MSX Memory Architecture

Index

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 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.

MSX Memory Architecture

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	PageN
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#N
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.	BANKN
MegaROM	A portion of the ROM that appears in the above bank	BANK#N

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.

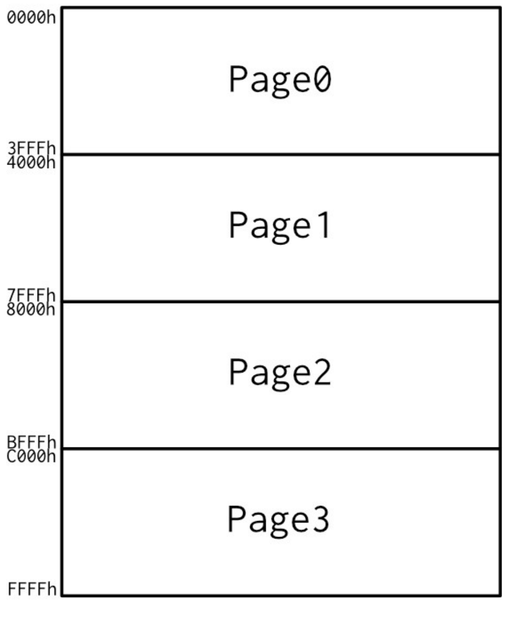


Figure 2-1: Page area classification visible in the Z80 memory space

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.

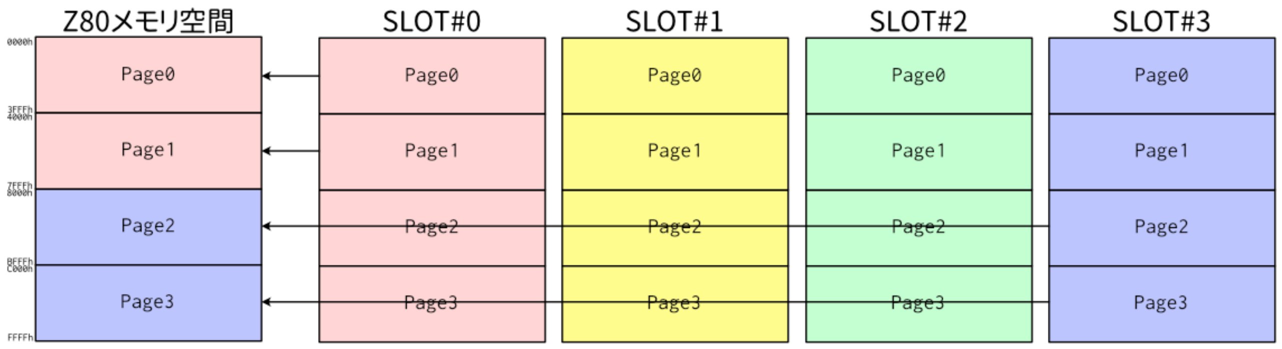


Figure 2-2: Basic slot selection

Figure 2-2: The arrow in the slot selection register corresponds to the basic slot selection register. This basic slot selection register is connected as a register that can be read and written to A8h in I/O space. The 8-bit value that can be written to I/O A8h corresponds to, as shown in Figure 2-3, slot # where bit1 and bit0 appear in Page0, slot # where bit 3 and bit2 appear in Page1, slot # where bit 5 and bit4 appear in Page2, and slot # where bit 7 and bit6 appear in Page3.

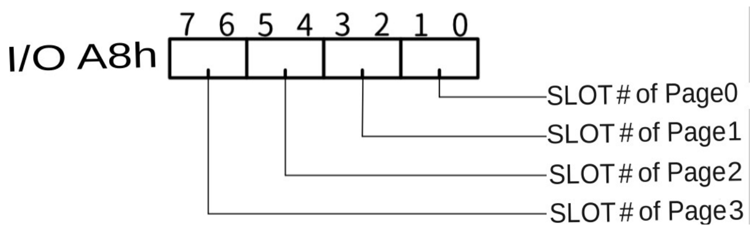
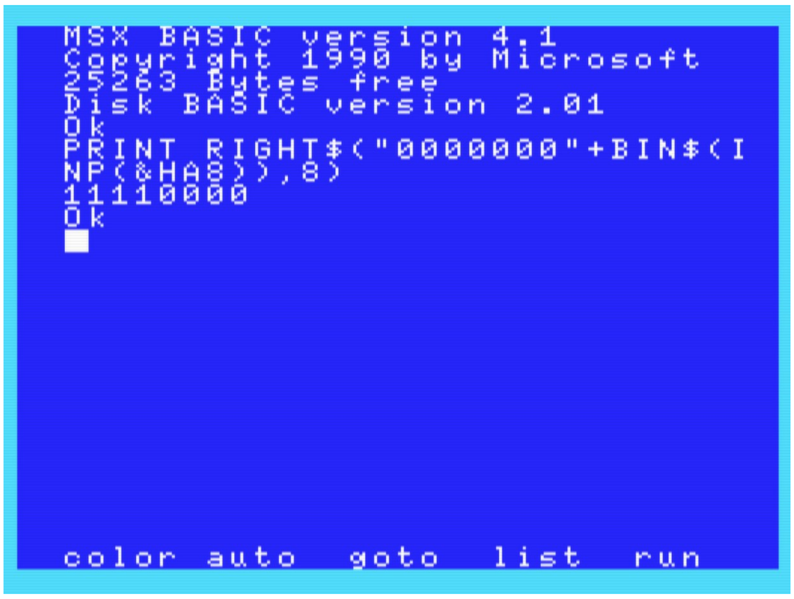


Figure 2-3: Basic Slot Select Register (I/O A8h)

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.



```
MSX BASIC version 4.1
Copyright 1990 by Microsoft
25263 Bytes free
Disk BASIC version 2.01
Ok
PRINT RIGHT$("0000000"+BIN$(I
NP(&HA8)),8)
11110000
Ok
```

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.

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.

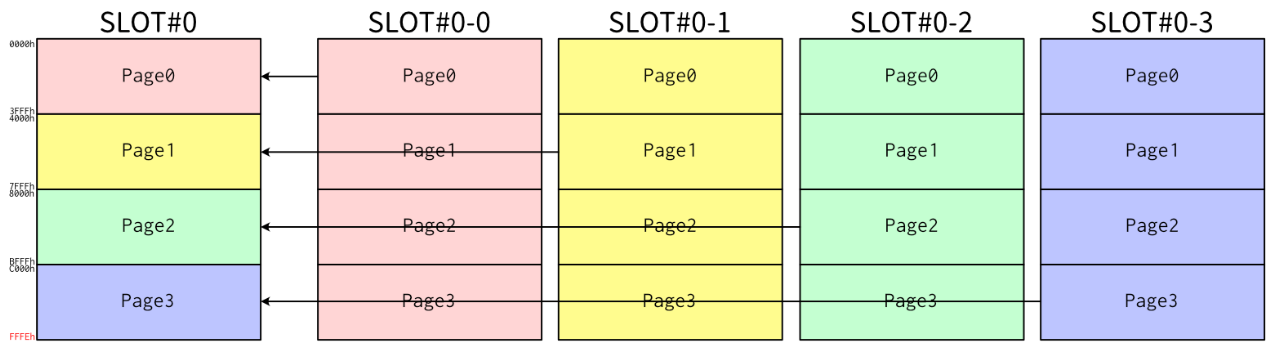


Figure 2-5: Image of expansion slot

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.

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

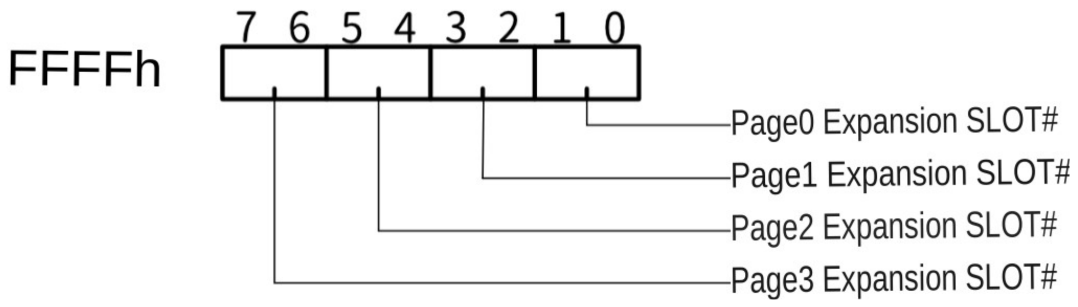


Figure 2-6:Expansion slot selection register

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.

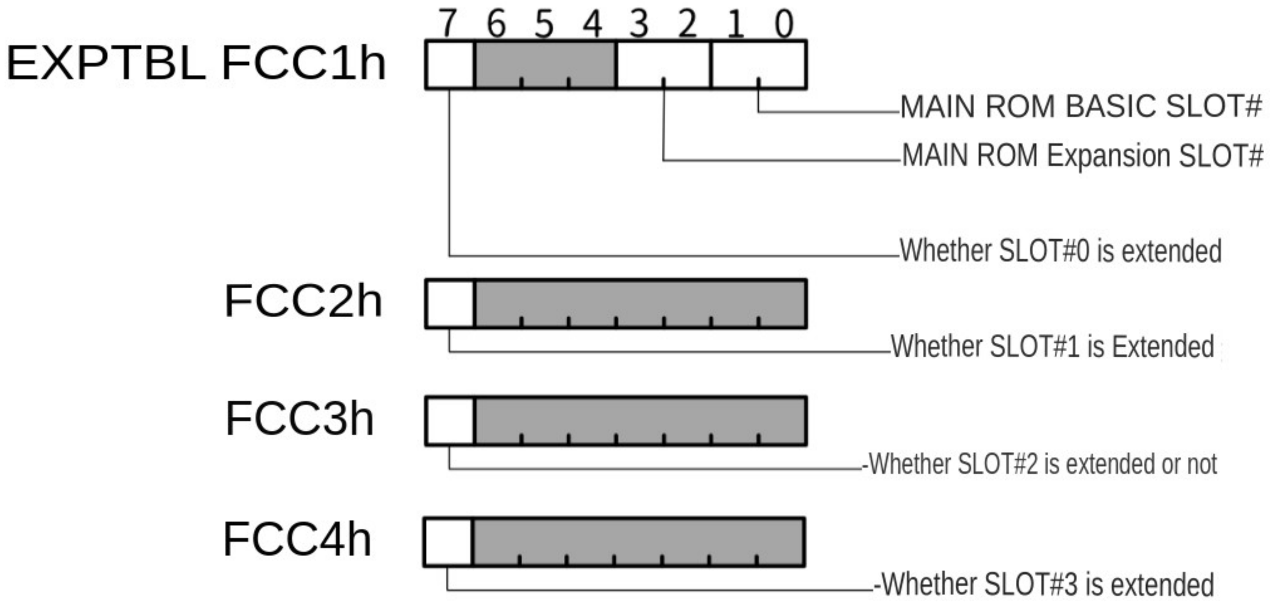

7 6 5 4 3 2 1 0
EXPTBL FCC1h
MAIN ROM BASIC SLOT#
MAIN ROM Expansion SLOT#
Whether SLOT#0 is extended
FCC2h
Whether SLOT#1 is Extended
FCC3h
-Whether SLOT#2 is extended or not
FCC4h
-Whether SLOT#3 is extended

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

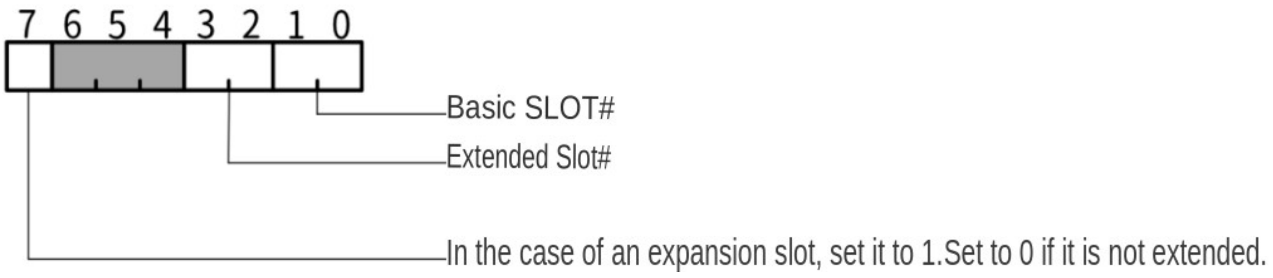
Slot-related BIOS routines are described separately here.

RDSLT (000Ch)

Reads the specified address 1 byte of the specified slot.

Input:

A register: Value indicating "specified slot".


7 6 5 4 3 2 1 0
Basic SLOT#
Extended Slot#
In the case of an expansion slot, set it to 1.Set to 0 if it is not extended.

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
```

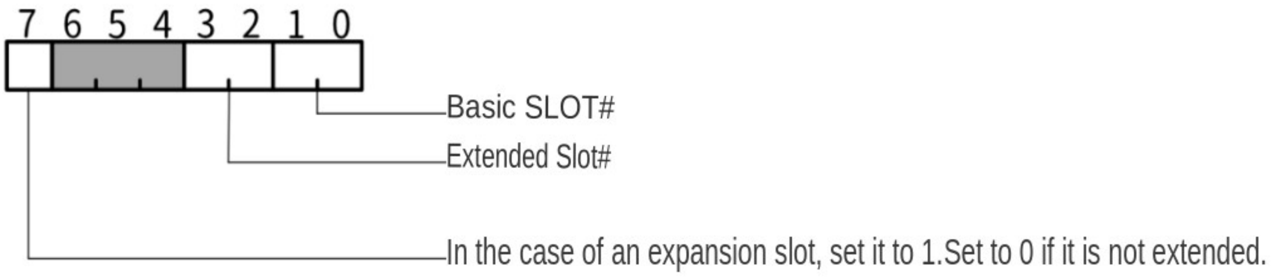
This will read the 1-byte value at address 1234h of SLOT#3-2 and return it to the A register.

WRSLT (0014h)

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

Input:

A register: Value indicating "specified slot".


7 6 5 4 3 2 1 0
Basic SLOT#
Extended Slot#
In the case of an expansion slot, set it to 1.Set to 0 if it is not extended.

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 RDSLT, 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
```

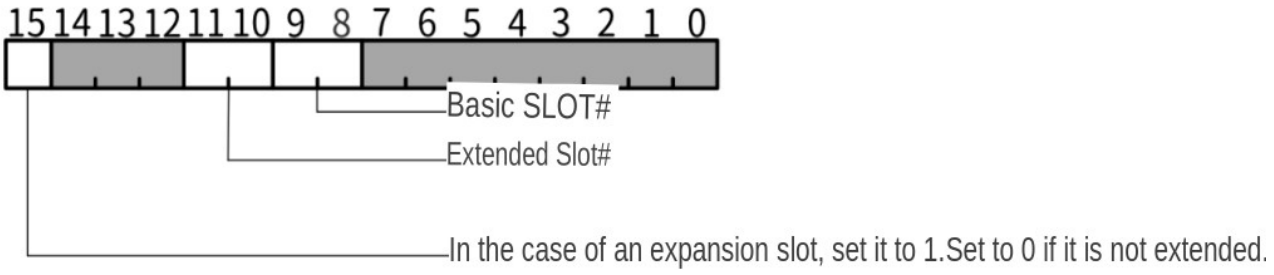
With this, ABh is written to 1234h address of SLOT#3-2.

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.


15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Basic SLOT#
Extended Slot#
In the case of an expansion slot, set it to 1.Set to 0 if it is not extended.

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.

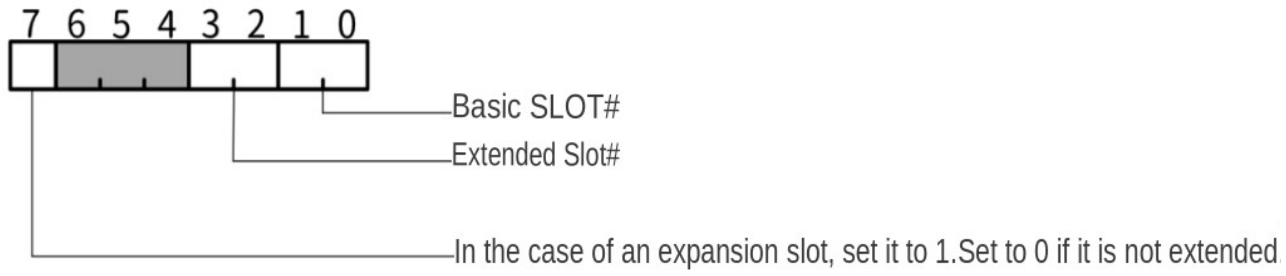
```
    LD      A, 1                  ; switch to SCREEN1
    LD      IY, [0xFCC1 - 1]      ; MAIN-ROM Slot to upper 8bit of IY
    LD      IX, 0x005F
    CALL    0x001C
```

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".


7 6 5 4 3 2 1 0
Basic SLOT#
Extended Slot#
In the case of an expansion slot, set it to 1.Set to 0 if it is not extended.

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.

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 SLOT#1-0
      LD      DE, 0x8000
      LD      HL, 0x4000
      LD      BC, 0x2000
      LDIR                    ; 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 SLOT#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 :

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 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.

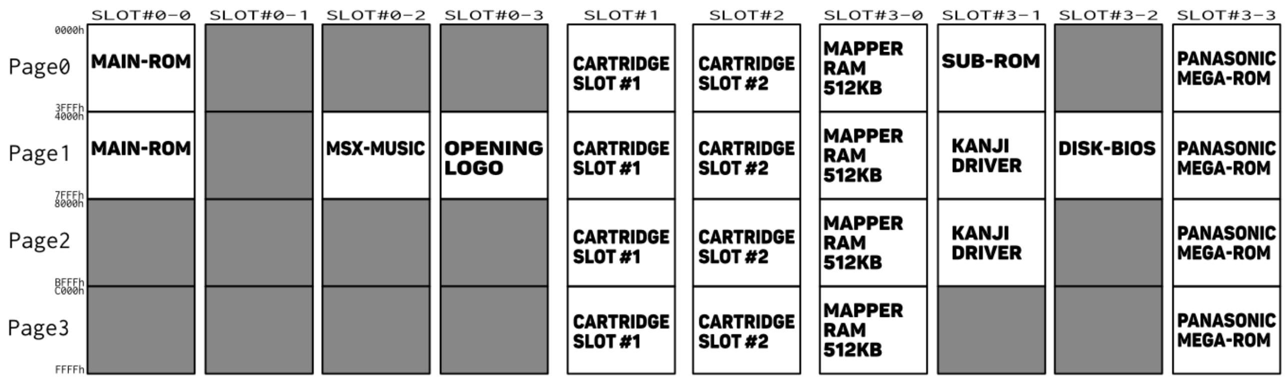


Figure 2-8. FS-A1GT slot configuration

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 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. be converted. On the other hand, MSX1 is not 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. 16KB x 256 = 4096KB = 4MB. An image of 128KB is shown in Figure 3-1.

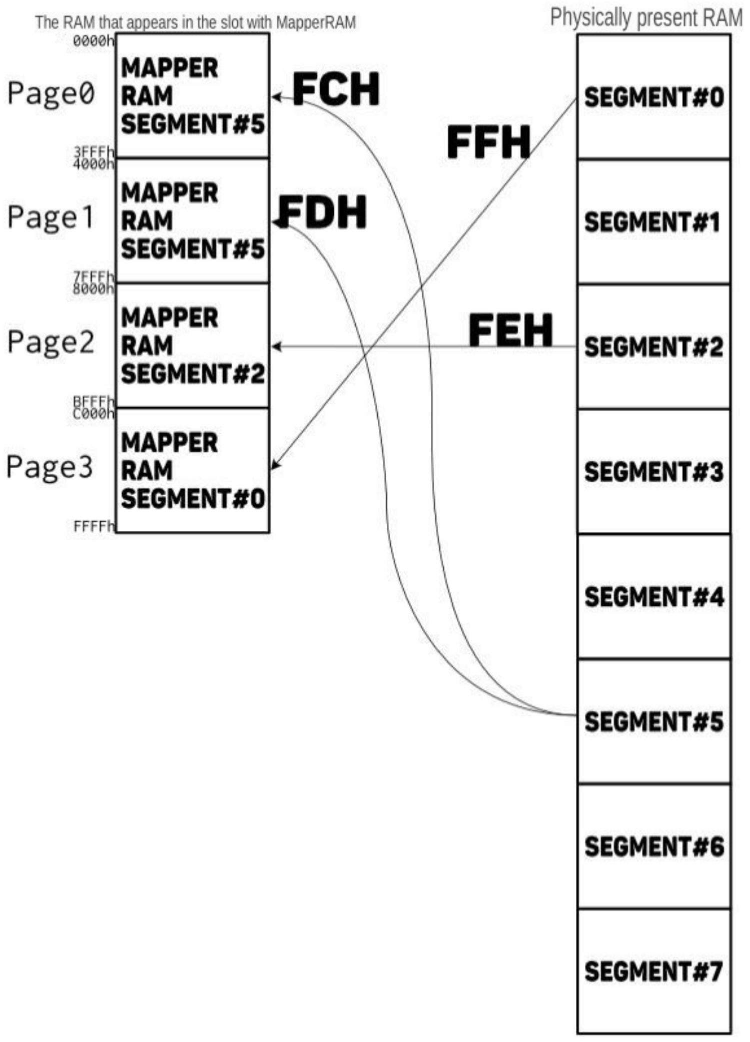


Figure 3-1. Image of slot appearance RAM of 128KB (8 segments)

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 128KB = 16KB × 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 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.

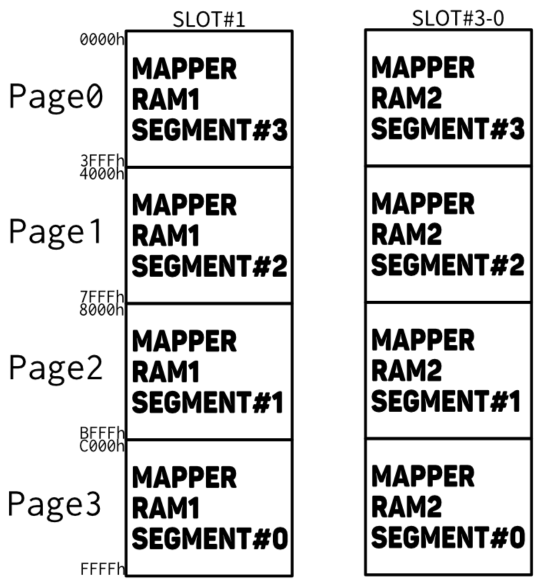


Figure 3-2. Memory mapper compatible RAM in two slots, SLOT#1 and SLOT#3-0

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

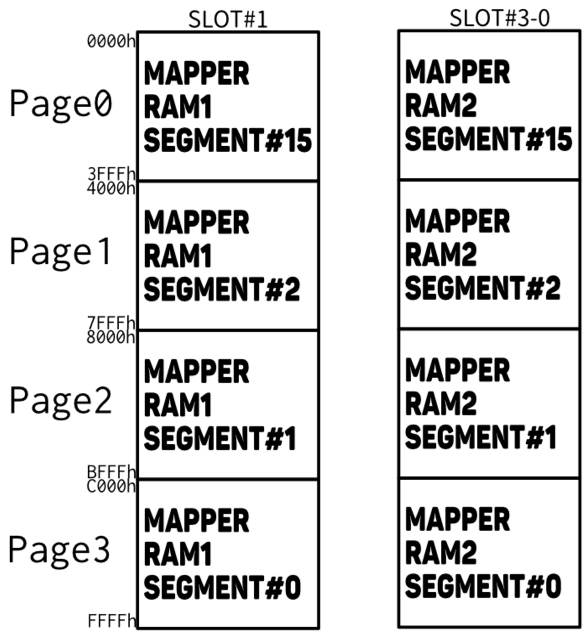


Figure 3-3. Change Segment of Page0 to 15

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.

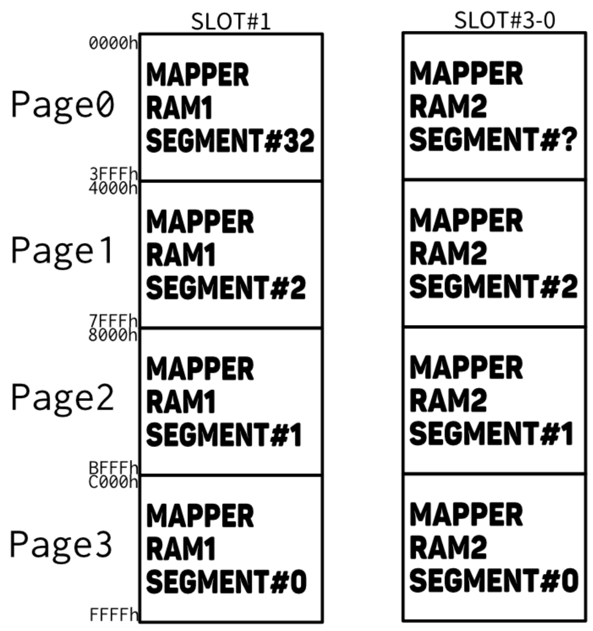


Figure 3-4. Change Segment # of Page0 to 32

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. 768KB = 16KB x 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.

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 largely 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.

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
                                          ; 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
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)

Output :

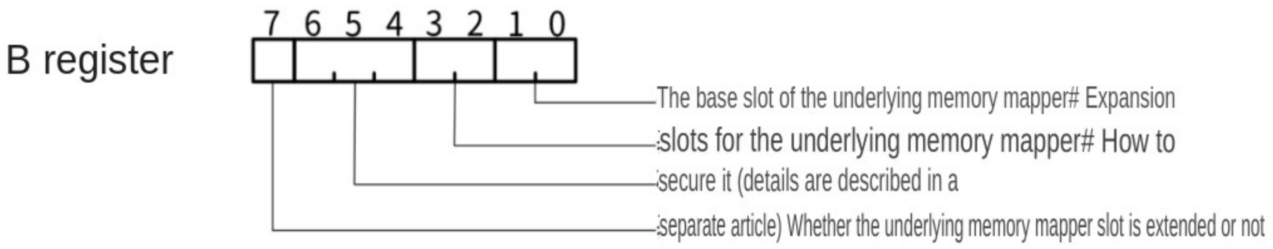
Cy flag: 0 = successful, 1 = no free segments

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.


B register
7 6 5 4 3 2 1 0
The base slot of the underlying memory mapper# Expansion
slots for the underlying memory mapper# How to
secure it (details are described in a
separate article) Whether the underlying memory mapper slot is extended or not

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 register: The meanings of the values specified bits 6,5,4are as follows:

000 Allocated from the base memory mapper.

001 Allocate from a memory mapper other than the reference memory mapper.

010 First, try to allocate from the reference memory mapper, and if there is no free space, allocate from other memory mappers.

011 First, attempt to allocate from a memory mapper other than the reference memory mapper, and if there is no free space, allocate from the reference memory mapper.

1?? undefined (do not specify)

The difference between the user segment and the system segment is that the user segment is a segment that is automatically released when the program ends. Normal memory usage uses user segments. System segments are not freed when the program terminates. Use the system segment only in cases such as resident programs that need to be maintained even after termination.

Also, user segments are assigned in ascending order of segment #. System segments are allocated in descending order of segment #. For example, if segments #4, 5, and 6 are free, allocating a user segment will allocate segment #4, and allocating a system segment will allocate segment #6.

The slot # of the secondary memory mapper can be found from the memory mapper information table.

About this Memory mapper information table section.

FRE_SEG

Free the specified segment.

Input：

A register: Segment # to free

B register: memory mapper slot # of the segment to free

Output：

Cy flag： 0 = free successfully, 1 = free unsuccessful

Detail：

The value specified in the B register is not the value of the B register given as input when calling ALL_SEG . Note that you must specify the value of the B register obtained as a result of calling ALL_SEG .

It fails only when specifying a segment that is not secured by anyone. Note that this will free the segment. For example, in MSXturboR, part of the ROM area such as BIOS is copied to the internal RAM, but on the primary memory mapper, that area is reserved as a system segment. Doing so protects the BIOS copy from other applications, but it also frees it.

RD_SEG

Intersegment read. A: Reads and returns the value of the address indicated by HL。

Input：

A register： Segment # to be read

HL register: In-segment address (higher 2 bits are invalid)

Output：

A register: read data

Other registers are preserved.

Detail：

Returns with interrupts disabled.

Since the Page2 segment is switched internally, the stack memory must be placed outside of Page2.

You must have previously switched to the slot of the memory mapper targeting Page2. This routine reads and returns the value at address HL in segment A of the memory mapper appearing on Page2. Therefore, it is necessary to switch to the slot of the memory mapper from which you want to read Page2.

This function is effective when you want to read from a random address that also includes a segment, but if you want to read continuously from the same segment, it is faster to use direct paging (PUT_P2), which will be described later.

Since processing speed is emphasized, error checks such as whether the specified segment has been secured are not performed. Therefore, there is no return value to indicate an error. The segment specified by the A register is assumed to be under its own control.

WR_SEG

It's an intersegment light. A: Write E to the address indicated by HL.

Input：

A register：Segment # to be read

HL register：In-segment address (upper 2 bits are invalid)

E register：Value to write

Output：

A register：Destroyed.

Other registers are preserved.

Detail：

Returns with interrupts disabled.

Since the Page2 segment is switched internally, the stack memory must be placed outside of Page2.

It is necessary to switch to the slot of the memory mapper that targets Page2 in advance. This routine writes the value of the E register to address HL in segment A of the memory mapper appearing on Page2. Therefore, you need to switch to the memory mapper slot where you want to write Page2.

This function is effective when you want to write from a random address that also includes a segment, but if you want to write continuously from the same segment, it is faster to use direct paging (PUT_P2), which will be described later.

Since processing speed is emphasized, error checks such as whether the specified segment has been secured are not performed. Therefore, there is no return value to indicate an error. The segment specified by the A register is assumed to be under its own control.

CAL_SEG

It's an intersegment call. A routine that calls a subroutine in a specified segment with an interface similar to the inter-slot call CALSLT in the BIOS.

Input :

IY register: segment number specified in upper 8 bits (lower 8 bits are ignored)

IX register: Address of subroutine to call

AF, BC, DE, HL registers: parameters passed to subroutines to call

Output :

AF, BC, DE, HL, IY registers: return values from the called subroutine

Other registers: Destroyed.

Detail :

This routine switches the page corresponding to the address of the subroutine to call to the specified segment and calls the address of the subroutine to call. It then returns the Page corresponding to the address of the subroutine to call to the original segment.

This routine does not switch slots. Segment switching only. It is assumed that the Page corresponding to the upper 2 bits of the IX register has already been switched to the slot of the memory mapper that has the target subroutine.

Since slot switching is a process that requires a relatively heavy load, it seems that slot switching is not included in this routine, assuming that the calling side will minimize it. Memory mapper segment switching is much faster than slot switching, so that process is included.

Of course, Page0 has entries for interrupt routines, and Page3 has BIOS work and so on. There is also stack memory somewhere. Pay close attention to these and perform slot switching in advance before using.

Since the inter-segment call itself exists on Page3, Page3's inter-segment call cannot be used. If Page3 is specified (the high-order 2bits of IX are 11), segment switching will not be performed and it will simply be called.

The back registers (AF', BC', DE', HL') are used in the inter-segment call routine. Please note that they cannot be used for subroutine calls that use these as inputs.

CALLS

It's an intersegment call. A routine that calls a subroutine in a specified segment with an interface similar to the inter-slot call CALLF in the BIOS.

Input :

It is a mechanism that embeds parameters in the code as shown below without using registers.

```
CALL    CALLS
 DB        segment #
 DW        subroutine address
```

AF, BC, DE, HL specifies the input parameters to the routine to be called.

Output :

AF, BC, DE, HL, IX, IYRegisters: Return values from called subroutines

Other registers: Destroyed.

Detail :

This routine does not switch slots. Segment switching only. It is assumed that the Page corresponding to the subroutine address has already been switched to the memory mapper slot containing the desired subroutine.

Since slot switching is a process that requires a relatively heavy load, it seems that slot switching is not included in this routine, assuming that the calling side will minimize it. Memory mapper segment switching is much faster than slot switching, so that process is included.

Of course, Page0 has entries for interrupt routines, and Page3 has BIOS work and so on. There is also stack memory somewhere. Pay close attention to these and perform slot switching in advance before using.

Since the inter-segment call itself exists on Page3, Page3's inter-segment call cannot be used. Even if you specify the address of Page3 as the subroutine address, the segment will not be switched and it will simply be called.

The back registers (AF', BC', DE', HL') are used in the inter-segment call routine. Please note that they cannot be used for subroutine calls that use these as inputs.

PUT_PH

Direct paging. Changes the mapper segment of the specified Page to the specified segment.

Input :

H register : Specify the target page number with the upper 2 bits.

A register : Specify the desired segment #.

Output :

None.

All registers are preserved.

Detailed :

Writes to the mapper segment selection register corresponding to the specified page. The specified page of all memory mappers is switched to the specified segment.

This routine only writes to the mapper segment selection register and stores the written value in the work area, and returns immediately. Compared to slot switching routines, it is considerably faster, so it can be used for frequent switching.

It is assumed that the caller of this routine will set the correspondence between the allocated segment number and the corresponding memory mapper slot number without contradiction. Please note that there is no mechanism to check if there is a contradiction because speed is prioritized.

The specified segment # will be the segment # obtained by securing with ALL_SEG. ALL_SEG also returns the corresponding slot #, but note that slot switching is the responsibility of the caller of this routine. The input does not have a value to select which slot among the possible multiple memory mappers, because it switches to all memory mappers indiscriminately. This area is quite peculiar, but as explained earlier, since the I/O addresses are the same, I think you can understand that it is not possible to switch segments for multiple memory mappers individually.

Note that the Page3 segment cannot be switched. If 0b11xxxxxx (where x is optional) is specified in the H register, return without doing anything.

GET_PH

Direct paging. Get the mapper segment # of the specified Page.

Input :

H register： Specify the target page number with the upper 2 bits.

Output：

A register： Retrieved segment #.

All registers are preserved.

Detailed：

It is assumed that the specified page has been switched to the desired memory mapper slot.

This routine reads and returns the "segment # specified last for the specified page" stored in the work area. We will be back instantly. Compared to slot switching routines, it is considerably faster, so it can be used for frequent switching.

It is assumed that the caller of this routine will set the correspondence between the allocated segment number and the corresponding memory mapper slot number without contradiction. Please note that there is no mechanism to check if there is a contradiction because speed is prioritized.

PUT_P0

Direct paging. Change Page0's mapper segment to the specified segment.

Input：

A register： Specify the desired segment #.

Output：

None.

All registers are preserved.

Detailed：

Please think that the target page of PUT_PH is fixed to Page0. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

GET_P0

Direct paging. Get mapper segment # of Page0.

Input:

None

Output:

A register: Retrieved segment #.

All registers are preserved.

Detailed：

Please assume that the target Page of GET_PH is fixed to Page0. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

PUT_P1

Direct paging. Change Page1's mapper segment to the specified segment.

Input:

A register: Specify the desired segment #.

Output:

None.

All registers are preserved.

Detailed :

Please think that the target page of PUT_PH is fixed to Page1. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

GET_P1

Direct paging. Get mapper segment # of Page1.

Input:

None

Output:

A register: Retrieved segment #.

All registers are preserved.

Detailed :

Please assume that the Page targeted by GET_PH is fixed to Page1. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

PUT_P2

Direct paging. Change Page2's mapper segment to the specified segment.

Input:

A register: Specify the desired segment #.

Output:

None.

All registers are preserved.

Detailed :

Please think that the target page of PUT_PH is fixed to Page2. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

GET_P2

Direct paging. Get mapper segment # of Page2.

Input:

None

Output:

A register: Retrieved segment #.

All registers are preserved.

Detailed :

Please assume that the Page targeted by GET_PH is fixed to Page2. If you use it normally, the target page is often statically determined during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

PUT_P3

Do nothing.

Input:

A register: Specify the desired segment #.

Output:

None.

All registers are preserved.

Detailed：

Please think that the target page of PUT_PH is fixed to Page3. If you use it normally, the target page is often determined statically during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

As a matter of fact, Page3 is where the mapper support routine exists, so it cannot be switched. Therefore, it should be noted that this routine returns without doing anything.

GET_P3

Direct paging. Get mapper segment # of Page3.

Input:

None

Output:

A register: Retrieved segment #.

All registers are preserved.

Detailed :

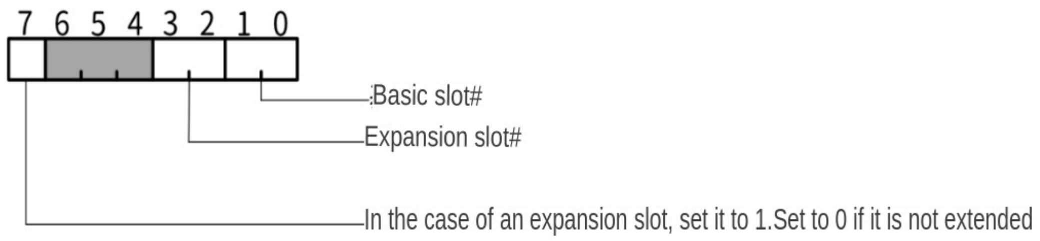
Please assume that the Page targeted by GET_PH is fixed to Page3. If you use it normally, the target page is often determined statically during coding. Please think that is prepared. There is also the advantage that you do not need to destroy the H register just to specify the Page.

Page3 cannot change Segment# with PUT_P3 or PUT_PH due to BIOS work. So GET_P3 always returns 0 in his A register, indicating Segment#0.

Memory mapper information table

When the above initialization sample program (Using mapper support routines) is executed, the address of the memory mapper information table is stored in mmap_table_ptr upon successful initialization.

Information about the memory mapper-compatible RAM currently installed in the MSX system is stored here, and it has an array structure with one entry for each memory mapper-compatible RAM. One entry has the following structure.

Offset address	Meaning
+0	Memory mapper enabled RAM slots#  7 6 5 4 3 2 1 0 Basic slot# Expansion slot# In the case of an expansion slot, set it to 1.Set to 0 if it is not extended
+1	Total number of segments (4-255). (8-255 for primary mapper)
+2	Total number of unused segments
+3	Total number of segments allocated as system segments (minimum 6 for primary mapper)
+4	Total number of segments assigned as user segments
+5～+7	Reserved area

The memory mapper-enabled RAM slot # contains a slot # that can be used for slot control routines in the BIOS, such as ENASLT. In the memory mapper support routine ALL_SEG, the primary mapper can be used with B=0 without specifying the slot #, but in order to actually use the allocated segment, it is necessary to switch to that slot with ENASLT etc. For the slot # that is required at that time, use the value of this "RAM slot # for memory mapper".

The total number of segments value will be at least 8 for the primary mapper. 8 segments x 16 KB = 128 KB, which matches the requirement specification of MSX-DOS2 "minimum 128 KB required".

The total number of segments allocated as system segments will be at least 6 for the primary mapper. This is because MSX-DOS2 itself reserves his 4 segments for his TPA and 2 segments for his DOS2 work area. In the case of MSXturboR, the BIOS copy area for DRAM mode is also this system segment, so the total number of segments already allocated as system segments is 10.

This 8-byte structure is lined up as many times as the number of memory mapper compatible RAMs, and then 0 is stored as a terminator. MSX becomes a MAIN-ROM slot if SLOT#0 is not expanded. Therefore, SLOT#0 does not exist in any model of memory mapper compatible RAM. Taking advantage of this, the memory mapper support routines often treat the value “0” specially where slot

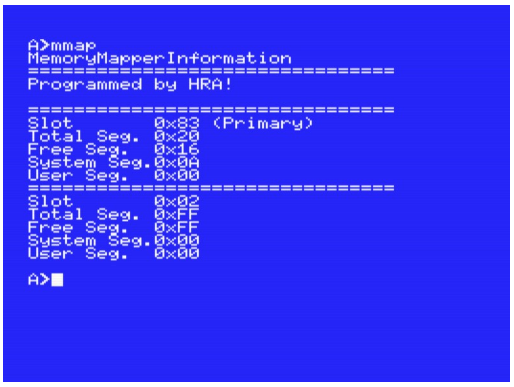
# is specified. The first byte of the memory mapper information table indicates the slot # of the memory mapper compatible RAM, but if 0 is entered here, it indicates the end of the memory mapper information table, which may be arranged continuously. It is treated as a terminator.

The mapper segment selection register has 256 ways from 0 to 255, 16KB x 256 segments = 4096KB, but the maximum total number of segments is 255. Only up to 4080KB can be used because it only supports up to segments.

As an example, the values in the memory mapper information table when a 4MB memory mapper compatible RAM cartridge is installed in SLOT#2 of the FS-A1GT are shown below.

Memory mapper	Offset address	meaning	value
primary	+0	Memory mapper enabled RAM slots #	83h (SLOT#3-0)
	+1	Total number of segments	32 segment
	+2	Total number of unused segments	22 segment
	+3	Total number of segments allocated as system segments	10 segment
	+4	Total number of segments assigned as user segments	0 segment
	+5～+7	Reserved area	all 0
Secondary 1	+8	Memory mapper enabled RAM slots#	02h (SLOT#2)
	+9	Total number of segments	255 segment
	+10	Total number of unused segments	255 segment
	+11	Total number of segments allocated as system segments	0 segment
	+12	Total number of segments assigned as user segments	0 segment
	+13～+15	Reserved area	all 0
Terminator	+16	Memory mapper enabled RAM slots#	0

Figure 3-5 shows a screen that was actually checked using the tool.



```
A>mmap
MemoryMapperInformation
================================
Programmed by HRA!

================================
Slot        0x83 (Primary)
Total Seg.  0x20
Free Seg.   0x16
System Seg.0x0A
User Seg.   0x00
================================
Slot        0x02
Total Seg.  0xFF
Free Seg.   0xFF
System Seg.0x00
User Seg.   0x00

A>█
```

Figure 3-5. Memory mapper information table dump tool display

Below is the source code of the memory mapper information table dump tool used in Figure 3-5.

Because another source that includes is not an essential part of the explanation in this section, (Appendix 1. Various sources used by the sample program) I'll post it on.

mmap.asm

```
; ==============================================================================
;  Memory Mapper Program to dump information
; ==============================================================================

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

               org            0x100

entry::
               ; Initialize
               ld             sp, [TPA_BOTTOM]
               call           mmap_init

               ; startup message
               ld             de, msg_entry
               call           puts

               ; mapper check
               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
               jp             bdos

; ==============================================================================
;       Dump a mapper information
;       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
               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. Mega ROM

A mega-ROM is a ROM that exceeds 1 Mbit. Since the memory space of Z80 is only 64KB=0.5Mbit, a bank switching mechanism is provided to handle 1Mbit or more.

The part or part group that implements the bank switching mechanism is called a mega ROM controller, or megacon for short. The MSX standard does not specify this content, and there are various types.

ASCII has released several megacon that can be used with MSX, and many software uses this. There are two types of ASCII megacon when roughly classified from the software point of view. There are two types, ASCII-8K type and ASCII-16K type. The types of megacon ICs are LZ93A13 (32pin), M60002-0125SP (42pin), BS6101 (42pin), NEOS MR6401 (28pin), BS6202 (42pin), IREM TAM-S1 (28pin), etc.[*] However, since this document focuses on software control, we will explain two types: ASCII-8K type and ASCII-16K type. No mention of IC differences.

In addition, Konami has its own megacon. This manual describes the three types of SCC-less version, SCC, and SCC-I.

The MSX game cartridge has already become a valuable commodity, so if you were to make your own mega ROM cartridge, it would be more realistic to create a new mega computer using programmable logic devices such as CPLDs and FPGAs. It is possible to create a megacon with your own specifications, and I think it would be good to design such a megacon for copy protection. However, at the stage of developing software such as games to be written to the cartridge, by developing compatible with ASCII or his Konami mega-computer used in conventional commercial products, there is an advantage that you can easily check the operation on the MSX emulator. there is. I think it would be advantageous in terms of development efficiency to make modifications according to the original specification megacon as needed after the software is assembled. Of course, I think that it is possible to make a cartridge with a megacon that is compatible with the conventional megacon.

For that reason, I think it is meaningful to know how to control the megacon, so I would like to explain each type of megacon in this chapter.

Panasonic's MSX2+ and later bodies have their own megacon. I don't think there is anyone who dares to make this compatible, but I would like to explain this megacon as it may be a reference when investigating the behavior of FS-A1GT. There are FS-A1FX/WX/WSX/ST/GT, but there seems to be subtle differences with revisions in this megacon. This document mainly describes the FS-A1GT megacon.

[*] information source: https://gigamix.hatenablog.com/entry/rom/

4.1. ASCII-8K type

The ASCII-8K type divides the ROM into 8KB units and assigns consecutive numbers. This is a method to access by making this appear in BANK0 (4000h-5FFFh), BANK1 (6000h-7FFFh), BANK2 (8000h-9FFFh), BANK3 (A000h-BFFFh). The Nth consecutive number assigned is written as BANK#N here.

Page 1 and Page 2 form 4 BANKs, with 2 BANKs inside the Page explained in the slot section. These four are called BANK0 to BANK3 in order from the smallest value address. By making several BANKs (this is the actual ROM) in the mega ROM appear in this BANK0 to BANK3 (this is the address range) while switching, it is possible to access the entire ROM. .

For example, 1Mbit ASCII-8K type mega-ROM is 1[Mbit] = 128[KB] = 8[KB] × 16[BANK], so there are 16 BANKs, so the configuration is as shown in Figure 4.1-1. will be:

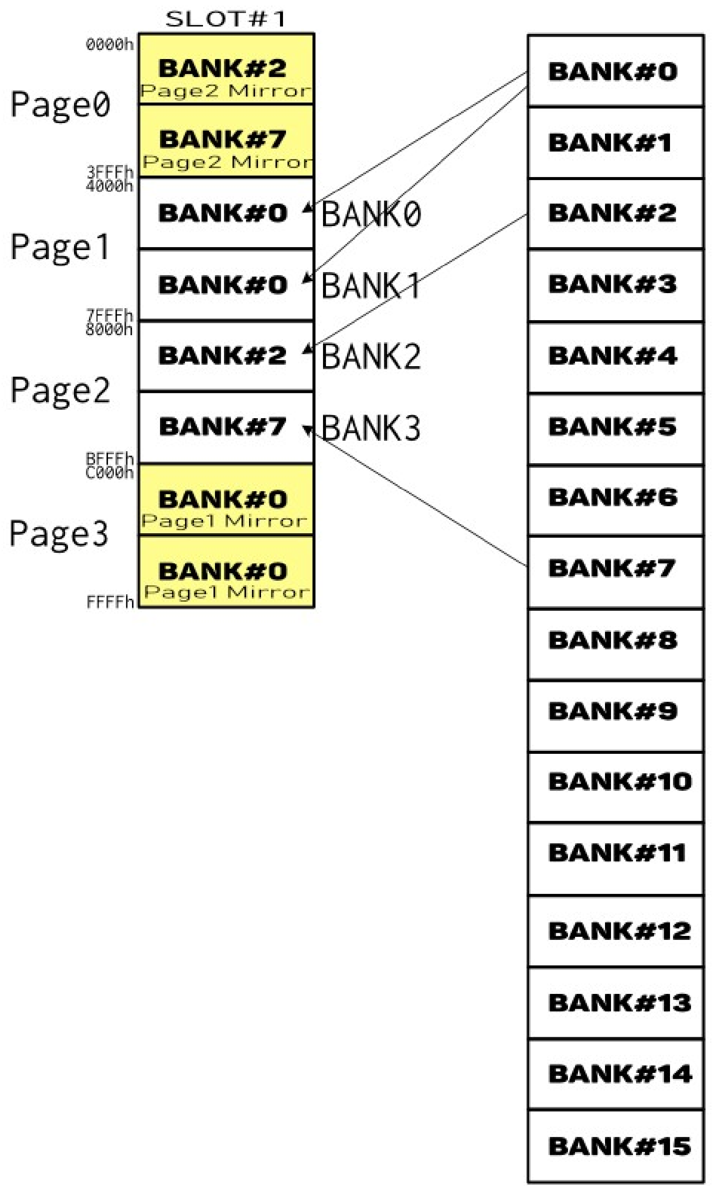


Figure 4.1-1. Image of ASCII-8K type 1Mbit mega ROM

Figure 4.1-1 is an image of 1Mbit mega ROM installed in SLOT#1. His Page0 in SLOT #1 has a mirror of Page2, and Page3 has a mirror of Page1.

BANK0 and BANK1 exist in Page1, and BANK2 and BANK3 exist in Page2.

BANK#0 to #15, which are 16 lined up on the right side of the figure, will be the ROM. 8KB x 16 = 128KB because there are 16 8KB each.

The four arrows connecting the left side and right side of the figure are the bank selection registers. There is a bank selection register for each of BANK0 to

BANK3, and any BANK# from BANK#0 to #15 can be specified. Therefore, the same BANK# can appear in multiple BANKs at the same time. In the figure, BANK#0 appears in BANK0 and BANK1.

The bank selection register is Memory mapped I/O. Table 4.1-1 summarizes the bank selection register addresses.

Table 4.1-1.ASCII-8K Bank Select Register Address

BANK number	Corresponding bank select register address
BANK0 (4000h-5FFFh)	6000h-67FFh (initial value BANK#0)
BANK1 (6000h-7FFFh)	6800h-6FFFh (initial value BANK#0)
BANK2 (8000h-9FFFh)	7000h-77FFh (initial value BANK#0)
BANK3 (A000h-BFFFh)	7800h-7FFFh (initial value BANK#0)

The address of the bank selection register indicates a range of 2048 bytes, but the actual size is only 1 byte. So 8KB × 256 = 2048KB = 2MB is the maximum capacity. The reason why it has a range is that only part of the address signal is used for determination in order to simplify the megacon circuit. We'll explain why it's simple in the hardware chapter.

As you can see, all bank selection registers exist on Page1. Since the bank selection registers of BANK2 and BANK3 in Page2 also exist in Page1, for example, when the RAM slot of the main body appears on Page1 and the slot of this mega ROM appears on Page2, when switching BANK2 and BANK3, Please note that it is necessary to temporarily switch Page1 to the MegaROM slot, rewrite the bank selection registers of BANK2 and BANK3, and then return Page1 to the RAM slot of the main unit.

The bank selection register is write-only. When read, the contents of the ROM corresponding to that address are read.

The reason why the bank selection registers are gathered in Page1 may be consideration for the backup SRAM. There is also a mega computer that supports SRAM for backup. This is a structure in which /WE via the mega-con is set to L only when some banks are accessed. If there is a bank selection register when accessing SRAM, it will be a hindrance, so it may be assumed that SRAM appears on Page2 and accessed.

4.2. ASCII-16K type

The ASCII-16K type divides the ROM into 16KB units and assigns consecutive numbers. This is a method to access by making this appear in BANK0 (4000h-7FFFh) and BANK1 (8000h-BFFFh). The Nth consecutive number assigned is written as BANK#N here.

The size of the mega-ROM BANK is the same 16KB as explained in the page on the slot. Page1 and Page2 constitute two banks. These two are called BANK0 to BANK1 in order from the smallest value address. In other words, the corresponding relationships are BANK0=Page1 and BANK1=Page2. By making several BANKs (this is the actual ROM) that are installed in the mega ROM appear in this BANK0 to BANK1 (this is the address range) while switching, it is possible to access the entire ROM.

For example, 1Mbit ASCII-16K type mega-ROM is 1[Mbit] = 128[KB] = 16[KB] × 8[BANK], so there are 8 BANKs, so the configuration is as shown in Figure 4.2-1 will be:

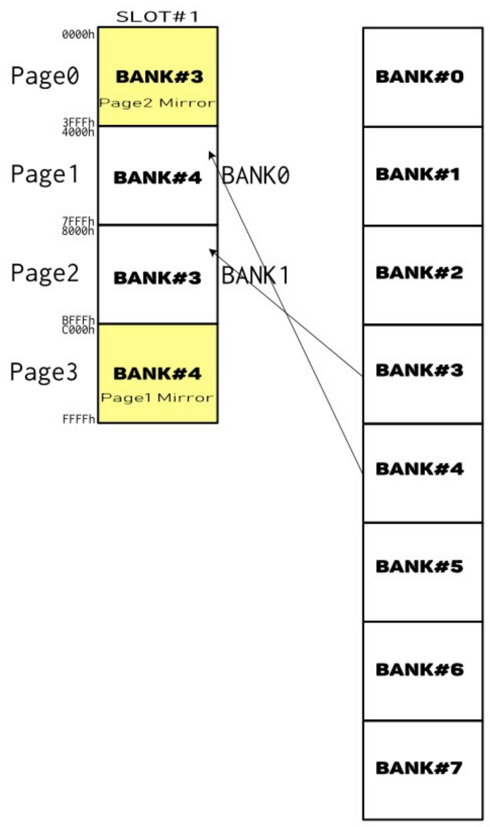


Figure 4.2-1.Image of ASCII-16K type 1Mbit mega ROM

Figure 4.2-1 is an image of 1Mbit mega ROM installed in SLOT#1. His Page0 in SLOT #1 has a mirror of Page2, and Page3 has a mirror of Page1.

BANK0 exists in Page1 and BANK1 exists in Page2.

BANK#0 to #7, which are 8 lined up on the right side of the figure, will be the ROM. 16KB x 8 = 128KB because there are 8 of them at 16KB each.

The two arrows connecting the left side of the figure and the right side of the figure are the bank selection registers. There is a bank selection register for each of BANK0 to BANK1, and any BANK# from BANK#0 to #7 can be specified. Therefore, the same BANK# can appear in multiple BANKs at the same time.

The bank selection register is Memory mapped I/O. Table 4.2-1 summarizes the bank selection register addresses.

Table 4.2-1.ASCII-16K Bank Select Register Address

BANK number	Corresponding bank select register address
BANK0 (4000h-7FFFh)	6000h-67FFh (initial value BANK#0)
BANK1 (8000h-BFFFh)	7000h-77FFh (initial value BANK#0)

The address of the bank selection register indicates a range of 2048 bytes, but the actual size is only 1 byte. So 16KB × 256 = 4096KB = 4MB is the maximum capacity. I think the reason for having a range is to simplify the megacon circuit. We'll explain why it's simple in the hardware chapter.

As you can see, all bank selection registers exist on Page1. Since the bank selection register of BANK1 in Page2 also exists in Page1, for example, when Page1 shows the main RAM slot and Page2 shows this mega ROM slot, when switching BANK1, temporarily please note that it is necessary to switch Page1 to the mega ROM slot, rewrite the bank selection register of BANK1, and then return Page1 to the RAM slot of the main unit.

The bank selection register is write-only. When read, the contents of the ROM corresponding to that address are read.

The reason why the bank selection registers are gathered in Page1 may be consideration for the backup SRAM. There is also a mega computer that supports SRAM for backup. This is a structure in which /WE via the mega-con is set to L only when some banks are accessed. If there is a bank selection register when accessing SRAM, it will be a hindrance, so it may be assumed that SRAM appears on Page2 and accessed.

4.3. Konami-8K type

This is the Mega ROM type used in Konami games before the SCC was installed.

The Konami-8K type divides the ROM into 8KB units and assigns consecutive numbers. This is a method to access by making this appear in BANK0 (4000h-5FFFh), BANK1 (6000h-7FFFh), BANK2 (8000h-9FFFh), BANK3 (A000h-BFFFh). The Nth consecutive number assigned is written as BANK#N here.

The BANK switching method is the same as ASCII-8K. However, the address of the bank selection register is different. The Konami-8K type bank selection register addresses are summarized in Table 4.3-1.

Table 4.3-1.Konami-8K Bank Select Register Address

BANK number	Corresponding bank select register address
BANK0 (4000h-5FFFh)	None (BANK#0 fixed)
BANK1 (6000h-7FFFh)	6000h-7FFFh (initial value BANK#1)
BANK2 (8000h-9FFFh)	8000h-9FFFh (initial value undefined)
BANK3 (A000h-BFFFh)	A000h-BFFFh (initial value undefined)

Unlike ASCII-8K , BANK0 is fixed at BANK#0 . A different BANK# cannot be specified.

BANKN's bank select register resides within that BANKN address range. If the slot of the BANKN you want to switch to appears in the Z80 memory space, you can specify the BANK#N that appears in that BANKN. For ROM only, I think it's easier to use than ASCII-8K.

The maximum capacity that can be handled is 512KB. In other words, up to 64 [BANK]. N of BANK#N takes the range of 0-63.

BANK0 to BANK3 exist on Page1 and Page2, but Page0 has a mirror of Page2 (BANK2 and BANK3) and Page3 has a mirror of Page1 (BANK0 and BANK1), just like ASCII-8K.

4.4. Konami SCC type

Any Konami ROM game that says it supports SCC falls into this category. There are also controllers compatible with this type of MegaROM controller, such as MegaFlashROM SCC, and because SCC sound sources can be used, some people create and release their own games with this type.

The BANK switching method is the same as ASCII-8K. However, the address of the bank selection register is different. Table 4.4-1 summarizes the bank selection register addresses of the Konami-SCC type.

Table 4.4-1.Konami-SCC Bank Select Register Address

BANK number	Corresponding bank select register address
BANK0 (4000h-5FFFh)	5000h-57FFh (initial value BANK#0)
BANK1 (6000h-7FFFh)	7000h-77FFh (initial value BANK#1)
BANK2 (8000h-9FFFh)	9000h-97FFh (initial value BANK#2)

BANK3 (A000h-BFFFh)	B000h-B7FFh (initial value BANK#3)

Unlike ASCII-8K, BANKN's bank selection register resides within its BANKN address range. If the slot of the BANKN you want to switch to appears in the Z80 memory space, you can change the BANK#N that appears in that BANKN. For ROM only, I think it's easier to use than ASCII-8K.

The maximum capacity that can be handled is 512KB, that is, up to 64 [BANK]. N of BANK#N takes the range of 0-63.

BANK0 to BANK3 exist on Page1 and Page2, but Page0 has a mirror of Page2 (BANK2 and BANK3) and Page3 has a mirror of Page1 (BANK0 and BANK1), just like ASCII-8K.

The Konami-SCC type megacon is equipped with an SCC sound source. There is a register for controlling the SCC sound source. The bank selection register only takes a range of 0 to 63, but if you write 63 (3Fh) to the bank selection register of BANK2, the SCC sound source register space will appear in BANK2 as shown in Table 4.4-2. In other words, BANK#63 cannot appear in BANK2. If you want to access BANK#63, please use BANK0/1/3.

Table 4.4-2. Register of SCC sound source (BANK2-BANK#63)

Address	Size (byte)	Meaning
8000h-8FFFh	4096	Unused??
9000h-97FFh	2048	Bank selection register for BANK2. The entity is 1 byte. Write only.
9800h-981Fh	32	wave memory－0 (for Ch.A), read/write
9820h-983Fh	32	wave memory－1 (for Ch.B), read/write
9840h-985Fh	32	wave memory－2 (for Ch.C), read/write
9860h-987Fh	32	wave memory－3 (for Ch.D/E), read/write
9880h-9881h	2	Ch.A frequency register(12bit)、Write only.
9882h-9883h	2	Ch.B frequency register(12bit)、Write only.
9884h-9885h	2	Ch.C frequency register(12bit)、Write only.
9886h-9887h	2	Ch.D frequency register(12bit)、Write only.
9888h-9889h	2	Ch.E frequency register(12bit)、Write only.
988Ah	1	Ch.A Volume (4bit)、Write only.
988Bh	1	Ch.B Volume (4bit)、Write only.

988Ch	1	Ch.C Volume (4bit), write only.
988Dh	1	Ch.D Volume (4bit), write only.
988Eh	1	Ch.E Volume (4bit), write only.
988Fh	1	output switch, Write only. bit0: Ch.A (0: mute, 1: output) bit1: Ch.B (0: mute, 1: output) bit2: Ch.C (0: mute, 1: output) bit3: Ch.D (0: mute, 1: output) bit4: Ch.E (0: mute, 1: output)
9890h-989Fh	16	9880h-988Fh mirror.
98A0h-98BFh	32	Unused. Not read/write.
98C0h-98DFh	32	Unused??
98E0h-98FFh	32	Mode register.Write only. The substance is 1byte. bit0: frequency register specification of treatment of 1 bit1: frequency register specification of treatment of 2 bit5: frequency register waveform pointer reset on write bit6: All Ch. Waveform Rotate bit7: Ch.D. Waveform Rotate
9900h-9FFFh	1792	Unused??

Even if you read a register that is written as write-only, you cannot read a value with meaning.

The explanation of the SCC sound source is outside the scope of this book, so I will omit it.

4.5. Konami SCC-I type

So-called Snatcher sound cartridges and SD Snatcher sound cartridges fall into this category. In fact, it has DRAM instead of ROM.

As a mega ROM controller, it is the same as the Konami-SCC type, so please refer to 4.4. Konami SCC type for bank selection registers. On the other hand, there are differences due to the SCC installation, so I would like to explain the differences in this section.

The Snatcher sound cartridge is equipped with 64KB of DRAM as BANK#0 to BANK#7. On the other hand, the SD Snatcher sound cartridge has 64KB of DRAM as BANK#8 to BANK#15.

There is a pattern on the sound cartridge board that allows you to add DRAM, so you can add two 4-bit × 64K DRAMs. As a board, four 4-bit × 64K DRAMs can be mounted, and the place where the DRAM is installed is different between Snatcher and SD Snatcher. If you add DRAM to the two empty places, it will be a sound cartridge with 128KB of DRAM installed as BANK#0 to BANK#15.

The Konami SCC-I type has operation mode setting registers at BFFEh and BFFFh. BFFEh and BFFFh are the same register, and the actual size is only 1 byte. Figure 4.5-1 shows the bitmap of the operation mode setting register. The initial value seems to be 00h.

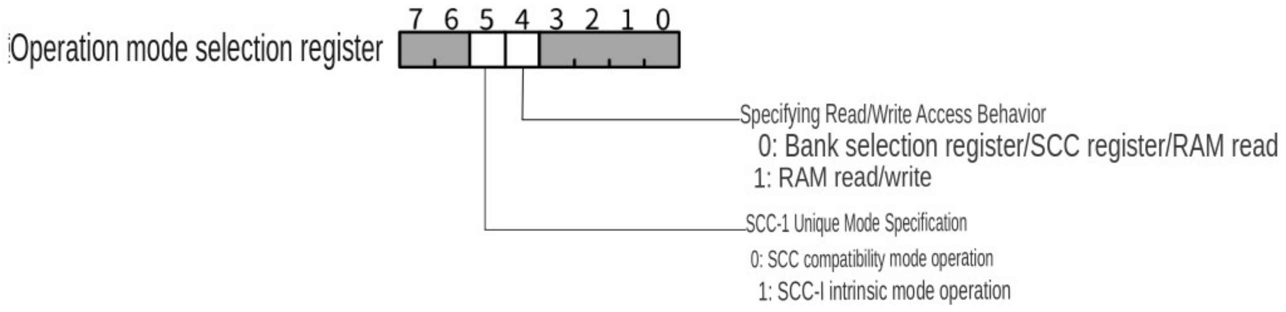

7 6 5 4 3 2 1 0
Operation mode selection register
Specifying Read/Write Access Behavior
0: Bank selection register/SCC register/RAM read
1: RAM read/write
SCC-1 Unique Mode Specification
0: SCC compatibility mode operation
1: SCC-I intrinsic mode operation

Fig 4.5-1. SCC-I Operation mode selection register

The operation mode selection register is write-only. Access is always possible regardless of the setting value of the operation mode selection register.

If you want to access the bank selection register or the SCC/SCC-I sound source register, you need to set bit4 to 0. Setting bit4 to 1 enables writing to the RAM in the sound cartridge. For example, when writing to address 9123h, if bit4 of BFFEh is 0, it will be written to the bank selection register of BANK2, but if bit4 of BFFEh is 1, it will be written to the RAM appearing in BANK2.

The SCC-I's SCC sound source section has two types: a mode compatible with the Konami SCC type and a mode expanded with a sound cartridge. Here, the former is called compatibility mode, and the latter is called eigenmode.

The compatibility mode is also slightly different compared to the Konami SCC type. The SCC sound source registers are shown in Table 4.5-1. Differences are shown in red.

Table 4.5-1. Compatibility mode SCC tone generator registers (BANK2-BANK#63)

Address	Size (byte)	meaning
8000h-8FFFh	4096	Unused??
9000h-97FFh	2048	BANK2 bank select register. The entity is 1 byte. Write only.
9800h-981Fh	32	wave memory – 0 (for Ch.A), readable and writable.
9820h-983Fh	32	wave memory – 1 (for Ch.B), readable and writable.
9840h-985Fh	32	wave memory – 2 (for Ch.C), readable and writable.
9860h-987Fh	32	wave memory – 3 Ch.D/E shared when writing. Ch.D when reading.
9880h-9881h	2	Ch.A frequency register(12bit), write only.
9882h-9883h	2	Ch.B frequency register(12bit), write only.
9884h-9885h	2	Ch.C frequency register(12bit), write only.
9886h-9887h	2	Ch.D frequency register(12bit), write only.
9888h-9889h	2	Ch.E frequency register(12bit), write only.
988Ah	1	Ch.A Volume (4bit), write only.
988Bh	1	Ch.B Volume (4bit), write only.
988Ch	1	Ch.C Volume (4bit), write only.
988Dh	1	Ch.D Volume (4bit), write only.
988Eh	1	Ch.E Volume (4bit), write only.
988Fh	1	output switch, Write only. bit0: Ch.A (0: mute, 1: output) bit1: Ch.B (0: mute, 1: output) bit2: Ch.C (0: mute, 1: output) bit3: Ch.D (0: mute, 1: output) bit4: Ch.E (0: mute, 1: output)
9890h-989Fh	16	9880h-988Fh mirror.
98A0h-98BFh	32	wave memory – 4 (for Ch.E), read only.
98C0h-98DFh	32	Mode register.Write only. The entity is 1 byte. bit0: frequency register specification of treatment of 1 bit1: frequency register specification of treatment of 2

		bit5: frequency register waveform pointer reset on write bit6: All Ch. Waveform Rotate bit7: Invalid
98E0h-98FFh	32	Unused??
98E0h-9FFFh	1792	Unused??

Even if you read a register that is written as write-only, you cannot read a value with meaning.

9860h-987Fh writes the same value to both Ch.D wave memory and Ch.E wave memory for compatibility with SCC when writing. Reading 9860h-987Fh returns the value of Ch.D wave memory. If you want to read Ch.E wave memory, you can use 98A0h-98BFh. While only compatible mode is used, reading 9860h-987Fh and reading 98A0h-98BFh will return the same value, but once you switch to unique mode and rewrite the wave memory of Ch.E, then switch to compatible mode If you read it back, it seems that you can read a different waveform properly.

Please note that the Konami-SCC type and the address of the mode register are different.

Next, let's move on to explaining eigenmodes.

The unique mode SCC sound source register appears in BANK3, unlike the compatibility mode. I think this change was made because it is more convenient to use BANK0-2 continuously. The SCC sound source register is BANK#128 (strictly speaking, it is the same even if BANK#128 to BANK#255 are specified because all BANK#s have bit7 = 1 in the bank selection register) . Eigen mode allows you to set different waveforms for Ch.D and Ch.E. Register addresses are summarized in Table 4.5-2.

Tabela 4.5-2. Registros de fonte de som SCC de modo próprio (BANK3-BANK#128)

Address	Size (byte)	meaning
A000h-AFFFh	4096	Unused??
B000h-B7FFh	2048	Bank selection register. The entity is 1 byte. Write only.
B800h-B81Fh	32	wave memory ─ 0 (for Ch.A), readable and writable.

B820h-B83Fh	32	wave memory－1 (for Ch.B), readable and writable.
B840h-B85Fh	32	wave memory－2 (for Ch.C), readable and writable.
B860h-B87Fh	32	wave memory－3 (for Ch.D), readable and writable.
B880h-B89Fh	32	wave memory－4 (for Ch.E), readable and writable.
B8A0h-B8A1h	2	Ch.A frequency register(12bit), write only.
B8A2h-B8A3h	2	Ch.B frequency register(12bit), write only.
B8A4h-B8A5h	2	Ch.C frequency register(12bit), write only.
B8A6h-B8A7h	2	Ch.D frequency register(12bit), write only.
B8A8h-B8A9h	2	Ch.E frequency register(12bit), write only.
B8AAh	1	Ch.A Volume (4bit), write only.
B8ABh	1	Ch.B Volume (4bit), write only.
B8ACh	1	Ch.C Volume (4bit), write only.
B8ADh	1	Ch.D Volume (4bit), write only.
B8AEh	1	Ch.E Volume (4bit), write only.
B8AFh	1	output switch, Write only. bit0: Ch.A (0: mute, 1: output) bit1: Ch.B (0: mute, 1: output) bit2: Ch.C (0: mute, 1: output) bit3: Ch.D (0: mute, 1: output) bit4: Ch.E (0: mute, 1: output)
B8B0h-B8BFh	16	B8A0h-B8AFh mirror.
B8C0h-B8DFh	32	Mode register.Write only. The entity is 1 byte. bit0: frequency register specification of treatment of 1 bit1: frequency register specification of treatment of 2 bit5: frequency register waveform pointer reset on write bit6: All Ch. Waveform Rotate bit7: Invalid
B8E0h-BFFDh	1822	Unused??
BFFEh-BFFFh	2	Operating mode selection register. The entity is 1 byte. Both addresses are the same.

Even if you read a register that is written as write-only, you cannot read a value with meaning.

The explanation of the SCC sound source is outside the scope of this book, so I will omit it.

4.6. Panasonic type

This is the mega ROM type installed in the Panasonic body (FS-A1FX/WX/WSX/ST/GT). Built-in software, MSX-JE, etc. are stored, and it is connected to SLOT#3-3.

The BANK size is 8KB, which is similar to the ASCII-8K type, but Page0 and Page3 are not mirrors, and they also have independent bank selection registers. The bank selection register also has 9 bits per BANK, and can be selected within the range of BANK#0 to BANK#511. 8[KB] × 512[BANK] = 4096[KB] = 4[MB] space can be handled.

He probably has the most memory attached to his FS-A1GT, but even with his FS-A1GT he doesn't use the full 4MB space. It seems that ROM-2MB, DRAM-512KB, and SRAM-32KB are connected in the form of part of his 4MB space (SRAM is battery-backed RAM for MSX-JE's learning dictionary).

The image seen from the slot is shown in Figure 4.6-1.

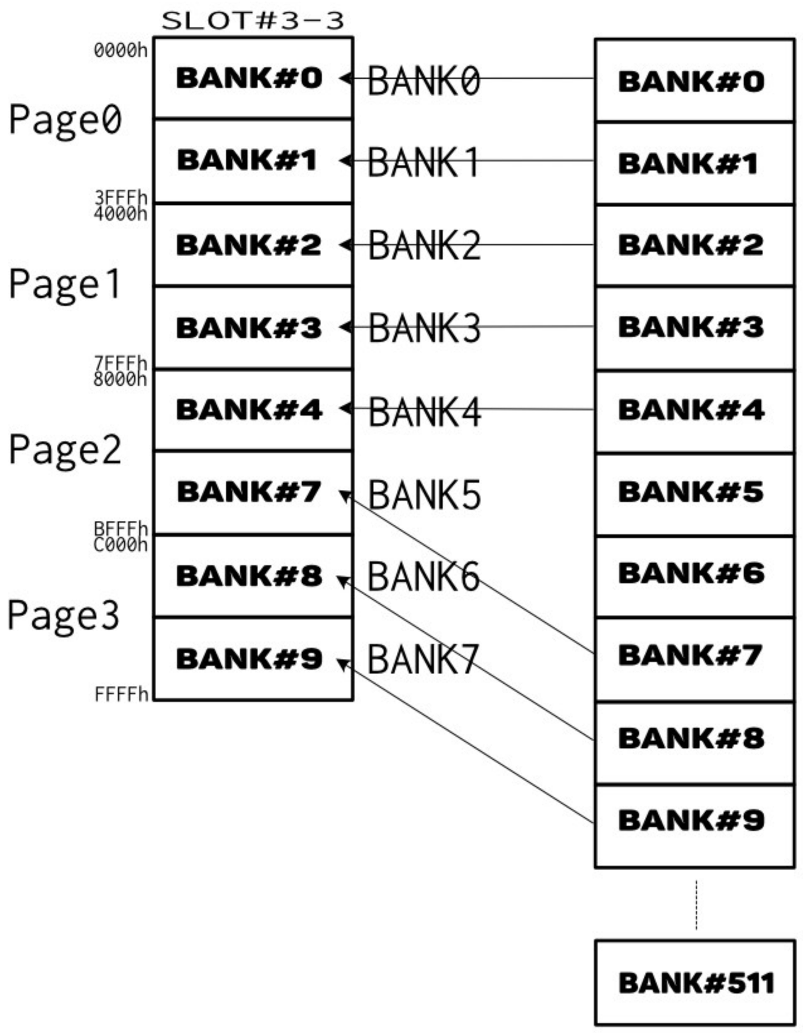


Figure 4.6-1. Image of Panasonic type Mega ROM

Address of bank register is shown in Table 4.6-1.

Table 4.6-1. Panasonic type Mega ROM bank register

Address	Size (byte)	meaning
6000h-63FFh	1024	The entity is 1 byte. Write only. Lower 8 bits of bank selection register of BANK0
6400h-67FFh	1024	The entity is 1 byte. Write only. Lower 8 bits of bank selection register of BANK1
6800h-6BFFh	1024	The entity is 1 byte. Write only. Lower 8 bits of bank selection register of BANK2
6C00h-6FFFh	1024	The entity is 1 byte. Write only. Lower 8 bits of bank selection register of BANK3

7000h-73FFh	1024	The entity is 1 byte. Write only. Lower 8 bits of bank selection register of BANK4
7400h-77FFh	1024	The substance is 1byte. Write only. BANK6 lower 8 bits of bank selection register
7800h-7BFFh	1024	The substance is 1byte. Write only. BANK5 lower 8 bits of bank selection register
7C00h-7FEFh	1008	The substance is 1byte. Write only. BANK7 lower 8 bits of bank selection register
7FF0h	1	Read-only. BANK0 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF1h	1	Read-only. BANK1 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF2h	1	Read-only. BANK2 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF3h	1	Read-only. BANK3 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF4h	1	Read-only. BANK4 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF5h	1	Read-only. BANK5 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF6h	1	Read-only. BANK6 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF7h	1	Read-only. BANK7 cannot be read unless bit2 of is set to 1. 7FF9h cannot be read unless bit2 of is set to 1.
7FF8h	1	Readable and writable. Upper 1bit of bank selection register. bit0: BANK0 high-order 1bit of bank selection register. bit1: BANK1 high-order 1bit of bank selection register. bit2: BANK2 high-order 1bit of bank selection register. bit3: BANK3 high-order 1bit of bank selection register.

		bit4: BANK4 high-order 1bit of bank selection register. bit5: BANK5 high-order 1bit of bank selection register. bit6: BANK6 high-order 1bit of bank selection register. bit7: BANK7 high-order 1bit of bank selection register. 7FF9h cannot be read or written unless bit4 of is set to 1.
7FF9h	1	Mode register. Readable and writable. bit0: Invalid bit1: Invalid bit2: 7FF0h-7FF7h activation bit3: 7FF9h reading activation bit4: 7FF8h activation bit5: Invalid bit6: Invalid bit7: Invalid To read this register, you must first write 1 to bit3 of this register.

According to the reference material, FS-A1WX/WSX does not seem to have 7FF8h. Bit4 of 7FF9h is also Invalid. FS-A1FX is unknown.

Although it is an area to write the lower 8 bits of the bank selection register, it is not in a clean order. Please note that the order of BANK5 and BANK6 has been reversed, as shown in red in the table.

"Then, what are you connecting with this megacon?" The case of FS-A1GT is summarized below.

FS-A1GT is equipped with a total of 2MB of ROM, but this megacon is connected so that almost all of the installed ROM can be accessed (only the 16-dot Kanji ROM seems to be not included). Not only that, it seems that the built-in RAM can also be accessed via this megacon.

I will summarize below as far as I can understand.

<<Currently under investigation>>

BANK#	Content
BANK#0-#39	Built-in software 320KB (40BANK) Each 16KB (2BANK) has a signature of WSXSEGxxPx
BANK#40-#43	MAIN-ROM 32KB(4BANK)

BANK#44-#47	Built-in software 32KB (4BANK) Each 16KB (2BANK) has a signature of WSXSEGxxPx
BANK#48-#55	DiskBIOS 64KB (8BANK)
BANK#56-#57	SUB-ROM 16KB (2BANK)
BANK#58-#61	Chinese character Driver 32KB (4BANK)
BANK#62-#63	MSX-MUSIC 16KB (2BANK)
BANK#64-#127	MSX-JE conversion dictionary 512KB (64BANK)
BANK#128-#131	MSX-JE for conversion dictionaries SRAM 32KB (4BANK)
BANK#132-#159	Disconnected 224KB (28BANK)
BANK#160-#191	ROM Disk Latter half?? 256KB (32BANK)
BANK#192-#255	Disconnected 512KB (64BANK)
BANK#256-#319	ROM Disk first half 512KB (64BANK)
BANK#320-#383	Disconnected 512KB (64BANK)
BANK#384-#447	MAIN-RAM 512KB (64BANK)
BANK#448-#511	MAIN-RAM 512KB (64BANK)、BANK#384-#447mirror

In turboR's primary mapper RAM, 4 segments are reserved as the BIOS copy area in order from the larger segment # value. For the primary mapper slot of SLOT#3-0, there is a feature that its 4 segments are write-protected in R800-DRAM mode. This probably protects the ROM cartridge software or memory mapper applications for MSX-DOS1 from accidentally overwriting the BIOS copy.

However, "BANK# where the contents of the primary mapper RAM appear" appearing in SLOT#3-3 has the feature that the entire area including the BIOS copy area can be written.

4.7. Pana amusement cartridge

Two types of PAC and FM-PAC released by Panasonic were on sale. Equipped with an 8KB battery-backed SRAM for saving game data, but it is disabled by default, and the SRAM appears when the prescribed processing shown in Table 4.7-1 is performed. I'm here.

Table 4.7-1. PAC SRAM switching

Prescribed processing	Effect
Write 5FFEh To 4Dh Write 5FFFh To 69h	Appear 4000h-5FFFh To SRAM
To 5FFEh or 5FFFh other than above	Disconnect SRAM at 4000h-5FFFh

8 KB is divided into 1 KB units To 8 areas To, and icons indicating which of these 8 areas are used are printed on the boxes of PAC compatible game software. There is no management information for these eight areas To, and each game maker uses them freely, so there is no way to determine which area is in use from the program. It was up to the user to overwrite the same area.

In the case of FM-PAC, the MSX-MUSIC ROM appears when the SRAM is disconnected, so its existence can be confirmed by detecting the signature "PAC2OPLL" from 4018h to To.

In the case of PAC, if the SRAM is disconnected, it behaves like an unconnected state, so even if it is read, a value with meaning cannot be obtained. FFh is generally returned no matter where you read it. When detecting, after confirming that there is no 4000h, 4001h To 41h, 42h and that it is not RAM, try writing 5FFEh, 5FFFh To 4Dh, 69h, and confirm that 4000h-5FFDh has changed to RAMTo is OK. You can check whether it is RAM or not with the following code, assuming that the address you want to check is set to HL To.

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

The original value is read from the A register To, all bits are reversed, and then To is written back. If it matches the written value, it is RAM, and if it does not match, it is not RAM (ROM, unconnected, etc.). In the case of RAM, it would be a problem to destroy Tovalue, so at the end To is reversed all bits to restore the original valueTo and write it back. Since the CPL and LD instructions do not change the Z flag, you can see the value of the Z flag determined by the CP instruction To. If the To Z flag is set after executing this process, it means that it is RAM, and if it is not set, it is not RAM.

5. ROM cartridge operation

When the power is turned on, MSX first boots from MAIN-ROM in SLOT#0 or SLOT#0-0. The BIOS program on the MAIN-ROM first checks the RAM installation status of Page2 and Page3, and searches for ROM for all slots with RAM appearing on Page2 and Page3. ROM search is performed in order of SLOT#0→SLOT#1→SLOT#2→SLOT#3. The expansion slots are SLOT#X-0 → SLOT#X-1 → SLOT#X-2 → SLOT#X-3.

In MSX, it is a rule to write the ROM header at the beginning of the ROM cartridge. If you write the appropriate ROM headers, the BIOS that interprets them will call the necessary programs in the ROM cartridge. The ROM header is supposed to be placed at the beginning of Page1 (4000h~) or at the beginning of Page2 (8000h~). It is better to use the beginning of Page1 (4000h~) for all machine language programs, and the beginning of Page2 (8000h~) to ROMize programs created in MSX-BASIC. Page1→Page2 is searched in order. ROM headers are summarized in Table 5.1.

Table 5.1. ROM header

Address	Size (byte)	Name
+0000h	2	ID
+0002h	2	INIT
+0004h	2	STATEMENT
+0006h	2	DEVICE
+0008h	2	TEXT
+000Ah	6	RESERVED

The ID is a signature that indicates that the ROM is out. Write 41h, 42h ("AB").

INIT writes the address of the initialization routine. If you do not initialize, please set 0000h To. On the other hand, for game cartridges, etc., please write the Address here, which is the entry point of the To game.

STATEMENT writes the address of the extended routine when extending the CALL instruction of MSX-BASIC. If you do not want to extend the CALL instruction, please set it to 0000h.

DEVICE writes the Address of the device extension routine. If you don't need it, please set it to 0000h.

TEXT specifies the storage address of the program written in MSX-BASIC that is in the cartridge. If the cartridge does not have MSX-BASIC program written, set to 0000h.

RESERVED is an area reserved for future expansion. Please fill it with 0.

5.1. Create a 16KB ROM cartridge program

Game software that does not use a disk generally starts from the address written in INIT. An example of a simple ROM program is shown below.

ROM_sample001.ASM

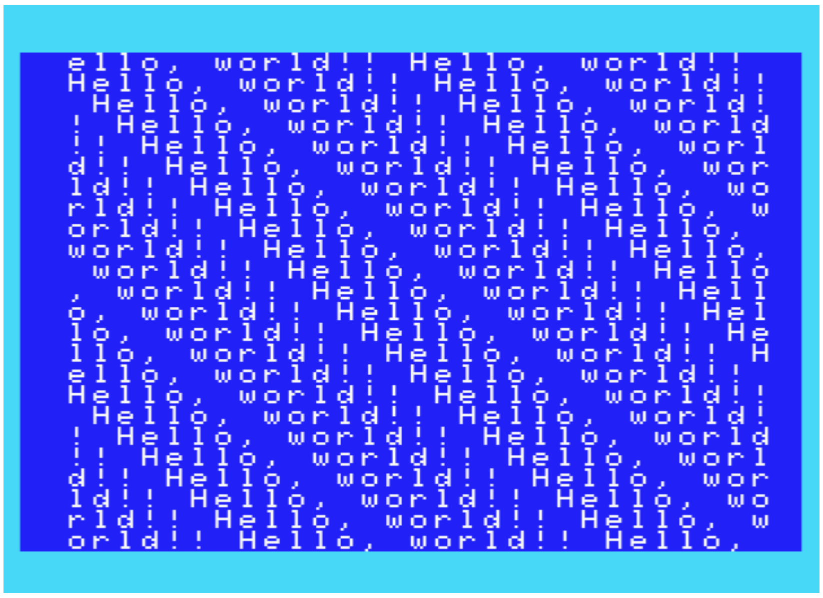
```
; ==============================================================================
;       ROM_sample001.ASM
; ------------------------------------------------------------------------------
;       Jan./31/2020 HRA!
; ==============================================================================

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             z
                call            chput
                jp              puts

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

                align           16384
```

Pass it to the assembler ZMA and assemble it with the following command.

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

After assembling, a 16KB ROM image file named ROM_sample001.ROM will be created. You can start it with an emulator, so please try it. If it is displayed as shown in Figure 5.1, it works as expected.



```
ello, world!! Hello, world!!
Hello, world!! Hello, world!!
 Hello, world!! Hello, world!
! Hello, world!! Hello, world
!! Hello, world!! Hello, worl
d!! Hello, world!! Hello, wor
ld!! Hello, world!! Hello, wo
rld!! Hello, world!! Hello, w
orld!! Hello, world!! Hello,
world!! Hello, world!! Hello,
 world!! Hello, world!! Hello
, world!! Hello, world!! Hell
o, world!! Hello, world!! Hel
lo, world!! Hello, world!! He
llo, world!! Hello, world!! H
ello, world!! Hello, world!!
Hello, world!! Hello, world!!
 Hello, world!! Hello, world!
! Hello, world!! Hello, world
!! Hello, world!! Hello, worl
d!! Hello, world!! Hello, wor
ld!! Hello, world!! Hello, wo
rld!! Hello, world!! Hello, w
orld!! Hello, world!! Hello,
```

Figure 5.1. Execution image of ROM_sapmle001.ROM

Figure 5.1. Execution image of ROM_sapmle001.ROM

```
            org                 0x4000
```

ZMA assembles the source code from top to bottom and writes it to the output file. At that time, it is assembled while internally calculating "what address the Z80 recognizes as the code." for calculating the absolute address of the label. org 0x4000 tells ZMA that this next line is at 0x4000.

A 16KB ROM should have 14-bit signals from A13 to A0 as address signal lines. Connect this to A13 to A0 of the cartridge slot as it is, A15 and A14 are not connected, /SLTSL of the cartridge slot is connected to /CS of the ROM, /RD of the cartridge slot is connected to /OE of the ROM. When making a Z80, the contents of this ROM appear in all of Page0, Page1, Page2, and Page3. Among these, Page1 is the one that the BIOS checks first, so place it in Page1. Make the program assuming it is written. Since the beginning of Page1 is 0x4000, it is set to org 0x4000.

```
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
```

I mentioned earlier that the ROM header must be written to 0x4000~0x400F. This is its ROM header.

Rom_header_id corresponds to the ID in Table 5.1. This is always the case with any ROM cartridge.

It is "AB". It's a mark to tell the BIOS "here's the ROM header".

Rom_header_init corresponds to INIT in Table 5.1. Enter the entry point address here enter. It's labeled entry_point in this program.

Others are not used this time, so fill them with 0.

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

Label entry_point is the entry point for this program. Stackpoint first and foremost initialize data. For normal programs, the value stored in himem should be used. himem contains the lowest address of the BIOS work area.

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

The subroutine puts will appear next, but if you specify the start address of the character string in HL and call it, a routine that displays that string (like her PRINT statement in MSX-BASIC). This time, as shown in Figure 5.1, so the label display_message is "Hello, world!!" is stored.

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

This is the contents of puts. The BIOS has a routine called CHPUT that controls the operation that, when written in MSX-BASIC, it works like PRINT CHR$(A); A here is the A register hey. With puts, the contents of the address indicated by the HL register are read into the A register, checked for 0, and if so, return, if not 0, use CHPUT to print one character, and repeat.

CHPUT preserves the contents of all registers, so the HL register is preserved.

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

The label display_message assigned to HL at main_loop is here.

In ZMA, the ds directive is "Define String". ASCII code for each letter in Hello, world!! is placed as is. The db pseudo-instruction uses 0 as a terminator. The terminator is what the subroutine puts from earlier interprets.

The terminator is what the subroutine puts from earlier interprets.

```
            align       16384
```

Finally there is the align instruction. ROM_sample001.ASM is a small program less than 16KB. This is a pseudo-instruction that adds padding to make it 16KB.

Just as CHPUT could be used earlier, when ROM INIT is called, Page0 is MAIN-ROM (BIOS), Page 1 is the ROM of the ROM cartridge, Page 3 is the BIOS work area, etc.

You can see that it is a RAM that Page2 differs depending on the model. For models with 16KB of RAM, this is SLOT#0 or SLOT#0-0 of SLOT#0/#0-0. The information placed on Page2 also differs depending on the model, so it may be unconnected or some built-in software. It could be ROM or whatever. It's not RAM anyway.

On models with 32KB or more of RAM, this is the RAM slot for Page2.

It is possible to determine whether the RAM is 32KB or more or the RAM is 16KB or less depending on whether Page2 is RAM or not. I don't think there are many software that bother to check it, but it works with 16KB, and it's different with 32KB. I think that it is necessary to judge when it behaves in a different way.

Furthermore, models with 8KB of RAM, such as the CASIO PV-7, have RAM only in E000h-FFFFh for Page3 does not exist. C000h-DFFFh cannot be written.

As a method to check whether it is RAM or not, read the value from the address you want to check and save it in a register. Same write the inverted value to the address, read it again, and if the inverted value can be read, the RAM is read. If not, it is not RAM (such as ROM or unconnected), and if it is RAM, it writes back the saved value. There is a way to The BIOS seems to do the same when checking for the presence of RAM. Note that the RAM SLOT#s for Page2 and Page3 are not necessarily the same. A program that assumes that is dangerous.

Memory Mapper compatible RAM must have RAM in all Pages 0 to 3 of that slot. For models with memory mapper compatible RAM, the RAM that appears on Page2 and Page3 is the same. It is SLOT#. The MSX1 and some MSX2 do not have RAM that supports memory mapper. Please be careful. If it is a cartridge for MSX2+ or later, it appears on Page2 and Page3 so it may not be a problem to assume that the RAM is the same SLOT#.

Stores RAM SLOT#s corresponding to Page0 to Page3 as a work area for the disk BIOS. There are areas from RAMAD0 to RAMAD3. However, when the ROM cartridge starts so this value is not stored yet. The disk BIOS itself has the same implementation as the ROM cartridge. Yes, most models with a built-in disc are equipped with SLOT#3-X. Disk BIOS initialization since it is before the routine is called, it is not stored. On the other hand, put the external drive in SLOT#1, it may be stored if the ROM cartridge you made is installed in SLOT#2. for that reason, SP initialization refers to HIMEM and other modules that may be present, such as disk BIOS it's safe to avoid destroying your work area. Remove unnecessary devices when playing games, etc. The promise of "please" is a preventive measure to prevent unexpected accidents due to such special cases.

5.2. Create a Mega ROM Cartridge Program

Here, we will create a program that runs on an ASCII-8K mega ROM.

The BIOS code in the MAIN-ROM in Page0 will switch Page2/Page3 to RAM and then go to Page1 in each slot. INIT of the mega ROM cartridge is called when searching for the slot of the mega ROM cartridge. At that point, Page2 is still RAM.

Booting from INIT is the same as 5.1. Creating a 16KB ROM cartridge program, but BANK#0 appears in both "Mega ROM BANK0 and BANK1" on Page1 vinegar. His BANK#0 appears in BANK2 and BANK3 in Page2 of the MegaROM slot, but Page2 in the MegaROM slot does not appear in the Z80 memory space. It is necessary to program with this point in mind.

5.1. Making a 16KB ROM cartridge program is a big difference.

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
; ==============================================================================
; ==============================================================================

                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
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
```

Now, I would like to explain in order, but I will omit the explanation for the same part as ROM_sample001.ASM. Let me do it.

```
main_loop:
               ; Put message (BANK#0 on BANK0)
```

```
            ld          hl, display_message1
            call        puts
```

Main_loop has changed greatly, but the first message display is the same as ROM_sample001.ASM.

```
            ; 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
```

Next, bank1_sel is declared as 0x6800 near the top of the program. In other words, it will be the address of the bank select register for his BANK1 in ASCII-8K. Since you're writing a 1 here, BANK1 will switch to his BANK#1. This image is shown in Figure 5.2-1.

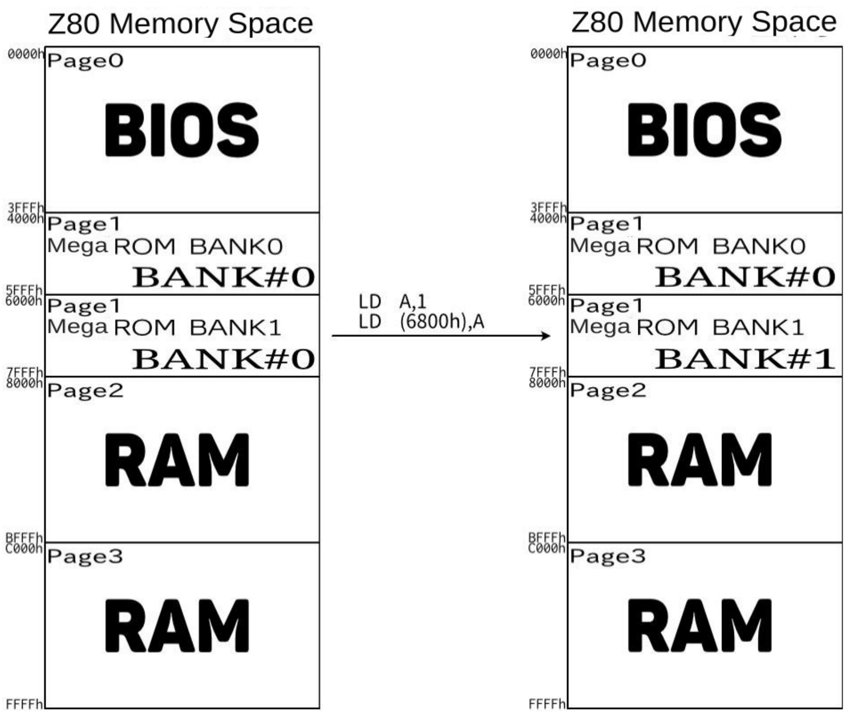


Figure 5.2-1. Image of BANK1 switching to BANK#1

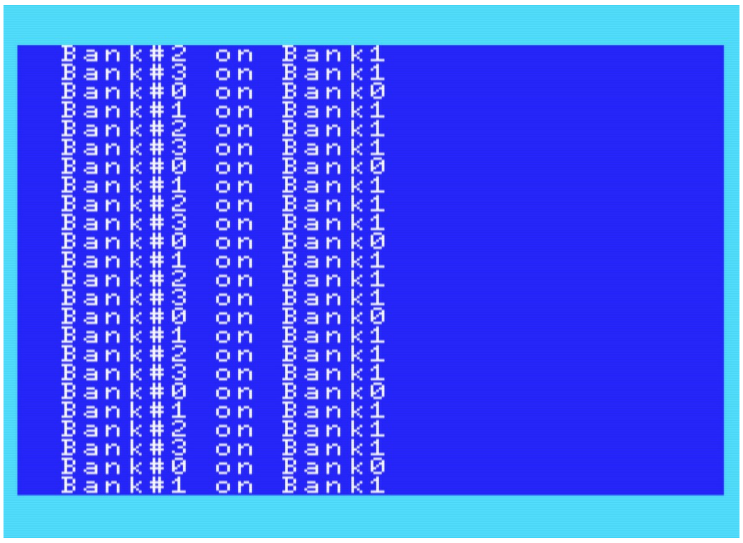
At startup, BANK#0 appears in both BANK0 and BANK1 areas, as shown in the memory map on the left side of Figure 5.2-1. Mega ROM address 6800h is the bank selection register for BANK1, but writing 1 here switches BANK1 to her BANK#1.

display_message2 will be the address of the string on her BANK #1 that appeared in BANK1. I am seeing this in 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
```

Similarly, for the bank selection register of BANK1, write 2 here to switch BANK1 to BANK#2 and display display_message3 in BANK#2. It then writes 3 to switch BANK1 to her BANK#3 and displays display_message4 in BANK#3.

jp main_loop loops back up and repeats. An image of this operation is shown in Figure 5.2-2.



```
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
Bank#2 on Bank1
Bank#3 on Bank1
Bank#0 on Bank0
Bank#1 on Bank1
```

Figure 5.2-2. Operation image of ROM_sample002.ASM

5.3. Detect own slot

If you program a 2KB ROM or even a mega ROM, Page2 will also want its own ROM to appear. To do that, it needs to detect the slot # of its own ROM. Here's how to do that.

Unfortunately, the BIOS doesn't provide a routine to get the current slot #, so unless you have a program to find out what slot you're in, you'll never know.

Below is a program that obtains the slot# in which his ROM cartridge is installed from the program code on the ROM cartridge.

ROM_sample003.ASM

```
; ==============================================================================
;      ROM_sample003.ASM
; ------------------------------------------------------------------------------
;      Feb./2/2020 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]

             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
             rrca
             rrca
             and         a, 3
             add         a, '0'
             call        chput

finish:
             jp          finish

; ==============================================================================
;      get_page1_slot
;      input)
;             HL .... address of target string (ASCII-Z)
; ==============================================================================
             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         z
             call        chput
             jp          puts
             endscope

; ================================================================================
;      DATA
; ================================================================================
display_message1:
             ds          "SLOT#"
             db          0

             align       8192
```

get_page1_slot is the routine to get the slot # of Page1. Please note that if you run it with Page3, it will run out of control. It is premised to run on either Page 0 to 2. Now let's talk about get_page1_slot.

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

The in instruction reads the contents of the basic slot selection register. Figure 5.3-1 shows the bitmap of the basic slot selection register. The yellow bit in Figure 5.3-1 indicates the slot # of Page1. Since slot # of other Page is not needed, it is cleared to 0 with and a, 0x0C. By shifting this to the right by 2 bits, the values 0 to 3 indicating SLOT#0 to SLOT#3 are stored in the A register.

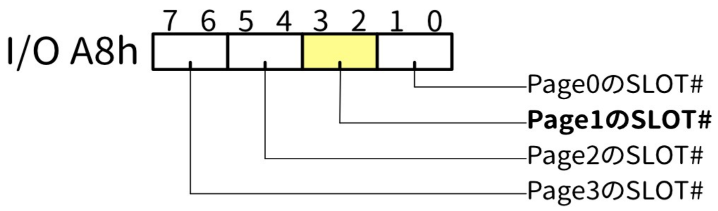


Figure 5.3-1. Basic Slot Selection Register

```
           push       af

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

Next, save the requested slot # on the stack with push af. It is also saved in the B register. Create an address corresponding to slot # of Page1 in EXPTBL. EXPTBL is an array, but the top address is FCC1h. HL=FCC1h+A is implemented. Since the A register is slot # from 0 to 3, overflow does not occur at the add instruction.

At LD A,[HL], a flag indicating the presence or absence of expansion slots is stored in the A register.

```
           and        a, 0x80
           jp         z, skip1
```

Look at the MSB at and a, 0x80. A = 0x80 if "slot is expanded" becomes "Zf = 0", and the next jp instruction is skipped. If "the slot is not expanded", A = 0x00, so "Zf = 1" and jump to skip1.

```
           ld         a, b
           rrca
           rrca
           ld         b, a
```

"If not extended" jumped to skip1, so only "if extended" is executed here. Prepare to read the "Expansion Slot Select Register".

Rotate the slot # stored in the B register 2 bits to the right and move it to the position of bit7, bit6. Since the "extension slot selection register" exists in Page3 of that slot, it is the position corresponding to Page3 in the basic slot selection register.

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

Use the in instruction to read the current "basic slot selection register value". After backing up this in the C register, clear the Page3 slot # to 0 with and a, 0x3F, and OR the values that moved the Page1 slot # saved in the B register earlier to bit7 and bit6.

Since Page3 also has a work area that is used during interrupt processing, interrupts must be disabled when Page3 is switched. It is out after di.

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

While Page3 is in the same slot as Page1, it reads address 0xFFFF. This becomes the "extended slot select register value" for the desired slot. However, please note that all bits are inverted. Returns the value of the basic slot selection register, as it has been read. I've returned it, so I'll allow interrupts.

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

Inverts the value read from the expansion slot selection register and converts it to a usable value. Creates the upper 6 bits of the slot # used by ENASLT.

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

If the slot is not expanded, jump to skip1 with A=0. In other words, the basic slot # put on the stack goes into A as it is.

If the slot is extended, the A register stores the upper 6 bits of "slot # used by ENASLT". The value that comes into the B register with pop bc is the basic slot # of 0 to 3, and the lower 2 bits are filled with values. By calling these, you can get the slot # used by ENASLT.

5.4. Page2/Page3 slot detection

The method is to detect the RAM slots appearing on Page2 and Page3 in the same way that the ROM appearing on Page1 was detected in 5.3. Detecting Own Slots.(*Page2 is not RAM for models with RAM8KB or RAM16KB). The detection program is shown below. The detection program is shown below:

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
; ================================================================================
                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

; ===============================================================================
;      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
                rrca
                rrca
                rrca
                rrca
                push         af

                ; Page2 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, 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
; ================================================================================
                scope get_page3_slot
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

; ==============================================================================
;      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)
; ==============================================================================
                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, skip1

                ld           a, '-'
                call         chput

                ld           a, b
                rrca
                rrca
                and          a, 3
                add          a, '0'
                call         chput
skip1:
                ld           hl, display_crlf
                call         puts
                ret
                endscope

; ==============================================================================
;      DATA
; ==============================================================================
display_message_page1:
                ds           "PAGE1 SLOT#"
                db           0
display_message_page2:
```

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

            align       8192
```

The details of the processing are almost the same as 5.3. Own slot detection, so the explanation is omitted.

The RAM for age2 and Page3 are not always in the same slot, so they must be obtained separately.

Special To, Page2 is often used while switching between "RAM slot" and "MegaROM slot" as needed.

5.5. Page0/Page3 slot switching

For slot switching, there is a convenient routine called ENASLT (0024h) in BIOS, but you cannot use this to switch Page0. Since ENASLT itself exists in his Page0, when ENASLT is called with Page0 switching settings, it loses itself and goes out of control.

Page3 can be switched by ENASLT, but please note that if ENASLT is called while the stack pointer is on Page3, it will run out of control. Since ENASLT uses the stack, it must be called after moving the stack pointer to a Page other than Page3. Also, since this is an area that contains the BIOS work area, it is necessary to pay attention to the timing of EI. There is also a hook in the BIOS work area that is called when an interrupt occurs, so if you switch to EI without proper care, it will go out of control when an interrupt occurs.

The maximum capacity is 64KB with a ROM cartridge that is not a mega ROM, but in that case you want to switch Page0 or Page3 to the slot of the ROM cartridge.

Below is a sample program that searches for RAM slots by switching Page0 slots, and then searches for RAM slots by switching Page3 slots.

ROM_sample005.ASM

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

enaslt       = 0x0024
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
             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
             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
             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
; ================================================================================
             scope get_page3_slot
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
; ==============================================================================
                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
                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                    ; SSBBAASS : 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; CCBBAA00 : 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)
; ============================================================================
             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
             rrca
             rrca
             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 of EXPTBL
             ld           de, slot_info       ; SLOT#0-0 of slot_info
             ld           b, 0                ; SLOT#0 display start from
slot_loop:
             ld           a, [hl]             ; EXPTBL read
             inc          hl                  ; Prepare for the next SLOT
             push         hl
             and          a, 0x80             ; Clear except expansion slot flag
             or           a, b                ; E0000000
             ld           b, a                ; Backup to B register
exp_slot_loop:
             ld           a, [de]             ; slot_info read
             inc          de                  ; Prepare for your next SLOT
             or           a, a                ; If 0, the slot does not exist and is
ignored
             jp           z, skip1
             ld           c, a                ; Backup slot_info in C register

             ld           a, b                ; Restore SLOT# to A register
             push         de
             push         bc
             push         af
             ld           hl, message_slot    ; Print "SLOT#"
             call         puts
             pop          af
             call         put_slot_no         ; Print SLOT#
             pop          bc
             push         bc
             ld           hl, message_ram
             dec          c                   ; Displays "RAM" if slot_info is 1, "Non
RAM" if 2
             jp           z, skip2
             ld           hl, message_non_ram
skip2:
             call         puts
             pop          bc
             pop          de
skip1:
             ld           a, b                ; Restore SLOT# to A register
             add          a, 0x04             ; To next expansion slot
             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
```

I will omit the description of the parts common to the samples that have appeared so far.

The routine to switch slots for Page0 is local_enaslt0. This routine switches Page0 to "slot# specified by A register".

```
          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
```

First, save the slot # in the B register.

”and a, 0x83” clears bit3 and bit2, which specify the expansion slot number, to 0. At the same time, reflect bit7, which indicates the presence or absence of expansion, in Sf (1 if negative, 0 if positive or zero).

For two's complement representation, bit7 = 1 for negative numbers and bit7 = 0 for positive numbers or zero. Therefore, you can see the presence or absence of expansion slots by looking at Sf. If not an expanded slot, jump to not_expanded.

”xor a, 0x80” clears bit7 to 0. and a, 0x7F is also OK.

6. Other supplementary matters

This chapter describes supplementary items that were not listed up to Chapter 5.

6.1. EXPTBL when using version upgrade adapter

There was a product called "NEOS MA-20 MSX version upgrade adapter" (hereafter, version upgrade adapter) that upgrades MSX1 to MSX2 equivalent. By attaching the MAIN-ROM/SUB-ROM/V9938/64KB RAM cartridge installed in MSX2 to MSX1, you can upgrade to MSX2 equivalent. What is of concern here is EXPTBL(FCC1h).

FCC1h is a work area that indicates whether or not SLOT#0 is expanded, but it is also a work area that also serves as a "MAIN-ROM slot". Since the MAIN-ROM exists in SLOT#0 on a normal main unit, the value of FCC1h itself can be used as it is for specifying slots such as ENASLT. ROM will be installed. Consider a case where the MSX2 version MAIN-ROM is installed in expansion slot number 0, say his SLOT#1-0, while his SLOT#0 on the main unit side is not expanded. The MSB of FCC1h indicates whether or not SLOT#0 is extended, so it becomes 0. I can't tell the difference. Since the MSB of FCC2h indicates the presence or absence of expansion of SLOT#1, it will match if the program is organized to determine which one, but (probably even when the version upgrade adapter comes out) it is on the market Many software regard FCC1h as slot # of MAIN-ROM that can be used as it is for slot designation of ENASLT.

Therefore, the upgrade adapter behaves to set the MSB of FCC1h to 1 regardless of whether SLOT#0 is extended or not.

For example, when the MSX2 version MAIN-ROM cartridge is installed in SLOT#1, the MAIN-ROM overwrites the value of FCC1h with the binary value of 10000001. In fact, these values generally work without problems.

Consider the case where SLOT#0 and SLOT#1 are not expanded and the MSX2 version MAIN-ROM is installed in SLOT#1. Since 81h is stored in FCC1h, "the program that wants to switch to MAIN-ROM" puts the value of FCC1h into the A register and he calls ENASLT. ENASLT then writes to 0FFFFh after switching to SLOT#1 using I/O A8h. If it is an expansion slot, 0FFFFh has an expansion slot register, but since his SLOT #1 is not actually expanded, writing to this 0FFFFh disappears. This is because writing is done to 0FFFFh of the

MSX2 version MAIN-ROM cartridge of SLOT#1. The MAIN-ROM cartridge has only ROMs on Page0 and Page1, and Page2 and Page3 are not connected. So a write access to 0FFFFh will disappear without writing anything.

Regarding the behavior of the version upgrade adapter, we confirmed it with the cooperation of Mr. Parup, who owns the real thing.

6.2. Do not write the expansion slot selection register unconditionally.

If you cannot use ENASLT for some reason, you will have to create your own slot switching routine. In such a case, you might think that writing to the expansion slot selection register will make the process a little easier, regardless of whether the slot is expanded or not. In fact, as explained in 6.1, EXPTBL when using the version upgrade adapter, the version upgrade adapter changes the MSB of EXPTBL (FCC1h) to 1 regardless of whether SLOT#0 is expanded or not. However, since this was SLOT#0, it is safe, and other slots are NG. If you write to 0FFFFh in a basic slot that is not connected to an expansion slot, it simply goes to that address. If it was in RAM, it will be written to RAM. You may think, "I don't use RAM at address 0FFFFh, so it's okay, right?", but a problem arises when a type of RAM that allows you to select the appearance address of a segment or bank, such as a memory mapper compatible RAM, is connected.

For example, problems arise when there are multiple memory-mapper-enabled RAMs, at least one of which is connected to the primary slot. Here are some concrete examples. There is a music player called MGSP v2.1.xDOS version (hereafter MGSP) that I distribute, which puts kanji fonts on memory mapper compatible RAM. Kanji fonts can also be placed in the secondary mapper. MGSP also puts his MGSDRV.COM into memory mapper enabled RAM. This is only for primary mappers.

When using MGSP with an unmodified FS-A1ST, an additional RAM cartridge (supporting memory mapper) is required. The 256KB on the main unit side is the primary mapper, and the additional RAM cartridge is the secondary mapper. Figure 7.2-1 shows the usage of memory mapper when FS-A1ST is started. The additional RAM cartridge is written on the assumption that it is installed in 64KB/SLOT#1.

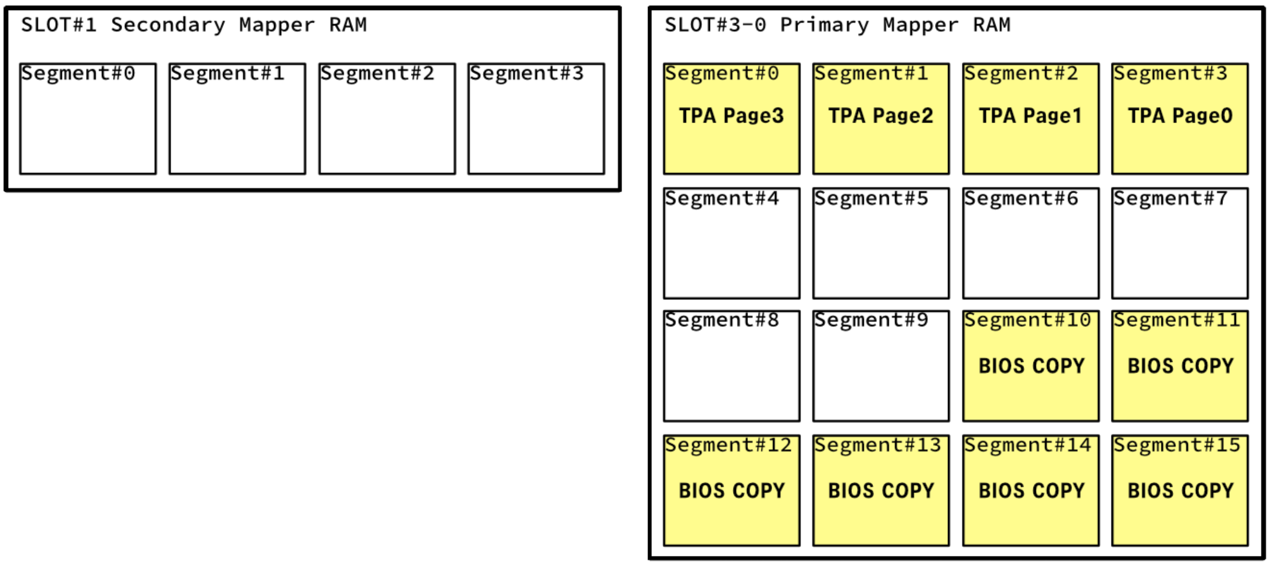


Figure 6.2-1. Memory mapper usage at FS-A1ST startup

The part painted yellow in Figure 6.2-1 is the used Segment#. More than half of the FS-A1ST is already used.

Now suppose that some resident program used two segments. For example, assume that MGSDRV.COM is made resident. Then the usage status changes as shown in Figure 6.2-2. Resident programs use system segments. The memory mapper's system segment is allocated starting with the highest segment #.

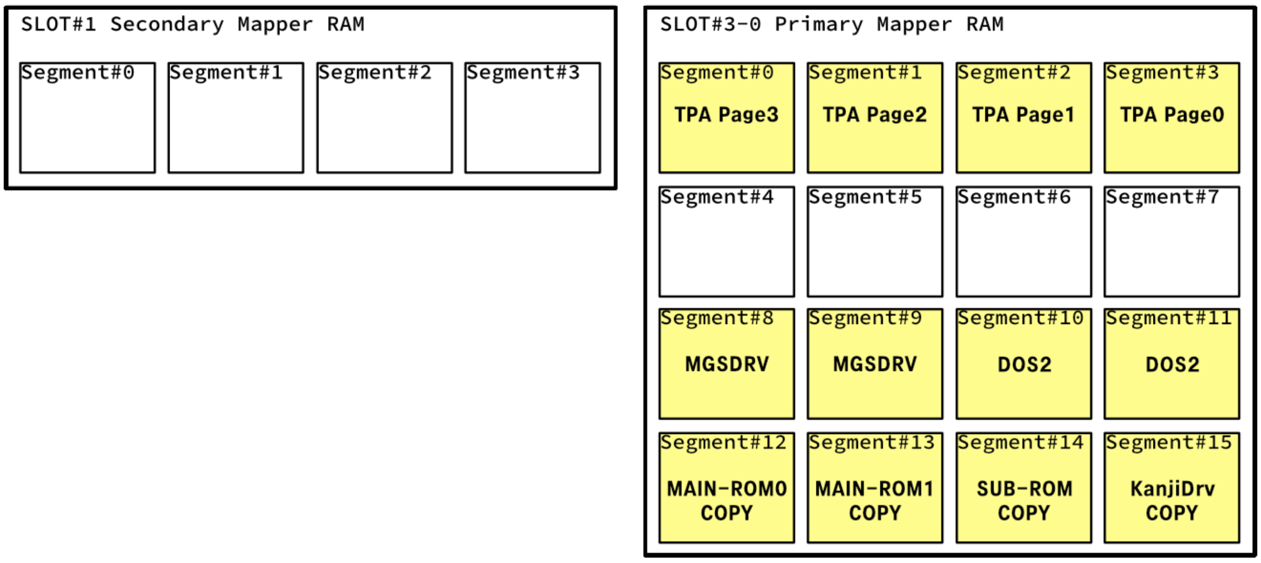


Figure 6.2-2. Memory mapper usage after MGSDRV residency

If MGSP is started from this state, MGSP cannot use the resident MGSDRV, so the memory for MGSDRV is newly used. Use 2 segments for this. Since MGSP is used for user segments, it is assigned from the smallest segment #. In addition, we use 4 segments for Kanji fonts, but the primary mapper has only 2 segments free. Kanji fonts also support secondary mappers, but they don't support allocation across multiple memory-mapper-enabled RAMs, so they are all relegated to secondary mappers. As a result, the usage situation is as shown in Figure 6.2-3.

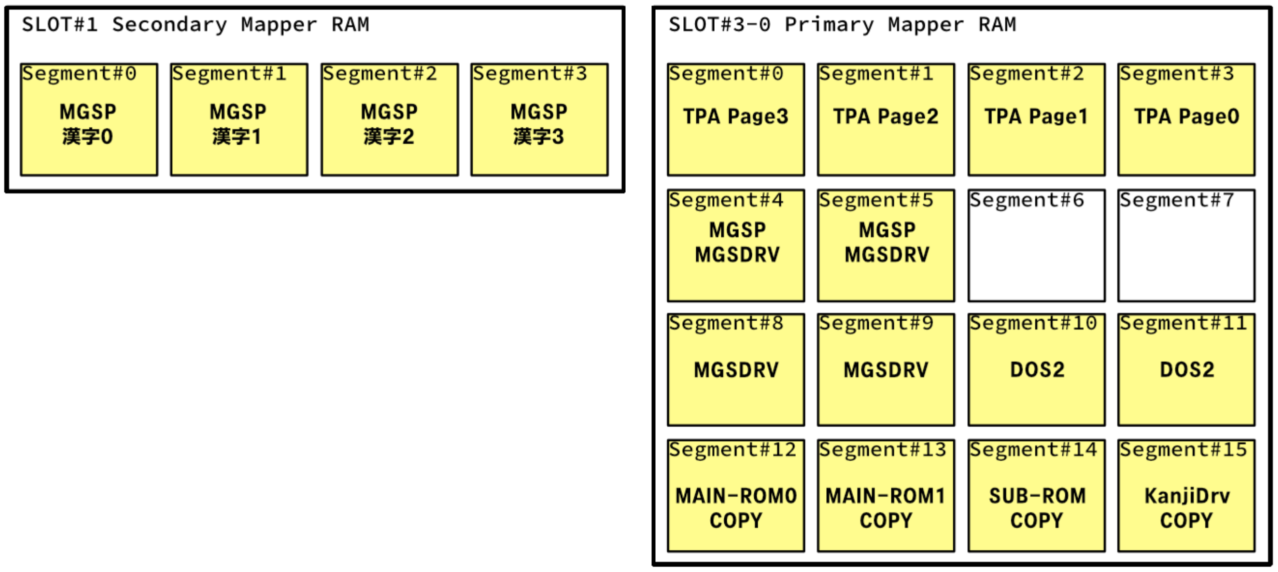


Figure 6.2-3. Memory allocation status during MGSP operation

Now let's turn our attention to the slots. Figure 6.2-4 shows the slot configuration when a memory mapper compatible RAM 64KB cartridge is installed in SLOT#1 of FS-A1ST.

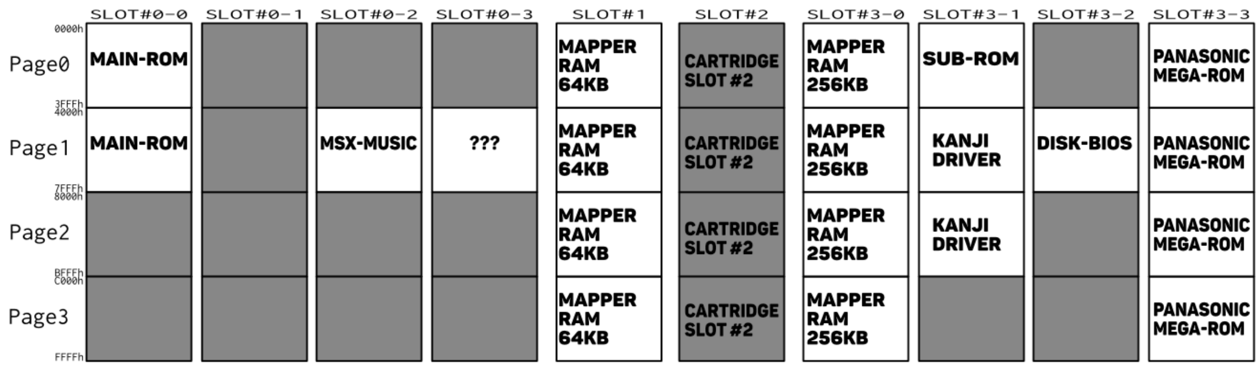


Figure 6.2-4. FS-A1ST + RAM64KB

MSX Memory Architecture

When MSX-DOS2 is started, Z80 memory space is as shown in Figure 6.2-5.

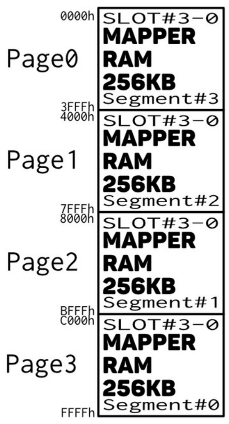


Figure 6.2-5. Z80 memory space when DOS2 is started

The Segment# that appears in each Segment of the memory mapper is common to all memory mappers, so at this point, the RAM64KB Segments of SLOT#1 are also Segment#3, Segment#2, Segment#1, Segment#0 in order from Page0. I'm here. SLOT #1 is not an expansion slot, and he is directly installed with 64KB of RAM.

MGSP stores Kanji fonts in RAM64KB of SLOT#1, so Page2 is switched to SLOT#1 to access Kanji fonts. Assuming that this "switch to SLOT#1" process was independently implemented within MGSP without using ENASLT, in its own slot switching routine, "regardless of whether or not the slot is expanded , write to 0FFFFh, where the expansion slot select register would reside"?

The "extension slot selection register" that appears at 0FFFFh is a register owned by the device called "extension slot". A device called an "expansion slot" accesses his 0FFFFh to its own expansion slot select register. On the other hand, accessing 0000h-FFFEh of itself propagates the access to the slot indicated by the extended slot selection register among the four slots hanging from itself. The expansion slot selection register is physically implemented on the expansion slot side.

Now, if we return to the above example of running MGSP with FS-A1ST + RAM64KB, if we write a value for the FFFFh of SLOT#1 that is not expanded, the value will be written from the beginning of segment#0 of the RAM64KB cartridge corresponding to the memory mapper that appears there at the position of +3FFFh.Since there is a kanji font there, the kanji font will be destroyed by 1byte.

As an example of "a case where writing to 0FFFFh unconditionally causes a problem regardless of the presence or absence of an expansion slot", I gave an example of combination with memory mapper compatible RAM. Depending on the situation, other combinations may also cause problems, so you should avoid writing to his 0FFFFh assuming expansion slot selection for slots that do not have expansion slots.

Appendix.

Appendix 1. Various sources used by the sample program

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
gspsiz      :=          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
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: Light pen (A=8~11) always returns 0 in
MSXturboR
gtpdl        :=          0x00de       ; note: MSXturboR always returns 0
tapion       :=          0x00e1       ; note: MSXturboR will always return Cy=1
(error)
tapin        :=          0x00e4       ; note: MSXturboR will always return Cy=1
(error)
tapiof       :=          0x00e7       ; note: MSXturboR will always return Cy=1
(error)
tapoon       :=          0x00ea       ; note: MSXturboR will always return Cy=1
(error)
tapout       :=          0x00ed       ; note: MSXturboR will always return Cy=1
(error)
tapoof       :=          0x00f0       ; note: MSXturboR will always return Cy=1
(error)
stmotr       :=          0x00f3       ; note: MSXturboR then return without doing
anything
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
sub_clrspr   :=          0x00f5
sub_calpat   :=          0x00f9
sub_calatr   :=          0x00fd
sub_gspsiz   :=          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 then the light pen (A=8~11)
always returns 0
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, Read from basic slot
wrprim       :=          0xF385       ; 7bytes, Write to basic slot
clprim       :=          0xF38C       ; 14bytes, basic slot call

cliksw       :=          0xF3DB       ; 1byte, Key click switch (0=OFF, others=ON)
csry         :=          0xF3DC       ; 1byte, the Y coordinate of the cursor
csrx         :=          0xF3DD       ; 1byte, the X coordinate of the cursor
cnsdfg       :=          0xF3DE       ; 1byte, Function key display switch (0=ON,
others=OFF)

rg0sav       :=          0xF3DF       ; 1byte, the R#0 value written to VDP
rg1sav       :=          0xF3E0       ; 1byte, the R#1 value written to VDP
rg2sav       :=          0xF3E1       ; 1byte, the R#2 value written to VDP
rg3sav       :=          0xF3E2       ; 1byte, the R#3 value written to VDP
rg4sav       :=          0xF3E3       ; 1byte, the R#4 value written to VDP
rg5sav       :=          0xF3E4       ; 1byte, the R#5 value written to VDP
rg6sav       :=          0xF3E5       ; 1byte, the R#6 value written to VDP
rg7sav       :=          0xF3E6       ; 1byte, the R#7 value written to VDP
```

```
statfl       :=           0xF3E7       ; 1byte, S#0 value read from VDP

fnkstr       :=           0xF87F       ; 16byte * 10, Read from basic slot

exbrsa       :=           0xFAF8       ; 1byte, Slot# of SUB-ROM

hokvld       :=           0xFB20

oldkey       :=           0xFBDA       ; 11bytes, Key matrix status (old)
newkey       :=           0xFBE5       ; 11bytes, Key matrix state (new)
keybuf       :=           0xFBF0       ; 40bytes, key buffer
linwrk       :=           0xFC18       ; 40bytes, Temporary storage used by screen
handlers
patwrk       :=           0xFC40       ; 8bytes, Temporary storage location used by the
pattern converter
bottom       :=           0xFC48       ; 2bytes, The first address of the installed
RAM. Normally 8000h for MSX2
himem        :=           0xFC4A       ; 2bytes, High address of available memory. Set
by <memory limit> in CLEAR statement
trptbl       :=           0xFC4C       ; 78bytes, Trap table used in interrupt
processing. One table is 3 bytes and the 1st byte is ON/OFF/STOP status. The remainder
is the branch destination text address

scrmod       :=           0xFCAF       ; 1byte, current screen mode number
oldscr       :=           0xFCB0       ; 1byte, Screen mode save area

exptbl       :=           0xFCC1       ; 4bytes, Presence or absence of expansion for
each slot
slttbl       :=           0xFCC5       ; 4bytes, Current selection status of expansion
slot selection register for each slot
sltatr       :=           0xFCC9       ; 64bytes, attributes for each slot
sltwrk       :=           0xFD09       ; 128bytes, Reserve a specific work area for
each slot
procnm       :=           0xFD89       ; 16bytes, extension statement, contains the
name of the extension device, ASCIIZ
device       :=           0xFD99       ; 1byte, Device identification for cartridges

h_timi       :=           0xFD9F       ; 5bytes, Hook when vertical sync interrupt
occurs
extbio       :=           0xFFCA

rg8sav       :=           0xFFE7
rg9sav       :=           0xFFE8
```

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 = characters entered
;               L = same as A
;       comment)
;               echoed to standard output
; ==============================================================================
D1F_CONIN                 := 0x01

; ==============================================================================
;       Console output
;       input)
;               C = D1F_CONOUT
;               E = characters to output
;       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
```

msxdos2.asm

```
; --------------------------------------------------------------------------------
;      FIB (File Information Block)
FIB_SIGNATURE       := 0         ; 1byte, Always 0xFF.
FIB_FILENAME        := 1         ; 13bytes, File name (ASCIIZ)
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_VERFY           := 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 (ASCIIZ) 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

; ------------------------------------------------------------------------------
;      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 (ASCIIZ) 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 (ASCIIZ) 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
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 (ASCIIZ)
;      output)
;             --
;      break)
;             all
;      comment)
;             print an ASCIIZ string to standard output
; ================================================================================
             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

```
; ================================================================================
;      MapperSupportRoutine's table offset (This area is reference only)
; ================================================================================
mmap_slot           := 0                ; 1byte, Mapper RAM slots#
mmap_total_seg      := 1                ; 1byte, total number of segments
mmap_free_seg       := 2                ; 1byte, Number of unused segments
mmap_sys_seg        := 3                ; 1byte, Number of segments allocated to the
system
mmap_user_seg       := 4                ; 1byte, Number of segments assigned to the user
mmap_reserved       := 5                ; 3byte, Reserved area

; ================================================================================
;      mmap_init
;      input)
;             --
;      output)
;             Zf ................. 1: No MemoryMapper support routines exist
;             [mmap_table_ptr] ... address of the mapper table
;      break)
;             all
; ================================================================================
             scope  mmap_init
mmap_init::
             ld          c, D2F_DOSVER      ; Check DOS version
             call        bdos
             or          a, a               ; A != 0 is not DOS
             jp          nz, mmap_error_exit
             ld          a, b
             cp          a, 2               ; If B < 2 then it is MSX-DOS1
             jp          c, mmap_error_exit

             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=MemoryMapperSupportRoutine ID, E=01h
             call        extbio
             or          a, a
             ret         z                  ; Error if mapper support routine does
not exist (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                                        ; Successful completion (Zf=0)
mmap_error_exit:
             xor          a, a                          ; Make Zf=1
             ret                                        ; End with error (Zf=1)
             endscope

; ==============================================================================
;      Revert slot configuration to TPA
; ==============================================================================
             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            ; Address of mapper table is stored
mapper_segment_p1:
             db           0            ; Segment of page1 at startup#
mapper_segment_p2:
             db           0            ; Segment of page2 at startup#

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
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
```

Caution!

We are not responsible for any damage caused by using the contents of this manual. Please use it under the responsibility of each person.

This document was created with reference to the following materials. Thanks to those who made the original material available.

【References】

Technical wiki plan

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

MSX Datapack wiki project

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

Gigamix Online Mega ROM Database

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

Megacon analysis data

http://www.big.or.jp/~saibara/msx/ese/mctr_anl-j.html

msx.org MegaROM Mappers

https://www.msx.org/wiki/MegaROM_Mappers

Konami Sound Cartridge - SCC+

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

Hiroyuki's Homepage SCC Laboratory

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

ASCAT home page Tech Guy full text

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

Special thanks to:

We were able to improve the degree of perfection thanks to various suggestions and cooperation from many people. I would like to take this opportunity to thank you. In particular, I am very grateful to Mr. Hidarikiki for

pointing out many things that I would not have noticed if I had just reviewed them myself.

February 19, 2021 HRA!