

- SMPL: Sample and hold signal Select
  whether to sample or hold the input signal.

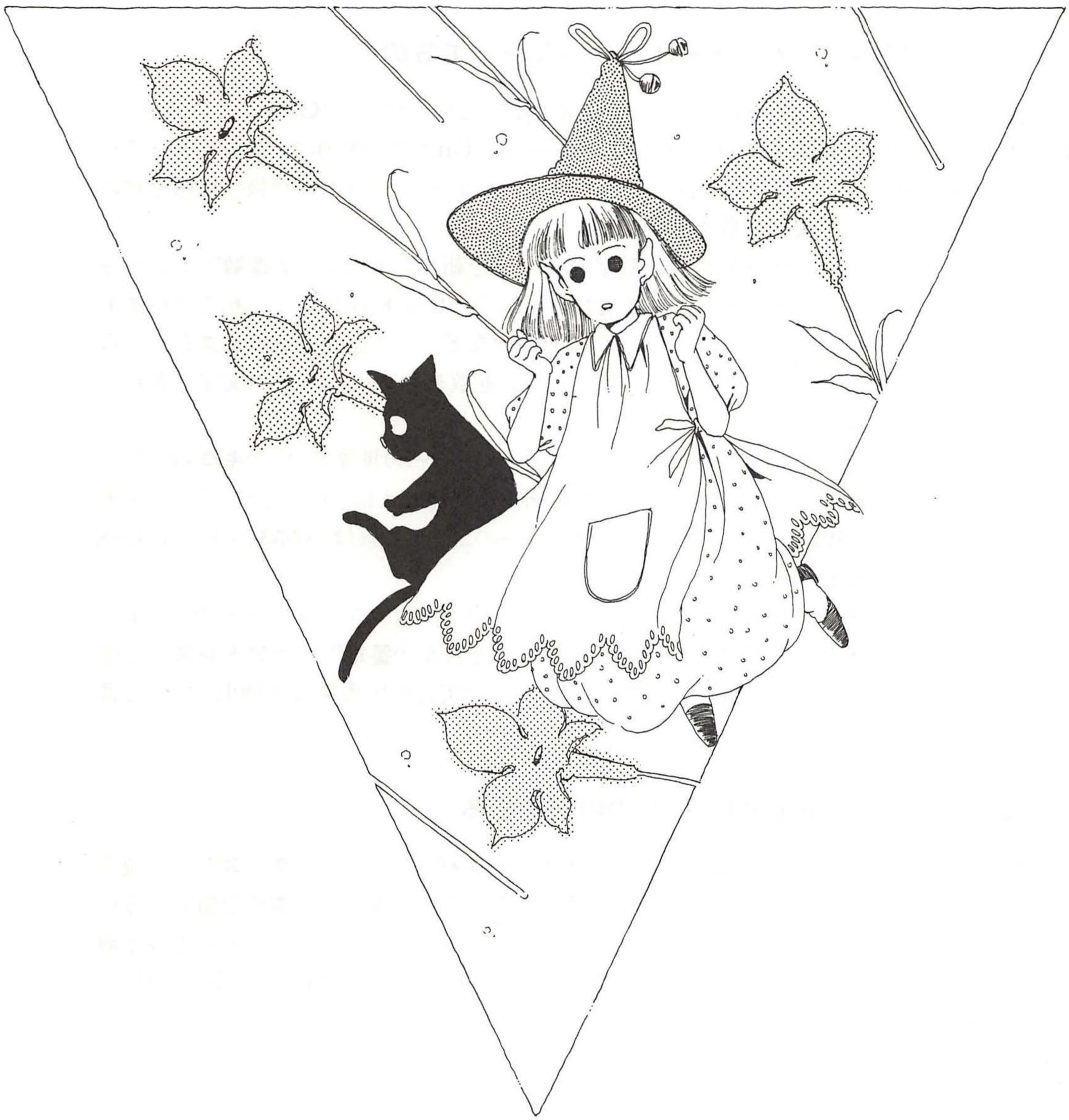
  0: Sample (at reset)
  1:Hold

- COMP: Comparator output signal. Compares the sample and hold output signal with the D/A converter output signal.
  0: D/A output > Sample hold output
  1: D/A output < sample hold output

- DA7~DA0: D/A output data
  When playing PCM data, you can play PCM sound by outputting the prepared data here. The data format is absolute binary, with 127 corresponding to level 0.

- CT1, CTO: Counterdata
  It counts up every 63.5 microseconds. During D/A conversion, it is synchronized with the count up and the data written to address 0A4H is repeatedly output. Also, writing data to 0A4H clears the counter.

第2章 SLOT



This chapter is a re-edited version of the articles "MSX2+ Technical Exploration" from the February 1989 and March 1989 issues of MSX Magazine, and "Technical Analysis" from the November 1990 issue.

2.1 What is a slot?

The hole for inserting the cartridge into the MSX is called the "cartridge slot." But the slot also serves the function of managing the MSX's memory. In this chapter, we will explain the most important and complex of the slots.

2.1.1 How is the CPU and memory connected?

The most important components of a computer are the CPU and memory. CPU is an abbreviation for Central Processing Unit, and is the device that manages the entire computer and performs calculations. Memory, on the other hand, is a pad-like device that stores the information that the CPU handles.

As you may know, the information handled by computers is represented as "binary numbers" combining the numbers 0 and 1. One digit of this binary number is called a "bit" and eight digits are called a "byte." Also, since expressing binary numbers as is in program lists etc. would result in too many digits, "hexadecimal numbers" are often used, which represent 4-bit binary numbers using the letters 0-9 and A-F.

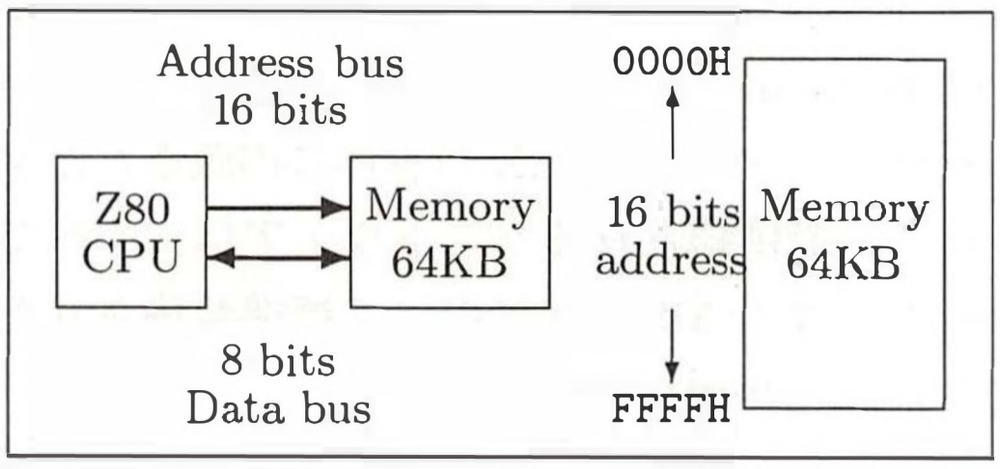
Make no mistake, in the computer world, the unit "kilo" (K) means 1024 times, not 1000 times. For example, 64 kilobytes of memory means 64 x 1024 = 65536 bytes of memory, which is also 65536 x 8 = 524288 bits.

To manage these memories, many microcomputers assign numbers to each byte of memory. These numbers are called "locations" or "addresses." In machine language programs, you often see things like "The execution start address is 8000H."

2.1.2 Exploring the inside of the 8-bit CPU Z80

As shown in Figure 2.1, the CPU and memory are connected by an "address bus" and a "data bus." The address bus is an electrical wire that sends signals from the CPU to memory specifying the memory address that the CPU wants to read or write. The data bus is an electrical wire that communicates the contents of memory. Note that the former is a one-way bus from the CPU to memory, while the latter is bidirectional.

figure 2.1: Memory for Z80 CPUs



The Z80 CPU used in the MSX can basically connect to 64KB of memory. A 16-bit address bus signal can specify one byte of memory between 0000H and FFFFH. The data bus for transferring data is 8-bit.

The "Z80", the CPU of MSX before turbo R, has an 8-bit (physically 8 wires) data bus and a 16-bit address bus. This means that it can read and write 64 kilobytes of memory one byte at a time. Such a CPU is called an "8-bit CPU", and a computer with an 8-bit CPU is called an "8-bit computer". That's why MSX is an 8-bit computer.

Now, to explain the address bus in detail, it is specified by a 16-bit address bus. The memory address that can be set is a binary number.

0000000000000000B

From (B is the symbol for binary numbers)

1111111111111111B

In decimal, this ranges from 0 to 65535, and in hexadecimal, it ranges from addresses 0000H to FFFFH (H is the symbol for hexadecimal). Each address contains 8 bits (=1 byte), which in decimal represents a value from 0 to 255. Furthermore, these memories expressed in bytes are converted to kilobytes, which equals 64. Therefore, an 8-bit computer can be connected to 64 kilobytes of memory.

I mentioned before that MSX is an 8-bit computer, but recently 16-bit computers (computers with 16-bit CPUs) have also become popular. In this case, the data bus is 16 bits, so twice as much information can be read and written at one time compared to 8-bit computers. There are also many address buses, which has the advantage of being able to connect more memory. However, due to the complex wiring, it is relatively expensive. Also, many large computers have 32-bit or 64-bit data buses and address buses.

Although the MSX turbo R was equipped with a 16-bit CPU, the R800, the data bus remained 8-bit so that conventional cartridges could be connected.

2.1.3 There are various types of memory depending on the function

There are many types of memory. First, they are classified according to the type of component: "ROM" and "RAM". ROM (Read Only Memory) is memory whose contents cannot be rewritten, but remain even when the power is turned off. All software built into the MSX (BASIC, etc.) and software supplied on cartridges is written into this ROM. Kanji ROM, which became standard equipment with the MSX2+, is a dedicated ROM with the shapes of kanji characters written into it.

RAM (Random Access Memory) is a type of memory whose contents can be freely rewritten, but whose contents are lost when the power is turned off. It is used to temporarily store the results of calculations in a program, or to load and execute a program from a floppy disk. For example, if you type in a short program from M Magazine to play a game, it will be stored in this RAM.

"SRAM" is a type of RAM that consumes little power and is used in battery-powered laptops and portable word processors, as well as battery-backed game cartridges for MSX and Famicom.

Memory can also be classified according to how it is used. As shown in Figure 2.1, memory that is directly connected to the CPU is called "primary memory" or "main memory." In other words, all ROM other than the MSX's kanji ROM and the 64 kilobytes of main RAM are the MSX's main memory.

MSX also has another memory called "Video RAM (VRAM)". Video RAM is RAM used to store the graphics and text to be displayed on the TV screen. Depending on the computer model, the video RAM may be directly connected to the CPU, but in MSX the CPU and video RAM are connected via a dedicated component called the VDP (short for Video Display Processor).

The basics explained so far are the most basic knowledge not only about MSX but also about computers. I think the introductory book on BASIC that comes with MSX will explain it in detail, so please refer to it.

2.1.4 What are MSX slots like?

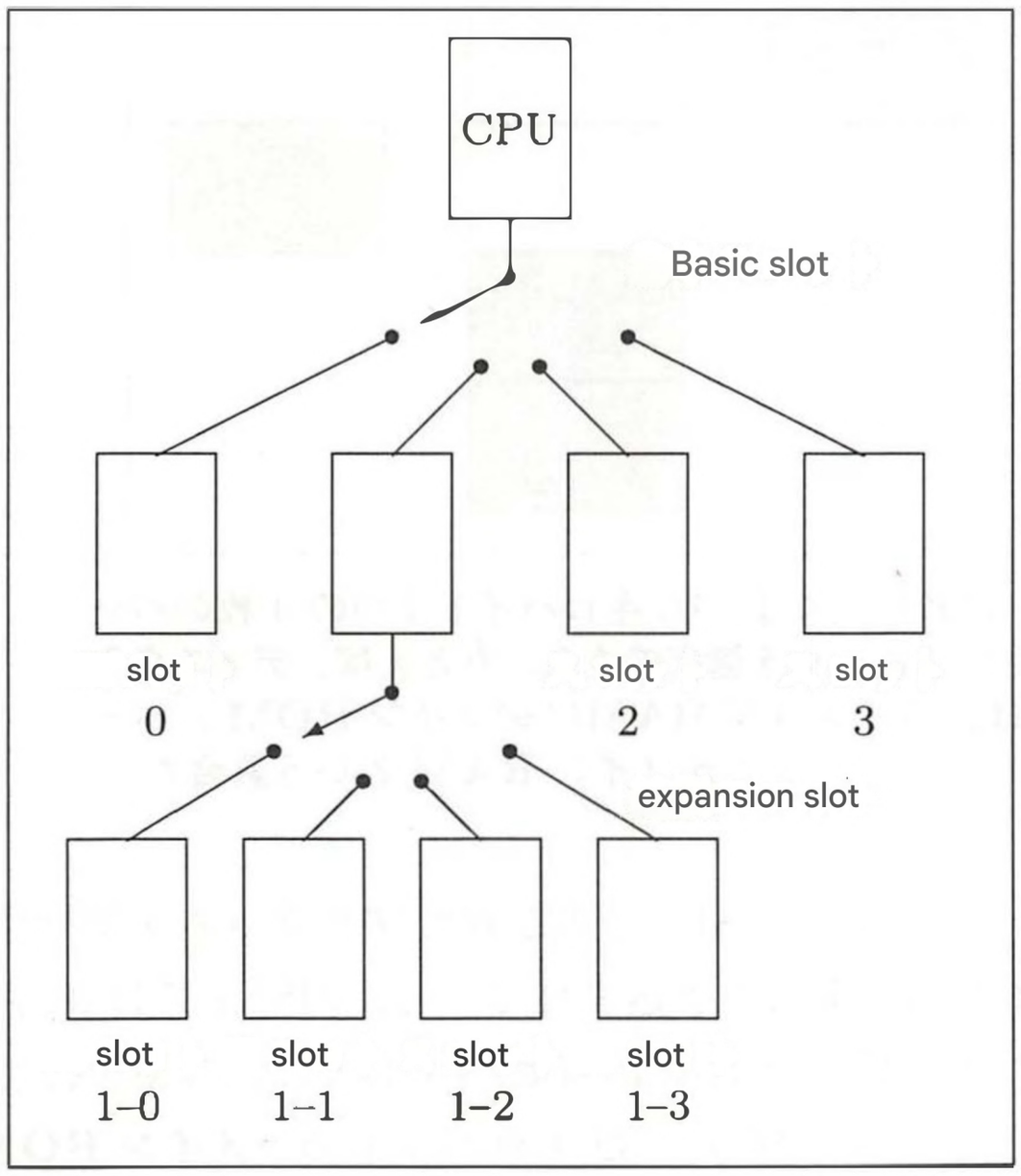
As mentioned at the beginning, the main memory that can be connected to an 8-bit CPU is 64KB. However, this was only the case for early 8-bit computers. Nowadays, there are various methods that allow you to connect more than 64KB of memory.

In the case of MSX, this is called "slot switching." As shown in Figure 2.2, four sets of 64 KB memory are prepared and can be switched between, making it possible to handle a maximum of 256 KB of memory.

These memories are called "basic slots"

They are also assigned to cartridge slots provided on the machine.

figure 2.2: MSX slot configuration (part 1)



In MSX, to handle memory exceeding 64 kilobytes, a method called “slot switching” is used. By switching between four 64-kilobyte memory blocks, up to 256 kilobytes of memory can be handled. These four memory blocks are called “primary slots,” and each of them can be expanded into four “secondary slots,” referred to as “expanded slots.”

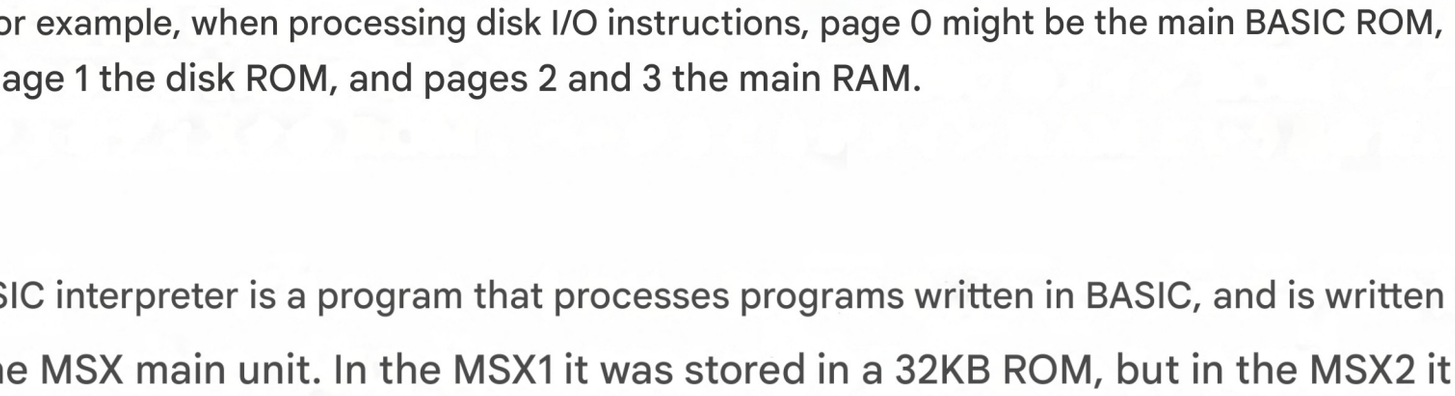
Additionally, you can use four sets of "expansion slots" instead of one basic slot, which gives you a total of 16 sets of 64KB memory, meaning you can connect a maximum of 1MB (1024KB) of memory. However, you cannot expand the expansion slots further.

Now, it's fine to be able to handle memory over 64KB by switching slots, but it's inconvenient to have the entire memory switched at the same time. So in MSX, memory is divided into units called "pages" and handled accordingly.

The 16KB from memory addresses 0000H to 3FFFH is page 0, and the 16KB from memory addresses 4000H to 8000H is page 1. Similarly, addresses 8000H to BFFFH is page 2, and addresses C000H to FFFFH is page 3. Each page has 16KB in one block, and a different slot can be selected for each page.

For example, when a BASIC disk I/O instruction is being processed, page 0 becomes the BASIC interpreter's main ROM, page 1 becomes the disk interface ROM, and pages 2 and 3 become main RAM (see Figure 2.3).

figure 2.3: MSX slot configuration (part 2)



The 64KB memory address is divided into four 16KB pages, with a selectable slot for each page. For example, when processing disk I/O instructions, page 0 might be the main BASIC ROM, page 1 the disk ROM, and pages 2 and 3 the main RAM.

The BASIC interpreter is a program that processes programs written in BASIC, and is written into the ROM of the MSX main unit. In the MSX1 it was stored in a 32KB ROM, but in the MSX2 it is 48KB. Therefore, in the MSX2 the ROM is divided into two, with the parts common to the MSX1 stored in the 32KB "main ROM" and the functions extended by the MSX2 stored in the 16KB "sub ROM".

Also, the MSX with built-in disk drive and the interface cartridge for external drives have a 16KB ROM built in. This is where the program (DISK-BASIC) for processing disk I/O in BASIC is written. Therefore, during disk I/O, the state shown in Figure 2.3 appears.

2.1.5 The secret to MSX's expandability was its slots

The MSX slots are used not only to add memory, but also to expand the functions of the MSX. The slots are also used to connect game cartridges and modem cartridges.

As I wrote above, when you connect a disk interface cartridge to the MSX, the 16KB ROM in the cartridge is connected to a slot. Then, when disk I/O is performed, the memory is automatically switched to the disk interface ROM slot. Therefore, whether the disk interface itself is built into the main unit or connected as a cartridge, there is no problem with the operation of the program.

Also, when a disk interface is connected, the "CALL FORMAT" command becomes available, and when a communications cartridge is connected, the "CALL TELCOM" extended BASIC command becomes available. These extended commands are processed by the ROM in the cartridge. With most computers other than MSX, when using peripheral devices, it is necessary to load the programs that control them from the disk. However, with MSX, simply connecting an interface cartridge automatically extends the BASIC commands.

In addition, "Japanese MSX-DOS2", an improved version of MSX-DOS, "HALNOTE", a popular integrated software, and "MSXView", a GUI (Graphical User Interface) for turbo R, were also among the software supplied on cartridges. In this way, the ability to easily expand functions by plugging ROM cartridges into the slot is an advantage of MSX that other personal computers do not have.

Although slots are a useful feature, developing a program that uses them effectively is quite difficult. Even for a programmer who is fluent in writing machine language programs for the Z80 CPU, it may take more than a year to truly understand the concept of slots.

2.1.6 How the MSX2+ slot has changed

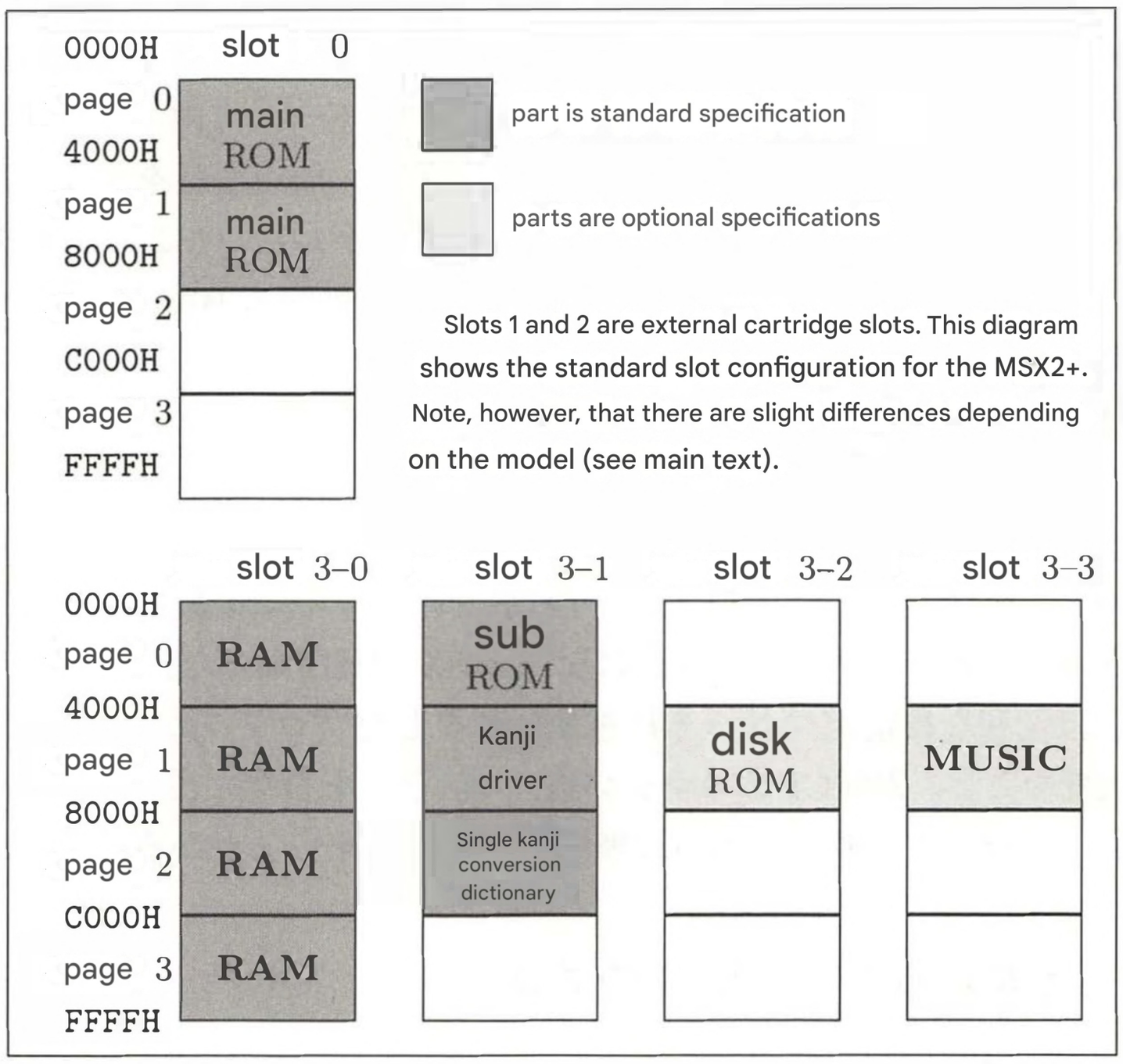
As I have written so far, the slots gave the MSX machine a lot of expandability and character. However, these slots were also a weakness of the MSX. That is, "Because the slot configuration differed depending on the model in the past, software compatibility problems were likely to occur."

For example, certain software may not work on models where slots 1 and 3 are assigned as cartridge slots. Also, if RAM is placed in the expansion slot (slot 3), the sub-ROM functions may not be available from DOS. These problems should be avoided by carefully creating a program and checking that it works on all MSX machines. However, making a program compatible with machines with all slot configurations may result in longer programs and slower execution speeds. Slots are not a straightforward matter.

With the arrival of the MSX2+, a certain degree of standard for slot configuration was finally established. Figures 2.4 and 2.5 show examples of MSX2+ slot configurations. Depending on the amount of software built into the console, there are cases where only slot 3 is expanded, and cases where both slot 0 and slot 3 are expanded.

Figure 2.4 shows an example where only slot 3 is expanded. The main BASIC ROM is placed in basic slot 0, and slots 1 and 2 are used as external cartridge slots.

Figure 2.4: Example of MSX2+ slot configuration (when expanding slot 3 only)



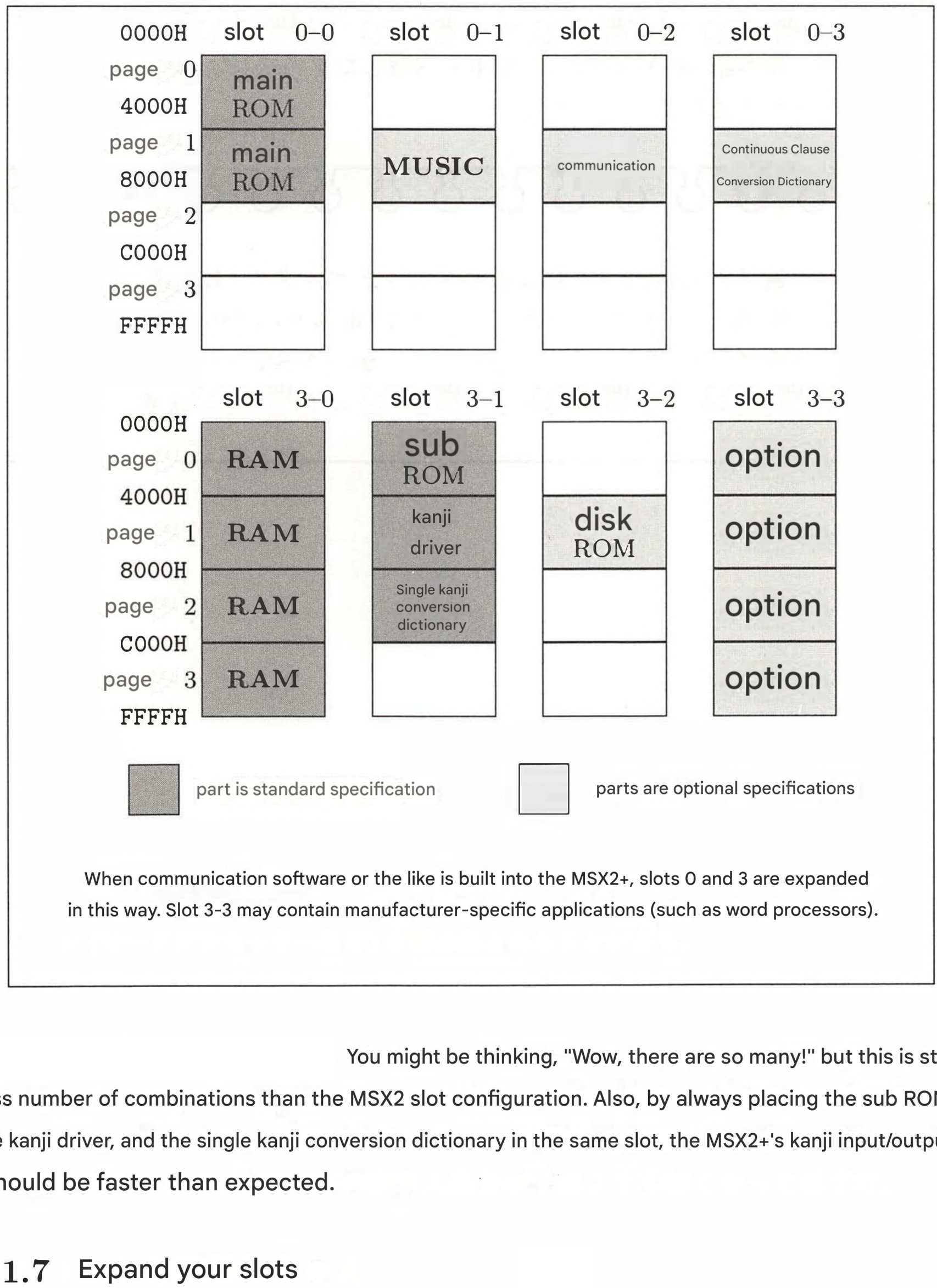
Then, place 64KB of RAM (main RAM) in one of the expansion slots of slot 3, so that pages 0 to 3 always have RAM in the same slot. Similarly, place a total of 48KB of ROM, including the sub-ROM, kanji driver, and single-kanji conversion dictionary, in one of the expansion slots of slot 3. In the diagram, the RAM is placed in slot 3-0 (the 0th expansion slot of basic slot 3), and the ROM in slot 3-1, but this will differ depending on the model.

In contrast, Figure 2.5 shows the case where both slot 0 and slot 3 are expanded. The main ROM is placed in expansion slot 0 of the base slot 0. The configuration of slot 3 is almost the same as in Figure 2.4. Please note that if a disk interface is built in, the ROM will always be placed in the expansion slot 3, not the expansion slot 0.

The light gray parts in Figure 2.4 and Figure 2.5. In other words, the disk interface, MSX-MUSIC (FM sound source), communication, and multi-phrase conversion dictionary ROMs are optional for the MSX2+. Therefore, they are sometimes not built into the main unit, but are connected as external cartridges.

Now, with the MSX2+ having the same amount of built-in software as shown in Figure 2.5, there are theoretically 36 possible slot configurations.

Figure 2.5: Example of MSX2+ slot configuration (when expanding slots 0 and 3)



When communication software or the like is built into the MSX2+, slots 0 and 3 are expanded in this way. Slot 3-3 may contain manufacturer-specific applications (such as word processors).

You might be thinking, "Wow, there are so many!" but this is still less number of combinations than the MSX2 slot configuration. Also, by always placing the sub ROM, the kanji driver, and the single kanji conversion dictionary in the same slot, the MSX2+'s kanji input/output should be faster than expected.

2.1.7 Expand your slots

MSX machines have one or two external cartridge slots (where game cartridges are usually inserted) which are all basic slots. By connecting a "slot expander", it is possible to expand the number of slots to four. For example, if you connect a slot expander to both of the main body's two slots, you will have a total of eight slots.

However, you should be aware that some cartridges do not work in the expansion slot. Japanese MSX-DOS2 (type with built-in RAM) is one of them. Check the software you want to use before expanding.

The product name of the slot expander is "MSX Expansion Slot Box EX-4". It is currently on sale from Nippon Electronics (Tel: 03-3486-4181) for 29,800 yen (excluding tax). In addition to this, several MSX manufacturers have released slot expanders, but most are now unavailable.

2.2 Try switching slots

When talking about MSX, the concept of slots is unavoidable. Here, we will focus on how to control slots with software and the changes in specifications for MSX2+.

2.2.1 To switch slots

I've explained the meaning of slots in MSX, but I haven't explained how to switch between slots. So from here on, I'll show you how to do it. Of course, it's not something as unsophisticated as "a human switching it with a switch," but rather it can be switched by a program.

First, the MSX CPU, the Z80, has something called an "I/O port." This is like a telephone line that allows the CPU to communicate with the outside world (i.e. peripheral devices such as VDP and FM sound sources). The Z80 has a total of 256 I/O ports, and they are differentiated by assigning addresses from 0 to 255 (00H to FFH in hexadecimal).

To handle these ports in BASIC, you read a byte from an I/O port with the "INP" function and write a byte with the "OUT" command, and in machine language the "IN" and "OUT" commands perform the same function.

Now, the basic slot is switched depending on the value written to the A8H address of this I/O port. Conversely, by reading the value of this address, you can find out the current status of the slot. Bits 7 and 6 correspond to page 3, 5 and 4 to page 2, 3 and 2 to page 1, 1 and 0 to page 0, and so on, so writing the value 11110000B (B stands for binary) will switch page 3 and 2 to slot 3, and pages 1 and 0 to slot 0. Also, to switch expansion slots you use memory address FFFFH, but this is complicated so we won't go into it here.

However, directly switching slots using a program is not only tedious, but it is also prone to compatibility issues, such as the program not working on some models.

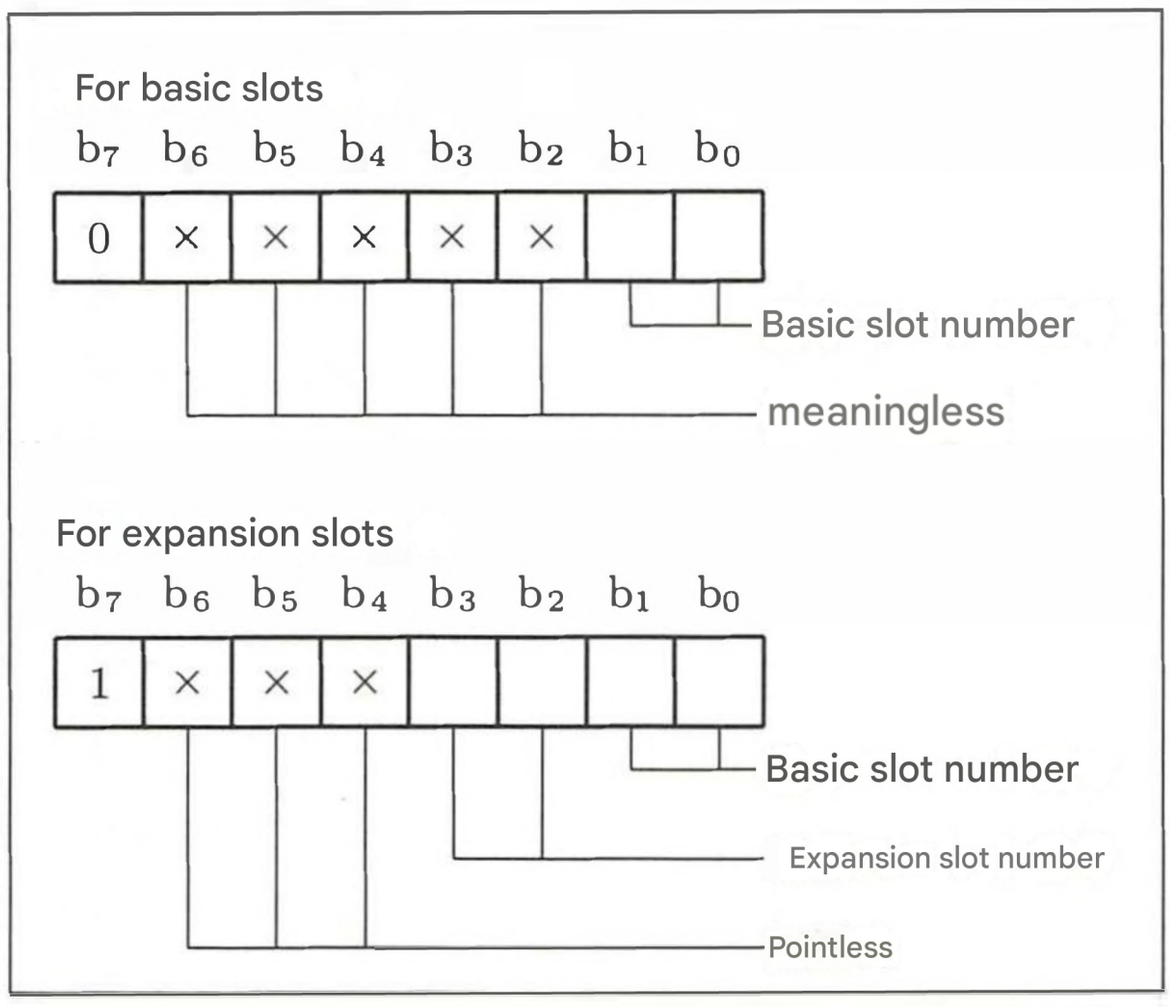
So, in reality, we switch slots using the "BIOS". BIOS stands for "Basic Input Output System". I'll explain it in detail later, but it's a set of machine language subroutines to control hardware. It has many functions other than switching slots, so let's check it out.

2.2.2 How to specify slot number

When using the BIOS to switch slots, you can specify the base slot number and the expansion slot number separately. However, this requires two registers (temporary data storage locations in the CPU), which is uneconomical. Therefore, a method is used to specify the base slot and expansion slot together by making good use of each bit of 8 bits (1 byte), as shown in Figure 2.6.

For example, to specify base slot 0, use the value 00000000B (00H in hexadecimal), and to specify base slot 3 and expansion slot 1, use the value 10000111B (87H).

figure 2.6: How to specify slot number



The slot number is represented by an 8-bit (= 1 byte) value as shown in the diagram. For example, to specify base slot number 0, use the value 00000000B, and to specify expansion slot 1 of base slot 3, use the value 10000111B. Note that the contents of each bit marked with an x in the diagram are ignored.

2.2.3 BIOS functions for handling slots

First, let's learn the symbols used to represent BIOS functions. E is the register that should be set before calling the BIOS. R is the register to which the BIOS returns the value. M is the register to which the BIOS writes a meaningless value, i.e. the original contents are destroyed. IYH is the upper byte of the IY register, and the contents of the lower byte are ignored. The address written at the beginning is the entry address for calling the BIOS.

RDSLT 000CH address

function Reads the contents of the address specified by the HL register in the slot specified by the A register.

E A slot number

HL address

R A read value

M AF, BC, DE

Note Interrupts are disabled.

WRSLT 0014H address

function Writes the contents of the E register to the address specified by the HL register in the slot specified by the A register.

E A slot number

HL address

EContents to be written

R none

M AF、BC、D

Note Interrupts are disabled.

CALSLT 001CH address

function Calls a subroutine in another slot.

E IX Address to call

IYH slot number

R Depends on who you call

M IX, IY, back register

Note The current state of the slot is saved on the stack and the desired subroutine is called. The contents of the AF, BC, DE, and HL registers are passed to the subroutine as is, and when the subroutine executes the RET instruction, it returns to the original program. In this case, the values of the AF, BC, DE, and HL registers are passed from the subroutine. The number of bytes of stack used depends on the slot configuration.

ENASLT 0024H address

function Switch slots.

E A slot number

H page (upper 2 bits)

R none

M AF、BC、DE、HL

Note For example, to switch between two pages, set a value between 80H and BFH in the H register. Interrupts are disabled.

CALLF 0030H address

function Calls a subroutine in another slot.

E As shown in the following program, write the slot number and address into the program after the "RST 30H" command.

```
RST   30H
DB    slot number
DW    address
```

R Depends on who you call

M IX, IY, back register

Note Except for the method of specifying the slot and address, it is the same as CALSLT. It is used for special purposes (hooks).

EXTROM 015CH address

function Call sub ROM.

E IX Address to call

R Depends on who you call

M IX, IY, back register

Note It works the same as CALSLT, except that the sub ROM slot is automatically selected.

The BIOS introduced above has some limitations. None of them can be used for page 3. They can only be used for page 0 when calling the main ROM from DOS. They can be used without problems for pages 2 and 3. Even though they "cannot be used", they may be usable depending on the slot configuration, so be careful not to create a program that only works on your MSX. In particular, if you use CALSLT to call the sub ROM from DOS, it may or may not work depending on the slot configuration and the type of disk interface.

2.2.4 How to know the slot configuration

As explained before, the slot configuration of the MSX varies depending on the model. There are also optional specifications such as disk interfaces, so it is no exaggeration to say that there are as many slot configurations as there are machines. Here we will introduce how to check the slot configuration of your machine and whether or not it has optional equipment.

Memory addresses F380H to FFFEH are called the "system work area", and important information for the BIOS and other devices is stored here. When a disk interface is connected, a "disk work area" is prepared at an address slightly smaller than the system work area. Information related to slots is stored in the system work area and disk work area, as shown in Table 2.1.

Although it is important to know which slot the main RAM is in, the “RAMAD0” in Table 2.1 is in the disk work area, so there is also the problem that you cannot know this information without the disk.

List 2.1 is a program that displays the slot configuration found from these system work areas in an easy-to-understand way. The configuration varies depending on the model, so try it out on your own MSX.

Other than those listed in Table 2.1, there are other system work areas that are useful for programs, but for details, please refer to the "MSX2 Technical Handbook". Also, unless specifically instructed, application programs should not rewrite these system work areas. Some programs use the system work area as a last resort when memory is insufficient, but be careful because this can cause compatibility issues.

Table 2.1: System work areas for slots

name	address	meaning
RAMAD0	F341H	RAM slot number for page 0 (1)
RAMAD1	F342H	RAM slot number for page 1 (1)
RAMAD2	F343H	Page 2 RAM slot number (1)
RAMAD3	F344H	Page 3 RAM Slot Number (1)
MASTER	F348H	Slot number of drive A's interface (1)
EXBRSA	FAF8H	Sub ROM slot number (0 for MSX1)
EXPTBL	FCC1H	Main ROM slot number
	FCC2H	Whether slot 1 is expanded (2)
	FCC3H	Whether slot 2 is expanded (2)
	FCC4H	Whether slot 3 is expanded (2)

(1) Valid only if disk is present.
(2) If it is extended then 80H, if not then .

2.2.5 Explore the system work area

Among the MSX2 games, there are some that display the title screen of SCREEN 12 when run on MSX2+. There are also programs that display a helpful error message if you try to communicate without a modem cartridge. In order to create such software, we will introduce a method for programs to check the type and configuration of the hardware.

First, to check whether a disk is present, the contents of address FFA7H are read. If it is C9H then there is no disk, if it is any other value there is a disk.

To check the "type of MSX", read address 2DH of the main ROM. 0 means MSX1, 1 means MSX2, 2 means MSX2+, and 3 means turbo R.

Generally, the contents of addresses 2BH and 2CH are 0, but in "MSX made for export to overseas", they contain numbers that indicate the type of keyboard or currency symbol. Since you only need to be concerned with these when making software for export, we will omit the list of numbers.

This is a bit off topic, but MSX computers have been exported in large numbers to Europe, the Soviet Union, and Kuwait in the Middle East. In neighboring South Korea, many of them are installed in schools and used in classes. It's a truly international machine.

Now, to find out the "video RAM capacity", read the FAFCH address. If bit 2 and bit 1 are 00, it's 16KB, if 01, it's 64KB, if 10, it's 128KB. Ignore the other bits, as they seem to be used for other purposes.

As shown in the list on the next page, we use "AND 6" to take the values of bits 2 and 1 and divide them by 2.

As an added bonus, this list also checks whether or not there is an "extended BIOS." This is a function for controlling optional hardware such as a communications modem, FM sound source, or kanji dictionary. If the content of bit 0 at address FB20H is 1 and the content of address FFCAH is not C9H, then there is some kind of extended BIOS function. Finding out what it is would require a complex machine language program, so we'll pass on that this time. Also, although this is not written in the specifications, the content of address FFCBH seems to be the slot number of a program that has an extended BIOS function.

By the way, we don't know what is written in bits that don't have a defined meaning, such as bits 6 to 4 of the value that represents the slot number. So we use "AND" to ignore those bits.

As I've mentioned many times before, slot configurations can vary between machines, even from the same manufacturer. So after testing it on your own machine, try it on your friends' machines too. It's interesting to test it on many machines and compile a table of the results.

list **2.1 (WHO_AM_I.BAS)**

```
100 ' Analizing slot structure of MSX
110 ' by nao-i on 9. Jan. 1989
120 CLEAR : DEFINT A-Z : CLS
130 VE = PEEK(&H2D) : ' Version No. of BASIC
140 IF VE=0 THEN PRINT "I am MSX1"
150 IF VE=1 THEN PRINT "I am MSX2"
160 IF VE=2 THEN PRINT "I am MSX2+"
165 IF VE=3 THEN PRINT "I am MSX turbo R"
170 IF VE>3 THEN PRINT "Who am I ?"
180 VR = (PEEK(&HFAFC) AND 6) ¥ 2 : ' size of VRAM
190 IF VR=0 THEN PRINT "VRAM 16KB"
200 IF VR=1 THEN PRINT "VRAM 64KB"
210 IF VR>1 THEN PRINT "VRAM 128KB"
220 MR = PEEK(&HFC49) : ' size of main RAM
230 IF MR >= &HE0 THEN PRINT "RAM 8KB"
240 IF MR < &HE0 AND MR >= &HC0 THEN PRINT "RAM 16KB"
250 IF MR < &HC0 THEN PRINT "RAM >= 32KB"
260 FOR SS = 0 TO 3 : 'EXPTBL
270   PRINT USING "Slot # is "; SS;
280   FF = PEEK(&HFCC1 + SS) AND 128
290   IF FF THEN PRINT "expanded slot" ELSE PRINT "primary slot"
300 NEXT SS
310 PRINT
320 SS = PEEK(&HFCC1) : PRINT "Main ROM is in "; : GOSUB 530
330 SS = PEEK(&HFAF8) : PRINT "Sub  ROM is in "; : GOSUB 530
340 IF PEEK(&HFFA7) <> &HC9 THEN 360
350 PRINT "I have disk(s)." : GOTO 450
360 PRINT "I have no disk."
370 SS = PEEK(&HF348) : PRINT "FDC ROM is in "; : GOSUB 530
380 SS = PEEK(&HF341) : PRINT "P0  RAM is in "; : GOSUB 530
390 SS = PEEK(&HF342) : PRINT "P1  RAM is in "; : GOSUB 530
400 SS = PEEK(&HF343) : PRINT "P2  RAM is in "; : GOSUB 530
410 SS = PEEK(&HF344) : PRINT "P3  RAM is in "; : GOSUB 530
420 PRINT "Bottom address of disk work area is ";
430 PRINT RIGHT$("00"+HEX$(PEEK(&HFC4B)),2);
440 PRINT RIGHT$("00"+HEX$(PEEK(&HFC4A)),2)
450 ' detectiong extended BIOS
460 IF (PEEK(&HFB20) AND 1) = 0 THEN GOTO 520
470 IF PEEK(&HFFCB) = &HC9 THEN GOTO 520
480 PRINT : PRINT "I have extended BIOS."
490 SS = PEEK(&HFFCB)
500 PRINT "ROM of the extended BIOS may be in "
510 GOSUB 530
520 END
530 ' displaying slot number
540 PRINT USING "primary slot #"; SS AND 3;
550 IF (SS AND 128) = 0 THEN 570
560 PRINT USING " extended slot #"; (SS AND 12) ¥ 4;
570 PRINT : RETURN
```

2.2.6 MSX2+ Hardware Specifications

The MSX2+ included minor hardware improvements. These are summarized in Table 2.2. However, these are I/O ports for which new specifications have been defined or added.

Table 2.2: MSX2+ I/O ports

I/O address	Purpose
7CH	Built-in FM sound source
7DH	Built-in FM sound source
DAH	Level 2 Kanji ROM
DBH	Level 2 Kanji ROM
F4H	Controlling initialization
F5H	device enable

This is a new specification definition or addition. However, in actual programs, it is better to use the BIOS rather than using the I/O ports directly.

Addresses 7CH and 7DH are I/O ports for controlling the FM tone generator built into the unit. In contrast, the FM tone generator supplied by cartridge (if released in the future) will use the same I/O port as Panasonic's "FM-PAC".

To check if the unit has a built-in FM sound source, read addresses 4018H to 401FH for each slot. If the contents match the string "APRLOPLL", that slot contains a ROM program that controls the FM sound source, and the unit has a built-in FM sound source.

On the other hand, in the case of FM sound cartridges, the contents from address 4018H seem to be four letters indicating the product type and the letter "OPLL", such as "PAC2OPLL".

In "MSX-Write" and some modem cartridges, when you select "BASIC" in the first menu, the MSX title screen appears as if it was reset. This is because, for the sake of software preparation, the software jumps to address 0 of the main ROM and performs the same process as a reset.

Previously, there was no way to reliably distinguish between a jump to address 0 and a real reset, which could lead to software malfunctions. However, starting with the MSX2+, hardware was added to the I/O port at address F4H to check the reset state. However, in reality, the BIOS is added to the MSX2+ main ROM as follows:

```
CALL 17AH
OR   80H
CALL 17DH
JP   0
```

The ROM cartridge program called during initialization is

```
CALL 17AH
```

Then, if bit 7 of the A register is 0, it's a real reset. If it's 1, it's a jump to 0, and you know that you've been called.

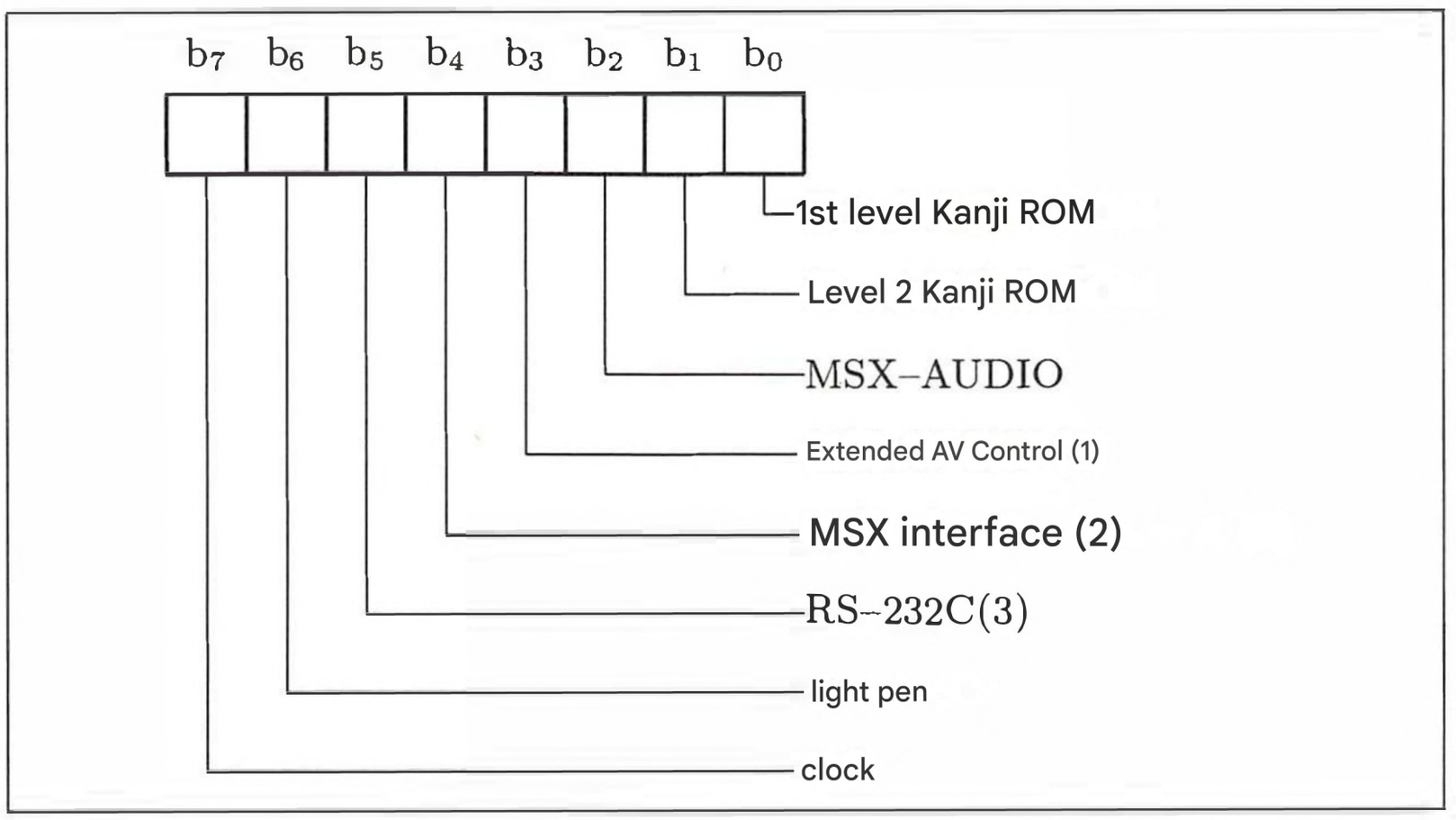
2.2.7 Device enable to prevent collisions

If you connect a kanji ROM cartridge to an MSX that has a built-in kanji ROM, not only will the kanji not be displayed correctly, but there is also the risk of hardware conflicts and failure. A useful feature to prevent this is "device enable," which is controlled by the I/O port address F5H.

The hardware shown in Figure 2.7 is disconnected from the bus at reset. Then, by writing a 1-byte (8-bit) value to the I/O port address F5H, the internal hardware corresponding to the bit that is set to 1 is connected to the bus. These processes are performed automatically after a reset or jump 0 (software transfers program execution to address 0 of the main ROM).

In MSX2, there was no provision for whether or not to disconnect already connected hardware when writing 0 to the I/O port address F5H. This meant that confusion could occur when commands such as "MSX-Write" tried to jump to address 0 of the ROM to reinitialize the BIOS.

figure 2.7: Device Enable



The I/O port address F5H is used to select whether to enable or disable the built-in hardware (the hardware corresponding to the bit with a 1 written to it is enabled).

(1) Superimpose function, etc., controlled by the F7H address of the I/O port.

(2) It is in the specifications but has not been put into practical use.

(3) This has nothing to do with modems.

Starting with MSX2+, however, the internal hardware is now unified so that writing 0 disconnects it from the bus. This means that by creating basic software for MSX2+ using I/O port addresses F4H and F5H, compatibility and reliability will be improved even more than before.

2.3 MSX Trubo R slot configuration

Here, I will explain the newly announced slot configuration of MSX turbo R. What is particularly noteworthy is that the slot configuration has finally been unified. This is quite significant.

2.3.1 Finally, the slot configuration has been unified.

Figure 2.8 shows the slot configuration of turbo R. The slot configuration was unified to accommodate faster CPU speeds and to make it easier to develop and debug application programs.

In this diagram, it looks like there is 64KB RAM in slot 3-0, but in reality, there is 256KB of main RAM connected through the memory mapper. The part exceeding 64KB is usually used for Japanese MSX-DOS2 work area, RAM disk, "DRAM mode" explained in another chapter, etc.

However, application programs can use the extended BIOS to switch mappers and use this RAM.

The DOS system ROM is stored in page 1 of slot 3-2. However, it also contains a 16KB DOS1 (MSX-DOS) ROM and a 48KB DOS2 ROM, which are automatically switched as needed.

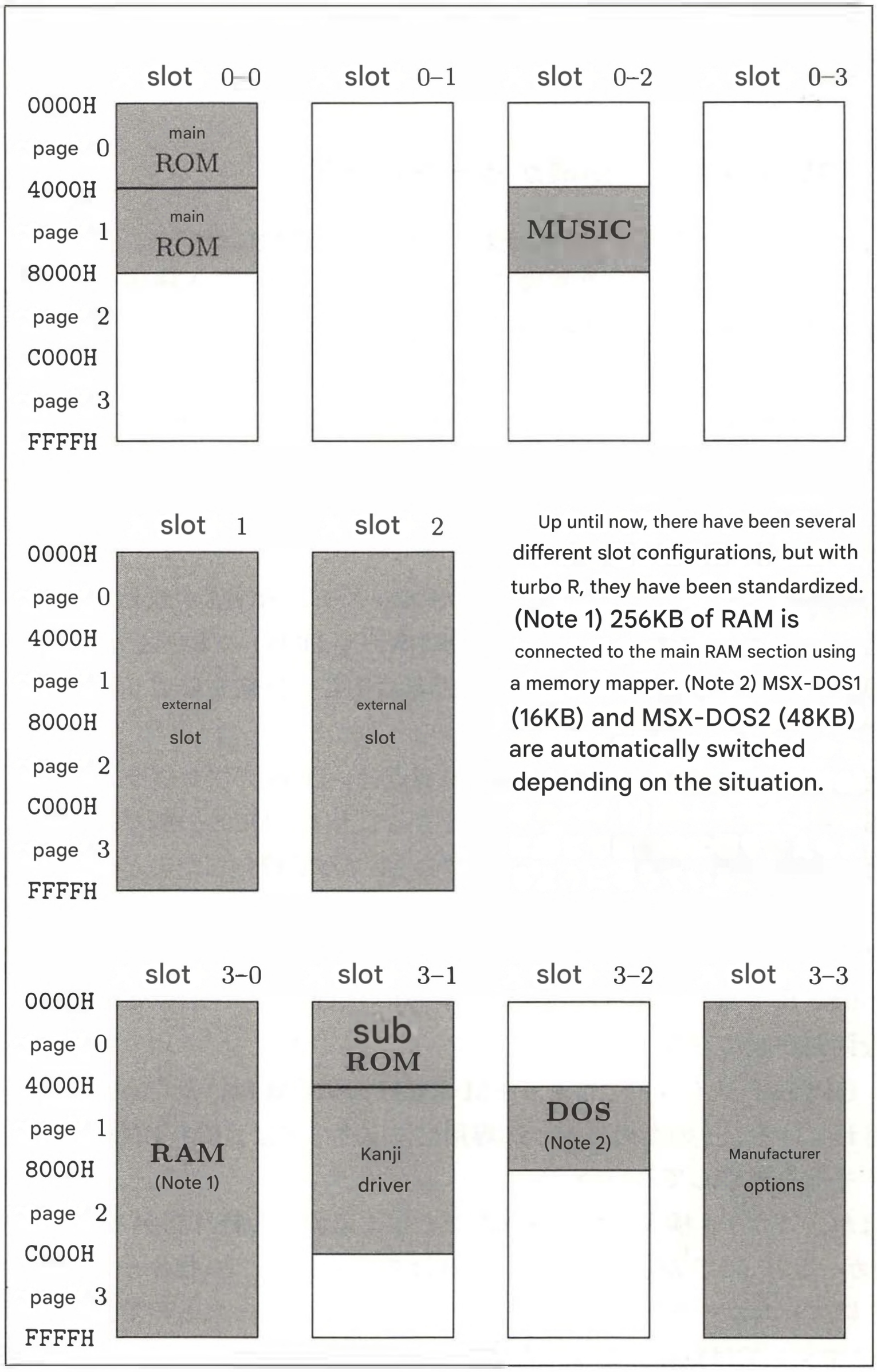
The biggest advantage of this standardized slot configuration is that DOS programs can make interslot calls to the subROM in the normal way, and that DOS interrupt processing programs can be placed in any address. As I mentioned in an old M Magazine, if there was RAM and subROM in the expanded slot 0, there was a possibility that the MSX-DOS interslot call function and interrupt processing programs would go out of control.

However, the turbo R has RAM and sub-ROM in the expanded slot 3, so this problem does not occur.

Also, the OPLL driver, or FM-BIOS ROM, is always placed in slots 0-2. Therefore, the turbo R dedicated software can omit the step of searching for the slot where FM-BIOS is located.

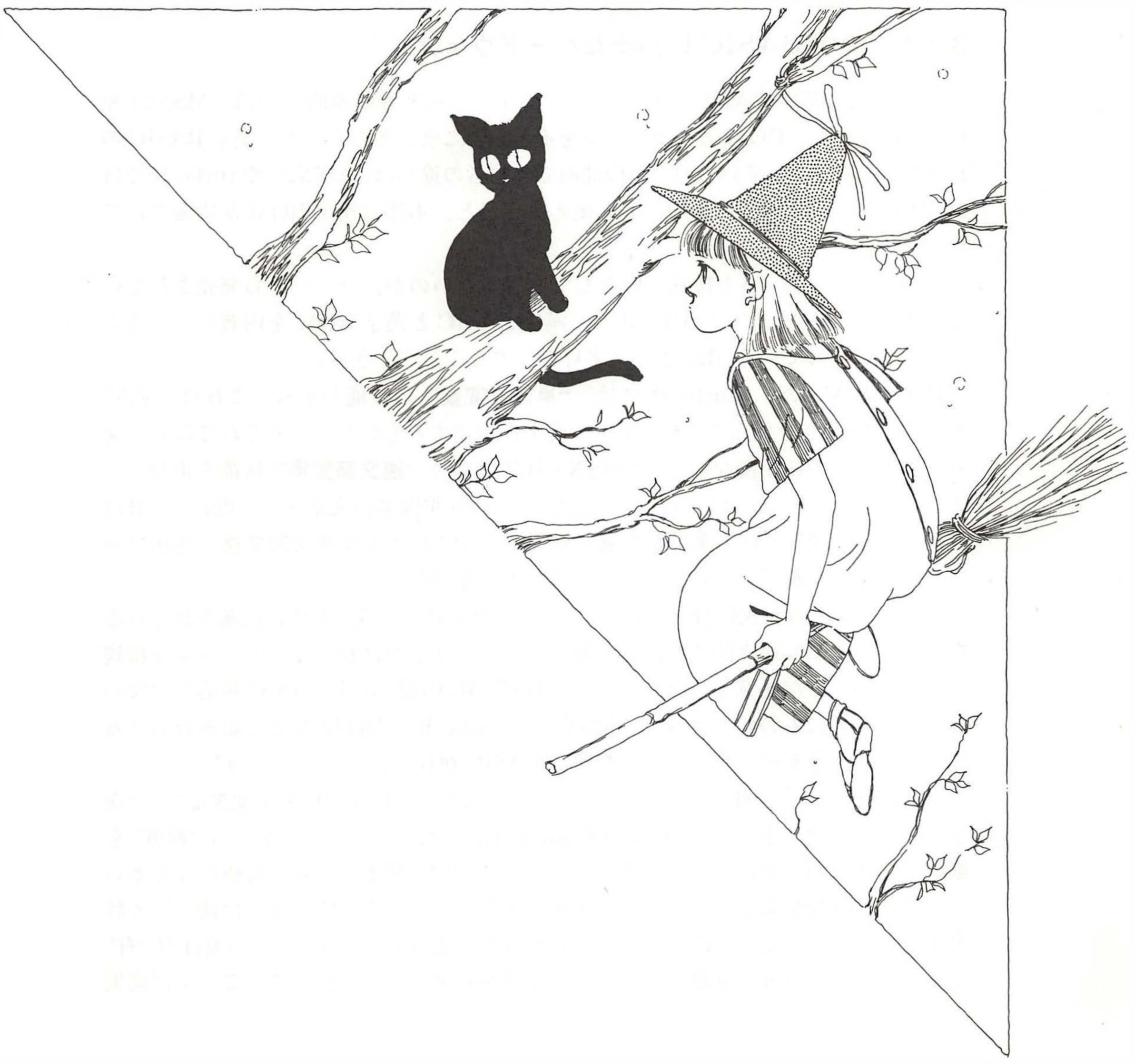
Another special example of the benefits of a unified slot configuration is the "Konami 10x cartridge." This required the 10x cartridge to be in slot 1 and the game cartridge in slot 2, and did not work on some MSX1 and MSX2 machines. However, the MSX2+ and turbo R fixed the external slots to slots 1 and 2, so this kind of special program could be easily and reliably implemented.

Figure 2.8: MSX turbo R slot configuration



And the biggest benefit for software houses is that they will be less bothered by bugs that arise with specific slot configurations, since the slot configuration is now standardized.
From the perspective of software developers, the standardization of slot configuration is far more pleasing than the faster CPU or the increased RAM capacity. Long live turbo R!

第3章 Kanji BASIC



This chapter is a re-edited version of the article "MSX2+ Technical Exploration Team" from the April 1989 issue of MSX Magazine.

3.1 Analyze Kanji BASIC

One of the features of MSX2+ and later machines and DOS2 is the ease of using kanji. Kanji can be used in BASIC program strings and file names. This chapter reports on the features of Kanji BASIC.

3.1.1 Hardware required for Kanji BASIC

How can I use Kanji BASIC? First, the most basic thing is to get an MSX2+ or turbo R main unit, or a DOS2 cartridge. Both of these have a Kanji BASIC ROM built in. The difference between the two in terms of functionality is that MSX2+ and turbo R can use the natural image modes of SCREEN 10~12, and have a Kanji ROM built into the main unit.

Another rather special example worth noting is the Japanese language cartridge "HBI-J1" released by Sony. It has Kanji BASIC and Kanji ROM built-in, so you can make your MSX2 compatible with Kanji.

DOS2, MSX2+, and turbo R have a "single kanji conversion" function. This allows you to specify kanji characters one by one using "reading" and "JIS code". However, this is inconvenient for inputting long sentences, so you can add a "multiple phrase conversion" function called "MSX-JE". For example, the multi-phrase conversion will convert a hiragana sentence such as "Kyou wa ii otenki desu" ("It's a good weather today") into a mixed kanji and kana sentence such as "Kyou wa ii otenki desu" ("It's a good weather today") in one go. This is the same method used in dedicated word processors.

Table 3.1 lists hardware that has MSX-JE built in. If it is built in, you can use the compound phrase conversion function as is, if not, you can just connect a cartridge. However, please note that built-in MSX-JE does not mean it supports Kanji BASIC. As mentioned at the beginning, by combining it with MSX2+, turbo R, DOS2, etc., you can use Kanji BASIC that supports compound phrase conversion.

MSX-JE has a learning function. For example, suppose you try to convert the hiragana character "kanji" and three candidates are displayed: "kanji," "kanji," and "kanji." If you select "kanji" from these, the next time you try to convert the same word, "kanji" will become the first candidate. Words you use frequently are given higher priority, so the more you use the dictionary, the more suited it will be to you and the more efficient the conversion will be. Devices with "Yes" in the "SRAM" entry in Table 3.1 are designed to store the results of this learning using memory whose contents remain even when the power is turned off.

Table 3.1: MSX-JE built-in hardware list

Manufacturer	Product name	form	Kanji ROM	SRAM
panasonic	FS-A1ST	MSX turbo R	1、2	Yes
panasonic	FS-A1WSX	MSX2+	1、2	Yes
panasonic	FS-A1WX	MSX2+	1、2	Yes
panasonic	FS-SR021	cartridge (1)	1、2	Yes
panasonic	FS-4600F	MSX2	1	Yes
panasonic	FS-PW1	printer (2)	1	Yes
sony	HB-F1XV	MSX2+	1、2	Yes
sony	HB-F1XDJ	MSX2+	1、2	Yes
sony	HBI-J1	cartridge	1、2	Yes
Sanyo	PHC-77	MSX2	1	Nothing
HAL Research Institute	HALNOTE	cartridge (3)	1、2	Yes
ascii	MSX-Write	cartridge	1	Nothing
ascii	MSX-Write Ⅱ	cartridge	1、2	Yes

The Kanji ROM item indicates whether or not the first and second level Kanji ROM is included.
"With" SRAM means that learning remains even when the power is turned off. (1) A1WX word processor on a cartridge. (2) A set of cartridge and dedicated printer. (3) A dedicated OS on a cartridge and disk.

3.1.2 What is MSX-JE compatible software?

MSX-JE is not just a word processor or an extended BASIC, but can be used in combination with various applications to provide a multi-phrase conversion function. Kanji ROMs and multi-phrase conversion dictionary ROMs are relatively expensive, so it is very economical for many software programs to share MSX-JE itself.

The way that applications use MSX-JE is defined in the specifications, so software marked as "MSX-JE compatible" can be combined with any MSX-JE. However, HALNOTE in Table 3.1 uses a dedicated OS called "HALOS" in combination with the cartridge and system disk, so there are restrictions on combinations with software that is not for HALOS.

3.1.3 Explaining the operating principle of the Kanji driver

Here is a simple explanation of how kanji characters are represented internally in computers, not just MSX. First, English letters and katakana can be represented with a 1-byte (8-bit) value, but to represent kanji characters, a 2-byte code is required. In JIS (Japanese Industrial Standards), kanji characters are represented with 2-byte codes as follows:

亜…… 3021H (1st level)

腕…… 4F53H (1st level)

弋…… 5021H (2nd level)

𠮟……737EH (2nd level)

However, this "JIS code" has the problem that it becomes difficult to process when English characters and kanji are mixed. Therefore, "Shift JIS code", which is a conversion of the JIS code to make it easier for computers to process, is used in many personal computers, including MSX. Some people call it "Microsoft Kanji Code", but this is something you should only remember if you are interested.

In any case, the Shift JIS code is a convenient kanji code designed to allow software designed for English characters to be used for Japanese characters as is or with only minor modifications. The Shift JIS kanji code is also used in the most popular operating systems for 16-bit PCs, MS-DOS, and OS-9, which are popular operating systems these days. This means that document files can be exchanged between different computers.

By the way, when you look at the kanji code table, you will notice that it contains a mixture of numbers and English letters. To avoid confusion, English letters and kanji characters that are represented by 2-byte kanji codes are collectively called "full-width characters." In contrast, characters that are represented by 1-byte character codes are called "half-width characters," and the two are distinguished from each other. For example, be careful because the half-width characters "ABCD" and the full-width characters "ABCD" are treated as completely different characters.

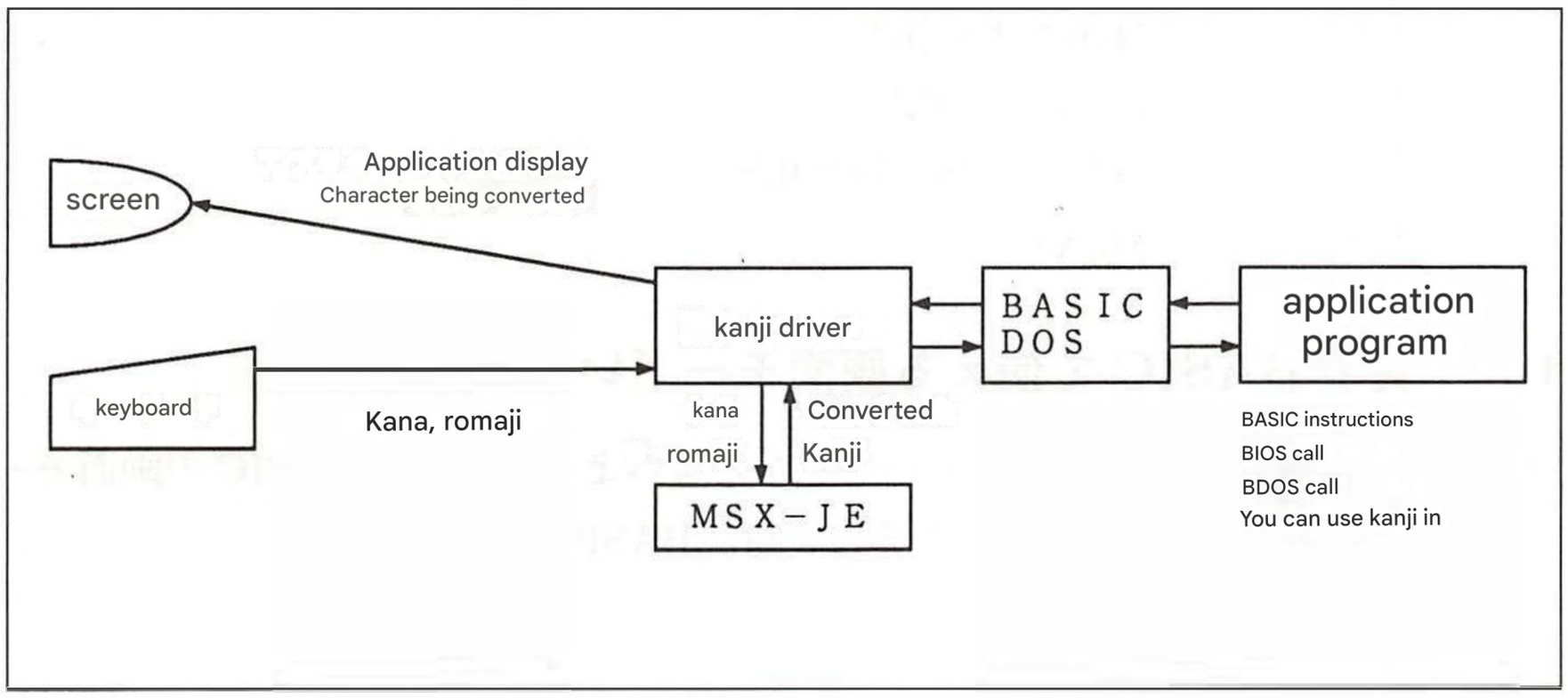
Now, let's get to the main topic. The kanji input/output function of MSX2+, turbo R, and DOS2 is called "kanji driver". This is not limited to BASIC and DOS, but can be used by many application programs.

Figure 3.1 shows how the Kanji driver works. To explain it step by step, it first reads (determines) the full-width "kana" or "romaji" entered from the keyboard and displays it on the screen. Then it calls MSX-JE and converts the hiragana to kanji (full-width characters). Finally, the converted kanji code is sent to the application program via BASIC or DOS.

Conversely, when an application program wants to display characters on the screen, it simply sends the kanji code to the kanji driver.

To application programs, this kanji driver simply appears as if kanji input/output functionality has been added to BASIC instructions, BIOS calls, and BDOS calls (a BIOS-like function used by DOS programs to perform input/output).

Figure 3.1: Kanji driver operation principle



In particular, a major advantage of the kanji driver is that the application program itself does not need to operate the MSX-JE dictionary.

3.1.4 JE compatible hardware & software

For your reference, we have compiled a list of MSX-JE compatible hardware and software that has been released so far.

- modem cartridge

panasonic	FS–CM1
panasonic	FS–CM820
sony	HBI–1200
canon	VM–300
Myojo Electric	V–3

- MSX with built-in modem

panasonic	FS–A1FM
sony	HB–T7
sony	HB–T600
mitsubishi	ML–TS2H

- software

sony	Document Sakuzaemon
sony	Postcard Shoemon
ascii	MSX–TERM
ascii	MSX–DOS2 TOOLS
ascii	MSXView

3.1.5 Various screen modes available in Kanji BASIC

Just as MSX2+ and turbo R have many screen modes, Kanji BASIC's screen modes are also complicated. Table 3.2 shows a list of them. In BASIC,

```
CALL KANJI
WIDTH
```

By command. In DOS2

```
KMODE
MODE
```

Each command specifies the screen mode. For example,

```
CALL KANJIO : WIDTH 32
```

This command will display 32 characters by 12 lines on the screen. To do the same thing in DOS2, use

```
KMODE 0
MODE 32
```

However, if you start up the English version of the system, the Kanji driver is not loaded, so you will need to go back to BASIC and call up the Kanji mode.

Let us now take a closer look at the meaning of Table 3.2. First, "screen pixel count" refers to the number of dots displayed on the screen. Kanji characters are expressed by combining these dots.

The 424-dot vertical screen mode uses a display method called "interlace." This method alternates between two images that are shifted vertically by half a dot, thereby increasing the number of dots on the screen. However, the downside is that the screen flickers, which can tire the eyes.

"Kanji dot count" refers to the number of dots required to display one kanji character. Kanji characters are usually displayed using 16 x 16 dots, but they can be compressed to 12 x 16 dots, allowing 40 kanji characters to be displayed on a 512 dot wide screen. Also, if you connect a Panasonic modem cartridge, the built-in 12 x 12 dot kanji ROM is automatically selected.

Table 3.2: Screen modes of Kanji BASIC

Number of screen dots	Number of kanji dots	Number of half-width characters	Setting method
256 × 212	16 × 16	32 × 12	CALL KANJI0 : WIDTH 32
256 × 212	12 × 16	40 × 12	CALL KANJI1 : WIDTH 40
256 × 424	16 × 16	32 × 24	CALL KANJI2 : WIDTH 32
256 × 424	12 × 16	40 × 24	CALL KANJI3 : WIDTH 40
512 × 212	16 × 16	64 × 12	CALL KANJI0 : WIDTH 64
512 × 212	12 × 16	80 × 12	CALL KANJI1 : WIDTH 80
512 × 424	16 × 16	64 × 24	CALL KANJI2 : WIDTH 64
512 × 424	12 × 16	80 × 24	CALL KANJI3 : WIDTH 80

"Number of half-width characters" is the number of columns and lines of half-width characters displayed on the screen. Naturally, in the case of full-width characters (kanji), the number of columns will be half of the value in the table.

One thing to note here is that not all of the rows in the table can be used by BASIC etc. One or two rows at the bottom of the screen are used for displaying function keys and for kanji conversion.

3.1.6 Kanji text and Kanji graphics

Well, the screen mode is actually a more complicated issue. In BASIC,

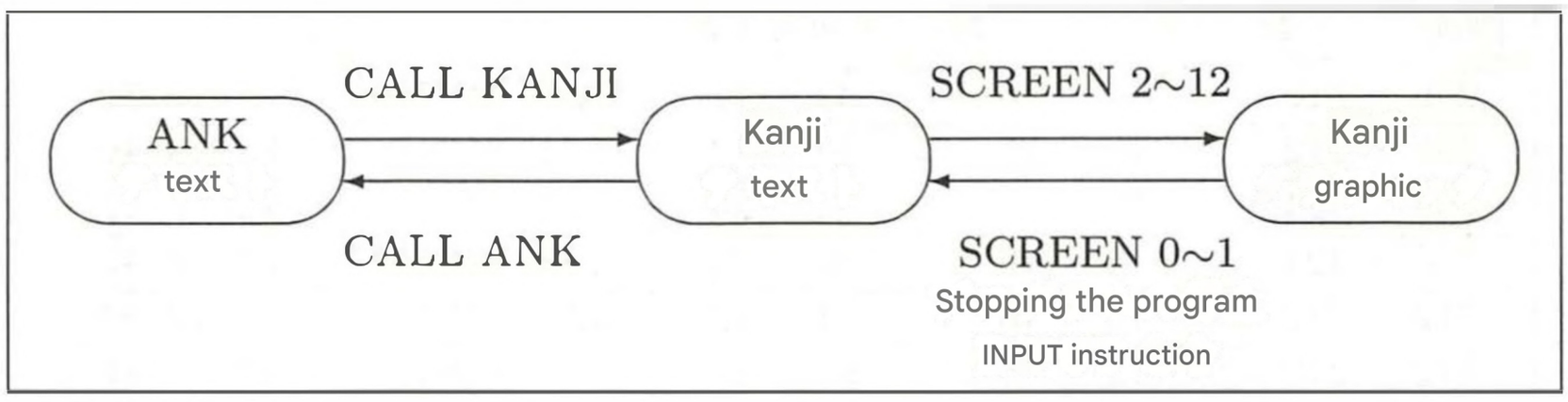
```
CALL KANJI1
WIDTH 40
SCREEN 0
```

When you execute this command, you will see that the second screen mode from the top in Table 3.2 will be selected. What I want you to pay attention to here is that even though "SCREEN 0" is specified, the VDP is set to "SCREEN 5" with 256 x 212 dots. This state is called "Kanji text mode".

In this mode, you can input and edit BASIC programs, input using the INPUT command, output using the PRINT statement, etc., but you cannot use graphic functions such as LINE and PAINT.

To operate graphics in kanji mode, you need to switch the screen to "kanji graphics mode" with a command such as "SCREEN 5". However, please remember that in this mode, although you can use the graphics functions and output kanji, you cannot input kanji as a general rule. Don't get confused. The above screen mode switching is summarized in Figure 3.2.

Figure 3.2: Switching display modes



When a program starts, ends, or an error occurs, make sure the screen mode is set as expected. If you forget to type "CALL KANJI", the program will work correctly right after you enter it, but it will not work if you turn off the power and reload it.

Also, if you use the "LINE" and "LINE INPUT" commands in the Kanji graphics mode, it will automatically switch back to the Kanji text mode. To avoid this, you can read the keyboard with the "INPUT$" and "INKEY$" functions.

3.1.7 This is the correct way to use the Kanji Driver.

The Kanji Driver is a wonderful piece of software that overturns the conventional wisdom that the Kanji function of MSX was useless. However, because it was added to BASIC later, it has some unexpected pitfalls. The difference between Kanji text and Kanji graphics that I just explained is one of those pitfalls. Here, I will list some points to be aware of when using the Kanji Driver.

When the "CALL KANJI" command is executed for the first time after resetting MSX, a work area for using the Kanji driver and MSX-JE is prepared. At that time, the contents of BASIC variables are erased, and a work area called the "software stack" that records the number of repetitions of the "FOR~NEXT" command and the line number to return to with the "RETURN" command is initialized. For example,

```
10 A=1
20 CALL KANJI0
30 PRINT A
```

If you run this program, the first time it will display "0" (meaning the variable value is 0), but from the second time onwards it will correctly display "1". Also, if you run a program like this, when "CALL KANJI0" is executed, the place to return to with the "RETURN" command will be forgotten, and the program will behave strangely.

```
10 GOSUB 80
   ▌
70 END
80 CALL KANJIO
90 RETURN
```

Another problem is that when the Kanji driver allocates a work area in memory, the BASIC work area is reduced by that amount. Programs that use the full memory capacity or programs that use machine language subroutines may run out of memory and not be able to run.

The kanji driver has the ability to convert Shift JIS code within a program into JIS code and print it on a kanji printer. This function works with the BASIC "LPRINT" command and the BIOS "LPTOUT". However, there are side effects when using "bit image printing", which controls the printer one dot at a time to print graphics. Therefore, before bit image printing, it is necessary to write a non-zero value to address F418H (called RAWPRT) in the system work area to disable kanji code conversion.

Now, the following is for advanced programmers. First, Table 3.3 lists the hooks that the Kanji driver rewrites. If other programs rewrite these hooks, the Kanji driver may not work properly, so be careful.

Next, when the kanji driver is called from the hook with an interslot call, the contents of the back registers and the IX and IY registers are destroyed.

Table 3.3: Hooks used by the Kanji driver

address	name	function
FDA4H	H.CHPU	display one character on screen
FDA9H	H.DSPC	Show cursor
FDAEH	H.ERAC	erase cursor
FDB3H	H.DSPF	Show function keys
FDB8H	H.ERAF	Clearing the display of function keys
FDBDH	H.TOTE	Switch screen to text mode
FDC2H	H.CHGE	read one character from keyboard
FDDBH	H.PINL	BASIC editor reads one line
FDE5H	H.INLI	read one line
FFB6H	H.LPTO	write a character on the printer

If an application program uses these hooks, the Kanji driver may not work correctly.

Therefore, Kanji programs that assume that the contents of these registers for the I/O-related BIOS are preserved will not work correctly in kanji mode.

第4章 V9958VDP



Sections 1 to 4 of this chapter are re-edited by the editorial department from "V9938 MSX-VIDEO Technical Data Book" and "V9958 Specifications." Since the common features of V9938 and V9958 are omitted, please refer to "MSX-Datapack" etc.

Sections 5 to 7 of this chapter are re-edited versions of the "MSX2+ Technical Exploration Team" articles from the December 1988, January, November, December, and January 1990 issues of "MSX Magazine."

In hardware documents such as the "V9958 Specification," "VDP modes" are used for explanations, but in this book, we use the BASIC screen mode to match the MSX Magazine article. The VDP modes and BASIC screen modes correspond as shown in Table 4.1.

Table 4.1: VDP modes and BASIC screen modes

VDP mode	BASIC screen mode
TEXT 1	SCREEN 0 : WIDTH 40
TEXT 2	SCREEN 0 : WIDTH 80
MULTI COLOR	SCREEN 3
GRAPHIC 1	SCREEN 1
GRAPHIC 2	SCREEN 2
GRAPHIC 3	SCREEN 4
GRAPHIC 4	SCREEN 5
GRAPHIC 5	SCREEN 6
GRAPHIC 6	SCREEN 7
GRAPHIC 7	SCREEN 8 (SCREEN 10~12)

4.1 V9958 Register list

Table 4.2: Mode Register

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
R#0	0	DG	IE_2†	IE_1	M_5	M_4	M_3	0	Mode 0
R#1	0	BL	IE_0	M_1	M_2	0	SI	MAG	Mode 1
R#2	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	Pattern name T.B.A.
R#3	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	Color T.B.A. (Low)
R#4	0	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	Pattern gen. T.B.A.
R#5	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	Sprite attr. T.B.A. (Low)
R#6	0	0	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	Sprite pat. gen. T.B.A.
R#7	TC_3	TC_2	TC_1	TC_0	BD_3	BD_2	BD_1	BD_0	Text / Back drop color
R#8	MS†	LP†	TP	CB*	VR*	0	SPD	BW*	Mode 2
R#9	LN	0	S_1*	S_0*	IL	EO	$\overline{NT}$*	DC*	Mode 3
R#10	0	0	0	0	0	A_{16}	A_{15}	A_{14}	Color T.B.A. (High)
R#11	0	0	0	0	0	0	A_{16}	A_{15}	Sprite attr. T.B.A. (High)
R#12	$T2_3$	$T2_2$	$T2_1$	$T2_0$	BC_3	BC_2	BC_1	BC_0	Text / Back color
R#13	ON_3	ON_2	ON_1	ON_0	OF_3	OF_2	OF_1	OF_0	Blinking period
R#14	0	0	0	0	0	A_{16}	A_{15}	A_{14}	VRAM access base addr.
R#15	0	0	0	0	S_3	S_2	S_1	S_0	Status reg. pointer
R#16	0	0	0	0	C_3	C_2	C_1	C_0	Color palette addr
R#17	AII	0	RS_5	RS_4	RS_3	RS_2	RS_1	RS_0	Control reg. point r
R#18	V_3	V_2	V_1	V_0	H_3	H_2	H_1	H_0	Display adjust
R#19	IL_7	IL_6	IL_5	IL_4	IL_3	IL_2	IL_1	IL_0	Display offset
R#20*	0	0	0	0	0	0	0	0	Color burst 1
R#21*	0	0	1	1	1	0	1	1	Color burst 2
R#22*	0	0	0	0	0	1	0	1	Color burst 3
R#23	DO_7	DO_6	DO_5	DO_4	DO_3	DO_2	DO_1	DO_0	Interrupt line
R#25‡	0	CMD	VDS*	YAE	YJK	WTE	MSK	SP2	
R#26‡	0	0	HO_8	HO_7	HO_6	HO_5	HO_4	HO_3	Horizontal scroll (High)
R#27‡	0	0	0	0	0	HO_2	HO_1	HO_0	Horizon al scroll (Low)

T.B.A. : table base address

Any bit marked "0" in this table must be written with a 0.

* (Editor's note) Since it is for hardware control, it cannot be rewritten by normal application programs.

†This flag exists in the V9938 but not in the V9958, so you must always write 0.

‡These are new registers added to V9958. Their initial value is 0, and the functions of V9958 are It is equivalent to V9938. Note that register 24 is a missing number.

Table 4.3: Command Register

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
R#32	SX_7	SX_6	SX_5	SX_4	SX_3	SX_2	SX_1	SX_0	Source X (Low)
R#33	0	0	0	0	0	0	0	SX_8	Source X (High)
R#34	SY_7	SY_6	SY_5	SY_4	SY_3	SY_2	SY_1	SY_0	Source Y (Low)
R#35	0	0	0	0	0	0	SY_9	SY_8	Source Y (High)
R#36	DX_7	DX_6	DX_5	DX_4	DX_3	DX_2	DX_1	DX_0	Destination X (Low)
R#37	0	0	0	0	0	0	0	DX_8	Destination X (High)
R#38	DY_7	DY_6	DY_5	DY_4	DY_3	DY_2	DY_1	DY_0	Destination Y (Low)
R#39	0	0	0	0	0	0	DY_9	DY_8	Destination Y (High)
R#40	NX_7	NX_6	NX_5	NX_4	NX_3	NX_2	NX_1	NX_0	Number of dot X (Low)
R#41	0	0	0	0	0	0	0	NX_8	Number of dot X (High)
R#42	NY_7	NY_6	NY_5	NY_4	NY_3	NY_2	NY_1	NY_0	Number of dot Y (Low)
R#43	0	0	0	0	0	0	NY_9	NY_8	Number of dot Y (High)
R#44	CH_3	CH_2	CH_1	CH_0	CL_3	CL_2	CL_1	CL_0	Color
R#45	0	MXC	MXD	MXS	DIY	DIX	EQ	MAJ	Argument
R#46	CM_3	CM_2	CM_1	CM_0	LO_3	LO_2	LO_1	LO_0	Command

Table 4.4: Status Register

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
S#0	F	5SF	C	$5S_4$	$5S_3$	$5S_2$	$5S_1$	$5S_0$	Status 0
S#1	FL†	LPS†	ID_4	ID_3	ID_2	ID_1	ID_0	FH	Status 1
S#2	TR	VR	HR	BD	1	1	EO	CE	Status 2
S#3	X_7	X_6	X_5	X_4	X_3	X_2	X_1	X_0	Column (Low)
S#4	1	1	1	1	1	1	1	X_8	Column (High)
S#5	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	Row (Low)
S#6	1	1	1	1	1	1	EO	Y_8	Row (High)
S#7	C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0	Color
S#8	BX_7	BX_6	BX_5	BX_4	BX_3	BX_2	BX_1	BX_0	Border X (Low)
S#9	1	1	1	1	1	1	1	BX_8	Border X (High)

†These bits relate to features that are present in the V9938 but not in the V9958, so these values are meaningless in the V9958.

The ID of V9958 is 00010B.

4.2 What's New in V9958

4.2.1 horizontal scroll

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	VDS	YAE	YJK	WTE	*MSK*	*SP2*
R#26	0	0	HO_8	HO_7	HO_6	HO_5	HO_4	HO_3
R#27	0	0	0	0	0	HO_2	HO_1	HO_0

HO8 ~ HO。 set the amount of horizontal screen scrolling in 2-dot units for SCREEN 6 and 7, and 1-dot units for all other screen modes.

If SP2 = 0 (initial value), the horizontal screen size is 1 page.

If SP2 = 1, the horizontal screen size is 2 pages.

If MSK = 0 (default value), the left edge of the screen is not masked.

If MSK = 1, the leftmost 16 pixels of the screen are displayed on SCREEN6 and 7, and the other screens are displayed on SCREEN8 and SCREEN9.

In this mode, the leftmost 8 pixels of the screen are masked and the border color is displayed.

For HO8 through HO3, the display screen is shifted to the left by the set value in 8-dot increments (16-dot increments for SCREEN 6 and 7).

Figure 4.1: Horizontal scrolling (SP2=0)

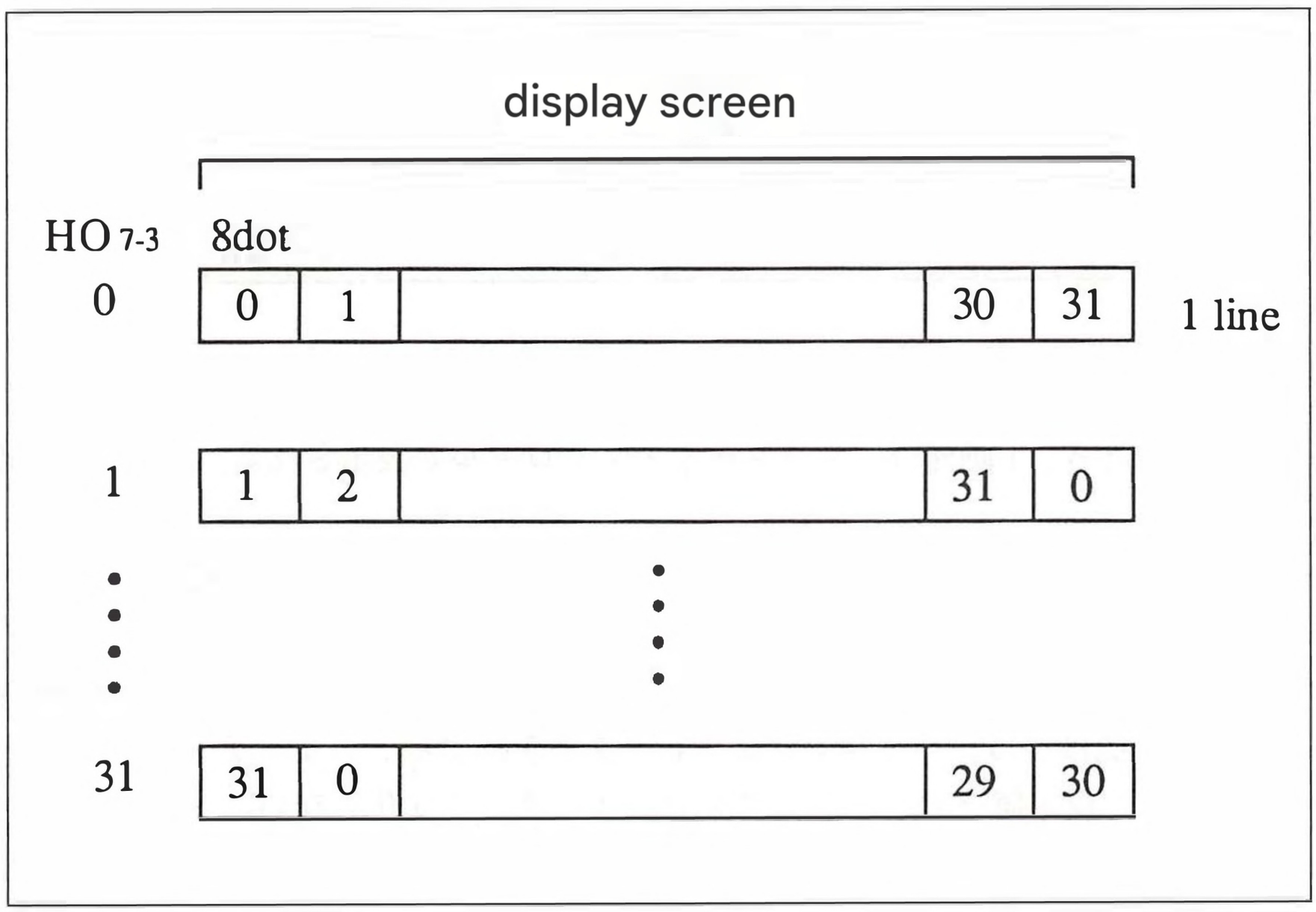
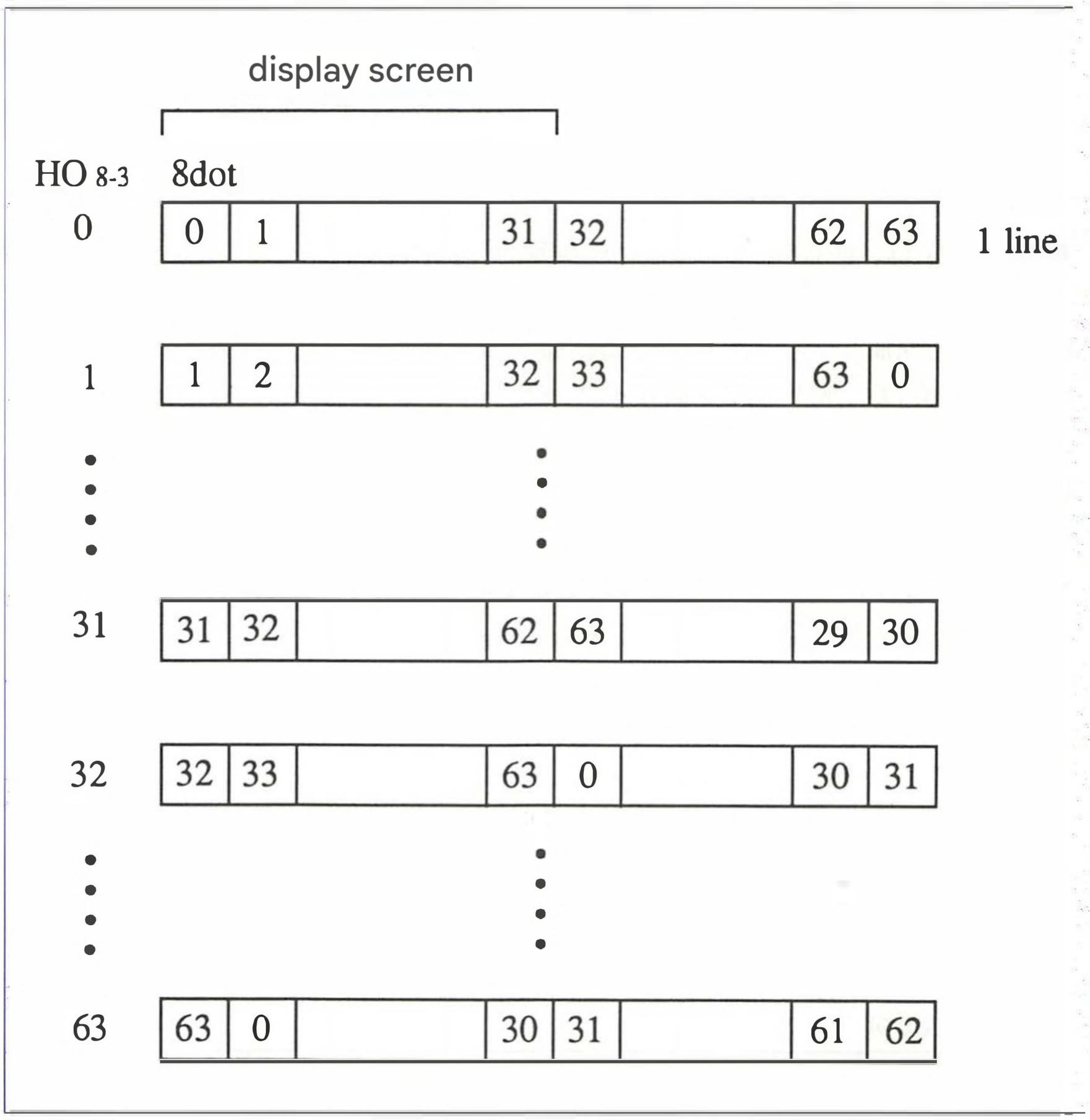


Figure 4.2: Horizontal scrolling (SP2=1)



When SP2 = 0, one screen's worth of data is displayed with horizontal scrolling. HO is ignored.

When SP2 = 1, two screens' worth of data are displayed with horizontal scrolling. Set A15, the base address of the pattern name table, to 1. For base addresses 0 to 31, the setting value for A15 = 0 will be used, and for addresses 32 to 63, the setting value for A15 = 1 will be used. The base addresses of the pattern generator table and color table remain the setting values and do not change when scrolling.

For $HO_8 \sim HO_3$ the display screen is shifted to the right by the set value in 1-dot increments

increments for SCREEN 3 6 and 7).

4.2.2 Wait

	b7	b6	b5	b4	b3	b2	b1	b0
R#25	0	CMD	VDS	YAE	YJK	*WTE*	MSK	SP2

If WTE = 0 (default), the wait function is disabled.

If WTE = 1, the wait function is enabled. When the CPU accesses the VRAM, all accesses to the V9958 ports are put into a wait state until the V9958 VRAM access is completed. There is no wait function for incomplete access to registers and color palettes, or for data ready for a command.

4.2.3 command

	b7	b6	b5	b4	b3	b2	b1	b0
R#25	0	*CMD*	VDS	YAE	YJK	WTE	MSK	SP2

If CMD = 0 (default), the command function is enabled only in screen modes SCREEN 5 to 12.

If CMD = 1, command functionality is enabled in full-screen mode.

In screen modes other than SCREEN 5 to 12, it operates as SCREEN 8.

Parameters are set in the SCREEN 8 X-Y coordinate system.

4.2.4 YJK style display

Register settings

	b7	b6	b5	b4	b3	b2	b1	b0
R#25	0	CMD	VDS	*YAE*	*YJK*	WTE	MSK	SP2

If YJK = 0 (default value), the data on the VRAM is in RGB format (3, 3, 2 bits each). The sprite display color will remain the same as before.

If YJK = 1, the data in VRAM is treated as YJK format, converted to RGB signal (5 bits each), and output as analog from RGB terminal. Palette is effective for sprite display color.

YAE selects the YJK data format.

If YAE = 0

There is no attribute. The data format is as follows. It is expressed by grouping four consecutive dots.

C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
Y					K_L		
Y					K_H		
Y					J_L		
Y					J_H		

If YAE = 1

Each dot has an attribute. The data format is as follows.

It is represented by grouping four dots.

C_7	C_6	C_5	C_4	C_3	C_2	C_1	C_0
Y				A	K_L		
Y				A	K_H		
Y				A	J_L		
Y				A	J_H		

If A = 0 (initial value), Y, J, and K will all be YJK format data.

If A = 1, the Y data is a color code and is output as RGB via the color palette. J and K are YJK format data.

Conversion formula between YJK and RGB formats (reference)

$$
\begin{aligned}
R &= Y + J \\
G &= Y + K \\
B &= \tfrac{5}{4}Y - \tfrac{1}{2}J - \tfrac{1}{4}K
\end{aligned}
$$

$$
\begin{aligned}
Y &\equiv \tfrac{1}{2}B + \tfrac{1}{4}R + \tfrac{1}{8}G \\
J &= R - Y \\
K &= G - Y
\end{aligned}
$$

(Editor's note) The Y value is an integer between 0 and 31 if the attribute is not present, and an even number between 0 and 30 if the attribute is present. The J and K values are integers between -32 and 31. The result of the conversion from YJK to RGB is clipped to 0 to 31.

4.3 Discontinued features of V9958

The following features that were present in V9938 have been discontinued:

- Composite video output
- Mouse/light pen interface

(Editor's note) MSX mice do not use the V9938's mouse interface function, so removing this function will have no effect.

4.4 V9958 Hardware Specifications (Changes)

Table 4.5: Terminal changes for V9958

number	V9958			V9938	
	name	I/O	explanation	name	I/O
4	$\overline{\text{VRESET}}$	I	Separate HSYNC/CSYNC 3-value logic inputs	$\overline{\text{VDS}}$	O
5	$\overline{\text{HSYNC}}$	O	HSYNC output or burst flag output	$\overline{\text{HSYNC}}$	I/O
6	$\overline{\text{CSYNC}}$	O		$\overline{\text{CSYNC}}$	I/O
8	CPUCLK / $\overline{\text{VDS}}$	O	CPUCLK output or VDS output	CPUCLK	O
21	AVDD (DAC)	I	analog power supply	VIDEO	O
26	$\overline{\text{WAIT}}$	O	I/O WAIT output	$\overline{\text{LPS}}$	I
27	$\overline{\text{HRESET}}$	I	Separate HSYNC/CSYNC 3-value logic input	$\overline{\text{LPD}}$	I

When the VDS flag in control register 25, bit 5 is 0, terminal 8 becomes the CPUCLK output; when the VDS flag is 1, terminal 8 becomes the VDS output.

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
R#25	0	CMD	*VDS*	YAE	YJK	WTE	MSK	SP2

Table 4.6: V9958 DC characteristics

VRESET, HRESET

symbol	item	minimum	standard	maximum	unit
V_{IL}	low level input voltage	-0.3		0.8	V
V_{IH}	High level input voltage	2.2		V_{CC}	V

HSYNC, CSYNC, CPUCLK / VDS, WAIT

symbol	item	Measurement conditions	minimum	standard	maximum	unit
V_{OL}	Low level output voltage	$I_{OL} = 1.6\text{mA}$			0.4	V
V_{OH}	High level output voltage	$I_{OH} = 0.1\text{mA}$	2.4			V

G, R, B

symbol	item	Measurement conditions	minimum	standard	maximum	unit
V_{RGB31}	Maximum output voltage	$R_L = 470\Omega$		2.8		V
V_{RGB0}	Minimum output voltage	$R_L = 470\Omega$		2.0		V
V_{P-P}	$V_{RGB31} - V_{RGB0}$	$R_L = 470\Omega$		0.8		V
D_{RGB}	Vp-p deviation	$R_L = 470\Omega$			5	%

4.5 V9958 and MSX2+

The part that controls the screen display of MSX is called the "Video Display Processor", or VDP for short. The VDP for MSX2 was called "V9938", but in MSX2+ and later machines, its functionality was upgraded to "V9958". Here, we will introduce the functions added to the V9958.

4.5.1 There are 12 screen modes in total

Screen mode is the screen state that can be switched using the BASIC SCREEN command. For example, if you want to write a BASIC program with a 40-column display,

```
SCREEN 0 : WIDTH 40
```

If you want to switch the screen to text mode and draw graphics,

```
SCREEN 8
```

Switch to the graphic mode by using etc. Having many screen modes is troublesome, so it's better to have fewer screen modes. But there is a reason why MSX2+ has many screen modes.

Table 4.7: MSX2+ display modes

number	Display method	resolution	number of colors
0	Character(1)	80 x 24 characters	Specify the text color and background color
1	Character(2)	32 × 24 characters	Specify the text color and background color
2	Table(3)	256 x 192 dots	16 colors, 2 colors for every 8 horizontal dots
3	Bitmap (4)	64 x 48 dots	16 colors
4	table	256 x 192 dots	16 colors, 2 colors for every 8 horizontal dots
5	bitmap	256 x 212 dots	16 colors
6	bitmap	512 x 212 dots	4 colors
7	bitmap	512 x 212 dots	16 colors
8	bitmap	256 x 212 dots	256 colors
9	It is designed to display Hangul characters and is not available on the Japanese MSX.		
10	YJK/RGB	256 x 212 dots	12,499 colors (see text)
11	YJK/RGB	256 x 212 dots	12,499 colors (see text)
12	YJK	256 x 212 dots	19,268 colors (see text)

(1) Each character is represented by 6 dots x 8 dots, and English letters and katakana are displayed.

(2) Each character is represented by 8 dots x 8 dots, and it displays English letters, katakana, and hiragana.

(3) The screen is created by combining 8x8 dot patterns.

(4) In a bitmap display, the color of a dot can be displayed without regard to the colors of its neighbors.

The first issue is speed. When you output characters to a screen in graphics mode, it takes time to display them. Therefore, when you are inputting programs, text mode is convenient because it displays characters quickly, although you cannot use figures or kanji.

The second reason is that the more powerful the screen (the more pixels and colors it can display), the more data it requires. For example, the screen data for a single SCREEN8 screen is 54,272 bytes, which means that only 12 images can be stored on one disk (2DD). For this reason, ROM cartridge games often use SCREEN 2 to save memory, which has a limited number of colors but uses less data and runs faster.

Thirdly, many screen modes were required to add features while maintaining compatibility. The first MSX only had four screen modes, SCREEN 0 to 3. With MSX2, SCREEN 4 to 8 were added to display high-resolution graphics. Furthermore, with MSX2+, SCREEN 10 to 12 were added to increase the number of colors.

These screen modes are summarized in Table 4.7. However, the screen mode issue does not end here. There is also an interlace mode that doubles the vertical resolution, and a kanji mode that was newly added to MSX2+.

4.5.2 Controlling the VDP register

The VDP, which controls the MSX screen display, contains "registers" just like the CPU, the heart of the machine. The VDP's registers can be operated by the CPU through the I/O ports. The registers that the CPU uses to control the VDP are called "control registers," and those that the CPU uses to know the status of the VDP are called "status registers." There is also a "command register" that the CPU operates to have the VDP execute advanced commands, but we won't go into detail here as we won't use it in the programs in this book.

The I/O port address that connects the CPU and the VDP usually ranges from 98H to 9BH. However, to be precise, it is determined by the contents of ROM addresses 6 and 7, as shown in Table 4.8. This was done to allow for the addition of a VDP outside the MSX main unit.

Table 4.8: VDP I/O Ports

port name	R/W	I/O address	Purpose
port 0	READ	Contents of ROM address 6	VRAM read
port 1	READ	Contents of ROM address 6 +1	status register
port 0	WRITE	Contents of ROM address 7	VRAM write
port 1	WRITE	Contents of ROM address 7 +1	control register
port 2	WRITE	Contents of ROM address 7 +2	pallet register
port 3	WRITE	Contents of ROM address 7 +3	indirect specification register

Please note that even if it is the same port 0, the I/O port address may be different when writing and when reading.

Now, to set the value of the VDP's control register, first write the data you want to set to port 1 in Table 4.8, and then write "register number + 128" to the same port 1. This "consecutively" condition is surprisingly important, as the VDP will become confused if an interrupt occurs between the two writes. Therefore, in this case, it is common to first disable interrupts with the "DI" command, and then write the two pieces of data in succession.

By the way, control registers are write-only. Once a value is set, it cannot be read. This makes it inconvenient when you want to change only a specific bit in a register. Therefore, the usual method is to write the value to be written to the control register to the system work area (RAM) as shown in Table 4.9. For example, if you want to change bit 4 of control register 1 to 1, you read the contents of address F3E0H in RAM, change bit 4 to 1, and write that value to control register 1 and address F3E0H in RAM. In the sample program for scan line interrupts in List 4.9, which will be introduced later, the subroutine "WRTVDP" performs the same operation.

Next, we will explain how to read the value of a status register. This involves setting the number of the status register you want to read in control register 15, reading the value of VDP port 1, and resetting control register 15 to 0, all while interrupts are disabled. In the sample program in List 4.9, this is the subroutine called "_VDPSTA."

Table 4.9: Control Register Storage Locations

register number	VDP function number	Save address	label
0	0	F3DFH	RG0SAV
⋮	⋮	⋮	⋮
7	7	F3E6H	RG7SAV
8	9	FFE7H	RG8SAV
⋮	⋮	⋮	⋮
23	24	FFF6H	RG23SA
25	26	FFFAH	RG25SA
26	27	FFFBH	RG26SA
27	28	FFFCH	RG27SA

Table 4.10: Other useful system work areas

address	label	meaning
F341H	RAMAD0	Slot number of RAM for page 0*
F342H	RAMAD1	Page 1 RAM slot number*
F343H	RAMAD2	Page 2 RAM slot number*
F344H	RAMAD3	Page 3 RAM Slot Number*
FAF5H	DPPAGE	display page number
FAF6H	ACPAGE	active page number
FD9AH	H.KEYI	interrupt hook
FD9FH	H.TIMI	timer interrupt hook

* Valid only if disk is present.

Table 4.11: System work areas added or changed in MSX2+

address	label	meaning
0FAFCH	MODE	See next table
0FAFDH	NORUSE	Kanji driver work area
0FD0AH	SLTWRK+1	
⋮	⋮	
0FD0FH	SLTWRK+6	
0FFFAH	RG25SA	Saving VDP registers
0FFFBH	RG26SA	
0FFFCH	RG27SA	

Table 4.12: 0FAFCH Address (MODE) Details

bit	meaning
b_7	1 is katakana, 0 is hiragana
b_6	If 1, then 2nd level kanji ROM is available
b_5	If it's 1 then SCREEN 11, if it's 0 then SCREEN 10
b_4	used internally
b_3	If 1, mask 3FFFH to VRAM address with SCREEN 0~3
b_2	VRAM capacity
b_1	00：16KB、01：64KB、10：128KB
b_0	If 1, convert to romaji and kana

However, since MSX2 and 2+ ROMs have BIOS with the same functions as these subroutines, you can usually use the BIOS without creating your own subroutines. However, these BIOS are in sub ROMs or call sub ROMs during processing,

The drawback is that it takes a little time, which is why it is not convenient for processes like scan line interrupts, and so it does not use the BIOS.

Other system work areas are listed in Tables 4.10 to 4.12 for reference.

4.5.3 V9958 Register

Figure 4.3 shows the three control registers added to the V9958. These registers are write-only, and the values written are recorded in the system work area in Table 4.9. Note that there is no control register 24, and that the register numbers are different from the VDP function numbers in BASIC.

Most of the features added in V9958 are controlled by control register 25. I'll leave the famous YJK (natural image) display and horizontal scrolling for later and start by introducing the remaining features.

Bit 7 of register 25 must be written as 0. Bit 5 is called "VDS" and controls the function of VDP terminal 8. However, normal programs are prohibited from modifying this bit.

Also, if you write a 0 to bit 6, you will be able to use "VDP commands" only for screens SCREEN 5 to 8, just like the V9938. Conversely, if you write a 1, you will be able to use VDP commands in full-screen mode. These VDP commands are functions that make the VDP perform tasks like the BASIC COPY and LINE commands. This is a bit of a minor point, but VDP commands for screens other than SCREEN 5 to 8 specify a location in the 128KB video RAM with an X coordinate value between 0 and 255 and a Y coordinate value between 0 and 511, just like SCREEN 8.

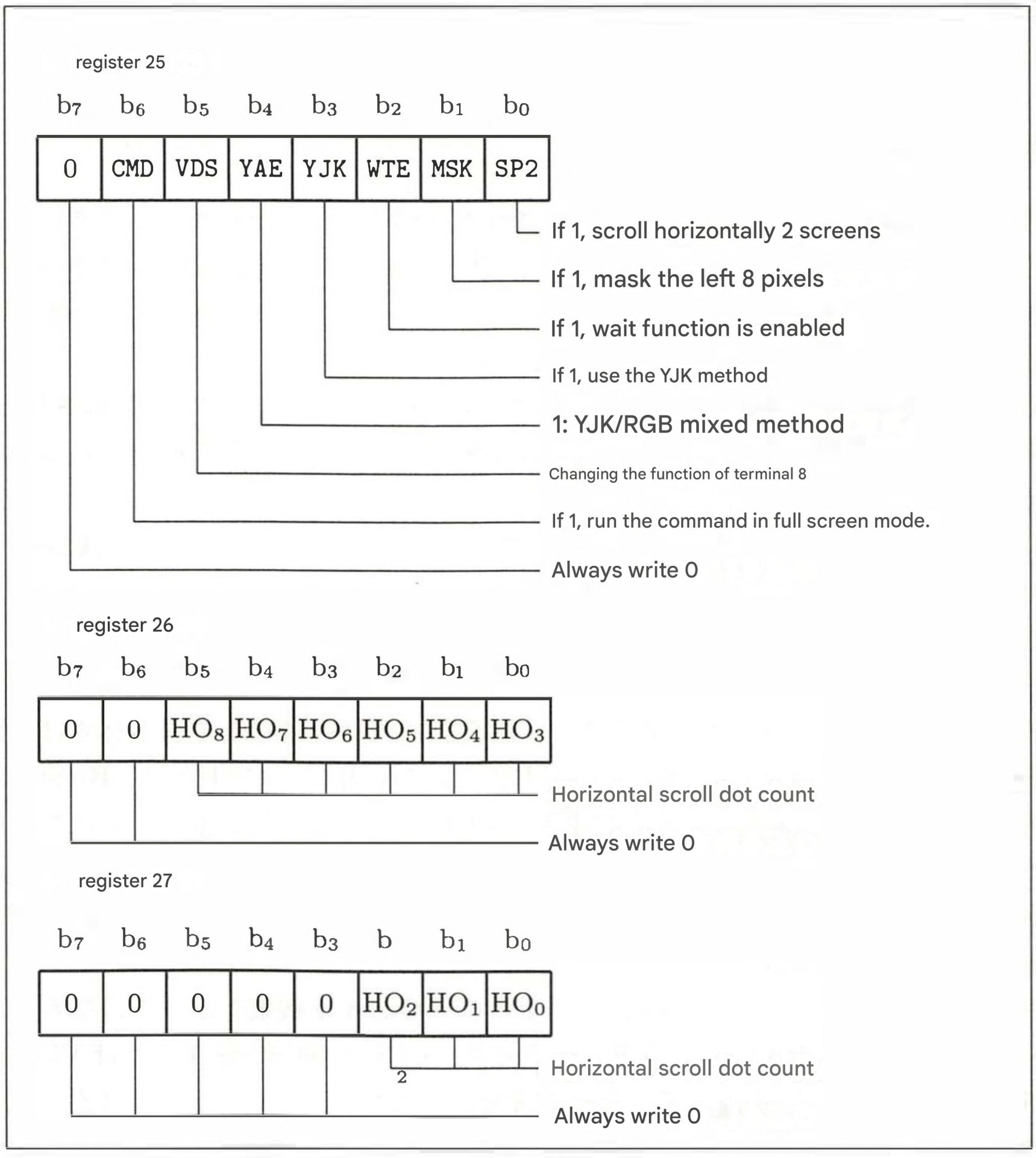
Furthermore, writing 1 to bit 2 enables the VDP's "wait function." This is a function that causes the VDP to send a "WAIT signal" to the CPU to wait if the CPU is too fast when reading or writing from or to the video RAM. However, there is no wait function for writing to the palette register or for transfers by VDP commands.

4.5.4 Horizontal scrolling with VDP

There are two ways to scroll horizontally: using one screen's worth of image data, or using two screens' worth of image data. We'll explain each method in turn.

First, when bit 0 "SP2" of control register 25 is 0, horizontal scrolling of one screen's worth of data occurs, as shown in the top of Figure 4.4. The number of dots to scroll can be specified with registers 26 and 27. Writing a value from 0 to 63 into register 26 scrolls the screen to the left by that value x 8 dots, and writing a value from 0 to 7 into register 27 moves the screen to the right by that number.

Figure 4.3: List of control register functions added to V9958

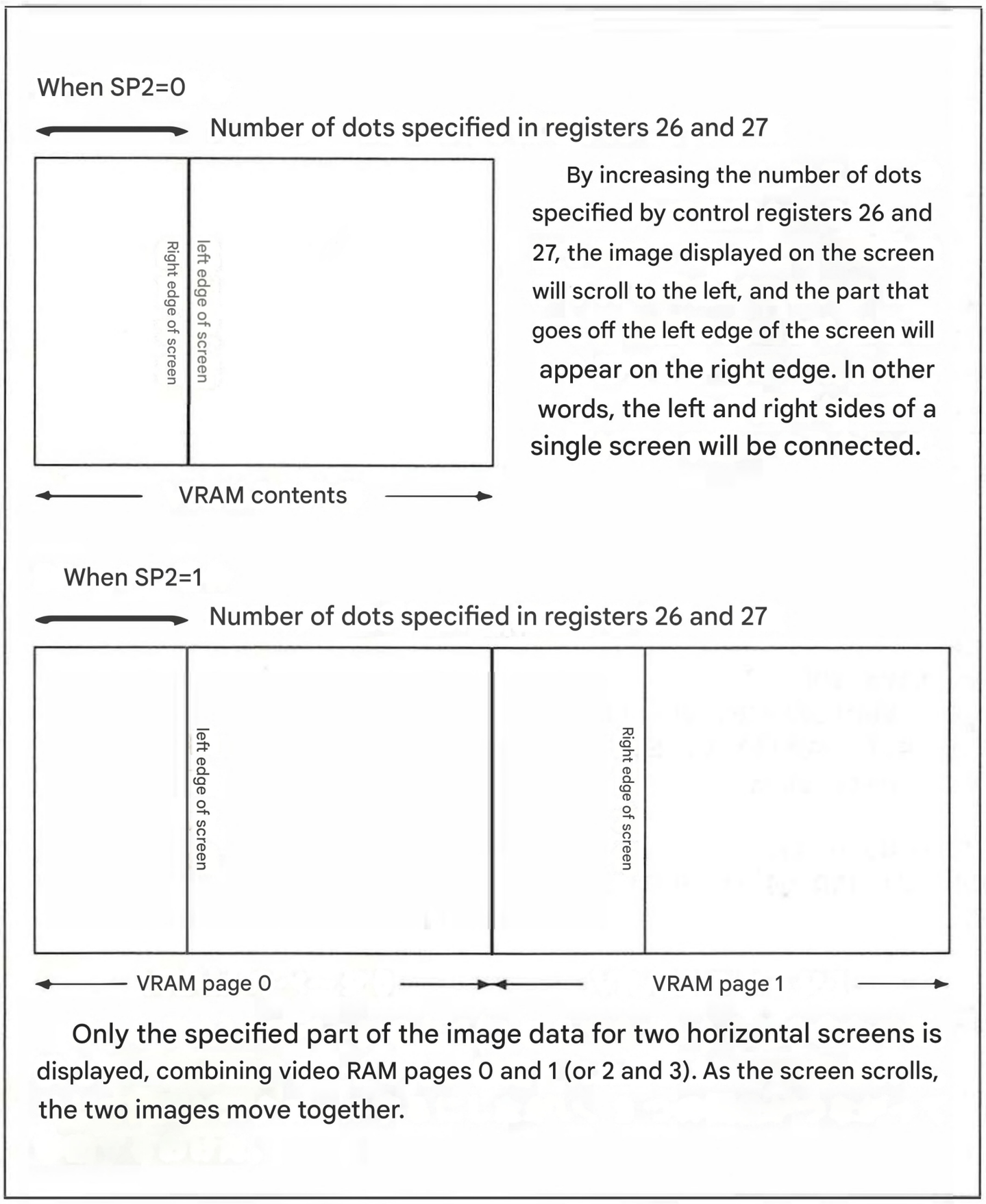


Note, however, that in SCREEN 6 and 7, scrolling will occur by twice the number of dots you specify. As you increase the number of dots in sequence from 0 to 255, the screen will scroll to the left, and any excess will appear from the right edge.

On the other hand, if bit 0 of register 25 is 1, a specified portion of two screens' worth of image data is displayed and horizontal scrolling begins, as shown in the bottom of Figure 4.4.

At this time, the image data is stored in pages 0 and 1 or 2 and 3 of the video RAM.

Figure 4.4: Two types of horizontal scrolling mechanisms



By increasing the number of dots specified by control registers 26 and 27, the image displayed on the screen will scroll to the left, and the part that goes off the left edge of the screen will appear on the right edge. In other words, the left and right sides of a single screen will be connected.

Only the specified part of the image data for two horizontal screens is displayed, combining video RAM pages 0 and 1 (or 2 and 3). As the screen scrolls, the two images move together.

For pages 0 and 1, set the display page to 1, for pages 2 and 3, set the display page to 3. The number of dots for horizontal scrolling goes from 0 to 511 (i.e. 2 times one screen scroll).

Also, if you set bit 1 of register 25 to 1, the 8 dots (16 dots for SCREEN6 and 7) on the left edge of the screen will not be displayed, and instead the screen border color will be displayed in that location. This is useful for horizontal scrolling, especially when scrolling horizontally by one screen's worth of data, to hide the part that goes off the screen and immediately appears on the other side. List 4.1 is an example program for horizontal scrolling, so please refer to it.

List 4.1 **(HSCROLL.BAS)**

```
100 ' hscroll.bas
110 ' by nao-i on 26. Oct. 1989
120 '
130 ONSTOP GOSUB 290: STOP ON
140 DEFINT A-Z: SCREEN 5
150 COLOR 15,1,1: CLS
160 LINE (0,0)-(255,211)
170 SET PAGE 1,1: COLOR 15,8,1: CLS
180 LINE (0,0)-(255,211)
190 V6=VDP(26) :VDP(26)=V6 OR 3
200 '*** scroll
210 FOR V7=0 TO 63
220   GOSUB 320: VDP(27)=V7: VDP(28)=7
230   FOR V8=6 TO 0 STEP -1
240     GOSUB 320: VDP(28)=V8
250   NEXT V8
260 NEXT V7
270 GOTO 200
280 '*** restore VDP
290 STOP OFF: VDP(26)=V6: SET PAGE 0,0
300 COLOR 15,4,7: SCREEN 0: END
310 '*** wait next vsync
320 TIME=0
330 IF TIME=0 GOTO 330
340 IF (VDP(-2) AND 64)=0 GOTO 340
350 RETURN
```

4.5.5 Whatever you do, don't use any tricks

Some bits in the VDP registers are specified to always write 0 or always write 1. Never ignore this specification and write strange values, as you never know what will happen.

Also, like the combination of YJK=0 and YAE=1, there are settings for which the VDP's behavior is not determined by the specifications, but you should not use these "tricks" that are not written in the specifications. Even if it works fine on your machine, it just happened to work. There is no guarantee that it will work properly on other MSX machines. Also, even if a trick is effective for the current V9958, it may not be effective if the VDP is improved in the future or if a manufacturer other than Yamaha makes a V9958-compatible VDP (currently all V9958s are made by Yamaha).

In the same way, you should never use CPU tricks (officially called "undefined instructions"). In the past, there was a case where a software ran out of control in turbo mode on Victor's HC-95 (which uses the HD64180, a CPU that is superior to the Z80) because of a Z80 trick.

4.6 Dissecting the YJK Method

4.6.1 Television broadcasting and the YJK system

The MSX2+'s Natural Image mode uses the YJK format instead of the traditional RGB format, which is similar to the signal used in color television broadcasts.

Many readers of this book may not know this, but television broadcasts used to be in black and white, and naturally all television receivers were black and white. Later, when color broadcasts began, two problems arose. If a color screen is broken down into the three colors RGB (red, green, blue) and broadcast, it uses up three channels. Also, black and white receivers need to be able to receive color broadcasts.

To solve this problem, the properties of the human eye were utilized. The retina has cells that sense brightness and cells that sense color. Because there are more cells that sense brightness and fewer cells that sense color, the human eye is insensitive to color changes.

In color television broadcasts, colors are represented by a combination of Y signals, which represent "brightness," and UV signals, which represent "hue." By taking advantage of the fact that the human eye is insensitive to color changes, the amount of UV signal (technically called frequency band) can be small, and YUV signals can be sent together on a single channel. Also, since the Y signal, which represents brightness, is the same as the signal used in black-and-white broadcasts, when you watch a color broadcast on a black-and-white receiver, only the Y signal is displayed, and you see a normal black-and-white screen.

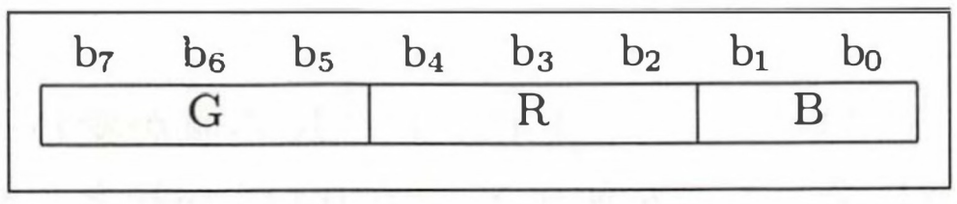
In this way, the YUV method was used in television to broadcast color images without increasing the amount of radio waves (i.e. the number of channels). On the other hand, the MSX2+ uses the YJK method, which is similar to the YUV method, to increase the number of colors without increasing the video RAM.

4.6.2 RGB and YJK data structures

RGB method (SCREEN 8)

In SCREEN 8, the brightness of R, G, and B is represented by 3, 3, and 2 bits, respectively, and Each dot can display any of 256 colors.

Figure 4.5: Data structure of RGB screen



YJK method (SCREEN 12)

SCREEN 12 uses the YJK format, which uses 4 horizontal dots as a set. Yo through Y3 in Figure 4.6 represent the brightness of each dot with 5 bits (i.e. 32 levels), and K and J represent the hue of the entire set of 4 dots with 12 bits (4096 colors). The method for converting YJK format data to RGB format is as follows:

$$\begin{aligned} R &= Y + J \\ G &= Y + K \\ B &= 1.25Y - 0.5J - 0.25K \end{aligned}$$

However, Y values range from 0 to 31, and J and K values range from -32 to 31. If the R, G, or B value calculated by the above formula is less than 0, 0 is output instead of that value, and similarly if it is greater than 31, 31 is output. This process is called "clipping from 0 to 31."

For example, the following four-byte data represents black, with Y = 0 and J = K = −32. Let's try calculating it with a calculator.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	1	0	0

Let's consider the opposite: converting RGB data to YJK data.

In other words, we can think of the conversion formula to RGB as a simultaneous linear equation with three unknowns, and solve it for Y, J, and K. This is a high school level problem of "algebra and geometry". The answer is the following three equations. Did you get it right?

$$\begin{aligned} Y &= (2R + G + 4B)/8 \\ J &= (6R - G - 4B)/8 \\ K &= (-2R + 7G - 4B)/8 \end{aligned}$$

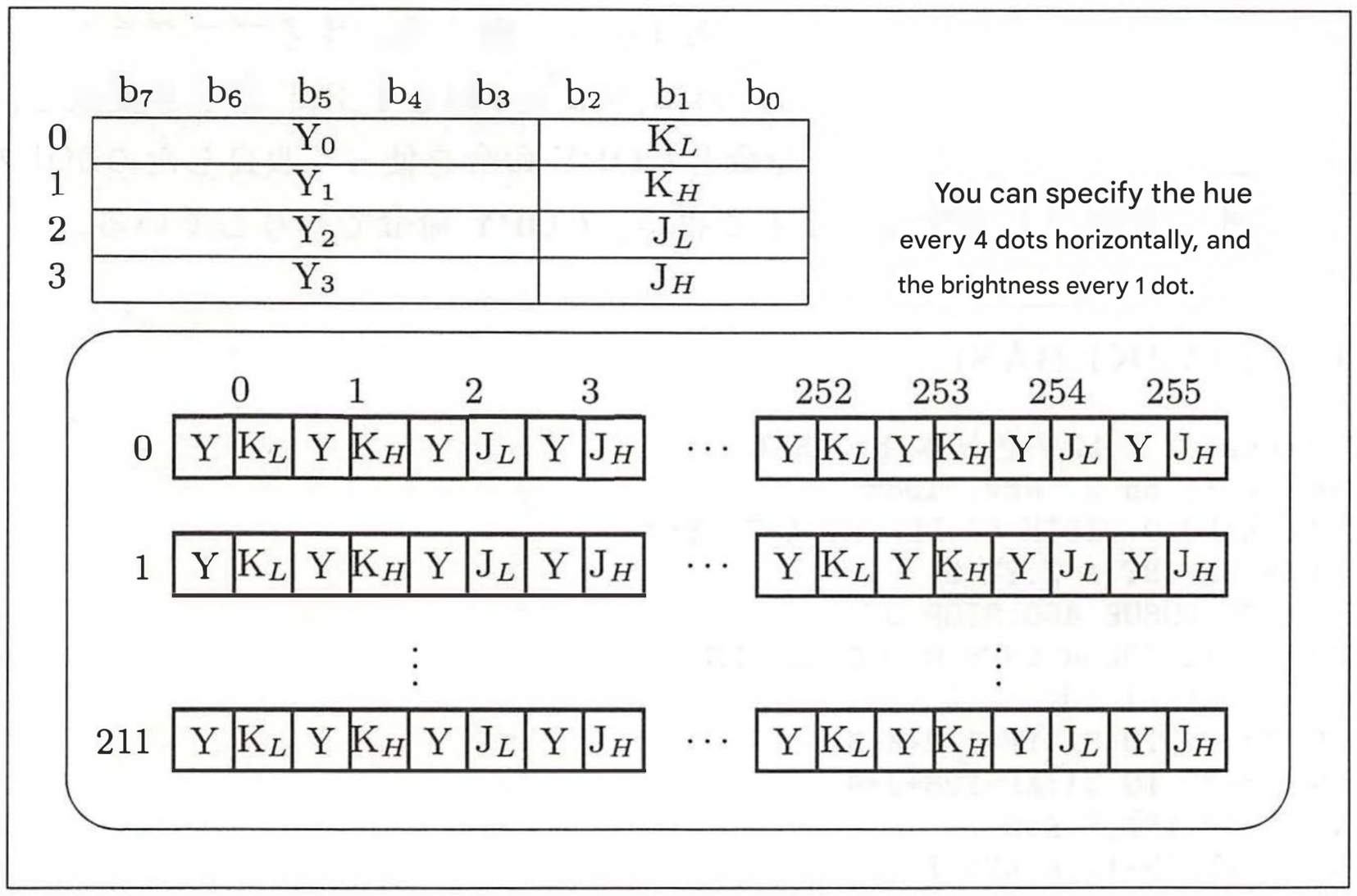
Even when displaying YJK format images, the signal output from the VDP is converted to the same analog RGB signal and composite (video) signal as before using the formula explained earlier, so no special monitor TV is required for the MSX2+.

In YJK screens, like SCREEN 8, one byte basically corresponds to one dot. However, the J and K values are specified every four dots horizontally, so for example,

```
PSET (0,0),0
```

When you do this, the K values of the four pixels from (0,0) to (3,0) change. This is why using the LINE command on SCREEN 12 results in "garbled colors." We'll explain this again later.

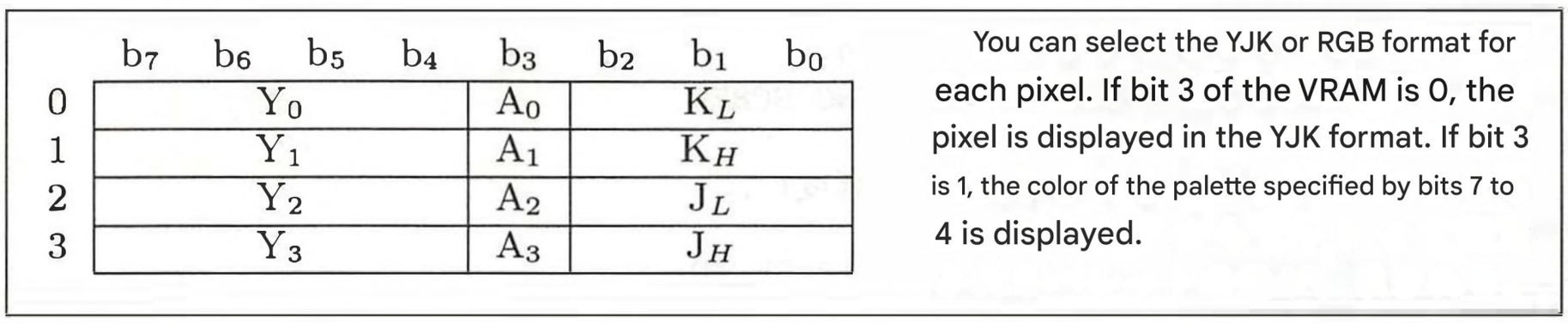
Figure 4.6: Data structure of the YJK method screen



Mixed method (SCREEN 10, 11)

The disadvantage of the YJK method is that it is difficult to overlay text or line drawings on natural images because the color can only be specified every four horizontal dots. Therefore, SCREEN 10 and 11 were developed, which combine the advantages of SCREEN 5, which uses the RGB method, and SCREEN 12, which uses the YJK method. In these screen modes, it is easy to overlay text on natural images, but the number of colors is fewer than SCREEN 12.

Figure 4.7: Data structure of a mixed mode screen



4.6.3 color sample program

The biggest selling point of the MSX2+ is the 19,268-color SCREEN 12. In order to increase the number of colors without increasing the video RAM (128KB, the same as the MSX2), a display method called YJK is used.

As explained so far, this is a method of specifying a color by combining the Y value, which represents luminance (brightness), and the J and K values, which represent hue.

List 4.2 is a program that displays all the colors that can be expressed with YJK. However, this program uses the PSET command 32,768 times to set the values of J and K, which takes a lot of time. Therefore, List 4.3 is an improvement using the LINE and COPY commands. The value of K is written in only one column on the left side with PSET, and then copied with the COPY command.

Listing 4.2 **(YJK1.BAS)**

```
100 ' Color samples for screens 11 and 12 by PSET
110 'by nao-i on 2. Nov. 1988
120 CALL KANJI0:WIDTH 64:DEFINT A-Z:YY=0
130 CLEAR:DEFINT A-Z:YY=0
140 ON STOP GOSUB 400:STOP ON
150 SCREEN 12:COLOR &HF8,0,0:CALL CLS
160 ' Set VRAM J and K
170 FOR K=-32 TO 31:YP=112+K+K
180 FOR J=-32 TO 31:XP=128+J*4
190 PSET (XP,YP),K AND 7
200 PSET (XP,YP+1),K AND 7
210 PSET (XP+1,YP),(K AND 56)¥8
220 PSET (XP+1,YP+1),(K AND 56)¥8
230 PSET (XP+2,YP),J AND 7
240 PSET (XP+2,YP+1),J AND 7
250 PSET (XP+3,YP),(J AND 56)¥8
260 PSET (XP+3,YP+1),(J AND 56)¥8
270 NEXT:NEXT
280 ' Change the value of Y
290 SC=12:NS=1:SCREEN 12
300 FOR Y=1 TO 31:GOSUB 360:NEXT
310 FOR Y=30 TO 0 STEP -1:GOSUB 360:NEXT
320 SC=11:NS=1:SCREEN 11
330 FOR Y=2 TO 30 STEP 2:GOSUB 360:NEXT
340 FOR Y=28 TO 0 STEP -2:GOSUB 360:NEXT
350 GOTO 280
360 ' Subroutine to rewrite Y in VRAM
370 IF NS THEN LOCATE 1,0:PRINT USING"SCREEN ##";SC:NS=0
380 LOCATE 1,1:PRINT USING" The Y value (brightness) is ##.  ";Y
390 LINE(0,48)-(255,175),(Y XOR YY)*8,BF,XOR:YY=Y:RETURN
400 'Triggered by on stop
410 SCREEN 0:COLOR 15,4,7:END
```

list **4.3 (YJK2.BAS)**

```
100 ' Color samples of screen 11 and 12
110 'by nao-i on 2. Nov. 1988
120 CALL KANJI0:WIDTH 64:DEFINT A-Z:YY=0
130 CLEAR:DEFINT A-Z:YY=0
140 ON STOP GOSUB 380:STOP ON
150 SCREEN 12:COLOR &HF8,0,0:CALL CLS
160 ' Set VRAM J and K
170 FOR K=-32 TO 31:YP=112+K+K
180 PSET (0,YP),K AND 7
190 PSET (0,YP+1),K AND 7
200 PSET (1,YP),(K AND 56)¥8
210 PSET (1,YP+1),(K AND 56)¥8:NEXT
220 FOR J=-32 TO 31:XP=128+J*4
230 LINE(XP+2,48)-(XP+2,175),J AND 7
240 LINE(XP+3,48)-(XP+3,175),(J AND 56)¥8
250 COPY(0,48)-(1,175)TO(XP,YO):NEXT
260 ' Change the value of Y
270 SC=12:NS=1:SCREEN 12
280 FOR Y=1 TO 31:GOSUB 340:NEXT
290 FOR Y=30 TO 0 STEP -1:GOSUB 340:NEXT
300 SC=11:NS=1:SCREEN 11
310 FOR Y=2 TO 30 STEP 2:GOSUB 340:NEXT
320 FOR Y=28 TO 0 STEP -2:GOSUB 340:NEXT
330 GOTO 260
340 ' Subroutine to rewrite Y in VRAM
350 IF NS THEN LOCATE 1,0:PRINT USING"SCREEN ##":SC:NS=0
360 LOCATE 1,1:PRINT USING"The Y value (brightness) is ##.  ";Y
370 LINE(0,48)-(255,175),(Y XOR YY)*8,BF,XOR:YY=Y:RETURN
380 'Triggered by on stop
390 SCREEN 0:COLOR 15,4,7:END
```

4.6.4 It's a deadly logical operation.

Logical operation is a Japanese term for logic operations. It refers to calculations performed on each bit of a binary number. There are four types: AND, OR, XOR, and NOT. An AND operation is an operation that sets a bit that has two values that are both 1 to 1. For example,

```
X%=&B00000011
Y%=&B00000101
```

When performing an AND operation on these, only bit 0 (the rightmost bit) of variables X% and Y% is 1, so

```
X% AND Y%=&B00000001
```

Similarly, an OR operation sets bits that are 1 in either or both of two values to 1, and an XOR operation sets bits that are 1 in only one of two values to 1. NOT is an operation that inverts each bit of a value, and writing inverted image data is also called a PRESET operation. It's difficult to understand in words, so we've summarized the details in Table 4.13.

Table 4.13: Logical Operations

symbol	meaning	example
PSET	Writes the specified color as is	
AND	Write the logical product of the original color	0011 AND 0101 → 0001
OR	Write the logical OR with the original color	0011 OR 0101 → 0111
XOR	Write the exclusive OR with the original color	0011 XOR 0101 → 0110
PRESET	Writes the bit-negated version of the specified color.	NOT 0101 → 1010

Now, to handle video RAM on MSX2 or MSX2+, you need to consider these logical operations. As an example, let's consider increasing the Y value of the video RAM from 1 (00001 in binary) to 2 (00010) in the program in List 4.3 previously posted. In other words,

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	0	1	?	?	?

The contents of the video RAM are

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	0	0	1	0	?	?	?

Bits 2 through 0 contain the value J or K, so do not change them.

What I came up with was to multiply the original Y value (1) by the XOR of the target value (2), which is 3, by 8.

```
LINE(0,48)-(255,175), 24, XOR
```

and it's done. With one rewriting, the Y value changes from 1 to 2. This corresponds to line 370 in listing 4.3

```
LINE (0,YO)-(255,YO+127), (Y XOR YY)*8, BF, XOR
```

where Y is the desired Y value and YY is the original Y value.

When a logical operation is specified with the LINE command, the result of the operation between the color to be written and the original color is written to the video RAM. For example, for a SCREEN 12 screen,

```
LINE (0,0)-(255,211), &B11111000, BF, AND
```

will leave bits 7 through 3 of the video RAM unchanged, but set bits 2 through 1 to 0, causing the entire screen to be displayed in shades of black and white.

4.6.5 so-called discoloration

The data structure of the YJK screen is as previously summarized in Figure 4.6. As with SCREEN 8, one dot on the screen basically corresponds to one byte of video RAM. A screen that is 256 dots wide by 212 dots high is displayed using 54,272 bytes of video RAM. However, the J and K values that specify the hue are specified every four dots horizontally, so for example, in SCREEN 12,

```
PSET (3,0),3
```

When you execute this, the value of J will change not only for the single dot at (3,0), but for the four dots from (0,0) to (3,0), and this area will turn red. If you save the SCREEN 12 screen to a file called "sample.s12" and then execute Listing 4.4, you should see what is known as "color corruption." Let's try it out for ourselves. As a rule, SCREEN 12 cannot display characters or lines.

list 4.4 (S12.BAS)

```
10 ' Example of discoloration
20 CALL KANJI
30 SCREEN 12:COLOR &HF9,2
40 BLOAD "sample.s12",S
50 LINE (0,0)-(211,211),3
60 LOCATE 4,6:COLOR &HF9,2
70 PRINT "In Screen 12 colors get corrupted"
80 GOTO 80
```

4.6.6 What is the difference between SCREEN 10 and 11?

The data structures of SCREEN 10 and 11 are summarized in Figure 4.7 above. Functionally, they are exactly the same. The differences between these screen modes are not apparent when displaying a BLOADed screen. So what is the difference? This becomes an issue when you try to write shapes or characters to the screen using BASIC commands, etc.

List 4.5 is a program example that shows the difference. It loads screen data recorded using the YJK method,

```
CIRCLE (128,106),100,6
```

Let's try this. In SCREEN 10, bit 3 of the video RAM becomes 1, and the value 6 (the red color number specified by the circle command, 0110 in binary) is written to bits 7 through 4. This makes it possible to draw a red circle without destroying the background YJK screen. However, in SCREEN 11, the value 6 (i.e. 00000110 across 8 bits) is written directly to the video RAM, causing a "color corruption" effect.

As you can see from this example, use SCREEN 10 to overlay text and shapes on a YJK screen.

To process YJK screen data, you need to use SCREEN 11.

list 4.5 (S10.BAS)

```
100 ' How to avoid discoloration
110 SCREEN 12
120 BLOAD "sample.s12",S
130 SCREEN 10
140 CIRCLE (128,106),100,6
150 FOR I%=1 TO 50:BEEP:NEXT
160 SCREEN 12
170 BLOAD "sample.s12",S
180 SCREEN 11
190 CIRCLE (128,106),100,6
200 GOTO 200
```

4.6.7 How to use subtitles in SCREEN 11

Listing 4.6 is a program that writes characters to the screen of SCREEN 11. Although we have just mentioned using SCREEN 10 to display characters, there are cases where SCREEN 11 is more convenient due to the PRINT command.

For example, when you use the LINE or PUT KANJI command on SCREEN 10, lines and characters are written in RGB format on the YJK screen. However, when you use the PRINT command, a square frame appears in the background and the characters are written inside it. This is not suitable for use as a subtitle.

That's where SCREEN 11 comes in. The SET PAGE command switches the video RAM to page 1, and BLOADs the background screen there. Then it resets the page to 0, switches the foreground color to 7 and the background color to 0, and displays text with the PRINT command. Then it copies this text to the desired part of the image data on page 1 by specifying "TAND" with the COPY command.

TAND is a function that does not copy the parts with color number 0 (transparent), but copies only the parts with other colors while performing an AND operation. As a result, the text frame written with color number 0 is ignored, and only the text written with color number 7 (light blue) is copied. Bits 7 to 3 of the video RAM in the area where you want to display the text become 0.

Next, write the same character on page 0 in the color specified by (C x 16 + 8), where C is the color number entered earlier in the program.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
C				1	0	0	0

In this way, bits 7 to 4 are the target color number, bit 3 which specifies RGB display is 1, and the remaining bits 2 to 0 are 0. If you copy this character by specifying TOR, the contents of the video RAM for the character will be as follows:

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
C				1	original value		

This means that the characters can be displayed in RGB format without destroying the YJK screen in the background. In this way, by copying characters written in another location with the COPY command, the lack of a logical operation function in the PRINT command can be compensated for.

The MSX2+ YJK method is difficult, but if you use it well, you can get great results. Let's all try our best to research it.

list 4.6 (S11.BAS)

```
100 ' SCREEN 11 telop
110 ' by nao-i on 4. Nov. 1988
120 SCREEN 0:WIDTH 32:CALL KANJI0
130 DEFINT A-Z:ON STOP GOSUB300:STOP ON
140 FILES:PRINT:INPUT "file name";FF$
150 OPEN FF$ FOR INPUT AS #1:CLOSE #1
160 INPUT "string";SS$
170 INPUT "color(1...15)";C
180 INPUT "X(0...240)";X
190 INPUT "Y(0...196)";Y
200 SCREEN 12:SET PAGE 1,1:BLOAD FF$,S
210 SCREEN 11
220 SET PAGE 0,0:COLOR 7,0:CALL CLS
230 L=LEN(SS$)*8-1
240 LOCATE 0,0:PRINT SS$
250 COPY (0,0)-(L,15),0 TO (X,Y),1,TAND
260 LOCATE 0,0:COLOR C*16+8,0:PRINT SS$
270 COPY (0,0)-(L,15),0 TO (X,Y),1,TOR
280 SET PAGE 1,1
290 GOTO 290
300 '*** called by STOP ***
310 SET PAGE 0,0:COLOR 15,4,7
```

4.6.8 This is a trick to display text in SCREEN 12

In principle, SCREEN 12, which uses the full YJK method, cannot display characters or lines. However, there is a trick to display white characters using logical operations, which I will introduce here. The program in List 4.7 is the trick. The structure of the program itself is almost the same as List 4.6. However, please note that

```
COLOR &HF8, 0
```

and,

```
COPY  ···  , TOR
```

These are the two parts. Now, you can set the Y value of the part where the text will be displayed to the maximum of 31, and the text will be displayed in a whiter color than the background. Conversely, if you set the text color to 3 and the logical operation to TAND, the text will be displayed in a darker color.

When you run the program, a list of files on the disk will be displayed first, so select the image file you want to use as the background. Next, write the caption you want to display on the screen and enter the position using coordinates. The caption should now be displayed on the SCREEN 12 screen. It might be fun to modify the program to display lots of captions.

list 4.7 (S12T.BAS)

```
100 ’Telop on SCREEN 12
110 ’ by nao-i on 4. Nov. 1988
120 SCREEN 0:WIDTH 32:CALL KANJI0
130 DEFINT A-Z:ON STOP GOSUB260:STOP ON
140 FILES:PRINT:INPUT "file name";FF$
150 OPEN FF$ FOR INPUT AS #1:CLOSE #1
160 INPUT "string";SS$
170 INPUT "X(0...240)";X
180 INPUT "Y(0...196)";Y
190 SCREEN 12:SET PAGE 1,1:BLOAD FF$,S
200 SET PAGE 0,0:COLOR &HF8,0:CALL CLS
210 L=LEN(SS$)*8-1
220 LOCATE 0,0:PRINT SS$
230 COPY (0,0)-(L,15),0 TO (X,Y),1,TOR
240 SET PAGE 1,1
250 GOTO 250
260 ’*** called by STOP ***
270 SET PAGE 0,0:COLOR 15,4,7
```

4.6.9 YJK method and VDP register

Here we will introduce a method for displaying in the YJK format by using a machine language program to manipulate the VDP registers instead of BASIC's SCREEN 10~12.

Bit 3 “YJK” and bit 4 “YAE” of control register 25 allow you to select between RGB screen display and YJK screen display.

First, set all registers except register 25 in the same way as for SCREEN 8.

In the BASIC language, specifying screen modes from 10 to 12 allows you to select the YJK format. However, in terms of the VDP's functionality, it is easier to understand if you think of it as switching the SCREEN 8 display between RGB format and YJK format.

If bit 3 (YJK) is 0 and bit 4 (YAE) is also 0, then what is displayed is the same as SCREEN 8. Then, without changing any other registers, if bit 3 (YJK) is switched to 1, the display switches to the YJK format, the same as SCREEN 12.

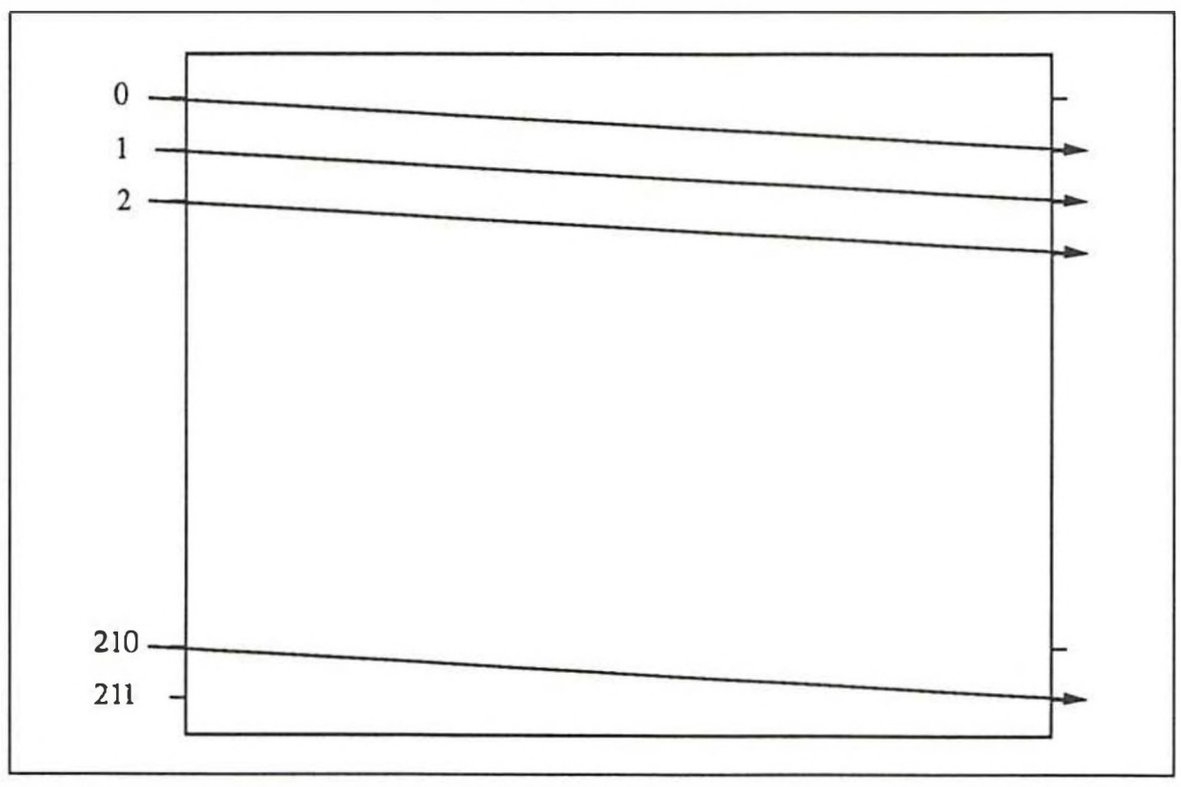
To display a screen in SCREEN 10 or 11, which uses a mixture of YJK and RGB, set YJK to 1 and YAE to 1. Note that if YJK is 0, the sprite's color will not be affected by the palette, but if YJK is 1, it can be changed by the palette.

4.7 Studying Scanline Interrupts

4.7.1 How does the monitor screen display?

You probably know that the MSX2 SCREEN5 screen is represented by a collection of dots (pixels) that are 256 pixels wide and 212 pixels high. Figure 4.8 shows how this is actually displayed on a monitor. The screen is displayed by sending the data for each pixel line by line in order, from the top left to the bottom right of the screen. At this time, a group of 256 pixels lined up horizontally is called a "scan line." In other words, 256 pixels gathered horizontally make up a scan line, and 212 scan lines gathered vertically make up the screen.

Figure 4.8: Scan lines on a television screen



A non-interlaced screen display method, where the gaps between the scan lines are clearly visible.

Not only computer screens, but also regular television broadcasts use these scanning lines. In Japan and the United States, the "NTSC" television broadcasting system uses 525 scanning lines to display the screen. However, the actual number of lines displayed on the screen is about 490. The remaining scan lines contain signals called "sync signals" that control the television and signals for teletext. 30 pictures are displayed per second, and each picture contains 525 scan lines, so 525 x 30 = 15,750 scan lines are displayed on the screen in just one second. This is the meaning behind the values "vertical scan frequency 30 Hz, horizontal scan frequency 15.75 kHz" in the TV broadcast and MSX screen display specifications.

A 640 x 400 pixel display, commonly found in 16-bit computers, requires a dedicated monitor with a horizontal scanning frequency of 24 kHz to display that many pixels. The latest computers have even finer screen resolution, and use monitors with a frequency of 31.5 kHz or 34 kHz.

Multi-scan monitors are compatible with screens with a variety of horizontal scan frequencies. There are a variety of models available, from those that support any frequency from 15.75kHz to 34kHz, to those that can switch between 15.75kHz and 24kHz. If you are planning to buy a new monitor, carefully research the specifications of the computer you have and the computer you want to use in the future, and choose a multi-scan monitor that supports many horizontal scan frequencies.

For reference, Western European countries use monitors in a format called "PAL," while France and the Soviet Union use "SECAM," which have a different number of scan lines than the NTSC format. Therefore, even if you connect a European computer or an MSX for export to a Japanese monitor, you will not be able to display the screen. Companies such as Sony and Victor make monitors that can switch between NTSC, PAL, and SECAM formats, but these are very expensive because they are used for special purposes such as inspecting computers for export (Panasonic has recently released a video deck that can play videos from around the world).

Also, this is a bit of a digression, but the "horizontal resolution" which indicates the performance of a VTR or video disc has nothing to do with the number of scanning lines. It is an indication of the fineness of the screen, equivalent to the number of dots in the horizontal direction on a computer screen. Vertical resolution is the same as the number of scanning lines, and is the same for all VTRs. However, "EDß" and "SVHS" have better horizontal resolution than conventional VTRs.

4.7.2 Interlaced TV broadcasting

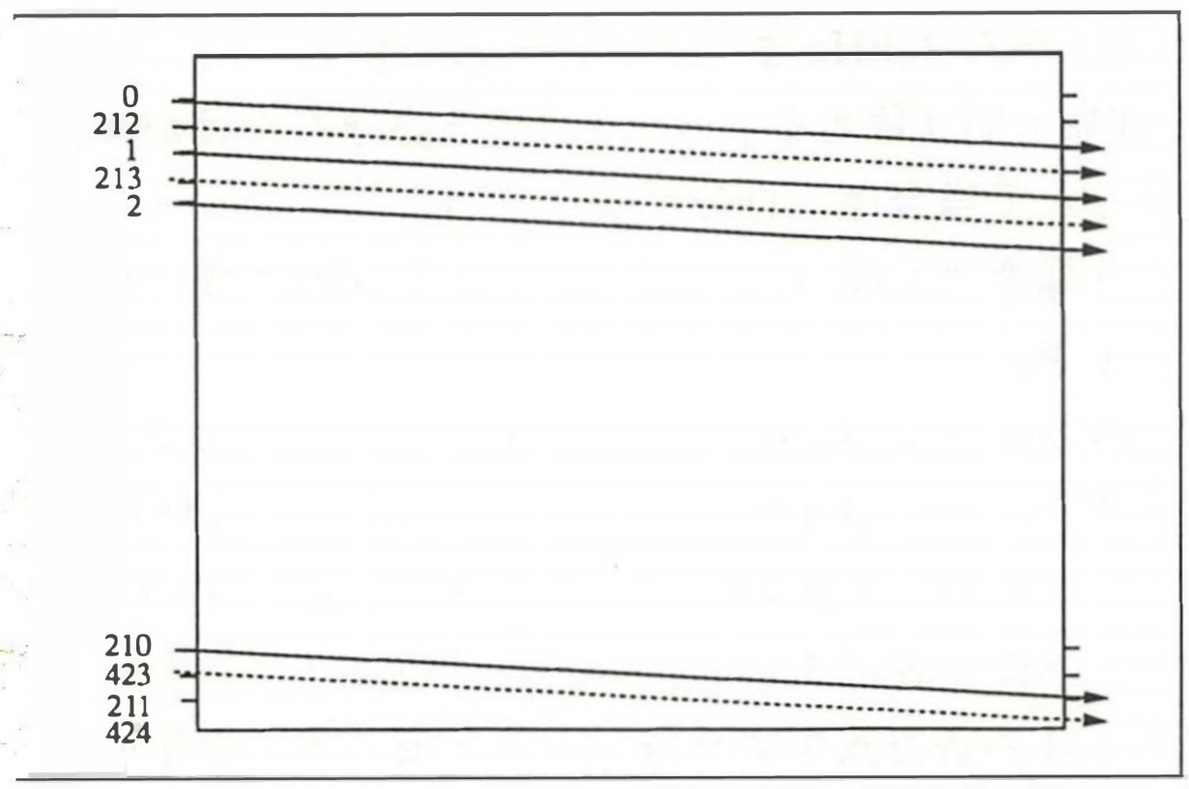
The mechanism for displaying a computer screen is shown in Figure 4.8, but when it comes to television broadcasting, strictly speaking, things are a little different. First, take a good look at Figure 4.9.

This explains the screen display method known as "interlace." This method is actually used in television broadcasts. First, let's call the scan line at the top of the screen number 0. Then, scan lines 1, 2, 3, and so on are displayed in the same way as explained in Figure 4.8. Then, when it gets to the bottom of the screen, scan lines 212, 213, 214, and so on are displayed in order, filling in the spaces between each of the scan lines just displayed.

The word interlace literally means "to interweave." By filling in the gaps between the scan lines explained in Figure 4.8 with another scan line, the gaps between pixels are eliminated. Incidentally, the method of displaying all the scan lines in sequence without interlacing, as in Figure 4.8, is called "non-interlace."

So why was the interlaced format adopted for television broadcasting? It was to make noise that occurs from various sources and the resulting disturbances on the screen less noticeable.

Figure 4.9: This is what happens in interlaced mode



This is how interlaced screens are displayed. The gaps between the scan lines are no longer visible.

For example, even if scan lines 0 and 1 are disturbed by noise, if scan line 212, which is displayed after a short delay, is normal, the disturbance on the screen will not be noticeable. Interlacing is an effective method because television broadcasts are susceptible to various types of noise, such as noise picked up while radio waves are traveling through the air, noise caused by other electrical appliances, and noise picked up from power outlets.

On the other hand, the disadvantage of the interlaced method is that the position of adjacent scanning lines is slightly misaligned, which makes the screen appear to flicker. This is not so noticeable on a moving screen such as a television broadcast, but when small characters are displayed in a still state, such as on a computer screen, the flickering is surprisingly noticeable. For this reason, most computers, including MSX, normally use non-interlaced screen display.

4.7.3 Interlaced screen on MSX2

I wrote "usually" non-interlaced screens are used on many computers because MSX2 and later can also use interlaced screens.

The first reason why the flickering interlaced method was adopted was to display a 424-dot vertical screen on a home television monitor (i.e. what are known as home televisions, not just for computers). In 1985, when the MSX2 was developed, multi-scan monitors were not common, and monitors with a horizontal scan frequency of 24 kHz were expensive, costing over 100,000 yen. As a result, the mainstream monitors to be used with the MSX2 were ordinary home televisions or low-resolution monitors with a horizontal scan frequency of 15.75 kHz. For example, with the MSX2+,

```
CALL KANJI2
```

By executing this command and switching the screen to interlaced mode, it became possible to display 24 lines of kanji vertically. However, as I wrote before, interlaced screen display flickers a lot and can strain your eyes, so try to take breaks from time to time.

The second purpose was more ambitious than the first, and was to enable "superimposition" and "video digitization" by connecting the MSX2 to a television screen. The MSX2 screen, which uses a horizontal scanning frequency of 15.75 kHz and an interlaced format, is ideal for superimposing a computer screen onto a television screen, and for video digitization, which involves importing television broadcasts or video camera images into a computer. Panasonic's FS-5500 and other computers with this function built in from the start, Victor's HC-95, which can be used with an optional board, was released a long time ago, but is still in use today for importing and processing image data.

The third advantage of the interlaced method is that it produces better photographs. In non-interlaced screen photographs, the gaps between the scan lines are visible, and when printed, a striped pattern called "moire" tends to appear. However, with interlaced screens, there are no gaps between the scan lines, so there is no need to worry about moire.

On MSX2 and later machines, you can switch the screen to interlaced mode as follows. However, Kanji mode can only be specified on MSX2+ and later.

For MSX2 machines	`SCREEN ,,,,,1`
For MSX2+ machines and later	`CALL KANJI3`

4.7.4 Exploring the principles of scan line interruption

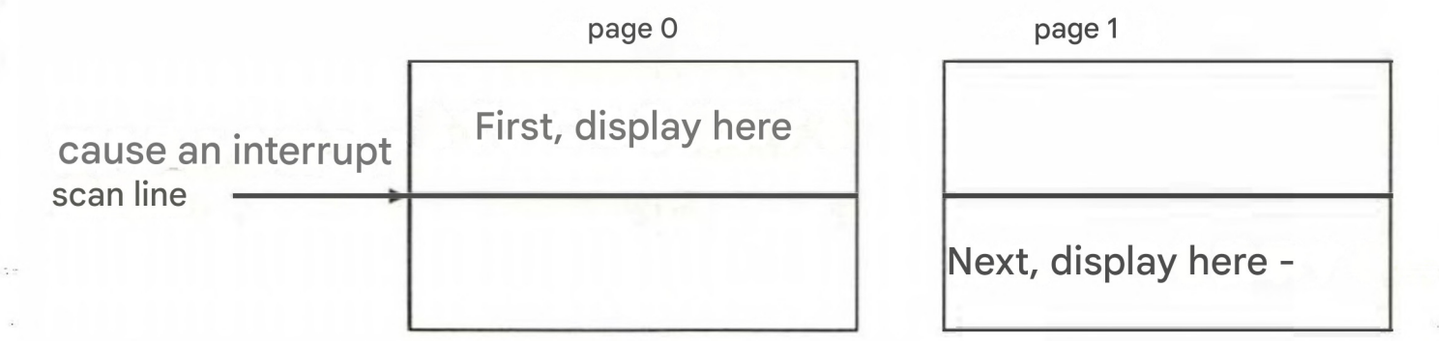
An "interrupt" is something that changes the flow of a program due to exceptional conditions. For example, the "ON STRIG GOSUB" command, which changes the flow of a BASIC program when a joystick button is pressed, is a type of interrupt handling command.

The principle of "scan line interrupt" is the same, and when the display of the specified scan line is finished, an interrupt is processed. However, scan line interrupts need to be processed quickly, so programming in BASIC would not be enough. Interrupt processing using a machine language program is required.

Let me briefly explain the principles of interrupt processing. First, the VDP, which controls the screen display, can send an interrupt signal to the CPU when certain conditions are met. There are three types of conditions: "light pen," "vertical retrace," and "scan line," but the light pen interrupt is not used on the MSX2.

First, the "vertical blanking interrupt" is an interrupt that occurs when the bottom edge of the screen has finished displaying and the next screen is being prepared to be displayed, and it occurs every 1/60th of a second. This is the true nature of MSX's "timer interrupt," which is used to adjust the timing of music played during games. The issue at hand today is the scan line interrupt, which occurs when the display of a specific scan line has finished. This interrupt also occurs every 1/60th of a second.

Figure 4.10: The principle of scan line interruption



By combining the vertical blanking interrupt with the scan line interrupt, it is possible to generate interrupts in two places: at the top of the screen and on any scan line. This means that the screen can be divided into two parts: the upper part from the vertical blanking interrupt to the scan line interrupt, and the lower part from the scan line interrupt to the vertical blanking interrupt.

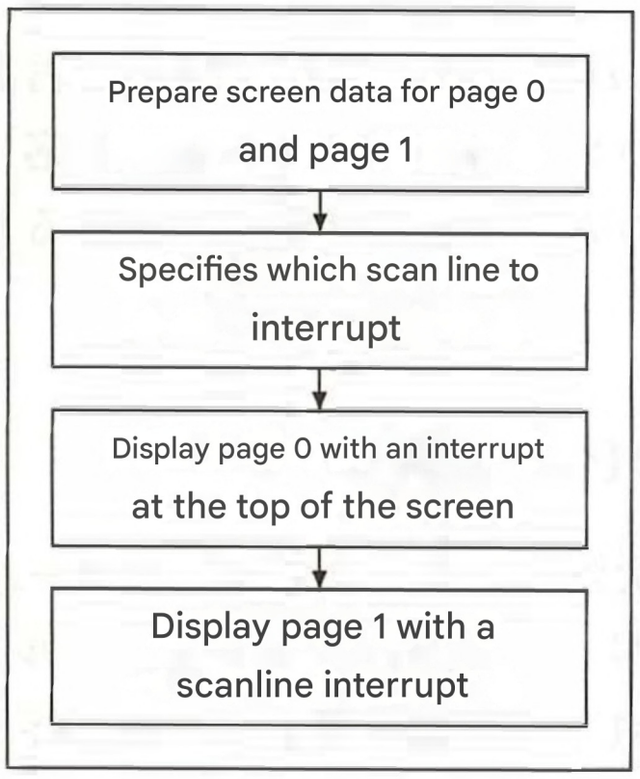
The MSX2 can also have multiple screens called "pages." With 128KB of video RAM, the MSX2 can have four screens on SCREEN 5 or 6, and two screens on SCREEN 7 and 8. By switching between these with the BASIC "SET PAGE" command, multiple screens can be displayed instantly.

This function is used in the scan line interrupt, which is used in games and other software. By creating an interrupt processing program in machine language and switching pages with the scan line interrupt, different pages are displayed on the top and bottom of the screen. Furthermore, by combining this with the VDP scrolling function, it is also possible to scroll only one side of the screen.

4.7.5 Introducing an example of scan line interrupts

To be honest, when I first saw the VDP specifications, I wondered, "What is the use of scan line interrupt?" Although the existence of the scan line interrupt function was introduced in the "MSX2 Technical Handbook" and other documents,

Figure 4.11: Scanline interrupt procedure



The screens prepared in advance on pages 0 and 1 are switched and displayed by interrupting at the specified scan line.

There were no specific program examples or application examples listed. I think it was not until the game "Xanac EX" developed by Compile and published by Pony Canyon that it was actually applied to video games and made everyone gasp in excitement.

One of the features that was expanded when MSX became MSX2 was "hardware vertical scrolling." This was a very useful feature for making shooting games, but since the entire screen scrolled as it was, it was not possible to fix the score display, which is essential for games, on the screen. However, in "Xanac EX,"

The screen scrolled vertically at high speed while the score was displayed at a fixed position at the top of the screen. At the time, the M Magazine editorial department was abuzz with what kind of technique this was, but the flickering at the boundary between the score and the scrolling area revealed that it was a scan line interrupt. Once this was realized, it was like a Columbus egg, and from then on, shooters using scan line interrupts were developed one after another. When you use the pause key to stop a game using a scan line interrupt, only one of the two screens will be displayed. Let's try it out for ourselves.

Furthermore, on the MSX2+, scan line interrupts are combined with "hardware horizontal scrolling" to allow horizontal scrolling of only a portion of the screen. In "F-1 Spirit 3D Special" released by Konami, the cockpit of an F1 machine is displayed at the bottom of the screen by using scan line interrupts while only the game screen is scrolled horizontally.

4.7.6 Let's finally get excited about the practical part!

Let me introduce an example program that actually uses scan line interrupts. I'll explain the details later, but this program was written in machine language. However, it can be called from BASIC language, so it should be useful for homemade games and the like. I'll also publish the source listing, so if you know assembler, I hope you'll try your best to analyze it.

4.7.7 VDP register used for scan line interrupt

The procedure for generating a scan-line interrupt is as follows. First, write the number of the scan line for which you want to generate an interrupt into control register 19, and change bit 4 of control register 0 to 1. Then, when the display of the specified scan line has finished, the VDP will issue an interrupt to the CPU. Also, if bit 0 of status register 1 is 1 when an interrupt is issued, you know that the cause of the interrupt is a scan-line interrupt. All of this is summarized in Figures 4.12 and 4.13.

Figure 4.12: VDP Registers for Generating Scan Line Interrupts

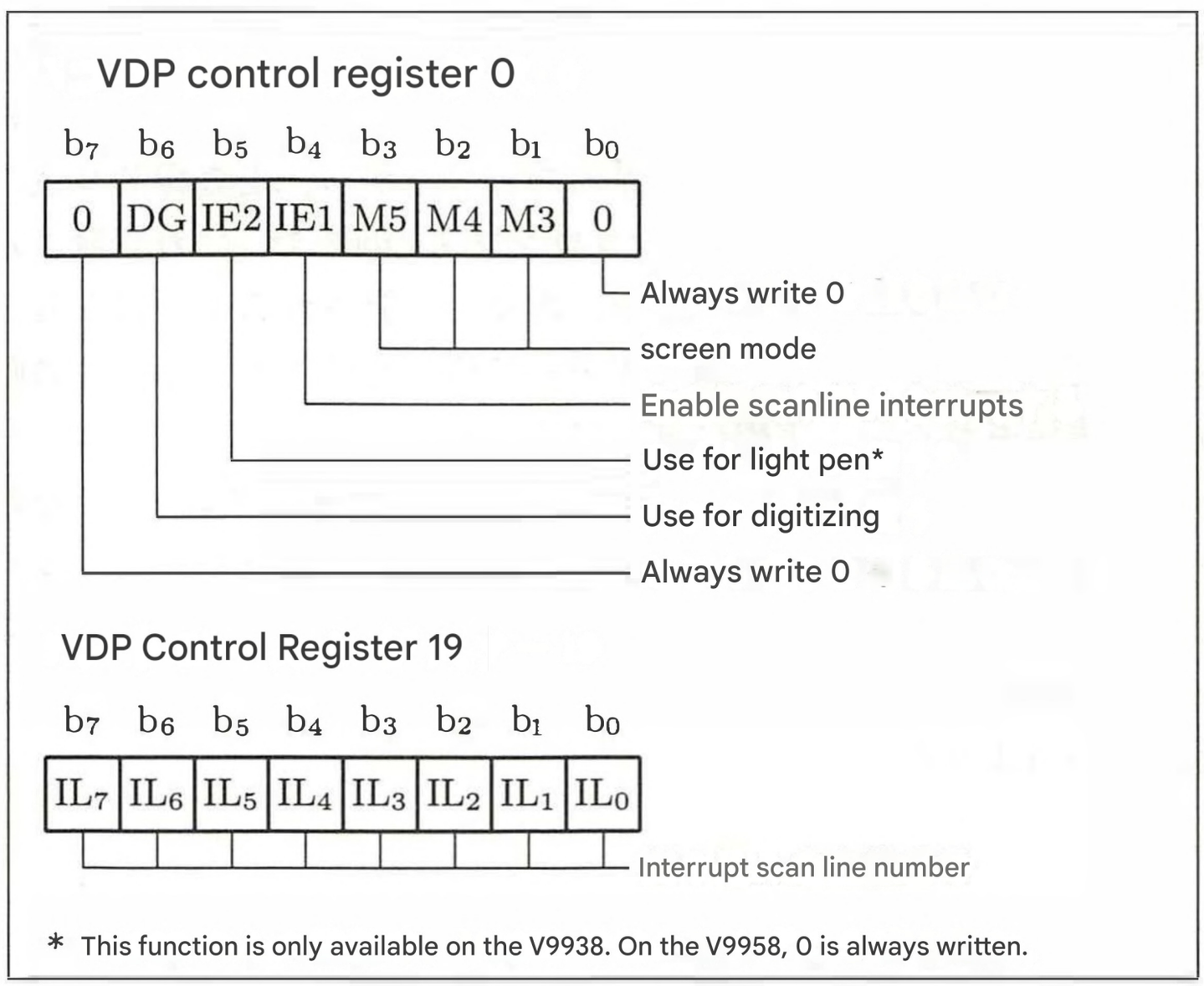
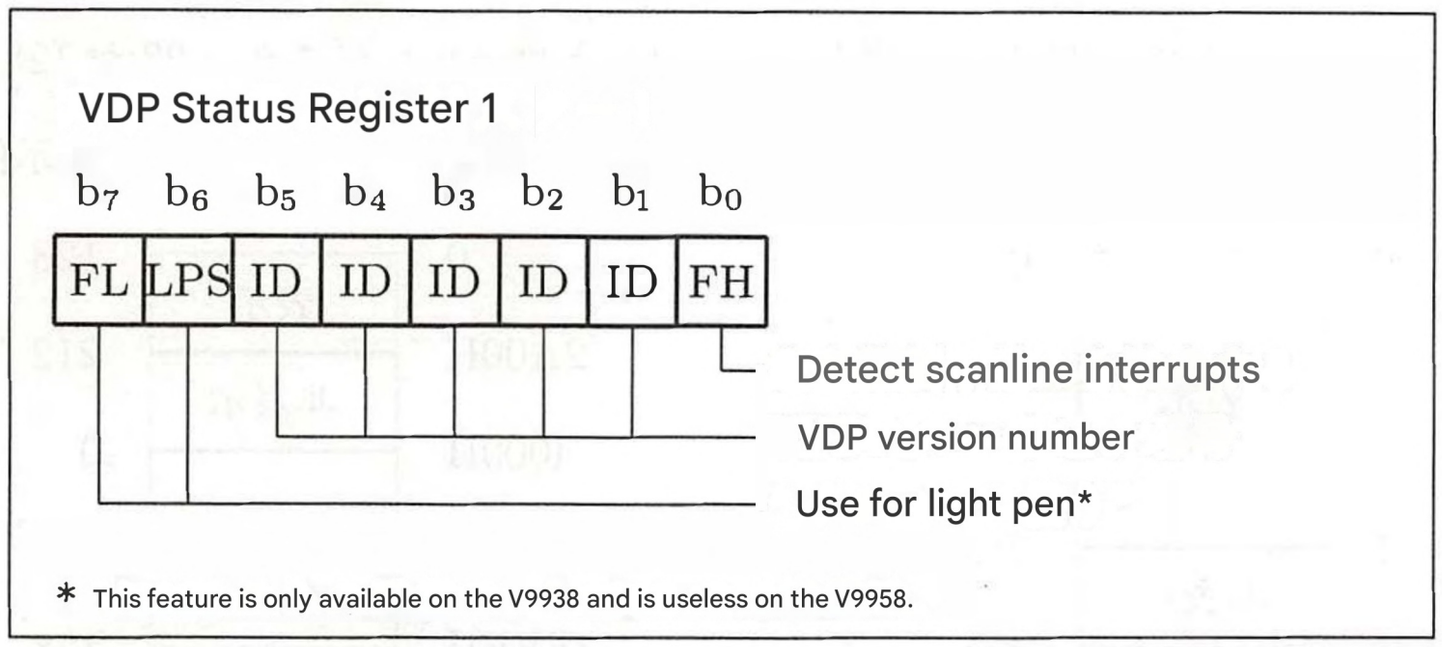


Figure 4.13: VDP Register to Detect Scan Line Interrupt



Well, that's all about the registers that are directly related to scan line interrupts. Now let's use other registers to control screen switching and hardware vertical scrolling.

With SCREEN 5 or SCREEN 6, you can change the display page (the page shown on the screen) by writing a page number to bits 6 and 5 of control register 2, just like the BASIC "SET PAGE" command (see Figure 4.14). Write a 0 to bit 7 and a 1 to bits 4 through 0. In the same way, with SCREEN 7 and SCREEN 8, you can change pages by specifying the page in bit 5 and always writing a 0 to bit 6.

Figure 4.14: VDP registers that control screen switching

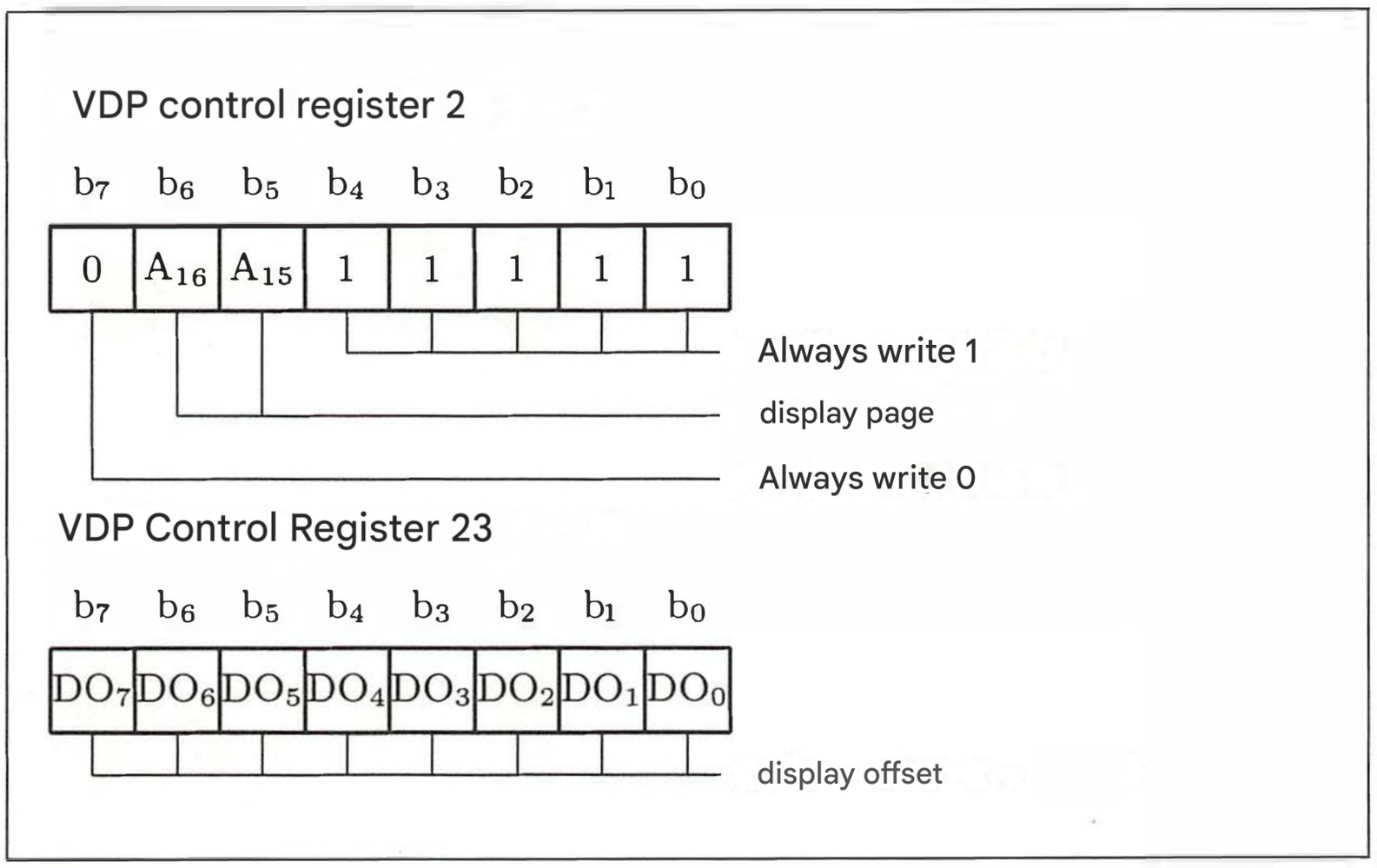
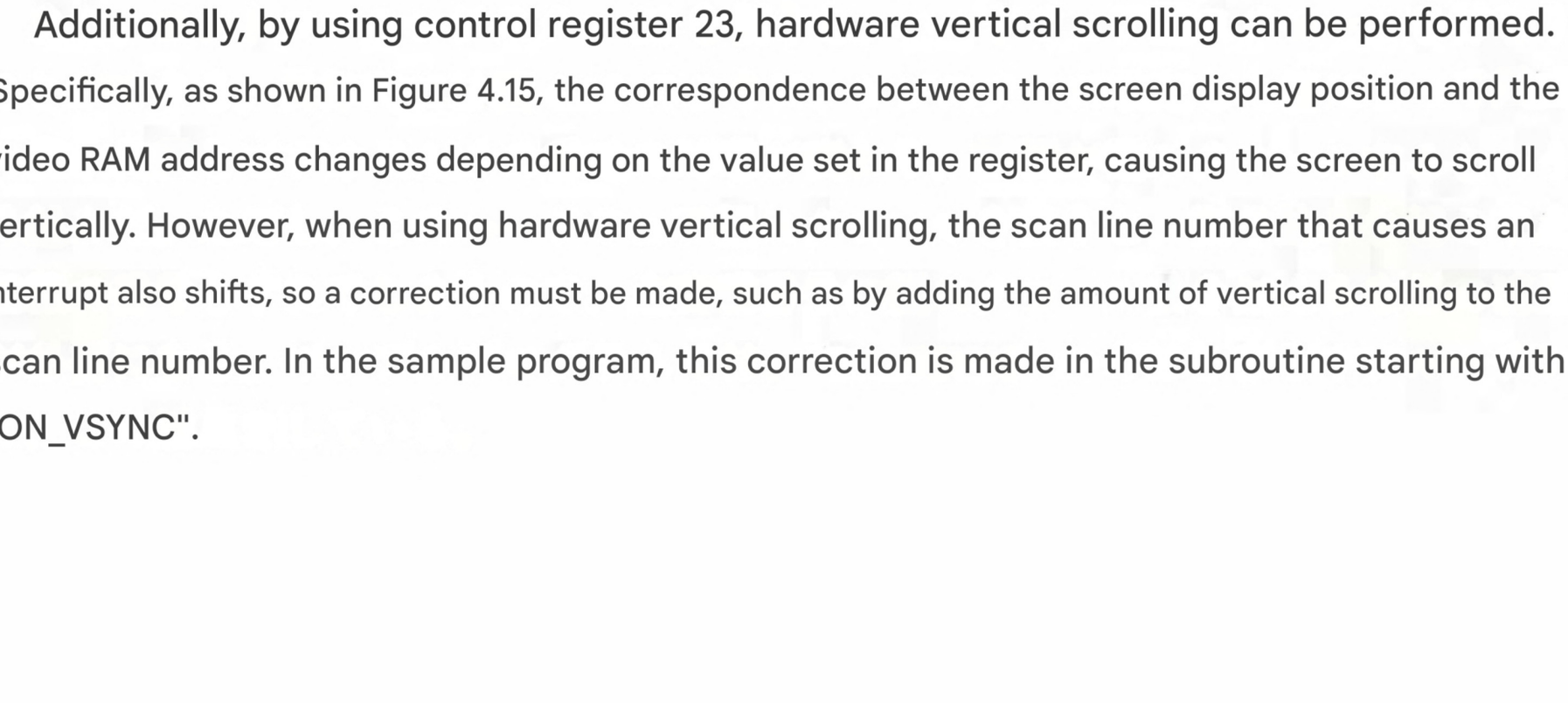


Figure 4.15: How hardware vertical scrolling works



Additionally, by using control register 23, hardware vertical scrolling can be performed. Specifically, as shown in Figure 4.15, the correspondence between the screen display position and the video RAM address changes depending on the value set in the register, causing the screen to scroll vertically. However, when using hardware vertical scrolling, the scan line number that causes an interrupt also shifts, so a correction must be made, such as by adding the amount of vertical scrolling to the scan line number. In the sample program, this correction is made in the subroutine starting with "ON_VSYNC".

4.7.8 How to assemble and how the BASIC part works

The sample program for generating a scan line interrupt is made up of a part written in assembler and a part written in BASIC. I will explain how the assembler part works next, but first I will introduce how to assemble it and how the BASIC part works.

The assembler program (source listing) itself is shown in Listing 4.9 on a later page. Enter this into an MSX-DOS screen editor and save it with the file name "ON-SCAN.Z80". Next, use the programs "M80.COM" and "L80.COM" included in "MSX-DOS TOOLS" to assemble and link as follows. Once you have a machine language file called "ONSCAN.BIN", you are done. The "DEL ONSCAN.BIN" part is used to delete the old file when you reassemble. It is not necessary the first time.

```
M80 ,=ONSCAN.Z80/R/Z
L80 ONSCAN,ONSCAN/N/E
DEL ONSCAN.BIN
REN ONSCAN.COM ONSCAN.BIN
```

Next, we will explain the operation of the BASIC program (see List 4.8) that controls the machine language file.

First, the section from lines 190 to 200 is for initializing video RAM page 0. To scroll the screen vertically, the entire page 0 needs to be initialized, but the “CLS” command can only initialize the part that is displayed on the screen.

So we use the “COPY” command to copy the contents of (0,0)-(255,127) on the screen to (0,128)-(255,255), initializing the entire page 0. Also, lines 210, 240, and 250 are for displaying test patterns.

To call a machine language program from a BASIC program, use the “USR function.” Line 260 specifies the scan line that will cause an interrupt, and displays page 0 on the top of the screen and page 1 on the bottom of the screen.

USR(256 + scanline number)

By changing the scan line number according to this format, the location where the interrupt occurs will change.

In addition, the page 0 section displayed at the top of the screen scrolls vertically by 290 lines.

USR (512 + scan line number)

Specify the value in the format shown below. Also, although it is not used this time,

USR(768+scanline number)

If you specify this, you can vertically scroll the part of page 1 displayed at the bottom of the screen. To stop the scan line interrupt,

USR(0)

All you have to do is execute.

In this program, the parameters of the USR function are used to simplify the assembler part. The meter is limited to integers. So,

USR(868.0)

Or something

USR(100+A!)

Real parameters such as

list 4.8 (ONSCAN.BAS)

```
100 '
110 ' onscan1.bas : test interrupt on scan
120 ' by nao-i on 24. Sep. 1989
130 '
140 CLEAR 300,&HAFFF
150 DEFINT A-Z
160 OPEN "GRP:" FOR OUTPUT AS #1
170 BLOAD "onscan.bin"
180 SCREEN 5
190 SET PAGE 0,0: COLOR 15, 6,1: CLS
200 COPY (0,0)-(255,127) TO (0,128)
210 CIRCLE (128,106),80: PAINT (128,106)
220 SET PAGE 1,1: COLOR 15, 1,1: CLS
230 DEFUSR0=&HB000
240 PSET (8,192),0,PRESET
250 PRINT #1,"This area will not scroll."
260 JK=USR(256+100)
270 FOR J=1 TO 5
280   FOR I=0 TO 255
290     JK=USR(512+I)
300   NEXT I
310 NEXT J
320 JK=USR(0): COLOR 15,4,4: SCREEN 0
330 END
```

4.7.9 This is the operating principle of the assembler part.

Now then, let's finally explain the operating principles of the main program in List 4.9, which actually handles the raster interrupt.

First, the three lines starting from "ASEG" are commands to use M80.COM and L80.COM to create special objects such as BASIC subroutines. M80.COM and L80.COM are usually used in pairs to create machine language files from programs written in assembler. However, the file extension created in this case is "COM", which means it becomes a machine language file that can be executed on MSX-DOS. So, if you want to create a file that can be handled by BASIC, as in this case, you need commands from ASEG.

Also, machine language programs that can be loaded in BASIC have a 7-byte header at the beginning,

```
FEH
Load start address
Load end address
execution start address
```

The following four lines of ASEG specify these data.

Now, the section after "AD_LOAD:" is the main body of the program. The first thing it does is process the parameters of the USR function. If the parameter is an integer, the content of the A register will be 2, so this is confirmed. Also, the value of the input parameter is recorded at addresses HL+2 and HL+3. At this time, if the upper byte of the parameter is 0 then cleanup is performed, if it is 1 then a scan line interrupt is set, if it is 2 then vertical scrolling of page 0 is performed, and if it is 3 then vertical scrolling of page 1 is performed.

When an interrupt actually occurs, the address FD9AH (HOOKDT) is called. In the next 5 bytes,

```
RST     30H
DB      slot number
DW      address
RET
```

If you write this, the specified program will be called when an interrupt occurs. A place like this that is called under certain conditions is called a "hook." Here, we are preparing to call "ON_H.KEYI:".

Address FD9FH is the part that calls the timer interrupt (ON_H.TIMI:), that is, the vertical blanking interrupt. Here, the value of VDP status register 0 when the interrupt occurs is stored in the A register, so the AF register should not be rewritten.

If you absolutely must rewrite the hook, save the original value of the hook as in the sample program, and jump to the saved hook when the interrupt processing is complete.

The program that is called when an interrupt occurs and operates the VDP is organized into subroutines beginning with "ON_VSYNC:" and "ON_SCAN:". By rewriting these, the scan line interrupt can be used for a different purpose.

The subroutine "_VDPSTA" reads the VDP status register. The subroutine "WRTVDP" writes the specified value to the VDP control register and saves it in the appropriate location (see Table 4.9). As mentioned before, the MSX2 and 2+ ROMs have BIOS with the same functions as these subroutines, so you can usually just use the BIOS without creating your own subroutines. However, these BIOSes are stored in subROMs or call subROMs during processing, which has the drawback of taking some time. For this reason, it is not convenient for interrupt processing like this one, so we chose not to use the BIOS.

This has nothing to do with BASIC machine language subroutines, but DOS programs must place interrupt handlers at addresses above 4000H, because problems occur on certain MSX slot configurations below that.

list 4.9 (ONSCAN.Z80)

```
;
;       onscan.z80 : test program for interrrupt on scan
;       by nao-i on 26. Sep. 1989
;
;       called as USR function from BASIC
;       USR(&H00xx)     restore registers and hooks
;       USR(&H01xx)     set interrupt line
;       USR(&H02xx)     set display offset line of page 0
;       USR(&H03xx)     set display offset line of page 1
;
        .Z80
START   EQU     0B000H  ; address to load and execute
USE_SUB         EQU     0
USE_WRTVDP      EQU     0
;
        IF      USE_WRTVDP
WRTVDP  EQU     0047H
        ENDIF
EXTROM  EQU     015FH
VDPSTA  EQU     0131H
SETPAG  EQU     013DH
;
RAMAD2  EQU     0F343H          ;  lot of  AM in page 2
RAMAD3  EQU     0F344H          ;  lot of  AM in page 3
RGOSAV  EQU     0F3DFH
DPPAGE  EQU     0FAF5H
ACPAGE  EQU     0FAF6H            s       R
H.KEYI  EQU     0FD9AH
H.TIMI  EQU     0FD9FH          s       R
RG8SAV  EQU     0FFE7H
;
        ASEG
        ORG     100H            ; to make .COM file
        .PHASE  START-7
;
        DB      0FEH            ; header to BLOAD
        DW      AD_LOAD         ; address to load
        DW      AD_NEXT-1       ; address of end of file
        DW      DO_NOTHING      ; address to execute
;
AD_LOAD:
        PUSH    AF
        PUSH    HL
        PUSH    DE
        PUSH    BC
        CP      2
        JR      NZ,RESET_SCAN   ; parameter is not integer
        INC     HL
        INC     HL
        LD      E,(HL)
        INC     HL
        LD      D,(HL)          ; DE = parameter of USR()
```

```
        LD      A,D
        OR      A
        JR      Z,RESET_SCAN
        DEC     A
        JR      Z,SET_SCAN
        DEC     A
        JR      Z,SET_DO
        DEC     A
        JR      Z,SET_D1
        JR      RESET_SCAN
;
SET_SCAN:
        LD      A,E
        LD      (ILSAV),A
        CALL    SET_ILREG           ; set interrupt line
        LD      A,(HOOKED)
        OR      A
        JR      NZ,RET_BASIC        ; hook is already set
;
        LD      HL,H.KEYI
        LD      DE,HOOKSA
        LD      BC,10
        LDIR                        ; save hooks
        LD      HL,HOOKDT
        LD      DE,H.KEYI
        LD      BC,10
        DI
        LDIR                        ; set hooks
        LD      A,(RAMAD2)
        LD      (H.KEYI+1),A
        LD      (H.TIMI+1),A
        LD      A,(RGOSAV)
        OR      00010000B
        LD      B,A
        LD      C,0
        CALL    WRTVDP              ; interrupt on
        LD      A,1
        LD      (HOOKED),A
        JR      RET_BASIC
;
RESET_SCAN:
        CALL    RESET_SCAN_SUB
        JR      RET_BASIC
;
SET_DO:                             ; set display offset of page 0
        LD      A,E
        LD      (DOVAL),A
        JR      RET_BASIC
SET_D1:                             ; set display offset of page 1
        LD      A,E
        LD      (D1VAL),A
;
RET_BASIC:
        EI
        POP     BC
```

```
        POP     DE
        POP     HL
        POP     AF
DO_NOTHING:
        RET
;
RESET_SCAN_SUB:
        DI
        LD      A,(RGOSAV)
        AND     11101111B
        LD      B,A
        LD      C,0
        CALL    WRTVDP          ; inturrupt off
        LD      BC,23           ; write 0 into reg#23
        CALL    WRTVDP          ; restore display offset
        XOR     A
        LD      (DPPAGE),A
        LD      BC,1F02H        ; write 1FH into reg#2
        CALL    WRTVDP          ; set page 0
        LD      A,(HOOKED)
        OR      A
        RET     Z
        LD      HL,HOOKSA
        LD      DE,H.KEYI
        LD      BC,10
        LDIR                    ; restore hooks
        XOR     A
        LD      (HOOKED),A
        RET
;
SET_ILREG:
        LD      A,(ILSAV)
        LD      HL,DOVAL
        ADD     A,(HL)          ; interrupt line = (ILSAV) + (DOVAL)
        LD      B,A
        LD      C,19
        JP      WRTVDP          ; write interrupt line # into reg#19
;
ON_H.KEYI:                      ; called from H.KEYI
        LD      A,1
        CALL    _VDPSTA         ; read status reg#1
        AND     1
        CALL    NZ,ON_SCAN
        JR      HOOKSA
;
ON_H.TIMI:                      ; called from H.TIMI
        PUSH    AF
        CALL    ON_VSYNC
        POP     AF              ; do not change AF in H.TIMI
        EI
        JR      HOOKSA+5
;
HOOKDT:
        RST     30H
        DB      0
```

```
        DW      ON_H.KEYI
        RET
        RST     30H
        DB      0
        DW      ON_H.TIMI
        RET
;
;       data area
;
HOOKED: DB      0       ; non-zero if hooks have been set
HOOKSA: DS      10      ; save area for hooks
DOVAL:  DB      0       ; display offset of page 0
D1VAL:  DB      0       ; display offset of page 1
ILSAV:  DB      0       ; interrrupt line
;
;       _VDPSTA : read a VDP status register
;       Entry   A       VDP status register #
;       Return  A       value of the status register
;       Modify  AF, BC
;       Note    compatible with ROM-BIOS
;               DI when return
;
_VDPSTA:
        IF      USE_SUB
        LD      IX,VDPSTA
        JP      EXTROM
        ELSE
        DI
        AND     00001111B
        LD      B,A
        LD      A,15
        LD      C,A
        CALL    WRTVDP
        LD      BC,(6)
        INC     C
        IN      A,(C)
        PUSH    AF
        LD      BC,15
        CALL    WRTVDP
        POP     AF
        RET
        ENDIF
;
;       WRTVDP : write a byte into VDP regis er
;       Entry   B       datum to write
;               C       VDP register #
;       Return  none
;       Modify  AF, BC                                  t
;       Note    compatible with ROM-BIOS
;               DI when return
;
        IFE     USE_WRTVDP
WRTVDP:
        PUSH    HL
        PUSH    DE
```

```
        LD      D,B             ; =datum
        LD      A,C             ; =register #
        LD      HL,RG0SAV
        CP      8
        DI
        JR      NC,SAVEREG
        LD      HL,RG8SAV-8
        CP      24
        JR      NC,NOSAVE
SAVEREG:
        XOR     A
        LD      B,A             ; BC=register #
        ADD     HL,BC           ; HL=RG?SAV
        LD      (HL),D          ; save datum
NOSAVE:
        LD      A,C             ; =register #
        LD      BC,(7)
        INC     C
        OUT     (C),D
        AND     00111111B
        OR      10000000B
        OUT     (C),A
        POP     DE
        POP     HL
        RET
        ENDIF                   ; IFE USE_WRTVDP
;
;       please modify following subroutines as you need
;
ON VSYNC:
        XOR     A
        LD      (DPPAGE),A
        LD      BC,1F02H        ; write 1FH into reg#2
        CALL    WRTVDP          ; set page 0
        LD      A,(DOVAL)
        LD      B,A
        LD      C,23
        CALL    WRTVDP          ; set display offset
        JP      SET_ILREG       ; set interrupt line
;
ON_SCAN:
        LD      A,1
        LD      (DPPAGE),A
        LD      BC,3F02H        ; write 3FH into reg#2
        CALL    WRTVDP          ; set page 1
        LD      A,(D1VAL)
        LD      B,A
        LD      C,23
        JP      WRTVDP          ; set display offset
;
AD_NEXT EQU     $               ; end of program + 1
        .DEPHASE
;
        END
```

4.7.10 This is the machine routine for the scan line interrupt.

Finally, for those who find it troublesome to type out the source listing for the scan line interrupt program, or for those who do not have an assembler, here is a BASIC program that automatically creates a machine language file. Type in the following program and run it to automatically create a file called "ONSCAN.BIN".

list 4.10 (MKONSCAN.BAS)

```
10 CLEAR 100,&HCFFF:DIMD(15)
20 PRINT"Making onscan.bin":AD=&HB000:C=0:L=0
30 FOR I=0TO15:READ A$:IF A$="*" GOTO100
40 A=VAL("&h"+A$):C=(C+A) AND 255:D(I)=(D(I)+A) AND 255
45 POKE AD,A:AD=AD+1:NEXT
50 READA$:A=VAL("&h"+A$):L=L+1
55 IF C<>A THEN PRINT "Error in line ";990+10*L:END
60 GOTO 30
100 '
110 PRINT"Saving"
120 BSAVE"onscan.bin",&HB000,&HB14D
130 PRINT"Done.":END
1000 DATA F5,E5,D5,C5,FE,02,20,53,23,23,5E,23,56,7A,BT,28, 5D
1010 DATA 4A,3D,28,08,3D,28,49,3D,28,4C,18,3F,7B,32,D9,B0, 00
1020 DATA CD,A1,B0,3A,CC,B0,B7,20,41,21,9A,FD,11,CD,B0,01, 33
1030 DATA 0A,00,ED,B0,21,C2,B0,11,9A,FD,01,0A,00,F3,ED,B0, B0
1040 DATA 3A,43,F3,32,9B,FD,32,A0,FD,3A,DF,F3,F6,10,47,0E, 20
1050 DATA 00,CD,F4,B0,3E,01,32,CC,B0,18,0F,CD,70,B0,18,0A, B4
1060 D TA 7B,32,D7,B0,18,04,7B,32,D8,B0,FB,C1,D1,E1,F1,C9, 61
1070 D ATA F3,3A,DF,F3,E6,EF,47,0E,00,CD,F4,B0,01,17,00,CD, E0
1080 DATA F4,B0,AF,32,F5,FA,01,02,1F,CD,F4,B0,3A,CC,B0, 7, 54
1090 DATA C8,21,CD,B0,11,9A,FD,01,0A,00,ED,B0,AF,32,CC, B0, 67
1100 DATA C9,3A,D9,B0,21,D7,B0,86,47,0E,13,C3,F4,B0,3E,01, 2F
1110 DATA CD,DA,B0,E6,01,C4,32,B1,18,13,F5,CD,1C,B1,F1,FB, BA
1120 DATA 18,10,F7,00,AE,B0,C9,F7,00,BA,B0,C9,00,00,00,00, 2A
1130 DATA 00,00,00,00,00,00,00,00,00,00,F3,E6,0F,47,3E,0F, A6
1140 DATA 4F,CD,F4,B0,ED,4B,06,00,0C,ED,78,F5,01,0F,00,CD, E7
1150 DATA F4,B0,F1,C9,E5,D5,50,79,21,DF,F3,FE,08,F3,30,07, EB
1160 DATA 21,DF,FF,FE,18,30,04,AF,47,09,72,79,ED,4B,07,00, 5D
1170 DATA 0C,ED,51,E6,3F,F6,80,ED,79,D1,E1,C9,AF,32,F5,FA, F3
1180 DATA 01,02,1F,CD,F4,B0,3A,D7,B0,47,0E,17,CD,F4,B0,C3, E7
1190 DATA A1,B0,3E,0 ,32,F5,FA,01,02,3F,CD,F4,B0,3A,D8,B0, 0D
1200 DATA 47,0E,17,C 1,F4,B0,*
3
```

第5章 MSX-MUSIC



This chapter is a re-edited version of the "MSX2+ Technical Exploration Team" articles published in the July 1990 to October 1990 issues of MSX Magazine.

5.1 What is an FM sound source?

The FM sound source with the specifications defined under the name MSX-MUSIC. We know that it makes game sound effects more powerful, but how does it work? This page will try to unravel the mystery.

5.1.1 The history of electronic musical instruments leading up to FM synthesis

Before explaining FM synthesis, let's look back at the history of electronic musical instruments.

Dr. Bog Moog combined an oscillator that can control the pitch with voltage with a filter that can control the tone with voltage to create an instrument called the "Moog synthesizer." The first record using this was released in 1968, and by the 1970s many musicians had begun using synthesizers. However, this synthesizer differed from the "digital synthesizers" that have become mainstream recently, as it was made with a combination of analog circuits such as transistors. In contrast to digital synthesizers, it is also called an "analog synthesizer."

However, this analog synthesizer had several drawbacks: it was sensitive to temperature changes, expensive, and prone to noise. I also bought an IC in Akihabara in the 1970s and built my own synthesizer, but I remember how difficult it was to tune.

To overcome these shortcomings, electronic musical instruments using digital circuits were developed. The simplest digital sound source is the "Programmable Sound Generator," or "PSG" for short. This is an LSI that converts the output of about four digital oscillators into an audio signal using a digital-to-analog (D/A) converter and outputs it. It is inexpensive and easy to use, so it is built into many personal computers, including the MSX.

One way to create more complex sounds than PSG is to use a "sampling sound source." This involves receiving the sound of another instrument through a microphone, converting it to a digital signal through an A/D (analog-to-digital) converter, storing it in memory, and then converting it back to an analog signal through a D/A converter for playback. An instrument that applies this technique is called a "sampling synthesizer." It offers a high degree of freedom in creating sounds, but the downside is that the hardware is expensive, requiring a large amount of memory and other factors.

Now, FM sound sources are attracting attention as a sound source that combines the low cost of PSG with the flexibility of sampling sound sources.

Table 5.1: Comparing the performance of electronic musical instruments

	analog synth	PSG	sampling synth	FM sound source
hardware	complicated	LSI	LSI + large capacity memory	LSI
stability	Sensitive to temperature changes	stable	stable	stable
tone	Variety	poor	All-purpose	Variety
amount of data		small	enormous	small
price	expensive	cheap	expensive	average

FM stands for "frequency modulation," the same as FM broadcasting and FM signals in modems.

As shown in Figure 5.1, the output of one digital oscillator modulates the frequency of another digital oscillator, creating a more complex sound than a PSG. A single digital oscillator is also called an "operator," and an FM sound source that includes two oscillators, as shown in Figure 5.1, is called a "two-operator FM sound source." Incidentally, the official name of "MSX-MUSIC" is "OPLL YM2413," and it is an LSI that contains nine sets of two-operator FM sound sources.

Figure 5.1: Exploring the structure of four types of electronic musical instruments

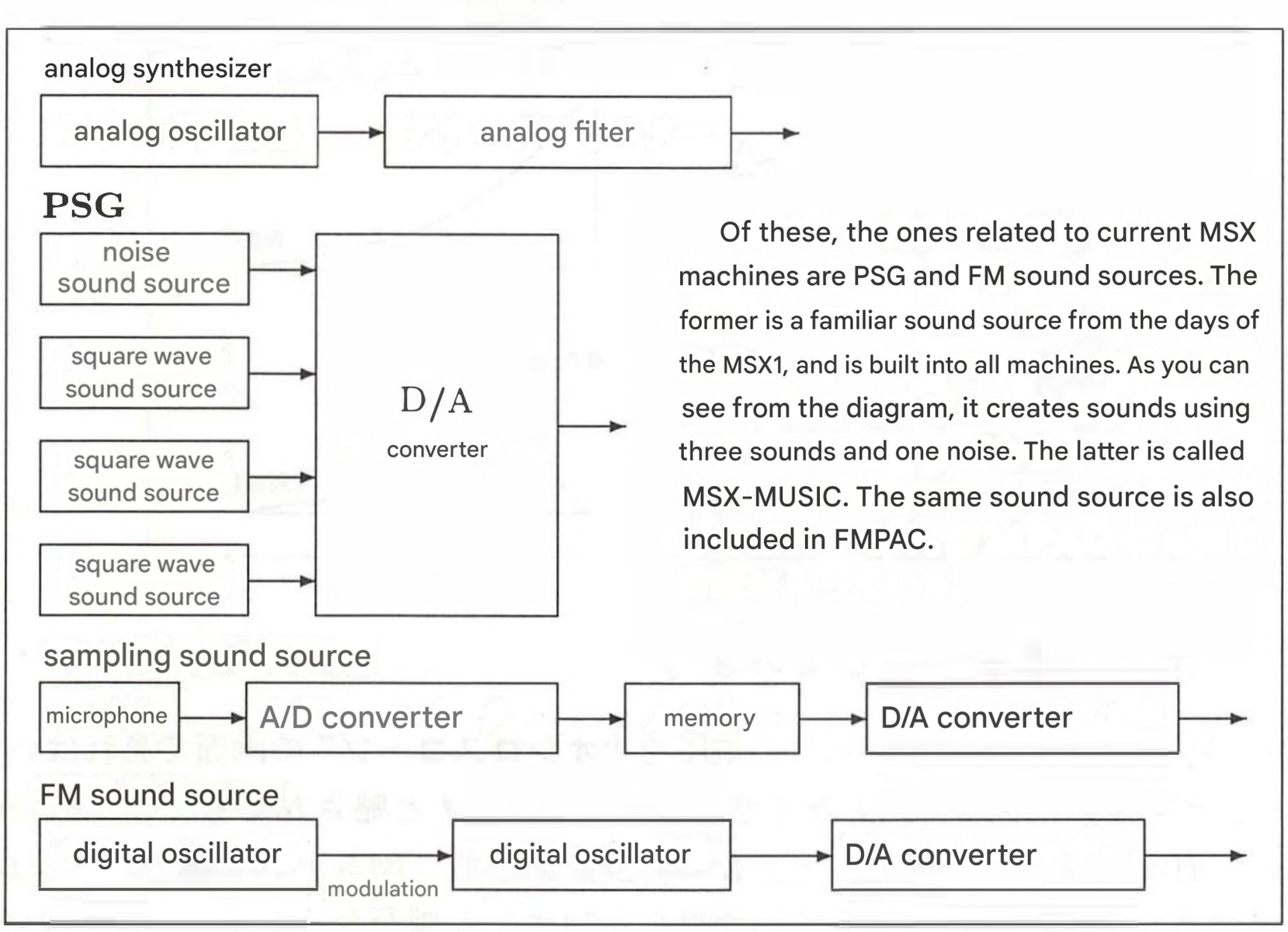
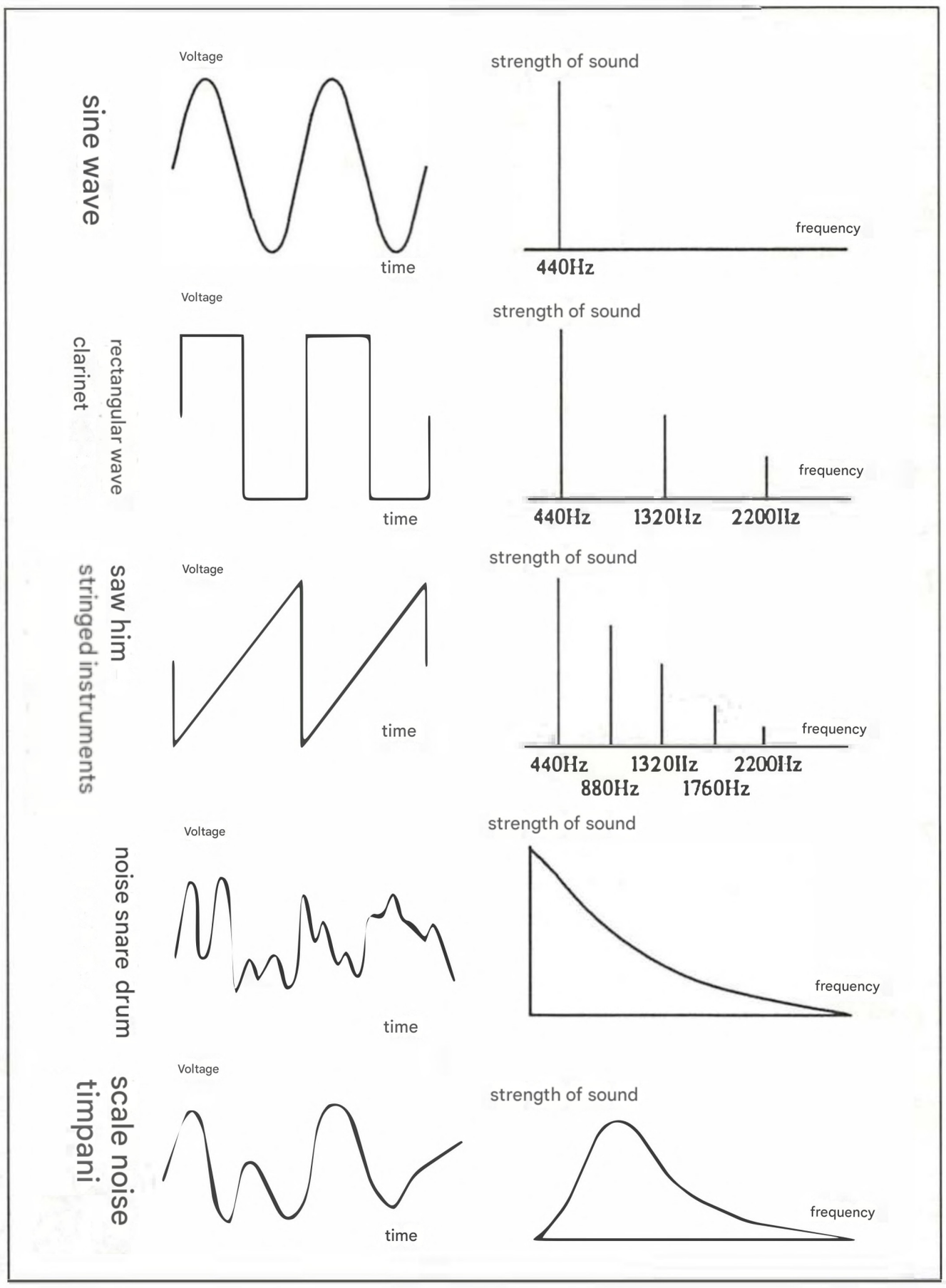


Figure 5.2: Analyzing the basic sound



5.1.2 Let's analyze the sound of musical instruments

To "see" sound, all you need to do is look at the voltage of the sound signal on the screen of an oscilloscope. Furthermore, you can determine the characteristics of a sound by breaking it down into its frequency components using a device called a "spectrum analyzer" (commonly abbreviated as "specta"). The left side of Figure 5.2 shows an exaggerated view of the waveform characteristics of a musical instrument sound as viewed on an oscilloscope, and the right side shows the frequency components measured by a spectrum analyzer.

The most basic sound is called a "sine wave" whose waveform is expressed by the trigonometric function sine. This contains only one frequency. The next is a "square wave" whose waveform is square, which contains the fundamental frequency and its odd multiples, such as 440Hz, 1320Hz, 2200Hz, etc. In real musical instruments, the sound of a clarinet is close to this square wave.

The next most basic type is the "sawtooth wave." This contains sounds at the fundamental frequency and its multiples, and is similar to the characteristics of stringed instruments. Analog synthesizers process sawtooth waves to create sounds similar to real instruments. Now, the sound of percussion instruments, especially snare drums, is very different from other instruments. It has no regularity and is more like "noise." When viewed with a spectrum analyzer, it contains sounds over a wide range of frequencies.

The sound of the timpani is somewhere between that of a stringed instrument and a percussion instrument, and contains the fundamental frequency and sounds close to it. It is also called "scale noise". By processing this scale noise, it is possible to synthesize sounds such as wind, waves, and whistling. The whistle played by a teacher in a physical education class, or the sound of a wind instrument played by an amateur, are also scale noise.

Instead of the sounds of these instruments, i.e., the waveforms shown in Figure 5.2, FM synthesis uses modulation to distort sine waves to create complex waveforms that contain the fundamental frequency and its multiples. This is a difficult technology, and the only way to imitate the sounds of instruments is to adjust the values through trial and error. Therefore, FM synthesis has a built-in program for synthesizing several instrument sounds, and it is common to select and use these sounds.

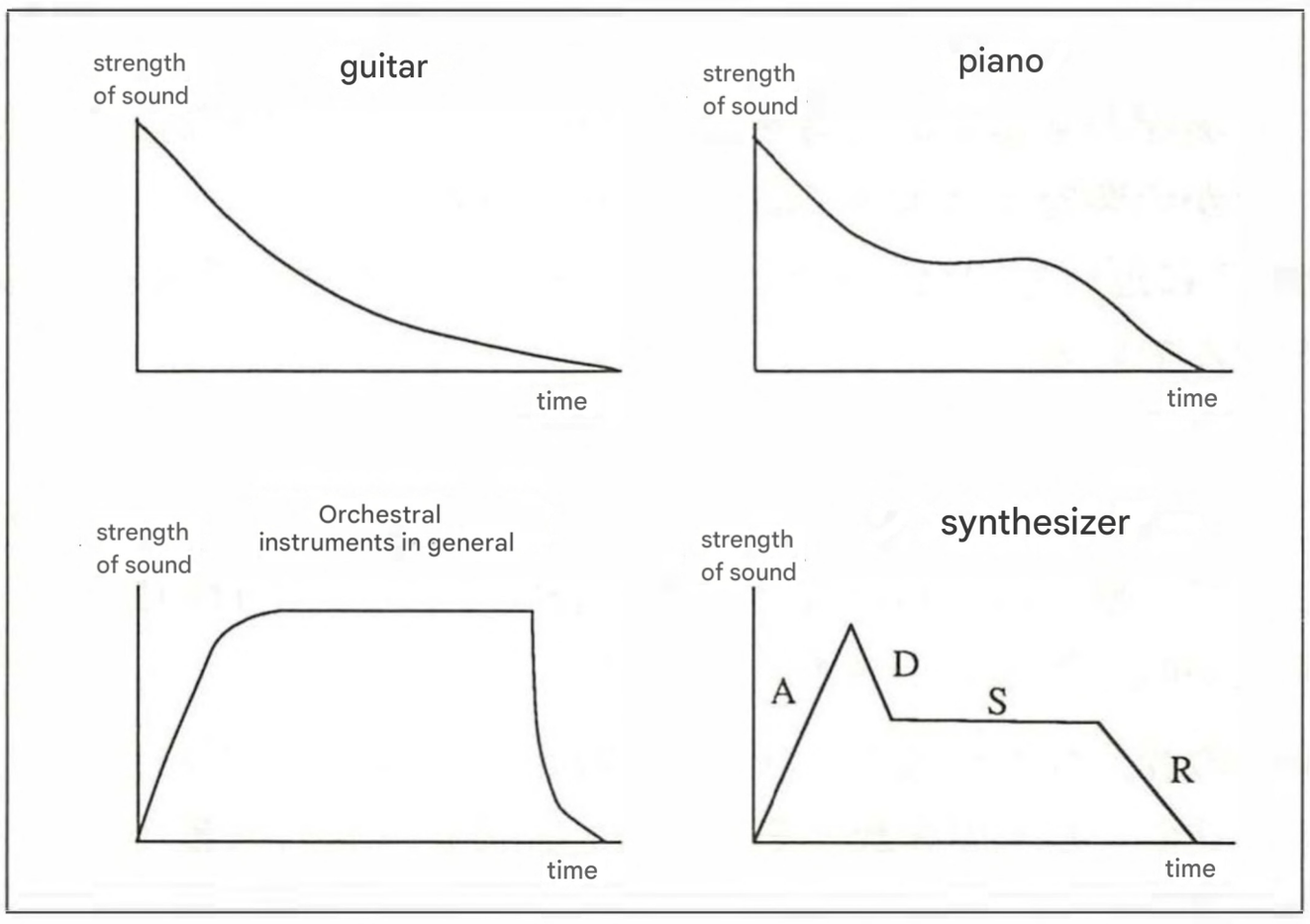
In addition to the basic waveform, another element that characterizes sound is the change in volume. This volume is called the "envelope," as it "wraps" around the basic waveform.

For example, when you play a guitar or percussion instrument, a loud sound comes out at the moment and then the sound gradually fades away. When you press a key on a piano, a loud sound comes out at first, which gradually fades away, and the sound continues at a fairly constant volume while the key is pressed. When you release the key, the sound becomes softer. Similarly, with a typical orchestral instrument, the sound starts slowly, then continues at the same volume, and finally stops just as quickly as the start (see Figure 5.3).

In contrast, analog synthesizers and FM sound sources synthesize envelopes by adjusting the speed of the attack (A), the decay (D), the strength of the sustain (S), and the speed of the release (R). The device used to do this is called "ADSR".

In the song "Blow Wind, Call Storm," the rock group Pinfloyd once created a sound effect in which a cymbal sound was recorded and played in reverse, gradually growing louder and then suddenly stopping. However, with a synthesizer, you can synthesize such a sound just by slowing down the attack, speeding up the decay, and setting the sustain to 0. Also, since synthesizing an envelope is easier than synthesizing the waveform of an FM sound source, it would be interesting to adjust the ADSR yourself to create sound effects.

Figure 5.3: Instrument and synth envelopes



5.1.3 The pitch is not necessarily equal temperament.

Suddenly, let's talk a little bit about classical music. First, let's organize the correspondence between musical scales and frequencies, as shown in Table 5.2.

Table 5.2: Relationship between musical scale and frequency

A	440.0Hz
A#	466.2Hz
B	493.9Hz
C	523.3Hz
C#	554.4Hz
D	587.3Hz
D#	622.3Hz
E	659.3Hz
F	698.5Hz
F#	740.0Hz
G	784.0Hz
G#	830.6Hz
a	880.0Hz

Among these, the frequency of the note A is 440Hz, and the frequency of the note a one octave above it is 880Hz. In other words, the frequencies of notes one octave apart are twice as high. Also, the ratio of the frequencies of notes that are a semitone apart is approximately 1.0595, and the ratio of the frequencies of notes that are 12 semitones (one octave) apart is 1.059512, or 2.

The rule that expresses the frequencies of all musical scales in a geometric progression like this is called "perfect equal temperament." This temperament seems to have been established between the Baroque and Classical periods. When I was in high school, I was taught that "Bach created the equal temperament," but recent research has shown that the prevailing theory is that the equal temperament was created after Bach. Most modern music is performed based on this equal temperament.

Before this perfect equal temperament was created, tunings were used in which the frequency ratio was a rational number. Unlike equal temperament, this has a different frequency ratio for semitones depending on the location. For example, the ratio of C to C# is different from the ratio of B to C. Because the rules are more complicated than equal temperament, various tunings are set out, as shown in Table 5.3. However, with tunings other than equal temperament, although the chords sound beautiful, they also have the problem of being difficult to transpose songs.

Table 5.3: Temperaments that can be set in MSX-Music

number	Assigned pitch	number	Assigned pitch
0	Pythagoras	11	Just intonation cis major (b minor)
1	mean tone	12	Just intonation d major (h minor)
2	Werckmeister	13	Just intonation es major (c minor)
3	Werckmeister (fix 1)	14	Just intonation e major (cis minor)
4	Werckmeister (fix 2)	15	Just intonation f major (d minor)
5	Kirnberger	16	Just intonation fis major (es minor)
6	Kirnberger (fixed)	17	Just intonation g major (e minor)
7	Vallotti Young	18	Just intonation gis major (f minor)
8	Rameau	19	Just intonation a major (fis minor)
9	Perfect equal temperament (default setting)	20	Just intonation b major (g minor)
10	Just intonation c major (a minor)	21	Just intonation h major (gis minor)

Instruments like the violin, which can change frequency continuously, can accommodate any temperament. But to change the temperament of a piano, all the strings must be retuned, so it is practically impossible to change the temperament. However, the FM tone generator on the MSX allows you to select the temperament shown in Table 5.3. This means that it is possible to create the sounds of instruments that are not so easily realized, such as a just-tuned guitar. By taking advantage of this feature and fine-tuning the temperament by manipulating the registers of the FM tone generator, it may be possible to play gagaku or Ryukyu music with precision.

The frequency of FM synthesis is so accurate that tuning becomes an issue, but the frequency of PSG is not so accurate. Therefore, it is dangerous to use PSG as a standard for tuning instruments or vocal training. However, I have a poor sense of pitch, so I don't really understand the difference between tunings. To master FM synthesis, you need accurate pitch and sense in addition to knowledge of mathematics, electricity, and music theory, but there are very few people like that. Just like composers, tone designers, and programmers work together, it will be necessary to divide roles when creating game music.

5.1.4 Analyzing MSX-MUSIC

MSX-MUSIC comes with 63 pre-installed sounds. 15 of these are built into the FM sound source LSI, and the remaining 48 are stored in ROM. The data for the sounds stored in ROM is

```
CALL VOICE COPY
```

It can be called with the command: List 5.1 shows the program for displaying this data. Note that if you specify a tone number built into the FM sound source, an error will occur. In the case of the program in List 5.1,

```
Voice No. * has no data.
```

A message like this is displayed.

list 5.1 (READFM.BAS)

```
100 ' read VOICE DATA of MSX-MUSIC
110 ' by nao-i on 20. Apr. 1990
120 '
130 CALL MUSIC : DEFINT A-Z
140 DIM VI(15),VD(31),VO(3)
150 PRINT "Voice Number (0,...,63; 64 for all; 65 for end) ";
160 INPUT ME
170 IF 0 <= ME AND ME <= 63 THEN VN=ME : GOSUB 200 : GOTO 150
180 IF ME = 64 THEN FOR VN=0 TO 63 : GOSUB 200 : NEXT VN : GOTO 150
190 END
200 '
210 ON ERROR GOTO 460
220 CALL VOICE COPY(@VN,VI)
230 ON ERROR GOTO 0
240 FOR I=0 TO 15
250   VD(I*2)=VI(I) AND 255
260   VD(I*2+1)=(VI(I) / 256) AND 255
270 NEXT I
280 NA$=""
290 FOR I=0 TO 8
300   IF VD(I) THEN NA$=NA$+CHR$(VD(I))
310 NEXT I
320 PRINT : PRINT "Voice No.";VN;" : " NA$
330 PRINT "Transpose=";VI(4);
340 PRINT "  Feedback=";(VD(10) AND 14) / 2
350 FOR I=0 TO 3: VO(I)=VD(I+16): NEXT I
360 PRINT "Operator 0" : GOSUB 490
370 FOR I=0 TO 3: VO(I)=VD(I+24): NEXT I
380 PRINT "Operator 1" : GOSUB 490
390 CALL BGM(0)
400 CALL VOICE(@VN,@VN,@VN)
410 PLAY #2,"CED<G>CR","V6EGF<B>ER","V4GBADGR"
420 CALL VOICE(@0,@0,@0)
430 CALL BGM(1)
440 ON ERROR GOTO 0
450 RETURN
460 '*** error
470 PRINT "Voice No.";VN;" has no datum."
480 RESUME 440
490 '*** print data of an operator
500 PRINT "  AM =";(VO(0) ¥ 128) AND 1;
510 PRINT "  PM =";(VO(0) ¥ 64) AND 1;
520 PRINT "  EG =";(VO(0) ¥ 32) AND 1;
530 PRINT "  KSR=";(VO(0) ¥ 16) AND 1;
```

```
540 PRINT "  MULT=";VO(0) AND 15;
550 PRINT "  LKS=";(VO(1) ¥ 64) AND 3;
560 PRINT "  TL=";VO(1) AND 63
570 PRINT "  ADSR=";(VO(2) ¥ 16) AND 15;","; VO(2) AND 15;",";
580 PRINT (VO(3) ¥ 16) AND 15;","; VO(3) AND 15
590 RETURN
600 ON ERROR GOTO 0
```

There are two types of operators operated in the program. Operator 1 is a "carrier operator" that creates the basic waveform, and operator 2 is a "modulator operator" that modulates 1. Explaining the meaning of each set value would take up a whole book, so I will not go into it here. I would like you to look it up yourself using a reference book or something. By the way, if you change "PRINT" in List 5.1 to "LPRINT" and print out the table of tone data, it may be useful as a reference for designing your own tones.

5.1.5 Try making rhythm sounds using FM sound source

Not just in MSX-MUSIC, but in general, FM synthesis is not good at producing percussion sounds. Actual percussion instruments and percussion sounds produced by analog synthesizers are irregular noises, but the percussion sounds produced by FM synthesis are too regular, which makes them sound "cheap" or "mechanical."

So MSX-MUSIC has a function to generate "rhythm sounds" in addition to the 63 kinds of instrument sounds. The 63 kinds of instrument sounds include percussion sounds, but these are sounds that are synthesized in the same way as the instrument sounds. The rhythm sounds mentioned here are something completely different.

As written in the manual, MSX-MUSIC has nine built-in two-operator FM sound sources, numbered from channel 1 to 9. If you use all of them as instrument sounds, you can play nine voices.

However, as a way to create rhythm sounds, it is also possible to assign channels 1 through 6 to normal instrument sounds, and use three sets of six operators from channels 7 through 9 as rhythm sounds. That is why the expression "9 instrument sounds or 6 instrument sounds + 1 percussion sound" is used to describe the functions of MSX-MUSIC.

To use these functions from BASIC, set the parameters of the “CALL MUSIC” instruction as follows:

For example, change

```
CALL MUSIC(0,0,1,1,1,1,1,1,1,1,1)
```

There are 9 instrument sounds.

```
CALL MUSIC(1,0,1,1,1,1,1,1)
```

Selects 6 instruments + 1 percussion sound.

For reference, the following program is designed to help you hear the difference between MSX-MUSIC tone data and rhythm sounds you've created yourself. First, play instrument tone number 31, a 2-operator "Bass Drum," four times, then play the 6-operator bass drum sound four times. Enter it in and listen with your own ears to hear how the rhythm sounds more similar to real drums.

list **5.2 (BASSDRUM.BAS)**

```
10 CALL MUSIC (1,0,1,1,1,3)
20 CALL BGM(0)
30 CALL VOICE(@31)
40 PLAY #2,"V15CCCC","","","","RRRRB!4B!4B!4B!4"
50 CALL VOICE(@0)
```

Also, since the MSX can play FM and PSG sounds simultaneously, it is possible to use FM to produce musical sounds and PSG to produce percussion sounds and sound effects. However, the balance between the volume of the FM sound and the PSG sound varies depending on the model of the MSX console, so a program is needed to adjust the volume for each machine.

5.2 Control FM sound source

FM synthesis has become indispensable for background music and sound effects in games these days.

On this page, we will try to control an FM synthesis from a machine language program.

5.2.1 Making sounds with a machine language program

As mentioned in the previous page, it was easy to control the FM sound source by using Extended BASIC. However, to use it for game sound effects or background music, the machine language program had to directly call "FM-BIOS" to control the FM sound source.

From here on, I will introduce libraries and example programs that allow programs written in MSX-C to operate FM sound sources. Naturally, to compile these programs and create executable machine language files, you will need "MSX-DOS TOOLS (or DOS2 TOOLS)" and "MSX-C ver.1.1 (or ver.1.2)".

List 5.3 is the test program "TESTFM.C" written in MSX-C. The array "fmdata" is the test data, which plays "Do Re Mi Fa So La Si Do" four times while hitting the bass drum and snare drum. How to create this data is also written in the document published on the msx.spec board on ASCII Net MSX.

list 5.3 (TESTFM.C)

```
/*
 *      testfm.c
 *      by nao-i on 29. May. 1 90
 *      (C) Isikawa 19 0      9
 *      free to use and 9 copy, but no guarantee or support
 */
#include <stdio.h>
#include "fmlib.h"
#pragma nonrec

#define TESTLENGTH      20
#define TESTTIMES       4

static  char    fmdata[] = {            /* test data            */
        14, 0,                          /* 0: offset to rythme  */
        33, 0,                          /* 2: offset to ch 1    */
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,   /* 4: offset to ch 2...6*/

        FM_RVOL + 31, 8,                /* 14: rhythm VOL = 8   */
        0x30, TESTLENGTH,               /* 16: B drum           */
        0x28, TESTLENGTH,               /* 18: S drum           */
```

```
        0x28, TESTLENGTH,               /* 20: S drum          */
        0x28, TESTLENGTH,               /* 22: S drum          */
        0x30, TESTLENGTH,               /* 24: B drum          */
        0x28, TESTLENGTH,               /* 26: S drum          */
        0x28, TESTLENGTH,               /* 28: S drum          */
        0x28, TESTLENGTH,               /* 30: S drum          */
        FM_END,                         /* 32: end of rhythm   */

        FM_VOL + 8,                     /* 33: Ch. 1 VOL = 8   */
        FM_INST + 3,                    /* 34: Guiter          */
        FM_SUSON,
        FM_LEGOFF,
        FM_Q, 6,
        FM_O4 + FM_C, TESTLENGTH,
        FM_O4 + FM_D, TESTLENGTH,
        FM_O4 + FM_E, TESTLENGTH,
        FM_O4 + FM_F, TESTLENGTH,
        FM_O4 + FM_G, TESTLENGTH,
        FM_O4 + FM_A, TESTLENGTH,
        FM_O4 + FM_B, TESTLENGTH,
        FM_O5 + FM_C, TESTLENGTH,
        FM_END                          /* end of Ch. 1        */
        };


VOID    main(argc, argv)
int     argc;
char    **argv;
{
        auto char fmwork[FMWORK]; /* address must be >= 8000H */
        auto char fmbuf[256];     /* address must be >= 8000H */
        char      fmstat;

        if ((fmstat = fmopen(fmwork)) == 1) {
                puts("No FM-BIOS.");
                exit(1);
        } else if (fmstat == 2) {
                puts("Bad address.");
                exit(1);
        }
        printf("fmopen : address of work area is %04X¥n",
          (unsigned)fmwork);
        memcpy(fmbuf, fmdata, sizeof(fmdata));
        fmstart(fmbuf, (char)TESTTIMES);
        do {
                fputs("Playing.¥015", stdout);
        } while (fmtest());
        fputs("¥nEnd of play.¥n", stdout);
        fmstop();
        fputs("fmstop : complete¥n", stdout);
        fmclose();
        fputs("fmclose : complete¥n", stdout);
        exit(0);
}
```

Let's briefly explain the program. First, prepare an auto array "fmwork" with a size of "FMWORK" bytes. Then, pass its address and call the library's "fmopen". This array needs to be placed at an address above 8000H, so declare it as auto instead of static.

By following the above procedure, "fmopen" should return 0 if the FM sound source is successfully prepared, 1 if there is no FM sound source, and 2 if the address of "fmwork" is 7FFFH or lower.

It is important to note that you must call "fmopen" before using the FM sound source, and call "fmclose" before the program ends. It would be inconvenient if the program ended before "fmclose" was called, so this library ignores the (CTRL+C or CTRL+STOP keys).

If you are creating a program that uses FM sound, it is important to properly handle CTRL+C and disk errors. In any case, be sure to call "fmclose" before terminating the program.

Now, when you call "fmstart" with the address where the data is stored and the parameters for the number of times to play, the FM-BIOS will immediately start playing. This FM-BIOS is designed to run on a timer interrupt, so you can continue playing while the program advances. In this program, I tried to display "Playing." on the screen while playing.

"fmtest" returns 1 if playing, and 0 if playing has finished. "fmstop" ends playing and initializes the FM-BIOS.

5.2.2 Provide an overview of the library

List 5.4 shows the header file “FMLIB.H”, which defines the functions and constants of the FM sound library.

list 5.4 (FMLIB.H)

```
/*
 *      fmlib.h : header file for fmlib
 *      by nao-i on 31. May. 1990
 * by nao-i on 24. Feb. 1991  FM_01 changed from 0 to 1
 *      (C) Isikawa 1990
 *      free to use and copy, but no guarantee or support
 */
extern  char    fmopen();       /* please call this first       */
extern  VOID    fmclose();      /* please  all this last        */
```

c

```
extern  VOID    fmwrite();      /* write to OPLL register       */
extern  VOID    fmotir();       /* write to OPLL register 0...7 */
extern  VOID    fmstart();      /* back ground music            */
extern  VOID    fmstop();       /* stop back ground music       */
extern  char    *fmread();      /* read data from ROM           */
extern  char    fmtest();       /* now playing ?                */
#define FMWORK  (0x00a0+32)     /* size of work area            */

#define FM_VOL          0x0060  /* volume 60H...6FH             */
#define FM_INST         0x0070  /* instulment 70H...7FH         */
#define FM_SUSOFF       0x0080  /* sustain off                  */
#define FM_SUSON        0x0081  /* sustain on                   */
#define FM_EXPINST      0x0082  /* expandet instulment          */
#define FM_USRINST      0x0083  /* user-defined instulment      */
#define FM_LEGOFF       0x0084  /* legato off                   */
#define FM_LEGON        0x0085  /* legato on                    */
#define FM_Q            0x0086  /* Q                            */
#define FM_END          0x00ff  /* end of data                  */
#define FM_RVOL         0x00a0  /* volume of rhythm             */

/* pitch */
#define FM_C    0       /* C    */
#define FM_CS   1       /* C#   */
#define FM_D    2
#define FM_DS   3
#define FM_E    4
#define FM_F    5
#define FM_FS   5
#define FM_G    7
#define FM_GS   8
#define FM_A    9
#define FM_AS   10
#define FM_B    11
/* octove */
#define FM_O1   1       /* FM_O1+FM_C means C of octove 0       */
#define FM_O2   13
#define FM_O3   25
#define FM_O4   37
#define FM_O5   49
#define FM_O6   61
#define FM_O7   73
#define FM_O8   85
```

The next long list, List 5.5, is the FM sound library. From the top of the list, it contains the address definitions for BIOS, etc., the macro definitions for calling FM-BIOS, the definition of the work area used by the library, and the library program itself.

list **5.5 (FMLIB.Z80)**

```
;
;       fmlib.z80 : library for MSX-C
;       by nao-i on 29. May. 1990
;       (C) ASCII 1988 for 'search', (C) Isikawa 1990
;       free to use and copy, but no guarantee or support
;
;       .Z80
;
;       address of BIOS and system work area
;
rdslt   equ     000ch
calslt  equ     001ch
enaslt  equ     0024h
breakv  equ     0f325h          ; ^C break vector
ramad0  equ     0f341h          ; slot # of RAM
ramad1  equ     0f342h
ramad2  equ     0f343h
ramad3  equ     0f344h
exptbl  equ     0fcc1h
h.timi  equ     0fd9fh          ; timer interrupt hook
;
;       address of FM-BIOS jump table
;
ids tr  e u     4018h+4
_wr o   e u     4110h
_ir io  e u     4113h
_ms ta  e u     4116h
_ms o   e u     4119h
_rd a   e u     411ch
_op ld  e u     411fh
_ts b   e u     4122h
;
;       MACROs to call FM-BIOS
;
CALLFM  MACRO   ADDRESS
        ld      ix,address
        ld      iy,(biosslot-1)
        call    calslt
        ENDM
;
JUMPFM  MACRO   ADDRESS
        ld      ix,address
        ld      iy,(biosslot-1)
        jp      calslt
        ENDM
;
        dseg
biosslot: ds    1               ; slot of FM-BIOS
breaks:   ds    2               ; saving ^C vector
p.ontime: ds    2               ; address of interrupt handler
p oldhook:ds    2               ; address of saved hook
;
```

```
        cseg
;
ontime:
        push    af
        ld      ix,_opldrv
        ld      iy,0            ; will be modifyed
ontime.slot equ $-1
        call    calslt
        pop     af
oldhook:
        nop
        nop
        nop
        nop
        nop
        ret
sizeof_ontime equ $-ontime
IF      sizeof_ontime GT 31
        .PRINTX "ontime routine too big"
ENDIF
;
hooktbl:
        rst     30h
        db      0               ; will be modifyed
        dw      0               ; will be modifyed
toret:
        ret
vvv:
        dw      toret           ; to ignore ^C
;
;       char    fmopen(address)
;       char    *address;       /* address of work area */
;
;       0 : successful
;       1 : no FM-BIOS
;       2 : bad address of work area
;
fmopen@::
        ld      a,2
        bi      7,h
        ret     z               ; address of work area < 8000H

        ld      (p.ontime),hl
        push    hl
        ex      de,hl
        ld      hl,ontime
        ld      bc,sizeof_ontime
        ldir                    ; trasfer ontime routine
        pop     hl
        push    hl
        ld      de,oldhook-ontime
        add     hl,de
        ld      (p.oldhook),hl
        pop     hl
        ld      de,32
```

```
        add     hl,de
        res     0,l             ; make sure address is even

        push    hl
        call    search
        ld      a,(biosslot)
        ld      hl,(p.ontime)
        ld      de,ontime.slot-ontime
        add     hl,de
        ld      (hl),a          ; modify LD IY,??00
        or      a
        ld      a,1
        pop     hl              ; address of work area
        ret     z               ; no FM-BIOS
        CALLFM  _iniopl
        ld      h,40h
        ld      a,(ramad1)      ; because FM-BIOS
        call    enaslt          ; does not restore slot1
;
        ld      hl,h.timi
        ld      de,(p.oldhook)
        ld      bc,5
        ldir                    ; save h.timi
        ld      hl,hooktbl
        ld      de,h.timi
        ld      bc,5
        di
        ldir                    ; set h.timi
        ld      a,(ramad2)
        ld      (h.timi+1),a
        ld      hl,(p.ontime)
        ld      (h.timi+2),hl
;
        ld      hl,(breakv)
        ld      (breaks),hl     ; save break vector
        ld      hl,vvv
        ld      (breakv),hl     ; set break vector
        ei
        xor     a
        ret
;
;       VOID    fmclose()
;
fmclose@::
        di
        ld      hl,(breaks)
        ld      (breakv),hl     ; restore break vector
        ld      hl,breaks
        ld      hl,(p.oldhook)
        ld      de,h.timi
        ld      bc,5
        ldir                    ; restore h.timi
        ei
        ret
;
```

```
;       VOID    fmwrite(RegNum, Datum)
;       char    RegNum;         /* OPLL register number         */
;       char    Datum;          /* datum to write               */
;
fmwrite@::
        JUMPFM  _wrtopl
;
;       void    fmotir(aData)
;       char    aData[8];
;
fmotir@::
        ld      b,8
        xor     a
        di
fmotir_loop:
        ld      e,(hl)
        inc     hl
        CALLFM  _wrtopl
        inc     a
        djnz    fmotir_ oop
        ret
;
;       void    fmstart(pDatum, Times)
;       char    *pData;         /* pointer to Music data        */
;       char    Times;          /* times to play                */
;
fmstart@::
        ld      a,e
        inc     a
        ret     z               ; make sure that Times != 255
        dec     a
        bit     7,h
        ret     z               ; make sure that pData >= 8000H
        JUMPFM  _mstart
;
;       VOID    fmstop()
;
fms op@::
        JUMPFM  _mstop
;
;       char    *fmread(ptr, num)
;   t   char    *ptr;
;       char    num;
;
fmread@::
        ld      a,e
        JUMPFM  _rddata
;
;       char    fmtest()
;
fmtest@::
        JUMPFM  _tstbgm
;
; Search FM-BIOS
;
```

```
search:
        ld      b,4
search_id:
        push    bc              ;save counter
        ld      a,4
        sub     b               ;make primary slot number
        ld      c,a             ;save it
        ld      hl,exptbl       ;point expand table
        ld      e,a
        ld      d,0
        add     hl,de
        ld      a,(hl)          ;get the slot is expanded or not
        add     a,a             ;expanded ?
        jr      nc,no_expanded  ;no..
        ld      b,4             ;number of expanded slots
search_exp:
        push    bc              ;save it
        ld      a,24h           ;[a]=00100100b
        sub     b               ;make secondary slot # A=001000ss
        rlca                    ;[a]=01000ss0b
        rlca                    ;[a]=1000ss00b
        or      c               ;make slot address A=1000sspp
        call    chkids          ;check id string
        pop     bc              ;restore counter
        jr      z,search_found  ;exit this loop if found
        djnz    search_exp
not_found:
        xor     a
        ld      (biosslot ,a
        pop     bc
        djnz    search_id
        ret
                        )
no_expanded:
        ld      a,c             ;get slot address
        call    chkids          ;check id string
        jr      nz,not_found    ;exit this loop is found
search_found:
        pop     bc
        ret
;
id_string:
        db      'OPLL'
id_string_len   equ     $-id_string
;
; Check ID srting
;  Entry : [A]=slot address to check
;  Return: Zero flag is set if ID is found
;  Modify: [AF],[DE],[HL]
;
chkids:
        ld      (biosslot),a
        push    bc              ;save environment
        ld      hl,idstrg
        ld      de,id_string
```

```
        ld      b,id_string_len
chkids_loop:
        push    af              ;save slot address
        push    bc              ;save counter
        push    de              ;save pointer to string
        call    rdslt           ;read a byte
        ei                      ;leave critical
        pop     de              ;restore pointer
        pop     bc              ;restore counter
        ld      c a             ;save data
        ld      a,(de)          ;get character
        cp      c,              ;same ?
        jr      nz,differ       ;no..
        pop     af              ;restore slot address
        inc     de              ;point next
        inc     hl
        djnz    chkids_loop
        pop     bc              ;restore environment
        xor     a               ;found set zero flag
        ret
;
differ:
        pop     af              ;restore slot address
        pop     bc              ;restore environment
        xor     a               ;clear zero flag
        inc     a
        ret
;
        end
```

To explain the key points of the program, first "fmopen@" transfers the timer interrupt processing program from "ontime" to a different address, searches for the slot where the FM-BIOS is located, initializes it, and sets the timer interrupt hook. The interrupt processing program and the data it references must be located above address 8000H, so after transferring the program, "ld iy, 0" is changed to "ld iy, FM-BIOS slot number * 256".

The program itself, which searches for the slot in which the FM-BIOS is located, was taken from the FM-BIOS specifications published on ASCII Net MSX. It searches for the FM-BIOS by checking whether the string "OPLL" exists in address 401CH for all slots.

Now, the 192 byte work area passed to "fmopen@" is used for the interrupt handler, the storage location for the original contents of "h.timi", and the FM-BIOS work area (160 bytes). Since the FM-BIOS work area needs to start at an even address, an extra work area is prepared and the starting address is rounded up.

This was not written in the specifications, but when I initialized the FM-BIOS with "CALLFM_iniopl", page 1 sometimes returned with the page switched to a different slot. It is necessary to return page 1 to the original slot.

As I wrote before, it would be a problem if the program terminated with the timer interrupt hook overwritten, so I overwrote the F325H address of the MSX-DOS work area to ignore the CTRL+C key. If the program uses a disk, you need to add a program to handle disk errors. And "fmclose@" is to restore the timer interrupt hook and CTRL+C key processing.

For the rest of the library, fill in the registers with the required values and run the FM-BIOS. You can use it just by calling . Please try it out for yourself.

5.2.3 Compiling with MSX-C

After typing up the lists in an MSX-DOS editor such as MED or KID, save Listing 5.3 as "TESTFM.C", Listing 5.4 as "FMLIB.H", and Listing 5.5 as "FMLIB.Z80". Compile them using the following steps to create "TESTFM.COM". To run the programs, simply type "TESTFM ↵" from the MSX-DOS command line.

list 5.6 (FMLIB.BAT)

```
m80 ,=fmlib.z80/r/m/z
cf testfm
cg testfm
m80 ,=testfm.mac/r/m/z
l80 testfm,fmlib,ck,clib/s,crun/s,cend,testfm/n/y/e:xmain
```

5.3 FM sound source data structure

This page explains the data structure of FM sound sources and how to actually specify sound source data. Let's use it together with the program that controls FM sound sources in machine language explained earlier.

5.3.1 Let's create FM sound data

On the previous page, we introduced a program example that calls the FM-BIOS from machine language to produce sound. Here, we will explain how to create FM sound source data to play using that program.

The FM-BIOS data structure can be summarized as a large array. Each data location in the array is counted in bytes from the beginning of the array and is called an "offset", where the beginning of the array has an offset of 0. The beginning of the array is sometimes called "byte 0", the next "byte 1", and so on.

Table 5.4 shows the data structure of 6 instruments + 1 percussion instrument, along with an example of the actual data. As you can see, the data itself is arranged toward the end of the array, and the first 14 bytes of the array contain the offsets where each piece of data will be placed.

Table 5.4: Data structure of 6 musical instruments + 1 percussion sound

offset	Content
0	Percussion sound data offset (Note 1)
2	Instrument sound 1 data offset (Note 2)
4	Instrument Sound 2 Data Offset (Note 2)
6	Instrument Sound 3 Data Offset (Note 2)
8	Offset of instrument sound 4 data (Note 2)
10	Instrument Sound 5 Data Offset (Note 2)
12	Offset of 6 instrument sounds (Note 2)
14	percussion instrument sound data
(Note 3)	Instrument sound 1 data
(Note 3)	Instrument sound 6 data

(Note 1) Always write two bytes, 0EH and 00H.

(Note 2) The offset in bytes from the beginning of the data in this table to the start position of the instrument sound data is specified in the order of low byte and high byte. Also, specify 0 for unused channels.

(Note 3) The instrument sound data is placed at the location specified by the offset.

To explain the data structure in order, the 0th and 1st bytes (offsets 0 and 1) are the offsets where the percussion sound data is placed, and are always written with the value 14. When this value is expressed as 2 bytes, low byte first, high byte second, so that the Z80 CPU can understand it, it becomes 0EH and 00H, respectively.

The following 2nd to 13th bytes (offsets 2 to 13) contain the offsets of data for channels 1 to 6 of the instrument sound. If an offset of 0 is specified at this time, that channel will not be used.

Table 5.5: Example of data structure for 6 musical instruments + 1 percussion sound

offset	Content
0	0EH 00H
2	21H 00H
4	00H 00H
6	00H 00H
8	00H 00H
10	00H 00H
12	00H 00H
14～	percussion instrument sound data
33～	Instrument sound 1 data

This is an example of the data structure for 6 instruments + 1 percussion instrument. The 14th to 32nd bytes contain the sound source data for the percussion instruments, and the 32nd byte contains the sound source data for the 1st channel of the instrument. The 2nd to 6th channels of the instrument are unused.

For example, in the data structure example in Table 5.5, 21H and 00H are written in the second and third bytes (offsets 2 and 3), which means that the sound source data for instrument channel 1 is located from offset 33 onwards. And because 00H is written in the fourth to thirteenth bytes (offsets 4-13), we can see that instrument channels 2-6 are not used.

In the program that controls the FM sound source introduced earlier, test data was embedded in the C language program, so it was necessary to count the length of the data and specify the offset. However, when I think about it, it might have been easier to create the sound source data with an assembler program like the one below. For reference, I would like to write:

```
FMDATA:
    DW 14
    DW CH1-FMDATA
    DW CH2-FMDATA
    DW CH3-FMDATA
    DW CH4-FMDATA
    DW CH5-FMDATA
    DW CH6-FMDATA
    DB Percussion instrument sound data...
CH1:
    DB Channel 1 data...
CH2:
    DB Channel 2 data...
(The same applies up to CH6)
```

This is because the MSX-DOS assembler (M80) automatically calculates the offsets when assembling the source listing in this program.

However, this program cannot be used as is for practical purposes. A program to convert Music Macro Language (MML), which plays the role of the PLAY statement in BASIC, into FM-BIOS sound source data will be required.

Now, when playing 9 instruments without percussion instruments, the data structure will be as shown in Table 5.6. Basically, it is the same as the data structure for 6 instruments + 1 percussion instrument.

First, the offset where the sound source data for instrument channels 1 to 9 is written is specified from byte 0 to byte 17 (offset 0 to 17). If 00H is specified, the corresponding channel will not be used.

Also, since the audio data for channel 1 always starts from the 18th byte (offset 18), the offset of the channel 1 data is always written with the values 12H, 00H (18 in decimal).

Table 5.6:9 Data structure of musical instrument sounds

offset	Content
0	Instrument sound 1 data offset (Note 1)
2	Instrument Sound 2 Data Offset (Note 2)
⋮	
16	Instrument Sound 9 Data Offset (Note 2)
18	Instrument sound 1 data
⋮	
(Note 3)	Instrument sound 9 data

(Note 1) Always write two bytes: 12H and 00H.

(Note 2) The offset in bytes from the beginning of the data in this table to the start position of the instrument sound data is specified in the order of low byte and high byte. Also, specify 0 for unused channels.

(Note 3) The instrument sound data is placed at the location specified by the offset.

By the way, this is a bit of a digression, but FM-BIOS decides whether there are percussion sounds (i.e. 6 instruments + 1 percussion sound, or only 9 instruments) depending on whether the beginning of the sound source data is 0EH or 12H. Therefore, it seems that the data for percussion sounds must always start from the 14th byte (offset 14), and if there are no percussion sounds, the data for the instrument sound on channel 1 must always start from the 18th byte (offset 18).

5.3.2 To specify percussion data

Figure 5.4 shows the details of the percussion sound data and the actual data string. Here, the five types of percussion are represented by the alphabet "BSTCH" and the sound data is represented by binary numbers.

First, let's try specifying the volume of each percussion instrument using the following two bytes of data.

```
101BSTCH
0000VVVV
```

Figure 5.4: Percussion sound data

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
1	0	1	B	S	T	C	H
0	0	0	0	volume $(0\sim15)$			
0	0	1	B	S	T	C	H
tone length $(1\sim255)$							

B	bass drum
S	snare drum
T	Tam
C	cymbal
H	hi hat

```
10110000  bass drum
00000000  Volume 0
10101000  snare drum
00000001  volume 1
00110000  bass drum
00010100  Length 20
00101000  snare drum
00010100  Length 20
00101000  snare drum
00010100  Length 20
00101000  snare drum
00010100  Length 20
11111111  End of data
```

To select a percussion instrument, specify 1 or 0 for each of the 5 bits marked BSTCH. The volume indicates the attenuation from the maximum volume, and the note length indicates the interval from one note to the next.

In this case, in the 5-bit string of "BSTCH", specify the percussion instrument for which you want to specify the volume with 1, and the others with 0. Also, the "VVVV" part contains a value from 0 to 15 (actually a binary value) that specifies the volume. Note, however, that "volume" in this case specifies the amount of attenuation relative to the maximum volume, so 0 will produce the maximum sound and 15 will produce the minimum sound.

For example, set the bass drum volume to 0 and the snare drum and cymbal volumes to 1. To do this,

```
10110000
00000000
10101010
00000001
```

All you have to do is specify.

Once you've finished specifying the volume, you can then specify the duration of each percussion instrument. However, since the duration of the percussion sound itself is always constant, "duration" in this case refers to the interval between one sound being played and the next. Then,

```
001BSTCH
```

The type of percussion instrument is specified by the byte, and the next byte specifies the note length (a value up to 255). To specify a note length greater than 255, first write 255, then specify the actual note length minus 255.

If this value is 255 or more, repeat the same operation. For example,

```
00110000
11111111
00000000
```

represents a bass drum, duration 255,

```
00110010
11111111
11111111
11111111
11101011
```

represents a bass drum and cymbal, with a note length of 1000.

The specifications for the FM sound source did not specify the unit of "tone length," but actual measurements using test data revealed that the unit of tone length was the same as the timer interrupt period, 1/60th of a second.

5.3.3 Let's specify the data of the instrument sound

Table 5.7 shows the details of the instrument sound data. Some of this data has meaning in only one byte of the table, while others have meaning in combination with the following one or two bytes.

The order in which you actually specify instrument sound data is volume, timbre, sustain, legato, and Q.

The volume is specified in the same way as percussion data, by the amount of attenuation from the maximum volume. In other words, 0 is the loudest sound, 15 is the quietest sound, and so on.

Sustain is used to adjust the decay of the instrument sound. As explained before, the envelope of the instrument sound is determined by the value "ADSR". To repeat, A stands for attack (how fast it rises), D for decay (how fast it decays), S for sustain (how strong it is), and R for release (how fast it fades away).

The "ADSR" of the OPLL built-in tones is fixed for each tone. However, when you turn on sustain, the release becomes slower and the sound is extended. Furthermore, since sustain can be specified for each channel, you can make detailed adjustments such as assigning a guitar sound to both channels 1 and 2 and turning on sustain only for channel 1.

When legato is turned on, the sound of one note is connected to the next note. However, if you use legato too much, the song will lose its sharpness, so you may need to turn on legato only for some channels.

Table 5.7: Instrument sound data

Value	meaning
00H	Rest, the next byte is note length
01H	C in octave 1, the next byte is the note length
⋮	⋮
5FH	A# in octave 7, the next byte is the note length
60H	Volume 0 (max volume)
⋮	⋮
6FH	Volume 15 (minimum volume)
70H	Tone 0 (ROM built-in tone or user-specified tone)
71H	Tone 1 (Violin)
⋮	⋮
7FH	Tone 15 (Electric Bass)
80H	sustain off
81H	sustain on
82H	The next byte (00H~3FH) is the ROM built-in tone number
83H	The next two bytes (lower, higher) are the address of the tone data.
84H	legato off
85H	legato on
86H	The next byte (01H~08H) is Q
FFH	End of data

The value that can be specified for Q is 1 to 8, and represents the ratio of the length of the note to the length of the actual sound. For example, if Q=6 and the length of the note is 80, a sound of 80×6÷8=60 will be produced, and a rest of 80×(8-6)÷8=20 will be inserted.

The combination of values specified for legato, sustain, and Q determines how the notes connect, and thus how smooth the song will be.

Table 5.8: Example of instrument sound data

68H	volume 8
73H	Tone 3 (Guitar)
81H	sustain on
84H	legato off
86H, 06H	$Q = 6$
25H, 14H	C in octave 4, duration 20
27H, 14H	D in octave 4, duration 20
29H, FFH, FFH, FFH, EBH	E in octave 4, duration 1000
FFH	End of data

Next, specify the scale and note length for each note. A byte value from 00H to 5FH represents the scale, and the next byte represents the note length. For example,

```
25H,14H
```

The two bytes of data above represent the note C in octave 4 and the note length 20, respectively.

Also, the method for expressing note lengths over 255 is the same as for percussion instruments. For example,

```
29H,FFH,FFH,FFH,EBH
```

These five bytes of data represent the E in octave 4 and a note length of 1000.

5.3.4 OPLL Things you can't do with a driver

Among the functions of FM-BIOS, the one that is called by timer interrupt and automatically plays the given data is called the "OPLL driver". Here, we will use this driver to

```
CALL PITCH
CALL TEMPER
CALL TRANSPOSE
```

I was going to explain how to achieve the same function as above. However, when I asked the developer of FM-BIOS, they told me to "do it yourself." In other words, it turned out that I had to rewrite the OPLL register myself to achieve this.

In the end, when using the OPLL driver, it is only possible to play in the standard temperament of 12-tone equal temperament with A at 440 Hz. The MML supported by BASIC has a function to set the volume in detail and a function to write values to the registers of the sound chip, but it is impossible to do the same thing with the FM-BIOS driver. It is also difficult to produce sound effects while playing music in the background of a game.

Thinking about it this way, it seems that to make FM synthesis easy to use, we need a driver with the added functions I've just mentioned, and a program to convert MML into that driver's data. If you have read the articles up to this point and have a reference book on FM synthesis introduced later, you should have all the information you need. If you're confident in your programming skills, then by all means, give it a try.

5.3.5 Let's add some sound data.

As I wrote before, the FM sound source LSI (OPLL) has 15 types of tone data, and the FM-BIOS ROM that controls it has 48 types of tone data. However, it is possible to add more tones using the data structure shown in Figure 5.5.

Figure 5.5: Timbre data

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	
0	AM	VIB	EGT	KSR	Multiple				Op.0
1	AM	VIB	EGT	KSR	Multiple				Op.1
2	KSL (Op.0)		Total LEVEL MODELATER						
3	KSL (Op.1)		空き	DC	DM	Feedback			
4	Attack (Op.0)				Decay (Op.0)				
5	Attack (Op.1)				Decay (Op.1)				
6	Sustain (Op.0)				Release (Op.0)				
7	Sustain (Op.1)				Release (Op.1)				

Op.0 is the modulator operator, and Op.1 is the carrier operator. These 8 bytes are written to registers 0 to 7 of the OPLL. For details, see "MSX2+ Powerful Usage Method (published by ASCII)".

This 8-byte tone data is written directly to registers 0 to 7 of the FM tone generator LSI (OPLL).

```
CALL VOICE
CALL VOICE COPY
```

Note that this is a different format than the 32-byte data used in

Inside OPLL, the tone for each channel is specified by a value from 0 to 15. In this case, tone 0 represents the original tone set by OPLL registers 0 to 7. In other words, only one type of tone data you have created or tone data stored in the FM-BIOS ROM can be used at a time.

This is due to the limitation on the number of tones, not the number of channels. So, for example, you can assign your own tones to channels 1 and 2, and the tones built into OPLL to channels 2 to 4.

If I were to explain the contents of the tone data, I could write a book on it, so I will skip that this time. Instead, I will introduce some reference books.

"Powerful MSX2+ Utilization Method" by Masakazu Sugitani, published by ASCII Publishing Bureau

However, there seem to be some errors in this book. I will post the corrections on the following pages, so please correct them yourself.

In addition, if you are using computer communication, I would like you to visit the ASCII Net MSX board called "msx.spec." This board contains the specifications for "FM-BIOS," which allows machine language programs to use MSX-MUSIC.

In addition to that, it also contains a variety of information about MSX.

5.3.6 Explain sample data

Finally, to wrap things up, let's look at an example where we actually specify data. 5.7 is part of the program list I posted earlier.

First, the top section specifies the offset of the data for each channel. This indicates that the data for percussion sounds starts from byte 14, the data for instrument channel 1 starts from byte 33, and channels 2 through 5 are unused.

The center part of the list is the data for percussion sounds. First, the volume of all percussion instruments is set to 8. The value of "FM_RVOL" is 0xa0, and adding 31 to this gives 0xbf, which specifies the volume of all percussion instruments. Next, the bass drum is specified with a note length of 20, the snare drum is also specified with a note length of 20, and so on, specifying the sounds in order. The value of "FM_END" is 0xff, which indicates the end of the data.

The bottom half of the list is the data for instrument sound channel 1. First, specify volume 8, tone 3 (guitar with built-in OPLL), sustain on, legato off, and Q=6. Then, play "Do Re Mi Fa So La Si Do" with a note length of 20, and end when you reach "FM_END". At this time, since it is inconvenient to directly specify the scale with a number from 0 to 95, the values for 12 notes and octave are separated. For example,

```
FM_O4 + FM_D
```

This specifies the D in octave 4. The value of “FM_O4” is 37 and the value of “FM_D” is 2, so adding them together gives 39, which represents the D in octave 4.

To create test data in assembler (M80), it may be useful to define constants as follows:

```
FM_C     EQU 1
FM_CS    EQU 2
           :
FM_VOL   EQU 60H
           :
```

and the volume,

```
DB FM_VOL + 8
```

Like this. The same scale,

```
DB FM_O4 + FM_C
```

You can specify it like this.

List 5.7 (Sample data for musical instruments)

```
#define TESTLENGTH      20
#define TESTTIMES       4

static  char    fmdata[] = {
/* Here, an offset is specified for each channel.                    */
        14, 0,
        33, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

/* This is percussion instrument sound data.          */
/* First, set the overall volume, then specify the    */
/* length of each instrument.                         */
        FM_RVOL + 31, 8,
        0x30, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        0x28, TESTLENGTH,
        FM_END,

/* This is the data for instrument sound channel 1.                  */
/* In this listing, channels 2-5 are unused.                         */
        FM_VOL + 8,
        FM_INST + 3,
        FM_SUSON,
        FM_LEGOFF,
        FM_Q, 6,
        FM_O4 + FM_C, TESTLENGTH,
        FM_O4 + FM_D, TESTLENGTH,
        FM_O4 + FM_E, TESTLENGTH,
        FM_O4 + FM_F, TESTLENGTH,
        FM_O4 + FM_G, TESTLENGTH,
        FM_O4 + FM_A, TESTLENGTH,
        FM_O4 + FM_B, TESTLENGTH,
        FM_O5 + FM_C, TESTLENGTH,
        FM_END
        };
```

5.4 Various things related to FM sound sources

I thought that I had finished everything about FM synthesis, but it seems that I have something left to cover. Please bear with me for a little longer.

5.4.1 Correction of the content of powerful usage method

Several errors have been found in "MSX2+ Powerful Usage Method" (published by ASCII, priced at 1,240 yen including tax), which has been introduced as a reference book for MSX2+ machines. There must be many people who have been left scratching their heads when their programs did not work as expected even when they followed the instructions in the book. On this page, we will correct the errors that we are currently aware of. If anyone finds any errors other than those listed here, please let the M Magazine editorial department know.

Let's start with the first correction. Look at "Table 4.4 Sound Library List" on page 147. Add "2 Guitar" to the "OPLL VOICE" column for sound number 10, which has the sound name "Guitar".

There were also some errors in the "abbreviations" in this table. The correct abbreviations will be displayed if you run the program (READFM.BAS) on page 136 of this book. Please check the abbreviations while playing each tone.

Next, the explanation of "VOICE COPY" on page 148. In the middle of the sentence, it says "The voice numbers that can be specified for the source (parameter 1) are the voice numbers 0-63 specified in the OPLL VOICE column," but the correct statement should be "The voice numbers that are not specified."

Similarly, the following statement, "If you specify as the source the number of a voice that is not specified in the OPLL VOICE column, it will result in an 'Illegal function call'", is the opposite of what you'd expect. The correct statement is "If you specify the number of a voice that is specified...".

There were also some errors in the OPLL register table on pages 151 to 158. I have corrected them and included them in Figure 5.6, so please refer to it. I think it would be useful to use this as a basis for revising the contents of the MSX2+ Powerful Usage Guide that you have at hand.

In relation to the register explanation, see page 155 for information on how to find the F-Number and There was also an error in the formula for BLOCK.

$$\text{F–Number} = (440 \times 2^{18} \div 50000) \div 2^4 - 1 = 288$$

The formula is, correctly,

$$\text{F–Number} = (440 \times 2^{18} \div 50000) \div 2^{(4\,-\,1)} = 288$$

That's what it means.

Finally, although this is not limited to how to use MSX2+ Powerful, I would like to correct the commonly used explanation of how to specify instrument sound data for FM sound sources, as it seems to be incorrect.

To specify the scale for each note, 00H~5FH are the C~
It is commonly said that it corresponds up to B in the 7th octave, but this is incorrect.

figure 5.6: OPLL Register List

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
00H	AM(M)	VIB(M)	EGT(M)	KSR(M)	Multiple(M)			
01H	AM(C)	VIB(C)	EGT(C)	KSR(C)	Multiple(C)			
02H	KSL(M)		Total Level Modelater					
03H	KSL(C)		Vacant	DC	DM	Feed Back		
04H	Attack(M)				Decay(M)			
05H	Attack(C)				Decay(C)			
06H	Sustain(M)				Release(M)			
07H	Sustain(C)				Release(C)			
0EH	Vacant		R	BD	SD	TOM	T–CT	HH
0FH	Inspection register							
10H ⋮ 18H	F–number							
20H ⋮ 28H	Vacant		Sus.	Key	Block		F–number	
30H ⋮ 38H	Inst.				Vol.			

For rhythm mode

	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
36H	Vacant				Bass Drum			
37H	Hi Hat				Snare Drum			
38H	Tom Tom				Top Cymbal			

In the table, the part marked (M) indicates operator 0, which acts as a modulator, and the part marked (C) indicates operator 1, which acts as a carrier. For more information, please refer to "How to use MSX2+ powerfully" or Yamaha's technical documentation.

To be precise, 00H is a rest, and the following 01H~5FH correspond to C in octave 1 to A# in octave 7.

5.4.2 MSX-MUSIC tone data list

In response to requests from programmers, we are posting a dump list of the tone data built into the MSX-MUSIC ROM.

The left side of Table 5.9 shows the 8 bytes of data written to OPLL registers 0 to 7 from the 32 bytes of voice data obtained by the BASIC "CALL VOICE COPY" statement. The voice data also includes 2 bytes of data called "voice transposition" that controls the pitch, but since this data is not directly written to the OPLL registers, it is not shown here.

Tone 60 and tone 61 look exactly the same in this table, but they are actually different tones because the voice transposition values are different. Also, for tone numbers that are marked "using data of OPLL" in the table, the tone built into OPLL is used, so the ROM does not contain tone data.

Now, everyone believed without a doubt that the 63 types of tone data stored in ROM obtained by FM-BIOS were the same as the tone data stored in ROM of BASIC. However, it has now been discovered that they are actually different. So, the right side of Table 5.9 shows the tone data (8 bytes for each tone) obtained using the "RDDATA" function of FM-BIOS.

As you can see by comparing the left and right sides of Table 5.9, the tone numbers and names are the same for Extended BASIC and FM-BIOS, but the tone data is slightly different. Therefore, if you create a prototype song using BASIC's MML and then convert that data for FM-BIOS, the difference in tones may cause problems.

Some people may find it strange that the value written to register 3 for most of the tone data in FM-BIOS is 20H. However, because bit 5 of register 3 is "empty", whether the value written to register 3 is 20H or not, the tone that is actually played will be the same.

Table 5.9: List of timbre data

number	Tone name	BASIC extension tone data		FM-BIOS sound data	
0	Piano 1	using data of OPLL(3)		31 11 0E 20	D9 B2 11 F4
1	Piano 2	30 10 0F 04	D9 B2 10 F4	30 10 0F 20	D9 B2 10 F3
2	Violin	using data of OPLL(1)		61 61 12 20	B4 56 14 17
3	Flute 1	using data of OPLL(4)		61 31 20 20	6C 43 18 26
4	Clarinet	using data of OPLL(5)		A2 30 A0 20	88 54 14 06
5	Oboe	using data of OPLL(6)		31 34 20 20	72 56 0A 1C
6	Trumpet	using data of OPLL(7)		31 71 16 20	51 52 26 24
7	Pipe Organ 1	34 30 37 06	50 30 76 06	34 30 37 20	50 30 76 06
8	Xylophone	17 52 18 05	88 D9 66 24	17 52 18 20	88 D9 66 24
9	Organ	using data of OPLL(8)		E1 63 0A 20	FC F8 28 29
10	Guitar	using data of OPLL(2)		02 41 15 20	A3 A3 75 05
11	Santool 1	19 53 0C 06	C7 F5 11 03	19 53 0C 20	C7 F5 11 03
12	Electric Piano 1	using data of OPLL(15)		23 43 09 20	DD BF 4A 05
13	Clavicode 1	03 09 11 06	D2 B4 F5 F6	03 09 11 20	D2 B4 F4 F5
14	Harpsicode 1	using data of OPLL(11)		01 00 06 20	A3 E2 F4 F4
15	Harpsicode 2	01 01 11 06	C0 B4 01 F7	01 01 11 20	C0 B4 01 F6
16	Vibraphone	using data of OPLL(12)		F9 F1 24 20	95 D1 E5 F2
17	Koto 1	13 11 0C 06	FC D2 33 84	13 11 0C 20	FC D2 33 83
18	Taiko	01 10 0E 07	CA E6 44 24	01 10 0E 20	CA E6 44 24
19	Engine 1	E0 F4 1B 87	11 F0 04 08	E0 F4 1B 20	11 F0 04 08
20	UFO	FF 70 19 07	50 1F 05 01	FF 70 19 20	50 1F 05 01
21	Synthesizer Bell	13 11 11 07	FA F2 21 F5	13 11 11 20	FA F2 21 F4
22	Chime	A6 42 10 05	FB B9 11 02	A6 42 10 20	FB B9 11 02
23	Synthesizer Bass	using data of OPLL(13)		40 31 89 20	C7 F9 14 04
24	Synthesizer	using data of OPLL(10)		42 44 0B 20	94 B0 33 F6
25	Synthesizer Percussion	01 03 0B 07	BA D9 25 06	01 03 0B 20	BA D9 25 06
26	Synthesizer Rhythm	40 00 00 07	FA D9 37 04	40 00 00 20	FA D9 37 04
27	Harm Drum	02 03 09 07	CB FF 39 06	02 03 09 20	CB FF 39 06
28	Cowbell	18 11 09 05	F8 F5 26 26	18 11 09 20	F8 F5 26 26
29	Close Hi-hat	0B 04 09 07	F0 F5 01 27	0B 04 09 20	F0 F5 01 27
30	Snare Drum	40 40 07 07	D0 D6 01 27	40 40 07 20	D0 D6 01 27
31	Bass Drum	00 01 07 06	CB E3 36 25	00 01 07 20	CB E3 36 25

number	Tone name	BASIC extension tone data	FM-BIOS sound data
32	Piano 3	11 11 08 04 FA B2 20 F5	11 11 08 20 FA B2 20 F4
33	Electric Piano 2	using data of OPLL(14)	11 11 11 20 C0 B2 01 F4
34	Santool 2	19 53 15 07 E7 95 21 03	19 53 15 20 E7 95 21 03
35	Brass	30 70 19 07 42 62 26 24	30 70 19 20 42 62 26 24
36	Flute 2	62 71 25 07 64 43 12 26	62 71 25 20 64 43 12 26
37	Clavicode 2	21 03 0B 05 90 D4 02 F6	21 03 0B 20 90 D4 02 F5
38	Clavicode 3	01 03 0A 05 90 A4 03 F6	01 03 0A 20 90 A4 03 F5
39	Koto 2	43 53 0E 85 B5 E9 85 04	43 53 0E 20 B5 E9 84 04
40	Pipe Organ 2	34 30 26 06 50 30 76 06	34 30 26 20 50 30 76 06
41	PohdsPLA	73 33 5A 06 99 F5 14 15	73 33 5A 20 99 F5 14 15
42	RohdsPRA	73 13 16 05 F9 F5 33 03	73 13 16 20 F9 F5 33 03
43	Orch L	61 21 15 07 76 54 23 06	61 21 15 20 76 54 23 06
44	Orch R	63 70 1B 07 75 4B 45 15	63 70 1B 20 75 4B 45 15
45	Synthesizer Violin	61 A1 0A 05 76 54 12 07	61 A1 0A 20 76 54 12 07
46	Synthesizer Organ	61 78 0D 05 85 F2 14 03	61 78 0D 20 85 F2 14 03
47	Synthesizer Brass	31 71 15 07 B6 F9 03 26	31 71 15 20 B6 F9 03 26
48	Tube	using data of OPLL(9)	61 71 0D 20 75 F2 18 03
49	Shamisen	03 0C 14 06 A7 FC 13 15	03 0C 14 20 A7 FC 13 15
50	Magical	13 32 81 03 20 85 03 B0	13 32 80 20 20 85 03 AF
51	Huwawa	F1 31 17 05 23 40 14 09	F1 31 17 20 23 40 14 09
52	Wander Flat	F0 74 17 47 5A 43 06 FD	F0 74 17 20 5A 43 06 FC
53	Hardrock	20 71 0D 06 C1 D5 56 06	20 71 0D 20 C1 D5 56 06
54	Machine	30 32 06 06 40 40 04 74	30 32 06 20 40 40 04 74
55	Machine V	30 32 03 03 40 40 04 74	30 32 03 20 40 40 04 74
56	Comic	01 08 0D 07 78 F8 7F FA	01 08 0D 20 78 F8 7F F9
57	SE–Comic	C8 C0 0B 05 76 F7 11 FA	C8 C0 0B 20 76 F7 11 F9
58	SE–Laser	49 40 0B 07 B4 F9 00 05	49 40 0B 20 B4 F9 FF 05
59	SE–Noise	CD 42 0C 06 A2 F0 00 01	CD 42 0C 20 A2 F0 00 01
60	SE–Star 1	51 42 13 07 13 10 42 01	51 42 13 20 13 10 42 01
61	SE–Star 2	51 42 13 07 13 10 42 01	51 42 13 20 13 10 42 01
62	Engine 2	30 34 12 06 23 70 26 02	30 34 12 20 23 70 26 02
63	Silence	00 00 00 00 00 00 00 00	00 00 FF 20 00 00 FF FF

APPENDIX



R800 Instructions Table

January 24, 1991

ASCII Co., Ltd.

Systems Division, MSX Magazine Editorial Department

If you are interested in machine language level programming, we recommend you try developing a program on the R800. We have posted an instruction table with mnemonics, instruction operations, and machine language codes, so please make use of it. Can you write a program that takes advantage of the speed of the R800?

A.1 How to use the instruction sheet

This table summarizes the R800 instructions, classified by type. The "mnemonic" in the table shows the name of each instruction, and the "instruction action" briefly describes its action.

In the instruction operation column, "←←" means to assign the contents of the right side to the left side, and anything in parentheses means the contents of the memory indicated by the enclosed register, etc. For example,

```
r ← [.hl]
```

This means that the memory contents at the address indicated by the .hl register are assigned to an 8-bit register. Note that the [n] and [.c] in the I/O instructions refer to the corresponding I/O port numbers.

The "Flag" column shows the operation of each flag, and the "Opcode" column shows the machine code for each instruction, written in binary and hexadecimal. The "B" and "C" to the right of the column show the length of each instruction (number of bytes) and the number of clocks required to execute the instruction, respectively.

In addition, the abbreviations that appear in the instruction table are summarized in the following legend for your reference. Also, the mnemonics in the table are different from those in the Z80 because they are copyrighted by Zilog. However, with the exception of the multiplication instruction added in the R800 and instructions whose operation was not officially guaranteed in the Z80, the operation of all instructions is the same even though there are differences in mnemonics. Please program while comparing it with the Z80 instruction table.

.a{7}	The most significant bit of register .a
.a{4..7}	Bits 4-7 of register .a
;	Separation of actions
.de:.hl	32-bit integer, upper 16 bits go into .de and lower 16 bits go into .hl
[.ix+d]	The address indicated by the value obtained by adding .ix to the 8-bit signed displacement
C	carry flag
Z	zero flag
P/V	Parity overflow flag
S	sign flag
N	subtraction flag
H	half carry flag
•	flag does not change
↕	The flag changes depending on the execution result.
0	flag is 0
1	flag is 1
?	becomes indeterminate
V	used as overflow flag
P	used as parity flag
IFF	The value of the interrupt flip-flop is entered
r,r'	8 bit register 、 .a,.b,.c,.d,.e,.h,.l
u,u'	8 bit register 、 .a,.b,.c,.d,.e,.ixh,.ixl
v,v'	8 bit register 、 .a,.b,.c,.d,.e,.iyh,.iyl
p	8 bit register 、 .ixh,.ixl
q	8 bit register 、 .iyh,.iyl
ss	16 bit register 、 .bc,.de,.hl,.sp
pp	16 bit register 、 .bc,.de,.ix,.sp
rr	16 bit register 、 .bc,.de,.iy,.sp
qq	16 bit register 、 .bc,.de,.hl,.af
e	short br instruction jump address difference, 8-bit signed immediate value (+127~-128)
k	brk instruction destination address 、 00h,08h,10h,18h,20h,28h,30h,38h
nn	16-bit immediate value or absolute address
n	8-bit immediate value
b	A value indicating the bit number of the bit operation instruction
NOT	flip the bits
∨	OR the bits
∀	XOR the bits
∧	AND the bits
tmp	Save the value temporarily
B	number of bytes of instruction
C	The minimum number of clocks required to execute an instruction

When a branch or call instruction has two clock counts written, the upper one indicates when the condition is not met and the lower one indicates when the condition is met.

Also, when an I/O command has two clock counts, the top one means that the transfer has not yet finished, and the bottom one means that the transfer has finished.

The clock counts in the instruction table listed here are 1/4 of the XTAL oscillation frequency in SYSCLK terms. Also, these are the values when executed with no wait states. When executed on DRAM, wait states are automatically inserted by page breaks or refreshes.

A.2 8 bit move instruction

mnemonic	command operation	flags S Z H P_V N C	opcode 76543210	Hex	B	C
ld r,r'	r←r'	• • • • • •	01 r r'		1	1
ld r,n	r←n	• • • • • •	00 r 110 ← n →		2	2
ld r,[.hl]	r←[.hl]	• • • • • •	01 r 110		1	2
ld r,[.ix+d]	r←[.ix+d]	• • • • • •	11011101 01 r 110 ← d →	DD	3	5
ld r,[.iy+d]	r←[.iy+d]	• • • • • •	11111101 01 r 110 ← d →	FD	3	5
ld [.hl],r	[.hl]←r	• • • • • •	01110 r		1	2
ld [.ix+d],r	[.ix+d]←r	• • • • • •	11011101 01110 r ← d →	DD	3	5
ld [.iy+d],r	[.iy+d]←r	• • • • • •	11111101 01110 r ← d →	FD	3	5
ld u,u'	u←u'	• • • • • •	11011101 01 u u'	DD	2	2
ld v,v'	v←v'	• • • • • •	11111101 01 v v'	FD	2	2
ld u,n	u←n	• • • • • •	11011101 00 u 110 ← n →	DD	3	3
ld v,n	v←n	• • • • • •	11111101 00 v 110 ← n →	FD	3	3
ld [.hl],n	[.hl]←n	• • • • • •	00110110 ← n →	36	2	3
ld [.ix+d],n	[.ix+d]←n	• • • • • •	11011101 00110110 ← d → ← n →	DD 36	4	5
ld [.iy+d],n	[.iy+d]←n	• • • • • •	11111101 00110110 ← d → ← n →	FD 36	4	5

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
ld .a,.i	.a←.i	↕ ↕ 0 IFF 0 •	11101101 01010111	ED 57	2	2
ld .a,.r	.a←.r	↕ ↕ 0 IFF 0 •	11101101 01011111	ED 5F	2	2
ld .i,.a	.i←.a	• • • • • •	11101101 01000111	ED 47	2	2
ld .r,.a	.r←.a	• • • • • •	11101101 01001111	ED 4F	2	2
ld .a,[.bc]	.a←[.bc]	• • • • • •	00001010	0A	1	2
ld .a,[.de]	.a←[.de]	• • • • • •	00011010	1A	1	2
ld .a,[nn]	.a←[nn]	• • • • • •	00111010 ← nn_l → ← nn_h →	3A	3	4
ld [.bc],.a	[.bc]←.a	• • • • • •	00000010	02	1	2
ld [.de],.a	[.de]←.a	• • • • • •	00010010	12	1	2
ld [nn],.a	[nn]←.a	• • • • • •	00110010 ← nn_l → ← nn_h →	32	3	4

	000	001	010	011	100	101	110	111
r	.b	.c	.d	.e	.h	.l		.a
u	.b	.c	.d	.e	.ixh	.ixl		.a
v	.b	.c	.d	.e	.iyh	.iyl		.a

A.3 16 bit move instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
ld ss,nn	ss←nn	• • • • • •	00 ss 0001 ← nn_l → ← nn_h →		3	3
ld .ix,nn	.ix←nn	• • • • • •	11011101 00100001 ← nn_l → ← nn_h →	DD 21	4	4
ld .iy,nn	.iy←nn	• • • • • •	11111101 00100001 ← nn_l → ← nn_h →	FD 21	4	4
ld .sp,.hl	.sp←.hl	• • • • • •	11111001	F9	1	1
ld .sp,.ix	.sp←.ix	• • • • • •	11011101 11111001	DD F9	2	2
ld .sp,.iy	.sp←.iy	• • • • • •	11111101 11111001	FD F9	2	2

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
ld ss,[nn]	$ss_h \leftarrow [nn+1]$ $ss_l \leftarrow [nn]$	• • • • • •	11 101 101 01 ss 1 011 ← nn_l → ← nn_h →	ED	4	6
ld .hl,[nn]	.h←[nn+1] .l←[nn]	• • • • • •	00 101 010 ← nn_l → ← nn_h →	2A	3	5
ld .ix,[nn]	.ixh←[nn+1] .ixl←[nn]	• • • • • •	11 011 101 00 101 010 ← nn_l → ← nn_h →	DD 2A	4	6
ld .iy,[nn]	.iyh←[nn+1] .iyl←[nn]	• • • • • •	1 1 1 1 1 101 00 101 010 ← nn_l → ← nn_h →	FD 2A	4	6
ld [nn],ss	$[nn+1] \leftarrow ss_h$ $[nn] \leftarrow ss_l$	• • • • • •	11 101 101 01 ss 0 011 ← nn_l → ← nn_h →	ED	4	6
ld [nn],.hl	[nn+1]←.h [nn]←.l	• • • • • •	00 100 010 ← nn_l → ← nn_h →	22	3	5
ld [nn],.ix	[nn+1]←.ixh [nn]←.ixl	• • • • • •	11 011 101 00 100 010 ← nn_l → ← nn_h →	DD 22	4	6
ld [nn],.iy	[nn+1]←.iyh [nn]←.iyl	• • • • • •	1 1 1 1 1 101 00 100 010 ← nn_l → ← nn_h →	FD 22	4	6

	00	01	10	11
ss	.bc	.de	.hl	.sp

A.4 exchange instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76 543 210	Hex	B	C
xch .de,.hl	.de↔.hl	• • • • • •	11 101 011	EB	1	1
xch .af,.af'	.af↔.af'	↕ ↕ ↕ ↕ ↕ ↕	00001 000	08	1	1
xch [.sp],.hl	.l↔[.sp];.h↔[.sp+1]	• • • • • •	11 100 011	E3	1	5
xch [.sp],.ix	.ixl↔[.sp] .ixh↔[.sp+1]	• • • • • •	11 011 101 11 100 011	DD E3	2	6
xch [.sp],.iy	.iyl↔[.sp] .iyh↔[.sp+1]	• • • • • •	11 111 101 11 100 011	FD E3	2	6
xchx	.bc↔.bc';.de↔.de';.hl↔.hl'	• • • • • •	11 011 001	D9	1	1

A.5 stack manipulation instructions

mnemonic	command operation	flags S Z H P/V N C	opcode 76 543 210	Hex	B	C
push qq	[.sp−2]←qq_l;[.sp−1]←qq_h .sp←.sp−2	• • • • • •	11 qq 0 101		1	4
push .ix	[.sp−2]←.ixl;[.sp−1]←.ixh .sp←.sp−2	• • • • • •	11 011 101 11 100 101	DD E5	2	5
push .iy	[.sp−2]←.iyl;[.sp−1]←.iyh .sp←.sp−2	• • • • • •	11 111 101 11 100 101	FD E5	2	5
pop qq	qq_l←[.sp];qq_h←[.sp+1] .sp←.sp+2	• • • • • •	11 qq 0 001		1	3
pop .ix	.ixl←[.sp];.ixh←[.sp+1] .sp←.sp+2	• • • • • •	11 011 101 11 100 001	DD E1	2	4
pop .iy	.iyl←[.sp];.iyh←[.sp+1] .sp←.sp+2	• • • • • •	11 111 101 11 100 001	FD E1	2	4

	00	01	10	11
qq	.bc	.de	.hl	.af

When pop.af is selected, all flags are changed.

A.6 block transfer instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
move [.hl++],[.de++]	[.de]←[.hl];.de←.de+1 .hl←.hl+1;.bc←.bc−1	• • 0 ↕ 0 • *1	11101101 10100000	ED A0	2	4
move [.hl−−],[.de−−]	[.de]←[.hl];.de←.de−1 .hl←.hl−1;.bc←.bc−1	• • 0 ↕ 0 • *1	11101101 10101000	ED A8	2	4
movem [.hl++],[.de++]	repeat;[.de]←[.hl];.de←.de+1 .hl←.hl+1;.bc←.bc−1;until .bc=0	• • 0 0 0 •	11101101 10110000	ED B0	2	4
movem [.hl−−],[.de−−]	repeat;[.de]←[.hl];.de←.de−1 .hl←.hl−1;.bc←.bc−1;until .bc=0	• • 0 0 0 •	11101101 10111000	ED B8	2	4

*1.bc−1=0 if 0, 1 otherwise

A.7 block search instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
cmp .a,[.hl++]	.a−[.hl];.hl←.hl+1 .bc←.bc−1	↕ ↕ ↕ ↕ 1 • *2 *1	11101101 10100001	ED A1	2	4
cmp .a,[.hl−−]	.a−[.hl];.hl←.hl−1 .bc←.bc−1	↕ ↕ ↕ ↕ 1 • *2 *1	11101101 10101001	ED A9	2	4
cmpm .a,[.hl++]	repeat;.a−[.hl];.hl←.hl+1 .bc←.bc−1;until .bc=0 or .a=[.hl]	↕ ↕ ↕ ↕ 1 • *2 *1	11101101 10110001	ED B1	2	5
cmpm .a,[.hl−−]	repeat;.a−[.hl];.hl←.hl−1 .bc←.bc−1;until .bc=0 or .a=[.hl]	↕ ↕ ↕ ↕ 1 • *2 *1	11101101 10111001	ED B9	2	5

*1. 0 when bc-1=0, 1 otherwise

*2. 1 if a=[.hl], 0 otherwise

A.8 multiplication instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
mulub .a,r	.hl←.a*r	0 ↕ • 0 • ↕	11101101 11 r 001	ED	2	14
muluw .hl,ss	.de:.hl←.hl*ss	0 ↕ • 0 • ↕	11101101 11 ss 0011	ED	2	36

In mulub, operation is not guaranteed when r is not .b, .c, .d..e.

In muluw, operation is not guaranteed unless ss is .bc or .sp.

A.9 addition instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
add .a,r	.a←.a+r	↕ ↕ ↕ V 0 ↕	10000 r		1	1
add .a,p	.a←.a+p	↕ ↕ ↕ V 0 ↕	11011101 10000 p	DD	2	2
add .a,q	.a←.a+q	↕ ↕ ↕ V 0 ↕	11111101 10000 q	FD	2	2
add .a,[.hl]	.a←.a+[.hl]	↕ ↕ ↕ V 0 ↕	10000110	86	1	2
add .a,[.ix+d]	.a←.a+[.ix+d]	↕ ↕ ↕ V 0 ↕	11011101 10000110 ← d →	DD 86	3	5
add .a,[.iy+d]	.a←.a+[.iy+d]	↕ ↕ ↕ V 0 ↕	11111101 10000110 ← d →	FD 86	3	5
add .a,n	.a←.a+n	↕ ↕ ↕ V 0 ↕	11000110 ← n →	C6	2	2
addc .a,r	.a←.a+r+C	↕ ↕ ↕ V 0 ↕	10001 r		1	1
addc .a,p	.a←.a+p+C	↕ ↕ ↕ V 0 ↕	11011101 10001 p	DD	2	2
addc .a,q	.a←.a+q+C	↕ ↕ ↕ V 0 ↕	11111101 10001 q	FD	2	2
addc .a,[.hl]	.a←.a+[.hl]+C	↕ ↕ ↕ V 0 ↕	10001110	8E	1	2
addc .a,[.ix+d]	.a←.a+[.ix+d]+C	↕ ↕ ↕ V 0 ↕	11011101 10001110 ← d →	DD 8E	3	5
addc .a,[.iy+d]	.a←.a+[.iy+d]+C	↕ ↕ ↕ V 0 ↕	11111101 10001110 ← d →	FD 8E	3	5
addc .a,n	.a←.a+n+C	↕ ↕ ↕ V 0 ↕	11001110 ← n →	CE	2	2
addc .hl,ss	.hl←.hl+ss+C	↕ ↕ ? V 0 ↕	11101101 01 ss 1010	ED	2	2
add .hl,ss	.hl←.hl+ss	• • ? • 0 ↕	00 ss 1001		1	1
add .ix,pp	.ix←.ix+pp	• • ? • 0 ↕	11011101 00 pp 1001	DD	2	2
add .iy,rr	.iy←.iy+rr	• • ? • 0 ↕	11111101 00 rr 1001	FD	2	2

mnemonic	command operation	flags S Z H P/V N C	opcode 76 543 210	Hex	B	C
inc r	r←r+1	↕ ↕ ↕ V 0 •	00 r 100		1	1
inc p	p←p+1	↕ ↕ ↕ V 0 •	11 011 101 00 p 100	DD	2	2
inc q	q←q+1	↕ ↕ ↕ V 0 •	11 111 101 00 q 100	FD	2	2
inc [.hl]	[.hl]←[.hl]+1	↕ ↕ ↕ V 0 •	00 110 100	34	1	4
inc [.ix+d]	[.ix+d]←[.ix+d]+1	↕ ↕ ↕ V 0 •	11 011 101 00 110 100 ← d →	DD 34	3	7
inc [.iy+d]	[.iy+d]←[.iy+d]+1	↕ ↕ ↕ V 0 •	11 111 101 00 110 100 ← d →	FD 34	3	7
inc ss	ss←ss+1	• • • • • •	00 ss 0011		1	1
inc .ix	.ix←.ix+1	• • • • • •	11 011 101 00 100 011	DD 23	2	2
inc .iy	.iy←.iy+1	• • • • • •	11 111 101 00 100 011	FD 23	2	2

	00	01	10	11
ss	.bc	.de	.hl	.sp
pp	.bc	.de	.ix	.sp
rr	.bc	.de	.iy	.sp

	000	001	010	011	100	101	110	111
p					.ixh	.ixl		
q					.iyh	.iyl		

A.10 subtraction instruction

mnemonic	command operation	flags S Z H P_V N C	opcode 76 543 210	Hex	B	C
sub .a,r	.a←.a−r	↕ ↕ ↕ V 1 ↕	10010 r		1	1
sub .a,p	.a←.a−p	↕ ↕ ↕ V 1 ↕	11011101 10010 p	DD	2	2
sub .a,q	.a←.a−q	↕ ↕ ↕ V 1 ↕	11111101 10010 q	FD	2	2
sub .a,[.hl]	.a←.a−[.hl]	↕ ↕ ↕ V 1 ↕	10010110	96	1	2
sub .a,[.ix+d]	.a←.a−[.ix+d]	↕ ↕ ↕ V 1 ↕	11011101 10010110 ← d →	DD 96	3	5
sub .a,[.iy+d]	.a←.a−[.iy+d]	↕ ↕ ↕ V 1 ↕	11111101 10010110 ← d →	FD 96	3	5
sub .a,n	.a←.a−n	↕ ↕ ↕ V 1 ↕	11010110 ← n →	D6	2	2
subc.a,r	.a←.a−r−C	↕ ↕ ↕ V 1 ↕	10011 r		1	1
subc.a,p	.a←.a−p−C	↕ ↕ ↕ V 1 ↕	11011101 10011 p	DD	2	2
subc.a,q	.a←.a−q−C	↕ ↕ ↕ V 1 ↕	11111101 10011 q	FD	2	2
subc.a,[.hl]	.a←.a−[.hl]−C	↕ ↕ ↕ V 1 ↕	10011110	9E	1	2
subc.a,[.ix+d]	.a←.a−[.ix+d]−C	↕ ↕ ↕ V 1 ↕	11011101 10011110 ← d →	DD 9E	3	5
subc.a,[.iy+d]	.a←.a−[.iy+d]−C	↕ ↕ ↕ V 1 ↕	11111101 10011110 ← d →	FD 9E	3	5
subc.a,n	.a←.a−n−C	↕ ↕ ↕ V 1 ↕	11011110 ← n →	DE	2	2
subc.hl,ss	.hl←.hl−ss−C	↕ ↕ ? V 1 ↕	11101101 01 ss 0010	ED	2	2

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
dec r	r←r−1	↕ ↕ ↕ V 1 •	00 r 101		1	1
dec p	p←p−1	↕ ↕ ↕ V 1 •	11011101 00 p 101	DD	2	2
dec q	q←q−1	↕ ↕ ↕ V 1 •	11111101 00 q 101	FD	2	2
dec [.hl]	[.hl]←[.hl]−1	↕ ↕ ↕ V 1 •	00110101	35	1	4
dec [.ix+d]	[.ix+d]←[.ix+d]−1	↕ ↕ ↕ V 1 •	11011101 00110101 ← d →	DD 35	3	7
dec [.iy+d]	[.iy+d]←[.iy+d]−1	↕ ↕ ↕ V 1 •	11111101 00110101 ← d →	FD 35	3	7
dec ss	ss←ss−1	• • • • • •	00 ss 1011		1	1
dec .ix	.ix←.ix−1	• • • • • •	11011101 00101011	DD 2B	2	2
dec .iy	.iy←.iy−1	• • • • • •	11111101 00101011	FD 2B	2	2

A.11 comparison command

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
cmp .a,r	.a−r	↕ ↕ ↕ V 1 ↕	10111 r		1	1
cmp .a,p	.a−p	↕ ↕ ↕ V 1 ↕	11011101 10111 p	DD	2	2
cmp .a,q	.a−q	↕ ↕ ↕ V 1 ↕	11111101 10111 q	FD	2	2
cmp .a,[.hl]	.a−[.hl]	↕ ↕ ↕ V 1 ↕	10111110	BE	1	2
cmp .a,[.ix+d]	.a−[.ix+d]	↕ ↕ ↕ V 1 ↕	11011101 10111110 ← d →	DD BE	3	5
cmp .a,[.iy+d]	.a−[.iy+d]	↕ ↕ ↕ V 1 ↕	11111101 10111110 ← d →	FD BE	3	5
cmp .a,n	.a−n	↕ ↕ ↕ V 1 ↕	11111110 ← n →	FE	2	2

A.12 logical operation instructions

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
and .a,r	.a←.a∧r	↕ ↕ 1 P 0 0	10100 r		1	1
and .a,p	.a←.a∧p	↕ ↕ 1 P 0 0	11011101 10100 p	DD	2	2
and .a,q	.a←.a∧q	↕ ↕ 1 P 0 0	11111101 10100 q	FD	2	2
and .a,[.hl]	.a←.a∧[.hl]	↕ ↕ 1 P 0 0	10100110	A6	1	2
and .a,[.ix+d]	.a←.a∧[.ix+d]	↕ ↕ 1 P 0 0	11011101 10100110 ← d →	DD A6	3	5
and .a,[.iy+d]	.a←.a∧[.iy+d]	↕ ↕ 1 P 0 0	11111101 10100110 ← d →	FD A6	3	5
and .a,n	.a←.a∧n	↕ ↕ 1 P 0 0	11100110 ← n →	E6	2	2
or .a,r	.a←.a∨r	↕ ↕ 0 P 0 0	10110 r		1	1
or .a,p	.a←.a∨p	↕ ↕ 0 P 0 0	11011101 10110 p	DD	2	2
or .a,q	.a←.a∨q	↕ ↕ 0 P 0 0	11111101 10110 q	FD	2	2
or .a,[.hl]	.a←.a∨[.hl]	↕ ↕ 0 P 0 0	10110110	B6	1	2
or .a,[.ix+d]	.a←.a∨[.ix+d]	↕ ↕ 0 P 0 0	11011101 10110110 ← d →	DD B6	3	5
or .a,[.iy+d]	.a←.a∨[.iy+d]	↕ ↕ 0 P 0 0	11111101 10110110 ← d →	FD B6	3	5
or .a,n	.a←.a∨n	↕ ↕ 0 P 0 0	11110110 ← n →	F6	2	2
xor .a,r	.a←.a∀r	↕ ↕ 0 P 0 0	10101 r		1	1
xor .a,p	.a←.a∀p	↕ ↕ 0 P 0 0	11011101 10101 p	DD	2	2
xor .a,q	.a←.a∀q	↕ ↕ 0 P 0 0	11111101 10101 q	FD	2	2
xor .a,[.hl]	.a←.a∀[.hl]	↕ ↕ 0 P 0 0	10101110	AE	1	2
xor .a,[.ix+d]	.a←.a∀[.ix+d]	↕ ↕ 0 P 0 0	11011101 10101110 ← d →	DD AE	3	5
xor .a,[.iy+d]	.a←.a∀[.iy+d]	↕ ↕ 0 P 0 0	11111101 10101110 ← d →	FD AE	3	5
xor .a,n	.a←.a∀n	↕ ↕ 0 P 0 0	11101110 ← n →	EE	2	2

A.13 bit manipulation instructions

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
bit b,r	z←NOT r{b}	? ↕ 1 ? 0 •	11001011 01 b r	CB	2	2
bit b,[.hl]	z←NOT [.hl]{b}	? ↕ 1 ? 0 •	11001011 01 b 110	CB	2	3
bit b,[.ix+d]	z←NOT [.ix+d]{b}	? ↕ 1 ? 0 •	11011101 11001011 ← d → 01 b 110	DD CB	4	5
bit b,[.iy+d]	z←NOT [.iy+d]{b}	? ↕ 1 ? 0 •	11111101 11001011 ← d → 01 b 110	FD CB	4	5
set b,r	r{b}←1	• • • • • •	11001011 11 b r	CB	2	2
set b,[.hl]	[.hl]{b}←1	• • • • • •	11001011 11 b 110	CB	2	5
set b,[.ix+d]	[.ix+d]{b}←1	• • • • • •	11011101 11001011 ← d → 11 b 110	DD CB	4	7
set b,[.iy+d]	[.iy+d]{b}←1	• • • • • •	11111101 11001011 ← d → 11 b 110	FD CB	4	7
clr b,r	r{b}←0	• • • • • •	11001011 10 b r	CB	2	2
clr b,[.hl]	[.hl]{b}←0	• • • • • •	11001011 10 b 110	CB	2	5
clr b,[.ix+d]	[.ix+d]{b}←0	• • • • • •	11011101 11001011 ← d → 10 b 110	DD CB	4	7
clr b,[.iy+d]	[.iy+d]{b}←0	• • • • • •	11111101 11001011 ← d → 10 b 110	FD CB	4	7

A.14 Rotate instruction

mnemonic	command operation	flags S Z H P_V N C	opcode 76543210	Hex	B	C
rola	C←.a{7};.a←.a*2;.a{0}←C	• • 0 • 0 ↕	00000111	07	1	1
rora	C←.a{0};.a←.a/2;.a{7}←C	• • 0 • 0 ↕	00001111	0F	1	1
rolca	tmp←C;C←.a{7};.a←.a*2;.a{0}←tmp	• • 0 • 0 ↕	00010111	17	1	1
rorca	tmp←C;C←.a{0};.a←.a/2;.a{7}←tmp	• • 0 • 0 ↕	00011111	1F	1	1
rol r	C←r{7} r←r*2;r{0}←C	↕ ↕ 0 P 0 ↕	11001011 00000 r	CB	2	2
rol [.hl]	C←[.hl]{7} [.hl]←[.hl]*2;[.hl]{0}←C	↕ ↕ 0 P 0 ↕	11001011 00000110	CB 06	2	5
rol [.ix+d]	C←[.ix+d]{7} [.ix+d]←[.ix+d]*2 [.ix+d]{0}←C	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00000110	DD CB 06	4	7
rol [.iy+d]	C←[.iy+d]{7} [.iy+d]←[.iy+d]*2 [.iy+d]{0}←C	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00000110	FD CB 06	4	7
ror r	C←r{0} r←r/2;r{7}←C	↕ ↕ 0 P 0 ↕	11001011 00001 r	CB	2	2
ror [.hl]	C←[.hl]{0} [.hl]←[.hl]/2;[.hl]{7}←C	↕ ↕ 0 P 0 ↕	11001011 00001110	CB 0E	2	5
ror [.ix+d]	C←[.ix+d]{0} [.ix+d]←[.ix+d]/2 [.ix+d]{7}←C	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00001110	DD CB 0E	4	7
ror [.iy+d]	C←[.iy+d]{0} [.iy+d]←[.iy+d]/2 [.iy+d]{7}←C	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00001110	FD CB 0E	4	7

mnemonic	command operation	flags S Z H P/V N C	opcode 76 543 210	Hex	B	C
rolc r	tmp←C;C←r{7} r←r*2;r{0}←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 010 r	CB	2	2
rolc [.hl]	tmp←C;C←[.hl]{7} [.hl]←[.hl]*2;[.hl]{0}←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 010 110	CB 16	2	5
rolc [.ix+d]	tmp←C C←[.ix+d]{7} [.ix+d]←[.ix+d]*2 [.ix+d]{0}←tmp	↕ ↕ 0 P 0 ↕	11 011 101 11 001 011 ← d → 00 010 110	DD CB 16	4	7
rolc [.iy+d]	tmp←C C←[.iy+d]{7} [.iy+d]←[.iy+d]*2 [.iy+d]{0}←tmp	↕ ↕ 0 P 0 ↕	11 111 101 11 001 011 ← d → 00 010 110	FD CB 16	4	7
rorc r	tmp←C;C←r{0} r←r/2;r{7}←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 011 r	CB	2	2
rorc [.hl]	tmp←C;C←[.hl]{0} [.hl]←[.hl]/2;[.hl]{7}←tmp	↕ ↕ 0 P 0 ↕	11 001 011 00 011 110	CB 1E	2	5
rorc [.ix+d]	tmp←C C←[.ix+d]{0} [.ix+d]←[.ix+d]/2 [.ix+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11 011 101 11 001 011 ← d → 00 011 110	DD CB 1E	4	7
rorc [.iy+d]	tmp←C C←[.iy+d]{0} [.iy+d]←[.iy+d]/2 [.iy+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11 111 101 11 001 011 ← d → 00 011 110	FD CB 1E	4	7
rol4 [.hl]	tmp←.a{0..3};.a{0..3}←[.hl]{4..7} [.hl]{4..7}←[.hl]{0..3};[.hl]{0..3}←tmp	↕ ↕ 0 P 0 •	11 101 101 11 101 111	ED 6F	2	5
ror4 [.hl]	tmp←.a{0..3};.a{0..3}←[.hl]{0..3} [.hl]{0..3}←[.hl]{4..7};[.hl]{4..7}←tmp	↕ ↕ 0 P 0 •	11 101 101 11 100 111	ED 67	2	5

A.15 shift instruction

mnemonic	command operation	flags S Z H $P_{/V}$ N C	opcode 76543210	Hex	B	C
shl r shla	C←r{7} r←r*2	↕ ↕ 0 P 0 ↕	11001011 00100 r	CB	2	2
shl [.hl] shla	C←[.hl]{7} [.hl]←[.hl]*2	↕ ↕ 0 P 0 ↕	11001011 00100110	CB 26	2	5
shl [.ix+d] shla	C←[.ix+d]{7} [.ix+d]←[.ix+d]*2	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00100110	DD CB 26	4	7
shl [.iy+d] shla	C←[.iy+d]{7} [.iy+d]←[.iy+d]*2	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00100110	FD CB 26	4	7
shr r	C←r{0} r←r/2	↕ ↕ 0 P 0 ↕	11001011 00111 r	CB	2	2
shr [.hl]	C←[.hl]{0} [.hl]←[.hl]/2	↕ ↕ 0 P 0 ↕	11001011 00111110	CB 3E	2	5
shr [.ix+d]	C←[.ix+d]{0} [.ix+d]←[.ix+d]/2	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00111110	DD CB 3E	4	7
shr [.iy+d]	C←[.iy+d]{0} [.iy+d]←[.iy+d]/2	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00111110	FD CB 3E	4	7
shrar	tmp←r{7};C←r{0} r←r/2;r{7}←tmp	↕ ↕ 0 P 0 ↕	11001011 00101 r	CB	2	2
shra [.hl]	tmp←[.hl]{7};C←[.hl]{0} [.hl]←[.hl]/2;[.hl]{7}←tmp	↕ ↕ 0 P 0 ↕	11001011 00101110	CB 2E	2	5
shra [.ix+d]	tmp←[.ix+d]{7} C←[.ix+d]{0} [.ix+d]←[.ix+d]/2 [.ix+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11011101 11001011 ← d → 00101110	DD CB 2E	4	7
shra [.iy+d]	tmp←[.iy+d]{7} C←[.iy+d]{0} [.iy+d]←[.iy+d]/2 [.iy+d]{7}←tmp	↕ ↕ 0 P 0 ↕	11111101 11001011 ← d → 00101110	FD CB 2E	4	7

The shl and shla instructions are exactly the same, so the operands are the same.

A.16 branch instruction

mnemonic	command operation	flags S Z H P_V N C	opcode 76543210	Hex	B	C
br nn	.pc←nn	• • • • • •	11000011 ← nn_l → ← nn_h →	C3	3	3
bnz nn	if z=0 .pc←nn	• • • • • •	11000010 ← nn_l → ← nn_h →	C2	3	3
bz nn	if z=1 .pc←nn	• • • • • •	11001010 ← nn_l → ← nn_h →	CA	3	3
bnc nn	if c=0 .pc←nn	• • • • • •	11010010 ← nn_l → ← nn_h →	D2	3	3
bc nn	if c=1 .pc←nn	• • • • • •	11011010 ← nn_l → ← nn_h →	DA	3	3
bpo nn	if P_V=0 .pc←nn	• • • • • •	11100010 ← nn_l → ← nn_h →	E2	3	3
bpe nn	if P_V=1 .pc←nn	• • • • • •	11101010 ← nn_l → ← nn_h →	EA	3	3
bp nn	if s=0 .pc←nn	• • • • • •	11110010 ← nn_l → ← nn_h →	F2	3	3
bm nn	if s=1 .pc←nn	• • • • • •	11111010 ← nn_l → ← nn_h →	FA	3	3
br [.hl]	.pc←[.hl]	• • • • • •	11101001	E9	1	1
br [.ix]	.pc←[.ix]	• • • • • •	11011101 11101001	DD E9	2	2
br [.iy]	.pc←[.iy]	• • • • • •	11111101 11101001	FD E9	2	2

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
short br e	.pc←.pc+e	• • • • • •	00011000 ← e−2 →	18	2	3
short bnz e	if z=0 .pc←.pc+e	• • • • • •	00100000 ← e−2 →	20	2	2 3
short bz e	if z=1 .pc←.pc+e	• • • • • •	00101000 ← e−2 →	28	2	2 3
short bnc e	if c=0 .pc←.pc+e	• • • • • •	00110000 ← e−2 →	30	2	2 3
short bc e	if c=1 .pc←.pc+e	• • • • • •	00111000 ← e−2 →	38	2	2 3
dbnz e	.b←.b−1;if .b≠0 .pc←.pc+e	• • • • • •	00010000 ← e−2 →	10	2	2

A.17 call instruction

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
call nn	[.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11001101 ← nn_l → ← nn_h →	CD	3	5
call nz,nn	if z=0 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11000100 ← nn_l → ← nn_h →	C4	3	3 5
call z,nn	if z=1 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11001100 ← nn_l → ← nn_h →	CC	3	3 5
call nc,nn	if c=0 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11010100 ← nn_l → ← nn_h →	D4	3	3 5
call c,nn	if c=1 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11011100 ← nn_l → ← nn_h →	DC	3	3 5
call po,nn	if P/V=0 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11100100 ← nn_l → ← nn_h →	E4	3	3 5
call pe,nn	if P/V=1 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11101100 ← nn_l → ← nn_h →	EC	3	3 5
call p,nn	if s=0 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11110100 ← nn_l → ← nn_h →	F4	3	3 5
call m,nn	if s=1 [.sp−2]←.pc_l;[.sp−1]←.pc_h .sp←.sp−2;.pc←nn	• • • • • •	11111100 ← nn_l → ← nn_h →	FC	3	3 5

mnemonic	command operation	flags S Z H $P_{/V}$ N C	opcode 76543210	Hex	B	C
ret	$.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11001001	C9	1	3
ret nz	if z=0 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11000000	C0	1	1 3
ret z	if z=1 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11001000	C8	1	1 3
ret nc	if c=0 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11010000	D0	1	1 3
ret c	if c=1 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11011000	D8	1	1 3
ret po	if $P_{/V}$=0 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11100000	E0	1	1 3
ret pe	if $P_{/V}$=1 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11101000	E8	1	1 3
ret p	if s=0 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11110000	F0	1	1 3
ret m	if s=1 $.pc_l \leftarrow [.sp]; .pc_h \leftarrow [.sp+1]; .sp \leftarrow .sp+2$	• • • • • •	11111000	F8	1	1 3
reti	interrupt return	• • • • • •	11101101 01001101	ED 4D	2	5
retn	Non Maskable Interrupt return	• • • • • •	11101101 01000101	ED 45	2	5
brk k	$[.sp-2] \leftarrow .pc_l; [.sp-1] \leftarrow .pc_h$ $.sp \leftarrow .sp-2; .pc_l \leftarrow k; .pc_h \leftarrow 0$	• • • • • •	11 k/8 111		1	4

A.18 input/output instructions

mnemonic	command operation	flags S Z H P/V N C	opcode 76543210	Hex	B	C
in .a,[n]	.a←[n]	• • • • • •	11011011 ← n →	DB	2	3
in r,[.c]	r←[.c]	↕ ↕ 0 P 0 •	11101101 01 r 000	ED	2	3
in .f,[.c]	[.c]	↕ ↕ 0 P 0 •	11101101 01110000	ED 70	2	3
in [.hl++],[.c]	[.hl]←[.c];.b←.b−1 .hl←.hl+1	? ↕ ? ? 1 • *1	11101101 10100010	ED A2	2	4
in [.hl−−],[.c]	[.hl]←[.c];.b←.b−1 .hl←.hl−1	? ↕ ? ? 1 • *1	11101101 10101010	ED AA	2	4
inm [.hl++],[.c]	repeat;[.hl]←[.c];.b←.b−1 .hl←.hl+1;until .b=0	? 1 ? ? 1 •	11101101 10110010	ED B2	2	4 3
inm [.hl−−],[.c]	repeat;[.hl]←[.c];.b←.b−1 .hl←.hl−1;until .b=0	? 1 ? ? 1 •	11101101 10111010	ED BA	2	4 3
out [n],.a	[n]←.a	• • • • • •	11010011 ← n →	D3	2	3
out [.c],r	[.c]←r	• • • • • •	11101101 01 r 001	ED	2	3
out [.c],[.hl++]	[.c]←[.hl];.b←.b−1 .hl←.hl+1	? ↕ ? ? 1 • *1	11101101 10100011	ED A3	2	4
out [.c],[.hl−−]	[.c]←[.hl];.b←.b−1 .hl←.hl−1	? ↕ ? ? 1 • *1	11101101 10101011	ED AB	2	4
outm [.c],[.hl++]	repeat;[.c]←[.hl];.b←.b−1 .hl←.hl+1;until .b=0	? 1 ? ? 1 •	11101101 10110011	ED B3	2	4 3
outm [.c],[.hl−−]	repeat;[.c]←[.hl];.b←.b−1 .hl←.hl−1;until .b=0	? 1 ? ? 1 •	11101101 10111011	ED BB	2	4 3

*1.1 when b−1=0, 0 otherwise

in .f,[.c] only changes the flags according to the contents of the port indicated by the .c register, and the contents are not stored anywhere.

A.19 CPU control instructions

mnemonic	command operation	flags S Z H P/V N C	opcode 76 543 210	Hex	B	C
adj .a	adjust to decimal	↕ ↕ ↕ P • ↕	00 100 111	27	1	1
not .a	.a←NOT .a	• • 1 • 1 •	00 101 111	2F	1	1
neg .a	.a←NOT .a+1	↕ ↕ ↕ V 1 ↕	11 101 101 01 000 100	ED 44	2	2
notc	C←NOT C	• • ? • 0 ↕	00 111 111	3F	1	1
setc	C←1	• • 0 • 0 1	00 110 111	37	1	1
nop	NO operation	• • • • • •	00 000 000	00	1	1
halt	HALT	• • • • • •	01 110 110	76	1	2
di	IFF←0	• • • • • •	11 110 011	F3	1	2
ei	IFF←1	• • • • • •	11 111 011	FB	1	1
im 0	interrupt mode 0	• • • • • •	11 101 101 01 000 110	ED 46	2	3
im 1	interrupt mode 1	• • • • • •	11 101 101 01 010 110	ED 56	2	3
im 2	interrupt mode 2	• • • • • •	11 101 101 01 011 110	ED 5E	2	3

index

ア

I/O port 57
access time 22
address 48
address bus 48
interlace 111
Wait function 95
ADSR 133
SECAM 111
SRAM 50
NTSC 110
FM–BIOS 139
MSX–Engine 17
MSX–JE 72
MSX–MUSIC 131
MSX types 62
envelope 133
OPLL YM2413 131
OPLL driver 156
operator 131
scale noise 133

カ

MSX made for export overseas 62
Expansion BIOS 62
Kanji ROM expansion 21

Kanji graphic mode 77
Kanji text mode 77
kanji driver 74
perfect equal temperament 134
brightness 99
carrier operator 137
Kilo (K) 48
Square wave 132
command register 92
control register 92

サ

sub ROM 52
sampling synthesizer 130
CPU 48
Hue 99
JIS code 74
system timer 20, 27
System work area 61
Shift JIS code 74
hexadecimal 48
main memory 50
Vertical retrace interrupt 114
horizontal resolution 111
status register 92
superimpose 113
slot expander 55
sine wave 132
full-width characters 74
software stack 78

タ

timer interrupt 114
Single kanji conversion 72
TAND 106
D/A converter 20

DRAM page access 24
DRAM mode 22
TC9769 17
Is there a disk? 62
disc work area 61
data bus 48
sync signal 110

ナ

binary number 48
sawtooth wave 133
non-interlaced 111

ハ

BIOS 57
byte 48
PAL 111
half-width characters 74
street address 48
PSG 130
bit 48
Bit image printing 79
Video RAM (VRAM) 50
Video RAM capacity 62
video digitization 113
VDP commands 95
VDP mode 82
hook 121
Programmable Sound Generator 130
Basic Input Output System 57
page 51
pause key control 20

マ

microsoft kanji code 74
multi scan monitor 111
undefined instruction 98

main ROM 52
main memory 50
memory mapper 22
moire 113
modulator operator 137

ヤ

USR function 119

ラ

RAM 50
rhythm sound 137
reset status 22
Continuous clause conversion 72
ROM 50

References

[1] ASCII Microsoft FE Headquarters, Nippon Musical Instruments Co., Ltd., “V9938 MSX-"VIDEO Technical Data Book", ASCII, 1985 (out of print)

[2] ASCII Corporation, "V9958 Specifications", not for sale, 1988

[3] ASCII Microsoft FE supervision, "MSX2 Technical Handbook", ASCII, 1986

[4] Seiichi Sugitani, "Powerful Use of MSX2+", ASCII, 1989

[5] ASCII Corporation, "MSX-Datapack", ASCII, 1991

Author biography

Naota Ishikawa

Naota Ishikawa

After graduating from Yokohama National University, he joined ASCII and was involved in the development of MSX. He then graduated from the Tokyo University of Science, Faculty of Science, Second Division, Mathematics Department, and the Graduate School of Science Research Department, Master's Program. He is currently enrolled in a doctoral program at the Keio University Graduate School of Science and Technology Research Department. He is also the author of a series of technical articles that have been serialized in MSX Magazine. naota@slab.sfc.keio.ac.jp

MSX turbo R Technical Handbook

First published on July 31, 1991

List price: 2,500 yen (2,427 yen)

Author Naota Ishikawa

Publisher Keiichiro Tsukamoto

Editor: Fumitaka Kojima

Publisher: ASCII Corporation

Three F Minamiaoyama Building, 6-11-1 Minamiaoyama, Minato-ku, Tokyo 107-24

Transfer Tokyo 4-161144

Representative (03)3486-7111

Publishing Sales Department: (03)3486-1977 (dial-in)

This document is protected by copyright law. Copying or duplicating this document, in whole or in part (including software and programs), in any manner whatsoever, without written consent from ASCII Corporation is prohibited.

Produced by Tokyo Shoseki Printing Co., Ltd.

Printing Tokyo Shoseki Printing Co., Ltd.

Edited by MSX Magazine Editorial Department

ISBN4-7561-0621-8 **Printed in Japan**

• 11619

Content list

- MSX turbo R system configuration
- R800 programming techniques
- List of BASIC commands added, deleted, and changed
- List of BIOS Add/Remove/Change Extensions
- How to use PCM recording and playback functions
- MSX2+ and MSX turbo R slots
- Kanji BASIC Overview
- V9958VDP Specifications
- Natural image display mode details
- Scanline interrupt practice
- MSX-MUSIC Control
- BASIC and BIOS sound data list
- R800CPU Instruction Code Table

ISBN4-7561-0621-8 C3055 P2500E

定価2,500円(本体2,427円)

MSX turbo R Technical handbook