

# MSX Memory Architecture

## Index

1. Introduction.....	4
1.1. Terminology.....	4
2. Slot.....	6
2.1. Basic Slot.....	6
2.2. Expansion slots.....	9
2.3. Slot-related BIOS routines.....	11
RDSLت (000Ch).....	11
WRSLT (0014h).....	12
CALSLت (001Ch).....	13
ENASLT (0024h).....	15
CALLF (0030h).....	16
2.4. Relationship between basic slots and expansion slots.....	17
3. メモリーマッパー対応 RAM.....	19
3.1. メモリーマッパーサポートルーチン.....	25
プライマリーマッパーとセカンダリーマッパー.....	25
マッパーサポートルーチンの使い方.....	26
ALL_SEG.....	29
FRE_SEG.....	30
RD_SEG.....	31
WR_SEG.....	32
CAL_SEG.....	33
CALLS.....	34
PUT_PH.....	35
GET_PH.....	36
PUT_P0.....	37
GET_P0.....	37
PUT_P1.....	38
GET_P1.....	38
PUT_P2.....	39
GET_P2.....	40
PUT_P3.....	40
GET_P3.....	41
メモリーマッパー情報テーブル.....	42
4. メガ ROM.....	48
4.1. ASCII-8K タイプ.....	49
4.2. ASCII-16K タイプ.....	51
4.3. Konami-8K タイプ.....	54
4.4. Konami SCC タイプ.....	55
4.5. Konami SCC-I タイプ.....	57
4.6. Panasonic タイプ.....	61
4.7. パナアミューズメントカートリッジ.....	65
5. ROM カートリッジの動作.....	67

## MSX Memory Architecture

5.1. 16KB の ROM カートリッジプログラムを作る.....	68
5.2. メガ ROM カートリッジプログラムを作る.....	73
5.3. 自身のスロットの検出.....	78
5.4. Page2/Page3 のスロット検出.....	84
5.5. Page0/Page3 のスロット切り替え.....	89
6. ハードウェア.....	98
7. その他の補足事項.....	99
7.1. バージョンアップアダプター利用時の EXPTBL.....	99
7.2. 拡張スロット選択レジスタは無条件に書いてはダメ.....	100
付録.....	105
付録 1. サンプルプログラムから利用している各種ソース.....	105

## 1. Introduction

This book is a summary of the ROM/RAM of the MSX. I decided to write this book because there are many different ROM/RAMs in the MSX and I have not seen any material describing them all in one place.

For the MSX alone, there is a wide range of memory space switching between basic and expansion slots, mapper segment switching using memory mapper-compatible RAM, and Panasonic-specific megarom bank switching for MSX2+ and later Panasonic machines. In addition to this, the mega-ROMs installed in the cartridge slots (ASCII-8K, ASCII-16K, Konami-8K, Konami-SCC, Konami-SCC-I) and Panamusement cartridges are also described. I have tried to cover all the major ones. Minor items are omitted because they are already difficult to obtain or there is no information to control them.

I hope this will be helpful for those who are going to create software for MSX.

Please note that there are some differences from the Zilog mnemonic. ZMA is a free software distributed at the following site.

ZMA distribution site

<https://github.com/hra1129/zma>

### 1.1. Terminology

MSX memory (slot, memory mapper, megarom) is managed by dividing the physical ROM and RAM address space into 16KB or 8KB units. In the Z80 memory space, these "16KB or 8KB areas" appear and are accessed. The address range in the Z80 memory space where the "16KB unit or 8KB unit area" appears, as well as the name of the "16KB unit or 8KB unit area" itself, is somewhat ambiguous, and may be described with a unique name depending on the document referred to. For example, we have confirmed that there are some documents that describe the "16 KB unit or 8 KB unit area" as a segment and others that describe it as a bank. In Document A, both the "16 KB unit area" of the memory mapper and the "16 KB unit or 8 KB unit area" of the megarom were described as segments. On the other hand, in Document B, the "16 KB unit area" of the memory mapper is a segment and the "16 KB unit or 8 KB unit area" of the megarom is a bank. In Document C, both the "16 KB unit

## MSX Memory Architecture

area" of the memory mapper and the "16 KB unit or 8 KB unit area" of the megarom are described as banks. As such, they are not uniquely named.

In order to avoid misunderstandings, the following names are used consistently in this document. When referring to other materials, it may help to avoid confusion if you consider the possibility of other names being used.

Category	Overview	Notation
Slot	Address area in 16 KB units where the specified slot appears	Page <b>N</b>
Slot	Register that specifies the basic slot # appearing in Page0-3 of the memory space visible from the Z80	Basic slot selection register
Slot	Register to specify the expansion slot # that appears in Page0-3 of the basic slot that is being eyed.	Expansion slot selection register
Memory Mapper	Memory mapper switching unit (16KB)	Segment# <b>N</b>
MegaROM	An address area in which the address of the slot in which the megarom is installed is divided into 8 KB or 16 KB units.	BANK <b>N</b>
MegaROM	A portion of the ROM that appears in the above bank	BANK# <b>N</b>

For example, if it says, "Make BANK#23 appear in BANK0," it is talking about megarom. If it says, "Make Segment#31 appear in Page1," it is talking about a memory mapper.

## 2. Slot

The Z80, the CPU of the MSX, has 64KB of memory space, but if you make the ROMs for the BIOS, internal software, etc. visible at the same time, it quickly becomes too small. Therefore, the MSX introduced the concept of slots, which allows up to 256KB of space to be handled. This "slot concept" is called the basic slot.

One basic slot can be further expanded into "four slots". These four expanded slots are called expansion slots. If all the basic slots are expanded, a total of 1MB of space can be handled. Considering the time when the first MSX was released, it was able to handle a vast amount of space, which was advanced for its time. The details are described below.

### 2.1. Basic Slot

As shown in Figure 2-1, the 64KB space is divided into 16KB areas called Pages.

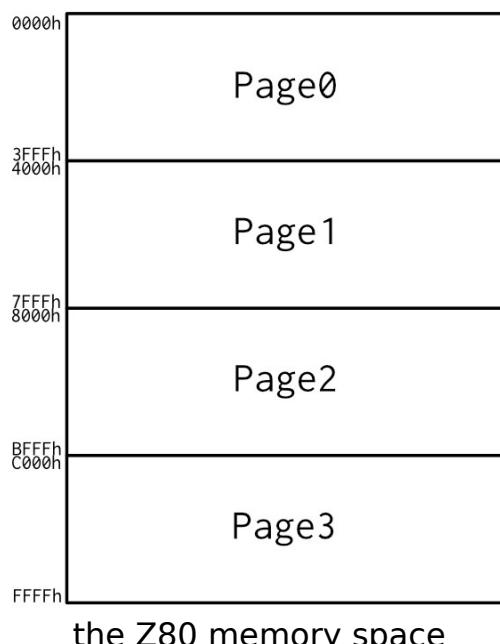


Figure 2-1: Page area classification visible in

the Z80 memory space

Four basic slots exist for each of these pages, and the basic slot selection register is used to select which basic slot should appear for each page. The image is shown in Figure 2-2.

## MSX Memory Architecture

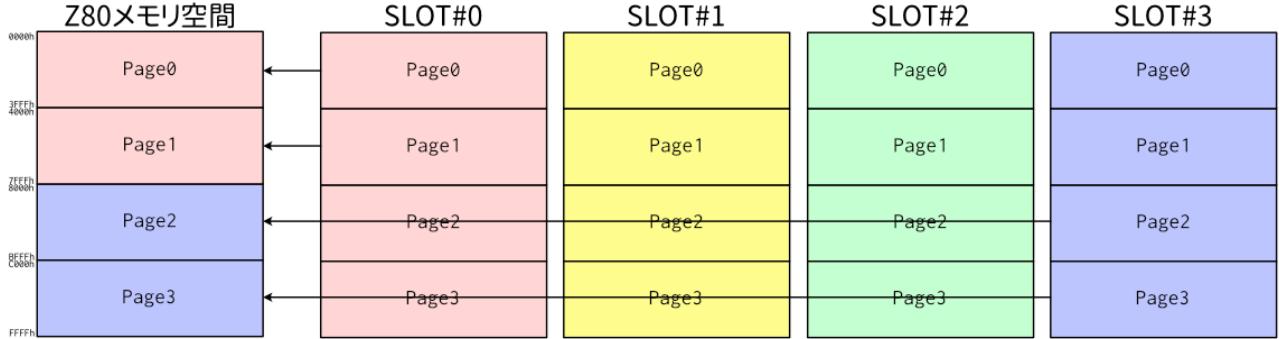


Figure 2-2: Basic slot selection

The arrow in Figure 2-2 corresponds to the basic slot selection register. The 8-bit value that can be written to I/O A8h corresponds to the slot# where bit1 and bit0 appear on Page0, the slot# where bit3 and bit2 appear on Page1, the slot# where bit5 and bit4 appear on Page2, and the slot# where bit7 and bit6 appear on Page3, as shown in Figure 2-3. As shown in Figure 2-3, the 8-bit values that can be written to I8h correspond to the slots where bit1 and bit0 appear on Page0, where bit3 and bit2 appear on Page1, where bit5 and bit4 appear on Page2, and where bit7 and bit6 appear on Page3.

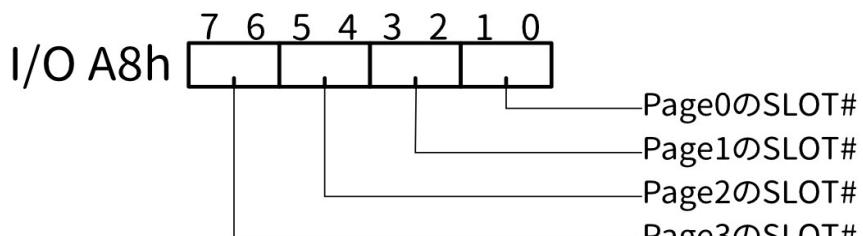
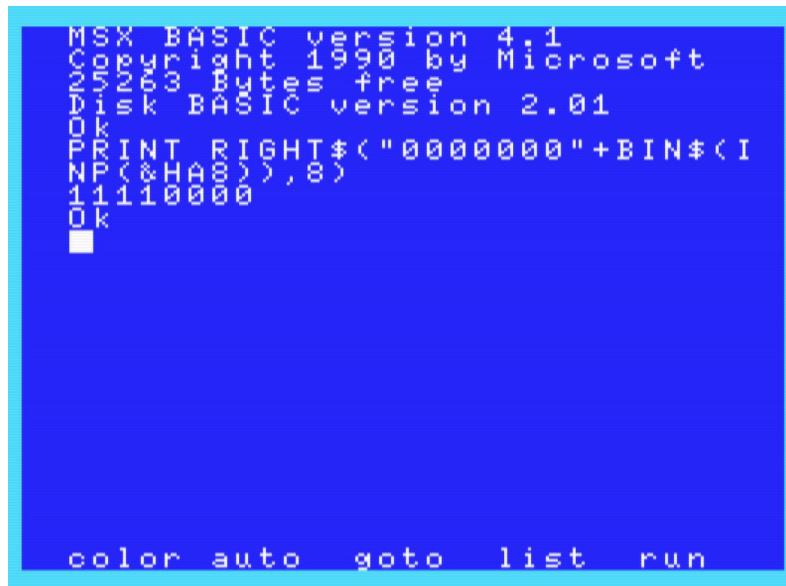


Figure 2-3:

Basic Slot Select Register (I/O A8h)

## MSX Memory Architecture

The basic slot selection register can also be read from MSX-BASIC; Figure 2-4 shows an image of the register read by the FS-A1GT.



A screenshot of an MSX-BASIC session on the FS-A1GT. The screen displays the following text:

```
MSX BASIC version 4.1
Copyright 1990 by Microsoft
25263 Bytes free
Disk BASIC version 2.01
Ok
PRINT RIGHT$("00000000"+BIN$(I
NP(&HA8)),8)
11110000
Ok
■
```

Below the main text, there is a command prompt:

```
color auto goto list run
```

Figure 2-4:  
Reading the  
Basic Slot

Selection Register from MSX-BASIC on FS-A1GT

This means that Page0 is "00" and therefore SLOT#0, Page1 is also "00" and therefore SLOT#0, Page2 is "11" and therefore SLOT#3, Page3 is also "11" and therefore SLOT#3.

In general, SLOT#1 will appear as cartridge slot 1, and SLOT#2 will appear as cartridge slot 2. In some cases, the slot # on the program does not match the cartridge slot # stamped on the main unit, or even SLOT#3 appears outside as a cartridge slot.

Although MSX-BASIC can write to A8h using the OUT instruction, switching to another slot may cause MSX-BASIC to run out of control because MSX-BASIC itself resides on MAIN-ROM appearing on Page0 and Page1, BASIC programs are stored in RAM appearing on Page2 and Page3, and MSX-BASIC and BIOS work areas are stored in RAM appearing on Page3. BASIC and BIOS work areas are stored in RAM appearing on Page3, switching to another slot may cause MSX-BASIC to run out of control.

The slot switching is also related to the expansion slot, see also 2.2. Expansion slots.

## MSX Memory Architecture

### 2.2. Expansion slots

2.1. Basic Slot: I explained that each Page selects one of the four slots to appear in the Z80 memory space. Expansion slots refer to a mechanism that further expands this one basic slot to four.

Base slots are not necessarily expanded. Cartridge slots are generally basic slots. Peripheral devices called expansion slot units can be installed in the cartridge slots to increase the number to four, but this expansion slot unit uses the expansion slot mechanism described in this section.

Slots hidden inside the main unit (SLOT#0 and SLOT#3) may be expanded. (It does not mean that something as large as an expansion slot unit is included in the main unit, and it means that the internal logic is equivalent.)

The expansion slot selection register determines which of the four expansion slots is connected to Page 0-3 of the expanded slots. It resides at FFFFh of that slot. The expansion slot selection register is mapped onto memory instead of I/O.

This image is shown in Figure 2-5.

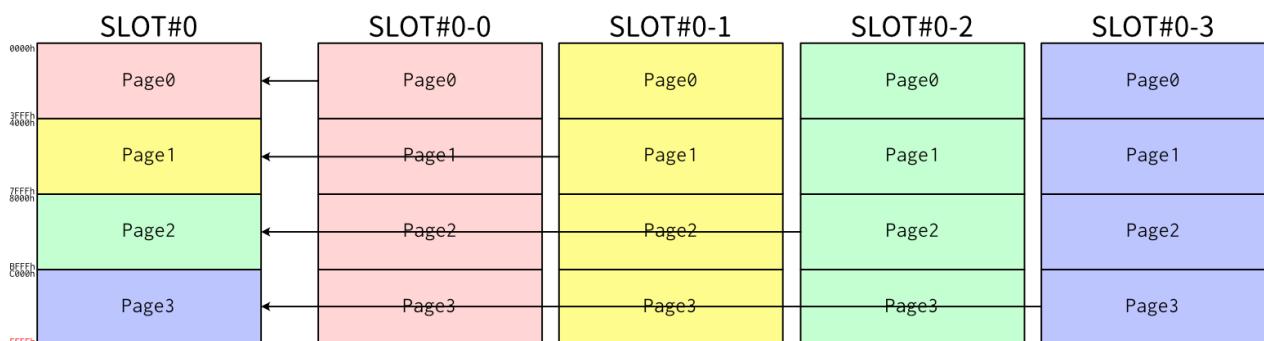


Figure 2-5: Image of expansion slot

Figure 2-5 shows an example where the basic slot SLOT#0 is expanded. SLOT#0-0, SLOT#0-1, SLOT#0-2, SLOT#0-3 exist as expansion slots of SLOT#0. The concept is almost the same as the basic slot, but the point to note is that Page3 is up to FFFEh. The expansion slot selection register is connected to FFFFh of SLOT#0, and FFFFh of expansion slots SLOT#0-X cannot be accessed.

## MSX Memory Architecture

A bitmap of the Expansion Slot Select Register FFFFh is shown in Figure 2-6.

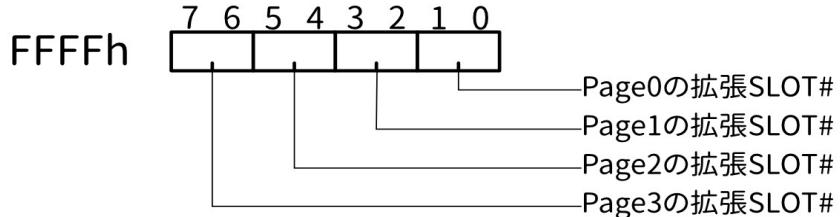


Figure 2-6:Expansion slot selection register

The value to write is similar to the basic slot selection register. However, when it is read, the value in which all bits are inverted can be read. This seems to be so for the purpose of determining whether this base slot has been expanded.

If SLOT#1 is not extended and a 64KB-RAM cartridge is installed, writing value A to FFFFh and reading it will read value A as is. On the other hand, if SLOT#1 is expanded, even if a 64KB-RAM cartridge is installed in SLOT#1-0, if the value A is written to FFFFh of SLOT#1, it will not be written to the expansion slot selection register. , and reading from FFFFh returns the inverse of the expansion slot selection register. In other words, since the value A was written, all bits of the value A are read out. The BIOS checks for the presence of an expansion slot by taking advantage of the inverted value read in the case of this expansion slot selection register.

The results of the investigation are stored in EXPTBL (FCC1h to FCC4h) shown in Figure 2-7.

## MSX Memory Architecture

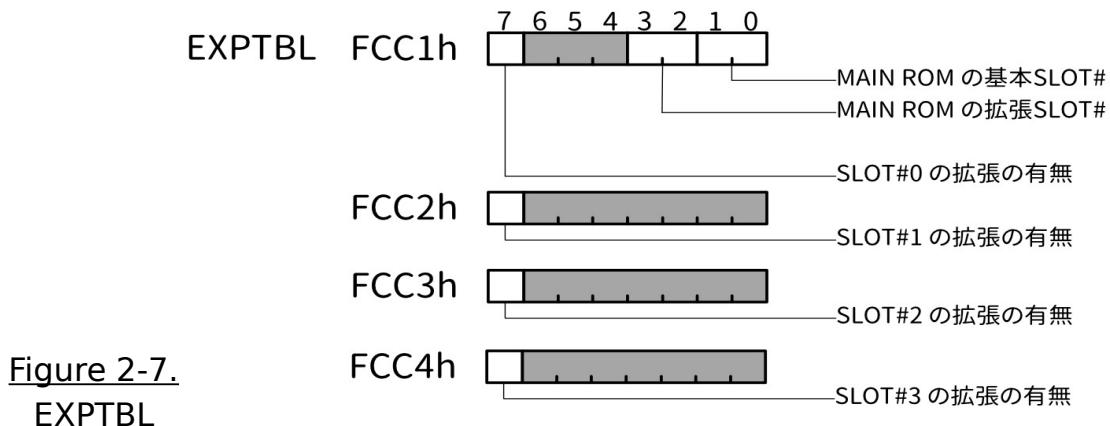


Figure 2-7.  
EXPTBL

The contents of EXPTBL are written by the BIOS according to the bit arrangement shown in Figure 2-7 immediately after starting MSX. The MAIN-ROM slot # stored in FCC1h is usually a value that indicates his SLOT#0 or SLOT#0-0, but don't make a decision, read this value and use the MAIN-ROM It is safe to use it as a slot. (For example, when using the MSX version upgrade adapter, the slot value other than SLOT#0 is stored.)

An expansion slot selection register exists for each basic slot to which an expansion slot is connected. For example, if SLOT#0 and SLOT#3 are extended, FFFFh of SLOT#0 contains the expansion slot selection register for slot 0, and FFFFh of SLOT#3 contains the expansion slot selection register for slot 3. I'm here. As some of you may have noticed by reading this, in order to switch the expansion slot, you must switch Page3 to that slot and make the expansion slot selection register appear at his FFFFh in the Z80 memory space. On the other hand, Page3 usually has stack memory and BIOS work area, so if you switch carelessly, it will run out of control. Therefore, the BIOS provides slot switching routines for easy and safe switching. The next section describes such 2.3. slot-related BIOS routines.

### 2.3. Slot-related BIOS routines

Slot-related BIOS routines are described separately here.

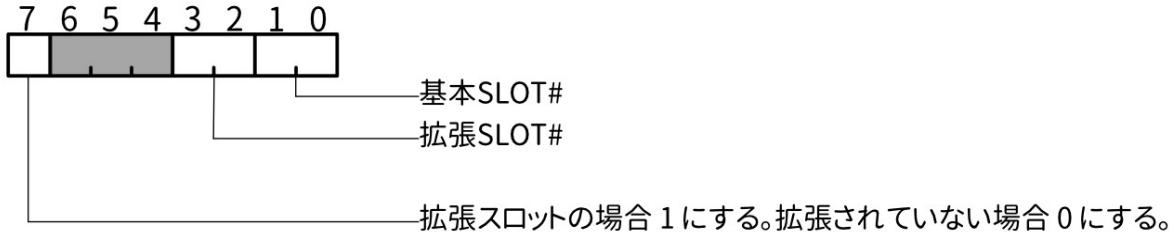
#### RDSL (000Ch)

Reads the specified address 1 byte of the specified slot.

## MSX Memory Architecture

Input:

A register: Value indicating "specified slot".



HL Register: A value that indicates a "designated address".

Output:

A Register: read value.

Detailed:

Calling this routine will destroy the AF, BC and DE registers.

The value of the A register specified before calling specifies whether or not there is an extension slot in bit7. In short, specify bit7 of the value stored in (FCC1h + target basic slot #) as it is. And set the target basic slot # to bit1, bit0. Specify the expansion slot # to bit3, bit2. SLOT#3-2 would be 10001011. 8Bh in hexadecimal notation.

BIOS switches the page corresponding to the upper 2 bits of the HL register to the slot indicated by the A register, then LD A,(HL) and restores the slot.

Use it as follows:

```
LD      HL, 0x1234
LD      A, 0x8B
CALL    0x000C
RET
```

This will read the 1-byte value at address 1234h of SLOT#3-2 and return it to the A register.

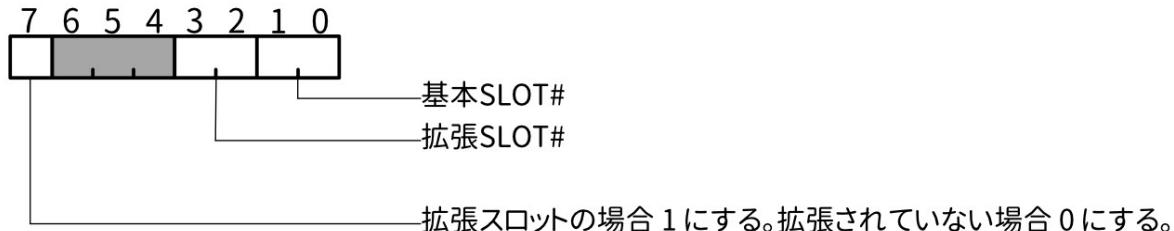
## **WRSLT (0014h)**

Writes 1 byte of the specified value to the specified address in the specified slot.

Input:

A register: Value indicating "specified slot".

## MSX Memory Architecture



HL Register: value indicating "specified address".

E Register: A value that indicates a "specified value".

Output:

none

Detailed:

Calling this routine destroys the AF, BC, and D registers.

The method of specifying the slot is the same as RDSL<sub>T</sub>, so please refer to the explanation there.

The BIOS switches the page corresponding to the upper 2 bits of the HL register to the slot indicated by the A register, then LD (HL),E and then returns the slot.

Use it as follows.

```
LD      HL, 0x1234
LD      A, 0x8B
LD      E, 0xAB
CALL    0x0014
RET
```

With this, ABh is written to 1234h address of SLOT#3-2.

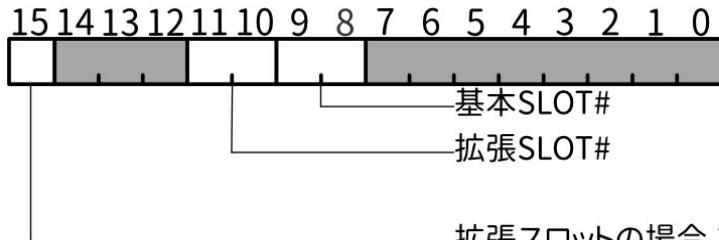
## **CALSLT (001Ch)**

Makes a subroutine call to the specified address of the specified slot (called an inter-slot call).

input:

IY register: Stores the value indicating the "specified slot" in the upper 8 bits. Lower 8 bits are ignored.

## MSX Memory Architecture



IX Register: A value that indicates the "specified address (subroutine address)".

Other registers: parameters of the subroutine to call.

output:

By subroutine you call.

detailed:

This routine calls the "specified address (subroutine address)" after switching the page corresponding to the "specified address (subroutine address)" to the "specified slot". Next, it returns the Page corresponding to the "specified address (subroutine address)" to the original slot.

The registers destroyed when calling this routine depend on the subroutine it calls.

For example, when the MSX-DOS application is started, Page0-3 are all in RAM, but you can use it when you want to call the BIOS routine of MAIN-ROM. Page0 during MSX-DOS operation is RAM, but slot-related subroutine calls exist at the same addresses as the BIOS and are available.

You can call BIOS routine CHGMOD (005Fh) to switch SCREEN mode from MSX-DOS application with code like below.

LD	A, 1	; switch to SCREEN1
of IY	LD IY, [0xFCC1 - 1]	; MAIN-ROM Slot to upper 8bit
LD	IX, 0x005F	
CALL	0x001C	

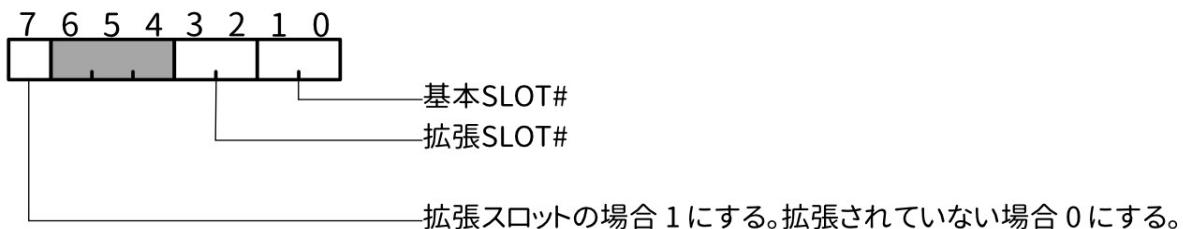
## MSX Memory Architecture

ENASLT (0024h)

Disables interrupts and switches the specified Page to the specified slot.  
Returns with interrupts disabled.

## Input:

A register : A value that indicates "specified slot".



H Register: Specify the target page with the upper 2 bits.

## Output :

none

### Detailed :

All registers are destroyed.

If you use an inter-slot call, you can use the subroutine call of another slot, but each time you switch the slot and return again, and so on. It becomes very inefficient in cases where subroutine calls, reads, and writes are frequently performed on the same slot. Therefore, you can reduce the frequency of slot switching by using ENASLT to switch slots, perform the necessary subroutine calls, reading and writing, and then return to the slot with ENASLT.

Slot switching is a process that requires a certain amount of load, so in cases where other slots are called in places that require processing speed, such as the main loop of the game, ENASLT's minimum You should consider switching limits.

ENASLT returns with interrupts disabled. This is a protective measure to avoid a runaway problem due to the interrupt handler routine changing to a different slot configuration than expected. If countermeasures are taken in the interrupt processing itself, EI can be performed immediately after returning from ENASLT.

Of course, when the program that calls ENASLT is in Page1, if you switch the Page1 slot, when you return from ENASLT, it will return to another code and run out of control, so please keep that in mind when using it.

## MSX Memory Architecture

Also, please do not switch Page 3. The body of the slot switching routine is often located on Page3's RAM, and the stack memory is also located on Page3.

By doing the following, the first 8KB of the ROM cartridge installed in SLOT#1 can be copied to Page2 from MSX-DOS.

```
LD      A, [0xFCC2]      ; Get extension of SLOT#1
AND    A, 0x80
OR     A, 0x01          ; SLOT#1 or SLOT#1-0
LD     H, 0x40          ; Page1
CALL   0x0024          ; Switch to Page1 of SLOT#1 or
SL0T#1-0
LD     DE, 0x8000
LD     HL, 0x4000
LD     BC, 0x2000
LDIR   DE, 0x8000        ; Copy to Page1 the beginning of
8192byte of 0x8000
LD     A, [0xF342]      ; Get slot with page1 of RAM
LD     H, 0x40          ; Page1
CALL   0x0024          ; switch to page1 of SLOT#1 or
SL0T#1-0
EI                           ; interrupt enable
```

\*Some devices have registers mapped onto memory addresses. In such devices, reading the register part may cause some functions to work. For example, some cartridges with an SD card slot have SPI communication functions mapped as registers in memory. Please note that SPI reception will work if read.

## **CALLF (0030h)**

Interslot call.

Input :

Store the value indicating the slot # and the address value immediately after the code calling 0030h.

The value indicating the slot # is the same as the value specified in the A register of CALSLT.

Output :

## MSX Memory Architecture

Depends on caller.

Detailed :

Destruction registers are caller dependent.

The functionality is the same as CALSLT, but IY and IX are not used to specify the slot and address. Also, 0030h can be CALL 0030h, but you can use RST 30h, which is more efficient. Specifically, call it as follows.

```
LD      A, 1          ; SCREEN1
RST    0x30
DB      0x80          ; MAIN-ROM SLOT#??
DW      0x005F        ; CHGMOD
L1:
```

The return address from the call at 0x0030 is location L1.

In the case of MSX, it is necessary to consider the possibility that the slot # will change, so in general, I think it is better to rewrite the slot # of DB 0x80 with a program that runs on RAM.

```
LD      A, [0xFCC1]    ; MAIN-ROM SLOT#
LD      [CALL_SLOT_NUM], A
LD      A, 1           ; SCREEN1
RST    0x30
CALL_SLOT_NUM:
DB      0x80
DW      0x005F        ; CHGMOD
L1:
```

## 2.4. Relationship between basic slots and expansion slots

There are four basic slots, SLOT#0, SLOT#1, SLOT#2, and SLOT#3, and expansion slots can be added independently to each of them. If an expansion slot is connected to SLOT#0, SLOT#0 increases to four: SLOT#0-0, SLOT#0-1, SLOT#0-2, SLOT#0-3. In this way, since each basic slot has an expansion slot, EXPTBL indicates whether SLOT#0 is expanded, FCC1h indicates whether SLOT#1 is expanded, FCC2h indicates whether SLOT#2 is expanded, and so on. There are four FCC3h and FCC4h that indicate whether or not there is an extension for SLOT#3. Some models have no expansion for all basic slots, and some have expansion for all built-in slots other than the cartridge slot. Of

## MSX Memory Architecture

course, there are also intermediate models. Therefore, when switching slots, it is safe to check EXPTBL and switch using the BIOS.

As an example, the slot configuration of FS-A1GT is shown in Figure 2-8.

	SLOT#0-0	SLOT#0-1	SLOT#0-2	SLOT#0-3	SLOT#1	SLOT#2	SLOT#3-0	SLOT#3-1	SLOT#3-2	SLOT#3-3
Page0 0000h 3FFFh 4000h	MAIN-ROM				CARTRIDGE SLOT #1	CARTRIDGE SLOT #2	MAPPER RAM 512KB	SUB-ROM		PANASONIC MEGA-ROM
Page1 7FFFh 8000h	MAIN-ROM		MSX-MUSIC	OPENING LOGO	CARTRIDGE SLOT #1	CARTRIDGE SLOT #2	MAPPER RAM 512KB	KANJI DRIVER	DISK-BIOS	PANASONIC MEGA-ROM
Page2 BFFFh C000h					CARTRIDGE SLOT #1	CARTRIDGE SLOT #2	MAPPER RAM 512KB	KANJI DRIVER		PANASONIC MEGA-ROM
Page3 FFFFh					CARTRIDGE SLOT #1	CARTRIDGE SLOT #2	MAPPER RAM 512KB			PANASONIC MEGA-ROM

Figure 2-8. FS-A1GT slot configuration

SLOT#0, SLOT#3 are extended. SLOT#1 and SLOT#2 are exposed as cartridge slots. Therefore, the following three slot selection registers exist in the main unit.

Basic slot selection register I/O A8h

SLOT#0 expansion slot selection register SLOT#0 - FFFFh

SLOT#3 Expansion slot selection register SLOT#3 - FFFFh

The configuration of SLOT#0 is set by the SLOT#0 expansion slot selection register to set "which SLOT#0 expansion slot appears".

The configuration of SLOT#3 sets "which SLOT#3 expansion slot appears" by the SLOT#3 expansion slot selection register,

Furthermore, what appears in the Z80 memory space is determined by the basic slot selection register.

The memory space of MSX is based on the concept of this slot, but there are cases where the memory device connected to each slot has its own expansion. Note that they work independently of slots. From the next chapter, we will pick up and explain particularly major ones among the mechanism of memory space expansion independent of slots.

As an aside, I will briefly explain the contents of each slot that appears in Figure 2-8. MAIN-ROM is the first program to be executed after the power is

## MSX Memory Architecture

turned on, and contains BIOS and MSX-BASIC. MSX-MUSIC is a ROM containing MSX-MUSIC and MSX-MIDI processing such as FM-BIOS and CALL MUSIC of MSX-BASIC. OPENING LOGO is a program that displays the MSX logo at startup. MAPPER RAM is a memory mapper compatible RAM described later. SUB-ROM is BIOS ROM added from MSX2. KANJI DRIVER is a ROM that supports CALL KANJI of MSX-BASIC. DISK-BIOS is ROM containing DISK-BASIC and MSX-DOS/DOS2. PANASONIC MEGA-ROM is equipped with MSX-JE, built-in word processing software, built-in software called A1 cockpit, MSX-View, 12-dot kanji ROM for MSX-View, dictionary SRAM for MSX-JE, RAM, MAIN-ROM, etc. Almost all the memory you have can be seen from here.

## **3. Memory mapper compatible RAM**

The RAM installed in the MSX main unit is roughly divided into two types: memory mapper compatible RAM and non-compatible RAM. Almost all MSX1 and some MSX2 internal RAMs are non-compatible RAMs, and some MSX2s and MSX2+ RAMs are memory mapper compatible RAMs.

A memory mapper is a RAM partitioned in units of 16KB, and it refers to a mechanism that allows this to appear in any area of Page 0 to 3 (memory can be mapped). RAM that does not support memory mapper is simply connected to RAM, so for example, RAM connected to Page 0 can appear in Page 0 by slot switching, but cannot appear in Pages 1 to 3.

The memory mapper compatible RAM is separated in units of 16KB and is numbered consecutively starting from 0. For example, a 64KB memory mapper compatible RAM has 4 segments, Segment#0 to 3, and a 256KB memory mapper compatible RAM has 16 segments, Segment#0 to 15. At least 64KB (4 segments) exist.

The memory mapper supported RAM has four registers: Page0 mapper segment selection register, Page1 mapper segment selection register, Page2 mapper segment selection register, and Page3 mapper segment selection register. This is a register that specifies the segment # that appears on each Page. The initial value of this register as hardware (value immediately after power-on) is undefined. If the BIOS is compatible with memory mapper compatible RAM after MSX2, it will be initialized to Page0 mapper segment selection register = 3, Page1 mapper segment selection register = 2, Page2 mapper segment selection register = 1, Page3 mapper segment selection register = 0 at startup. On the other hand, MSX1 is not

## MSX Memory Architecture

initialized. For memory mapper-supported RAMs configured with a combination of general-purpose parts, the initial value of all mapper segment selection registers is often 0. In some cases, the memory mapper compatible RAM built into the recently marketed combo cartridge has the same value as the BIOS that supports the memory mapper compatible RAM as the initial value of the hardware. However, it is dangerous to expect such initial values.

The mapper segment selection register is connected to I/O FCh, FDh, FEh, and FFh. The reason will be described later, but it is write-only and cannot be read. The Page0 mapper segment selection register is FCh, the Page3 mapper segment selection register is FFh, and so on. In other words, one mapper segment selection register is 8bit, so the maximum capacity is 256 segments from 0 to 255.  $16\text{KB} \times 256 = 4096\text{KB} = 4\text{MB}$ . An image of 128KB is shown in Figure 3-1.

## MSX Memory Architecture

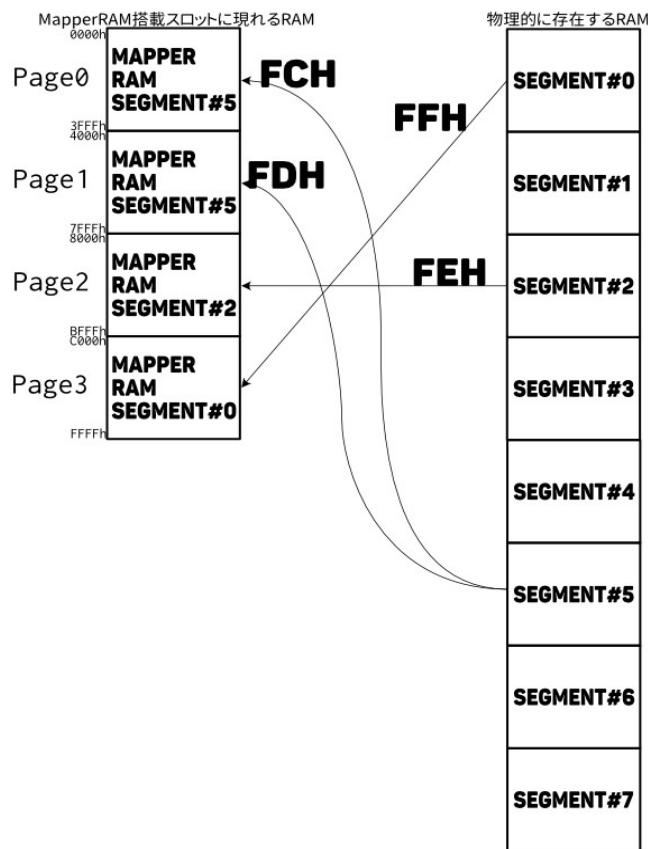


Figure 3-1. Image of slot appearance RAM of 128KB (8 segments)

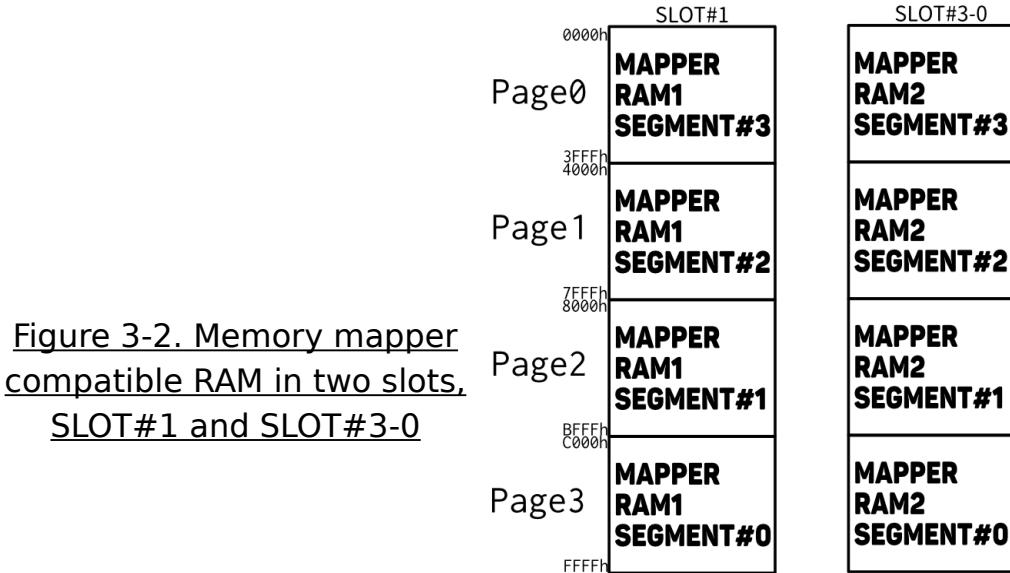
The left side in Figure 3-1 corresponds to the slot in which memory mapper compatible RAM is installed. Pages 0 to 3 are all RAM. The right side is the actual installed RAM. Since  $128\text{KB} = 16\text{KB} \times 8$ , there are 8 segments of RAM, which are numbered sequentially from Segment#0 to 7. The left side and the right side are connected by four lines, but this is an image of the mapper segment selection register. Figure 3-1 specifies I/O FCh = 5, FDh = 5, FEh = 2, FFh = 0. In this way, the same segment can appear on different Pages at the same time. Segment#6 can be changed to Page0 at one time, and Segment#6 to Page1 at the next timing. The large-capacity RAM that appears in the 64KB space realized by this is connected to a normal slot, so the memory mapper is connected to the slot that is part of the basic slot and expansion slot.

There is also a cartridge version of memory mapper compatible RAM. You can add RAM by using the cartridge version. Interestingly, the I/O address of the mapper segment selection register is the same for both the built-in memory mapper RAM and the cartridge type memory mapper RAM.

There are cartridge-type memory mapper-compatible RAMs, and there are also built-in memory mapper-compatible RAMs, but they can coexist. If multiple

## MSX Memory Architecture

memory mapper-enabled RAMs are present, writing to the memory mapper's mapper segment selection register switches segments in all connected memory mapper-enabled RAMs in the same manner. For example, if SLOT#1 is equipped with a cartridge-type 4MB RAM compatible with memory mapper, and SLOT#3-0 of the main unit is equipped with a 512KB RAM compatible with memory mapper, it will look like Figure 3-2. image.



SLOT#1 contains a 4MB Memory Mapper-enabled RAM cartridge, which we'll call MAPPER RAM1 for purposes of explanation. Similarly, SLOT#3-0 is connected to a 512KB memory mapper compatible RAM inside the main unit, which is called MAPPER RAM2. Since each is an independent device, for example, SLOT#1 segments cannot appear in SLOT#3-0. 4MB = 256 segments of RAM are available in SLOT#1 and 512KB = 32 segments of RAM are available in SLOT#3-0.

Here, if 15 is written to I/O FCh, the state will be as shown in Figure 3-3

## MSX Memory Architecture

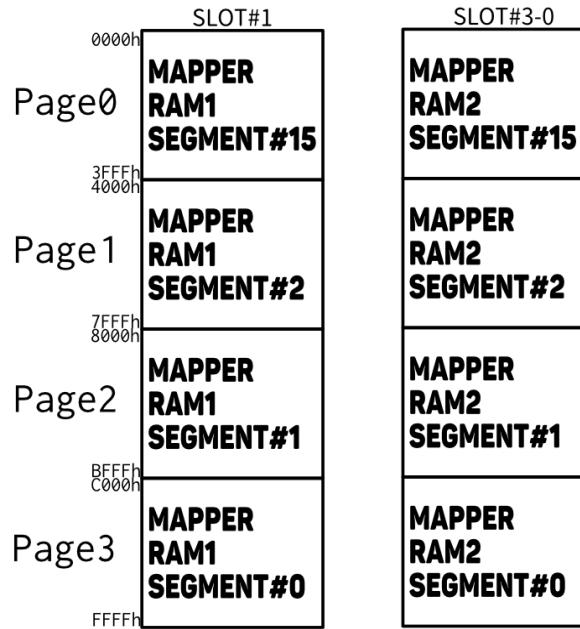


Figure 3-3. Change Segment of Page0 to 15

Regardless of whether SLOT#1 and SLOT#3-0 appear in the Z80 memory space, both Page0 segments of MAPPER RAM1 and MAPPER RAM2 are switched to Segment#15. So what happens if we write 32 to the I/O FCh? It should look like Figure 3-4.

## MSX Memory Architecture

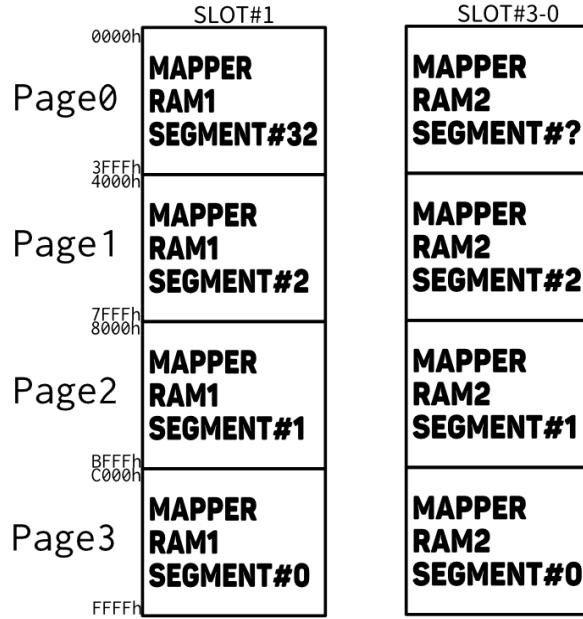


Figure 3-4. Change Segment # of Page0 to 32

MAPPER RAM1 has 256 segments (0 to 255), but MAPPER RAM2 has only 32 segments (0 to 31). I tried writing the value corresponding to Segment#32 to the mapper segment selection register 0xFC, but MAPPER RAM1 is simply switched to Segment#32. Since MAPPER RAM2 does not have Segment#32, its operation is undefined. In general, the high-order bit of segment # is ignored or ignored, so it often behaves like Segment#32 = Segment#0.

There is a 768KB memory mapper compatible RAM called MEM-768, but the capacity is incomplete.  $768\text{KB} = 16\text{KB} \times 48$ , so there are 48 segments, but if 48 is written to FCh of I/O, how many segments will appear? . Thanks to the cooperation of the person who owns it (\*1), I was able to confirm it. The mapper segment selection register is equipped with 6bits, and the upper 2bits are decoded. The MEM-768 has three 256KB RAMs, and one of these three is selected according to the decoding result of the upper 2 bits. Since there are 2 bits, there are 4 combinations, but the value corresponding to 3 seems to be unconnected. Therefore, if you specify Segment #48 to #63, it will be in a state where nothing is connected, it cannot be written, and it seems that FFh is returned when reading. However, just because MEM-768 is designed that way, if you specify a segment # that does not exist, it is safe to interpret that what will appear is undefined.

(\*1) Takashi Kobayashi ran the test program and confirmed it.

## MSX Memory Architecture

This address is basically write-only because multiple memory mapper-enabled RAMs are connected to I/O FCh, FDh, FEh, and FFh. Reading is strictly prohibited. Depending on the combination of hardware, the value of the mapper segment selection register may be returned from multiple memory mapper compatible RAMs, causing bus contention and physically destroying the MSX itself. We are unconfirmed whether there is a model that is really broken.

By using memory mapper compatible RAM, it is possible to achieve a large capacity of 4MB per slot, so there is a demand for "multiple applications to use memory separately". To do so, it is necessary to manage a mapper segment selection register that all memory mapper-enabled RAMs switch in tandem. The memory mapper support routine that is installed in the extended BIOS when MSX-DOS2 is installed is provided as a management mechanism. The next section describes how to use the memory mapper support routines.

### **3.1. Memory Mapper Support Routines**

In addition to the BIOS installed in the MAIN-ROM and SUB-ROM, MSX has a mechanism called extended BIOS. A device module that is retrofitted in a cartridge or the like exposes the BIOS routines for its control.

Memory mapper support routines are provided using this extended BIOS mechanism. However, the memory mapper compatible RAM is simply RAM, and does not have a "mechanism to set the memory mapper support routine". This is installed in MSX-DOS2 and its successor, Nextor. However, you don't need to check for the presence of MSX-DOS2 or Nextor to check for the presence of memory mapper support routines. Extended BIOS has a common procedure for various routines registered, and the memory mapper support routine follows that rule.

### **Primary mapper and secondary mapper**

Before I explain how to use it, I will explain the types of memory mappers.

If only one slot of memory mapper-enabled RAM exists, it becomes the primary mapper. In that case there is no secondary mapper.

If there are two or more memory mapper-compatible RAM slots, one will be the primary mapper and the rest will be secondary mappers.

## MSX Memory Architecture

In MSX2/2+, the RAM that supports the memory mapper with the largest capacity becomes the primary mapper. In MSXturboR, the memory mapper compatible RAM in SLOT#3-0 built into the main unit becomes the primary mapper. Up to MSX2/2+, there was only a difference in capacity between the built-in cartridge and the expansion cartridge, so it was a specification that the one with the largest capacity was selected as the primary mapper so as to minimize the trouble of switching slots. If multiple memory mapper compatible RAMs with the largest capacity are installed, the memory mapper compatible RAM installed in the slot with the smaller number becomes the primary mapper. In the case of MSXturboR, the built-in RAM can be accessed at high speed for R800, while the expansion cartridge operates at compatible speed with MSX2/2+, so the built-in RAM is overwhelmingly faster. I'm here. Because of that, I think turboR was changed to use the built-in RAM as the primary mapper.

The memory mapper support routine has a memory mapper supported RAM information table, and there is a routine that returns this table address. By referring to this table, the application can easily know the slot # and number of segments of each memory mapper compatible RAM. At the top of this information table is always the primary mapper information. Secondary mappers are lined up after the second one. The primary mapper is slightly easier to access.

Next, the 64KB RAM space exposed while MSX-DOS2/Nextor is running is called TPA, and this TPA is also allocated to the primary mapper.

Only this primary mapper can be specified as the transfer source/transfer destination buffer when performing file read/write access with the DOS functions of MSX-DOS2/Nextor. I think the big difference between primaries and secondaries is that specifying a region for the secondary mapper doesn't work. Note that if you want to read the contents of a file to a secondary mapper, you will need to load it into the primary mapper first and then block transfer it to the secondary mapper.

As you can see, there is no small difference between the primary mapper and the secondary mapper. Especially in his FS-A1ST, there is less free space because the internal RAM is always primary in turboR and the BIOS occupies some segments in R800-DRAM mode. If there is no problem with the secondary mapper, I think that the environment that can be handled will increase if you actively use the secondary mapper.

## MSX Memory Architecture

### Using mapper support routines

According to extended BIOS rules, first check if extended BIOS exists. Next, check to see if the mapper support routines exist. Then, copy the obtained jump table to somewhere on RAM, and then use the entry in the jump table by calling. The first procedure is tedious, but once you have the jump table, the rest is a simple and very fast routine. Below is a detailed explanation of how to use it.

First of all, it is necessary to confirm whether the extended BIOS mechanism itself exists. Specifically, if you read the contents of HOKVLD (FB20h) and bit0 is 0, the extended BIOS itself does not exist. If 1, extended BIOS is present.

Next, check if there is a mapper support routine in the extended BIOS. Set the A register to 0 for judgment, the D register to 4, which indicates the device number of the mapper support routine, and the E register to the function number 1, then call the extended BIOS entry EXTBIO (FFCAh). The A register remains 0 if no mapper support routine exists. If the A register has changed to something other than 0, the mapper support routines are present.

Next, specify 0 in the A register, device number 4 in the D register, function number 2 in the E register, and call EXTBIO. Then, the address of the mapper support routine jump table in the system work area is returned to the HL register. This is a 48-byte area and is a table in which 16 JP xxxx are arranged. If you copy it to a fixed address so that it is easy to use, it will be easier to call.

An example of the mapper support routine confirmation process is shown below.

```
hokvld := 0xFB20
extbio := 0xFFCA

mmap_init::
    ld      a, [hokvld]
    and     a, 1
    ret     z          ; Error if extended BIOS does not
exist (Zf=1)

        ; get MapperSupportRoutine's table
    xor     a, a
    ld      de, 0x0401      ; D=Device ID, E=01h
    call    extbio
    or      a, a
    ret     z          ; The mapper support routines
```

## MSX Memory Architecture

```
                                ; Error if not present (Zf=1)
ld      [mmap_table_ptr], hl
; get jump table
xor    a, a
ld     de, 0x0402      ; D=Device ID, E=02h
call   extbio
push   bc
ld     de, mapper_jump_table
ld     bc, 16 * 3
ldir
pop    bc
xor    a, a          ; A=0
inc    a              ; A=1, Zf=0
ret    ; Successful completion (Zf=0)
mmap_table_ptr::           dw    0      ;Address of memory mapper
information table
```

Calling mmap\_init will return A register = 0, Z flag = 1 if the mapper support routine does not exist. If the mapper support routine exists, A register = 1, Z flag = 0 will be returned.

The mapper\_jump\_table must be placed in RAM and has the following configuration.

```
mapper_jump_table::
mapper_all_seg::          ; +00h
    db    0xc9, 0xc9, 0xc9
mapper_fre_seg::          ; +03h
    db    0xc9, 0xc9, 0xc9
mapper_rd_seg::           ; +06h
    db    0xc9, 0xc9, 0xc9
mapper_wr_seg::           ; +09h
    db    0xc9, 0xc9, 0xc9
mapper_cal_seg::          ; +0Ch
    db    0xc9, 0xc9, 0xc9
mapper_calls::             ; +0Fh
    db    0xc9, 0xc9, 0xc9
mapper_put_ph::            ; +12h
    db    0xc9, 0xc9, 0xc9
mapper_get_ph::             ; +15h
    db    0xc9, 0xc9, 0xc9
mapper_put_p0::             ; +18h
    db    0xc9, 0xc9, 0xc9
mapper_get_p0::             ; +1Bh
    db    0xc9, 0xc9, 0xc9
```

## MSX Memory Architecture

```
mapper_put_p1::          ; +1Eh
    db      0xc9, 0xc9, 0xc9
mapper_get_p1::          ; +21h
    db      0xc9, 0xc9, 0xc9
mapper_put_p2::          ; +24h
    db      0xc9, 0xc9, 0xc9
mapper_get_p2::          ; +27h
    db      0xc9, 0xc9, 0xc9
mapper_put_p3::          ; +2Ah
    db      0xc9, 0xc9, 0xc9
mapper_get_p3::          ; +2Dh
    db      0xc9, 0xc9, 0xc9
```

A 48-byte area filled with C9h. C9h is the opcode for RET. If mmap\_init is called and A=1 is returned, JP xxxx will be written here. It is a call entry for each function of the mapper support routine.

Each entry in the mapper support routine is named, eg +00h would be named ALL\_SEG for him. In the above example, mapper\_all\_seg is used so that the name is less likely to overlap with others. Just add mapper\_ to the beginning and make it lower case. We have labeled 16 entries with this rule. If you want to use ALL\_SEG, you can use it by CALL mapper\_all\_seg.

Now that we are ready to call each entry, we will explain each entry in turn.

## **ALL\_SEG**

I think the name stands for allocate segment. It becomes a routine that attempts to allocate one segment.

Input :

A register

0: Secure one segment as a user segment

1: Secure one segment as a system segment

B register

0: Secure from primary mapper

0 Other than: Secure from multiple mappers (see details)

## MSX Memory Architecture

Output :

Cy flag: 0 = successful, 1 = no free segments

A register: allocated mapper memory segment #

B register: allocated mapper memory slot # (0 if called with B=0)

Detail :

The values to be set in the B register are as follows.



ENASLT Specify the reference memory mapper slot with the same bitmap as the slot # specification method used in etc. The vacant bits 6, 5 and 4 specify how to reserve for that criterion.

B レジスタの bit6,5,4 に指定する値の意味は下記の通り。

000 基準となるメモリーマッパーから確保する。

001 基準となるメモリーマッパー以外のメモリーマッパーから確保する。

010 まず基準となるメモリーマッパーから確保を試み、空きがなければ他のメモリーマッパーから確保する。

011 まず基準となるメモリーマッパー以外から確保を試み、空きがなければ基準となるメモリーマッパーから確保する。

1?? 未定義 (指定しないこと)

ユーザーセグメントとシステムセグメントの違いですが、ユーザーセグメントはプログラムの終了とともに自動的に解放されるセグメントです。通常のメモリ利用はユーザーセグメントを利用します。システムセグメントは、プログラムが終了しても解放されません。常駐プログラムなど、終了しても維持されなければ困るケースでのみシステムセグメントを使います。

また、ユーザーセグメントはセグメント#が小さい順に割り当てられます。システムセグメントはセグメント#が大きい順に割り当てられます。例えば、セグメント#4,5,6 が空いている場合に、

## MSX Memory Architecture

ユーザーセグメントを確保するとセグメント#4が確保され、システムセグメントを確保しようとするとセグメント#6が確保されます。

セカンダリメモリーマッパーのスロット#ですが、メモリーマッパー情報テーブルから知ることができます。これについては、メモリーマッパー情報テーブルの節で説明します。

## **FRE\_SEG**

指定のセグメントを解放する。

入力：

A レジスタ：解放するセグメント#

B レジスタ：解放するセグメントのメモリーマッパーのスロット#

出力：

Cy フラグ：0 = 解放に成功, 1 = 解放に失敗

詳細：

B レジスタに指定する値は、ALL\_SEG 呼び出し時に入力として与えた B レジスタの値ではありません。ALL\_SEG を呼び出した結果、得られた B レジスタの値を指定する必要がありますのでご注意下さい。

失敗するのは、誰も確保していないセグメントを指定している場合のみで、A レジスタ・B レジスタの値を誤って「無関係のプログラムが確保済みのセグメント」を指定してしまった場合はそのセグメントを解放してしまいますので要注意です。例えば、MSXturboR では、BIOS 等の一部の ROM 領域を内蔵 RAM にコピーしていますが、プライマリーメモリーマッパー上で、その領域はシステムセグメント扱いで確保されている状態になっています。そうすることで、他のアプリケーションから BIOS コピーを保護しているわけですが、これも解放できてしまいます。

## **RD\_SEG**

インターフェースセグメントリードです。A:HL で示されるアドレスの値を読み出して返します。

入力：

A レジスタ：読み出し対象となるセグメント#

HL レジスタ：セグメント内アドレス（上位 2bit は無効）

## MSX Memory Architecture

出力：

A レジスタ：読み出したデータ

その他のレジスタは保存されます。

詳細：

割り込み禁止で戻ります。

内部で Page2 のセグメントを切り替えてしまうので、スタックメモリは Page2 以外へ置く必要があります。

事前に、Page2 を対象とするメモリーマッパーのスロットに切り替えておく必要があります。このルーチンは、Page2 に現れているメモリーマッパーのセグメント A にあるアドレス HL の値を読みだして返します。そのため、Page2 を読み出したいメモリーマッパーのスロットへ切り替えておく必要があります。

セグメントも含むランダムアドレスから読み出したい場合に有効な機能ですが、同一セグメントから連続で読み出したい場合は、後述のダイレクトページング(PUT\_P2)を使った方が高速です。

処理速度重視のため、指定のセグメントが確保済みであるかなどのエラーチェックは行われません。そのため、エラーを示す返値がありません。A レジスタで指定するセグメントは、自身の管理下にある前提です。

## **WR\_SEG**

インターフェースアダプタです。A:HL で示されるアドレスへ E を書き込みます。

入力：

A レジスタ：読み出し対象となるセグメント#

HL レジスタ：セグメント内アドレス (上位 2bit は無効)

E レジスタ：書き込む値

出力：

A レジスタ：破壊されます。

その他のレジスタは保存されます。

## MSX Memory Architecture

詳細：

割り込み禁止で戻ります。

内部で Page2 のセグメントを切り替えてしまうので、スタックメモリは Page2 以外へ置く必要があります。

事前に、Page2 を対象とするメモリーマッパーのスロットに切り替えておく必要があります。このルーチンは、Page2 に現れているメモリーマッパーのセグメント A にあるアドレス HL へ E レジスタの値を書き込みます。そのため、Page2 を書き込みたいメモリーマッパーのスロットへ切り替えておく必要があります。

セグメントも含むランダムアドレスから書き込みたい場合に有効な機能ですが、同一セグメントから連続で書き込みたい場合は、後述のダイレクトページング(PUT\_P2)を使った方が高速です。

処理速度重視のため、指定のセグメントが確保済みであるかなどのエラーチェックは行われません。そのため、エラーを示す返値がありません。A レジスタで指定するセグメントは、自身の管理下にある前提です。

## **CAL\_SEG**

インターフェースコールです。BIOS にあるインターフェースコール CALSLT と似たインターフェースで、指定のセグメントにあるサブルーチンを呼び出すルーチンです。

入力：

IY レジスタ：上位 8bit に指定のセグメントの番号 (下位 8bit は無視される)

IX レジスタ：呼び出すサブルーチンのアドレス

AF, BC, DE, HL レジスタ：呼び出すサブルーチンへ渡すパラメータ

出力：

AF, BC, DE, HL, IY レジスタ：呼び出したサブルーチンからの返値

他のレジスタ：破壊されます。

詳細：

## MSX Memory Architecture

このルーチンは、呼び出すサブルーチンのアドレスに対応する Page を、指定のセグメントに切り替えて、呼び出すサブルーチンのアドレスを CALL します。その後、呼び出すサブルーチンのアドレスに対応する Page を、もとのセグメントに戻します。

このルーチンはスロットの切替は行いません。セグメントの切替のみです。IX レジスタの上位 2bit に対応する Page が、目的のサブルーチンを有するメモリーマッパーのスロットに切り替え済みである前提です。

スロットの切替処理は、比較的負荷の掛かる処理であるため、呼び出し側で必要最小限にすることを想定して、このルーチン内ではスロット切替を内包していないようです。メモリーマッパーのセグメント切替は、スロット切替と比べれば格段に高速であるため、その処理を内包しています。

当然ながら、Page0 には割り込みルーチンのエントリがあり、Page3 には BIOS ワークなどがあります。どこかに STACK メモリもあります。これらに十分注意してスロット切り替えを事前実施してから利用する形になります。

インターフェースコール自体が Page3 に存在するため、Page3 のインターフェースコールは利用できません。Page3 を指定 (IX の上位 2bit が 11) すると、セグメント切替は行われず、単純に CALL されるだけになります。

インターフェースコールルーチンの中で裏レジスタ (AF', BC', DE', HL') を使っています。これらを入力として利用するようなサブルーチンコールには利用できませんのでご注意下さい。

## **CALLS**

インターフェースコールです。BIOS にあるインターフェースコール CALLF と似たインターフェースで、指定のセグメントにあるサブルーチンを呼び出すルーチンです。

入力：

レジスタを使わず、下記のようにコード上にパラメータを埋め込む仕組みです。

CALL CALLS

DB セグメント#

DW サブルーチンアドレス

AF, BC, DE, HL には呼び出すルーチンへの入力パラメータを指定する。

## MSX Memory Architecture

出力：

AF, BC, DE, HL, IX, IY レジスタ：呼び出したサブルーチンからの返値

その他のレジスタ：破壊されます。

詳細：

このルーチンはスロットの切替は行いません。セグメントの切替のみです。サブルーチンアドレスに対応する Page が、目的のサブルーチンを有するメモリーマッパーのスロットに切り替え済みである前提です。

スロットの切替処理は、比較的負荷の掛かる処理であるため、呼び出し側で必要最小限にすることを想定して、このルーチン内ではスロット切替を内包していないようです。メモリーマッパーのセグメント切替は、スロット切替と比べれば格段に高速であるため、その処理を内包しています。

当然ながら、Page0 には割り込みルーチンのエントリがあり、Page3 には BIOS ワークなどがあります。どこかに STACK メモリもあります。これらに十分注意してスロット切り替えを事前実施してから利用する形になります。

インターフェースコール自体が Page3 に存在するため、Page3 のインターフェースコールは利用できません。サブルーチンアドレスに Page3 のアドレスを指定しても、セグメント切替は行われず、単純に CALL されるだけになります。

インターフェースコールルーチンの中で裏レジスタ (AF', BC', DE', HL') を使っています。これらを入力として利用するようなサブルーチンコールには利用できませんのでご注意下さい。

## **PUT\_PH**

ダイレクトページング。指定した Page のマッパーセグメントを、指定のセグメントに変更する。

入力：

H レジスタ：上位 2bit で対象の Page 番号を指定。

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

## MSX Memory Architecture

レジスタはすべて保存されます。

詳細：

指定のページに対応するマッパーセグメント選択レジスタへの書き込みを行います。すべてのメモリーマッパーの指定のページは指定のセグメントに切り替わります。

このルーチンは、マッパーセグメント選択レジスタへ書き込む処理と、その書き込んだ値をワークエリアに記憶する処理のみを実施して、即座に戻ってきます。スロット切り替えルーチンなどと比べれば、かなり高速なので、頻繁な切替にも利用できます。

確保したセグメントの番号と、それに対応するメモリーマッパーのスロットの番号の対応関係は、このルーチンを呼び出す側が矛盾無く設定することを前提としています。速度優先のため、そこに矛盾があった場合にチェックする機構は無いのでご注意下さい。

指定するセグメント#は ALL\_SEG で確保して得られたセグメント#になります。ALL\_SEG は、対応するスロット#も返しますが、そのスロット切替は、このルーチンを呼び出す側の責務になるのでご注意下さい。入力に、複数搭載されているかもしれないメモリーマッパーのうち、どのスロットのものなのか選択する値がないのは、無差別にすべてのメモリーマッパーに対して切替を行うためです。このあたりが、かなり独特なのですが、先だって説明したように I/O アドレスが同じなので複数あるメモリーマッパー個別にセグメント切替が出来ないことは、ご理解頂けると思います。

注意ですが、Page3 のセグメントは切り替えられません。H レジスタに 0b11xxxxxx (x は任意)を指定した場合、何もせずに戻ります。

## **GET\_PH**

ダイレクトページング。指定した Page のマッパーセグメント#を取得する。

入力：

H レジスタ：上位 2bit で対象の Page 番号を指定。

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

## MSX Memory Architecture

詳細：

指定のページは、所望のメモリーマッパーのスロットに切り替え済みである前提となります。

このルーチンは、ワークエリアに記憶してある「指定のページに対して最後に指定したセグメント#」を読み出して返します。即座に戻ってきます。スロット切り替えルーチンなどと比べれば、かなり高速なので、頻繁な切替にも利用できます。

確保したセグメントの番号と、それに対応するメモリーマッパーのスロットの番号の対応関係は、このルーチンを呼び出す側が矛盾無く設定することを前提としています。速度優先のため、そこに矛盾があった場合にチェックする機構は無いのでご注意下さい。

## **PUT\_P0**

ダイレクトページング。Page0 のマッパーセグメントを、指定のセグメントに変更する。

入力：

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

PUT\_PH の対象となる Page が、Page0 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page0 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

## **GET\_P0**

ダイレクトページング。Page0 のマッパーセグメント#を取得する。

## MSX Memory Architecture

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

GET\_PH の対象となる Page が、Page0 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page0 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

## **PUT\_P1**

ダイレクトページング。Page1 のマッパーセグメントを、指定のセグメントに変更する。

入力：

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

PUT\_PH の対象となる Page が、Page1 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page1 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

## MSX Memory Architecture

### **GET\_P1**

ダイレクトページング。Page1 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

GET\_PH の対象となる Page が、Page1 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page1 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

### **PUT\_P2**

ダイレクトページング。Page2 のマッパーセグメントを、指定のセグメントに変更する。

入力：

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

## MSX Memory Architecture

詳細：

PUT\_PH の対象となる Page が、Page2 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page2 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

## **GET\_P2**

ダイレクトページング。Page2 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

GET\_PH の対象となる Page が、Page2 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page2 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

## **PUT\_P3**

なにもしません。

## MSX Memory Architecture

入力：

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

PUT\_PH の対象となる Page が、Page3 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page3 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

実際のところ、Page3 はマッパーサポートルーチンが存在する場所なので、切り替えることが出来ません。そのため、本ルーチンは何もせずに戻るので要注意です。

## **GET\_P3**

ダイレクトページング。Page3 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

## MSX Memory Architecture

GET\_PH の対象となる Page が、Page3 固定になっているものと思って下さい。普通の使い方をしていれば、目的となる Page は、コーディングの最中に静的に決まることが多いので、少しでも高速化するためにどのページか判断する処理を省略できる「Page3 専用版」が用意されているとお考え下さい。Page 指定のためだけに H レジスタを破壊しなくて良いというメリットもあります。

Page3 は、BIOS ワークもある関係で PUT\_P3 や PUT\_PH で Segment# を変更できません。そのため、GET\_P3 では、常に Segment#0 を示す 0 が A レジスタに返ってきます。

## メモリーマッパー情報テーブル

上記の初期化のサンプルプログラム (Using mapper support routines) を実行すると、初期化成功時には mmap\_table\_ptr にメモリーマッパー情報テーブルのアドレスが格納されます。

ここには、MSX システムに現在搭載されているメモリーマッパー対応 RAM の情報が格納されており、1 つのメモリーマッパー対応 RAM につき 1 エントリある配列構造になっています。1 つのエントリは下記の構造になっています。

## MSX Memory Architecture

オフセットアド 意味	
レス	
+0	メモリーマッパー対応 RAM のスロット#
+1	セグメントの総数(4~255)。 (プライマリマッパーの場合は、8~255)
+2	未使用のセグメントの総数
+3	システムセグメントとして割り当て済みのセグメントの総数 (プライマリマッパーの場合は、最低でも 6)
+4	ユーザーセグメントとして割り当て済みのセグメントの総数
+5~+7	予約領域

メモリーマッパー対応 RAM のスロット#には、ENASLT 等の BIOS にあるスロット制御ルーチンに使えるスロット#が入っています。メモリーマッパーサポートルーチン ALL\_SEG では、プライマリマッパーはスロット#を指定せずに B=0 で利用できますが、実際に確保したセグメントを利用するためには、ENASLT 等でそのスロットに切り替える必要があります。その際にスロット#が必要になるスロット#は、この "メモリーマッパー対応 RAM のスロット#" の値を使って下さい。

セグメントの総数の値は、プライマリマッパーの場合は最低でも 8 になります。8 セグメント ×16KB=128KB で、MSX-DOS2 の「最低 128KB 必要」という要求スペックと一致しています。

システムセグメントとして割り当て済みのセグメントの総数が、プライマリマッパーの場合は最低でも 6 になります。これは、MSX-DOS2 自身が TPA 用に 4 セグメントと DOS2 ワークエリア用に 2 セグメント確保するためです。MSXturboR の場合、DRAM モード用の BIOS コピー置き場もこのシステムセグメントになるため、システムセグメントとして割り当て済みのセグメントの総数は 10 になります。

この 8 バイトの構造が、メモリーマッパー対応 RAM の数だけ並んでおり、その次はターミネータとして 0 が格納されています。MSX は、SLOT#0 が拡張されていない場合、MAIN-ROM のスロットになります。従って、SLOT#0 がメモリーマッパー対応 RAM の機種は存在しません。このことを利用して、メモリーマッパーサポートルーチンでは、スロット#を指定する場所で、「0」という値をしばしば特別な扱いで使っています。メモリーマッパー情報テーブルの先頭 1byte は、メモリーマッ

## MSX Memory Architecture

パー対応 RAM のスロット#を示していますが、ここに 0 が入っている場合、複数個連続配置される可能性のあるメモリーマッパー情報テーブルの終わりを示す「ターミネータ」の扱いになっています。

マッパーセグメント選択レジスタは 0~255 の 256 通りで 16KB × 256 セグメント = 4096KB となります。セグメントの総数の最大値は 255 のため、4096KB のメモリーマッパー対応 RAM を装着しても、マッパーサポートルーチンは 255 セグメントまでしか対応しませんので、4080KB までしか使えません。

例として、FS-A1GT に 4MB のメモリーマッパー対応 RAM カートリッジを SLOT#2 に装着した場合の、メモリーマッパー情報テーブルの値を下記に示します。

メモリーマッパー	オフセットアドレス	意味	値
プライマリ	+0	メモリーマッパー対応 RAM のスロット#	83h (SLOT#3-0)
	+1	セグメントの総数	32 segment
	+2	未使用のセグメントの総数	22 segment
	+3	システムセグメントとして割り当てるセグメントの総数	10 segment
	+4	ユーザーセグメントとして割り当てるセグメントの総数	0 segment
	+5~+7	予約領域	all 0
セカンダリ 1	+8	メモリーマッパー対応 RAM のスロット#	02h (SLOT#2)
	+9	セグメントの総数	255 segment
	+10	未使用のセグメントの総数	255 segment
	+11	システムセグメントとして割り当てるセグメントの総数	0 segment
	+12	ユーザーセグメントとして割り当てるセグメントの総数	0 segment
	+13~+15	予約領域	all 0
ターミネータ	+16	メモリーマッパー対応 RAM のスロット#	0

## MSX Memory Architecture

図 3-5.に、実際にツールを使って確認した画面を示します。

```
A>mmap
MemoryMapperInformation
=====
Programmed by HRA!
=====
Slot 0x83 (Primary)
Total Seg. 0x20
Free Seg. 0x16
System Seg. 0x00
User Seg. 0x00
=====
Slot 0x02
Total Seg. 0x0F
Free Seg. 0x0F
System Seg. 0x00
User Seg. 0x00
A>■
```

図 3-5. メモリーマッパー情報

テーブルダンプツールの表示

以下に、図 3-5.で用いたメモリーマッパー情報テーブルダンプツールのソースコードを掲載します。

include している別のソースは、この節で説明する本質的な部分では無いので、付録 1. サンプルプログラムから利用している各種ソースに掲載しておきます。

### mmap.asm

```
; =====
; Memory Mapper の情報をダンプするプログラム
; =====

        include      "msxbios.asm"
        include      "msxdos1.asm"
        include      "msxdos2.asm"

        org         0x100

entry:::
        ; 初期化
        ld          sp, [TPA_BOTTOM]
        call        mmap_init

        ; 起動メッセージ
        ld          de, msg_entry
        call        puts

        ; マッパーチェック
        ld          ix, [mmap_table_ptr]
        xor        a, a
mapper_check_loop:::
        call        dump_one
        jr          c, exit_loop
        ld          de, 8
        add        ix, de
        jr          mapper_check_loop

exit_loop:::
        ld          b, 0
        ld          c, D2F_TERM
        bdos

; =====
; Dump a mapper information
```

## MSX Memory Architecture

```
;      input)          ix .... target mapper table
;           a ..... 0: primary mapper, others: not primary mapper
;      break)         a ..... a + 1
; =====
;           scope      dump_one
dump_one:::
; end check
    ld          b, [ix + 0]
    inc         b
    dec         b
    scf
    ret         z

    push        af
    ld          de, msg_separator
    call        puts

    ld          a, [ix + 0]           ; slot number
    call        dec2hex

    pop         af
    push        af
    ld          de, msg_primary_mapper_mark
    or          a, a
    call        z, puts
    ld          de, msg_crlf
    call        puts

    ld          de, msg_total
    call        puts
    ld          a, [ix + 1]           ; total segments
    call        dec2hex
    ld          de, msg_crlf
    call        puts

    ld          de, msg_free
    call        puts
    ld          a, [ix + 2]           ; free segments
    call        dec2hex
    ld          de, msg_crlf
    call        puts

    ld          de, msg_system
    call        puts
    ld          a, [ix + 3]           ; system segments
    call        dec2hex
    ld          de, msg_crlf
    call        puts

    ld          de, msg_user
    call        puts
    ld          a, [ix + 4]           ; user segments
    call        dec2hex
    ld          de, msg_crlf
    call        puts
    pop         af
    or          a, a               ; Cy = 0
    inc         a
    ret
endscope

; =====
; Dump A register value by hex
;      input)          a ..... target number
; =====
;           scope      dec2hex
dec2hex:::
    ld          b, a
```

## MSX Memory Architecture

```
rrca
rrca
rrca
rrca
and      a, 0x0F
add      a, '0'
cp       a, '9' + 1
jr       c, skip1
add      a, 'A' - '0' - 10
skip1:
ld       [hex2byte], a
ld       a, b
and      a, 0x0F
add      a, '0'
cp       a, '9' + 1
jr       c, skip2
add      a, 'A' - '0' - 10
skip2:
ld       [hex2byte + 1], a
ld       de, hex2byte
call    puts
ret
hex2byte:::
ds      "00"
db      0
endscope

; =====
;     Data area
; =====
msg_total:::
ds      "Total Seg. 0x"
db      0
msg_free:::
ds      "Free Seg. 0x"
db      0
msg_system:::
ds      "System Seg.0x"
db      0
msg_user:::
ds      "User Seg. 0x"
db      0
msg_not_enough_memory:::
ds      "Not enough memory!!"
msg_crlf:::
db      0x0D, 0x0A, 0
msg_separator:::
ds      "====="
db      0x0D, 0x0A
msg_slot:::
ds      "SLOT      0x"
db      0
msg_primary_mapper_mark:::
ds      " (Primary)"
db      0
msg_entry:::
ds      "MemoryMapperInformation"
db      0x0D, 0x0A
ds      "====="
db      0x0D, 0x0A
ds      "Programmed by HRA!"
db      0x0D, 0x0A
db      0x0D, 0x0A, 0

include  "stdio.asm"
include  "memmapper.asm"
```

## 4. メガ ROM

メガ ROM とは、1Mbit を越える ROM のことです。Z80 のメモリ空間は  $64KB = 0.5Mbit$  しかありませんので、1Mbit 以上を扱えるようにするためにバンク切り替えの仕組みを備えています。

バンク切り替えの仕組みを実現する部品または部品群のことをメガ ROM コントローラー、略してメガコンと呼びます。MSX 規格ではこの内容までは規定されておらず、様々な種類が存在しています。

ASCII が MSX で利用可能なメガコンをいくつかリリースしており、これを採用しているソフトが多いです。ASCII のメガコンには、ソフトウェアから見て大雑把に分類すると 2 種類存在しています。ASCII-8K タイプと ASCII-16K タイプの 2 種類になります。メガコンの IC の種類は LZ93A13 (32pin), M60002-0125SP (42pin), BS6101 (42pin), NEOS MR6401 (28pin), BS6202 (42pin), IREM TAM-S1 (28pin) 等[\*]、多数あるようですが、本書ではソフトウェア制御を焦点としていますので、ASCII-8K タイプ・ASCII-16K タイプの 2 種類について説明します。IC の違いについては言及しません。

他には、Konami が独自のメガコンを搭載しています。本書では SCC 無し版・SCC・SCC-I の 3 種類について説明します。

すでに MSX のゲームカートリッジも貴重な存在になっていますので、メガ ROM カートリッジを自作するとすれば、CPLD や FPGA 等のプログラマブルロジックデバイスを使ってメガコンを新たに作ってしまう方が現実的です。独自仕様のメガコンを作ることも可能ですし、コピーガードのためにそういうメガコンを設計するのも良いかと思います。しかし、そのカートリッジに書き込むゲームなどのソフトウェア開発の段階では、ASCII か Konami の、従来市販品で使われていたメガコンと互換で開発することにより、MSX のエミュレーター上で気楽に動作確認できるメリットがあります。ソフトウェアが組み上がってから、必要に応じて独自仕様メガコンに合わせた修正を行うというのも開発効率面で有利かと思います。もちろん、従来メガコンと互換のメガコンのままカートリッジにしても良いかと思います。

そういう事情から、メガコンの制御の仕方について知っておくことは有意義だと思いますので、本章ではメガコンの種類毎にそれぞれ説明したいと思います。

Panasonic の MSX2+ 以降の本体には、独自のメガコンが入っています。あえてこの互換にする人もいないとは思いますが、FS-A1GT の挙動を調べたりする際の参考になるかもしれないこのメガコンも説明したいと思います。FS-A1FX/WX/WSX/ST/GT とありますが、このメガコンにもリビジョンがあって微妙な差異があるようです。本書では主に FS-A1GT のメガコンについて説明します。

[\*] 情報ソース: <https://gigamix.hatenablog.com/entry/rom/>

## 4.1. ASCII-8K タイプ

ASCII-8K タイプは、ROM を 8KB 単位の領域に区切り、連番を振っています。これを BANK0 (4000h-5FFFh), BANK1 (6000h-7FFFh), BANK2 (8000h-9FFFh), BANK3 (A000h-BFFFh) に出現させてアクセスする方式です。振った連番の N 番目を、ここでは BANK#N と表記します。

スロットのところで説明した Page の中に 2 つの BANK が存在している形で、Page1 と Page2 で 4 つの BANK を構成しています。この 4 つを小さい値のアドレスから順に BANK0～BANK3 と呼びます。メガ ROM 内にいくつか搭載している BANK (これは ROM の実体)を、この BANK0～BANK3 (これはアドレス範囲)に、切り替えながら出現させることで、ROM の全域にアクセス出来るようになっています。

例えば、1Mbit の ASCII-8K タイプ メガ ROM は、 $1[\text{Mbit}] = 128[\text{KB}] = 8[\text{KB}] \times 16[\text{BANK}]$  なので、16 個の BANK がありますので、図 4.1-1.のような構成になります。

## MSX Memory Architecture

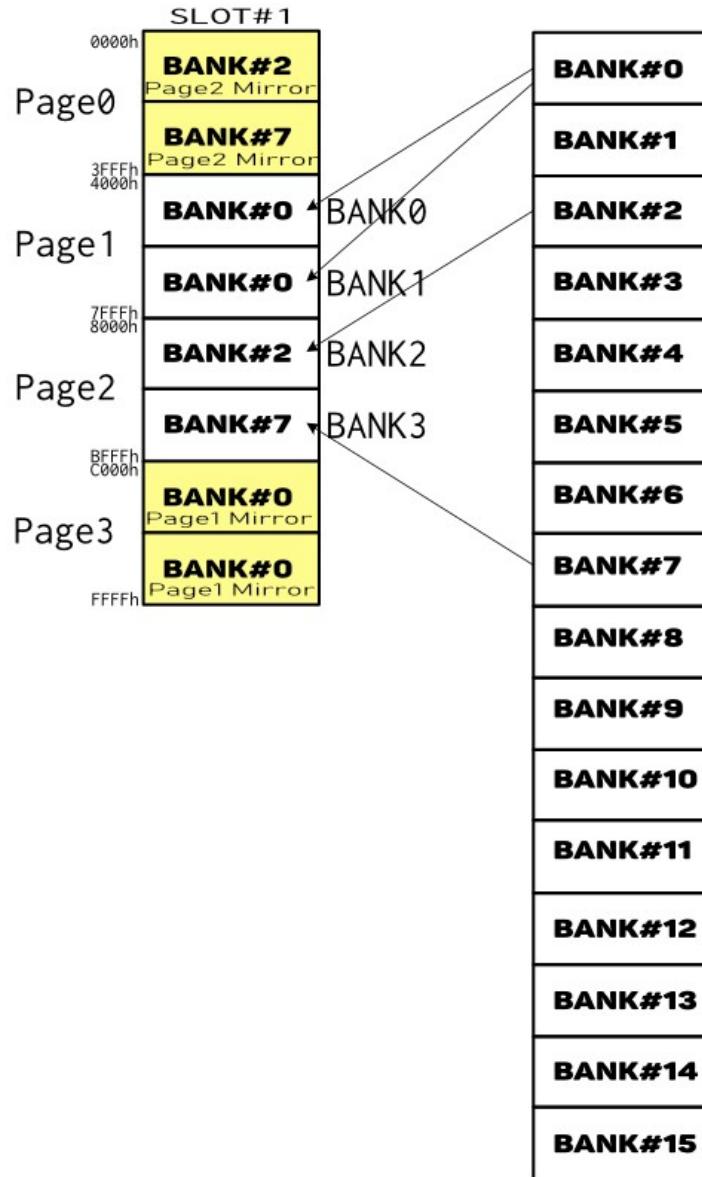


図 4.1-1. ASCII-8K タイプ 1Mbit メガ ROM のイメージ

図 4.1-1. は、SLOT#1 に装着した 1Mbit メガ ROM のイメージです。SLOT#1 の Page0 には Page2 のミラー、Page3 には Page1 のミラーが出現しています。

Page1 の中に BANK0 と BANK1 が、Page2 の中に BANK2, BANK3 が存在しています。

図右側に 16 個並んでいる BANK#0～#15 が ROM になります。1 つ 8KB で 16 個あるので 8KB × 16 個 = 128KB です。

図左側と図右側を結ぶ 4 本の矢印がバンク選択レジスタとなります。BANK0～BANK3 のそれぞれにバンク選択レジスタが存在し、BANK#0～#15 の任意の BANK# を指定できます。そのため、同じ BANK# を同時に複数の BANK に出現させることも出来ます。図では BANK0 と BANK1 に BANK#0 を出現させていますね。

## MSX Memory Architecture

バンク選択レジスタは、Memory mapped I/O となっています。バンク選択レジスタのアドレスを表 4.1-1.にまとめます。

表 4.1-1.ASCII-8K バンク選択レジスタのアドレス

BANK 番号	対応するバンク選択レジスタのアドレス
BANK0 (4000h-5FFFh)	6000h-67FFh (初期値 BANK#0)
BANK1 (6000h-7FFFh)	6800h-6FFFh (初期値 BANK#0)
BANK2 (8000h-9FFFh)	7000h-77FFh (初期値 BANK#0)
BANK3 (A000h-BFFFh)	7800h-7FFFh (初期値 BANK#0)

バンク選択レジスタのアドレスが 2048byte もの範囲を示していますが、実体は 1byte しかありません。そのため、 $8\text{KB} \times 256 = 2048\text{KB} = 2\text{MB}$  が最大容量となります。範囲を持っている理由は、メガコンの回路をシンプルにするために、アドレス信号を一部のみ利用して判定しているためです。なぜシンプルになるのかについては、ハードウェアの章で説明します。

ご覧のように、バンク選択レジスタはすべて Page1 に存在しています。Page2 にある BANK2 及び BANK3 のバンク選択レジスタも Page1 に存在するため、例えば Page1 に本体の RAM のスロットを出現させ、Page2 にこのメガ ROM のスロットを出現させてるときに、BANK2 や BANK3 を切り替える場合は、一時的に Page1 をメガ ROM のスロットに切り替えて BANK2 や BANK3 のバンク選択レジスタを書き替えてから、Page1 を本体の RAM のスロットへ戻すといった処理が必要になりますのでご注意下さい。

バンク選択レジスタは、書き込み専用です。読み出した場合はそのアドレスに対応した ROM の内容が読み出されます。

バンク選択レジスタが Page1 に集まっている理由ですが、バックアップ用 SRAM への配慮かもしれません。バックアップ用 SRAM に対応しているメガコンもあります。一部のバンクにアクセスした場合にのみメガコンを経由した /WE が L になる構造です。SRAM にアクセスする場合にバンク選択レジスタがあると邪魔になるので、SRAM は Page2 に出現させてアクセスすることを想定しているのかもしれません。

## 4.2. ASCII-16K タイプ

ASCII-16K タイプは、ROM を 16KB 単位の領域に区切り、連番を振っています。これを BANK0 (4000h-7FFFh), BANK1 (8000h-BFFFh) に出現させてアクセスする方式です。振った連番の N 番目を、ここでは BANK#N と表記します。

## MSX Memory Architecture

スロットのところで説明した Page と、メガ ROM の BANK のサイズが同じ 16KB です。Page1 と Page2 で 2 つの BANK を構成しています。この 2 つを小さい値のアドレスから順に BANK0～BANK1 と呼びます。つまり、BANK0=Page1、BANK1=Page2 という対応関係になります。メガ ROM 内にいくつか搭載している BANK (これは ROM の実体)を、この BANK0～BANK1 (これはアドレス範囲)に、切り替えながら出現させることで、ROM の全域にアクセス出来るようになっています。

例えば、1Mbit の ASCII-16K タイプ メガ ROM は、 $1[\text{Mbit}] = 128[\text{KB}] = 16[\text{KB}] \times 8[\text{BANK}]$  なので、8 個の BANK がありますので、図 4.2-1. のような構成になります。

## MSX Memory Architecture

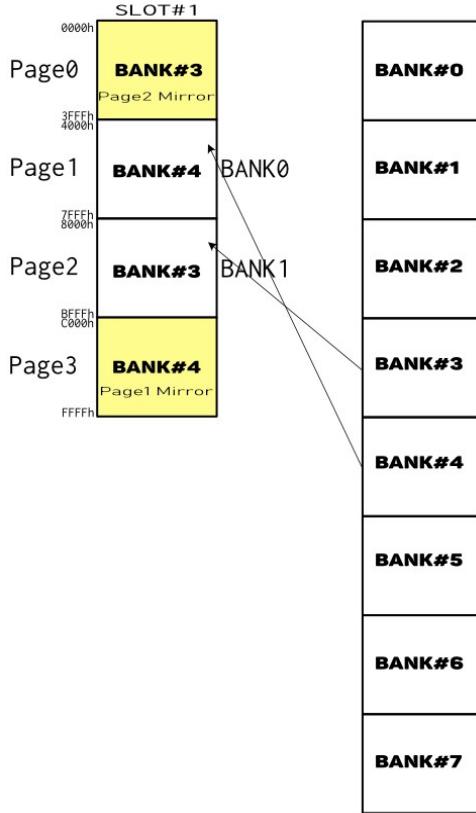


図 4.2-1.ASCII-16K タイプ  
1Mbit メガ ROM のイメージ

図 4.2-1. は、SLOT#1 に装着した 1Mbit メガ ROM のイメージです。SLOT#1 の Page0 には Page2 のミラーが、Page3 には Page1 のミラーが出現しています。

Page1 の中に BANK0 が、Page2 の中に BANK1 が存在しています。

図右側に 8 個並んでいる BANK#0～#7 が ROM になります。1 つ 16KB で 8 個あるので  $16\text{KB} \times 8 \text{ 個} = 128\text{KB}$  です。

図左側と図右側を結ぶ 2 本の矢印がバンク選択レジスタとなります。BANK0～BANK1 のそれぞれにバンク選択レジスタが存在し、BANK#0～#7 の任意の BANK# を指定できます。そのため、同じ BANK# を同時に複数の BANK に出現させることも出来ます。

バンク選択レジスタは、Memory mapped I/O となっています。バンク選択レジスタのアドレスを表 4.2-1. にまとめます。

表 4.2-1.ASCII-16K バンク選択レジスタのアドレス

BANK 番号	対応するバンク選択レジスタのアドレス
BANK0 (4000h-7FFFh)	6000h-67FFh (初期値 BANK#0)
BANK1 (8000h-BFFFh)	7000h-77FFh (初期値 BANK#0)

## MSX Memory Architecture

バンク選択レジスタのアドレスが 2048byte もの範囲を示していますが、実体は 1byte しかありません。そのため、 $16\text{KB} \times 256 = 4096\text{KB} = 4\text{MB}$  が最大容量となります。範囲を持っている理由は、メガコンの回路をシンプルにするためだと思います。なぜシンプルになるのかについては、ハードウェアの章で説明します。

ご覧のように、バンク選択レジスタはすべて Page1 に存在しています。Page2 にある BANK1 のバンク選択レジスタも Page1 に存在するため、例えば Page1 に本体の RAM のスロットを出現させ、Page2 にこのメガ ROM のスロットを出現させてるときに、BANK1 を切り替える場合は、一時的に Page1 をメガ ROM のスロットに切り替えて BANK1 のバンク選択レジスタを書き替えてから、Page1 を本体の RAM のスロットへ戻すといった処理が必要になりますのでご注意下さい。

バンク選択レジスタは、書き込み専用です。読み出した場合はそのアドレスに対応した ROM の内容が読み出されます。

バンク選択レジスタが Page1 に集まっている理由ですが、バックアップ用 SRAM への配慮かもしれません。バックアップ用 SRAM に対応しているメガコンもあります。一部のバンクにアクセスした場合にのみメガコンを経由した /WE が L になる構造です。SRAM にアクセスする場合にバンク選択レジスタがあると邪魔になるので、SRAM は Page2 に出現させてアクセスすることを想定しているのかもしれません。

### 4.3. Konami-8K タイプ

Konami のゲームで、SCC 搭載前に利用されていたメガ ROM タイプです。

Konami-8K タイプは、ROM を 8KB 単位の領域に区切り、連番を振っています。これを BANK0 (4000h-5FFFh), BANK1 (6000h-7FFFh), BANK2 (8000h-9FFFh), BANK3 (A000h-BFFFh) に出現させてアクセスする方式です。振った連番の N 番目を、ここでは BANK#N と表記します。

BANK の切り替え方は ASCII-8K 同じです。ただし、バンク選択レジスタのアドレスが異なります。Konami-8K タイプのバンク選択レジスタのアドレスを、表 4.3-1.にまとめます。

表 4.3-1.Konami-8K バンク選択レジスタのアドレス

BANK 番号	対応するバンク選択レジスタのアドレス
BANK0 (4000h-5FFFh)	なし (BANK#0 固定)
BANK1 (6000h-7FFFh)	6000h-7FFFh (初期値 BANK#1)
BANK2 (8000h-9FFFh)	8000h-9FFFh (初期値 不定)
BANK3 (A000h-BFFFh)	A000h-BFFFh (初期値 不定)

## MSX Memory Architecture

ASCII-8K と異なり、BANK0 は BANK#0 に固定です。異なる BANK# を指定できません。

BANKN のバンク選択レジスタは、その BANKN アドレス範囲の中に存在しています。切り替えたい BANKN のスロットが Z80 メモリ空間に出現している状態なら、その BANKN に出現する BANK#N を指定できることになります。ROMだけの場合、ASCII-8K よりも使いやすいと思います。

扱える容量は、最大 512KB までです。つまり最大 64[BANK]までですね。BANK#N の N は 0 ~63 の範囲を採ります。

BANK0～BANK3 は、Page1 と Page2 に存在していますが、ASCII-8K と同様に Page0 には Page2(BANK2 及び BANK3)の、Page3 には Page1(BANK0 及び BANK1)のミラーが出現しています。

### 4.4. Konami SCC タイプ

Konami の ROM ゲームで、SCC 対応と書かれているものはすべてこれに該当します。 MegaFlashROM SCC 等、このタイプのメガ ROM コントローラー互換のコントローラーを搭載しているものもあり、かつ SCC 音源を利用できるため、このタイプで自作ゲームを作成してリリースしている方もいます。

BANK の切り替え方は ASCII-8K と同じです。ただし、バンク選択レジスタのアドレスが異なります。Konami-SCC タイプのバンク選択レジスタのアドレスを、表 4.4-1. にまとめます。

表 4.4-1.Konami-SCC バンク選択レジスタのアドレス

BANK 番号	対応するバンク選択レジスタのアドレス
BANK0 (4000h-5FFFh)	5000h-57FFh (初期値 BANK#0)
BANK1 (6000h-7FFFh)	7000h-77FFh (初期値 BANK#1)
BANK2 (8000h-9FFFh)	9000h-97FFh (初期値 BANK#2)
BANK3 (A000h-BFFFh)	B000h-B7FFh (初期値 BANK#3)

ASCII-8K と異なり、BANKN のバンク選択レジスタは、その BANKN アドレス範囲の中に存在しています。切り替えたい BANKN のスロットが Z80 メモリ空間に出現している状態なら、その BANKN に出現する BANK#N を変更できることになります。ROMだけの場合、ASCII-8K よりも使いやすいと思います。

## MSX Memory Architecture

扱える容量は、最大 512KB、つまり最大 64[BANK]までです。BANK#N の N は 0~63 の範囲を採ります。

BANK0~BANK3 は、Page1 と Page2 に存在していますが、ASCII-8K と同様に Page0 には Page2(BANK2 及び BANK3)の、Page3 には Page1(BANK0 及び BANK1)のミラーが出現しています。

Konami-SCC タイプのメガコンは、SCC 音源を搭載しています。SCC 音源の制御用レジスタが存在します。バンク選択レジスタは、0~63 の範囲しか採りませんが、BANK2 のバンク選択レジスタに 63 (3Fh) を書き込むと、BANK2 に表 4.4-2. に示すような SCC 音源のレジスタ空間が出現します。つまり、BANK2 に BANK#63 を出現させることは出来ません。BANK#63 にアクセスしたい場合は、BANK0/1/3 のいずれかをご利用下さい。

表 4.4-2. SCC 音源のレジスタ(BANK2-BANK#63)

アドレス	サイズ(byte)	意味
8000h-8FFFh	4096	未使用??
9000h-97FFh	2048	BANK2 のバンク選択レジスタ。実体は 1byte。書き込み専用。
9800h-981Fh	32	ウェーブメモリー 0 (Ch.A 用)、読み書き可能。
9820h-983Fh	32	ウェーブメモリー 1 (Ch.B 用)、読み書き可能。
9840h-985Fh	32	ウェーブメモリー 2 (Ch.C 用)、読み書き可能。
9860h-987Fh	32	ウェーブメモリー 3 (Ch.D/E 用)、読み書き可能。
9880h-9881h	2	Ch.A 周波数レジスタ(12bit)、書き込み専用。
9882h-9883h	2	Ch.B 周波数レジスタ(12bit)、書き込み専用。
9884h-9885h	2	Ch.C 周波数レジスタ(12bit)、書き込み専用。
9886h-9887h	2	Ch.D 周波数レジスタ(12bit)、書き込み専用。
9888h-9889h	2	Ch.E 周波数レジスタ(12bit)、書き込み専用。
988Ah	1	Ch.A 音量 (4bit)、書き込み専用。
988Bh	1	Ch.B 音量 (4bit)、書き込み専用。
988Ch	1	Ch.C 音量 (4bit)、書き込み専用。
988Dh	1	Ch.D 音量 (4bit)、書き込み専用。
988Eh	1	Ch.E 音量 (4bit)、書き込み専用。
988Fh	1	出力スイッチ、書き込み専用。 bit0: Ch.A (0: ミュート, 1: 出力) bit1: Ch.B (0: ミュート, 1: 出力) bit2: Ch.C (0: ミュート, 1: 出力)

## MSX Memory Architecture

		bit3: Ch.D (0: ミュート, 1: 出力) bit4: Ch.E (0: ミュート, 1: 出力)
9890h-989Fh	16	9880h-988Fh のミラー。
98A0h-98BFh	32	未使用。読み書き不可。
98C0h-98DFh	32	未使用??
98E0h-98FFh	32	モードレジスタ。書き込み専用。実体は 1byte。 bit0: 周波数レジスタの扱いの指定 1 bit1: 周波数レジスタの扱いの指定 2 bit5: 周波数レジスタ書き込みで波形ポインタリセット bit6: 全 Ch. 波形ローテート bit7: Ch.D 波形ローテート
9900h-9FFFh	1792	未使用??

書き込み専用と書かれているレジスタを読み出しても意味のある値は読み出せません。

SCC 音源の説明は、本書のスコープ外となりますので省略させていただきます。

## 4.5. Konami SCC-I タイプ

いわゆる Snatcher サウンドカートリッジ、SD Snatcher サウンドカートリッジが該当します。実際、ROM ではなく DRAM が搭載されています。

メガ ROM コントローラーとしては、Konami-SCC タイプと同じですので、バンク選択レジスタなどについては 4.4. Konami SCC タイプを参照下さい。一方で、SCC 搭載による違いがありますので、本節ではその違いについて説明したいと思います。

Snatcher サウンドカートリッジは、BANK#0～BANK#7 として 64KB の DRAM が搭載されています。一方で、SD Snatcher サウンドカートリッジは、BANK#8～BANK#15 として 64KB の DRAM が搭載されています。

サウンドカートリッジの基板には DRAM を追加出来るパターンが存在しており、4bit × 64K の DRAM を 2 つ追加出来るようになっています。基板としては 4bit × 64K の DRAM を 4 つ搭載可能になっていて、Snatcher と SD Snatcher では DRAM が取り付けられている場所が異なっています。空いてる 2 力所に DRAM を追加すると、BANK#0～BANK#15 として 128KB の DRAM が搭載されているサウンドカートリッジになります。

Konami SCC-I タイプでは、BFFEh, BFFFh に動作モード設定レジスタが存在しています。BFFEh と BFFFh は同じレジスタで、実体は 1byte しかありません。図 4.5-1. に動作モード設定レジスタのビットマップを示します。初期値は 00h のようです。

## MSX Memory Architecture

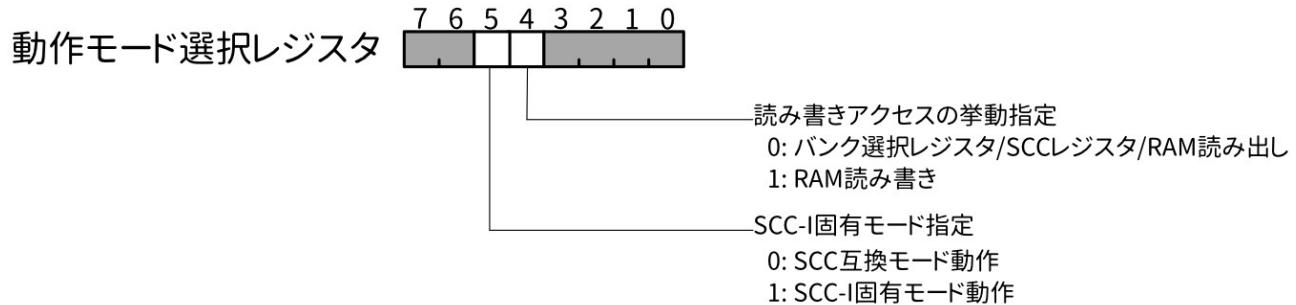


図 4.5-1. SCC-I 動作モード選択レジスタ

動作モード選択レジスタは、書き込み専用です。動作モード選択レジスタの設定値に関わらず常にアクセス出来ます。

バンク選択レジスタや、SCC/SCC-I 音源のレジスタにアクセスしたい場合は、bit4 を 0 にする必要があります。bit4 を 1 にすると、サウンドカートリッジ内の RAM への書き込みが出来るようになります。例えば、9123h 番地に書き込んだ場合、BFFEh の bit4 が 0 ならば BANK2 のバンク選択レジスタへの書き込みとなりますが、BFFEh の bit4 が 1 ならば BANK2 に出現している RAM への書き込みとなります。

SCC-I の SCC 音源部は、Konami SCC タイプと互換のモードと、サウンドカートリッジで拡張されたモードの 2 種類が存在します。ここでは、前者を互換モード、後者を固有モードと呼びます。

互換モードも Konami SCC タイプと比べて若干の差があります。SCC 音源のレジスタは表 4.5-1 のようになります。差異の部分を赤字にしてあります。

表 4.5-1. 互換モードの SCC 音源のレジスタ(BANK2-BANK#63)

アドレス	サイズ(byte)	意味
8000h-8FFFh	4096	未使用??
9000h-97FFh	2048	BANK2 のバンク選択レジスタ。実体は 1byte。書き込み専用。
9800h-981Fh	32	ウェーブメモリー 0 (Ch.A 用)、読み書き可能。
9820h-983Fh	32	ウェーブメモリー 1 (Ch.B 用)、読み書き可能。
9840h-985Fh	32	ウェーブメモリー 2 (Ch.C 用)、読み書き可能。
9860h-987Fh	32	ウェーブメモリー 3 <i>書き込み時は Ch.D/E 共用。読み出し時は Ch.D。</i>
9880h-9881h	2	Ch.A 周波数レジスタ(12bit)、書き込み専用。

## MSX Memory Architecture

9882h-9883h	2	Ch.B 周波数レジスタ(12bit)、書き込み専用。
9884h-9885h	2	Ch.C 周波数レジスタ(12bit)、書き込み専用。
9886h-9887h	2	Ch.D 周波数レジスタ(12bit)、書き込み専用。
9888h-9889h	2	Ch.E 周波数レジスタ(12bit)、書き込み専用。
988Ah	1	Ch.A 音量 (4bit)、書き込み専用。
988Bh	1	Ch.B 音量 (4bit)、書き込み専用。
988Ch	1	Ch.C 音量 (4bit)、書き込み専用。
988Dh	1	Ch.D 音量 (4bit)、書き込み専用。
988Eh	1	Ch.E 音量 (4bit)、書き込み専用。
988Fh	1	出力スイッチ、書き込み専用。 bit0: Ch.A (0: ミュート, 1: 出力) bit1: Ch.B (0: ミュート, 1: 出力) bit2: Ch.C (0: ミュート, 1: 出力) bit3: Ch.D (0: ミュート, 1: 出力) bit4: Ch.E (0: ミュート, 1: 出力)
9890h-989Fh	16	9880h-988Fh のミラー。
98A0h-98BFh	32	ウェーブメモリー 4 (Ch.E 用)、読み出し専用。
98C0h-98DFh	32	モードレジスタ。書き込み専用。実体は 1byte。 bit0: 周波数レジスタの扱いの指定 1 bit1: 周波数レジスタの扱いの指定 2 bit5: 周波数レジスタ書き込みで波形ポインタリセット bit6: 全 Ch. 波形ローテート bit7: 無効
98E0h-98FFh	32	未使用??
98E0h-9FFFh	1792	未使用??

書き込み専用と書かれているレジスタを読み出しても意味のある値は読み出せません。

9860h-987Fh は、書き込み時には SCC との互換のために、Ch.D のウェーブメモリーと、Ch.E のウェーブメモリーの両方に同じ値を書きに行きます。9860h-987Fh を読み出すと Ch.D のウェーブメモリーの値が返されます。Ch.E のウェーブメモリーを読みたい場合は、98A0h-98BFh が使えます。互換モードしか使っていない場合は、9860h-987Fh の読み出しどと、98A0h-98BFh の読み出しどは同じ値を返しますが、一度固有モードに切り替えて Ch.E のウェーブメモリーを書き替えてから互換モードに戻して読むと、ちゃんと異なる波形を読み出せるようです。

Konami-SCC タイプと、モードレジスタのアドレスが異なっていますのでご注意下さい。

次に固有モードの説明に移ります。

## MSX Memory Architecture

固有モードの SCC 音源レジスタは、互換モードと異なり、BANK3 に出現します。この方が BANK0~2 を連続的に使って便利なので、このように変更したのだと思います。SCC 音源のレジスタは、BANK#128 (厳密には、バンク選択レジスタの bit7 = 1 であるすべての BANK# のため、BANK#128~BANK#255 の BANK#を指定しても同じ)になります。固有モードでは、Ch.D と Ch.E に異なる波形を設定できます。レジスタのアドレスを、表 4.5-2.にまとめます。

表 4.5-2. 固有モードの SCC 音源のレジスタ(BANK3-BANK#128)

アドレス	サイズ(byte)	意味
A000h-AFFFh	4096	未使用??
B000h-B7FFh	2048	バンク選択レジスタ。実体は 1byte。書き込み専用。
B800h-B81Fh	32	ウェーブメモリー 0 (Ch.A 用)、読み書き可能。
B820h-B83Fh	32	ウェーブメモリー 1 (Ch.B 用)、読み書き可能。
B840h-B85Fh	32	ウェーブメモリー 2 (Ch.C 用)、読み書き可能。
B860h-B87Fh	32	ウェーブメモリー 3 (Ch.D 用)、読み書き可能。
B880h-B89Fh	32	ウェーブメモリー 4 (Ch.E 用)、読み書き可能。
B8A0h-B8A1h	2	Ch.A 周波数レジスタ(12bit)、書き込み専用。
B8A2h-B8A3h	2	Ch.B 周波数レジスタ(12bit)、書き込み専用。
B8A4h-B8A5h	2	Ch.C 周波数レジスタ(12bit)、書き込み専用。
B8A6h-B8A7h	2	Ch.D 周波数レジスタ(12bit)、書き込み専用。
B8A8h-B8A9h	2	Ch.E 周波数レジスタ(12bit)、書き込み専用。
B8AAh	1	Ch.A 音量 (4bit)、書き込み専用。
B8ABh	1	Ch.B 音量 (4bit)、書き込み専用。
B8ACh	1	Ch.C 音量 (4bit)、書き込み専用。
B8ADh	1	Ch.D 音量 (4bit)、書き込み専用。
B8AEh	1	Ch.E 音量 (4bit)、書き込み専用。
B8AFh	1	出力スイッチ、書き込み専用。 bit0: Ch.A (0: ミュート, 1: 出力) bit1: Ch.B (0: ミュート, 1: 出力) bit2: Ch.C (0: ミュート, 1: 出力) bit3: Ch.D (0: ミュート, 1: 出力) bit4: Ch.E (0: ミュート, 1: 出力)
B8B0h-B8BFh	16	B8A0h-B8AFh のミラー。
B8C0h-B8DFh	32	モードレジスタ。書き込み専用。実体は 1byte。 bit0: 周波数レジスタの扱いの指定 1

## MSX Memory Architecture

		bit1: 周波数レジスタの扱いの指定 2 bit5: 周波数レジスタ書き込みで波形ポインタリセット bit6: 全 Ch. 波形ローテート bit7: 無効
B8E0h-BFFDh	1822	未使用??
BFFEh-BFFFh	2	動作モード選択レジスタ。実体は 1byte。どちらのアドレスも同じ。

書き込み専用と書かれているレジスタを読み出しても意味のある値は読み出せません。

SCC 音源の説明は、本書のスコープ外となりますので省略させていただきます。

## 4.6. Panasonic タイプ

Panasonic 製本体(FS-A1FX/WX/WSX/ST/GT)に搭載されているメガ ROM タイプです。内蔵ソフトや、MSX-JE 等が格納されていて、SLOT#3-3 に接続されています。

BANK のサイズは 8KB で、ASCII-8K タイプに似ていますが、Page0 と Page3 はミラーでは無く、それらも独立したバンク選択レジスタを持っている点が大きく異なります。また、バンク選択レジスタも、1 つの BANK あたり 9bit あり、BANK#0～BANK#511 の範囲で選択できるようになっています。8[KB] × 512[BANK] = 4096[KB] = 4[MB] の空間を扱えることになります。

おそらく、最も大容量のメモリが接続されたのは FS-A1GT だと思いますが、その FS-A1GT でさえも 4MB 空間をフルに使っているわけではありません。ROM-2MB、DRAM-512KB、SRAM-32KB が、その 4MB 空間の一部分という形で接続されているようです(SRAM は、MSX-JE の学習辞書用のバッテリーバックアップされた RAM です)。

スロットから見たイメージを図 4.6-1. に示します。

## MSX Memory Architecture

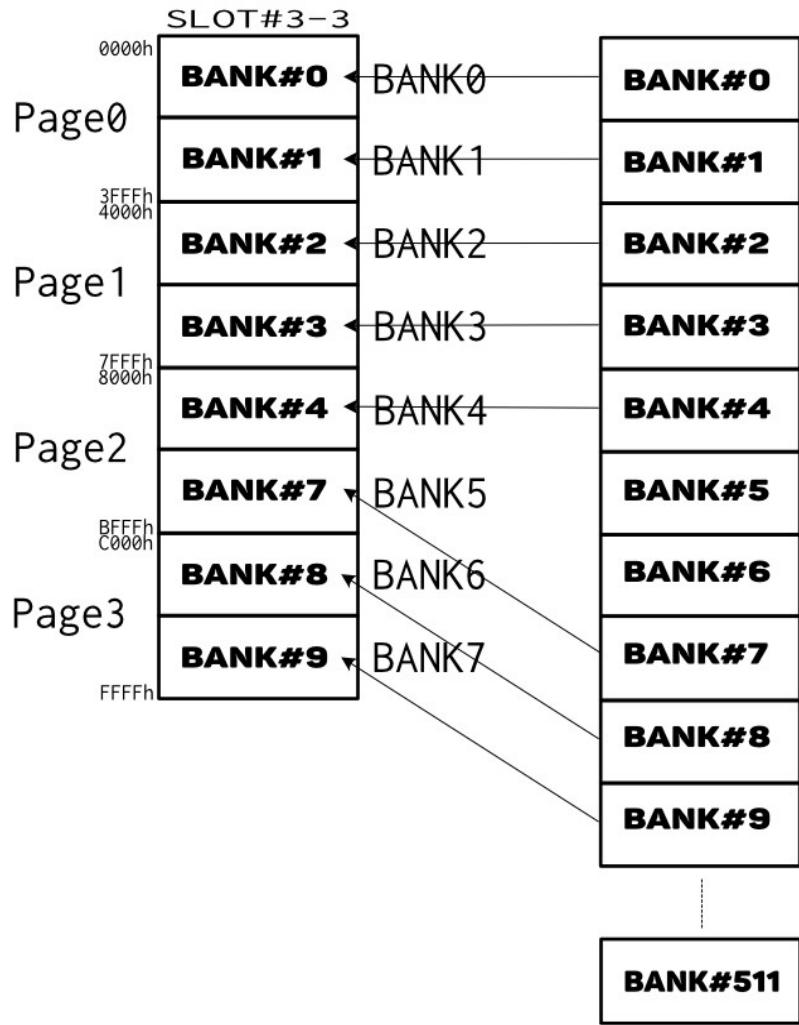


図 4.6-1.

Panasonic タイ  
プメガ ROM の

イメージ

バンクレジスタのアドレスを表 4.6-1.に示します。

表 4.6-1. Panasonic タイプ メガ ROM バンクレジスタ

アドレス	サイズ(byte)	意味
6000h-63FFh	1024	実体は 1byte。書き込み専用。 BANK0 のバンク選択レジスタ下位 8bit
6400h-67FFh	1024	実体は 1byte。書き込み専用。 BANK1 のバンク選択レジスタ下位 8bit
6800h-6BFFh	1024	実体は 1byte。書き込み専用。 BANK2 のバンク選択レジスタ下位 8bit
6C00h-6FFFh	1024	実体は 1byte。書き込み専用。 BANK3 のバンク選択レジスタ下位 8bit

## MSX Memory Architecture

7000h-73FFh	1024	実体は 1byte。書き込み専用。 BANK4 のバンク選択レジスタ下位 8bit
7400h-77FFh	1024	実体は 1byte。書き込み専用。 BANK6 のバンク選択レジスタ下位 8bit
7800h-7BFFh	1024	実体は 1byte。書き込み専用。 BANK5 のバンク選択レジスタ下位 8bit
7C00h-7FEFh	1008	実体は 1byte。書き込み専用。 BANK7 のバンク選択レジスタ下位 8bit
7FF0h	1	読み出し専用。 BANK0 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF1h	1	読み出し専用。 BANK1 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF2h	1	読み出し専用。 BANK2 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF3h	1	読み出し専用。 BANK3 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF4h	1	読み出し専用。 BANK4 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF5h	1	読み出し専用。 BANK5 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF6h	1	読み出し専用。 BANK6 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。
7FF7h	1	読み出し専用。 BANK7 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になつていないと読み出し出来ない。

## MSX Memory Architecture

		い。
7FF8h	1	読み出し・書き込み可能。 バンク選択レジスタ上位 1bit。 bit0: BANK0 のバンク選択レジスタの上位 1bit。 bit1: BANK1 のバンク選択レジスタの上位 1bit。 bit2: BANK2 のバンク選択レジスタの上位 1bit。 bit3: BANK3 のバンク選択レジスタの上位 1bit。 bit4: BANK4 のバンク選択レジスタの上位 1bit。 bit5: BANK5 のバンク選択レジスタの上位 1bit。 bit6: BANK6 のバンク選択レジスタの上位 1bit。 bit7: BANK7 のバンク選択レジスタの上位 1bit。 7FF9h の bit4 が 1 になっていないと読み書き出来ない。
7FF9h	1	モードレジスタ。読み出し・書き込み可能。 bit0: 無効 bit1: 無効 bit2: 7FF0h-7FF7h 有効化 bit3: 7FF9h 読み出し有効化 bit4: 7FF8h 有効化 bit5: 無効 bit6: 無効 bit7: 無効 このレジスタを読み出すためには、まずこのレジスタの bit3 に 1 を書き込む必要がある。

参考資料によると、FS-A1WX/WSX では、7FF8h は存在しないようです。7FF9h の bit4 も無効です。FS-A1FX は不明。

バンク選択レジスターの下位 8 ビットを書き込む領域ですが、綺麗な順番になっていません。表で赤字で示したように、BANK5 と BANK6 の順番が入れ替わっていますのでご注意下さい。

では、このメガコンを使って、何を接続しているのか？ FS-A1GT の場合について下記にまとめます。

FS-A1GT には合計 2MB もの ROM が搭載されていますが、このメガコンでは搭載 ROM のほぼすべてにアクセス出来るように接続されています（16 ドット漢字 ROM だけは含まれて無さそうです）。それだけでなく、内蔵の RAM もこのメガコンを経由してアクセス出来るようになっているようです。

分かる範囲で下記にまとめます。<<現在調査中>>

BANK#	内容
-------	----

## MSX Memory Architecture

BANK#0-#39	内蔵ソフト 320KB (40BANK) 16KB (2BANK) ごとに WSXSEGxxPx のシグネチャが付いている
BANK#40-#43	MAIN-ROM 32KB(4BANK)
BANK#44-#47	内蔵ソフト 32KB (4BANK) 16KB (2BANK) ごとに WSXSEGxxPx のシグネチャが付いている
BANK#48-#55	DiskBIOS 64KB (8BANK)
BANK#56-#57	SUB-ROM 16KB (2BANK)
BANK#58-#61	漢字 Driver 32KB (4BANK)
BANK#62-#63	MSX-MUSIC 16KB (2BANK)
BANK#64-#127	MSX-JE 変換辞書 512KB (64BANK)
BANK#128-#131	MSX-JE 変換辞書用 SRAM 32KB (4BANK)
BANK#132-#159	未接続 224KB (28BANK)
BANK#160-#191	ROM Disk 後半?? 256KB (32BANK)
BANK#192-#255	未接続 512KB (64BANK)
BANK#256-#319	ROM Disk 前半 512KB (64BANK)
BANK#320-#383	未接続 512KB (64BANK)
BANK#384-#447	MAIN-RAM 512KB (64BANK)
BANK#448-#511	MAIN-RAM 512KB (64BANK)、BANK#384-#447 のミラー

turboR のプライマリマッパー RAM は、セグメント#の値が大きい方から順に 4 セグメントが、 BIOS コピー領域として確保されています。SLOT#3-0 のプライマリーマッパースロットに対しては、 R800-DRAM モードではその 4 セグメントが書き込み禁止になるという特徴が有ります。これは、 おそらく ROM カートリッジソフトや、 MSX-DOS1 用のメモリーマッパー対応アプリケーションが、 誤って BIOS コピーを上書きしてしまわないように保護しているのだと思います。

しかし、SLOT#3-3 に現れる「プライマリマッパー RAM の内容が出現する BANK#」は、当該 BIOS コピー領域も含め全域書き込みできるという特徴があります。

## 4.7. パナアミューズメントカートリッジ

Panasonic から発売されていた PAC, FM-PAC の 2 種類が販売されていました。ゲームデータのセーブ用に 8KB のバッテリーバックアップされた SRAM を搭載していますが、デフォルトでは無効になっており、表 4.7-1. に示す所定の処理を行うと SRAM が出現する構造になっています。

## MSX Memory Architecture

表 4.7-1. PAC の SRAM 切り替え

所定の処理	効果
5FFEh に 4Dh を書き込み 5FFFh に 69h を書き込む	4000h-5FFFh に SRAM を出現させる
5FFEh または 5FFFh を上記以外にする	4000h-5FFFh の SRAM を切り離す

8KB を 1KB 単位に 8 つの領域に分けていていることになっており、PAC 対応ゲームソフトの箱等に、この 8 つの領域のどこを使用するのか示すアイコンが印刷されていました。この 8 つの領域には管理情報などなく、各ゲームメーカーが自由に使っているため、プログラムからどの領域が使用中であるかを判別する手段はありません。同じ領域を上書きするかどうかは、ユーザー任せでした。

FM-PAC の場合、SRAM を切り離した状態では MSX-MUSIC の ROM が出現するので、4018h ~ にあるシグネチャ "PAC2OPLL" を検出すれば存在を確認できます。

PAC の場合、SRAM を切り離すと未接続状態のような挙動になるため、読み出しても意味のある値は得られません。どこを読んでも概ね FFh が返ってきます。検出する場合は、「4000h, 4001h に 41h, 42h が無いこと」「RAM でないこと」を確認した後、5FFEh, 5FFFh に 4Dh, 69h を書き込んでみて、「4000h-5FFDh」が RAM に変わっていることを確認すれば OK です。RAM かどうかの確認は、確認したいアドレスを HL に設定していたとすると、下記のコードで確認できます。

```
LD A, [HL]
CPL
LD [HL], A
CP A, [HL]
CPL
LD [HL], A
```

もともとの値を A レジスタに読み込み、全ビット反転した後に書き戻し、書き戻した値と一致していれば RAM 、一致しなければ RAM ではない (ROM や未接続など) と分かります。RAM だった場合に値を破壊してしまうと困るので、最後にまた全ビット反転して元の値に戻し、書き戻しています。CPL 命令と LD 命令は、Z フラグを変化させないので、CP 命令による結果で確定した Z フラグの値を見ることが出来ます。この処理を実行した後に Z フラグが立つていれば RAM 、立っていないければ RAM ではないということですね。

## 5. ROM カートリッジの動作

MSX は、電源を投入すると、まず SLOT#0 または SLOT#0-0 にある MAIN-ROM から起動します。MAIN-ROM 上にある BIOS プログラムは、まず Page2, Page3 の RAM の搭載状況を調べ、Page2 及び Page3 に RAM を出現させた状態ですべてのスロットに対して ROM の検索を行います。ROM の検索は、SLOT#0→SLOT#1→SLOT#2→SLOT#3 の順で行われます。拡張スロットは、SLOT#X-0→SLOT#X-1→SLOT#X-2→SLOT#X-3 ですね。

MSX では、ROM カートリッジの先頭に ROM ヘッダを書き込んでおく決まりになっています。適切な ROM ヘッダを書き込んでおけば、それを解釈した BIOS が、ROM カートリッジ内の必要なプログラムを呼び出してくれます。ROM ヘッダは、Page1 の先頭(4000h～)か、Page2 の先頭(8000h～)に置くことになっています。すべて機械語のプログラムは Page1 の先頭(4000h～)を、MSX-BASIC で作られたプログラムの ROM 化は Page2 の先頭(8000h～)を使うのが良いでしょう。Page1→Page2 の順で検索されます。ROM ヘッダを表 5.1. にまとめます。

表 5.1. ROM ヘッダ

アドレス	サイズ(byte)	名前
+0000h	2	ID
+0002h	2	INIT
+0004h	2	STATEMENT
+0006h	2	DEVICE
+0008h	2	TEXT
+000Ah	6	RESERVED

ID は、ROM 出あることを示すシグネチャです。41h, 42h ( "AB" ) を書き込みます。

INIT は、初期化ルーチンのアドレスを書き込みます。初期化を行わない場合は、0000h にしておいて下さい。一方で、ゲームカートリッジなどでは、ここにゲームのエントリーポイントとなるアドレスを書き込んで下さい。

STATEMENT は、MSX-BASIC の CALL 命令を拡張する場合に、その拡張ルーチンのアドレスを書き込みます。CALL 命令を拡張しないのであれば、0000h にしておいて下さい。

DEVICE は、デバイス拡張ルーチンのアドレスを書き込みます。不要であれば 0000h にしておいて下さい。

TEXT は、カートリッジ内にある MSX-BASIC で書かれたプログラムの格納アドレスを指定します。MSX-BASIC プログラムを書き込んだカートリッジでない場合は 0000h にしておいて下さい。

RESERVED は、将来の拡張のために用意された領域です。0 で埋めておいて下さい。

### 5.1. 16KB の ROM カートリッジプログラムを作る

ディスクを使わないゲームソフトは、INIT に書かれているアドレスから起動するのが一般的です。シンプルな ROM プログラムの例を下記に示します。

ROM\_sample001.ASM

```
; =====
;      ROM_sample001.ASM
; -----
;      Jan./31/2020 HRA!
; =====

chput  = 0x00A2
himem  = 0xfc4a

          org      0x4000
; =====
;      ROM Header
; =====
rom_header_id:
          ds      "AB"
rom_header_init:
          dw      entry_point
rom_header_statement:
          dw      0
rom_header_device:
          dw      0
rom_header_text:
          dw      0
rom_header_reserved:
          space   0x0010 - 0x000A, 0

; =====
;      Program entry point
; =====
entry_point:
          ; Initialize Stack Pointer
          ld      sp, [himem]

main_loop:
          ; Put message
          ld      hl, display_message
          call   puts
          jp      main_loop

; =====
;      puts
;      input)
;      HL .... address of target string (ASCII-Z)
; =====
puts:
          ld      a, [hl]
          inc   hl
          or     a, a
          ret
          call   chput
          jp      puts

display_message:
          ds      "Hello, world!! "
          db      0
```

## MSX Memory Architecture

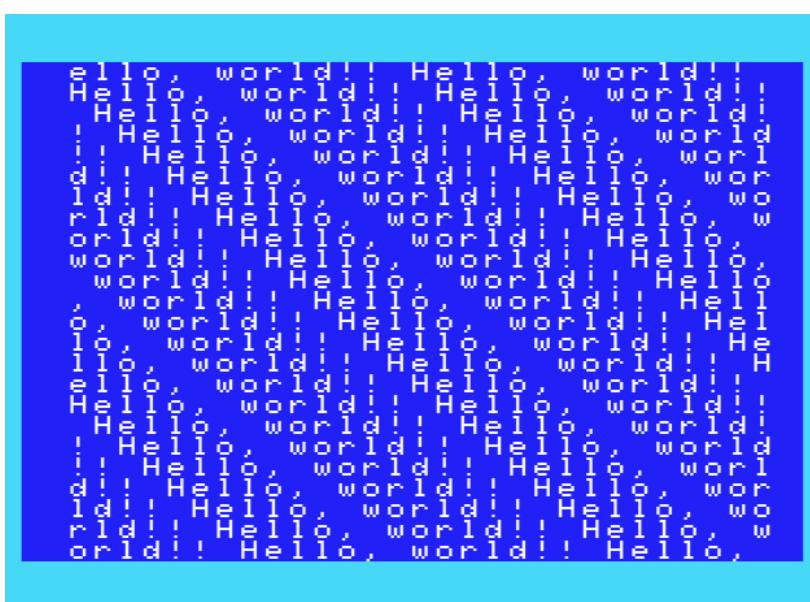
```
align      16384
```

アセンブラー ZMA にパスを通して、下記コマンドによってアセンブルしてください。

```
zma ROM_sample001.ASM ROM_sample001.ROM
```

アセンブルすると、ROM\_sample001.ROM という 16KB の ROM イメージファイルが出来上がります。エミュレータで起動できますので、動かしてみて下さい。図 5.1. のように表示されれば期待通りの動作となります。

図 5.1.



ROM\_sample001.ROM の実行イメージ

では、この ROM\_sample001.ASM について、順に説明していきたいと思います。

まず、冒頭です。

```
org      0x4000
```

ZMA は、ソースコードを上から順にアセンブルして出力ファイルに書き出していきます。その際に、「Z80 が何番地と認識しているコードなのか」を内部で計算しながらアセンブルしています。ラベルの絶対アドレスを計算するためです。org 0x4000 とすることで、この次の行は 0x4000 番地です、と ZMA に知らせることになります。

16KB の ROM は、アドレス信号線として A13~A0 の 14bit の信号を持っているはずです。これをカートリッジスロットの A13~A0 にそのまま接続して、A15・A14 は未接続、ROM の /CS には

## MSX Memory Architecture

カートリッジスロットの/SLTSL を接続、ROM の/OE にはカートリッジスロットの/RD を接続でカートリッジを作る場合、Z80 からは Page0, Page1, Page2, Page3 のすべてにこの ROM の内容が現れる構成になります。この中で BIOS が一番最初に調べるのが Page1 なので、Page1 に置かれている前提のプログラムにします。Page1 の先頭が 0x4000 なので、org 0x4000 にしてあります。

```
rom_header_id:  
    ds          "AB"  
rom_header_init:  
    dw          entry_point  
rom_header_statement:  
    dw          0  
rom_header_device:  
    dw          0  
rom_header_text:  
    dw          0  
rom_header_reserved:  
    space      0x0010 - 0x000A, 0
```

0x4000～0x400F には ROM ヘッダを書き込んでおく必要があることは先に述べました。これは、その ROM ヘッダです。

rom\_header\_id は、表 5.1. の ID に対応します。ここは、どんな ROM カートリッジでも必ず "AB" です。BIOS に対して「ここに ROM ヘッダがありますよ」と教えるための目印ですね。

rom\_header\_init は、表 5.1. の INIT に対応します。ここにエントリーポイントのアドレスを記入します。このプログラムでは entry\_point というラベルになっていますね。

その他は、今回使わないので 0 で埋めておきます。

```
entry_point:  
    ; Initialize Stack Pointer  
    ld          sp, [himem]
```

ラベル entry\_point は、このプログラムのエントリーポイントです。まず真っ先にスタックポインタを初期化します。普通のプログラムであれば、himem に格納されている値を使えば良いでしょう。himem には、BIOS ワークエリアの下限アドレスが格納されています。

```
main_loop:  
    ; Put message
```

## MSX Memory Architecture

ld	hl, display_message
call	puts
jp	main_loop

ラベル main\_loop は、その名の通りメインループの入り口に相当します。

サブルーチン puts は、この次に登場しますが、HL に文字列の先頭アドレスを指定して呼び出すと、その文字列を表示 (MSX-BASIC の PRINT 文のようなもの) するルーチンです。今回、図 5.1. に示したように Hello, world!! と繰り返し表示するので、ラベル display\_message には "Hello, world!!" が格納されています。

puts:	
ld	a, [hl]
inc	hl
or	a, a
ret	z
call	chput
jp	puts

これが puts の中身です。 BIOS には CHPUT というルーチンが用意されていて、動作を MSX-BASIC で書くと PRINT CHR\$( A ); のような動作になります。ここでいう A は、A レジスタですね。 puts では、HL レジスタが示すアドレスの内容を A レジスタに読み出し、0 かどうか調べて、0 であれば戻る、0 でなければ CHPUT を使って 1 文字表示して、を繰り返しています。

CHPUT は、全レジスタの内容が保存されますので、HL レジスタは維持されています。

display_message:	
ds	"Hello, world!! "
db	0

main\_loop のところで、HL に代入されている display\_message というラベルは、ここにあります。

ZMA では、ds 疑似命令は「Define String」です。Hello, world!! の各文字の ASCII コードがそのまま配置されます。db 疑似命令で 0 をおいてターミネータとしています。

ターミネータは、先ほどのサブルーチン puts が解釈するものですね。

## MSX Memory Architecture

align	16384
-------	-------

最後に align 命令です。ROM\_sample001.ASM は 16KB に満たない小さなプログラムです。これを 16KB になるようにパディングを追加する疑似命令です。

先ほど、CHPUT が使えたように、ROM の INIT が呼ばれる時点では、Page0 は MAIN-ROM (BIOS)、Page1 は ROM カートリッジの ROM、Page3 は BIOS ワークエリアなどが置かれている RAM になっていることが分かると思います。

Page2 は、機種によって異なります。

RAM 16KB の機種では、SLOT#0 または SLOT#0-0 になっています。SLOT#0/#0-0 の Page2 に配置される情報は、これまた機種によって異なるので、未接続だったり、何か内蔵ソフトの ROM だったり、いろいろあり得ます。いずれにしても、RAM ではありません。

RAM32KB 以上の機種では、Page2 の RAM のスロットになっています。

Page2 が RAM かどうかで、RAM32KB 以上なのか、RAM16KB 以下なのかを判別可能です。わざわざチェックしているソフトは少ないと思いますが、16KB でも動作して、32KB だとまた違った動作をするような場合は判別する必要があると思います。

さらに、CASIO PV-7 などの RAM 8KB の機種は、Page3 も E000h-FFFFh にしか RAM が存在していません。C000h-DFFFh は書き込めません。

RAM かどうかを調べる方法としては、調べたいアドレスから値を読み出してレジスタに保存。同じアドレスに反転した値を書き込んで、もう一度読み出して反転した値が読み出せれば RAM、読み出せなければ RAM でない (ROM や未接続など) 、RAM だった場合は保存していた値を書き戻す。という方法があります。BIOS が RAM の存在確認する際もそのようにしているようです。

Page2 と Page3 の RAM の SLOT# は、必ずしも同じとは限らないことに注意して下さい。同じだと決め打つプログラムは、危険です。

メモリーマッパー対応 RAM は、かならずそのスロットの Page0~3 すべてに RAM が存在しています。本体がメモリーマッパー対応 RAM である機種は、Page2 と Page3 に出現する RAM が同じ SLOT# になっています。MSX1 と一部の MSX2 は、本体 RAM がメモリーマッパー対応 RAM でないでご注意下さい。MSX2+以降向けのカートリッジであれば、Page2 と Page3 に出現している RAM が同じ SLOT# であると決め打っても問題ないのかもしれません。

ディスク BIOS のワークエリアとして、Page0~Page3 に対応する RAM の SLOT# を格納した RAMAD0~RAMAD3 という領域があります。しかし、ROM カートリッジが起動するタイミングでは、まだこの値は格納されていません。ディスク BIOS 自体も、ROM カートリッジと同じ実装で

## MSX Memory Architecture

あり、ディスク内蔵機では SLOT#3-X に搭載されている機種が殆どです。ディスク BIOS の初期化ルーチンが呼ばれる前なので、格納されていないわけです。一方で、外付けドライブを SLOT#1 に、作った ROM カートリッジを SLOT#2 に装着した場合は格納されているかもしれません。そのため、SP の初期化は HIMEM を参照して、存在しているかもしれないディスク BIOS 等他のモジュールのワークエリアを破壊しないようにするのが安全です。ゲームなどをするときは不要なデバイスを外して下さい、というお約束は、こういった特殊ケースによる不慮の事故を未然に防ぐための予防策なのです

## 5.2. メガ ROM カートリッジプログラムを作る

ここでは、ASCII-8K のメガ ROM で動作するプログラムを作ります。

Page0 にある MAIN-ROM の BIOS コードが Page2/Page3 を RAM に切り替えてから、各スロットの Page1 を探索していきます。メガ ROM カートリッジのスロットを探索する際に、メガ ROM カートリッジの INIT が呼ばれる仕組みです。その時点では、Page2 はまだ RAM なのです。

INIT から起動するのは、5.1. 16KB の ROM カートリッジプログラムを作ると同じですが、Page1 に現れている「メガ ROM の BANK0 と BANK1」には、ともに BANK#0 が出現しています。メガ ROM のスロットの Page2 にある BANK2 と BANK3 も BANK#0 が出現していますが、Z80 メモリ空間には、メガ ROM のスロットの Page2 は出現していません。その点に注意してプログラムする必要がある点が、5.1. 16KB の ROM カートリッジプログラムを作ると大きく異なる点です。

5.1. 16KB の ROM カートリッジプログラムを作る少し改造して、メガ ROM カートリッジプログラムに改造したプログラムの例を下記に示します。

### ROM\_sample002.ASM

```
; =====
;      ROM_sample002.ASM
; -----
;      Feb./1/2020 HRA!
; =====

chput      = 0x00A2
himem     = 0xfc4a

bank0_sel   = 0x6000
bank1_sel   = 0x6800
bank2_sel   = 0x7000
bank3_sel   = 0x7800

; =====
; =====
;      ROM BANK#0
; =====
; =====
```

## MSX Memory Architecture

```
        org      0x4000
; =====
; ROM Header
; =====
rom_header_id:
    ds      "AB"
rom_header_init:
    dw      entry_point
rom_header_statement:
    dw      0
rom_header_device:
    dw      0
rom_header_text:
    dw      0
rom_header_reserved:
    space   0x0010 - 0x000A, 0

; =====
; Program entry point
; =====
entry_point:
    ; Initialize Stack Pointer
    ld      sp, [himem]

main_loop:
    ; Put message (BANK#0 on BANK0)
    ld      hl, display_message1
    call    puts
    ; Put message (BANK#1 on BANK1)
    ld      a, 1                      ; BANK#1
    ld      [bank1_sel], a            ; BANK1 changes to BANK#1
    ld      hl, display_message2
    call    puts
    ; Put message (BANK#2 on BANK1)
    ld      a, 2                      ; BANK#2
    ld      [bank1_sel], a            ; BANK1 changes to BANK#2
    ld      hl, display_message3
    call    puts
    ; Put message (BANK#3 on BANK1)
    ld      a, 3                      ; BANK#3
    ld      [bank1_sel], a            ; BANK1 changes to BANK#3
    ld      hl, display_message4
    call    puts
    jp      main_loop

; =====
; puts
; input)
; =====
;       HL .... address of target string (ASCII-Z)
; =====
puts:
    ld      a, [hl]
    inc    hl
    or     a, a
    ret    z
    call   chput
    jp      puts

display_message1:
    ds      "BANK#0 on BANK0"
    db      0x0D, 0x0A, 0

    align  8192

; =====
; =====
; ROM BANK#1
; =====
; =====

        org      0x6000
```

## MSX Memory Architecture

```
display_message2:  
    ds          "BANK#1 on BANK1"  
    db          0x0D, 0x0A, 0  
    align      8192  
  
; ======  
; ======  
; ROM BANK#2  
; ======  
; ======  
  
        org      0x6000  
display_message3:  
    ds          "BANK#2 on BANK1"  
    db          0x0D, 0x0A, 0  
    align      8192  
  
; ======  
; ======  
; ROM BANK#3  
; ======  
; ======  
  
        org      0x6000  
display_message4:  
    ds          "BANK#3 on BANK1"  
    db          0x0D, 0x0A, 0  
    align      8192  
  
; ======  
; ======  
; Padding  
; ======  
; ======  
    ds          "Padding"  
    align      131072
```

では、順番に説明したいと思いますが、ROM\_sample001.ASM と同じ部分に関しては、説明を省略させていただきます。

```
main_loop:  
    ; Put message (BANK#0 on BANK0)  
    ld      hl, display_message1  
    call    puts
```

大きく変わっているのは、main\_loop ですが、最初のメッセージ表示に関しては ROM\_sample001.ASM と同じです。

```
; Put message (BANK#1 on BANK1)  
ld      a, 1                      ; BANK#1  
ld      [bank1_sel], a            ; BANK1 changes to BANK#1  
ld      hl, display_message2
```

## MSX Memory Architecture

call	puts
------	------

次に、bank1\_sel ですが、プログラムの上の方で 0x6800 と宣言されています。つまり、ASCII-8K の BANK1 のバンク選択レジスタのアドレスになります。ここに 1 を書き込んでいるので、BANK1 が BANK#1 に切り替わります。このイメージを図 5.2-1. に示します。

## MSX Memory Architecture

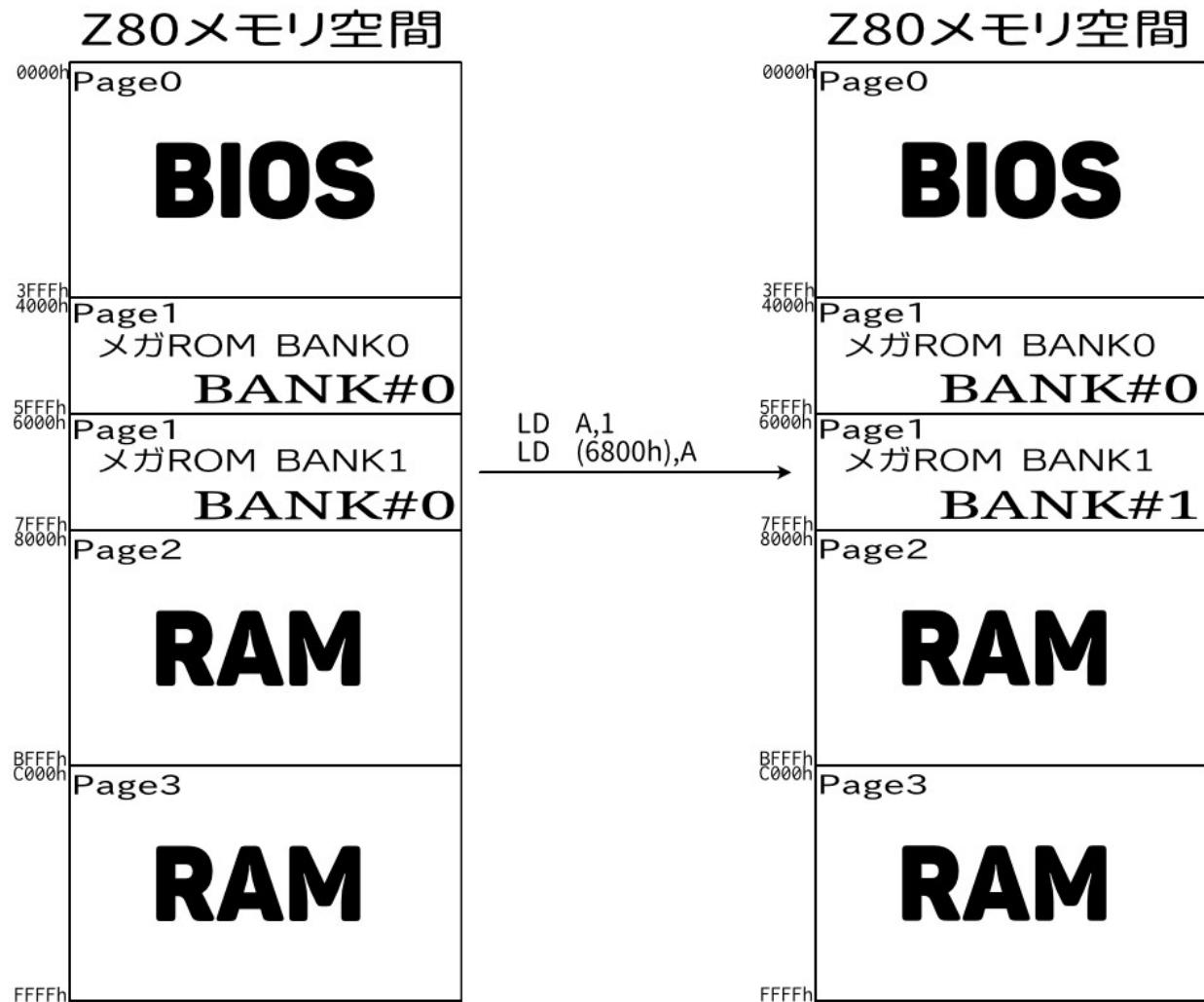


図 5.2-1. BANK1 が BANK#1 に切り替わるイメージ

起動時には、図 5.2-1 の左側のメモリマップのように、BANK0 と BANK1 の領域には、両方とも BANK#0 が出現しています。メガ ROM の 6800h 番地は BANK1 のバンク選択レジスタですが、ここに 1 を書き込むと BANK1 が BANK#1 に切り替わります。

display\_message2 は、BANK1 に出現させた BANK#1 上の文字列のアドレスになります。  
call puts でこれを表示しています。

```
; Put message (BANK#2 on BANK1)
ld      a, 2                      ; BANK#2
ld      [bank1_sel], a            ; BANK1 changes to BANK#2
ld      hl, display_message3
call    puts
; Put message (BANK#3 on BANK1)
ld      a, 3                      ; BANK#3
ld      [bank1_sel], a            ; BANK1 changes to BANK#3
ld      hl, display_message4
call    puts
```

## MSX Memory Architecture

```
jp      main_loop
```

こちらも同様に、BANK1 のバンク選択レジスタに対して、ここに 2 を書き込んで BANK1 を BANK#2 に切り替えて、BANK#2 にある display\_message3 を表示。次に 3 を書き込んで BANK1 を BANK#3 に切り替えて、BANK#3 にある display\_message4 を表示しています。  
jp main\_loop で上に戻って繰り返します。

この動作イメージを、図 5.2-2. に示します。

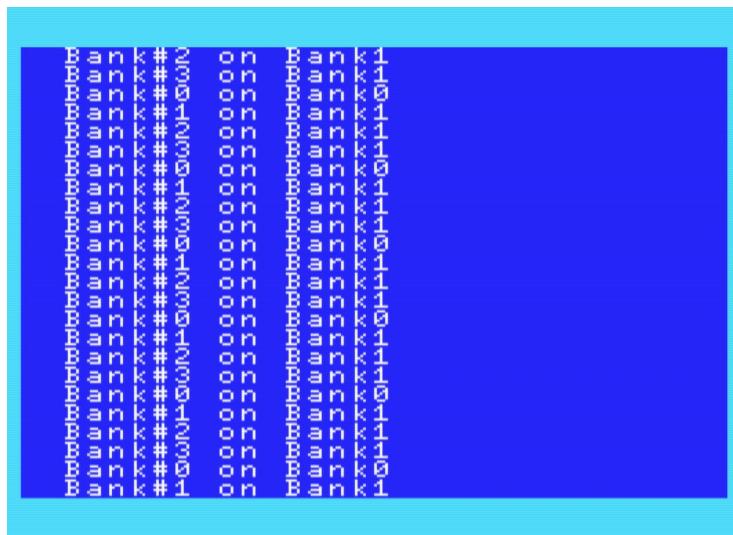


図 5.2-2.

ROM\_sample002.ASM の動作イメージ

### 5.3. 自身のスロットの検出

32KB の ROM や、メガ ROM にプログラムする場合、Page2 も自身の ROM を出現させたくないな  
ります。そのためには、自身の ROM のスロット#を検出する必要があります。ここでは、その方法に  
ついて説明します。

残念ながら、BIOS には現在のスロット#を得るルーチンは用意されていませんので、調べるプロ  
グラムを用意しなければ、自分がどのスロットにいるのか分かりません。

ROM カートリッジ上のプログラムコードから、その ROM カートリッジ自身が装着されている  
SLOT#を求めるプログラムを下記に示します。

ROM\_sample003.ASM

```
; =====
;      ROM_sample003.ASM
; -----
;      Feb./2/2020 HRA!
```

## MSX Memory Architecture

```
; =====
chput      = 0x00A2
himem     = 0xfc4a

        org      0x4000
; =====
; ROM Header
; =====
rom_header_id:
        ds      "AB"
rom_header_init:
        dw      entry_point
rom_header_statement:
        dw      0
rom_header_device:
        dw      0
rom_header_text:
        dw      0
rom_header_reserved:
        space   0x0010 - 0x000A, 0

; =====
; Program entry point
; =====
entry_point:
        ; Initialize Stack Pointer
        ld      sp, [himem]

        call    get_page1_slot

        ; puts slot#
        push   af
        ld      hl, display_message1
        call    puts
        pop    af

        ld      b, a
        and   a, 3
        add   a, '0'
        call   chput

        ld      a, b
        or    a, a
        jp    p, finish

        ld      a, '-'
        call   chput

        ld      a, b
        rrc a
        rrc a
        and   a, 3
        add   a, '0'
        call   chput

finish:
        jp      finish

; =====
; get_page1_slot
; input)
;       HL .... address of target string (ASCII-Z)
```

## MSX Memory Architecture

```
; =====
; scope get_page1_slot
get_page1_slot::
    ; Page1 Base-Slot Detection
    in      a, [0xA8]
    and     a, 0x0C
    rrca
    rrca
    push    af

    ; Page1 Expand-Slot Detection
    ld      b, a
    add    a, 0xC1
    ld      l, a
    ld      h, 0xFC
    ld      a, [hl]
    and    a, 0x80
    jp      z, skip1

    ld      a, b
    rrca
    rrca
    ld      b, a
    in      a, [0xa8]
    ld      c, a
    and    a, 0x3F
    or     a, b
    di
    out   [0xa8], a
    ld      a, [0xFFFF]
    ld      b, a
    ld      a, c
    out   [0xa8], a
    ei
    ld      a, b
    cpl
    and    a, 0x0C
    or     a, 0x80
skip1:
    pop    bc
    or     a, b
    ret
endscope

; =====
; puts
; input)
;       HL .... address of target string (ASCII-Z)
; =====
; scope puts
puts::
    ld      a, [hl]
    inc    hl
    or     a, a
    ret
    call   chput
    jp      puts
endscope

; =====
; DATA
; =====
display_message1:
```

## MSX Memory Architecture

```
ds      "SLOT#"
db      0
align  8192
```

`get_page1_slot` が Page1 のスロット#を得るルーチンです。Page3 で動作させると暴走しますのでご注意下さい。Page0~2 のいずれかで動かすことが前提です。では、`get_page1_slot`について説明します。

```
get_page1_slot::
    ; Page1 Base-Slot Detection
    in     a, [0xA8]
    and   a, 0x0C
    rrca
    rrca
```

`in` 命令で、基本スロット選択レジスタの内容を読み出しています。図 5.3-1.に基本スロット選択レジスタのビットマップを示します。図 5.3-1.で黄色くなっているビットが Page1 のスロット#を示しています。他の Page のスロット#は必要ないので `and a, 0x0C` で 0 クリアしてしまいます。これを右に 2 ビットシフトすることで、A レジスタに SLOT#0~SLOT#3 を示す 0~3 の値が格納された状態になります。

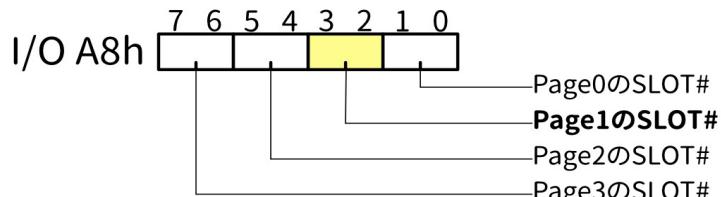


図 5.3-1. 基本ス

ロット選択レジスタ

```
push    af
        ; Page1 Expand-Slot Detection
        ld     b, a
        add   a, 0xC1
        ld     l, a
        ld     h, 0xFC
        ld     a, [hl]
```

## MSX Memory Architecture

次に、求めたスロット#は push af でスタックに保存しておきます。ついでに B レジスタにも保存しています。EXPTBL の中の Page1 のスロット#に対応するアドレスを作り出します。EXPTBL は配列になっていますが、先頭アドレスは FCC1h です。HL=FCC1h+A を実施しています。A レジスタは 0~3 のスロット#なので、add 命令のところでは桁あふれは発生しません。

LD A,[HL] のところで、拡張スロットの有無を示すフラグが A レジスタに格納されます。

and	a, 0x80
jp	z, skip1

and a, 0x80 で MSB を調べます。「スロットが拡張されている場合」は A = 0x80 になるので「Zf = 0」になり、次の jp 命令はスルーします。「スロットが拡張されていない場合」は A = 0x00 になるので「Zf = 1」になり、skip1 ヘジャンプします。

ld	a, b
rrca	
rrca	
ld	b, a

「拡張されていない場合」は、skip1 の方へジャンプしましたので、ここは「拡張されている場合」のみ実行されます。「拡張スロット選択レジスタ」を読むための準備をします。

B レジスタに保存していたスロット#を、右に 2 ビットローテイトして、bit7, bit6 の位置に移動させます。「拡張スロット選択レジスタ」は、そのスロットの Page3 に存在するため、基本スロット選択レジスタの中の Page3 に対応する位置ですね。

in	a, [0xa8]
ld	c, a
and	a, 0x3F
or	a, b
di	
out	[0xa8], a

in 命令で、現在の「基本スロット選択レジスタの値」を読み出します。これを C レジスタにバックアップしてから and a, 0x3F で Page3 のスロット#を 0 クリアして、先ほど B レジスタに保存しておいた Page1 のスロット#を bit7, bit6 へ移動した値を OR しています。

## MSX Memory Architecture

Page3 には割り込み処理中に使うワークエリアもあるため、Page3 を切り替える場合は割り込み禁止にする必要があります。di してから out ですね。

```
ld      a, [0xFFFF]
ld      b, a
ld      a, c
out    [0xa8], a
ei
```

Page3 が、Page1 と同じスロットになっている間に、0xFFFF 番地を読み出しています。これが目的のスロットの「拡張スロット選択レジスタの値」になります。ただし、全ビット反転していることに注意して下さい。読み終えたので、基本スロット選択レジスタの値を戻します。戻したので割り込みを許可します。

```
ld      a, b
cpl
and   a, 0x0C
or    a, 0x80
```

拡張スロット選択レジスタを読み出した値を反転して使える値に変換。ENASLT で使うスロット# の上位 6bit を作り出します。

```
skip1:
pop   bc
or    a, b
ret
```

スロットが拡張されていない場合は、A=0 で skip1 へ飛んできます。つまり、スタックに積んでおいた基本スロット#がそのまま A に入ります。

スロットが拡張されている場合は、A レジスタには「ENASLT で使うスロット#」の上位 6bit が格納されています。pop bc で B レジスタに入ってくる値は 0~3 の基本スロット#で下位 2bit に値が詰まっています。これらを or することで、「ENASLT で使うスロット#」が得られます。

## 5.4. Page2/Page3 のスロット検出

やり方は、5.3. 自身のスロットの検出で Page1 に出現している ROM を検出したのと同じ方法で、Page2 と Page3 に出現している RAM のスロットを検出します（※RAM8KB または RAM16KB の機種では Page2 は RAM になっていません）。検出プログラムを下記に示します。

## MSX Memory Architecture

### ROM\_sample004.ASM

```
; =====
; ROM_sample004.ASM
; -----
; Feb./3/2020 HRA!
; =====

chput      = 0x00A2
himem     = 0xfc4a

        org      0x4000
; =====
; ROM Header
; =====

rom_header_id:
        ds      "AB"
rom_header_init:
        dw      entry_point
rom_header_statement:
        dw      0
rom_header_device:
        dw      0
rom_header_text:
        dw      0
rom_header_reserved:
        space   0x0010 - 0x000A, 0

; =====
; Program entry point
; =====

entry_point:
        ; Initialize Stack Pointer
        ld      sp, [himem]

        ; page1
        ld      hl, display_message_page1
        call    puts
        call    get_page1_slot
        call    put_slot_no

        ; page2
        ld      hl, display_message_page2
        call    puts
        call    get_page2_slot
        call    put_slot_no

        ; page3
        ld      hl, display_message_page3
        call    puts
        call    get_page3_slot
        call    put_slot_no

finish:
        jp      finish

; =====
;     get_page1_slot
;     input)
;     none
;     output)
;     A .... slot number of page1
; =====
```

## MSX Memory Architecture

```
        scope get_page1_slot
get_page1_slot::
    ; Page1 Base-Slot Detection
    in      a, [0xA8]
    and    a, 0x0C
    rrc a
    rrc a
    push   af

    ; Page1 Expand-Slot Detection
    ld      b, a
    add    a, 0xC1
    ld      l, a
    ld      h, 0xFC
    ld      a, [hl]
    and    a, 0x80
    jp      z, skip1

    ld      a, b
    rrc a
    rrc a
    ld      b, a
    in      a, [0xa8]
    ld      c, a
    and    a, 0x3F
    or     a, b
    di
    out   [0xa8], a
    ld      a, [0xFFFF]
    ld      b, a
    ld      a, c
    out   [0xa8], a
    ei
    ld      a, b
    cpl
    and    a, 0x0C
    or     a, 0x80
skip1:
    pop   bc
    or     a, b
    ret
endscope

; =====
; get_page2_slot
; input)
;     none
; output)
;     A .... slot number of page2
; =====
        scope get_page2_slot
get_page2_slot::
    ; Page2 Base-Slot Detection
    in      a, [0xA8]
    and    a, 0x30
    rrc a
    rrc a
    rrc a
    rrc a
    push   af

    ; Page2 Expand-Slot Detection
    ld      b, a
```

## MSX Memory Architecture

```
add      a, 0xC1
ld       l, a
ld       h, 0xFC
ld       a, [hl]
and      a, 0x80
jp       z, skip1

ld       a, b
rrca
rrca
ld       b, a
in       a, [0xa8]
ld       c, a
and      a, 0x3F
or       a, b
di
out      [0xa8], a
ld       a, [0xFFFF]
ld       b, a
ld       a, c
out      [0xa8], a
ei
ld       a, b
cpl
and      a, 0x30
rrca
rrca
or       a, 0x80
skip1:
pop      bc
or       a, b
ret
endscope

; =====
; get_page3_slot
;   input)
;     none
;   output)
;     A .... slot number of page3
; =====
get_page3_slot::          ; Page3 Base-Slot Detection
  in      a, [0xA8]
  and      a, 0xC0
  rlca
  rlca
  push      af

  ; Page3 Expand-Slot Detection
  ld      b, a
  add      a, 0xC1
  ld      l, a
  ld      h, 0xFC
  ld      a, [hl]
  and      a, 0x80
  jp       z, skip1

  ld      a, [0xFFFF]
  cpl
  and      a, 0xC0
  rrca
```

## MSX Memory Architecture

```
        rrc a
        rrc a
        rrc a
        or      a, 0x80
skip1:
        pop      bc
        or      a, b
        ret
        ends scope

; =====
; puts
; input)
; HL .... address of target string (ASCII-Z)
; =====

puts::           scope puts
        ld      a, [hl]
        inc     hl
        or      a, a
        ret     z
        call    chput
        jp      puts
        ends scope

; =====
; put_slot_no
; input)
; A .... SLOT# (ENASLT format)
; =====

put_slot_no::   scope put_slot_no
        ld      b, a
        and    a, 3
        add    a, '0'
        call   chput

        ld      a, b
        or      a, a
        jp      p, skip1

        ld      a, '-'
        call   chput

        ld      a, b
        rrc a
        rrc a
        and    a, 3
        add    a, '0'
        call   chput

skip1:
        ld      hl, display_crlf
        call   puts
        ret
        ends scope

; =====
; DATA
; =====

display_message_page1:
        ds      "PAGE1 SLOT#"
        db      0
display_message_page2:
```

## MSX Memory Architecture

```
        ds      "PAGE2 SLOT#"
        db      0
display_message_page3:
        ds      "PAGE3 SLOT#"
        db      0
display_crlf:
        db      0x0D, 0x0A, 0
align    8192
```

処理の内容は、ほぼ 5.3. 自身のスロットの検出と同じであるため、説明は省略します。

Page2 と Page3 の RAM は、同じスロットとは限りませんので個別に求める必要があります。

特に、Page2 を「RAM のスロット」と「メガ ROM のスロット」とを必要に応じて切り替えながら使用することは多々あると思います。

## 5.5. Page0/Page3 のスロット切り替え

スロット切替には、BIOS の ENASLT (0024h) という便利なルーチンが用意されていますが、これを使って Page0 を切り替えることは出来ません。ENASLT 自体が Page0 に存在するため、ENASLT を Page0 切り替えの設定で呼び出すと自分自身を見失って暴走します。

Page3 は、ENASLT で切り替えることが出来ますが、スタックポインタが Page3 にある状態で ENASLT を呼ぶと暴走しますのでご注意下さい。ENASLT 内ではスタックを利用してますので、必ず Page3 以外の Page ヘスタックポインタを移動してから呼び出す必要があります。また、BIOS ワークエリアがある領域なので、EI するタイミングにも注意が必要です。割り込みが入ると呼び出されるフックも BIOS ワークエリアに存在するので、適切なケアをしていない状態で切り替えたまま EI すると、割り込みが入った段階で暴走します。

メガ ROM でない ROM カートリッジで最大容量は 64KB になりますが、その場合 Page0 や Page3 を ROM カートリッジのスロットに切り替えたくなります。

Page0 のスロットを切り替えて RAM のスロットを探索し、次に Page3 のスロットを切り替えて RAM のスロットを探索するサンプルプログラムを下記に示します。

### ROM\_sample005.ASM

```
; =====
; ROM_sample005.ASM [RAM 32KB required]
; -----
; Feb./3/2020 HRA!
; =====
enaslt      = 0x0024
```

## MSX Memory Architecture

```
chget      = 0x009F
chput      = 0x00A2
himem      = 0xFC4A
exptbl     = 0xFCC1

        org      0x4000
; =====
; ROM Header
; =====
rom_header_id:
        ds      "AB"
rom_header_init:
        dw      entry_point
rom_header_statement:
        dw      0
rom_header_device:
        dw      0
rom_header_text:
        dw      0
rom_header_reserved:
        space   0x0010 - 0x000A, 0

; =====
; Program entry point
; =====
entry_point:
        ; Initialize Stack Pointer
        ld      hl, 0xC000
        ld      sp, hl

        call    get_page3_slot
        ld      [p3_ram_slot], a

        ld      hl, exptbl
        ld      de, exptbl_copy
        ld      bc, 4
        ldir

main_loop:
        call    check_page0
        ld      hl, message_page0
        call    display_slot_info
        call    press_enter_key

        call    check_page3
        ld      hl, message_page3
        call    display_slot_info
        call    press_enter_key

        jp      main_loop

press_enter_key:
        ld      hl, message_press_enter_key
        call    puts
        call    chget
        ret

; =====
; clear_slot_info
; =====
; scope clear_slot_info
clear_slot_info::
        ld      hl, slot_info
```

## MSX Memory Architecture

```
ld      de, slot_info + 1
ld      bc, 16 - 1
xor    a, a
ld      [hl], a
ldir
ret
endscope

; =====
;     check_page0
; =====
scope check_page0
check_page0::
    call    clear_slot_info          ; B = 0
    ld      hl, exptbl
    ld      de, slot_info

slot_loop:
    ld      a, [hl]
    and    a, 0x80
    or     a, b

exp_slot_loop:
    ld      c, a
    push   bc
    push   hl
    push   de
    call   local_enaslt0
    pop    de

    ; RAM check
    ld      hl, 0x0000

check_ram_loop:
    ld      a, 1
    bit    6, h
    jp     nz, check_ram_exit
    ld      a, [hl]
    cpl
    ld      [hl], a
    cp     a, [hl]
    cpl
    ld      [hl], a
    inc    hl
    jp     z, check_ram_loop
    ld      a, 2

check_ram_exit:
    pop   hl
    pop   bc

    ld      [de], a
    inc    de
    ld      a, c
    add    a, 0x04
    jp     p, not_expanded
    bit    4, a
    jp     z, exp_slot_loop
    jp     next_slot

not_expanded:
    inc    de
    inc    de
    inc    de

next_slot:
    inc    hl
    inc    b
    bit    2, b
```

## MSX Memory Architecture

```
        jp      z, slot_loop

        ld      a, [exptbl]
        call    local_enaslt0
        ei
        ret
        endscope

; =====
;     check_page3
; =====
        scope check_page3
check_page3:::
        call    clear_slot_info          ; B = 0
        ld     hl, exptbl_copy
        ld     de, slot_info
slot_loop:
        ld     a, [hl]
        and   a, 0x80
        or    a, b
exp_slot_loop:
        ld     c, a
        push  bc
        push  hl
        push  de
        ld     h, 0xC0
        call  enaslt
        pop   de

        ; RAM check
        ld     hl, 0xC000
check_ram_loop:
        ld     a, h
        cp     a, 0xFF
        ld     a, 1
        jp     z, check_ram_exit
        ld     a, [hl]
        cpl
        ld     [hl], a
        cp     a, [hl]
        cpl
        ld     [hl], a
        inc   hl
        jp     z, check_ram_loop
        ld     a, 2
check_ram_exit:
        pop   hl
        pop   bc

        ld     [de], a
        inc   de
        ld     a, c
        add   a, 0x04
        jp     p, not_expanded
        bit   4, a
        jp     z, exp_slot_loop
        jp     next_slot

not_expanded:
        inc   de
        inc   de
        inc   de
next_slot:
        inc   hl
```

## MSX Memory Architecture

```
inc      b
bit      2, b
jp      z, slot_loop

ld      a, [p3_ram_slot]
ld      h, 0xC0
call    enaslt
ei
ret
endscope

; =====
; get_page3_slot
;   input)
;     none
;   output)
;     A .... slot number of page3
; =====

get_page3_slot::          ; Page3 Base-Slot Detection
in      a, [0xA8]
and    a, 0xC0
rlca
rlca
push   af

; Page3 Expand-Slot Detection
ld      b, a
add   a, 0xC1
ld      l, a
ld      h, 0xFC
ld      a, [hl]
and   a, 0x80
jp      z, skip1

ld      a, [0xFFFF]
cpl
and   a, 0xC0
rrca
rrca
rrca
rrca
or      a, 0x80
skip1:
pop   bc
or      a, b
ret
endscope

; =====
; local_enaslt0
;   input)
;     A .... slot number
;   output)
;     none
;   break)
;     af, bc, de
; =====

local_enaslt0::           ; scope local_enaslt0
ld      b, a                  ; B = Target SLOT#, E000ssSS
and   a, 0x83                  ; E00000SS : SS = Target SLOT#
```

## MSX Memory Architecture

```
jp      p, not_expanded
xor    a, 0x80
ld     c, a           ; C = Target SLOT#
rrca
rrca
or     a, c           ; SS0000SS : SS = Target SLOT#
ld     c, a
in     a, [0xA8]
ld     e, a
and    a, 0b00111100; 00BBAA00 : AA = Page1 SLOT#, BB = Page2 SLOT#
or     a, c           ; SSBAAASS : SS = Target SLOT#
di
out    [0xA8], a      ; Change slot
and    a, 0b00111111
ld     d, a
ld     a, e
and    a, 0b11000000
or     a, d
ld     d, a
ld     a, [0xFFFF]
cpl
and    a, 0xFC
ld     c, a
ld     a, b
rrca
rrca
and    a, 0b00000011; 000000ss
or     a, c
ld     [0xFFFF], a
ld     a, d
out    [0xA8], a
ret

not_expanded:
ld     c, a           ; C = Target SLOT#
in     a, [0xA8]
and    a, 0b11111100; CCBBA00 : AA = Page1 SLOT#, BB = Page2 SLOT#,
CC = Page3 SLOT#
or     a, c           ; CCBBAASS : SS = Target SLOT#
di
out    [0xA8], a
ret
endscope

; =====
; puts
; input)
;       HL .... address of target string (ASCII-Z)
; =====
;           scope puts
puts::
ld     a, [hl]
inc   hl
or     a, a
ret   z
call  chput
jp    puts
endscope

; =====
; put_slot_no
; input)
;       A .... SLOT# (ENASLT format)
; =====
```

## MSX Memory Architecture

```
        scope put_slot_no
put_slot_no:::
    ld      b, a
    and    a, 3
    add    a, '0'
    call   chput

    ld      a, b
    or     a, a
    jp     p, not_expanded

    ld      a, '-'
    call   chput

    ld      a, b
    rrc a
    rrc a
    and    a, 3
    add    a, '0'
    call   chput
    ret

not_expanded:
    ld      hl, message_padding
    call   puts
    ret
    endscope

; =====
; display_slot_info
; input)
;       HL .... Header message address
; =====
        scope display_slot_info
display_slot_info:::
    call   puts
    ld    hl, exptbl           ; SLOT#0-0 の EXPTBL
    ld    de, slot_info        ; SLOT#0-0 の slot_info
    ld    b, 0                  ; SLOT#0 から表示開始

slot_loop:
    ld      a, [hl]             ; EXPTBL を読む
    inc    hl                  ; 次の SLOT に備える
    push   hl
    and    a, 0x80             ; 拡張スロットフラグ以外は消去
    or     a, b                ; E0000000
    ld      b, a                ; B レジスタにバックアップ

exp_slot_loop:
    ld      a, [de]             ; slot_info を読む
    inc    de                  ; 次の SLOT に備える
    or     a, a                ; 0 の場合、存在しないスロットなので無視
    jp     z, skip1
    ld      c, a                ; C レジスタに slot_info をバックアップ

    ld      a, b                ; A レジスタに SLOT# を復元
    push   de
    push   bc
    push   af
    ld      hl, message_slot   ; "SLOT#" を表示
    call   puts
    pop    af
    call   put_slot_no         ; SLOT# を表示
```

## MSX Memory Architecture

```
pop      bc
push    bc
ld      hl, message_ram
dec      c           ; slot_info が 1 なら "RAM", 2 なら "Non-
RAM" を表示
jp      z, skip2
ld      hl, message_non_ram
skip2:
call    puts
pop      bc
pop      de
skip1:
ld      a, b           ; A レジスタに SLOT# を復元
add      a, 0x04         ; 次の拡張スロットへ
ld      b, a
bit      4, a
jp      z, exp_slot_loop
pop      hl
and      a, 3
inc      a
ld      b, a
bit      2, a
jp      z, slot_loop
ret
endscope

; =====
; DATA
; =====
message_page0:
ds      "PAGE0:"
db      0x0D, 0x0A, 0
message_page3:
ds      "PAGE3:"
db      0x0D, 0x0A, 0
message_slot:
ds      "  SLOT#"
db      0
message_padding:
ds      "
db      0
message_ram:
ds      ": RAM"
db      0x0D, 0x0A, 0
message_non_ram:
ds      ": Non-RAM"
db      0x0D, 0x0A, 0
message_press_enter_key:
ds      "Press enter key!!"
db      0x0D, 0x0A, 0

; =====
; WORK AREA (Page2)
; =====
p3_ram_slot = 0x8000          ; 1byte : Page3 RAM slot#
exptbl_copy = p3_ram_slot + 1  ; 4bytes : Copy of EXPTBL
slot_info   = exptbl_copy + 4   ; 16bytes: SLOT information 0:N/A, 1:RAM, 2:Non-RAM
align      8192
```

これまでに出てきたサンプルと共に部分の説明は省略します。

## MSX Memory Architecture

Page0 のスロットを切り替えるルーチンは、local\_enaslt0 になります。Page0 を「A レジスタで指定したスロット#」に切り替えるルーチンです。

```
scope local_enaslt0
local_enaslt0::
    ld      b, a          ; B = Target SLOT#, E000ssSS
    and    a, 0x83         ; E00000SS : SS = Target SLOT#
    jp      p, not_expanded
    xor    a, 0x80
```

まず、B レジスタにスロット#を保存しておきます。

and a, 0x83 により、拡張スロット番号を指定している bit3, bit2 を 0 クリアします。と同時に拡張の有無を示す bit7 を Sf (負数なら 1, 正数かゼロなら 0) に反映させます。

二の補数表現の場合、負数なら bit7 = 1、正数かゼロなら bit7 = 0 です。そのため、Sf を見ることで拡張スロットの有無を見ることが出来ます。拡張スロットでない場合は、not\_expanded へジャンプします。

xor a, 0x80 は、bit7 を 0 クリアしています。and a, 0x7F でも OK です。

## 6. ハードウェア

## 7. その他の補足事項

本章では、6章までに挙げられなかった補足事項について記載します。

### 7.1. バージョンアップアダプター利用時の EXPTBL

MSX1 を MSX2 相当にバージョンアップする「NEOS MA-20 MSX バージョンアップアダプター」（以後、バージョンアップアダプタ）という製品が存在しました。MSX2 に搭載されている MAIN-ROM/SUB-ROM/V9938/64KB RAM のカートリッジを MSX1 に取り付けることで、MSX2 相当にバージョンアップします。ここで気になるのが、EXPTBL(FCC1h)です。

FCC1h は、「SLOT#0 の拡張の有無」を示すワークエリアですが、一方で「MAIN-ROM のスロット」を兼ねたワークエリアもあります。普通の本体は、MAIN-ROM は SLOT#0 に存在するため、FCC1h の値そのものを ENASLT 等のスロット指定にそのまま使えるわけですが、バージョンアップアダプタの場合 SLOT#0 以外のスロットに MSX2 版 MAIN-ROM が搭載されることになります。MSX2 版 MAIN-ROM が拡張スロットの 0 番、例えば SLOT#1-0 に装着されており、一方で本体側の SLOT#0 は拡張されていなかった場合を考えてみて下さい。FCC1h の MSB は、SLOT#0 の拡張の有無を示すので 0 になるわけですが、下位 4bit には SLOT#1-0 を示す 0001 が格納されると、SLOT#1 なのか SLOT#1-0 なのか区別がつきません。FCC2h の MSB が SLOT#1 の拡張の有無を示してるので、どちらなのか判別するようにプログラムを組んでいれば整合しますが、（おそらくバージョンアップアダプタが出た時点でも）世の中に出回っているソフトは FCC1h を ENASLT のスロット指定にそのまま使える MAIN-ROM のスロット#と見なしているものが多い、ということです。

そこで、バージョンアップアダプタは、FCC1h の MSB を「SLOT#0 が拡張されているか否かにかかわらず、1 にする」という拳動になっています。

例えば、SLOT#1 に MSX2 版 MAIN-ROM カートリッジを装着すると、その MAIN-ROM は、FCC1h の値を 2 進数で 10000001 という値で上書きします。実は、このような値で概ね問題なく動くわけです。

SLOT#0 及び SLOT#1 が拡張されておらず、MSX2 版 MAIN-ROM が SLOT#1 に装着されている場合を考えてみます。FCC1h には 81h が格納されているため、「MAIN-ROM へ切り替えたいプログラム」は、FCC1h の値を A レジスタに入れて ENASLT を呼びます。すると、ENASLT は、I/O A8h を使って SLOT#1 に切り替えた後に、その FFFFh に書き込みます。拡張スロットであれば FFFFh は拡張スロットレジスタが存在しますが、実際は SLOT#1 は拡張されていないので、この FFFFh への書き込みは消滅します。なぜならば、SLOT#1 の MSX2 版 MAIN-ROM カートリッジの FFFFh へ書き込みに行つたことになるためです。MAIN-ROM カートリッジは Page0,Page1

## MSX Memory Architecture

に ROM が存在するのみで、Page2, Page3 は何も繋がっていません。そのため FFFFh への書き込みアクセスは、何も書き込まれずに消滅します。

バージョンアップアダプタの挙動に関しては、実物を所有されているばるぶ様にご協力いただき、確認しました。

## 7.2. 拡張スロット選択レジスタは無条件に書いてはダメ

何らかの事情で ENASLT が使えない場合に、自分でスロット切り替えルーチンを作ることになります。そのような場合に、スロットの拡張の有無に関係なく、拡張スロット選択レジスタに書き込んでしまえば、少し処理が楽になるのでは無いか？と思うかもしれません。実際に、7.1. バージョンアップアダプター利用時の EXPTBL で説明したように、バージョンアップアダプタは、SLOT#0 の拡張の有無に関係なく EXPTBL(FCC1h) の MSB を 1 に変更してしまいます。しかし、これは SLOT#0 だったからセーフなのであって、その他のスロットでは NG です。拡張スロットが接続されていない基本スロットの FFFFh に書き込むと、素直にそのアドレスに書き込みに行きます。RAM だった場合には RAM に書き込まれてしまいます。「FFFh 番地の RAM なんて使わないから大丈夫でしょ？」と思うかもしれません、メモリーマッパー対応 RAM のようにセグメントやバンクの出現アドレスを選べるタイプの RAM が接続されている場合に問題が起こります。

例えば、メモリーマッパー対応 RAM が複数存在し、少なくともその一方は基本スロットに接続されている場合に問題が起ります。具体的な例を挙げておきます。私が配布している MGSP v2.1.x DOS 版(以降 MGSP) という音楽プレイヤーがありますが、これは漢字フォントをメモリーマッパー対応 RAM 上に置きます。漢字フォントはセカンダリーマッパーに置くことにも対応しています。MGSP は MGSDRV.COM もメモリーマッパー対応 RAM 上に置きます。こちらはプライマリマッパーのみ対応です。

MGSP を無改造の FS-A1ST で利用する場合には、増設 RAM カートリッジ(メモリーマッパー対応)が必要になり、本体側の 256KB がプライマリマッパー、増設 RAM カートリッジがセカンダリーマッパーになります。FS-A1ST 起動時のメモリーマッパーの使用状況を図 7.2-1. に示します。増設 RAM カートリッジは 64KB・SLOT#1 に装着の想定で書いています。

## MSX Memory Architecture

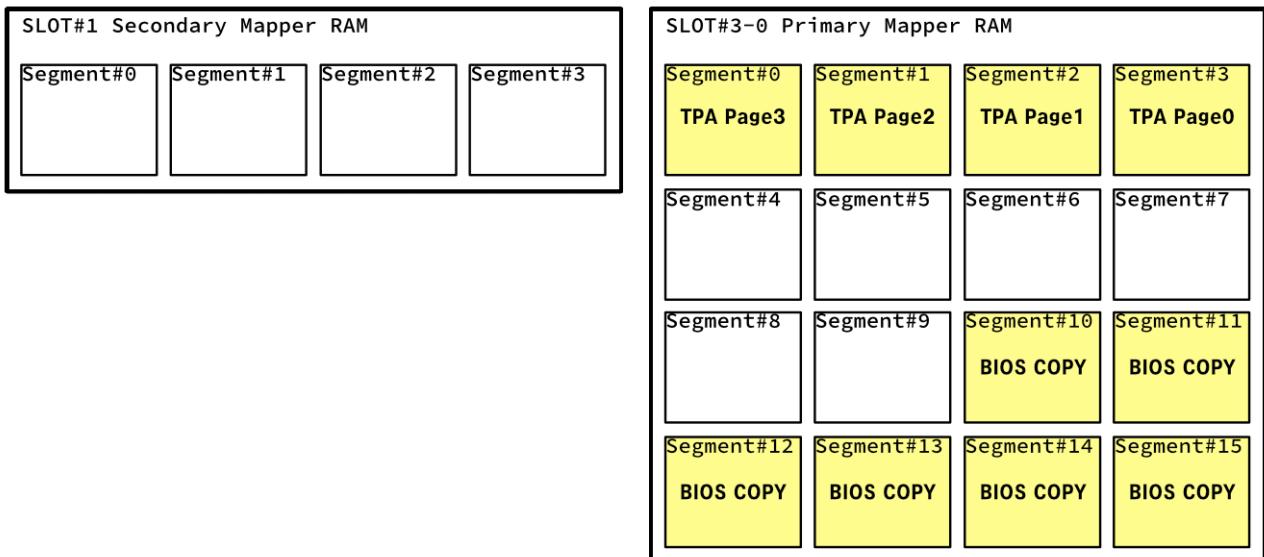


図 7.2-1. FS-A1ST 起動時のメモリーマッパー使用状況

図 7.2-1. で黄色く塗られている部分が、使われている Segment# になります。FS-A1ST の場合、半分以上がすでに使用済みになっています。

ここで、何らかの常駐型のプログラムが 2 セグメント使用したとします。例えば、MGSDRV.COM を常駐させた場合を想定します。そうすると、図 7.2-2. のように使用状況が変化します。常駐型のプログラムは、システムセグメントを使用します。メモリーマッパーのシステムセグメントは、セグメント# の大きい方から割り当てられます。

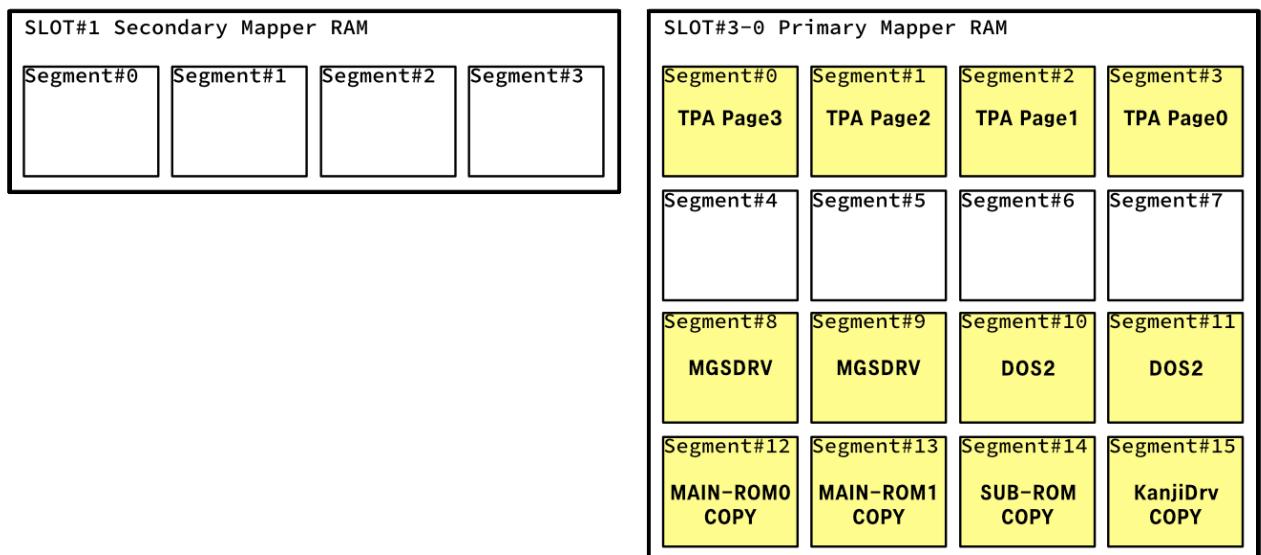


図 7.2-2. MGSDRV 常駐後のメモリーマッパー使用状況

この状態から、MGSP を起動させると、MGSP は常駐している MGSDRV は利用できませんので、MGSDRV 用のメモリを新たに使用してしまいます。これに 2 セグメントを使います。MGSP はユー

## MSX Memory Architecture

ザーセグメントで使用するため、セグメント#の小さい方から割り当てられます。さらに、漢字フォント用に4セグメント使いますが、プライマリーマッパーには2セグメントしか空きがありません。漢字フォントは、セカンダリーマッパーにも対応していますが、複数のメモリーマッパー対応RAMをまたがる確保には対応していないため、すべてセカンダリーマッパーの方へ追いやられます。結果として、図7.2-3.のような使用状況になります。

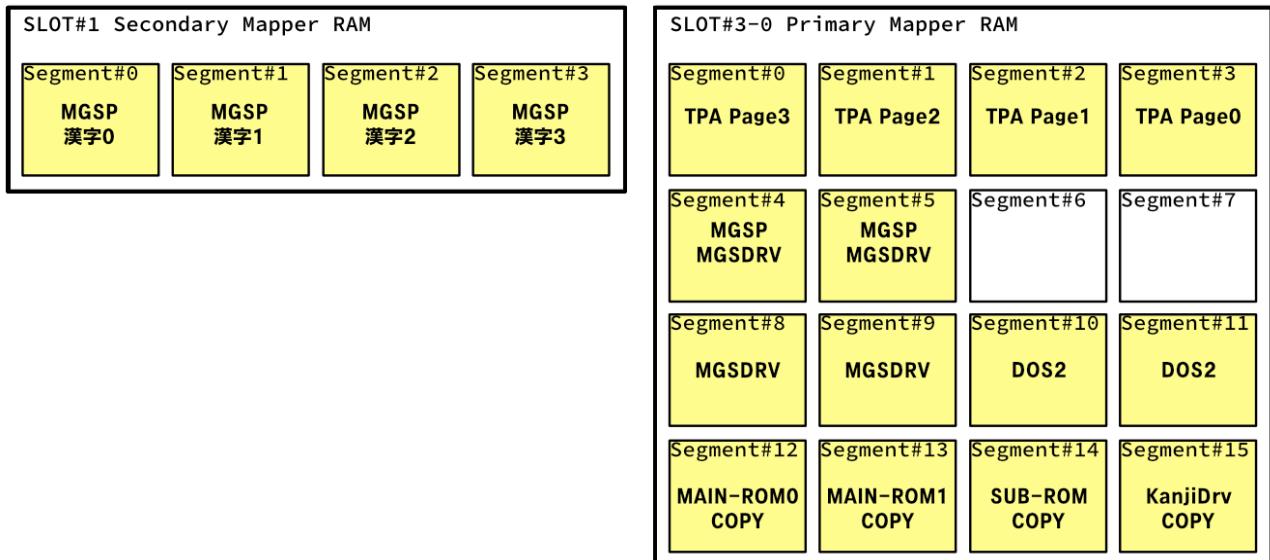


図7.2-3. MGSP動作中のメモリ確保状況

ここで改めて、スロットの方に目を向けてみます。FS-A1STのSLOT#1にメモリーマッパー対応RAM 64KBカートリッジを装着した状態のスロット構成は図7.2-4.のようになっています。

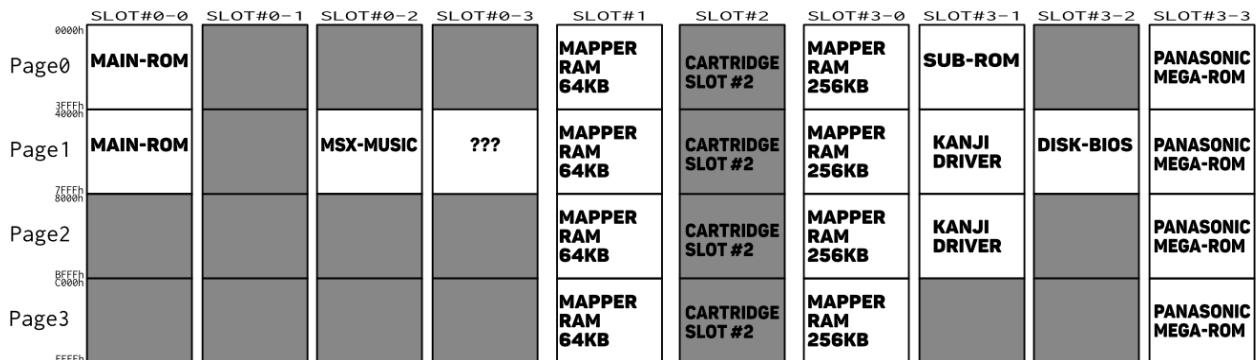


図7.2-4. FS-A1ST + RAM64KB

MSX-DOS2が起動した状態では、Z80メモリ空間は図7.2-5.のようになっています。

## MSX Memory Architecture

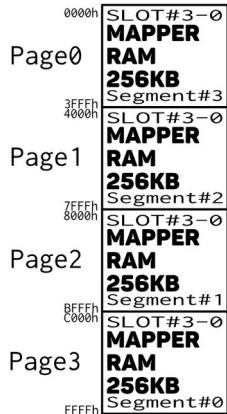


図 7.2-5. DOS2 起動時の Z80 メモリ空間

メモリーマッパーの各 Segment に出現する Segment# は全メモリーマッパーで共通ですので、この時点で SLOT#1 の RAM64KB の Segment も Page0 から順に Segment#3, Segment#2, Segment#1, Segment#0 となっています。SLOT#1 は拡張スロットになっておらず、ダイレクトに RAM64KB が装着されている状態です。

MGSP は、漢字フォントを SLOT#1 の RAM64KB に格納しているので、Page2 を SLOT#1 に切り替えて漢字フォントにアクセスします。この「SLOT#1 に切り替える」という処理を、ENASLT を使わずに MGSP 内で独自に実装していたと仮定して、その独自のスロット切り替えルーチン内で「スロットが拡張されているか否かに関わらず、拡張スロット選択レジスタが存在するであろう FFFFh に書き込む」という実装になっていたらどうなるでしょうか？

FFFFh に現れる「拡張スロット選択レジスタ」は、あくまで「拡張スロット」というデバイスが有するレジスタになります。「拡張スロット」というデバイスは、自身の FFFFh へのアクセスは、自身の拡張スロット選択レジスタへ接続します。一方で、自身の 0000h-FFF Eh へのアクセスは、自身にぶら下がる 4 つのスロットの中から、拡張スロット選択レジスタが指し示すスロットにアクセスを伝搬させます。拡張スロット選択レジスタは、物理的に拡張スロット側に実装されるものです。

「拡張スロット」というデバイスが存在しない基本スロットそのものの場合は、そこに装着されているカートリッジなどへ FFFFh へのアクセスも伝搬させます。そうしないと、そこに「拡張スロット」が装着された場合、「拡張スロット」は FFFFh へのアクセスを受け取ることが出来なくなるからです。

ここで、上記 FS-A1ST + RAM64KB で MGSP を動かした例に話を戻しますと、拡張されていない SLOT#1 の FFFFh に対して値を書き込むと、そこに出現しているメモリーマッパー対応 RAM64KB カートリッジの Segment#0 の先頭から +3FFFh の位置にその値が書き込まれてし

## MSX Memory Architecture

まいります。そこには漢字フォントが存在していますので、漢字フォントが 1byte 破壊されてしまうことになります。

「拡張スロットの有無にかかわらず無条件に FFFFh に書き込むと問題が起こるケース」として、メモリーマッパー対応 RAM との組み合わせケースを例に挙げました。状況によっては他の組み合わせでも問題が出る場合があるかもしれないで、やはり拡張スロットが存在しないスロットに対して、拡張スロット選択を想定した FFFFh への書き込みは避けるべきでしょう。

## 付録.

### 付録 1. サンプルプログラムから利用している各種ソース

msxbios.asm

chkram	:=	0x0000
synchr	:=	0x0008
rdslt	:=	0x000C
chrgtr	:=	0x0010
wrslt	:=	0x0014
outdo	:=	0x0018
calslt	:=	0x001c
dcompr	:=	0x0020
enaslt	:=	0x0024
getypr	:=	0x0028
callf	:=	0x0030
keyint	:=	0x0038
initio	:=	0x003b
inifnk	:=	0x003e
disscr	:=	0x0041
enascr	:=	0x0044
wrtvdp	:=	0x0047
rdvrm	:=	0x004a
wrtvrm	:=	0x004d
setrd	:=	0x0050
setwrt	:=	0x0053
filvrm	:=	0x0056
ldirmv	:=	0x0059
ldirvm	:=	0x005c
chgmod	:=	0x005f
chgclr	:=	0x0062
nmi	:=	0x0066
clrspr	:=	0x0069
initxt	:=	0x006c
init32	:=	0x006f
inigrp	:=	0x0072
inimlt	:=	0x0075
settxt	:=	0x0078
sett32	:=	0x007b
setgrp	:=	0x007e
setmlt	:=	0x0081
calpat	:=	0x0084
calatr	:=	0x0087
gpssiz	:=	0x008a
grpprt	:=	0x008d
gicini	:=	0x0090
wrtpsg	:=	0x0093
rdpsg	:=	0x0096
strtms	:=	0x0099
chsns	:=	0x009c
chget	:=	0x009f
chput	:=	0x00a2
lptout	:=	0x00a5
lptstt	:=	0x00a8
cnvchr	:=	0x00ab
pinlin	:=	0x00ae
inlin	:=	0x00b1

## MSX Memory Architecture

```
qinlin      :=      0x00b4
breakx      :=      0x00b7
beep        :=      0x00c0
cls         :=      0x00c3
posit       :=      0x00c6
fnksb       :=      0x00c9
erafnk      :=      0x00cc
dspfnk      :=      0x00cf
totext      :=      0x00d2
gtstck      :=      0x00d5
gttrig      :=      0x00d8
gtpad       :=      0x00db ; note: MSXturboR ではライトペン(A=8~11)は常に0が返る
gtpdl       :=      0x00de ; note: MSXturboR では常に0が返る
tapion      :=      0x00e1 ; note: MSXturboR では常にCy=1(エラー)が返る
tapin       :=      0x00e4 ; note: MSXturboR では常にCy=1(エラー)が返る
tapiof      :=      0x00e7 ; note: MSXturboR では常にCy=1(エラー)が返る
tapoon      :=      0x00ea ; note: MSXturboR では常にCy=1(エラー)が返る
tapout      :=      0x00ed ; note: MSXturboR では常にCy=1(エラー)が返る
tapoof      :=      0x00f0 ; note: MSXturboR では常にCy=1(エラー)が返る
stmotr      :=      0x00f3 ; note: MSXturboR では何もせずに返る
chgcap      :=      0x0132
chgsnd      :=      0x0135
rslreg      :=      0x0138
wslreg      :=      0x013b
rdvdp       :=      0x013e
snsmat      :=      0x0141
isflio       :=      0x014a
outdlp      :=      0x014d
kilbuf      :=      0x0156
calbas      :=      0x0159

; require MSX2
subrom      :=      0x015c
extrom      :=      0x015f
eol          :=      0x0168
bigfil       :=      0x016b
nsetrd      :=      0x016e
nstwrt      :=      0x0171
nrdvrm      :=      0x0174
nwrvrm      :=      0x0177
rdres        :=      0x017A
wrres        :=      0x017D

; require MSXturboR
chgcpu      :=      0x0180
getcpu      :=      0x0183
pcmply      :=      0x0186
pcmrec      :=      0x0189

; SUB-ROM (require MSX2)
sub_gtpprt  :=      0x0089
sub_nvbxln  :=      0x00c9
sub_nvbxfl  :=      0x00cd
sub_chgmod  :=      0x00d1
sub_initxt  :=      0x00d5
sub_init32  :=      0x00d9
sub_inigrp  :=      0x00dd
sub_inimlt  :=      0x00e1
sub_settxt  :=      0x00e5
sub_sett32  :=      0x00e9
sub_setgrp  :=      0x00ed
sub_setmlt  :=      0x00f1
```

## MSX Memory Architecture

sub_clrspr	:=	0x00f5	
sub_calpat	:=	0x00f9	
sub_calatr	:=	0x00fd	
sub_gsp siz	:=	0x0101	
sub_getpat	:=	0x0105	
sub_wrtvrm	:=	0x0109	
sub_rdvrm	:=	0x010d	
sub_chgclr	:=	0x0111	
sub_clssub	:=	0x0115	
sub_dspfnk	:=	0x011d	
sub_wrtvdp	:=	0x012d	
sub_vdpsta	:=	0x0131	
sub_setpag	:=	0x013d	
sub_iniplt	:=	0x0141	
sub_rstplt	:=	0x0145	
sub_getplt	:=	0x0149	
sub_setplt	:=	0x014d	
sub_beep	:=	0x017d	
sub_prompt	:=	0x0181	
sub_newpad	:=	0x01ad	; note: MSXturboR ではライトペン(A=8~11)は常に 0 が返る
sub_chgmdp	:=	0x01b5	
sub_knjprt	:=	0x01bd	
sub_redclk	:=	0x01f5	
sub_wrtclk	:=	0x01f9	
h_prompt	:=	0xF24F	; hook: Disk change prompt (Require DISK DRIVE)
diskve	:=	0xF323	; 2bytes (Require DISK DRIVE)
breakv	:=	0xF325	; 2bytes (Require DISK DRIVE)
ramad0	:=	0xF341	; 1byte, Page0 RAM Slot (Require DISK DRIVE)
ramad1	:=	0xF342	; 1byte, Page1 RAM Slot (Require DISK DRIVE)
ramad2	:=	0xF343	; 1byte, Page2 RAM Slot (Require DISK DRIVE)
ramad3	:=	0xF344	; 1byte, Page3 RAM Slot (Require DISK DRIVE)
masters	:=	0xF348	; 1byte, MasterCartridgeSlot (Require DISK DRIVE)
rdprim	:=	0xF380	; 5bytes, 基本スロットからの読み込み
wrprim	:=	0xF385	; 7bytes, 基本スロットへの書き込み
clprim	:=	0xF38C	; 14bytes, 基本スロットコール
cliksw	:=	0xF3DB	; 1byte, キークリックスイッチ (0=OFF, others=ON)
csry	:=	0xF3DC	; 1byte, カーソルのY座標
csrx	:=	0xF3DD	; 1byte, カーソルのX座標
cnsdfg	:=	0xF3DE	; 1byte, ファンクションキー表示スイッチ (0=ON, others=OFF)
rg0sav	:=	0xF3DF	; 1byte, VDP R#0 に書き込んだ値
rg1sav	:=	0xF3E0	; 1byte, VDP R#1 に書き込んだ値
rg2sav	:=	0xF3E1	; 1byte, VDP R#2 に書き込んだ値
rg3sav	:=	0xF3E2	; 1byte, VDP R#3 に書き込んだ値
rg4sav	:=	0xF3E3	; 1byte, VDP R#4 に書き込んだ値
rg5sav	:=	0xF3E4	; 1byte, VDP R#5 に書き込んだ値
rg6sav	:=	0xF3E5	; 1byte, VDP R#6 に書き込んだ値
rg7sav	:=	0xF3E6	; 1byte, VDP R#7 に書き込んだ値
statfl	:=	0xF3E7	; 1byte, VDP S#0 から読み込んだ値
fnkstr	:=	0xF87F	; 16byte * 10, ファンクションキーに対応する文字列
exbrsa	:=	0xFAF8	; 1byte, SUB-ROM のスロット#
hokvld	:=	0xFB20	

## MSX Memory Architecture

```
oldkey      :=      0xFBDA      ; 11bytes, キーマトリクスの状態(旧)
newkey      :=      0xFBE5      ; 11bytes, キーマトリクスの状態(新)
keybuf      :=      0xFBFO      ; 40bytes, キーバッファ
linwrk      :=      0xFC18      ; 40bytes, スクリーンハンドラが使う一時保管場所
patwrk      :=      0xFC40      ; 8bytes, パターンコンバータが使う一時保管場所
bottom      :=      0xFC48      ; 2bytes, 実装したRAMの先頭番地。MSX2では通常 8000h
himem       :=      0xFC4A      ; 2bytes, 利用可能なメモリの上位番地。CLEAR文の<メモリ上限>
により設定される
trptbl      :=      0xFC4C      ; 78bytes, 割り込み処理で使うトラップテーブル。一つのテーブル
は 3bytes で 1バイト目が ON/OFF/STOP の状態。残りが分岐先テキストアドレス

scrmod      :=      0xFCAF      ; 1byte, 現在のスクリーンモード番号
oldscr      :=      0xFCB0      ; 1byte, スクリーンモード保存エリア

exptbl      :=      0xFCC1      ; 4bytes, 各スロットの拡張の有無
sltbl       :=      0xFCC5      ; 4bytes, 各スロットの拡張スロット選択レジスタの現在の選択状況
sltatr      :=      0xFCC9      ; 64bytes, 各スロット用の属性
sltwrk      :=      0xFD09      ; 128bytes, 各スロット用の特定のワークエリアを確保
procnm      :=      0xFD89      ; 16bytes, 拡張ステートメント, 拡張デバイスの名前が入る,
ASCIIIZ
device      :=      0xFD99      ; 1byte, カートリッジ用の装置識別
h_timi       :=      0xFD9F      ; 5bytes, 垂直同期割り込み発生時のフック
extbio      :=      0xFFCA
rg8sav      :=      0FFE7
rg9sav      :=      0FFE8
```

## msxdos1.asm

```
BDOS          := 0x0005
DMA           := 0x0080
TPA_BOTTOM    := 0x0006

; =====
;   Terminate program
;   input)
;     C = D1F_TERM0
;   output)
;   --
; =====
D1F_TERM0     := 0x00

; =====
;   Console input
;   input)
;     C = D1F_CONIN
;   output)
;     A = 入力された文字
;     L = Aと同じ
;   comment)
;   標準出力にエコーされる
; =====
D1F_CONIN     := 0x01

; =====
```

## MSX Memory Architecture

```
;     Console output
;     input)
;         C = D1F_CONOUT
;         E = 出力する文字
;     output)
;
; =====
D1F_CONOUT          := 0x02

D1F_AUXIN           := 0x03
D1F_AUXOUT          := 0x04
D1F_LSTOUT          := 0x05
D1F_DIRIO           := 0x06
D1F_DIRIN           := 0x07
D1F_INNOE           := 0x08
D1F_STROUT          := 0x09
D1F_BUFIN           := 0x0A
D1F_CONST            := 0x0B
D1F_CPMVER          := 0x0C
D1F_DSKRST          := 0x0D
D1F_SELDSK          := 0x0E
D1F_FOPEN            := 0x0F
D1F_FCLOSE           := 0x10
D1F_SFIRST           := 0x11
D1F_SNEXT            := 0x12
D1F_FDEL             := 0x13
D1F_RDSEQ            := 0x14
D1F_WRSEQ            := 0x15
D1F_FMAKE            := 0x16
D1F_FREN             := 0x17
D1F_LOGIN            := 0x18
D1F_CURDRV           := 0x19
D1F_SETDTA           := 0x1A
D1F_ALLOC             := 0x1B

D1F_RDRND            := 0x21
D1F_WRRND            := 0x22
D1F_FSIZE            := 0x23
D1F_SETRND           := 0x24

D1F_WRBLK            := 0x26
D1F_RDBLK            := 0x27
D1F_WRZER            := 0x28

D1F_GDATE            := 0x2A
D1F_SDATE            := 0x2B
D1F_GTIME            := 0x2C
D1F_STIME            := 0x2D
D1F_VERIFY           := 0x2E
D1F_RDABS            := 0x2F
D1F_WRABS            := 0x30
D1F_DPARM            := 0x31

; =====
;     error code
; =====
D1E_EOF              := 0xC7
```

## MSX Memory Architecture

```
; -----  
; FIB (File Information Block)  
FIB_SIGNATURE      := 0          ; 1byte, Always 0xFF.  
FIB_FILENAME       := 1          ; 13bytes, File name (ASCIIIZ)  
FIB_ATTRIBUTE      := 14         ; 1byte, See below for details.  
FIB_UPDATE_TIME   := 15         ; 2bytes, Last update time.  
FIB_UPDATE_DATE   := 17         ; 2bytes, Last update date.  
FIB_CLUSTER        := 19         ; 2bytes, First cluster  
FIB_SIZE           := 21         ; 4bytes, File size  
FIB_DRIVE          := 25         ; 1byte, Logical drive  
FIB_INTERNAL       := 26         ; 38bytes, Internal information (Must not be changed).  
  
; -----  
; FIB_ATTRIBUTE  
FIB_ATTR_READ_ONLY := 0b0000_0001  
FIB_ATTR_HIDDEN    := 0b0000_0010  
FIB_ATTR_SYSTEM    := 0b0000_0100  
FIB_ATTR_VOLUME    := 0b0000_1000  
FIB_ATTR_DIRECTORY := 0b0001_0000  
FIB_ATTR_ARCHIVE   := 0b0010_0000  
FIB_ATTR_RESERVED  := 0b0100_0000  
FIB_ATTR_DEVICE    := 0b1000_0000  
  
; -----  
; Error code  
D2E_NCOMP          := 0xFF        ; Incompatible disk  
D2E_WRERR          := 0xFE        ; Write error  
D2E_DISK            := 0xFD        ; Disk error  
D2E_NRDY           := 0xFC        ; Not ready  
D2E_VERIFY          := 0xFB        ; Verify error  
D2E_DATA            := 0xFA        ; Data error  
D2E_RNF             := 0xF9        ; Sector not found  
D2E_WPROT           := 0xF8        ; Write protected disk  
D2E_UFORM           := 0xF7        ; Unformatted disk  
D2E_NDOS            := 0xF6        ; Not a DOS disk  
D2E_WDISK           := 0xF5        ; Wrong disk  
D2E_WFILE           := 0xF4        ; Wrong disk for file  
D2E_SEEK             := 0xF3        ; Seek error  
D2E_IFAT             := 0xF2        ; Bad file allocation table  
D2E_NOUPB           := 0xF1        ; -  
D2E_IFORM           := 0xF0        ; Cannot format this drive  
D2E_SUCCESS          := 0x00        ; -  
  
; -----  
; find first  
; input)  
;     C = D2F_FFIRST  
;     DE = Directory path name address (ASCIIIZ) or FIB address  
;     B = Search target attribute  
;         bit0: read only  
;         bit1: hidden  
;         bit2: system file  
;         bit3: volume label  
;         bit4: directory  
;         bit5: archive  
;         bit6: RESERVED  
;         bit7: device  
;     IX = Result area address (64bytes)  
; output)  
;     A = Error code  
;     [IX] = FIB of the file  
D2F_FFIRST          := 0x40
```

## MSX Memory Architecture

```
; -----  
; find next  
; input)  
;     C = D2F_FNEXT  
;           IX = Result area address of D2F_FFIRST  
; output)  
;     A = Error code  
;           [IX] = FIB of the file  
D2F_FNEXT      := 0x41  
  
;  
; -----  
; find new entry  
; input)  
;     C = D2F_FNEW  
;           DE = Directory path name address (ASCIIIZ) or FIB address  
;           B = Search target attribute  
;                 bit0: read only  
;                 bit1: hidden  
;                 bit2: system file  
;                 bit3: volume label  
;                 bit4: directory  
;                 bit5: archive  
;                 bit6: RESERVED  
;                 bit7: new creation flag  
;           IX = FIB with template  
; output)  
;     A = Error code  
;           [IX] = FIB of the file  
D2F_FNEW       := 0x42  
  
;  
; -----  
; open  
; input)  
;     C = D2F_OPEN  
;           DE = File path name address (ASCIIIZ) or FIB address  
;           A = Open mode  
;                 bit0: Disable write access  
;                 bit1: Disable read access  
;                 bit2: Succession  
;                 others: always 0  
; output)  
;     A = Error code  
;           B = New file handle  
D2F_OPEN        := 0x43  
  
D2F_CREATE      := 0x44  
D2F_CLOSE        := 0x45  
D2F_ENSURE       := 0x46  
D2F_DUP          := 0x47  
D2F_READ         := 0x48  
D2F_WRITE        := 0x49  
D2F_SEEK          := 0x4A  
D2F_IOCTL         := 0x4B  
D2F_HTEST         := 0x4C  
D2F_DELETE        := 0x4D  
D2F_RENAME        := 0x4E  
D2F_MOVE          := 0x4F  
D2F_ATTR          := 0x50  
D2F_FTIME         := 0x51  
D2F_HDELETE        := 0x52  
D2F_HRENAME        := 0x53
```

## MSX Memory Architecture

D2F_HMOVE	:= 0x54
D2F_HATTR	:= 0x55
D2F_HFTIME	:= 0x56
D2F_HGETDTA	:= 0x57
D2F_GETVFY	:= 0x58
D2F_GETCD	:= 0x59
D2F_CHDIR	:= 0x5A
D2F_PARSE	:= 0x5B
D2F_PFILE	:= 0x5C
D2F_CHKCHR	:= 0x5D
D2F_WPATH	:= 0x5E
D2F_FLUSH	:= 0x5F
D2F_FORK	:= 0x60
D2F_JOIN	:= 0x61
D2F_TERM	:= 0x62
D2F_DEFAB	:= 0x63
D2F_DEFER	:= 0x64
D2F_ERROR	:= 0x65
D2F_EXPLAIN	:= 0x66
D2F_FORMAT	:= 0x67
D2F_RAMD	:= 0x68
D2F_BUFFER	:= 0x69
D2F_ASSIGN	:= 0x6A
D2F_GENV	:= 0x6B
D2F_SENV	:= 0x6C
D2F_FENV	:= 0x6D
D2F_DSKCHK	:= 0x6E
D2F_DOSVER	:= 0x6F
D2F_REDIR	:= 0x70

## stdio.asm

```
; =====
; puts
;   input)
;     hl .... Target string (ASCIIIZ)
;   output)
;   --
;   break)
;   all
;   comment)
;   標準出力にASCIIIZ文字列を表示する
; =====
; scope      puts
puts::    ld       a, [de]
           inc      de
           or      a, a
           ret      z
           push    de
           ld       c, D1F_DIRIO
           ld       e, a
           call    bdos
           pop      de
           jr      puts
endscope
```

## mmapper.asm

## MSX Memory Architecture

```
; =====
; MapperSupportRoutine's table offset (This area is reference only)
; =====
mmap_slot      := 0          ; 1byte, マッパーRAMのスロット#
mmap_total_seg := 1          ; 1byte, 総セグメント数
mmap_free_seg   := 2          ; 1byte, 未使用セグメント数
mmap_sys_seg    := 3          ; 1byte, システムに割り当てられたセグメント数
mmap_user_seg   := 4          ; 1byte, ユーザーに割り当てられたセグメント数
mmap_reserved   := 5          ; 3byte, 予約領域

; =====
; mmap_init
;   input)
;   --
;   output)
;       Zf ..... 1: MemoryMapper サポートルーチンが存在しない
;       [mmap_table_ptr] ... マッパーテーブルのアドレス
;   break)
;   all
; =====
        scope mmap_init
mmap_init:::
    ld      c, D2F_DOSVER      ; DOS バージョンをチェック
    call   bdos
    or     a, a                ; A != 0 は DOS ではない
    jp     nz, mmap_error_exit
    ld     a, b
    cp     a, 2                ; B < 2 の場合 MSX-DOS1 である
    jp     c, mmap_error_exit

    ld     a, [hokvld]
    and   a, 1
    ret   z                   ; 拡張 BIOS が存在しない場合はエラー (Zf=1)

    ; get MapperSupportRoutine's table
    xor   a, a
    ld    de, 0x0401          ; D=MemoryMapperSupportRoutine ID, E=01h
    call  extbio
    or    a, a
    ret   z                   ; マッパーサポートルーチンが存在しない場合はエラー
(Zf=1)
    ld    [mmap_table_ptr], hl

    ; get jump table
    xor   a, a
    ld    de, 0x0402          ; D=MemoryMapperSupportRoutine ID, E=02h
    call  extbio
    ld    de, mapper_jump_table
    ld    bc, 16 * 3
    ldir

    ; get current segment on Page1
    call  mapper_get_p1
    ld   [mapper_segment_p1], a
    call  mapper_get_p2
    ld   [mapper_segment_p2], a

    xor   a, a                ; A=0
    inc   a                    ; A=1, Zf=0
    ret   z                   ; 正常終了 (Zf=0)

mmap_error_exit:
```

## MSX Memory Architecture

```
        xor      a, a          ; Zf=1 にする
        ret
        endscope

; =====
;   スロット構成を TPA に戻す
; =====

        scope    mmap_change_to_tpa
mmap_change_to_tpa:::
        ; change slot of Page1
        ld      h, 0x40
        ld      a, [ramad1]
        call    enaslt

        ; change slot of page2
        ld      h, 0x80
        ld      a, [ramad2]
        call    enaslt

        ; change mapper segment of page1
        ld      a, [mapper_segment_p1]
        call    mapper_put_p1

        ; change mapper segment of page2
        ld      a, [mapper_segment_p2]
        call    mapper_put_p2
        ei
        ret
        endscope

; =====
;   WORKAREA
; =====

mmap_table_ptr:::
        dw      0          ; マッパーテーブルのアドレスが格納される
mmaper_segment_p1:::
        db      0          ; 起動時の page1 のセグメント#
mmaper_segment_p2:::
        db      0          ; 起動時の page2 のセグメント#

mmap_jump_table:::
mmaper_all_seg:::      ; +00h
        db      0xc9, 0xc9, 0xc9
mmaper_fre_seg:::      ; +03h
        db      0xc9, 0xc9, 0xc9
mmaper_rd_seg:::       ; +06h
        db      0xc9, 0xc9, 0xc9
mmaper_wr_seg:::       ; +09h
        db      0xc9, 0xc9, 0xc9
mmaper_cal_seg:::      ; +0Ch
        db      0xc9, 0xc9, 0xc9
mmaper_calls:::        ; +0Fh
        db      0xc9, 0xc9, 0xc9
mmaper_put_ph:::        ; +12h
        db      0xc9, 0xc9, 0xc9
mmaper_get_ph:::        ; +15h
        db      0xc9, 0xc9, 0xc9
mmaper_put_p0:::        ; +18h
        db      0xc9, 0xc9, 0xc9
mmaper_get_p0:::        ; +1Bh
        db      0xc9, 0xc9, 0xc9
mmaper_put_p1:::        ; +1Eh
```

## MSX Memory Architecture

```
        db      0xc9, 0xc9, 0xc9
mapper_get_p1::    ; +21h
        db      0xc9, 0xc9, 0xc9
mapper_put_p2::    ; +24h
        db      0xc9, 0xc9, 0xc9
mapper_get_p2::    ; +27h
        db      0xc9, 0xc9, 0xc9
mapper_put_p3::    ; +2Ah
        db      0xc9, 0xc9, 0xc9
mapper_get_p3::    ; +2Dh
        db      0xc9, 0xc9, 0xc9
```

## MSX Memory Architecture

### 注意

本書に記載の内容を用いて、何らかの損害が発生したとしても当方は責任を負いません。各自の責任の下でご利用下さい。

本書は、下記の資料を参考に作成しています。元の資料を公開して下さった方々に感謝します。

#### 【参考文献】

テクハン wiki 化計画

<http://ngs.no.coocan.jp/doc/wiki.cgi/TechHan>

MSX Datapack wiki 化計画

<http://ngs.no.coocan.jp/doc/wiki.cgi/datapack?page=FrontPage>

Gigamix Online メガ ROM データベース

<https://gigamix.hatenablog.com/entry/rom/>

メガコン解析資料

[http://www.big.or.jp/~saibara/msx/ese/mctr\\_anl-j.html](http://www.big.or.jp/~saibara/msx/ese/mctr_anl-j.html)

msx.org MegaROM Mappers

[https://www.msx.org/wiki/MegaROM\\_Mappers](https://www.msx.org/wiki/MegaROM_Mappers)

Konami Sound Cartridge - SCC+

<http://bifi.msxnet.org/msxnet/tech/soundcartridge.html>

ひろゆきのホームページ SCC 実験室

<http://d4.princess.ne.jp/msx/scc/>

ASCAT home page テクガイ本文全編

<http://www.ascat.jp/index.html>

Special thanks to:

大勢の方から、いろいろなご指摘やご協力をいただき、完成度を上げることが出来ました。この場を借りて御礼申し上げます。特に、れふてい様からは、私自身が見直すだけでは気が付かない部分も多々ご指摘いただき、非常に感謝しております。

2021年2月19日 HRA!