

MSX Memory Architecture

目次

1. はじめに.....	4
1.1. 用語.....	4
2. スロット.....	6
2.1. 基本スロット.....	6
2.2. 拡張スロット.....	9
2.3. スロット関連の BIOS ルーチン.....	11
RDSLT (000Ch).....	12
WRSLT (0014h).....	13
CALSLT (001Ch).....	14
ENASLT (0024h).....	15
CALLF (0030h).....	17
2.4. 基本スロットと拡張スロットの関係.....	18
3. メモリーマッパー対応 RAM.....	19
3.1. メモリーマッパーサポートルーチン.....	25
プライマリマッパーとセカンダリーマッパー.....	25
マッパーサポートルーチンの使い方.....	27
ALL_SEG.....	29
FRE_SEG.....	31
RD_SEG.....	32
WR_SEG.....	33
CAL_SEG.....	34
CALLS.....	35
PUT_PH.....	36
GET_PH.....	37
PUT_P0.....	38
GET_P0.....	38
PUT_P1.....	39
GET_P1.....	40
PUT_P2.....	40
GET_P2.....	41
PUT_P3.....	42
GET_P3.....	43
メモリーマッパー情報テーブル.....	43
4. メガ ROM.....	49
4.1. ASCII-8K タイプ.....	50

MSX Memory Architecture

4.2. ASCII-16K タイプ.....	53
4.3. Konami-8K タイプ.....	55
4.4. Konami SCC タイプ.....	56
4.5. Konami SCC-I タイプ.....	59
4.6. Panasonic タイプ.....	63
4.7. パナアミューズメントカートリッジ.....	68
5. ROM カートリッジの動作.....	69
5.1. 16KB の ROM カートリッジプログラムを作る.....	70
5.2. メガ ROM カートリッジプログラムを作る.....	76
5.3. 自身のスロットの検出.....	81
5.4. Page2/Page3 のスロット検出.....	87
5.5. Page0/Page3 のスロット切り替え.....	92
6. ハードウェア.....	102
7. その他の補足事項.....	103
7.1. バージョンアップアダプター利用時の EXPTBL.....	103
7.2. 拡張スロット選択レジスタは無条件に書いてはダメ.....	104
付録.....	109
付録 1. サンプルプログラムから利用している各種ソース.....	109

1. はじめに

本書は、MSX の ROM/RAM についてまとめたものです。MSX には様々な ROM/RAM があり、一カ所にまとめて記載されている資料を見かけないので、書いてみることにしました。

MSX 本体だけでも、基本スロット・拡張スロットによるメモリ空間の切り替えや、メモリーマッパー対応 RAM によるマッパーセグメント切り替え、MSX2+以降の Panasonic 機では Panasonic 固有のメガ ROM バンク切り替え等多岐にわたります。これに加え、カートリッジスロットに装着するメガ ROM (ASCII-8K, ASCII-16K, Konami-8K, Konami-SCC, Konami-SCC-I) やパナアミュージメントカートリッジについても記載しています。メジャーどころは一通り抑えたつもりです。マイナーなものに関しては、すでに入手困難だったり、制御のための情報が無かったりするため、省略させていただきます。

これから MSX 用のソフトウェアを作る方の参考になれば幸いです。

本書には、随所にアセンブラによるサンプルプログラムを掲載していますが、私(HRA!)が自作したアセンブラ ZMA 向けのプログラムになっています。Zilog ニーモニックと若干異なる部分がありますのでご注意下さい。違いについては、ZMA の取扱説明書を参照下さい。ZMA は下記サイトに配布しているフリーソフトです。

ZMA 配布サイト

<http://hraroom.s602.xrea.com/msx/zma/index.html>

1.1. 用語

MSX のメモリー関連（スロット、メモリーマッパー、メガ ROM）は、物理的な ROM や RAM のアドレス空間を 16KB 単位または 8KB 単位の領域に分けて管理しています。Z80 のメモリ空間に、それら「16KB 単位または 8KB 単位の領域」を出現させてアクセスする構造になっています。この「16KB 単位または 8KB 単位の領域」を出現させる Z80 メモリ空間上のアドレス範囲や、出現させる「16KB 単位または 8KB 単位の領域」そのものの呼び名がやや曖昧で、参照する資料によって固有の呼び名で記載されている場合があります。例えば、「16KB 単位または 8KB 単位の領域」のことをセグメント記載している資料と、バンクと記載している資料があることを確認しています。資料 A で

MSX Memory Architecture

は、メモリーマップの「16KB 単位の領域」も、メガ ROM の「16KB 単位または 8KB 単位の領域」も、セグメントと記載されていました。一方で、資料 B では、メモリーマップの「16KB 単位の領域」はセグメント、メガ ROM の「16KB 単位または 8KB 単位の領域」はバンク。資料 C では、メモリーマップの「16KB 単位の領域」も、メガ ROM の「16KB 単位または 8KB 単位の領域」もバンクと記載。このように、一意に名前が決まっていません。

誤解無く伝えるために、本資料では下記の呼び名に統一して記載します。他の資料を参照する際には、別の呼び名になっている可能性も考慮すると混乱を避けられるかもしれません。

大分類	概要	呼び名
スロット	指定のスロットが出現する 16KB 単位のアドレス領域	PageN
スロット	Z80 から見えるメモリ空間の Page0～3 に出現する基本ス ロット#を指定するレジスタ	基本スロット選択レジスタ
スロット	着目している基本スロットの Page0～3 に出現する拡張ス ロット#を指定するレジスタ	拡張スロット選択レジスタ
メモリーマップ	メモリーマップの切り替え単 位(16KB)	Segment#N
メガ ROM	メガ ROM が装着されているス ロットのアドレスを 8KB 単位ま たは 16KB 単位で区切ったアド レス領域	BANKN
メガ ROM	上記バンクに出現する ROM の 一部分	BANK#N

例えば、「BANK0 に BANK#23 を出現させる」と書いてあれば、メガ ROM の話となります。

「Page1 に Segment#31 を出現させる」と書いてあれば、メモリーマップの話となります。

2. スロット

MSX の CPU である Z80 は、64KB のメモリ空間を持っていますが、BIOS や内蔵ソフト等の ROM を同時に見えるようにしてしまうとすぐに手狭になってしまいます。そこで MSX ではスロットという考え方を導入することにより、256KB までの空間が扱えるようになっています。この「スロットという考え方」を基本スロットと呼びます。

基本スロット 1 つを、さらに「4 つのスロット」に拡張することが出来ます。この 4 つのスロットに拡張したスロットを拡張スロットと呼びます。すべての基本スロットを拡張すれば、合計 1MB の空間を扱えるようになります。最初の MSX が発売された時期から考えると広大な空間が扱えるようになっていて、当時としては先進的だったのではないかと思います。以下に詳細を説明します。

2.1. 基本スロット

図 2-1. に示すように、64KB の空間は Page と呼ばれる 16KB の領域に分けられています。

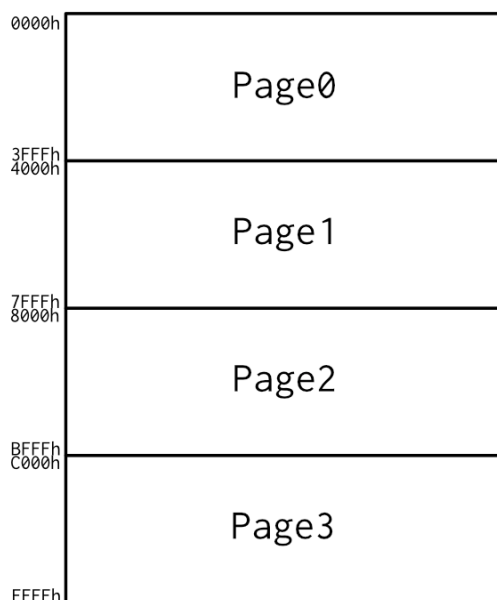


図 2-1. Z80 メモリ空間に見える Page の領域区分

MSX Memory Architecture

この各 Page それぞれに 4 つの基本スロットが存在しており、基本スロット選択レジスタによって各 Page それぞれの基本スロットを出現させるか選択するようになっていきます。そのイメージを図 2-2.に示します。

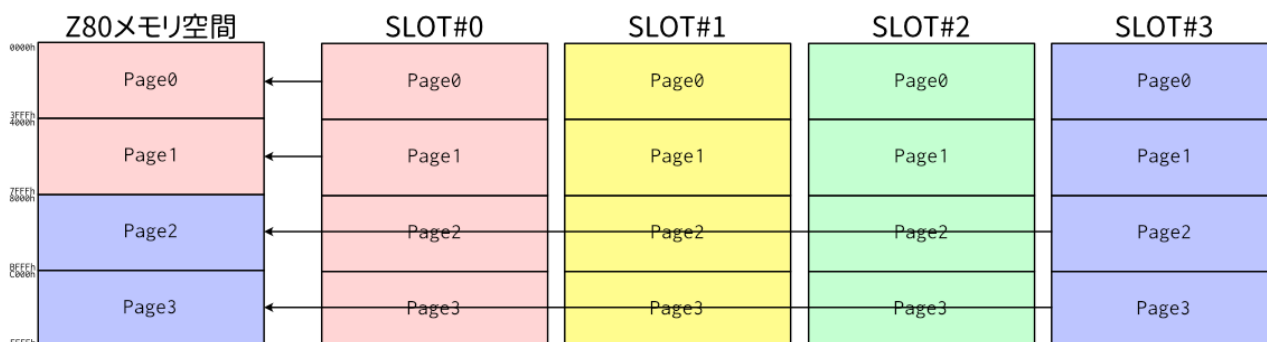


図 2-2. 基本スロット選択

図 2-2.における矢印が基本スロット選択レジスタに相当します。この基本スロット選択レジスタは、I/O 空間の A8h に読み書きできるレジスタとして接続されています。I/O A8h に書き込める 8bit の値は、図 2-3 に示すように、bit1, bit0 が Page0 に出現するスロット#、bit 3, bit2 が Page1 に出現するスロット#、bit 5, bit4 が Page2 に出現するスロット#、bit 7, bit6 が Page3 に出現するスロット#、に対応しています。

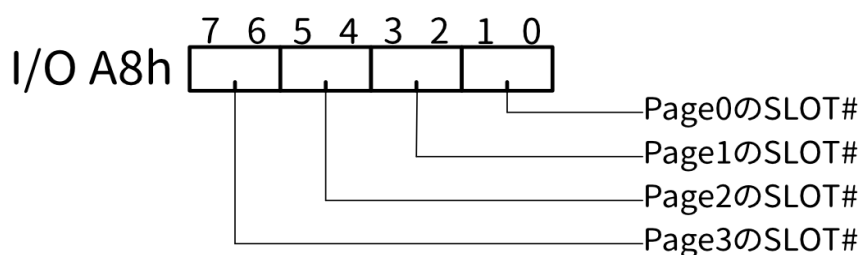
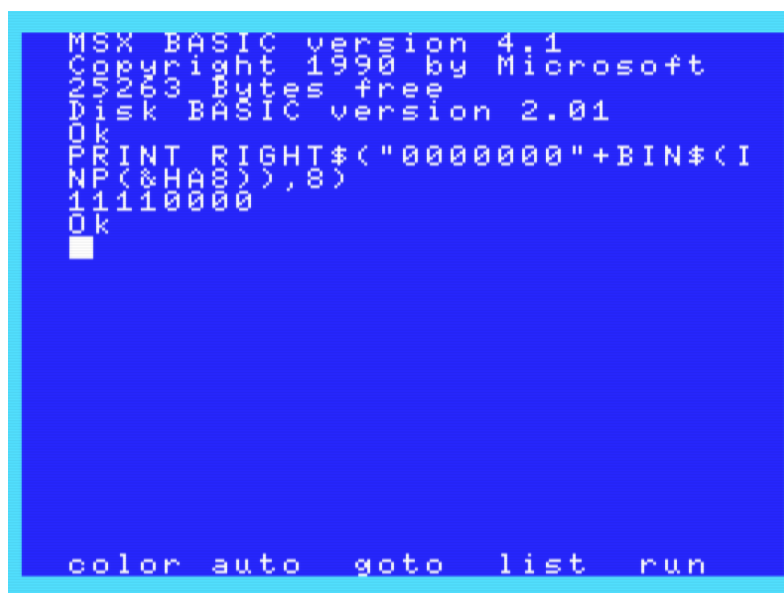


図 2-3. 基本スロット選択レジスタ (I/O A8h)

MSX Memory Architecture

基本スロット選択レジスタは MSX-BASIC から読み出すことができます。FS-A1GT で読み出したイメージを図 2-4.に示します。



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

図 2-4. FS-A1GT で MSX-BASIC から基本スロット選択レジスタを読む

これは、Page0 は"00" なので SLOT#0, Page1 も"00"なので SLOT#0, Page2 は"11"なので SLOT#3, Page3 も"11"なので SLOT#3 になっていることが分かります。

一般的に SLOT#1 はカートリッジスロット 1、SLOT#2 はカートリッジスロット 2 として外部に現れていることが多いです。希にプログラム上のスロット#と本体に刻印されているカートリッジスロット#が一致していない機種や、SLOT#3 までカートリッジスロットとして外に出ている機種も存在しています。

MSX-BASIC から OUT 命令で A8h へ書き込むことも出来ませんが、MSX-BASIC 自体が Page0, Page1 に出現している MAIN-ROM 上に存在していること、BASIC プログラムが Page2, Page3 に出現している RAM に記録されていること、MSX-BASIC や BIOS のワークエリアが Page3 に出現している RAM に記録されていることから、他のスロットに切り替えてしまうと MSX-BASIC が暴走してしまう恐れがあります。

スロット切替には、拡張スロットも関連しています。2.2. 拡張スロットも併せて参照下さい。

2.2. 拡張スロット

2.1. 基本スロットで、各 Page は 4 つのスロットから 1 つを選択して Z80 メモリ空間に出現させることを説明しました。拡張スロットは、さらにこの一つの基本スロットを 4 つに拡張する仕組みのことを指しています。

基本スロットは、必ずしも拡張されているとは限りません。一般的にカートリッジスロットは基本スロットです。拡張スロットユニットと呼ばれる周辺機器をカートリッジスロットに装着して、4 つに増やすことが出来ますが、この拡張スロットユニットは本節の拡張スロットの仕組みを利用したもののになります。

本体内部に隠れているスロット(SLOT#0 や SLOT#3)は、拡張されていることがあります。(拡張スロットユニットのような大きなものが本体に入っているわけではなく、内部ロジックとして等価であるという意味です。)

拡張されたスロットの Page0~3 に、4 つの拡張スロットのうちどのスロットが接続されるかどうかを決めるのが拡張スロット選択レジスタです。これは、そのスロットの FFFFh に存在しています。I/O ではなく、メモリ上に拡張スロット選択レジスタがマッピングされています。

このイメージを図 2-5. に示します。

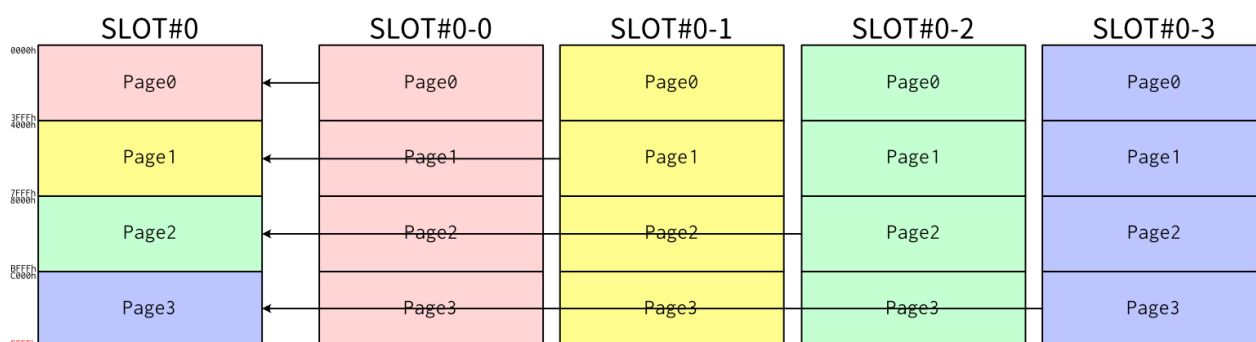


図 2-5. 拡張スロットのイメージ

図 2-5. は、基本スロット SLOT#0 が拡張されている例を示しています。SLOT#0 の拡張スロットとして SLOT#0-0, SLOT#0-1, SLOT#0-2, SLOT#0-3 が存在しています。基本スロットの考え方とほぼ同じですが、Page3 が FFFEh までである点が注意点です。SLOT#0 の FFFFh には拡張スロット選択レジスタが接続されていて、拡張スロット SLOT#0-X の FFFFh にはアクセス出来ません。

MSX Memory Architecture

拡張スロット選択レジスタ FFFFh のビットマップを図 2-6. に示します。

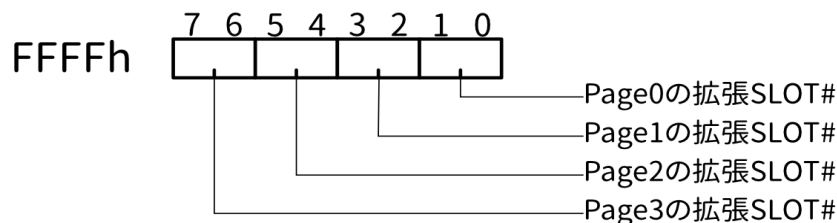


図 2-6. 拡張スロット選択レジスタ

書き込む値は、基本スロット選択レジスタと似た感じになっています。しかし、読み出すと全ビット反転した値が読み出せます。これは、この基本スロットが拡張されているか否か判定する目的でそのようなになっているものと思われます。

仮に SLOT#1 が拡張されておらず、64KB-RAM カートリッジが装着されていると、FFFFh に値 A を書き込んで読み出すと値 A がそのまま読み出されます。一方で、SLOT#1 が拡張されている場合に、SLOT#1-0 に 64KB-RAM カートリッジを装着したとしても、SLOT#1 の FFFFh に値 A を書き込んだ場合、拡張スロット選択レジスタへ書き込むことになり、FFFFh から読み出すと拡張スロット選択レジスタの反転が返ってきます。つまり、値 A を書き込んだので値 A の全ビット反転が読み出されるわけです。BIOS は、この拡張スロット選択レジスタの場合に反転した値が読まれることを利用して、拡張スロットの有無を調べています。

調べた結果は、図 2-7. に示す EXPTBL(FCC1h~FCC4h) に格納されています。

MSX Memory Architecture

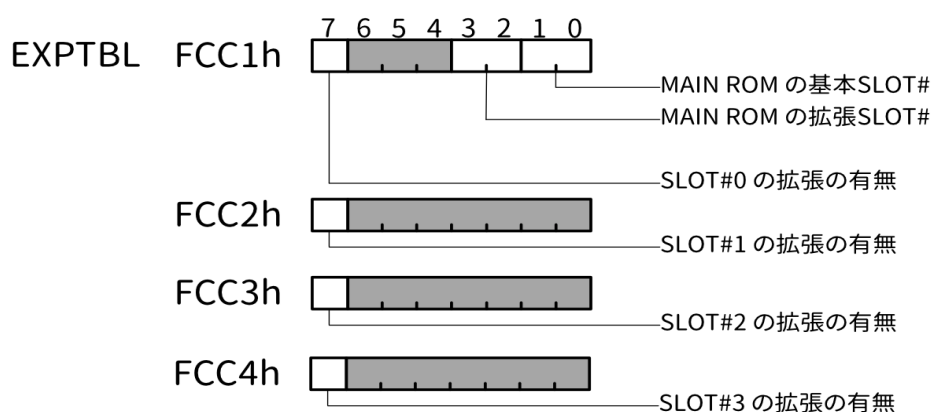


図 2-7. EXPTBL

EXPTBL の内容は、MSX 起動後すぐに BIOS が図 2-7.のビット配置に従って書き込みます。FCC1h に格納されている MAIN-ROM のスロット#は、通常は SLOT#0 か SLOT#0-0 を示す値になっていますが、決め打ちにせず、この値を読み出して MAIN-ROM のスロットとして利用するのが無難です。（例えば MSX バージョンアップアダプタを使用しているときは SLOT#0 ではないスロットの値が格納されています。）

拡張スロット選択レジスタですが、拡張スロットが接続されている基本スロット毎に存在しています。例えば、SLOT#0, SLOT#3 が拡張されているとすれば、SLOT#0 の FFFFh にはスロット 0 の拡張スロット選択レジスタ、SLOT#3 の FFFFh にはスロット 3 の拡張スロット選択レジスタが存在しています。これを読んでお気づきの方も居られるかと思いますが、拡張スロットを切り替えるためには Page3 をそのスロットに切り替えて、拡張スロット選択レジスタを Z80 メモリ空間の FFFFh に出現させなければなりません。一方で、Page3 には通常スタックメモリや BIOS ワークエリアなどがあるので迂闊に切り替えると暴走します。そこで、簡単に安全に切り替えられるように BIOS にはスロット切替ルーチンが用意されています。次節ではそのような 2.3. スロット関連の BIOS ルーチンについて説明します。

2.3. スロット関連の BIOS ルーチン

ここでは、スロット関連の BIOS ルーチンについて個別に説明します。

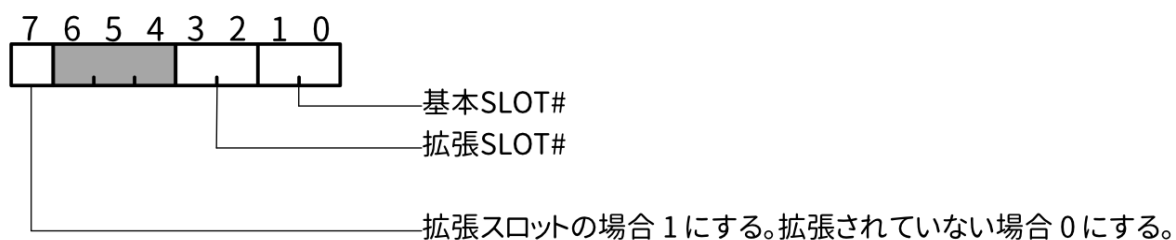
MSX Memory Architecture

RDSLT (000Ch)

指定のスロットの指定のアドレス 1byte を読み出します。

入力:

A レジスタ: 「指定のスロット」を示す値。



HL レジスタ: 「指定のアドレス」を示す値。

出力:

A レジスタ: 読み出した値

詳細:

このルーチン呼び出すと、AF,BC,DE レジスタが破壊されます。

呼び出し前に指定する A レジスタの値は、bit7 に拡張スロットの有無を指定します。要するに (FCC1h + 目的の基本スロット#) に格納されている値の bit7 をそのまま指定します。そして、目的の基本スロット#を bit1,bit0 に。拡張スロット#を bit3,bit2 に指定します。SLOT#3-2 なら、10001011 になります。16 進数表記なら 8Bh です。

BIOS が、HL レジスタの上位 2bit に対応する Page を、A レジスタで示されるスロットに切り替えてから、LD A,(HL) して、スロットを元に戻します。

下記のように使います。

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

これで、SLOT#3-2 の 1234h 番地にある 1byte の値を読んできて A レジスタに返してくれます。

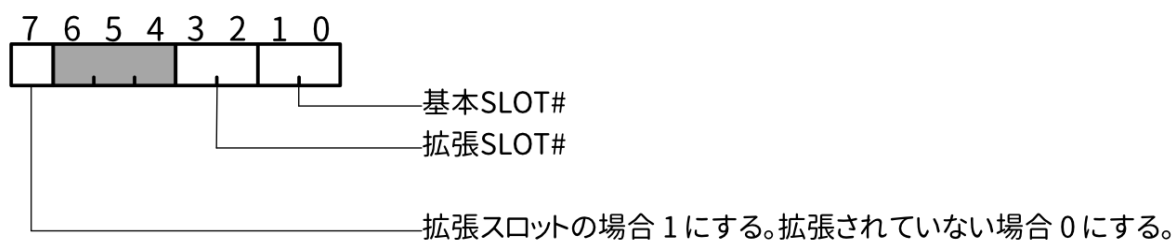
MSX Memory Architecture

WRSLT (0014h)

指定のスロットの指定のアドレスへ 指定の値を 1byte を書き込みます。

入力:

A レジスタ: 「指定のスロット」を示す値。



HL レジスタ: 「指定のアドレス」を示す値。

E レジスタ: 「指定の値」を示す値。

出力:

なし

詳細:

このルーチン呼び出すと、AF,BC,D レジスタが破壊されます。

スロットの指定の仕方は、RDSLT と同じなので、そちらの説明を参照して下さい。

BIOS が、HL レジスタの上位 2bit に対応する Page を、A レジスタで示されるスロットに切り替えてから、LD (HL),E してから、スロットを元に戻します。

下記のように使います。

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

これで、SLOT#3-2 の 1234h 番地に ABh を書き込んでくれます。

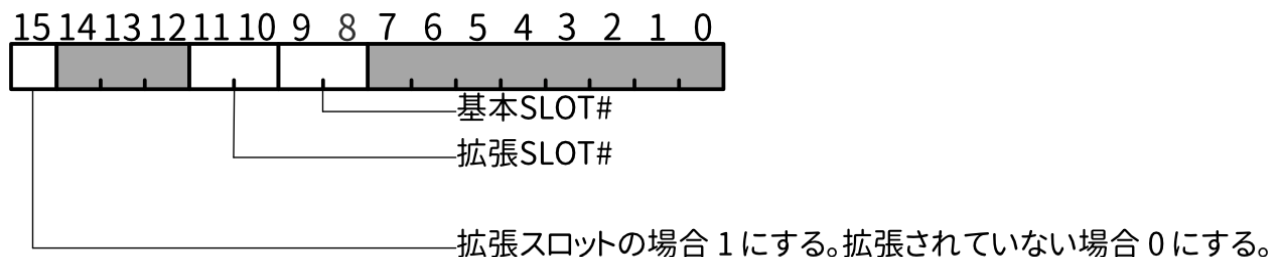
MSX Memory Architecture

CALSLT (001Ch)

指定のスロットの指定のアドレスをサブルーチンコールします（インタースロットコールと呼びます）。

入力:

IY レジスタ：「指定のスロット」を示す値を上位 8bit に格納する。下位は 8bit は無視される。



IX レジスタ：「指定のアドレス（サブルーチンのアドレス）」を示す値。

その他のレジスタ：呼び出すサブルーチンのパラメータ。

出力:

呼び出すサブルーチンによる。

詳細:

このルーチンは、「指定のアドレス（サブルーチンのアドレス）」に対応する Page を、「指定のスロット」に切り替えた後、「指定のアドレス（サブルーチンのアドレス）」を CALL します。次に「指定のアドレス（サブルーチンのアドレス）」に対応する Page を、元のスロットに戻してくれます。

このルーチンを呼び出したときに破壊されるレジスタは、呼び出すサブルーチンによって異なります。

例えば、MSX-DOS アプリケーションが起動した時点では Page0~3 のすべてが RAM になっていますが、MAIN-ROM の BIOS ルーチンを呼び出したい場合などに利用できます。MSX-DOS 動作時の Page0 は RAM ですが、スロット関連のサブルーチンコールは BIOS と同じアドレスに存在していて、利用可能になっています。

下記のようなコードで、MSX-DOS アプリケーションから SCREEN モードを切り替える BIOS ルーチン CHGMOD (005Fh) を呼び出すことができます。

MSX Memory Architecture

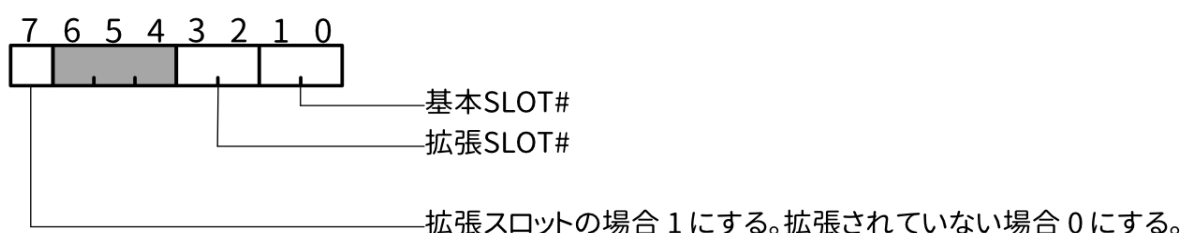
LD	A, 1	; SCREEN1 に切り替える
LD	IY, [0xFCC1 - 1]	; MAIN-ROM スロットを IY の上位 8bit へ
LD	IX, 0x005F	
CALL	0x001C	

ENASLT (0024h)

割り込み禁止状態にして、指定の Page を指定のスロットに切り替えます。割り込み禁止状態のまま戻ります。

入力:

A レジスタ: 「指定のスロット」を示す値。



H レジスタ: 上位 2bit で対象の Page を指定。

出力:

なし

詳細:

すべてのレジスタが破壊されます。

インタースロットコールを使えば、他のスロットのサブルーチンコールを利用することはできますが、その都度スロットを切り替えて、また戻して、といった処理が発生します。頻繁に同じスロットに対してサブルーチンコールしたり、読み書きしたりするようなケースでは非常に効率が悪くなります。そこで、ENASLT を使ってスロットを切り替えて、必要なサブルーチンコール・読み書きを行ってから、また ENASLT でスロットを戻すことで、スロット切替の頻度を抑えることができます。

MSX Memory Architecture

スロットの切替は、それなりに負荷の掛かる処理になっていますので、ゲームのメインループのような処理速度を要する場所で他のスロットを呼び出すようなケースでは、インタースロットコールを多用するより ENASLT による最小限の切替を検討した方が良いでしょう。

ENASLT は、割り込み禁止で戻ります。これは、割り込み処理ルーチンが、想定と異なるスロット構成に変化していることにより暴走する問題を回避するための防御策です。もし割り込み処理自体に対策を入れてあれば ENASLT から戻り次第 EI することができます。

当然ですが、ENASLT を呼び出すプログラムが Page1 にある状態で、Page1 のスロットを切り替えてしまうと ENASLT から戻ったときに別のコードへ戻ってしまい暴走しますので、その点も留意の上ご利用下さい。

また、Page3 の切替はおやめ下さい。スロット切り替えルーチンの本体が Page3 の RAM 上にいたり、スタックメモリも Page3 にいることが多いです。

下記のようにすることで、MSX-DOS から、SLOT#1 に装着された ROM カートリッジの先頭 8KB を Page2 へコピーすることが出来ます。

```
LD      A, [0xFCC2]      ; SLOT#1 の拡張の有無取得
AND     A, 0x80
OR      A, 0x01          ; SLOT#1 か SLOT#1-0
LD      H, 0x40          ; Page1
CALL    0x0024           ; Page1 を SLOT#1 か SLOT#1-0 に切り替える
LD      DE, 0x8000
LD      HL, 0x4000
LD      BC, 0x2000
LDIR                    ; Page1 の先頭 8192byte を 0x8000~ へコピー
LD      A, [0xF342]      ; Page1 の RAM があるスロットを取得
LD      H, 0x40          ; Page1
CALL    0x0024           ; Page1 を SLOT#1 か SLOT#1-0 に切り替える
EI                        ; 割り込み許可
```

※メモリアドレス上にレジスタがマッピングされているデバイスが存在します。そういったデバイスで、レジスタ部分を読み出すと、何らかの機能が動いてしまう場合があります。例えば、SD カードスロットを搭載したカートリッジで、SPI 通信機能がメモリ上のレジスタとしてマッピングされているものがあります。読み出すと SPI 受信が働いてしまいますのでご注意下さい。

CALLF (0030h)

インタースロットコールします。

入力：

0030h をコールしているコードのすぐ後にスロット#を示す値とアドレス値を格納。

スロット#を示す値は、CALSLT の A レジスタに指定する値と同じ。

出力：

呼び出し先による。

詳細：

破壊レジスタは呼び出し先による。

機能としては、CALSLT と同じですが、スロットとアドレスの指定に IY, IX を使いません。また、0030h は CALL 0030h でも良いですが、より効率の良い RST 30h を使えます。具体的には下記のように呼び出します。

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

L1:

0x0030 の呼び出しからの戻り番地は、L1 の位置になります。

MSX の場合、スロット#が変わる可能性を考慮しなければならないので、一般的には RAM 上で動作するプログラムで、DB 0x80 のスロット#のところをプログラムで書き替えて使うのが良いと思います。

LD	A, [0xFCC1]	; MAIN-ROM SLOT#
LD	[CALL_SLOT_NUM], A	
LD	A, 1	; SCREEN1

MSX Memory Architecture

```

    RST      0x30
CALL_SLOT_NUM:
    DB       0x80
    DW       0x005F          ; CHGMOD
L1:

```

2.4. 基本スロットと拡張スロットの関係

基本スロットが SLOT#0, SLOT#1, SLOT#2, SLOT#3 の4つあり、このそれぞれに独立して拡張スロットを増設することが出来ます。SLOT#0 に拡張スロットが接続されていると、SLOT#0 は、SLOT#0-0, SLOT#0-1, SLOT#0-2, SLOT#0-3 の4つに増えます。このように、基本スロットそれぞれに拡張スロットの有無があるために、EXPTBL は SLOT#0 の拡張の有無を示す FCC1h、SLOT#1 の拡張の有無を示す FCC2h, SLOT#2 の拡張の有無を示す FCC3h, SLOT#3 の拡張の有無を示す FCC4h の4つが存在しているわけです。すべての基本スロットが拡張無しの機種もあれば、カートリッジスロット以外の内蔵スロットはすべて拡張されている機種もあります。もちろんその中間的な機種もあります。そのため、スロットを切り替える際は EXPTBL をチェックして BIOS を使って切り替えるのが安全です。

例として、FS-A1GT のスロット構成を図 2-8. に示します。

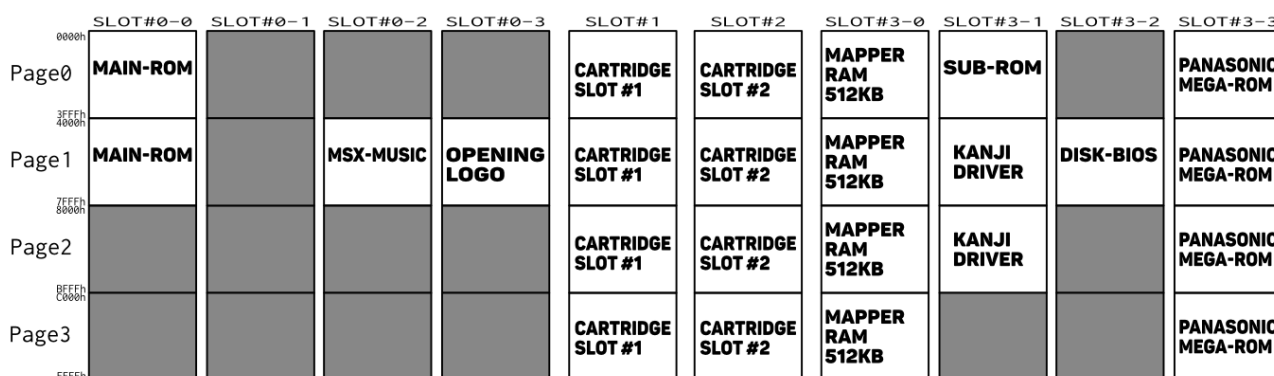


図 2-8. FS-A1GT のスロット構成

SLOT#0, SLOT#3 が拡張されていますね。SLOT#1, SLOT#2 はカートリッジスロットとしてそのまま外部に出ています。そのため、本体内に存在するスロット選択用レジスタは下記の3つが存在します。

基本スロット選択レジスタ I/O A8h

MSX Memory Architecture

SLOT#0 拡張スロット選択レジスタ SLOT#0 - FFFFh

SLOT#3 拡張スロット選択レジスタ SLOT#3 - FFFFh

SLOT#0 の構成は SLOT#0 拡張スロット選択レジスタによって「どの SLOT#0 拡張スロットが出現するか」を設定し、

SLOT#3 の構成は SLOT#3 拡張スロット選択レジスタによって「どの SLOT#3 拡張スロットが出現するか」を設定し、

さらに Z80 メモリ空間に出現する内容は、基本スロット選択レジスタによって決定されます。

MSX のメモリ空間は、このスロットという概念が基本ですが、この各スロットに接続するメモリデバイス、さらに独自の拡張が施されているケースがあります。それらは、スロットとは独立に機能することに注意して下さい。次の章からは、スロットとは独立したメモリ空間拡張の仕組みの中で、特にメジャーなものをピックアップして説明していきます。

余談ですが、図 2-8.に登場する各スロットの内容について簡単に説明しておきます。MAIN-ROM は、電源投入後真っ先に実行されるプログラムであり、BIOS や MSX-BASIC が入っている ROM です。MSX-MUSIC は、FM-BIOS や MSX-BASIC の CALL MUSIC 等の MSX-MUSIC 及び MSX-MIDI の処理が入った ROM です。OPENING LOGO は、起動時の MSX ロゴを表示するプログラムです。MAPPER RAM は、後述のメモリーマッパー対応 RAM です。SUB-ROM は、MSX2 から追加になった BIOS の ROM です。KANJI DRIVER は、MSX-BASIC の CALL KANJI をサポートする ROM です。DISK-BIOS は、DISK-BASIC や MSX-DOS/DOS2 が入っている ROM です。PANASONIC MEGA-ROM は、MSX-JE・内蔵ワープロソフト・A1 コックピットと呼ばれる内蔵ソフト・MSX-View・MSX-View 用 12 ドット漢字 ROM・MSX-JE 用辞書 SRAM・RAM・MAIN-ROM 等搭載しているメモリがほぼすべてここから覗けるようになってます。

3. メモリーマッパー対応 RAM

MSX 本体に搭載されている RAM は、大きく分けてメモリーマッパー対応の RAM と非対応の RAM の二種類が存在します。ほぼすべての MSX1 と、一部の MSX2 の本体 RAM が非対応の RAM で、一部の MSX2 と、MSX2+以降の RAM はメモリーマッパー対応 RAM となっています。

メモリーマッパーとは、16KB の単位で区切られた RAM で、これを Page0~3 の好きな領域に出現させられるようにした（メモリをマッピングできるようにした）仕組みのことを指します。メモリ

MSX Memory Architecture

マッパー非対応の RAM は、単純に RAM が繋がっているだけなので、例えば Page0 に繋がった RAM はスロット切替によって Page0 に出現させることは出来ても、Page1~3 に出現させることは出来ません。

メモリーマッパー対応 RAM は、16KB 単位で区切られており、0 からの連番がふられています。例えば、64KB のメモリーマッパー対応 RAM であれば Segment#0~3 の 4 セグメント、256KB のメモリーマッパー対応 RAM であれば Segment#0~15 の 16 セグメント存在することになります。最低でも 64KB(4 セグメント)は存在します。

メモリーマッパー対応 RAM は、Page0 マッパーセグメント選択レジスタ、Page1 マッパーセグメント選択レジスタ、Page2 マッパーセグメント選択レジスタ、Page3 マッパーセグメント選択レジスタの 4 つのレジスタを持っています。これは、それぞれの Page に出現するセグメント#を指定するレジスタです。このレジスタのハードウェアとしての初期値（電源投入直後の値）は未定義です。MSX2 以降のメモリーマッパー対応 RAM に対応した BIOS であれば起動時に、Page0 マッパーセグメント選択レジスタ=3, Page1 マッパーセグメント選択レジスタ=2, Page2 マッパーセグメント選択レジスタ=1, Page3 マッパーセグメント選択レジスタ=0 に初期化されます。一方で MSX1 の場合は初期化されません。汎用部品の組み合わせで構成されたメモリーマッパー対応 RAM の場合、すべてのマッパーセグメント選択レジスタは初期値が 0 の場合が多いです。最近出回っているコンボカートリッジに内蔵のメモリーマッパー対応 RAM はハードの初期値としてメモリーマッパー対応 RAM に対応した BIOS と同じ値が設定されているケースもあります。しかし、そのような初期値を期待してしまうのは危険です。

マッパーセグメント選択レジスタは、I/O の FCh, FDh, FEh, FFh に接続されています。理由は後で述べますが書き込み専用で、読み出しではいけません。Page0 マッパーセグメント選択レジスタが FCh, Page3 マッパーセグメント選択レジスタが FFh といった具合です。つまり、1 つのマッパーセグメント選択レジスタは 8bit のため、0~255 の 256 セグメントが最大容量ということになります。 $16\text{KB} \times 256 = 4096\text{KB} = 4\text{MB}$ ですね。128KB の場合のイメージを図 3-1.に示します。

MSX Memory Architecture

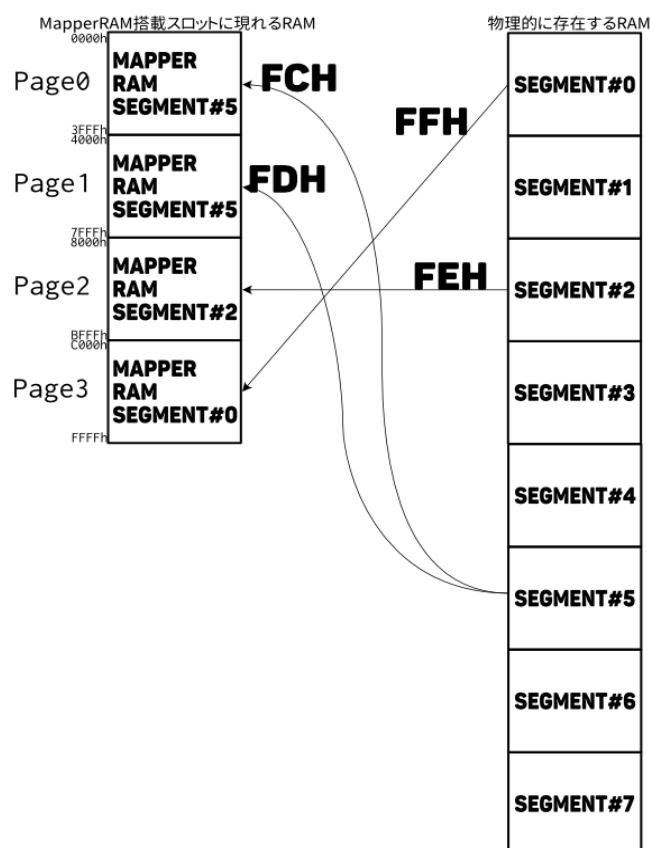


図 3-1. 128KB(8 セグメント)のスロット出現 RAM のイメージ

図 3-1.で左側が、メモリーマッパー対応 RAM が装着されているスロットに相当します。Page0～3 すべてが RAM になります。右側が、実際に搭載されている RAM です。128KB = 16KB × 8 であるため、8 セグメント分の RAM が存在し、Segment#0～7 の連番がふられています。左側と右側が 4 本の線で結ばれていますが、これがマッパーセグメント選択レジスタのイメージです。図 3-1.では I/O FCh = 5, FDh = 5, FEh = 2, FFh = 0 が指定されています。このように同じセグメントを異なる Page に同時に出現させることも出来ます。あるときは Page0 に Segment#6 を、次のタイミングでは Page1 に Segment#6 を、という出現 Page を変更することも出来るため、スロットよりも柔軟なセグメント指定が可能になっています。これによって実現される 64KB の空間に現れる大容量 RAM ですが、普通のスロットに接続されるため、基本スロット・拡張スロットの一部であるスロットに対してメモリーマッパーが繋がることになります。

メモリーマッパー対応 RAM のカートリッジ版も存在しています。カートリッジ版を使うことで RAM を増設できるわけです。興味深いことに、本体内蔵のメモリーマッパー対応 RAM も、カート

MSX Memory Architecture

リッジタイプのメモリーマッパー対応 RAM も、マッパーセグメント選択レジスタの I/O アドレスが同じになっています。

カートリッジタイプのメモリーマッパー対応 RAM が存在し、本体内蔵のメモリーマッパー対応 RAM も存在していますが、これらは共存出来ます。複数のメモリーマッパー対応 RAM が存在する場合、メモリーマッパーのマッパーセグメント選択レジスタに書き込みを行うと、接続されているすべてのメモリーマッパー対応 RAM のセグメントが同じように切り替わります。例えば、SLOT#1 にカートリッジタイプの 4MB メモリーマッパー対応 RAM が搭載されていて、本体の SLOT#3-0 には 512KB のメモリーマッパー対応 RAM が搭載されている場合は、図 3-2. のようなイメージになります。

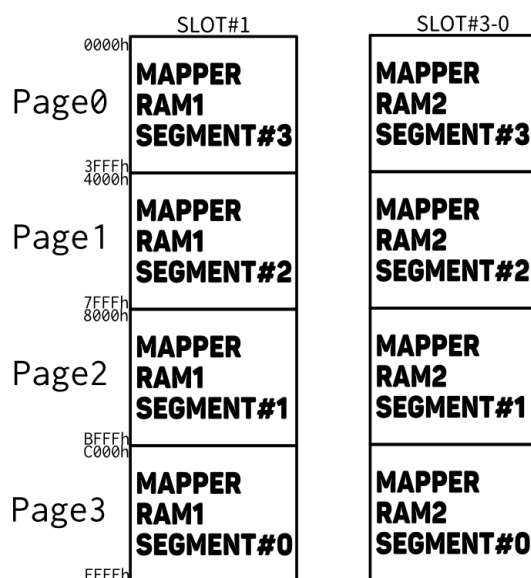


図 3-2. SLOT#1 と SLOT#3-0 の 2 スロットにメモリーマッパー対応 RAM

SLOT#1 には、4MB のメモリーマッパー対応 RAM カートリッジが装着されており、これを説明のために MAPPER RAM1 と呼ぶことにします。同様に SLOT#3-0 には 512KB のメモリーマッパー対応 RAM が本体内で接続されており、これを MAPPER RAM2 と呼ぶことにします。それぞれ独立したデバイスであるため、例えば SLOT#1 のセグメントを SLOT#3-0 に出現させることは出来ません。SLOT#1 の中で 4MB = 256 セグメントの RAM が使えて SLOT#3-0 の中で 512KB = 32 セグメントの RAM が使える、といった状況です。

ここで、I/O FCh に 15 を書き込むと、図 3-3. のような状態になります。

MSX Memory Architecture

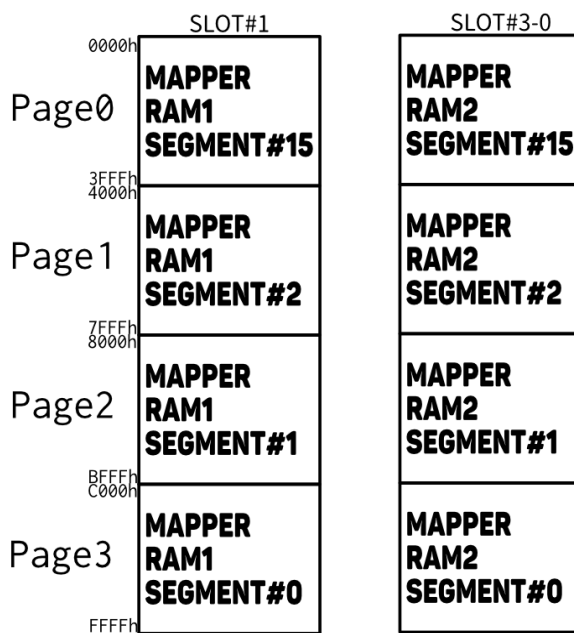


図 3-3. Page0 のセグメントを 15 に変更

SLOT#1, SLOT#3-0 が、Z80 のメモリ空間に出現しているか否かにかかわらず、MAPPER RAM1 と MAPPER RAM2 の両方の Page0 のセグメントが Segment#15 に切り替わります。では、I/O FCh に 32 を書き込んだらどうなるのでしょうか？図 3-4.のようになります。

MSX Memory Architecture

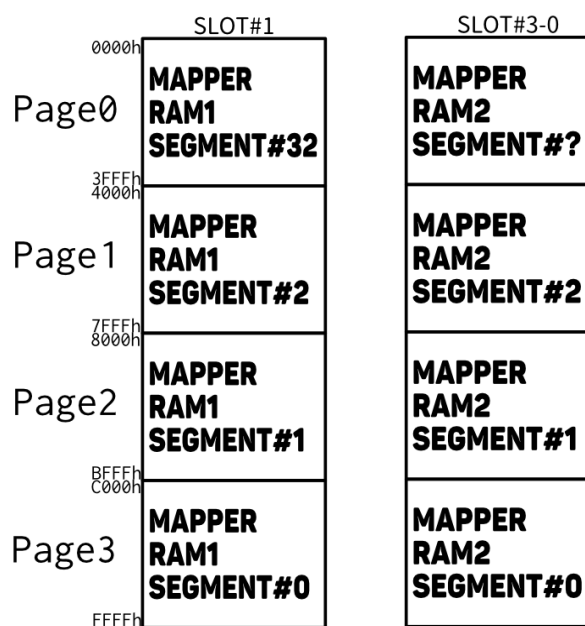


図 3-4. Page0 のセグメント#を 32 に変更

MAPPER RAM1 の方は 256 セグメント(0~255)ありますが、MAPPER RAM2 の方は 32 セグメント(0~31)しか存在しません。Segment#32 に相当する値をマッパーセグメント選択レジスタ 0xFC に書き込んでみたわけですが、MAPPER RAM1 は素直に Segment#32 に切り替わります。MAPPER RAM2 は Segment#32 が無いので、動作としては不定です。一般的にはセグメント#の上位ビットが無いが無視されているので Segment#32 = Segment#0 のような挙動をすることが多いです。

MEM-768 という 768KB のメモリーマッパー対応 RAM が存在していますが、容量が中途半端です。768KB = 16KB × 48 なので 48 セグメント存在するわけですが、では I/O の FCh に 48 を書き込んだら、いったい何番のセグメントが出現するのか？。所有している方のご協力(*1)により、確認できました。マッパーセグメント選択レジスタは 6bit 搭載されており、この上位 2bit をデコードしています。MEM-768 には 256KB の RAM が 3 つ搭載されており、上位 2bit のデコード結果によりこの 3 つのいずれかを選択しています。2bit あるので 4 通りの組み合わせがありますが、3 に相当する値が未接続になっているようです。そのため、Segment#48~#63 を指定すると、何も繋がっていない状態になり、書き込めず、読み込むと概ね FFh が返ってくる状態になるようです。ただし、MEM-768 がそのような設計になっているだけで、存在しないセグメント#を指定した場合には、何が登場するかは未定義と解釈するのが無難です。

(*1) Takashi Kobayashi 様にテストプログラムを実行していただき、確認しました。

MSX Memory Architecture

I/O FCh, FDh, FEh, FFh に複数のメモリーマップ対応 RAM が接続されることから、このアドレスは基本的に書き込み専用とされています。読み出し厳禁とされています。ハードの組み合わせによっては複数のメモリーマップ対応 RAM からマップセグメント選択レジスタの値が返され、バス競合が発生して MSX 本体が物理的に壊れる可能性があるようです。当方では、本当に壊れる機種があるかは未確認です。

メモリーマップ対応 RAM を使うことによって、1 スロットあたり 4MB もの大容量を実現できるので、「複数のアプリケーションでメモリを棲み分けて使う」といった要求が出てきます。そのためには、すべてのメモリーマップ対応 RAM が連動して切り替わってしまうマップセグメント選択レジスタを管理する必要があります。その管理機構として用意されているのが、MSX-DOS2 を搭載すると拡張 BIOS にインストールされるメモリーマップサポートルーチンになります。次の節では、メモリーマップサポートルーチンの使い方について説明します。

3.1. メモリーマップサポートルーチン

MSX には、MAIN-ROM や SUB-ROM に搭載の BIOS の他に、拡張 BIOS という仕組みが存在しています。カートリッジなどで後付けしたデバイスモジュールが、その制御のための BIOS ルーチンを公開する仕組みですね。

メモリーマップサポートルーチンは、この拡張 BIOS の仕組みを使って提供されます。しかしながら、メモリーマップ対応 RAM は単純に RAM だけで、「メモリーマップサポートルーチンを仕掛ける仕組み」は搭載していません。これを搭載しているのは、MSX-DOS2 や、その後継の Nextor になります。といっても、メモリーマップサポートルーチンの有無を確認するために、MSX-DOS2 や Nextor の存在確認をする必要はありません。拡張 BIOS には登録される各種ルーチンで共通の手続きがあり、メモリーマップサポートルーチンもそのルールに則っています。

プライマリーマップとセカンダリーマップ

使い方を説明する前に、メモリーマップの種類について説明します。

メモリーマップ対応 RAM が 1 スロットしか存在しない場合、それがプライマリーマップとなります。その場合、セカンダリーマップは存在しません。

MSX Memory Architecture

メモリーマップパー対応 RAM が2 スロット以上存在する場合、1 つがプライマリーマップパー、残りはセカンダリーマップパーとなります。

MSX2/2+では、最も容量が多いメモリーマップパー対応 RAM がプライマリーマップパーになります。MSXturboR では、本体内蔵の SLOT#3-0 にあるメモリーマップパー対応 RAM がプライマリーマップパーになります。MSX2/2+までは、内蔵と増設カートリッジで、容量しか差が無かったので、スロット切り替えの手間を最小限に出来るように最も容量が多いものがプライマリーマップパーとして選択される仕様でした。最も容量が多いメモリーマップパー対応 RAM が複数装着されている場合は、より小さい番号のスロットに搭載しているメモリーマップパー対応 RAM がプライマリーマップパーになります。MSXturboR の場合、内蔵の RAM は R800 向けに高速アクセス出来るようになっており、一方で増設カートリッジは MSX2/2+との互換速度で動作するために、内蔵 RAM の方が圧倒的に高速になっています。その関係で、turboR は内蔵 RAM をプライマリーマップパーにするよう仕様変更されたのだと思います。

メモリーマップパーサポートルーチンでは、メモリーマップパー対応 RAM の情報テーブルを持っており、このテーブルアドレスを返すルーチンが存在しています。アプリケーションは、このテーブルを参照することにより、各メモリーマップパー対応 RAM のスロット#やセグメント数などを簡単に知ることが出来るようになっていきます。この情報テーブルの先頭には、かならずプライマリーマップパーの情報が格納されています。セカンダリーマップパーは、二つ目以降として並んでいます。プライマリーマップパーの方が、少しだけアクセスが簡単なのです。

次に、MSX-DOS2/Nextor が動作中に表に出ている 64KB の RAM 空間は TPA と呼ばれますが、この TPA もプライマリーマップパーに割り付けられています。

MSX-DOS2/Nextor の DOS ファンクションで、ファイル読み書きアクセスをする場合の転送元・転送先バッファとして指定できるのは、このプライマリーマップパーのみです。セカンダリーマップパーの領域を指定すると正常に機能しない点が、プライマリーとセカンダリーの大きな差だと思います。ファイルの内容をセカンダリーマップパーへ読み出したい場合は、一旦プライマリーマップパーへ読み込んだ後、それをセカンダリーマップパーへブロック転送する必要がありますのでご注意ください。

このように、プライマリーマップパーとセカンダリーマップパーには、少なからず差があります。turboR では内蔵 RAM が必ずプライマリーになる上に、R800-DRAM モードでは BIOS が一部セグメントを占拠しているため、特に FS-A1ST では空きが少なくなっています。セカンダリーマップパーでも問題ない場合は、積極的にセカンダリーマップパーを利用した方が対応可能な環境が増えると思います。

マッパーサポートルーチンの使い方

拡張 BIOS のルールに則って、まず拡張 BIOS が存在するか確認します。次に、マッパーサポートルーチンが存在するか確認します。そして、得られたジャンプテーブルを RAM 上のどこかへコピーして、それ以降はそのジャンプテーブルのエントリを CALL する形で利用します。最初の手続きは面倒ですが、一度ジャンプテーブルを得てしまえば、それ以降はシンプルで非常に高速に動作するルーチンです。では、以下に具体的な使い方の説明に入ります。

まず最初に、拡張 BIOS の仕組みそのものが存在するかどうかを確認する必要があります。具体的には、HOKVLD (FB20h) の内容を読み出して、bit0 が 0 ならば拡張 BIOS そのものが存在しません。1 ならば拡張 BIOS が存在します。

次に、拡張 BIOS の中にマッパーサポートルーチンが存在するか調べます。A レジスタを判定用に 0 に、D レジスタにマッパーサポートルーチンのデバイス番号を示す 4 を、E レジスタに機能番号 1 を指定して、拡張 BIOS のエントリである EXTBIO (FFCAh) を CALL します。マッパーサポートルーチンが存在しなければ A レジスタが 0 のままです。A レジスタが 0 以外に変わっていれば、マッパーサポートルーチンが存在しています。

次に、A レジスタに 0 を、D レジスタにデバイス番号 4 を、E レジスタに機能番号 2 を指定して、EXTBIO を CALL します。すると、システムワーク領域にあるマッパーサポートルーチンのジャンプテーブルのアドレスが HL レジスタに返ってきます。これは 48byte の領域で、JP xxxx が 16 個並んでいるテーブルです。使いやすいように決まったアドレスにコピーしておくと呼び出しが楽になります。

以下に、マッパーサポートルーチンの確認処理の例を示します。

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

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

    ; get MapperSupportRoutine's table
    xor     a, a
    ld      de, 0x0401   ; D=デバイス ID, E=01h
    call    extbio
```

MSX Memory Architecture

```
        or            a, a
        ret           z                ; マッパーサポートルーチンが
                                        ; 存在しない場合はエラー (Zf=1)
        ld            [mmap_table_ptr], hl
        ; get jump table
        xor           a, a
        ld            de, 0x0402      ; D=デバイス 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           ; 正常終了 (Zf=0)
mmap_table_ptr::
        dw            0                ; メモリーマッパー情報テーブルのアドレス
```

mmap_init を CALL すると、マッパーサポートルーチンが存在しない場合は、A レジスタ = 0, Z フラグ = 1 が返ってきます。マッパーサポートルーチンが存在していれば、A レジスタ = 1, Z フラグ = 0 が返ってきます。

mapper_jump_table は、RAM に置く必要があり、下記のような構成になります。

```
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
```

MSX Memory Architecture

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

C9h が詰まった 48byte の領域です。C9h は、RET のオペコードです。mmap_init を呼び出して、A=1 が返ってきた場合、ここに JP xxxx が書き込まれます。マッパーサポートルーチンの各機能の呼び出しエントリとなっています。

マッパーサポートルーチンの各エントリには、名前が付けられており、例えば +00h は ALL_SEG という名前になります。上の例では、他と名前が被りにくように mapper_all_seg としています。頭に mapper_ を付けて小文字にただけですね。このルールで 16 個のエントリにラベルを付けています。ALL_SEG を使いたければ、CALL mapper_all_seg すれば使えるわけです。

各エントリを呼び出せる準備が出来るようになったので、各エントリについて順番に説明します。

ALL_SEG

名前は、allocate segment の略だと思います。1 セグメント確保を試みるルーチンになります。

入力：

A レジスタ

0: ユーザーセグメントとして 1 セグメント確保する

1: システムセグメントとして 1 セグメント確保する

MSX Memory Architecture

B レジスタ

0: プライマリマッパーから確保する

0 以外: 複数マッパーから確保する(詳細参照)

出力:

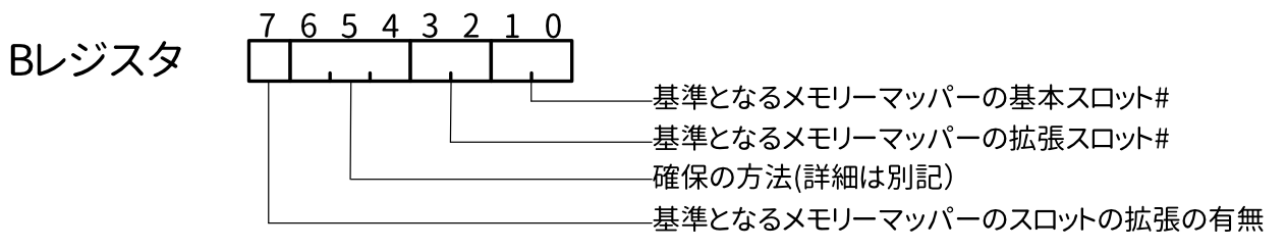
Cy フラグ: 0 = 成功, 1 = 空きセグメントが無い

A レジスタ: 確保したマッパーメモリのセグメント#

B レジスタ: 確保したマッパーメモリのスロット# (B=0 で呼び出した場合は 0)

詳細:

B レジスタに設定する値は、下記のようにになっている。



ENASLT 等で使うスロット#の指定方法と同じビットマップで、基準となるメモリーマッパーのスロットを指定する。空いている bit6,5,4 で、その基準に対してどのように確保するかを指定する。

B レジスタの bit6,5,4 に指定する値の意味は下記の通り。

000 基準となるメモリーマッパーから確保する。

001 基準となるメモリーマッパー以外のメモリーマッパーから確保する。

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

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

1?? 未定義 (指定しないこと)

MSX Memory Architecture

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

また、ユーザーセグメントはセグメント#が小さい順に割り当てられます。システムセグメントはセグメント#が大きい順に割り当てられます。例えば、セグメント#4,5,6 が空いている場合に、ユーザーセグメントを確保するとセグメント#4 が確保され、システムセグメントを確保しようするとセグメント#6 が確保されます。

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

FRE_SEG

指定のセグメントを解放する。

入力：

A レジスタ：解放するセグメント#

B レジスタ：解放するセグメントのメモリーマッパーのロット#

出力：

Cy フラグ：0 = 解放に成功, 1 = 解放に失敗

詳細：

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

失敗するのは、誰も確保していないセグメントを指定している場合のみで、A レジスタ・B レジスタの値を誤って「無関係のプログラムが確保済みのセグメント」を指定してしまった場合はそのセグメントを解放してしまいますので要注意です。例えば、MSXturboR では、BIOS 等の一部の ROM 領域を内蔵 RAM にコピーしていますが、プライマリーメモリーマッパー上で、その領域はシステムセ

MSX Memory Architecture

グメント扱いで確保されている状態になっています。そうすることで、他のアプリケーションから BIOS コピーを保護しているわけですが、これも解放できてしまいます。

RD_SEG

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

入力：

A レジスタ：読み出し対象となるセグメント#

HL レジスタ：セグメント内アドレス (上位 2bit は無効)

出力：

A レジスタ：読み出したデータ

その他のレジスタは保存されます。

詳細：

割り込み禁止で戻ります。

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

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

セグメントも含むランダムアドレスから読み出したい場合に有効な機能ですが、同一セグメントから連続で読み出したい場合は、後述のダイレクトページング(PUT_P2)を使った方が高速です。

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

WR_SEG

インターセグメントライトです。A:HL で示されるアドレスへ E を書き込みます。

入力：

A レジスタ：読み出し対象となるセグメント#

HL レジスタ：セグメント内アドレス (上位 2bit は無効)

E レジスタ：書き込む値

出力：

A レジスタ：破壊されます。

その他のレジスタは保存されます。

詳細：

割り込み禁止で戻ります。

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

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

セグメントも含むランダムアドレスから書き込みたい場合に有効な機能ですが、同一セグメントから連続で書き込みたい場合は、後述のダイレクトページング(PUT_P2)を使った方が高速です。

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

MSX Memory Architecture

CAL_SEG

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

入力：

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

IX レジスタ：呼び出すサブルーチンのアドレス

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

出力：

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

その他のレジスタ：破壊されます。

詳細：

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

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

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

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

MSX Memory Architecture

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

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

CALLS

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

入力：

レジスタを使わず、下記のようにコード上にパラメータを埋め込む仕組みです。

CALL CALLS

DB セグメント#

DW サブルーチンアドレス

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

出力：

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

その他のレジスタ：破壊されます。

詳細：

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

MSX Memory Architecture

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

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

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

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

PUT_PH

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

入力：

H レジスタ：上位 2bit で対象の Page 番号を指定。

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

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

MSX Memory Architecture

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

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

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

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

GET_PH

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

入力：

H レジスタ：上位 2bit で対象の Page 番号を指定。

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

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

MSX Memory Architecture

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

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

PUT_P0

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

入力：

Aレジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

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

GET_P0

ダイレクトページング。Page0のマッパーセグメント#を取得する。

MSX Memory Architecture

入力：

なし

出力：

A レジスタ： 取得したセグメント#。

レジスタはすべて保存されます。

詳細：

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

PUT_P1

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

入力：

A レジスタ： 目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

MSX Memory Architecture

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

GET_P1

ダイレクトページング。Page1 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

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

PUT_P2

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

MSX Memory Architecture

入力：

A レジスタ： 目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

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

GET_P2

ダイレクトページング。Page2 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ： 取得したセグメント#。

レジスタはすべて保存されます。

詳細：

MSX Memory Architecture

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

PUT_P3

なにもしません。

入力：

A レジスタ：目的のセグメント#を指定する。

出力：

なし。

レジスタはすべて保存されます。

詳細：

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

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

MSX Memory Architecture

GET_P3

ダイレクトページング。Page3 のマッパーセグメント#を取得する。

入力：

なし

出力：

A レジスタ：取得したセグメント#。

レジスタはすべて保存されます。

詳細：

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

Page3 は、BIOS ワークもある関係で PUT_P3 や PUT_PH で Segment# を変更できません。そのため、GET_P3 では、常に Segment#0 を示す 0 が A レジスタに返ってきます。

メモリーマッパー情報テーブル

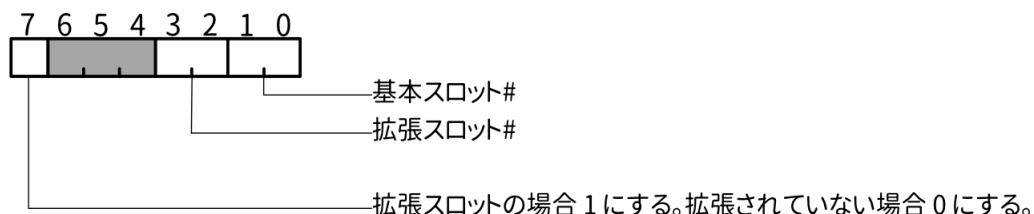
上記の初期化のサンプルプログラム（マッパーサポートルーチンの使い方）を実行すると、初期化成功時には mmap_table_ptr にメモリーマッパー情報テーブルのアドレスが格納されます。

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

MSX Memory Architecture

オフセットアドレス 意味

+0 メモリーマップ対応 **RAM** のスロット#



+1 セグメントの総数(4~255)。
(プライマリマップの場合は、8~255)

+2 未使用のセグメントの総数

+3 システムセグメントとして割り当て済みのセグメントの総数
(プライマリマップの場合は、最低でも 6)

+4 ユーザーセグメントとして割り当て済みのセグメントの総数

+5~+7 予約領域

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

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

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

この 8 バイトの構造が、メモリーマップ対応 RAM の数だけ並んでおり、その次はターミネータとして 0 が格納されています。MSX は、SLOT#0 が拡張されていない場合、MAIN-ROM のスロット

MSX Memory Architecture

になります。従って、SLOT#0 がメモリーマッパー対応 RAM の機種は存在しません。このことを利用して、メモリーマッパーサポートルーチンでは、スロット#を指定する場所で、「0」という値をしばしば特別な扱いで使っています。メモリーマッパー情報テーブルの先頭 1byte は、メモリーマッパー対応 RAM のスロット#を示していますが、ここに 0 が入っている場合、複数個連続配置される可能性のあるメモリーマッパー情報テーブルの終わりを示す「ターミネータ」の扱いになっています。

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

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

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

MSX Memory Architecture

図 3-5.に、実際にツールを使って確認した画面を示します。

```
A>mmap
MemoryMapperInformation
=====
Programmed by HRA!
=====
Slot 0x03 (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>■
```

図 3-5. メモリーマッパー情報テーブルダンプツールの表示

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

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

mmap.asm

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

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

        org          0x100

entry::

        ; 初期化
        ld           sp, [TPA_BOTTOM]
        call         mmap_init

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

MSX Memory Architecture

```
        ; マッパーチェック
        ld      ix, [mmap_table_ptr]
        xor     a, a
mapper_check_loop::
        call    dump_one
        jr      c, exit_loop
        ld      de, 8
        add     ix, de
        jr      mapper_check_loop
exit_loop::
        ld      b, 0
        ld      c, D2F_TERM
        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::
        ; 終了チェック
        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
```

MSX Memory Architecture

```

        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
; =====
dec2hex:: scope      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

```


MSX Memory Architecture

```
msg_not_enough_memory::
    ds            "Not enough memory!!"
msg_crlf::
    db            0x0D, 0x0A, 0
msg_separator::
    ds            "=====
    db            0x0D, 0x0A
msg_slot::
    ds            "SLOT      0x"
    db            0
msg_primary_mapper_mark::
    ds            " (Primary)"
    db            0
msg_entry::
    ds            "MemoryMapperInformation"
    db            0x0D, 0x0A
    ds            "=====
    db            0x0D, 0x0A
    ds            "Programmed by HRA!"
    db            0x0D, 0x0A
    db            0x0D, 0x0A, 0

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

4. メガ ROM

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

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

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

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

すでに MSX のゲームカートリッジも貴重な存在になっていますので、メガ ROM カートリッジを自作するとすれば、CPLD や FPGA 等のプログラマブルロジックデバイスを使ってメガコンを新たに

MSX Memory Architecture

作ってしまう方が現実的です。独自仕様のメガコンを作ることも可能ですし、コピーガードのためにそういったメガコンを設計するのも良いかと思います。しかし、そのカートリッジに書き込むゲームなどのソフトウェア開発の段階では、ASCII か Konami の、従来市販品で使われていたメガコンと互換で開発することにより、MSX のエミュレーター上で気楽に動作確認できるメリットがあります。ソフトウェアが組み上がってから、必要に応じて独自仕様メガコンに合わせた修正を行うというのも開発効率面で有利かと思います。もちろん、従来メガコンと互換のメガコンのままカートリッジにしても良いかと思います。

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

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

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

4.1. ASCII-8K タイプ

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

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

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

MSX Memory Architecture

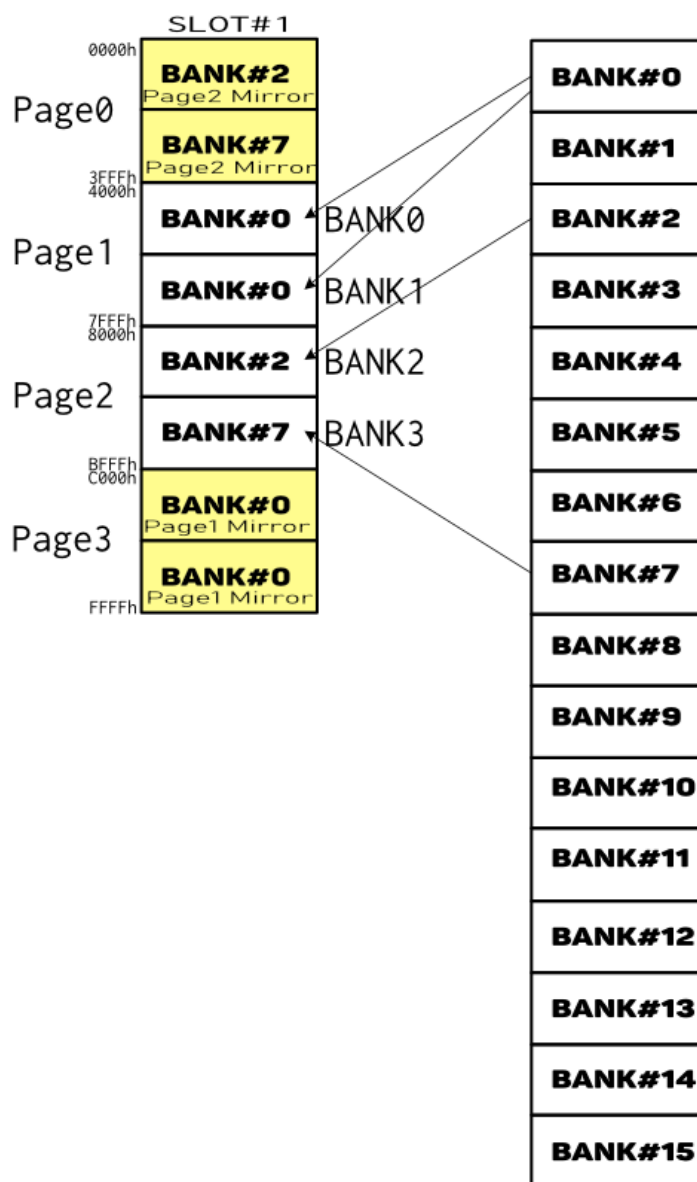


図 4.1-1. ASCII-8K タイプ 1Mbit メガ ROM のイメージ

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

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

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

MSX Memory Architecture

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

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

表 4.1-1.ASCII-8K バンク選択レジスタのアドレス

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

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

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

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

バンク選択レジスタがPage1に集まっている理由ですが、バックアップ用SRAMへの配慮かもしれません。バックアップ用SRAMに対応しているメガコンもあります。一部のバンクにアクセスした場合にのみメガコンを経由した/WEがLになる構造です。SRAMにアクセスする場合にバンク選択

MSX Memory Architecture

レジスタがあると邪魔になるので、SRAM は Page2 に出現させてアクセスすることを想定しているのかもしれませんが。

4.2. ASCII-16K タイプ

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

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

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

MSX Memory Architecture

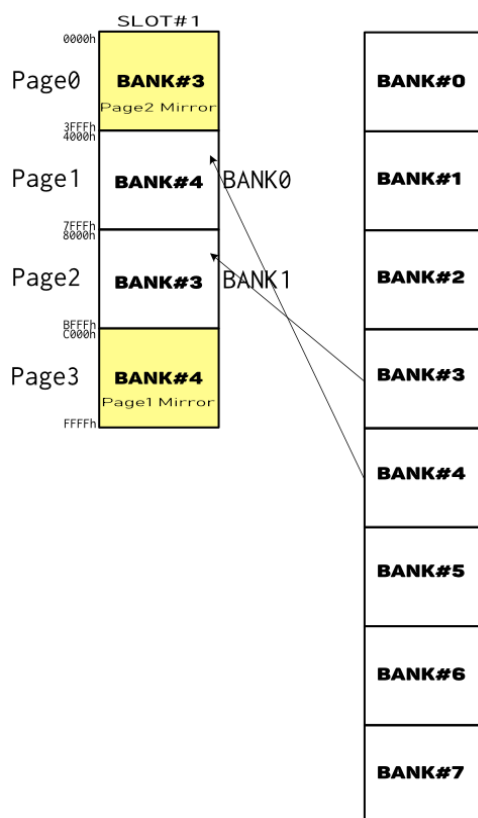


図 4.2-1.ASCII-16K タイプ 1Mbit メガ ROM のイメージ

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

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

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

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

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

表 4.2-1.ASCII-16K バンク選択レジスタのアドレス

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

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

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

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

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

4.3. Konami-8K タイプ

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

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

MSX Memory Architecture

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

表 4.3-1.Konami-8K バンク選択レジスタのアドレス

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

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

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

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

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

4.4. Konami SCC タイプ

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

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

表 4.4-1. Konami-SCC バンク選択レジスタのアドレス

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

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

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

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

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

表 4.4-2. SCC 音源のレジスタ(BANK2-BANK#63)

アドレス	サイズ(byte)	意味
8000h-8FFFh	4096	未使用??
9000h-97FFh	2048	BANK2 のバンク選択レジスタ。実体は 1byte。書き込み専用。
9800h-981Fh	32	ウェーブメモリー 0 (Ch.A 用)、読み書き可能。
9820h-983Fh	32	ウェーブメモリー 1 (Ch.B 用)、読み書き可能。

MSX Memory Architecture

9840h-985Fh	32	ウェーブメモリー 2 (Ch.C 用)、読み書き可能。
9860h-987Fh	32	ウェーブメモリー 3 (Ch.D/E 用)、読み書き可能。
9880h-9881h	2	Ch.A 周波数レジスタ(12bit)、書き込み専用。
9882h-9883h	2	Ch.B 周波数レジスタ(12bit)、書き込み専用。
9884h-9885h	2	Ch.C 周波数レジスタ(12bit)、書き込み専用。
9886h-9887h	2	Ch.D 周波数レジスタ(12bit)、書き込み専用。
9888h-9889h	2	Ch.E 周波数レジスタ(12bit)、書き込み専用。
988Ah	1	Ch.A 音量 (4bit)、書き込み専用。
988Bh	1	Ch.B 音量 (4bit)、書き込み専用。
988Ch	1	Ch.C 音量 (4bit)、書き込み専用。
988Dh	1	Ch.D 音量 (4bit)、書き込み専用。
988Eh	1	Ch.E 音量 (4bit)、書き込み専用。
988Fh	1	出力スイッチ、書き込み専用。 bit0: Ch.A (0: ミュート, 1: 出力) bit1: Ch.B (0: ミュート, 1: 出力) bit2: Ch.C (0: ミュート, 1: 出力) bit3: Ch.D (0: ミュート, 1: 出力) bit4: Ch.E (0: ミュート, 1: 出力)
9890h-989Fh	16	9880h-988Fh のミラー。
98A0h-98BFh	32	未使用。読み書き不可。
98C0h-98DFh	32	未使用??
98E0h-98FFh	32	モードレジスタ。書き込み専用。実体は 1byte。 bit0: 周波数レジスタの扱いの指定 1 bit1: 周波数レジスタの扱いの指定 2 bit5: 周波数レジスタ書き込みで波形ポインタリセット bit6: 全 Ch. 波形ローテート bit7: Ch.D 波形ローテート
9900h-9FFFh	1792	未使用??

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

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

4.5. Konami SCC-I タイプ

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

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

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

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

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

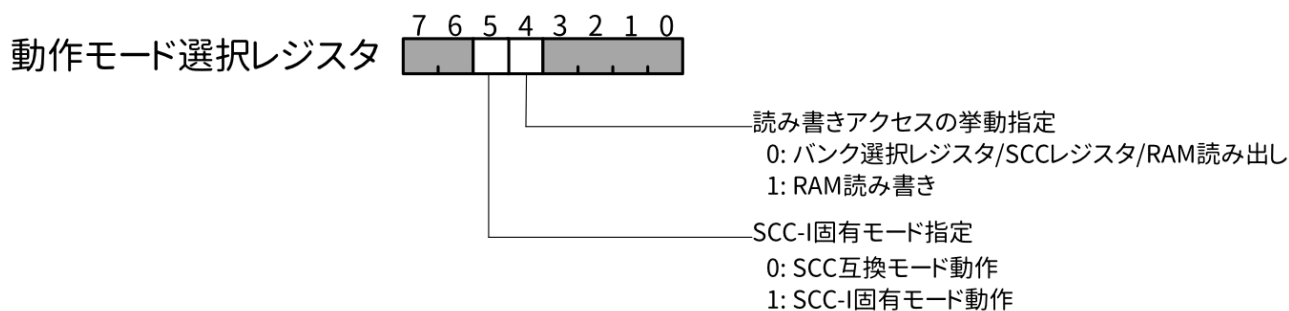


図 4.5-1. SCC-I 動作モード選択レジスタ

MSX Memory Architecture

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

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

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

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

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

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

MSX Memory Architecture

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

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

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

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

次に固有モードの説明に移ります。

MSX Memory Architecture

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

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

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

MSX Memory Architecture

		bit3: Ch.D (0: ミュート, 1: 出力) bit4: Ch.E (0: ミュート, 1: 出力)
B8B0h-B8BFh	16	B8A0h-B8AFh のミラー。
B8C0h-B8DFh	32	モードレジスタ。書き込み専用。実体は 1byte。 bit0: 周波数レジスタの扱いの指定 1 bit1: 周波数レジスタの扱いの指定 2 bit5: 周波数レジスタ書き込みで波形ポインタリセット bit6: 全 Ch. 波形ローテート bit7: 無効
B8E0h-BFFDh	1822	未使用??
BFFEh-BFFFh	2	動作モード選択レジスタ。実体は 1byte。どちらのアドレスも同じ。

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

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

4.6. Panasonic タイプ

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

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

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

スロットから見たイメージを図 4.6-1.に示します。

MSX Memory Architecture

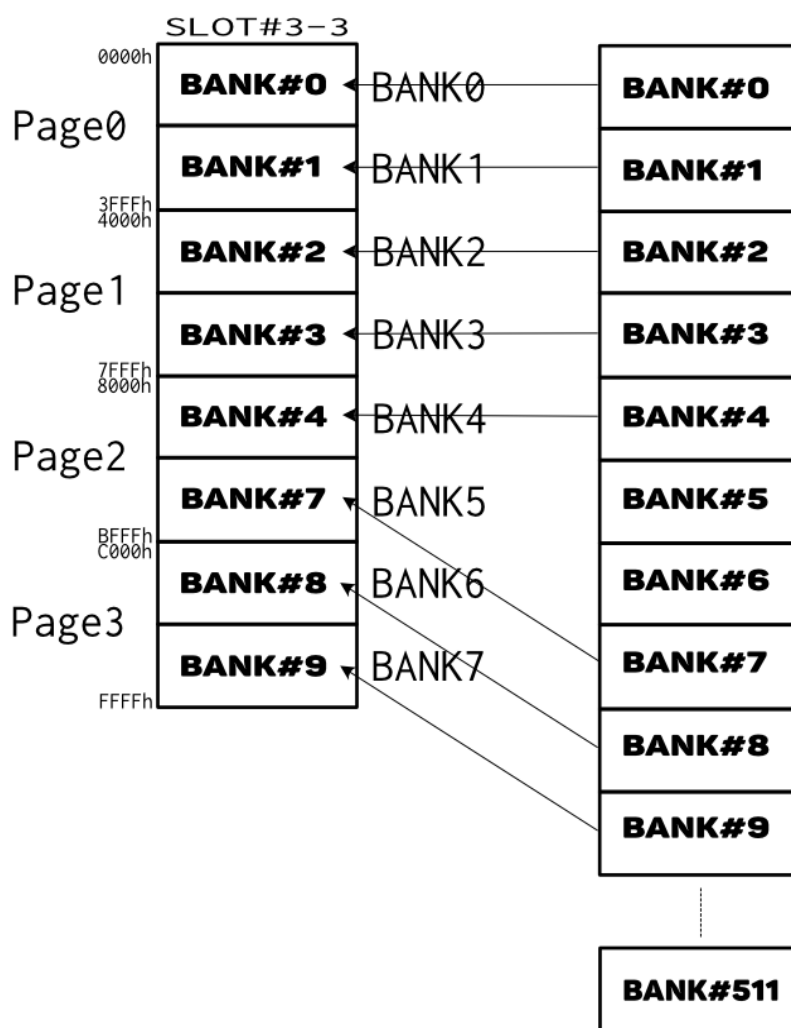


図 4.6-1. Panasonic タイプ メガ ROM のイメージ

バンクレジスタのアドレスを表 4.6-1.に示します。

表 4.6-1. Panasonic タイプ メガ ROM バンクレジスタ

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

MSX Memory Architecture

		BANK2 のバンク選択レジスタ下位 8bit
6C00h-6FFFh	1024	実体は 1byte。書き込み専用。 BANK3 のバンク選択レジスタ下位 8bit
7000h-73FFh	1024	実体は 1byte。書き込み専用。 BANK4 のバンク選択レジスタ下位 8bit
7400h-77FFh	1024	実体は 1byte。書き込み専用。 BANK6 のバンク選択レジスタ下位 8bit
7800h-7BFFh	1024	実体は 1byte。書き込み専用。 BANK5 のバンク選択レジスタ下位 8bit
7C00h-7FEFh	1008	実体は 1byte。書き込み専用。 BANK7 のバンク選択レジスタ下位 8bit
7FF0h	1	読み出し専用。 BANK0 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF1h	1	読み出し専用。 BANK1 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF2h	1	読み出し専用。 BANK2 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF3h	1	読み出し専用。 BANK3 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF4h	1	読み出し専用。 BANK4 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF5h	1	読み出し専用。 BANK5 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF6h	1	読み出し専用。 BANK6 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF7h	1	読み出し専用。

MSX Memory Architecture

		BANK7 のバンク選択レジスタの値下位 8bit。 7FF9h の bit2 が 1 になっていないと読み出し出来ない。
7FF8h	1	読み出し・書き込み可能。 バンク選択レジスタ上位 1bit。 bit0: BANK0 のバンク選択レジスタの上位 1bit。 bit1: BANK1 のバンク選択レジスタの上位 1bit。 bit2: BANK2 のバンク選択レジスタの上位 1bit。 bit3: BANK3 のバンク選択レジスタの上位 1bit。 bit4: BANK4 のバンク選択レジスタの上位 1bit。 bit5: BANK5 のバンク選択レジスタの上位 1bit。 bit6: BANK6 のバンク選択レジスタの上位 1bit。 bit7: BANK7 のバンク選択レジスタの上位 1bit。 7FF9h の bit4 が 1 になっていないと読み書き出来ない。
7FF9h	1	モードレジスタ。読み出し・書き込み可能。 bit0: 無効 bit1: 無効 bit2: 7FF0h-7FF7h 有効化 bit3: 7FF9h 読み出し有効化 bit4: 7FF8h 有効化 bit5: 無効 bit6: 無効 bit7: 無効 このレジスタを読み出すためには、まずこのレジスタの bit3 に 1 を書き込む必要がある。

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

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

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

MSX Memory Architecture

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

分かる範囲で下記にまとめます。<<現在調査中>>

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

turboR のプライマリマッパー RAM は、セグメント#の値が大きい方から順に 4 セグメントが、BIOS コピー領域として確保されています。SLOT#3-0 のプライマリマッパー slots に対しては、R800-DRAM モードではその 4 セグメントが書き込み禁止になるという特徴があります。これは、お

MSX Memory Architecture

そらく ROM カートリッジソフトや、MSX-DOS1 用のメモリーマッパー対応アプリケーションが、誤って BIOS コピーを上書きしてしまわないように保護しているのだと思います。

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

4.7. パナアミューズメントカートリッジ

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

表 4.7-1. PAC の SRAM 切り替え

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

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

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

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

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

MSX Memory Architecture

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

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

5. ROM カートリッジの動作

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

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

表 5.1. ROM ヘッダ

アドレス	サイズ(byte)	名前
------	-----------	----

MSX Memory Architecture

+0000h	2	ID
+0002h	2	INIT
+0004h	2	STATEMENT
+0006h	2	DEVICE
+0008h	2	TEXT
+000Ah	6	RESERVED

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

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

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

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

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

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

5.1. 16KB の ROM カートリッジプログラムを作る

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

ROM_sample001.ASM

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

MSX Memory Architecture

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

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

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

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
```

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

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

MSX Memory Architecture

アセンブルすると、ROM_sample001.ROM という 16KB の ROM イメージファイルが出来上がります。エミュレータで起動できますので、動かしてみてください。図 5.1.のように表示されれば期待通りの動作となります。

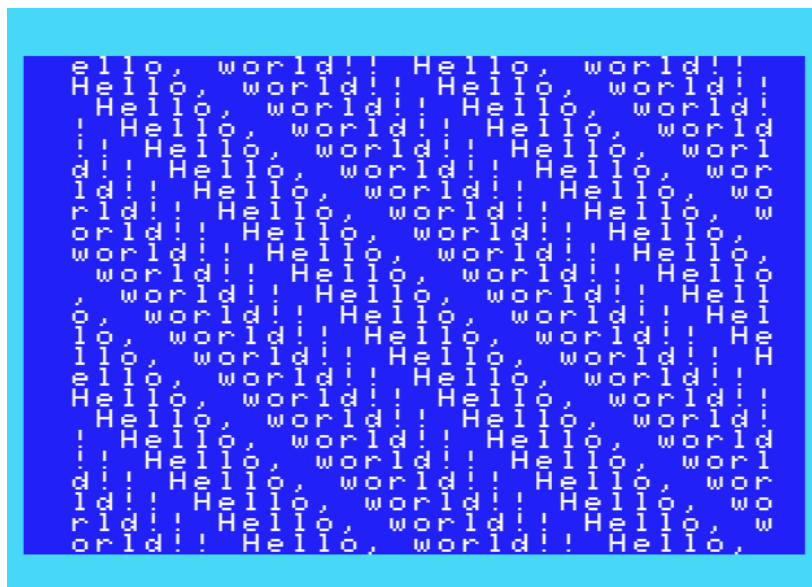


図 5.1. ROM sapmle001.ROM の実行イメージ

では、この ROM_sample001.ASM について、順に説明してきたいと思います。

まず、冒頭です。

org	0x4000
-----	--------

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

16KB の ROM は、アドレス信号線として A13~A0 の 14bit の信号を持っているはずです。これをカートリッジスロットの A13~A0 にそのまま接続して、A15・A14 は未接続、ROM の/CS にはカートリッジスロットの/SLTSL を接続、ROM の/OE にはカートリッジスロットの/RD を接続でカートリッジを作る場合、Z80 からは Page0, Page1, Page2, Page3 のすべてにこの ROM の内容

MSX Memory Architecture

が現れる構成になります。この中で BIOS が一番最初に調べるのが Page1 なので、Page1 に置かれている前提のプログラムにします。Page1 の先頭が 0x4000 なので、org 0x4000 にしてあります。

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

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

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

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

その他は、今回使わないので 0 で埋めておきます。

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

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

MSX Memory Architecture

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

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

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

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

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

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

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

MSX Memory Architecture

main_loop のところで、HL に代入されている display_message というラベルは、ここにあります。

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

ターミネータは、先ほどのサブルーチン puts が解釈するものです。

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

最後に align 命令です。ROM_sample001.ASM は 16KB に満たない小さなプログラムです。これを 16KB になるようにパディングを追加する疑似命令です。

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

Page2 は、機種によって異なります。

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

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

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

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

RAM かどうかを調べる方法としては、調べたいアドレスから値を読み出してレジスタに保存。同じアドレスに反転した値を書き込んで、もう一度読み出して反転した値を読み出せば RAM、読み出せ

MSX Memory Architecture

なければ RAM でない（ROM や未接続など）、RAM だった場合は保存していた値を書き戻す。という方法があります。BIOS が RAM の存在確認する際もそのようにしているようです。

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

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

ディスク BIOS のワークエリアとして、Page0～Page3 に対応する RAM の SLOT# を格納した RAMAD0～RAMAD3 という領域があります。しかし、ROM カートリッジが起動するタイミングでは、まだこの値は格納されていません。ディスク BIOS 自体も、ROM カートリッジと同じ実装であり、ディスク内蔵機では SLOT#3-X に搭載されている機種が殆どです。ディスク BIOS の初期化ルーチンが呼ばれる前なので、格納されていないわけです。一方で、外付けドライブを SLOT#1 に、作った ROM カートリッジを SLOT#2 に装着した場合は格納されているかもしれません。そのため、SP の初期化は HIMEM を参照して、存在しているかもしれないディスク BIOS 等他のモジュールのワークエリアを破壊しないようにするのが安全です。ゲームなどをするときは不要なデバイスを外して下さい、というお約束は、こういった特殊ケースによる不慮の事故を未然に防ぐための予防策なのです。

5.2. メガ ROM カートリッジプログラムを作る

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

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

INIT から起動するのは、5.1. 16KB の ROM カートリッジプログラムを作ると同じですが、Page1 に現れている「メガ ROM の BANK0 と BANK1」には、ともに BANK#0 が出現しています。メガ ROM のスロットの Page2 にある BANK2 と BANK3 も BANK#0 が出現していますが、Z80

MSX Memory Architecture

メモリ空間には、メガ ROM のスロットの Page2 は出現していません。その点に注意してプログラムする必要がある点が、5.1. 16KB の ROM カートリッジプログラムを作ると大きく異なる点です。

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

ROM_sample002.ASM

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

chput      = 0x00A2
himem      = 0xfc4a

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

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

                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
```

MSX Memory Architecture

```
        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
; =====
; =====
```

MSX Memory Architecture

```
org            0x6000
display_message4:
    ds         "BANK#3 on BANK1"
    db         0x0D, 0x0A, 0

    align      8192

; =====
; =====
;      Padding
; =====
; =====
; =====
    ds         "Padding"

    align      131072
```

では、順番に説明したいと思います。ROM_sample001.ASM と同じ部分に関しては、説明を省略させていただきます。

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

大きく変わっているのは、main_loop ですが、最初のメッセージ表示に関しては ROM_sample001.ASM と同じです。

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

次に、bank1_sel ですが、プログラムの上の方で 0x6800 と宣言されています。つまり、ASCII-8K の BANK1 のバンク選択レジスタのアドレスになります。ここに 1 を書き込んでいるので、BANK1 が BANK#1 に切り替わります。このイメージを図 5.2-1. に示します。

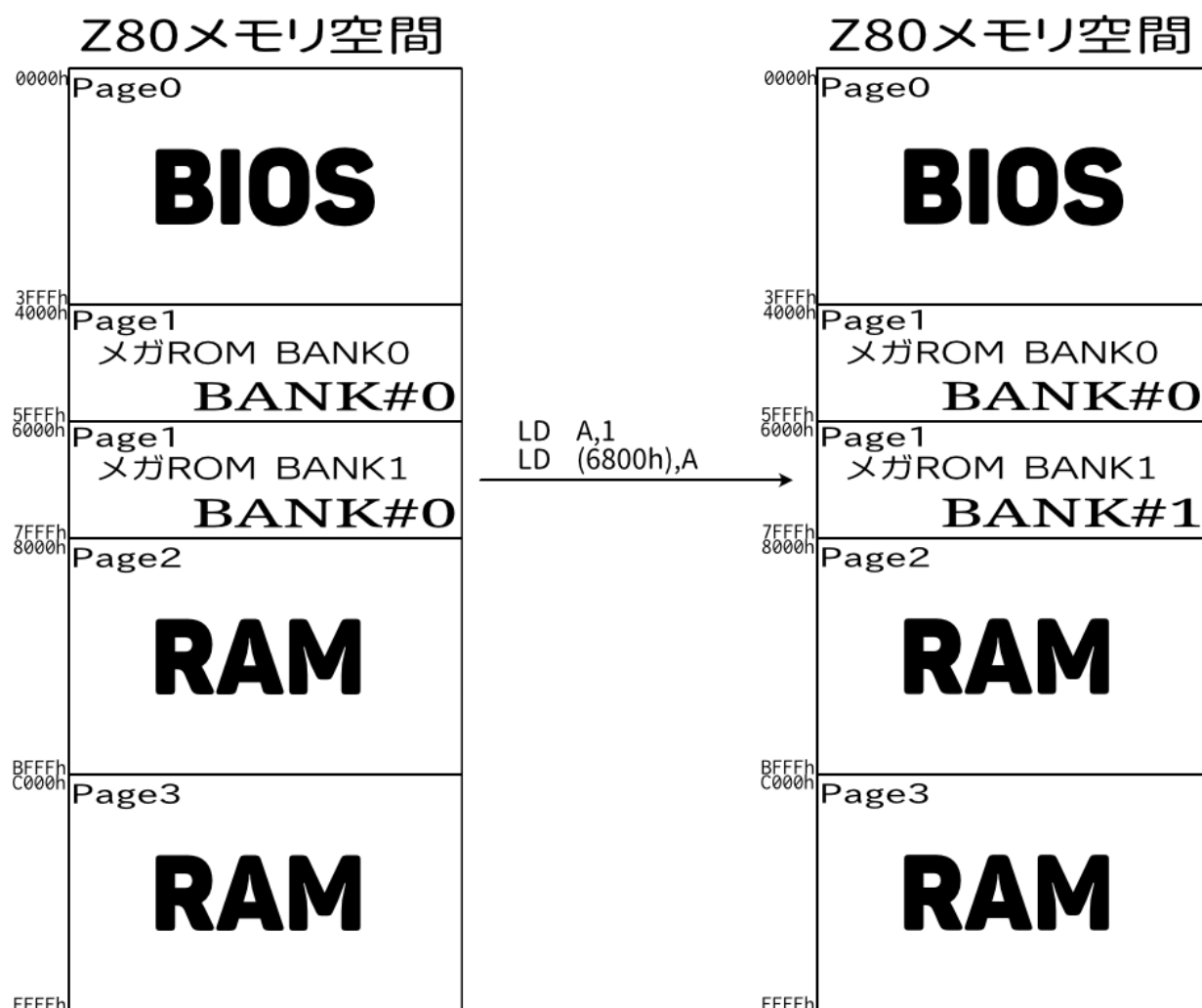


図 5.2-1. BANK1 が BANK#1 に切り替わるイメージ

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

display_message2 は、BANK1 に出現させた BANK#1 上の文字列のアドレスになります。call puts でこれを表示しています。

```

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

```


MSX Memory Architecture

```
ld      a, 3                      ; BANK#3
ld      [bank1_sel], a           ; BANK1 changes to BANK#3
ld      hl, display_message4
call    puts
jp      main_loop
```

こちらも同様に、**BANK1** のバンク選択レジスタに対して、ここに **2** を書き込んで **BANK1** を **BANK#2** に切り替えて、**BANK#2** にある `display_message3` を表示。次に **3** を書き込んで **BANK1** を **BANK#3** に切り替えて、**BANK#3** にある `display_message4` を表示しています。 `jp main_loop` で上に戻って繰り返します。

この動作イメージを、図 5.2-2. に示します。

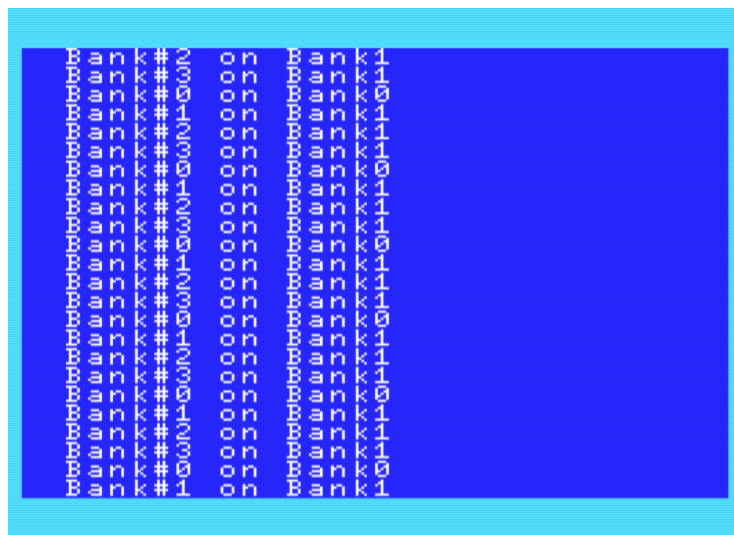


図 5.2-2. ROM sample002.ASM の動作イメージ

5.3. 自身のスロットの検出

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

残念ながら、BIOS には現在のスロット#を得るルーチンは用意されていないので、調べるプログラムを用意しなければ、自身がどのスロットにいるのかわかりません。

MSX Memory Architecture

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

ROM_sample003.ASM

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

MSX Memory Architecture

```
        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
```

MSX Memory Architecture

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

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

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

MSX Memory Architecture

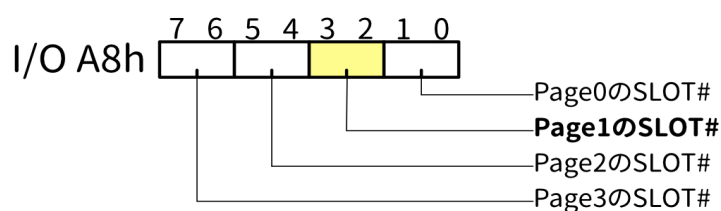


図 5.3-1. 基本スロット選択レジスタ

```
push    af

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

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

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

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

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

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

MSX Memory Architecture

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

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

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

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

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

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

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

MSX Memory Architecture

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

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

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

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

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

5.4. Page2/Page3 のスロット検出

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

MSX Memory Architecture

ROM_sample004.ASM

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

chput      = 0x00A2
himem      = 0xfc4a

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

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

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

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

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

finish:
    jp        finish

; =====
```


MSX Memory Architecture

```
; 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::
```

MSX Memory Architecture

```
        ; 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
```

MSX Memory Architecture

```
        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, '-'
```

MSX Memory Architecture

```

        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
```

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

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

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

5.5. Page0/Page3 のスロット切り替え

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

Page3 は、ENASLT で切り替えることが出来ますが、スタックポインタが Page3 にある状態で ENASLT を呼ぶと暴走しますのでご注意ください。ENASLT 内ではスタックを利用していますので、必ず

MSX Memory Architecture

Page3 以外の Page ヘスタックポインタを移動してから呼び出す必要があります。また、BIOS ワークエリアがある領域なので、EI するタイミングにも注意が必要です。割り込みが入ると呼び出されるフックも BIOS ワークエリアに存在するので、適切なケアをしていない状態で切り替えたまま EI すると、割り込みが入った段階で暴走します。

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

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

ROM_sample005.ASM

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

enaslt      = 0x0024
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
```

MSX Memory Architecture

```
        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
```

MSX Memory Architecture

```

        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:
```

MSX Memory Architecture

```
        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)
```


MSX Memory Architecture

```
;          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#
```

MSX Memory Architecture

```
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; CCBAA00 : 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
```

MSX Memory Architecture

```
        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 の EXPTBL
        ld      de, slot_info      ; SLOT#0-0 の slot_info
        ld      b, 0               ; SLOT#0 から表示開始

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

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

        ld      a, b              ; Aレジスタに SLOT# を復元
        push    de
        push    bc
        push    af
        ld      hl, message_slot   ; "SLOT#" を表示
        call    puts
        pop     af
        call    put_slot_no        ; SLOT# を表示
        pop     bc
        push    bc
        ld      hl, message_ram
        dec     c                  ; slot_info が 1 なら "RAM", 2 なら "Non-
```

MSX Memory Architecture

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

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

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

これまでに出てきたサンプルと共通の部分の説明は省略します。

MSX Memory Architecture

Page0 のスロットを切り替えるルーチンは、local_enaslt0 になります。Page0 を「A レジスタで指定したスロット#」に切り替えるルーチンです。

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

まず、B レジスタにスロット#を保存しておきます。

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

二の補数表現の場合、負数なら bit7 = 1、正数かゼロなら bit7 = 0 です。そのため、Sf を見ることで拡張スロットの有無を見ることが出来ます。拡張スロットでない場合は、not_expanded ヘジャンプします。

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

6. ハードウェア

7. その他の補足事項

本章では、6 章までに挙げられなかった補足事項について記載します。

7.1. バージョンアップアダプター利用時の EXPTBL

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

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

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

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

SLOT#0 及び SLOT#1 が拡張されておらず、MSX2 版 MAIN-ROM が SLOT#1 に装着されている場合を考えてみます。FCC1h には 81h が格納されているため、「MAIN-ROM へ切り替えたいプログラム」は、FCC1h の値を A レジスタに入れて ENASLT を呼びます。すると、ENASLT は、I/O

MSX Memory Architecture

A8h を使って SLOT#1 に切り替えた後に、その FFFFh に書き込みます。拡張スロットであれば FFFFh は拡張スロットレジスタが存在しますが、実際は SLOT#1 は拡張されていないので、この FFFFh への書き込みは消滅します。なぜならば、SLOT#1 の MSX2 版 MAIN-ROM カートリッジの FFFFh へ書き込みに行ったことになるためです。MAIN-ROM カートリッジは Page0, Page1 に ROM が存在するのみで、Page2, Page3 は何も繋がっていません。そのため FFFFh への書き込みアクセスは、何も書き込まれずに消滅します。

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

7.2. 拡張スロット選択レジスタは無条件に書いてはダメ

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

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

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

MSX Memory Architecture

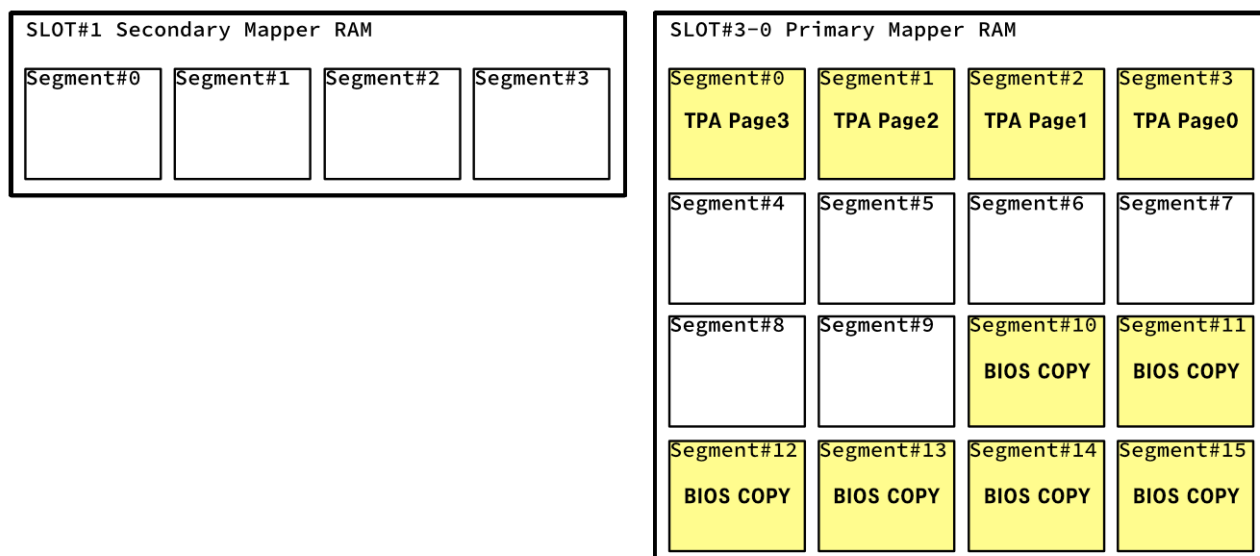


図 7.2-1. FS-A1ST 起動時のメモリーマップー使用状況

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

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

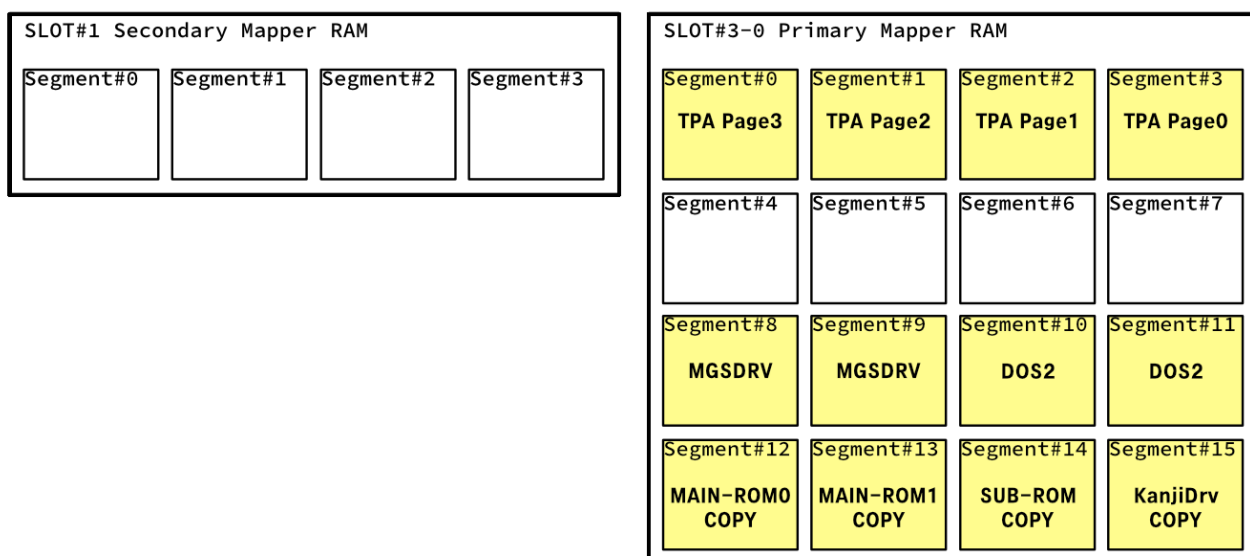


図 7.2-2. MGSDRV 常驻後のメモリーマップー使用状況

MSX Memory Architecture

この状態から、MGSP を起動させると、MGSP は常駐している MGSDRV は利用できませんので、MGSDRV 用のメモリを新たに使用してしまいます。これに 2 セグメント使います。MGSP はユーザーセグメントで使用するため、セグメント#の小さい方から割り当てられます。さらに、漢字フォント用に 4 セグメント使いますが、プライマリマッパーには 2 セグメントしか空きがありません。漢字フォントは、セカンダリーマッパーにも対応していますが、複数のメモリーマッパー対応 RAM をまたがる確保には対応していないため、すべてセカンダリーマッパーの方へ追いやられます。結果として、図 7.2-3.のような使用状況になります。

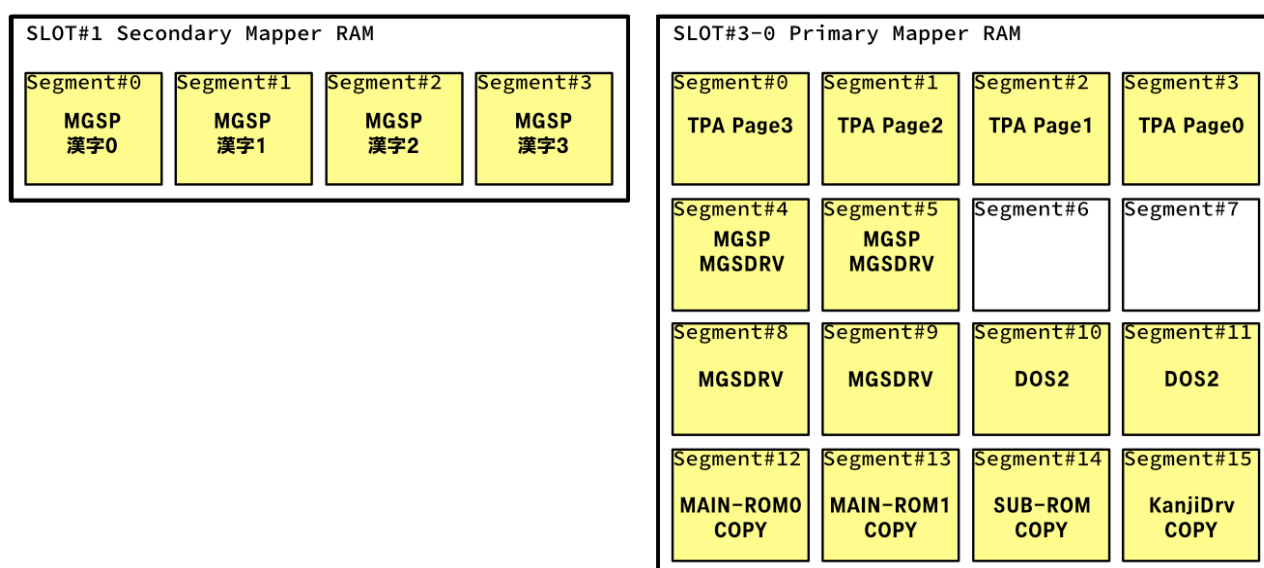


図 7.2-3. MGSP 動作中のメモリ確保状況

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

MSX Memory Architecture

	SLOT#0-0	SLOT#0-1	SLOT#0-2	SLOT#0-3	SLOT#1	SLOT#2	SLOT#3-0	SLOT#3-1	SLOT#3-2	SLOT#3-3
Page0 0000h 3FFFh 4000h	MAIN-ROM				MAPPER RAM 64KB	CARTRIDGE SLOT #2	MAPPER RAM 256KB	SUB-ROM		PANASONIC MEGA-ROM
Page1 3FFFh 4000h 7FFFh 8000h	MAIN-ROM		MSX-MUSIC	???	MAPPER RAM 64KB	CARTRIDGE SLOT #2	MAPPER RAM 256KB	KANJI DRIVER	DISK-BIOS	PANASONIC MEGA-ROM
Page2 7FFFh 8000h BFFFh C000h					MAPPER RAM 64KB	CARTRIDGE SLOT #2	MAPPER RAM 256KB	KANJI DRIVER		PANASONIC MEGA-ROM
Page3 BFFFh C000h FFFFh					MAPPER RAM 64KB	CARTRIDGE SLOT #2	MAPPER RAM 256KB			PANASONIC MEGA-ROM

図 7.2-4. FS-A1ST + RAM64KB

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

Page0 0000h 3FFFh 4000h	SLOT#3-0 MAPPER RAM 256KB Segment#3
Page1 3FFFh 4000h 7FFFh 8000h	SLOT#3-0 MAPPER RAM 256KB Segment#2
Page2 7FFFh 8000h BFFFh C000h	SLOT#3-0 MAPPER RAM 256KB Segment#1
Page3 BFFFh C000h FFFFh	SLOT#3-0 MAPPER RAM 256KB Segment#0

図 7.2-5. DOS2 起動時の Z80 メモリ空間

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

MGSP は、漢字フォントを SLOT#1 の RAM64KB に格納しているので、Page2 を SLOT#1 に切り替えて漢字フォントにアクセスします。この「SLOT#1 に切り替える」という処理を、ENASLT を使わずに MGSP 内で独自に実装していたと仮定して、その独自のスロット切り替えルーチン内で「ス

MSX Memory Architecture

ロットが拡張されているか否かに関わらず、拡張スロット選択レジスタが存在するであろう FFFFh に書き込む」という実装になっていたらどうなるのでしょうか？

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

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

ここで、上記 FS-A1ST + RAM64KB で MGSP を動かした例に話を戻しますと、拡張されていない SLOT#1 の FFFFh に対して値を書き込むと、そこに出現しているメモリーマッパー対応 RAM64KB カートリッジの Segment#0 の先頭から +3FFFh の位置にその値が書き込まれてしまいます。そこには漢字フォントが存在していますので、漢字フォントが 1byte 破壊されてしまうことになります。

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

付録.

付録 1. サンプルプログラムから利用している各種ソース

msxbios.asm

chkram	:=	0x0000
synchr	:=	0x0008
rdslt	:=	0x000C
chrgrtr	:=	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
gpsiz	:=	0x008a
grpprt	:=	0x008d
gicini	:=	0x0090
wrtpsg	:=	0x0093
rdpsg	:=	0x0096
strtms	:=	0x0099
chsns	:=	0x009c
chget	:=	0x009f
chput	:=	0x00a2
lptout	:=	0x00a5

MSX Memory Architecture

lptstt	:=	0x00a8	
cnvchr	:=	0x00ab	
pinlin	:=	0x00ae	
inlin	:=	0x00b1	
qinlin	:=	0x00b4	
breakx	:=	0x00b7	
beep	:=	0x00c0	
cls	:=	0x00c3	
posit	:=	0x00c6	
fnksb	:=	0x00c9	
erafnk	:=	0x00cc	
dspfkn	:=	0x00cf	
totext	:=	0x00d2	
gtstck	:=	0x00d5	
gttrig	:=	0x00d8	
gtpad	:=	0x00db	; note: MSXturboR ではライトペン (A=8~11) は常に 0 が返る
gtpdl	:=	0x00de	; note: MSXturboR では常に 0 が返る
tapion	:=	0x00e1	; note: MSXturboR では常に Cy=1 (エラー) が返る
tapin	:=	0x00e4	; note: MSXturboR では常に Cy=1 (エラー) が返る
tapiof	:=	0x00e7	; note: MSXturboR では常に Cy=1 (エラー) が返る
tapoon	:=	0x00ea	; note: MSXturboR では常に Cy=1 (エラー) が返る
tapout	:=	0x00ed	; note: MSXturboR では常に Cy=1 (エラー) が返る
tapoof	:=	0x00f0	; note: MSXturboR では常に Cy=1 (エラー) が返る
stmotr	:=	0x00f3	; note: MSXturboR では何もせずに返る
chgcap	:=	0x0132	
chgsnd	:=	0x0135	
rslreg	:=	0x0138	
wslreg	:=	0x013b	
rdvdp	:=	0x013e	
snsmat	:=	0x0141	
isflio	:=	0x014a	
outdlp	:=	0x014d	
kilbuf	:=	0x0156	
calbas	:=	0x0159	
; require MSX2			
subrom	:=	0x015c	
extrom	:=	0x015f	
eol	:=	0x0168	
bigfil	:=	0x016b	
nsetrd	:=	0x016e	
nstwrt	:=	0x0171	
nrdvrm	:=	0x0174	
nwrvm	:=	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	

MSX Memory Architecture

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

MSX Memory Architecture

rg2sav	:=	0xF3E1	; 1byte, VDP R#2 に書き込んだ値
rg3sav	:=	0xF3E2	; 1byte, VDP R#3 に書き込んだ値
rg4sav	:=	0xF3E3	; 1byte, VDP R#4 に書き込んだ値
rg5sav	:=	0xF3E4	; 1byte, VDP R#5 に書き込んだ値
rg6sav	:=	0xF3E5	; 1byte, VDP R#6 に書き込んだ値
rg7sav	:=	0xF3E6	; 1byte, VDP R#7 に書き込んだ値
statfl	:=	0xF3E7	; 1byte, VDP S#0 から読み込んだ値
fnkstr	:=	0xF87F	; 16byte * 10, ファンクションキーに対応する文字列
exbrsa	:=	0xFAF8	; 1byte, SUB-ROM のスロット#
hokvld	:=	0xFB20	
oldkey	:=	0xFBDA	; 11bytes, キーマトリクスの状態(旧)
newkey	:=	0xFBE5	; 11bytes, キーマトリクスの状態(新)
keybuf	:=	0xFBF0	; 40bytes, キーバッファ
linwrk	:=	0xFC18	; 40bytes, スクリーンハンドラが使う一時保管場所
patwrk	:=	0xFC40	; 8bytes, パターンコンバータが使う一時保管場所
bottom	:=	0xFC48	; 2bytes, 実装した RAM の先頭番地。MSX2 では通常 8000h
himem	:=	0xFC4A	; 2bytes, 利用可能なメモリの上位番地。CLEAR 文の<メモリ上限>
により設定される			
trptbl	:=	0xFC4C	; 78bytes, 割り込み処理で使うトラップテーブル。一つのテーブル
は 3bytes で 1バイト目が ON/OFF/STOP の状態。残りが分岐先テキストアドレス			
scrmod	:=	0xFCAF	; 1byte, 現在のスクリーンモード番号
oldscr	:=	0xFCB0	; 1byte, スクリーンモード保存エリア
exptbl	:=	0xFCC1	; 4bytes, 各スロットの拡張の有無
slttbl	:=	0xFCC5	; 4bytes, 各スロットの拡張スロット選択レジスタの現在の選択状況
sltatr	:=	0xFCC9	; 64bytes, 各スロット用の属性
sltwrk	:=	0xFD09	; 128bytes, 各スロット用の特定のワークエリアを確保
procnm	:=	0xFD89	; 16bytes, 拡張ステートメント, 拡張デバイスの名前が入る,
ASCIIIZ			
device	:=	0xFD99	; 1byte, カートリッジ用の装置識別
h_timi	:=	0xFD9F	; 5bytes, 垂直同期割り込み発生時のフック
extbio	:=	0xFFCA	
rg8sav	:=	0xFFE7	
rg9sav	:=	0xFFE8	

msxdos1.asm

```

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

; =====
;   Terminate program
;   input)
;       C = D1F_TERM0
;   output)
;       --
;

```


MSX Memory Architecture

```
; =====
D1F_TERM0                := 0x00

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

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

D1F_AUXIN                 := 0x03
D1F_AUXOUT                := 0x04
D1F_LSTOUT                := 0x05
D1F_DIRIO                 := 0x06
D1F_DIRIN                 := 0x07
D1F_INNOE                 := 0x08
D1F_STROUT                := 0x09
D1F_BUFIN                 := 0x0A
D1F_CONST                 := 0x0B
D1F_CPMVER                := 0x0C
D1F_DSKRST                := 0x0D
D1F_SELDISK               := 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
```

MSX Memory Architecture

```
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_WERR           := 0xFE        ; Write error
D2E_DISK           := 0xFD        ; Disk error
D2E_NRDY           := 0xFC        ; Not ready
D2E_VERIFY         := 0xFB        ; Verify error
D2E_DATA           := 0xFA        ; Data error
D2E_RNF            := 0xF9        ; Sector not found
D2E_WPROT          := 0xF8        ; Write protected disk
D2E_UFORM          := 0xF7        ; Unformatted disk
D2E_NDOS           := 0xF6        ; Not a DOS disk
D2E_WDISK          := 0xF5        ; Wrong disk
```

MSX Memory Architecture

```
D2E_WFILE      := 0xF4      ; Wrong disk for file
D2E_SEEK       := 0xF3      ; Seek error
D2E_IFAT       := 0xF2      ; Bad file allocation table
D2E_NOUPB      := 0xF1      ; -
D2E_IFORM      := 0xF0      ; Cannot format this drive
D2E_SUCCESS    := 0x00      ; -

; -----
;   find first
;   input)
;       C = D2F_FFIRST
;       DE = Directory path name address (ASCIIIZ) or FIB address
;       B = Search target attribute
;           bit0: read only
;           bit1: hidden
;           bit2: system file
;           bit3: volume label
;           bit4: directory
;           bit5: archive
;           bit6: RESERVED
;           bit7: device
;       IX = Result area address (64bytes)
;   output)
;       A = Error code
;       [IX] = FIB of the file
D2F_FFIRST     := 0x40

; -----
;   find next
;   input)
;       C = D2F_FNEXT
;       IX = Result area address of D2F_FFIRST
;   output)
;       A = Error code
;       [IX] = FIB of the file
D2F_FNEXT      := 0x41

; -----
;   find new entry
;   input)
;       C = D2F_FNEW
;       DE = Directory path name address (ASCIIIZ) or FIB address
;       B = Search target attribute
;           bit0: read only
;           bit1: hidden
;           bit2: system file
;           bit3: volume label
;           bit4: directory
;           bit5: archive
;           bit6: RESERVED
;           bit7: new creation flag
;       IX = FIB with template
;   output)
;       A = Error code
;       [IX] = FIB of the file
D2F_FNEW       := 0x42
```

MSX Memory Architecture

```
; -----  
;  
;   open  
;   input)  
;  
;       C = D2F_OPEN  
;  
;       DE = File path name address (ASCIIIZ) or FIB address  
;  
;       A = Open mode  
;  
;           bit0: Disable write access  
;  
;           bit1: Disable read access  
;  
;           bit2: Succession  
;  
;           others: always 0  
;  
;   output)  
;  
;       A = Error code  
;  
;       B = New file handle  
D2F_OPEN          := 0x43  
  
D2F_CREATE        := 0x44  
D2F_CLOSE         := 0x45  
D2F_ENSURE        := 0x46  
D2F_DUP           := 0x47  
D2F_READ          := 0x48  
D2F_WRITE         := 0x49  
D2F_SEEK          := 0x4A  
D2F_IOCTL         := 0x4B  
D2F_HTEST         := 0x4C  
D2F_DELETE        := 0x4D  
D2F_RENAME        := 0x4E  
D2F_MOVE          := 0x4F  
D2F_ATTR          := 0x50  
D2F_FTIME         := 0x51  
D2F_HDELETE       := 0x52  
D2F_HRENAME       := 0x53  
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
```

MSX Memory Architecture

D2F_DSKCHK	:= 0x6E
D2F_DOSVER	:= 0x6F
D2F_REDIR	:= 0x70

stdio.asm

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

mmapper.asm

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

; =====
;      mmap_init
;      input)
;      --
;      output)
;      Zf ..... 1: MemoryMapper サポートルーチンが存在しない
;      [mmap_table_ptr] ... マッパーテーブルのアドレス
;      break)
;      all
```

MSX Memory Architecture

```
; =====
; scope mmap_init
mmap_init::
    ld      c, D2F_DOSVER      ; DOS バージョンをチェック
    call    bdos
    or      a, a                ; A != 0 は DOS ではない
    jp      nz, mmap_error_exit
    ld      a, b
    cp      a, 2                ; B < 2 の場合 MSX-DOS1 である
    jp      c, mmap_error_exit

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

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

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

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

    xor     a, a                ; A=0
    inc     a                    ; A=1, Zf=0
    ret     z                    ; 正常終了 (Zf=0)
mmap_error_exit:
    xor     a, a                ; Zf=1 にする
    ret     z                    ; エラー終了 (Zf=1)
endscope

; =====
;   スロット構成を TPA に戻す
; =====
; scope mmap_change_to_tpa
mmap_change_to_tpa::
    ; change slot of Page1
    ld      h, 0x40
    ld      a, [ramad1]
    call    enaslt

    ; change slot of page2
```

MSX Memory Architecture

```
        ld        h, 0x80
        ld        a, [ramad2]
        call      enaslt

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

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

; =====
;          WORKAREA
; =====
mmap_table_ptr::
        dw        0          ; マッパーテーブルのアドレスが格納される
mapper_segment_p1:
        db        0          ; 起動時の page1 のセグメント#
mapper_segment_p2:
        db        0          ; 起動時の page2 のセグメント#

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
```

MSX Memory Architecture

db	0xc9, 0xc9, 0xc9
----	------------------

MSX Memory Architecture

注意

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

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

【参考文献】

テクハン wiki 化計画

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

MSX Datapack wiki 化計画

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

Gigamix Online メガ ROM データベース

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

メガコン解析資料

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

msx.org MegaROM Mappers

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

Konami Sound Cartridge - SCC+

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

ひろゆきのホームページ SCC 実験室

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

ASCAT home page テクガイ本文全編

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

Special thanks to:

MSX Memory Architecture

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

2021 年 2 月 19 日 HRA!