

MSX Documents

VDP

目次

1. はじめに.....	6
2. VDP の種類.....	7
2. VDP の基本的な動作.....	8
2.1. VDP へのアクセス概要.....	9
2.1.1. MSX-BASIC からのアクセス概要.....	9
2.1.2. 機械語からのアクセス概要.....	9
2.2. 各種スクリーンモード.....	10
2.2.1. SCREEN1 (GRAPHIC1).....	10
2.2.2. SCREEN0:WIDTH 40 (TEXT1).....	20
2.2.3. SCREEN0:WIDTH 80 (TEXT2).....	26
2.2.4. SCREEN2 (GRAPHIC2).....	30
2.2.5. SCREEN3 (MULTI COLOR).....	39
2.2.6. SCREEN4 (GRAPHIC3).....	48
2.2.7. SCREEN5 (GRAPHIC4).....	51
2.2.8. SCREEN6 (GRAPHIC5).....	54
2.2.9. SCREEN7 (GRAPHIC6).....	56
2.2.10. SCREEN8 (GRAPHIC7).....	58
2.2.11. SCREEN9.....	60
2.2.12. SCREEN10 (GRAPHIC7, YJK+RGB mode).....	61
2.2.13. SCREEN11 (GRAPHIC7, YJK+RGB mode).....	63
2.2.14. SCREEN12 (GRAPHIC7, YJK mode).....	64
2.3. VDP へのアクセス詳細.....	69
2.3.1. I/O 制御によるアクセス.....	69
2.3.1.1. コントロールレジスタライト.....	71
2.3.1.2. ステータスレジスタリード.....	74
2.3.1.3. パレットレジスタ.....	79
2.3.1.4. VRAM への書き込み.....	84
2.3.1.5. VRAM からの読み出し.....	90
2.3.1.6. 間接的なコントロールレジスタへの書き込み(インクリメントあり).....	91
2.3.1.7. 間接的なコントロールレジスタへの書き込み(インクリメントなし).....	93
2.3.2. MSX-BASIC からのアクセス.....	94
2.4. VDP のレジスタ.....	99
2.4.1. コントロールレジスタ.....	100

2.4.1.1. Mode Register 0 (R#0).....	101
2.4.1.2. Mode Register 1 (R#1).....	103
2.4.1.3. Pattern Name Table Address (R#2).....	108
2.4.1.4. Color Table Base Address Register Low (R#3).....	112
2.4.1.5. Pattern Generator Table Base Address Register (R#4).....	114
2.4.1.6. Sprite Attribute Table Base Address Register Low (R#5).....	116
2.4.1.7. Sprite Pattern Generator Table Base Address Register (R#6).....	118
2.4.1.8. Text Color/Back Drop Color Register (R#7).....	121
2.4.1.9. Mode Register 2 (R#8).....	127
2.4.1.10. Mode Register 3 (R#9).....	130
2.4.1.11. Color Table Base Address Register High (R#10).....	134
2.4.1.12. Sprite Attribute Table Base Address Register High (R#11).....	137
2.4.1.13. Text Color/Back Color Register (R#12).....	138
2.4.1.14. Blinking Period Register (R#13).....	139
2.4.1.15. VRAM Access Base Address Register (R#14).....	140
2.4.1.16. Status Register Pointer (R#15).....	141
2.4.1.17. Color Palette Address Register (R#16).....	142
2.4.1.18. Register Pointer (R#17).....	143
2.4.1.19. Display Adjust Register (R#18).....	144
2.4.1.20. Interrupt Line Register (R#19).....	146
2.4.1.21. Color Burst Register 1 (R#20).....	166
2.4.1.22. Color Burst Register 2 (R#21).....	166
2.4.1.23. Color Burst Register 3 (R#22).....	166
2.4.1.24. Display Offset Register (R#23).....	167
2.4.1.25. N/A (R#24).....	169
2.4.1.26. Mode Register 4 (R#25).....	170
2.4.1.27. Horizontal Scroll Register High (R#26).....	171
2.4.1.28. Horizontal Scroll Register Low (R#27).....	172
2.4.1.29. VDP コマンド制御レジスタ (R#32～R#46).....	173
2.4.2. ステータスレジスタ.....	174
2.4.2.1. Status Register 0 (S#0).....	175
2.4.2.2. Status Register 1 (S#1).....	176
2.4.2.3. Status Register 2 (S#2).....	177
2.4.2.4. Column Register Low (S#3).....	178

2.4.2.5. Column Register High (S#4).....	179
2.4.2.6. Row Register Low (S#5).....	180
2.4.2.7. Row Register High (S#6).....	181
2.4.2.8. Color Register (S#7).....	182
2.4.2.9. Border Register Low (S#8).....	183
2.4.2.10. Border Register High (S#9).....	184
2.5. スプライト.....	185
2.5.1. スプライトモード 1.....	187
2.5.2. スプライトモード 2.....	192
(1) 水平最大 8 枚並べられる.....	192
(2) スプライトプレーンにはライン単位で色を付けられる.....	198
(3) ライン単位で重ね合わせの指定が出来る.....	202
(4) ライン単位で衝突判定の有無を指定できる.....	205
2.6. VDP コマンド.....	205
2.6.1. HMMC (High-speed Move CPU to VRAM).....	211
2.6.2. YMMM (High-speed Move VRAM to VRAM, y only).....	222
2.6.3. HMMM (High-speed move VRAM to VRAM).....	231
2.6.4. HMMV (High-speed move VDP to VRAM).....	239
2.6.5. LMMC (Logical move CPU to VRAM).....	243
2.6.6. LMCM (Logical move VRAM to CPU).....	244
2.6.7. LMMM (Logical move VRAM to VRAM).....	245
2.6.8. LMMV (Logical move VDP to VRAM).....	246
2.6.9. LINE.....	247
2.6.10. SRCH.....	248
2.6.11. PSET.....	249
2.6.12. POINT.....	250
2.6.13. STOP.....	251
3. 解析.....	252
3.1. SCREEN6 以下と SCREEN7 以上の DRAM 使用方法について.....	253
3.2. RGB 出力とコンポジット出力.....	256
3.3. 192 ライン/212 ラインモード.....	259
3.4. 存在しないレジスタへのアクセス.....	259
3.5. アドレスオートインクリメントのカウンタの挙動.....	261
3.6. 透明スプライトの衝突判定.....	264

4. 逆引	266
4.1. アドレス指定	266
4.2. サンプルプログラム	266
付録. VRAM マップ	267
SCREEN0 (WIDTH40) : TEXT1: TMS9918/V9938/V9958	268
SCREEN0 (WIDTH80) : TEXT2: V9938/V9958	269
SCREEN1 : GRAPHIC1: TMS9918/V9938/V9958	270
SCREEN2 : GRAPHIC2: TMS9918/V9938/V9958	271
SCREEN3 : MULTI COLOR: TMS9918/V9938/V9958	272
SCREEN4 : GRAPHIC3: V9938/V9958	273
SCREEN5, 6 : GRAPHIC4, 5: V9938/V9958	274
SCREEN7, 8 : GRAPHIC6, 7: V9938(VRAM128KB 以上)/V9958	275
SCREEN10,11,12 : GRAPHIC7: V9958	276
付録. コントロールレジスタ	277
付録. ステータスレジスタ	286
付録. 参考文献	288

1. はじめに

本書は、MSX における VDP (Video Display Porcessor) の使い方を、なるべく分かりやすくまとめたつむりの資料です。

下記の条件を満たしていることを前提として記述しています。

- (1) MSX-BASIC で簡単なプログラムを組める
- (2) Z80 のアセンブラプログラムが理解できる
- (3) プログラミングの基礎的な部分が理解できており、簡単なゲームソフトくらいなら自作できる

この条件を満たすためのレクチャーは、基本的に省略させていただきます。

また、説明の便宜上、特に注意書きのない場合は MSX1/2/2+ を想定して記述してあります。MSXturboR はやや特殊になるので、turboR 固有の情報はその旨明示します。

MSX 全般を示す場合は MSX、各バージョンを区別する場合は MSX1・MSX2・MSX2+・MSXturboR と記述します。

本書では、サンプルプログラムを多数用意しています。MSX-BASIC で記述可能なものは、MSX-BASIC で作成していますが、一部アセンブラで記述する必要のあるサンプルについては、私が作成したアセンブラ ZMA 向けの記述とさせていただきます。ザイログニーモニックと一部異なる部分がありますが、大きな違いは無いので中身の理解の妨げにはならないかと思います。

ZMA はフリーソフトとして下記の URL で公開しています。

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

2. VDP の種類

VDP にはいくつかの種類があります。各 MSX には表 2.1. の VDP が搭載されています。

表 2.1. VDP の種類

MSX	VDP
MSX1	TMS9918相当
MSX2	V9938
MSX2+	V9958
MSXturboR	V9958

TMS9918 は、NTSC 出力・PAL 出力の違いや、接続する DRAM の種類の違いなどいろいろな型番があります。また、TI 社製ではない互換品もありますが、本資料では TMS9918 と表記させていただきます。また、TMS9918/V9938/V9958 をまとめて記述する場合は VDP と表記します。

MSX1 を MSX2 相当にバージョンアップする MSX バージョンアップアダプターという製品がありました。これは、カートリッジの中に V9938 を搭載して大幅な表示の機能アップを実現しています。ただし、内蔵の TMS9918 と I/O アドレスが衝突しないように、異なる I/O アドレスにマッピングされています。これについては、後述の I/O アクセスのところで詳しく説明したいと思います。

2. VDP の基本的な動作

VDP がどのような仕組みで、画像をモニターへ出力しているのか軽く説明したいと思います。

VDP には CPU から直接アクセスできない DRAM が接続されており、この DRAM に格納されている情報に基づいてモニターへ出力する信号を決定しています。この DRAM のことを VRAM と呼びます。

従ってこの VRAM を書き換えると、画面表示を変化させることができます。標準の VRAM マップは、付録. VRAM マップにまとめておきます。

MSX-BASIC からは、VPOKE 命令を使うことによって、この VRAM を書き換えられます。画面に表示される背景画像・スプライトなどはすべてこの VRAM に格納されている情報に基づいて表示されるため、VRAM を適切に書き換えてあげれば、表示を変更することができます。

VDP には、コントロールレジスタ・ステータスレジスタ・パレットレジスタと呼ばれる内部レジスタが存在します。VRAM に記録していない情報は、すべてこれらレジスタの中に記録されています。MSX-BASIC では VDP(n) システム変数というかたちで読み書きできるようになっていますが、実際の VDP のレジスタ類はコントロールレジスタ・パレットレジスタは書き込み専用、ステータスレジスタは読み出し専用のレジスタになっています。BASIC から書き込み専用のレジスタが読み出せるのは、書き込み時に DRAM 上に書き込んだ値を保存しておいて、読み出し時にはその DRAM 上の値を返すことで読み出しているように見せかけています。具体的なレジスタの制御方法については、後述します。ここでは「VRAM 以外にも挙動を決める値を保持している場所がある」程度に理解して頂ければ問題ありません。

MSX の VDP は、画面モードによって VRAM の構成が大きく変わります。そのため、各画面ごとに個別に説明したいと思います。

2.1. VDP へのアクセス概要

CPUとVDPとVRAMの接続関係のイメージを図2.1.1.に示します。

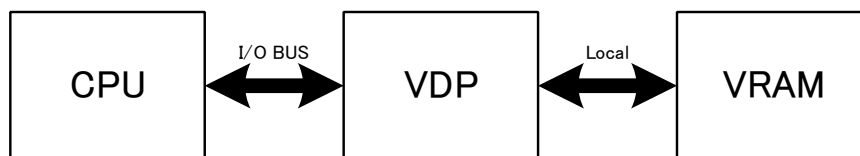


図 2.1.1. CPU/VDP/VRAM 接続関係

CPU から VDP には I/O BUS で接続されています。VRAM は VDP 側に接続されており、CPU から VRAM へアクセスするには VDP を介してアクセスすることになります。従って、CPU から VRAM にアクセスする際も I/O BUS 使用することになります。

2.1.1. MSX-BASIC からのアクセス概要

MSX-BASIC からは、VRAM への書き込みには VPOKE 命令、VRAM からの読み出しは VPEEK()関数を利用してアクセスできます。CPU メモリへのアクセスの POKE 命令・PEEK()関数と名前が似ていますが、POKE・PEEK は CPU が直接アクセスできるメモリ、VPOKE・VPEEK は I/O BUS を用いて VDP を介した VRAM へのアクセスであり、処理が全く異なりますのでご注意ください。

また、BASE()システム変数・VDP()システム変数を使って、VDP の各レジスタへアクセスできます。

これらシステム変数については、各レジスタの説明の際に合わせて説明します。

2.1.2. 機械語からのアクセス概要

BIOS が用意している各種 VDP 制御ルーチンを利用するのが最も簡単です。

一方で、OUT 命令・IN 命令を使って直接制御することも可能です。

処理速度がそれほど必要ない場合は、BIOS を使ったアクセスが簡単かつ安全です。しかしながら、VDP 関連は処理速度を要求されることも多いので、OUT 命令・IN 命令による直接制御が許されています。

2.2. 各種スクリーンモード

2.2.1. SCREEN1 (GRAPHIC1)

MSX-BASIC で SCREEN1 と呼ばれているモードです。

VDP の仕様書では GRAPHIC1 という名前が付けられています。

VRAM を構成する要素を、表 2.2.1.1. にまとめました。

表 2.2.1.1. SCREEN1 の VRAM 要素

VRAMを構成する要素	概要
Pattern Generator Table	PCGの形状を定義する領域
Pattern Name Table	画面上のPCG配置を定義する領域
Sprite Attribute Table	スプライトの表示位置・色・パターン番号を定義する領域
Color Table	PCGの色を定義する領域
Palette Table	カラーパレットの値を保持する領域 (MSX2以降。MSX-BASICのみ)
Sprite Generator Table	スプライトの形状を定義する領域

MSX Documents VDP

SCREEN1 は、8ドット×8ドットを1パーツとして、256 種類のパーツを定義できます。これを画面上に32個×24個敷き詰めることで画面を構成しています。画面には必ずこの 256 種類のうちどれかが表示されている状態です。画面上の位置に対して、256 種類のパーツのどれが表示されているかの情報を格納しているのが Pattern Name Table になります。これは初期値では 1800h~1AFFh になっていて、1byte で 1 パーツを表現しています。画面の位置とアドレスの対応関係を図 2.2.1.1.に示します。

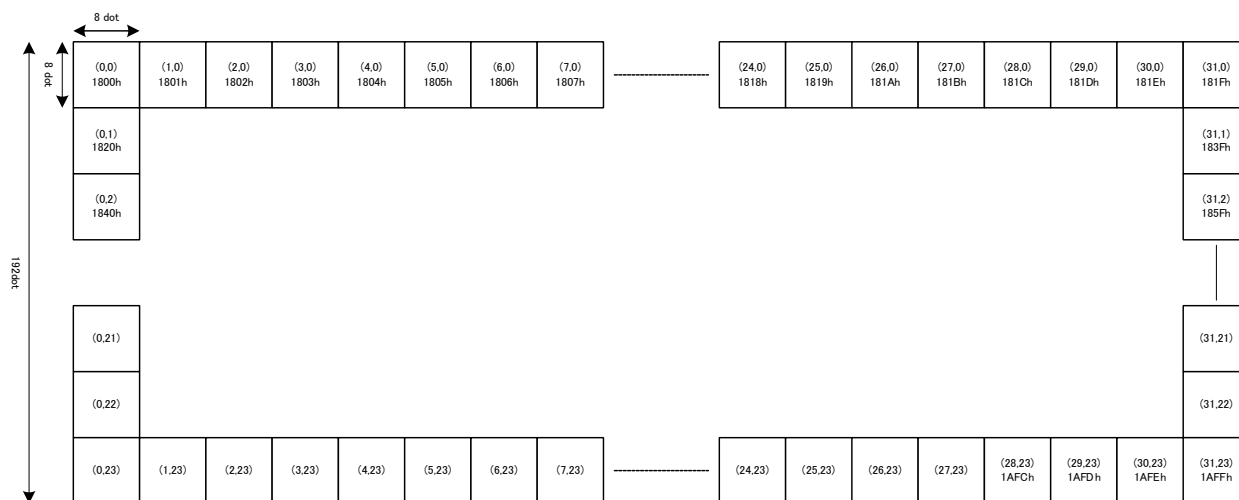


図 2.2.1.1. SCREEN1 の Pattern Name Table とアドレスの関係

左上隅を (0,0) として、X 座標は右が正、Y 座標は下が正の向きだとすると、(x,y) の座標に相当するパーツのアドレスは、 $1800h + x + y \times 32$ で求められます。つまり、(0,0) の場合は 1800h になるわけです。

試しに、MSX-BASIC で下記のように入力して、最後に [RETURN] を押してみてください。

```
SCREEN1:VPOKE&H1800,ASC("A"):LOCATE0,5
```

MSX Documents VDP

入力すると写真 2.2.1.1. のような状態。[RETURN] を押すと写真 2.2.1.2. のような状態になります。

VRAM の 1800h 番地に A の文字コードを書き込むことで、無事 “A” を表示できたわけです。

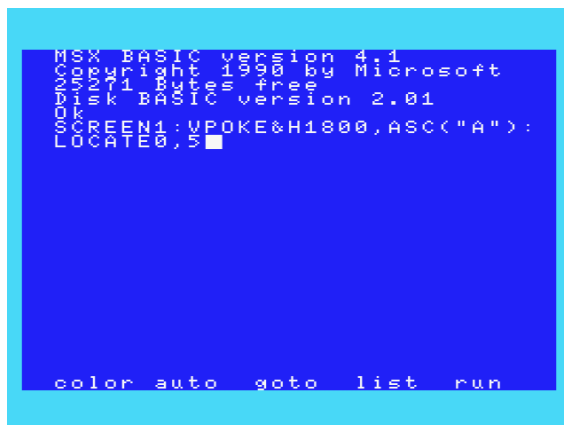


写真 2.2.1.1. Pattern Name Table への書き込みプログラム



写真 2.2.1.2. 左上隅に A が表示される

MSX Documents VDP

写真 2.2.1.2. では、A と OK とファンクションキーの表示以外、何も表示されていないように見えますが、ここには 32 番のパーツが表示されています。それを確認するために、続けて PRINT VPEEK(&H1801) を実行してみましょう。&H1801 は、先ほど A のパーツを表示したところのすぐ右隣の位置に対応します。その値を読み出して番号として表示してみる命令です。

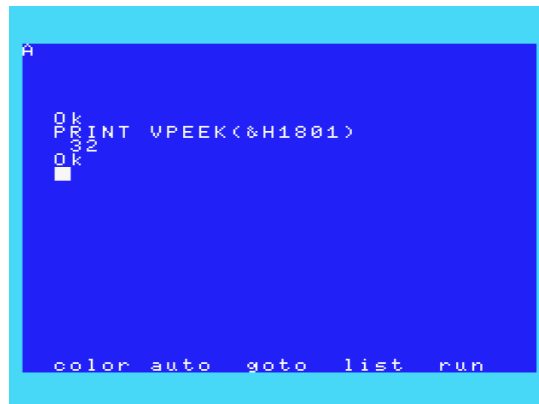


写真 2.2.1.3. A の隣のパーツ番号

32 番はスペース記号のパーツに対応しています。PRINT “(“+CHR\$(32)+”)” で確認できますね。

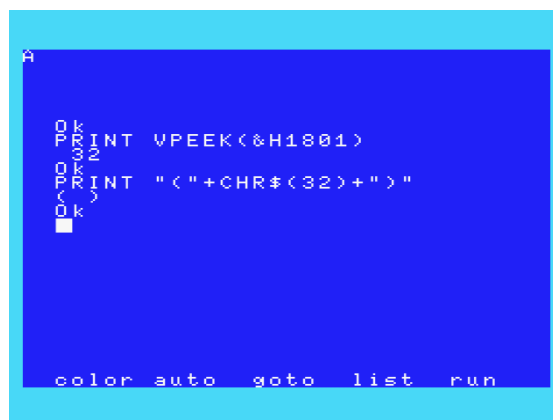



写真 2.2.1.4. 32 番のパーツを確認

MSX Documents VDP

念のため、&H1800 も同様に確認してみましょう。



```
OK
PRINT VPEEK(&H1801)
OK
PRINT "<" + CHR$(32) + ">"
OK
PRINT VPEEK(&H1800)
OK
PRINT "<" + CHR$(65) + ">"
OK
color auto goto list run
```

写真 2.2.1.5. &H1800 のパーツ番号のパーツを確認

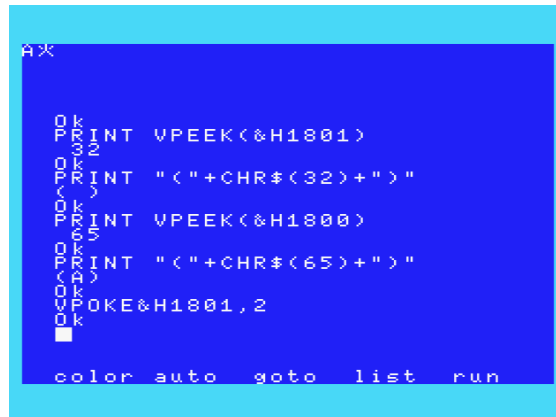
このように、MSX-BASIC のキャラクターコードと、VDP のパーツ番号は一致しています。

しかし、0 番～31 番のパーツ番号は、MSX-BASIC のキャラクターコードとは異なります。

MSX-BASIC のキャラクターコードの 0 番～31 番は、コントロールコードと呼ばれる文字ではない機能が割り当てられています。例えば、13 番はカーソルを左端へ移動するキャリッジリターンという機能が割り当てられています。カーソル自体が、MSX-BASIC 中のソフトウェアが作り出している表現のうちの一つで、VDP の機能ではないので、コントロールコードは MSX-BASIC というソフトウェアを制御するための特別な文字であることが分かります。PRINT 命令がそのキャラクターコードを表示するように指示されると、Pattern Name Table の書き換えではなく、MSX-BASIC 内のカーソルを示す制御を行うように条件判断しているのです。

MSX Documents VDP

VDP の Pattern Name Table に格納されている値は、あくまで表示パーツの番号であるため、当然ながら 0 番～31 番のパーツも存在しています。MSX-BASIC では、ここにグラフィックキャラクタを割り当てています。試しに、2 番のパーツを表示させてみましょう。

A screenshot of an MSX-BASIC terminal window with a blue background and a light blue border. The terminal shows the following commands and their outputs:

```
Ok  
PRINT VPEEK(&H1801)  
32  
Ok  
PRINT "<"+CHR$(32)+">"  
<  
Ok  
PRINT VPEEK(&H1800)  
65  
Ok  
PRINT "<"+CHR$(65)+">"  
<A>  
Ok  
VPOKE&H1801,2  
Ok  
■
```

At the bottom of the terminal, there is a status bar with the text "color auto goto list run".

写真 2.2.1.6. &H1801 に 2 番のパーツを表示

グラフィックキャラクタの一つである“火”がでてきました。

MSX-BASIC の WIDTH 命令は、BASIC の LOCATE 座標位置をずらしたり、PRINT の表示範囲を狭めたりする、いわば MSX-BASIC 用の制御命令です。Pattern Name Table は WIDTH 命令に影響されないのでご注意ください。上記の例にわかるように、OK の表示より左側に "A" や "火" が表示されています。

MSX Documents VDP

Pattern Generator Table は、このパーツ1つ1つの形状定義が格納されている領域です。Pattern Generator Table には、1bit = 1ドットに対応して、bit が 0 なら背景、bit が 1 なら前景の扱いで、下位 bit が右、上位 bit が左で水平 8ドットを 1byte に格納しています。各パーツは 8byte で構成されていて、パーツの上のラインから 8 ライン分。1byte は水平 8ドットで、8x8ドットのパーツ形状を定義しています。Pattern Generator Table の先頭アドレスからパーツ番号*8 のアドレスが、そのパーツの定義情報に該当します。試しに、MSX-BASIC で“A” のパーツ形状の定義をダンプしてみましょう。各ビットがドットに対応しているので2進数でダンプすると分かりやすいです。

```
FOR I=0 TO 7:PRINT RIGHT$("0000000",
0)+BIN$(VPEEK(ASC("A")*8+I)),
8>:NEXT I
0001000000
0100100000
1000100000
1000100000
1111100000
1000100000
1000100000
0000000000
Ok
■

color auto goto list run
```

写真 2.2.1.7. パーツ“A”の形状定義をダンプ

MSX Documents VDP

写真 2.2.1.7.に示したように、背景が 0、前景が1となって“Ａ”の形をかたどっているのを確認できます。

これを書き換えてしまえば、“A”の形を変えられます。

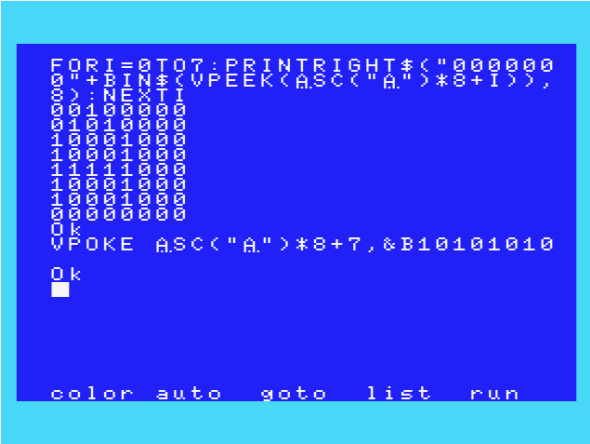


写真 2.2.1.8. パーツ“Ａ”の下端を書き換えてみる

“Ａ”の形状の下端に 10101010 に対応する模様が出てきているのを確認できます。

このように、各パーツには好きな形状を設定できます。プログラムを書く上で、数字やアルファベットの記号が定義されていた方が都合が良いので、MSX-BASIC は初期化時にそれらを Pattern Generator Table に書き込んでいるわけですが、Pattern Generator Table を書き換えることでオリジナルのキャラクタを表示させたり、文字フォント形状を変更したりすることができるわけです。

図 2.2.1.2.にパターンの格納イメージを示します。白い部分が前景(ビットが 1 の部分)、青い部分が背景(ビットが 0 の部分)になっています。

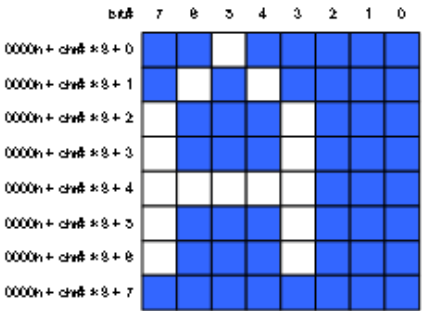


図 2.2.1.2. パターン形状の格納イメージ

MSX Documents VDP

【note】

デフォルトのフォントの右3ドットがすべて背景になっていて隙間を空けすぎでは?と思った方も居るかもしれませんが、これは SCREEN0 とフォントデータを共通にするためです。

SCREEN0 では、水平 40 文字に増える代わりに、1 パーツが水平 6 ドットに減ります。右 2 ドットが欠けてしまうのです。水平 8 ドットで、右端 1 列しか背景にしていない場合、SCREEN1 では見やすい大きなフォントになりますが、一方で SCREEN0 では右が 1 ライン欠けた上に、右のパーツとくっついてしまって読みづらくなるわけです。

SCREEN0 用と SCREEN1 用のフォントを両方持つと ROM を無駄に消費してしまうこともあり、幅が狭い方に合わせたフォントになっているわけです。

次に Color Table ですが、これはパーツの色情報を格納したテーブルになります。

SCREEN1 では、連続するパーツ番号の 8 パーツが同じ色になるという制約があります。256 パーツあるので、 $256/8 = 32$ 種類。Color Table の 1byte 目は、パーツ#0~#7 の 8 パーツの色。2byte 目は、パーツ#8~#15 の色。3byte 目は、パーツ#16~#23 の色。・・・32byte 目は、パーツ#248~#255 の色。を指定しています。その中に格納されている値は、図 2.2.1.3. にしめすビットマップになっています。下位 4bit で背景色の番号 0~15 を、上位 4bit で前景色の番号 0~15 を指定しています。

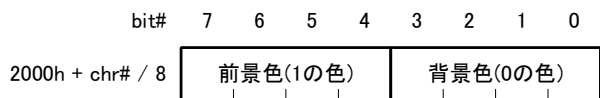


図 2.2.1.3. パーツの色指定

例によって MSX-BASIC で試してみましょう。

```
MSX BASIC version 2.1
Copyright 1990 by Microsoft
25271 Bytes free
Disk BASIC version 2.1
Ok

0123456789
VPOKE&H2000+ASC(">")/8,&HC9
Ok

color auto goto list run
```

写真 2.2.1.9. カラーテーブルを書き換えてみる

写真 2.2.1.9.の例では、“0”のパーツに色を付けています。前景色として &HC (濃い緑), 背景色として &H9 (オレンジ) を指定しています。Color Table の中の 1byte を書き換えると、連続する番号の 8 パーツの色が一緒に変化してしまうため、“0”のパーツだけでなく、“1”～“7” まで同じ色になっているのが確認できます。“0”～“7” で 1 つの組なのです。

Palette Table ですが、これは MSX-BASIC version 2.0 から追加になっており、V9938/V9958 のカラーパレット変更命令 COLOR=(パレット番号, R, G, B) を実行した際に、パレットの色の変更とともに、変更した値を Palette Table へ書き出しています。つまり、VDP の機能ではないのです。VDP の機能では無いため、Palette Table を VPOKE 命令で書き換えても、BLOAD”ファイル名”,S でディスクから読み込んでも、パレットは変化しません。Palette Table の内容を実際のパレットに反映させたい場合は、COLOR=RESTORE を実行することになります。VDP がカラーパレットをどのように扱っているかは、別途 2.3. VDP へのアクセス詳細にて説明します。

Sprite Attribute Table, Sprite Generator Table に関しては、後述のスプライトにて説明します。

2.2.2. SCREEN0:WIDTH 40 (TEXT1)

MSX-BASIC で SCREEN0 と呼ばれているモードです。

VDP の仕様書では TEXT1 という名前が付けられています。

VRAM を構成する要素を、表 2.2.2.1. にまとめました。

表 2.2.2.1. SCREEN0(WIDTH40)の VRAM 要素

VRAMを構成する要素	概要
Pattern Generator Table	PCGの形状を定義する領域
Pattern Name Table	画面上のPCG配置を定義する領域
Palette Table	カラーパレットの値を保持する領域 (MSX2以降。MSX-BASICのみ)

各要素の意味は、SCREEN1 の同名要素と同じ意味になります。

ご覧の通り、スプライト関連のテーブルがありません。このモードではスプライトを表示することができません。

Pattern Generator Table に パーツの形状情報を格納しているのは SCREEN1 と同じですが、SCREEN0(WIDTH40)では水平 6ドット×垂直 8ドットのパーツになります。1 バイト＝1 ライン、1ドット 1ビットであることには違いありませんが、1 バイトのうち下位 2ビットは無効である点が SCREEN1と異なります。図 2.2.2.1-1.に SCREEN0(WIDTH40)の Pattern Generator Table の内容イメージを示します。

MSX Documents VDP

	bit#	7	6	5	4	3	2	1	0
0800h + chr# * 8 + 0		■	■	□	■	■	■	■	■
0800h + chr# * 8 + 1		■	□	■	□	■	■	■	■
0800h + chr# * 8 + 2		□	■	■	■	□	■	■	■
0800h + chr# * 8 + 3		□	■	■	■	□	■	■	■
0800h + chr# * 8 + 4		□	□	□	□	□	■	■	■
0800h + chr# * 8 + 5		□	■	■	■	□	■	■	■
0800h + chr# * 8 + 6		□	■	■	■	□	■	■	■
0800h + chr# * 8 + 7		■	■	■	■	■	■	■	■

非表示

図 2.2.2.1-1. Pattern Generator Table の格納イメージ

右 2 ドットが欠けていることを下記のプログラム(SC0PGT.BAS)で確認してみます。

```

100 SCREEN0:WIDTH40:COLOR15,4,7:DEFINT A-Z
110 A=&H800+ASC("A")*8:D=0
120 FOR I=0 TO 7
130 D=(D/2)+&B100000000
140 VPOKE A+I,D
150 NEXT
160 A=&H800+ASC("B")*8:D=0
170 FOR I=0 TO 7
180 D=(D*2)+&B000000001
190 VPOKE A+I,D
200 NEXT
210 PRINT "ABABABAB"
220 END

```

このプログラムを実行した結果を、写真 2.2.2.1-1. に示します。

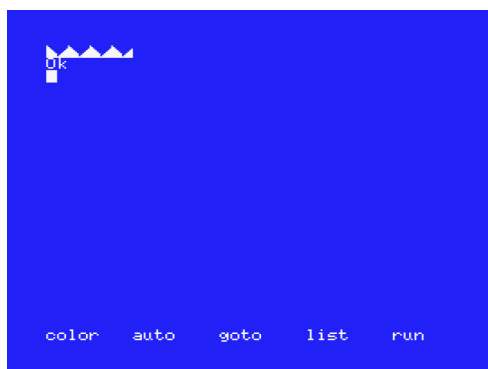


写真 2.2.2.1-1. SCREEN0(WIDTH40)で右 2 ドット欠け確認

MSX Documents VDP

AとBに図 2.2.2.1-2.の形状を定義して、ABABABABと表示していますが、右 2ドットが欠けているのでいびつな形になっているのが確認できます。

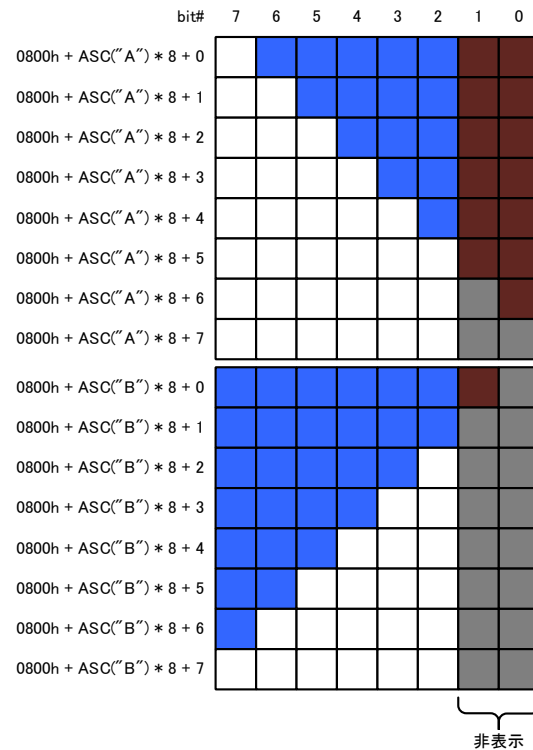


図 2.2.2.1-2. AとBに定義した形状

写真 2.2.2.1-1.の ABABABAB 表示部分を拡大したものを写真 2.2.2.1-2. に示します。

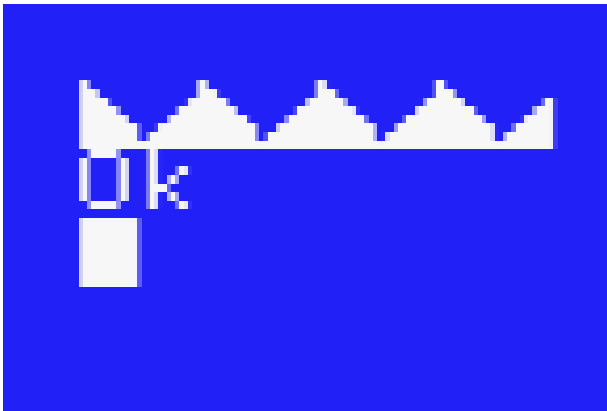


写真 2.2.2.1-2. 部分拡大

このように、Pattern Generator Table に格納する値の下位 2 ビットは、画面には何の影響も及ぼしません。VRAM としては存在しているので、値自体は記憶しています。

MSX Documents VDP

PCG の色は、VRAM ではなくコントロールレジスタ R#7 で全文字に対して一括して指定します。そのため、画面全体で前景と背景の 2 色しか使えません。R#7 の上位 4bit に前景色番号、下位 4bit に背景色番号を指定します。例えば、前景色に 1(黒)=&H1、背景色に 12(濃い緑)=&HC を指定するには写真 2.2.2.1-3.のようになります。



写真 2.2.2.1-3. 前景 1, 背景 12 に色変更

この色情報は VRAM には格納されていません。コントロールレジスタ R#7 のみです。

Pattern Name Table は図 2.2.2.1-3. に示しました。

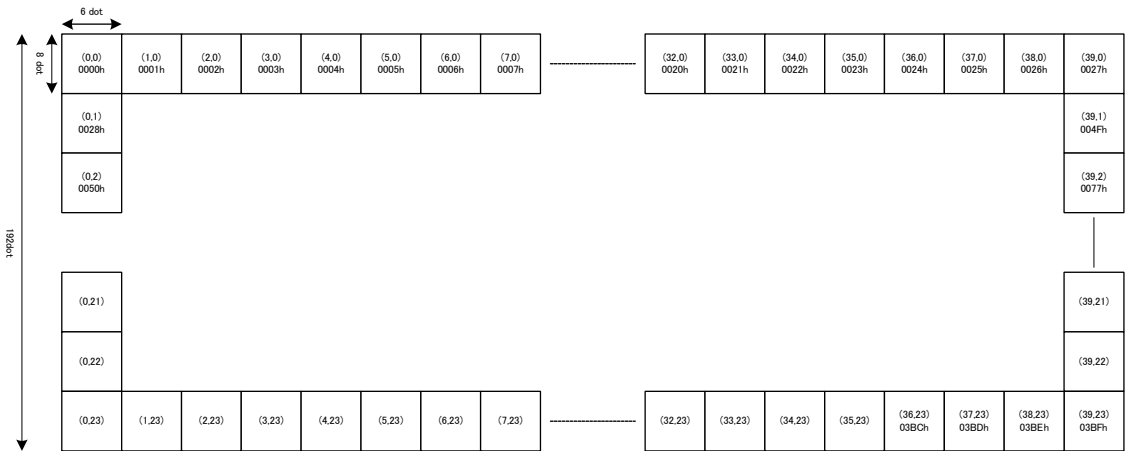


図 2.2.2.1-3. Pattern Name Table

SCREEN1 の Pattern Name Table と似ていますが、水平に並べられる文字数が 40 のときに文字で異なる点です。

MSX Documents VDP

LOCATE で指定する X, Y 座標と、表示したいパーツ番号を P とすれば、MSX-BASIC では下記の記述で Pattern Name Table に書き込めます。

```
VPOKE Y*40+X, P
```

写真 2.2.2.1-4. に上記記述の実行例を示します。

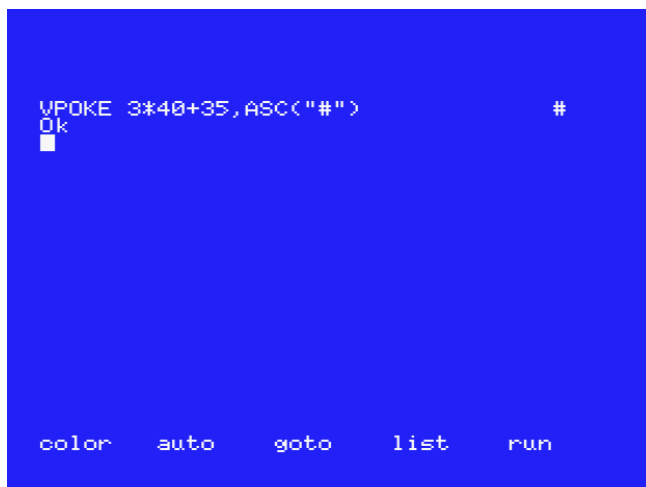


写真 2.2.2.1-4. 実行例

MSX-BASIC の WIDTH 命令は、BASIC の LOCATE 座標位置をずらしたり、PRINT の表示範囲を狭めたりする、いわば MSX-BASIC 用の制御命令です。Pattern Name Table は WIDTH 命令に影響されないのでご注意ください。

SCREEN0(WIDTH40)は非常にシンプルで、VDP が用意する VRAM 上の領域は Pattern Name Table と Pattern Generator Table しか存在しません。

MSX2 以降では Palette Table が存在しますが、これは MSX-BASIC がソフトウェア的に用意している「読み出せないパレットレジスタのバックアップ」であるため、VDP としては関与していません。

2.2.3. SCREEN0:WIDTH 80 (TEXT2)

このモードは、V9938/V9958 のみで利用可能です。

VDP の仕様書では TEXT2 という名前が付けられています。

VRAM を構成する要素を、表 2.2.3.1. にまとめました。

表 2.2.3.1. SCREEN0(WIDTH80)の VRAM 要素

VRAMを構成する要素	概要
Pattern Generator Table	PCGの形状を定義する領域
Pattern Name Table	画面上のPCG配置を定義する領域
Palette Table	カラーパレットの値を保持する領域 (MSX2以降。MSX-BASICのみ)
Blink Table	点滅の有無を指定する領域

パーツが 6dot × 8dot である点、Pattern Generator Table・Pattern Name Table・Palette Table の扱いに関しては、SCREEN0(WIDTH40) と同じなので、詳細な説明はそちらを参照ください。

Pattern Name Table は、1 行 80 桁に増えているので、MSX-BASIC からのアクセス方法は下記のように変更になります。

VPOKE Y*80+X, P

Pattern Name Table のイメージを図 2.2.3.1. に示します。

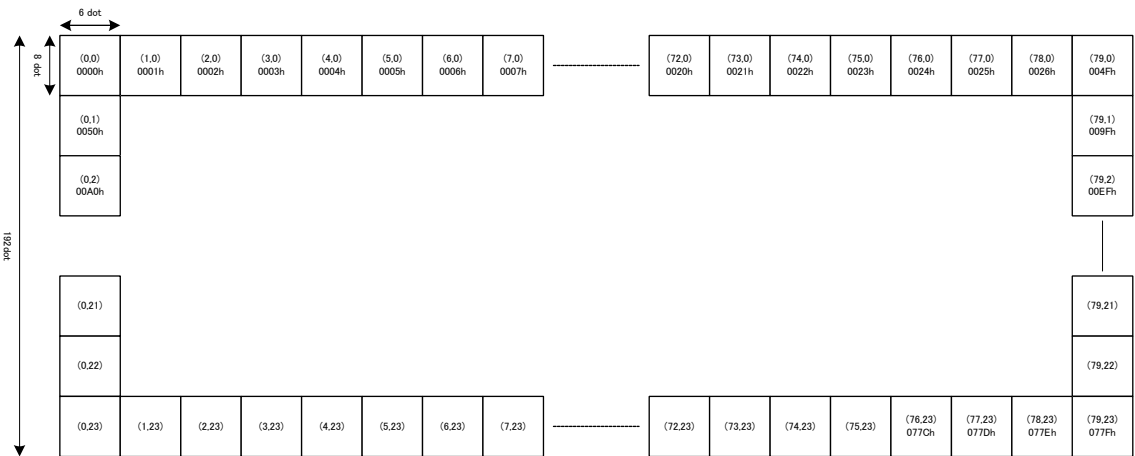


図 2.2.3.1. Pattern Name Table

MSX Documents VDP

Pattern Generator Table は、SCREEN0(WIDTH40)では 0800h からでしたが、SCREEN0(WIDTH80)では 1000h からに変更になります。アドレスが異なるのみで、使い方は SCREEN0(WIDTH40)と同じです。

MSX Documents VDP

ここからは、SCREEN0(WIDTH80)特有の Blink Table について説明します。

Blink Table は、各文字の点滅を指定する機能なのですが、パーツでは無く文字位置に紐付いて点滅指定が可能です。Blink Table を使うサンプルプログラム(SC0BLINK.BAS)を下記に示します。

```
100 SCREEN0:WIDTH80:DEFINT A-Z:COLOR15,4,7
110 PRINT "ABCDE"
120 PRINT "ABCDE"
130 PRINT "ABCDE"
140 FOR I=0 TO 239:VPOKE&H800+I,0:NEXT
150 X=0:Y=0:GOSUB 210
160 X=1:Y=1:GOSUB 210
170 X=3:Y=2:GOSUB 210
180 VDP(13)=&H6
190 VDP(14)=&H11
200 END
210 A=&H800+(X/8)+Y*10
220 B=2^(7-(X AND 7))
230 VPOKE A, VPEEK(A) OR B
240 RETURN
```

この実行結果を写真 2.2.3.1.に示します。

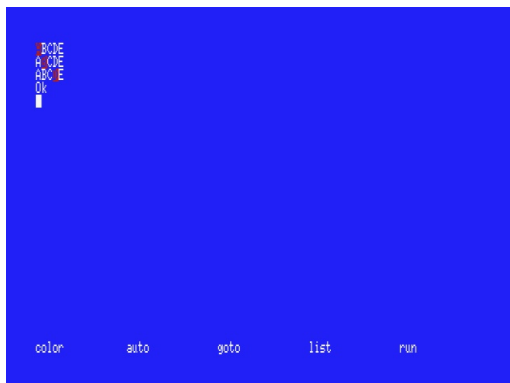


写真 2.2.3.1. Blink Table を使った例

140 行目は、Blink Table を 0 クリアしています。MSX-BASIC では Blink Table を使う命令が存在しないため、Blink Table は VRAM の初期値のままになっています。これをそのまま表示させてしまうと画面全体にランダムに点滅が出てしまうことがあるので、0 で埋めて綺麗にしています。

150 行目～170 行目では、Blink Table に書き込みをしています。210 行目のサブルーチンと呼んでいますが、このサブルーチンは、Blink Table の(X,Y)の位置に 1 を立てる処理となっています。つまり、(X,Y)の位置を点滅させるルーチンです。Blink Table の配置などについては後述します。

180 行目は、VDP(13) (コントロールレジスタ R#12 に対応) に書き込んでますが、このレジスタには点滅時の色を指定します。&H06 となっているので、前景色は 0(透明)、背景色は 6(赤)となっています。ちなみに、0(透明)を指定した場合は、VDP(7) (コントロールレジスタ R#7) の背景色で指定した色になります。VDP(7) でも 0 が指定されている場合は、0 番のパレットの色になります。

190 行目は、VDP(14) (コントロールレジスタ R#13 に対応)に書き込んでいますが、点滅周期を指定しています。上位 4bit が R#12 で指定した色が表示される時間、下位 4bit が R#7 で指定した色が表示される時間です。両方とも単位は 1/6 秒なので、&H66 とすれば、1 秒単位で R#12, R#7 の色が切り替わることになります。作用するのは Blink Table で 1 が指定されている位置のみです。周期に 0 を指定すると、無しになります。例えば、&H10 を指定すると、Blink Table が 1 になっている位置は、R#12 で指定された色になり、点滅はしません。特別な値として &H00 がありますが、&H00 を指定すると、R#7 で指定した色固定になります。

Blink Table の VRAM イメージを図 2.2.3.2. に示します。図 2.2.3.1.の Pattern Name Table と同じ座標系で、Pattern Name Table の 1 パーツに対応する情報は、1bit になっています。1byte で 8dot 分の情報を保持しており、上位ビットが左・下位ビットが右に配置されるようなマッピングになっています。

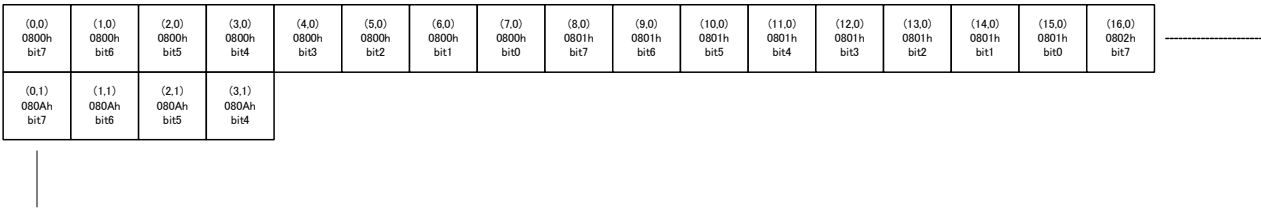


図 2.2.3.1. Blink Table

例えば 0800h に書き込むとき、&B10000000 を書き込むと、(0,0) が点滅します。&B01000000 を書き込むと (1,0) が点滅します。このように、2 進数表記したときの並びと一致している特徴があります。

【note】

MSX の CPU である Z80 はリトルエンディアン(下位が若い番地にマッピングされる)のシステムですが、VDP のドット位置と VRAM のビット並びに関してはビッグエンディアン的な並びになっています。おそらく、表示を司るデバイスなので、VDP 設計者は見た目と整合する方を選択したのだと思います。

2.2.4. SCREEN2 (GRAPHIC2)

このモードは、SCREEN1と似ていますが、SCREEN1とは下記の点で異なります。

- (1) 画面を上・中・下の3つに分割し、それぞれで256パーツの形状定義が可能
- (2) 8ドット×8ドットの各パーツの各ラインごとに自由な2色を設定可能

(1)の特徴ですが、水平32パーツ、垂直8パーツの領域で分割されています。説明の便宜上これらの名前を、(0,0)-(31,7)の上領域、(0,8)-(31,15)の中領域、(0,16)-(31,23)の下領域とします。Pattern Name TableのサイズはSCREEN1と同じですが、これを上領域・中領域・下領域に分割しているため、図2.2.4.1-1のような対応関係になります。

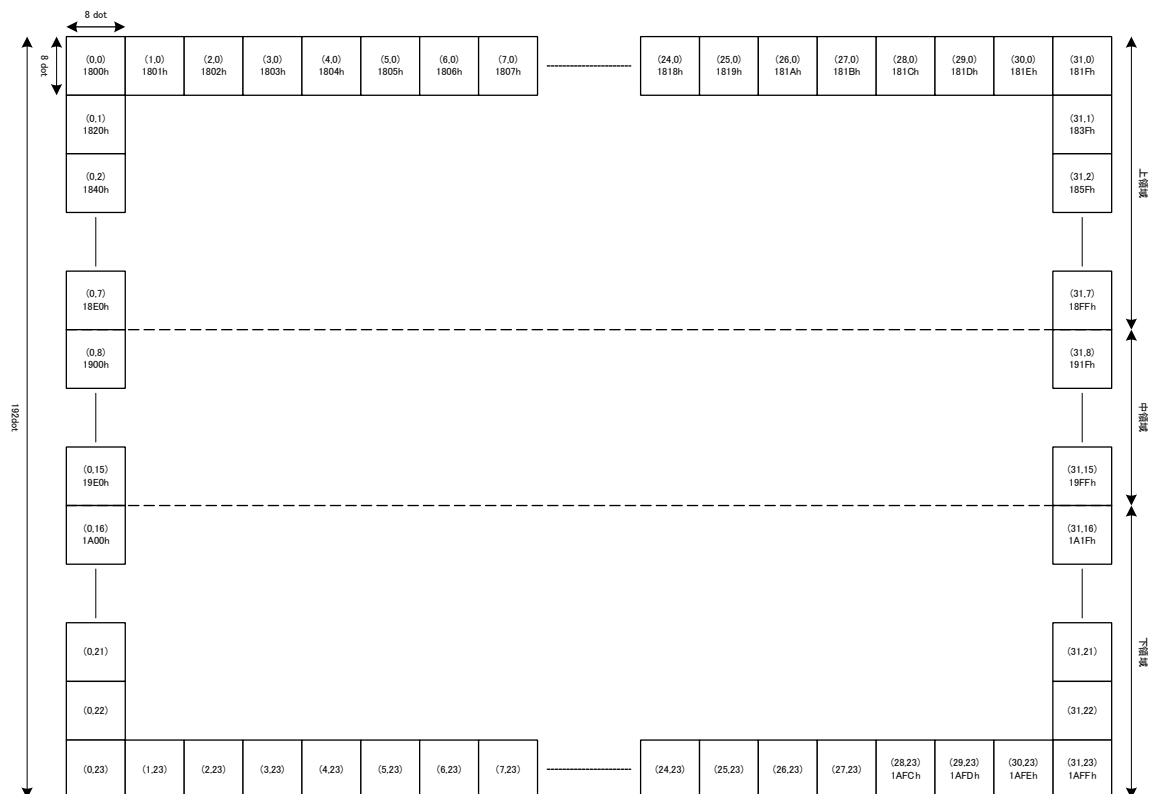


図 2.2.4.1-1. SCREEN2 の上中下分割イメージ

上領域・中領域・下領域のそれぞれの領域は、32×8のサイズですが、これは全部で256パーツとなります。一方で、パーツの形状定義も256パーツ可能なため、領域内をすべて違うパーツで敷き詰めることができます。MSX-BASIC

MSX Documents VDP

では、上領域・中領域・下領域のすべてに 0～255 のパーツを敷き詰めて、パーツの形状定義をいじることでビットマップグラフィックのような表現を実現しています。

上領域・中領域・下領域それぞれで 256 個のパーツ形状を定義できるため、当然ながらそれぞれの領域に対して Pattern Generator Table が存在します。通常は、上領域は 0000h～07FFh、中領域は 0800h～0FFFh、下領域は 1000h～17FFh になっています。

MSX-BASIC で SCREEN2 を実行すると、Pattern Name Table には、0～255 を 3 回繰り返して設定してあります。

従って、グラフィック座標 (X,Y) として、X の取り得る範囲は 0～255、Y の取り得る範囲は 0～191 とした場合、下記の計算により、当該グラフィック座標(X,Y)のドットを含む VRAM アドレスが求められます。

$$A = (Y \text{ and } 7) + (X \text{ and } 248) + (Y \text{ and } 248) * 32$$

X の下位 3bit は、1 バイトの VRAM 値のうち、何ビット目かを示す 0～7 の値で、0 が MSB、7 が LSB に対応するため、対応する桁を 1 にするには、VPEEK(A) or $2^{\wedge}((7-X) \text{ and } 7)$ をすれば良いことがわかります。確認のためのサンプルプログラム (SC2LINE.BAS) を下記に示します。

```
100 SCREEN2:DEFINT A-Z
110 FOR I=0 TO 191: X=I: Y=191-I: GOSUB 130: NEXT I
120 GOTO 120
130 A=(Y AND 7)+(X AND 248)+(Y AND 248)*32
140 M=2^((7-X) AND 7)
150 VPOKEA,VPEEK(A) OR M
160 RETURN
```

MSX Documents VDP

これを動かしてみたときのイメージを写真 2.2.4.1-1. に示します。

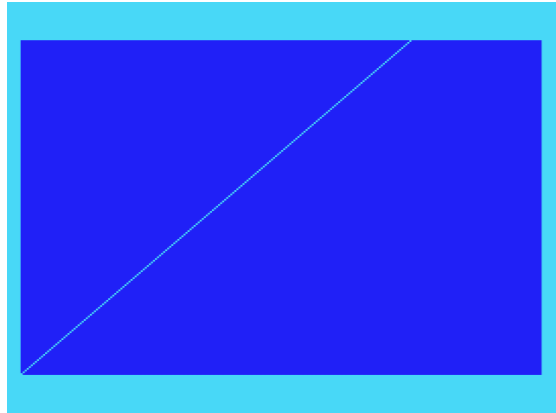


写真 2.2.4.1-1. Pattern Generator Table の書き換えによる線描画

130～160 行目がグラフィック座標 (X,Y) に点を描画するサブルーチンです。VRAM アドレス A を計算して、X 位置に対応する bit を 1 にしたマスク M を計算。最後に VRAM のアドレス A に M を OR して点を描画しています。

簡単に残りを説明すると、100 行目は初期化。110 行目がメインループ。X を 0,1,2, ... , 191、Y を 191, 190, 189, ... , 0 と順に変えながら、上記の点の描画のサブルーチンを呼んで左下から右上へ伸びる直線を描画しています。120 行目はテキスト画面へ戻らないように無限ループしています。MSX-BASIC はプログラム終了時に SCREEN0 か 1 でない場合、0 か 1 に戻す処理を入れてしまうため、せっかく描画した線が消えてしまいます。そのため、無限ループ待ちをして止めています。

Pattern Generator Table1 (上領域用) の画面位置 (左上端) との対応関係のイメージを図 2.2.4.1-2. に示します。

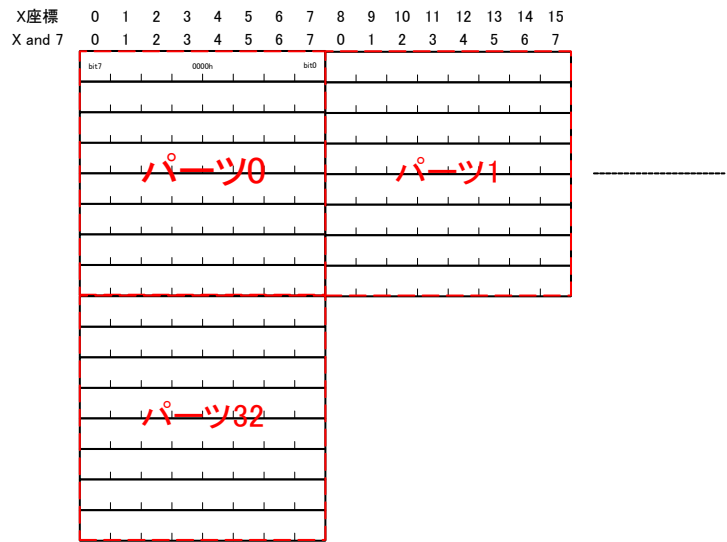


図 2.2.4.1-2. Pattern Generator Table1 と画面位置の対応関係

MSX Documents VDP

Pattern Name Table は、0, 1, 2, ... と連続する番号を順番に設定してあります。これは MSX-BASIC の内部で設定しています。従って左上隅の 8x8ドットはパーツ 0 が配置されています。その右はパーツ 1。パーツ 0 の下にはパーツ 32 が配置されています。

X 座標の下位 3bit は、パーツの中の水平 8ドットのうちどこなのか?を示す値となります。X and 7 が下位 3bit の取り出しとなります。これは VRAM に書き込む値のビット位置を決定するのに使います。

X 座標の上位 5bit は、水平に並ぶ 8x8ドットパーツの中の左から何番目か?を示す値となります。X ¥ 8 が上位 5bit の取り出しとなります。これは Pattern Generator Table のアドレスを決定する要素の一つになります。Pattern Generator Table は、1 パーツ 8byte で構成されているので、パーツ番号を選ぶ値は 8 倍することでアドレスに変換できます。(X ¥ 8) * 8 は、下位 3bit を 0 にするものなので、(X and 248) と同じですね。

Y 座標の下位 3bit(bit2~0)は、パーツの中の垂直 8ドットのうちどこなのか?を示す値となります。Y and 7 が下位 3bit の取り出しとなります。Pattern Generator Table 内で、1 パーツ内の垂直位置は連続アドレスになるため、この下位 3bit はそのままアドレスに使います。

Y 座標の中位 3bit(bit5~3)は、上領域・中領域・下領域それぞれ垂直に 8 パーツ並んでいるかのパーツ位置になります。Y 座標上位 2bit(bit7~6)は、上領域・中領域・下領域のどれかを示す値になります。3 つの領域の Pattern Generator Table は必ず連続アドレスにマッピングされるため、bit7~bit3 の 5bit をまとめて扱っても問題ありません。水平に並ぶパーツは 32 パーツなので、1 つ下のパーツ番号は +32 であることから、bit7~bit3 を切り出した 5bit に対して 32 倍を掛ければパーツ番号に、さらに Pattern Generator Table は 1 パーツ 8byte で構成されているので 8 倍でアドレスに変換されることが分かります。(Y ¥ 8) * 32 * 8 となりますね。(Y ¥ 8) * 8 * 32 と並べ替え、さらに (Y and 248) * 32 となります。

これらを踏まえて、再度アドレス計算の式を確認してみましょう。

$$A = (Y \text{ and } 7) + (X \text{ and } 248) + (Y \text{ and } 248) * 32$$

さて、点は打てましたが、色の方は Color Table に格納されている値で決定されます。Color Table も上領域・中領域・下領域のそれぞれに存在していて連続アドレスにマッピングされています。通常、2000h にマッピングされています。上記サンプル(SC2LINE.BAS)では、Color Table に何も書き込んでいませんでしたが、この初期値がどうなっているのか確認するために、VRAM の内容を表示するように修正したものを下記(SC2LINE2.BAS)に示します。

MSX Documents VDP

```
100 SCREEN2:DEFINT A-Z
110 FOR I=0 TO 191: X=I: Y=191-I: GOSUB 140: NEXT I
120 OPEN "GRP:" AS #1: PSET (0,8),15: PRINT #1, RIGHT$("0"+HEX$(VPEEK(&H2000)),2)
130 GOTO 130
140 A=(YAND7)+(XAND248)+(YAND248)*32
150 M=2^((7-X)AND7)
160 VPOKEA,VPEEK(A) OR M
170 RETURN
```

&H2000 は、左上隅の水平 8 ドットの色になりますが、初期状態は Color Table 全域が同じ値で埋められているのでこの 1 カ所のみ表示します。

追加したのは 120 行目。(0,8)を左上とする位置に VPEEK(&H2000)の値を 2 桁 16 進数で描画するコードです。

MSX-BASIC は、内部でグラフィックカーソルというグラフィック画面用の着目座標値を持っていて、座標指定を省略するとこの位置から描画する動作になっています。PSET でグラフィックカーソルを移動して、PRINT #1 でその位置から文字描画しているわけです。このとき PSET(0,0),15 とすると左上隅に 15 番の色で点を打ってしまっていて、VRAM & H2000 を書き換えてしまうので、少し下にずらして PSET (0,8),15 としています。

これを実行したイメージを写真 2.2.4.1-2.に示します。

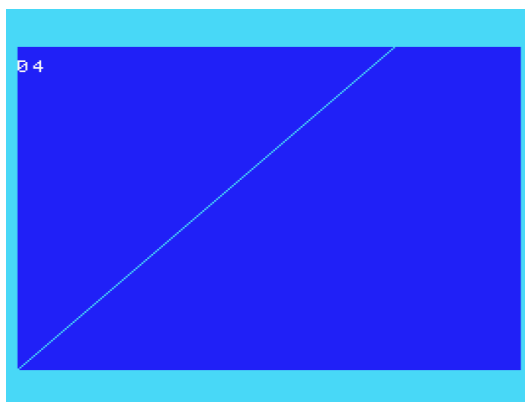


写真 2.2.4.1-2. &H2000 の内容を描画

04h となっています。Color Table は、上位 4bit が「Pattern Generator Table に対応 bit が 1 の画素の色」、下位 4 bit が「対応 bit が 0 の画素の色」に対応しています。この場合、1 は 0 番の色、0 は 4 番の色になっているわけですね。0 は透明になるので、周辺色（写真では水色の領域）が透けて見えますので、線が水色になった挙動と一致します。

では次に、この線を描いたところの Color Table をいじってみましょう。

MSX Documents VDP

前景 (Pattern Generator Table で、対応 bit が 1 になっている画素)は 12(濃い緑), 背景(同 0 になっている画素)は 1(黒)にしてみます。Color Table に書き込む値としては、&HC1 ですね。修正したプログラムを下記(SC2LINE3.B AS)に示します。

```
100 SCREEN2:DEFINT A-Z
110 FOR I=0 TO 191: X=I: Y=191-I: GOSUB 140: NEXT I
120 FOR I=0 TO 191: X=I: Y=191-I: GOSUB 180: NEXT I
130 GOTO 130
140 A=(YAND7)+(XAND248)+(YAND248)*32
150 M=2^(7-X)AND7
160 VPOKEA,VPEEK(A)OR M
170 RETURN
180 A=(YAND7)+(XAND248)+(YAND248)*32+&H2000
190 VPOKEA,&HC1
200 RETURN
```

これを実行したイメージを写真 2.2.4.1-3.に示します。

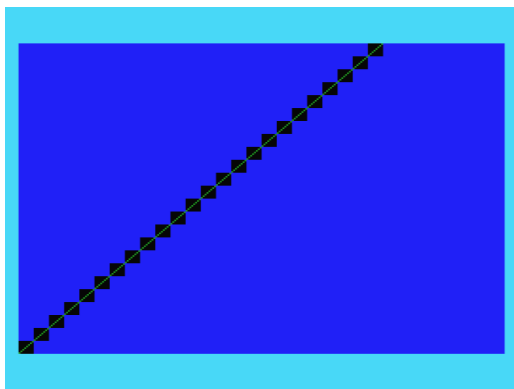


写真 2.2.4.1-3. Color Table に &HC1 を書き込む

水平連続 8 ドットにつき 2 色しか配色できないのがよく分かる動作だと思います。

Palette Table に関しては、SCREEN1 と同様ですので説明は省略します。

Sprite 関連は、2.5. スプライトを参照ください。

MSX Documents VDP

MSX マガジン誌などで SCREEN1.5 という名前を見かけたことがある人も居ると思いますが、これは VDP としては SCREEN2 相当だけど、MSX-BASIC は SCREEN1 と誤解している状態のことを示しています。

MSX-BASIC は、SCREEN2 と認識している状態で MSX-BASIC の実行を終えると、実行前の画面モードへ戻す仕様になっています。しかし、VDP を直接制御して画面モードを変更したり、あるいは MSX-BASIC のワークエリアに存在する情報を書き換えることで、上記誤解の動作を実現しています。具体的には下記のようにします。フォントの形状定義などもあり、MSX-BASIC では遅すぎるので、今回はアセンブラを利用します。プログラムは下記(SC1P5.ASM)のとおりです。

```
; -----  
;   SCREEN1.5 に切り替える  
; =====  
;   2020/03/21   t.hara  
; -----
```

```
ROM_FONT      := 0x1BBF
```

```
FILVRM        := 0x0056
```

```
LDIRVM        := 0x005C
```

```
CHGMOD        := 0x005F
```

```
SCRMOD        := 0xFCAF
```

```
PGT1          := 0x0000
```

```
PGT2          := 0x0800
```

```
PGT3          := 0x1000
```

```
COLT          := 0x2000
```

```
PNT           := 0x1800
```

```
set_screen_1p5::
```

```
    ; SCREEN2 へ切り替える
```

```
    ld      a, 2
```

```
    call    CHGMOD
```

```
    ; MSX-BASIC を「SCREEN1」だと誤認させる
```

```
    ld      a, 1
```

```
    ld      [SCRMOD], a
```

```
    ; Pattern Generator Table にフォントを転送する
```

```
    ld      hl, ROM_FONT
```

```
    ld      de, PGT1
```

```
    ld      bc, 8*256
```

```
    call    LDIRVM
```

```
    ld      hl, ROM_FONT
```

MSX Documents VDP

```
ld      de, PGT2
ld      bc, 8*256
call    LDIRVM

ld      hl, ROM_FONT
ld      de, PGT3
ld      bc, 8*256
call    LDIRVM

; Color Table に色を書き込む
ld      hl, COLT
ld      a, 0xF4          ; 前景: 15, 背景: 4
ld      bc, 0x800*3
call    FILVRM

; Pattern Name Table に ' ' を敷き詰める
ld      hl, PNT
ld      a, ' '
ld      bc, 32*24
call    FILVRM
ret
```

これを実行する BASIC プログラムを下記(SC1P5.BAS)に示します。

```
100 CLEAR100,&HD7FF:SCREEN1:DEFINT A-Z
110 FOR I=0 TO &H44:READ D:POKE &HD800+I,D:NEXT
120 DEFUSR0=&HD800:A=USR(0)
130 END
140 DATA &H3E,&H02,&HCD,&H5F,&H00,&H3E,&H01,&H32,&HAF,&HFC,&H21,&HBF,&H1B,&H11,&H00,&H00
150 DATA &H01,&H00,&H08,&HCD,&H5C,&H00,&H21,&HBF,&H1B,&H11,&H00,&H08,&H01,&H00,&H08,&HCD
160 DATA &H5C,&H00,&H21,&HBF,&H1B,&H11,&H00,&H10,&H01,&H00,&H08,&HCD,&H5C,&H00,&H21,&H00
170 DATA &H20,&H3E,&HF4,&H01,&H00,&H18,&HCD,&H56,&H00,&H21,&H00,&H18,&H3E,&H20,&H01,&H00
180 DATA &H03,&HCD,&H56,&H00,&HC9
```

140～180 行目にある数字の羅列は、上記 SC1P5.ASM のアセンブル結果なので、SC1P5.ASM が無くても動作します。とりあえず &HD800～ に配置したので、RAM16KB の機種でも動作すると思います。別の番地に配置しても動作しますので、100 行目の &HD7FF、110 行目の &HD800 を必要に応じて修正してください。RAM8KB の機種の場合、プログラムをシェイプアップする必要があると思います。わかりやすさ重視で書いているので、やっている内容にしては長いプログラムになっています。

実行すると SCREEN1.5 になります。

MSX Documents VDP

注意する点は、MSX-BASIC は BASIC のプログラム入力時に、カーソルのパーツ(パーツ番号 255)の形状を頻繁に更新しています。カーソルが重なった位置は、文字が反転したように見えますが、これは MSX-BASIC が反転した形状をパーツ 255 にセットして、Pattern Name Table は 255 を設定することで反転したような表示にしています。上記プログラムの実行後は、実際は SCREEN2 の VDP 設定なのだけでも、MSX-BASIC は SCREEN1 だと認識している状態のため、カーソル形状も SCREEN1 の Pattern Generator Table に一致する Pattern Generator Table 1 (上領域)しか設定されません。中領域・下領域のパーツ 255 はブランクのままなので、カーソルが見えなくなってしまうます。

同様の理由で、INS キーを押して挿入モードにした時のカーソル形状なども上領域しか追従しません。

中領域と下領域のパーツ 255 を白い■にしておけば、反転表示や挿入モードカーソルはないけどもカーソル位置を見失うことはないですね。必要であれば、各自やってみてください。

2.2.5. SCREEN3 (MULTI COLOR)

1画面を水平64ドット、垂直48ドットで表現するモードです。このモードの1ドットはSCREEN0,1,2の4ドット×4ドットのサイズに相当します。かなりドットが粗いですが、任意の位置に好きな色を配置できます。この大きなドットを水平2ドット、垂直8ドットの縦長のブロックを1パーツとして扱っています。さらに1パーツは水平2ドット、垂直2ドット単位で4つに分解されていて、上から順にA,B,C,Dとすると、Y座標の下位3bitが0か1の場合はA, 下位3bitが2か3の場合はB, 4か5の場合はC, 6か7の場合はDが表示される特殊な構造になっています。Pattern Name Tableは、この4つそれぞれ独立して指定できるので、水平32パーツ、垂直24パーツの768パーツで画面全体を覆い、グラフィック画面の様相を実現しています。おそらく、SCREEN2の演算回路とある程度回路を共通化するためにこのような構造になっているのだと思います。

Pattern Generator Tableの1byteが、2画素の情報を保持しており、1パーツは連続する8byteで構成されています。1byteの内容は、上位4bitに左の画素の色、下位4bitに右の画素の色を指定します。

MSX-BASICでは、Pattern Name Tableは、0, 1, 2, ..., 31, 0, 1, 2, ..., 31, 0, 1, ..., 191と、0～31の連番を4回繰り返して格納、次に32～63の連番を4回繰り返して格納、としているため、ドットの位置を示す(X,Y)から、VRAMアドレス(Pattern Generator Table)を求める式は下記ようになります。

$$(X \div 2) * 8 + (Y \text{ and } 7) + (Y \text{ and } 248) * 32$$

このアドレスに対して、X and 1が0なら上位4bit、X and 1が1なら下位4bitに所望の値を書き込むことで、点を打つことができます。画面イメージを図2.2.5.1.に示します。

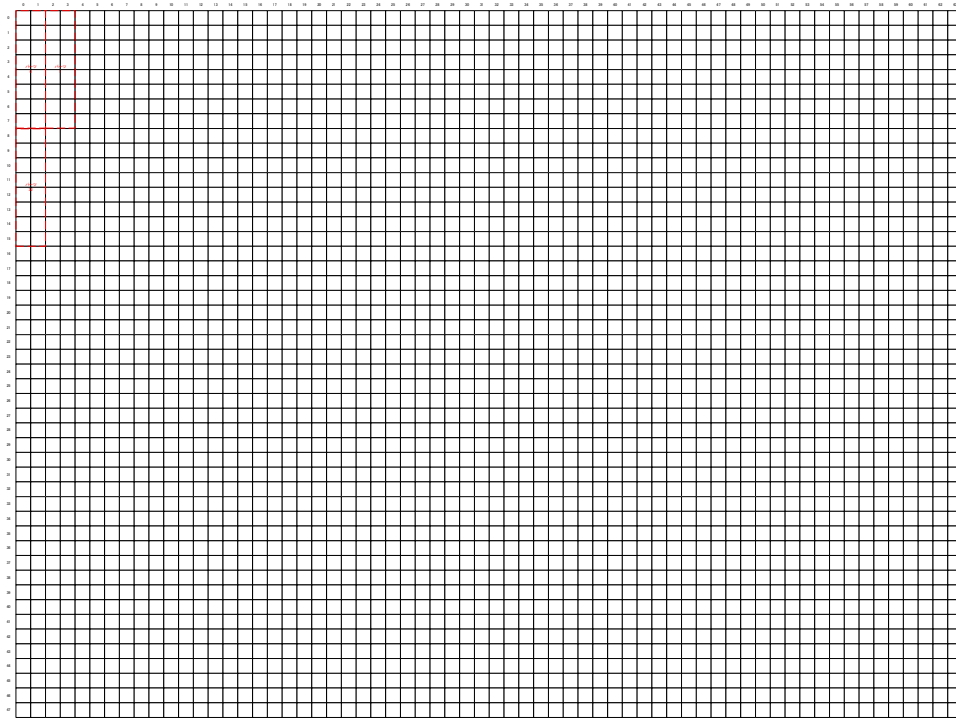


図 2.2.5.1. SCREEN3 の画面イメージ

赤枠で囲われている水平 2 画素、垂直 8 画素が 1 パーツに相当します。このパーツが水平 32 個、垂直 6 個並んでいます。パーツは 256 種類ありますが、このパーツを画面上にどのように配置するかを決めるのが Pattern Name Table です。Pattern Name Table は、0800h への 192byte になります。MSX-BASIC では、0800h には 0, 0801h には 1, ..., 08BFh には 191 と、連番が格納されます。

この各パーツの形状を決めるのが Pattern Generator Table になります。Pattern Generator Table は、0000h への 2048byte になります。1 パーツ 8byte で、256 パーツ定義できます。0~191 のパーツしか使われていないため、64 個のパーツが未使用の状態です。Pattern Generator Table に書き込む 1byte の値の中身のイメージを図 2.2.5.2. に示します。

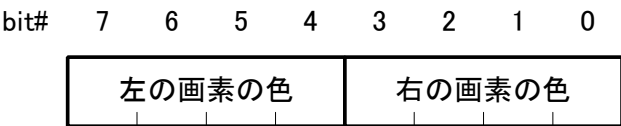


図 2.2.5.2. Pattern Generator Table に設定する値

MSX-BASIC と同じパーツの配置であれば、先ほどの式で、画面上のグラフィック座標 (X,Y) に対応する Pattern Generator Table のアドレスを求めることができます。

MSX Documents VDP

では、実際に左上から右下へ直線を描画してみましょう。サンプルプログラム(SC3LINE.BAS)を下記に示します。

```
100 COLOR15,4,7:SCREEN3:DEFINT A-Z:C(0)=&HC4:C(1)=&H4C
110 FOR I=0 TO 47:X=I:Y=I:GOSUB 130:NEXT
120 GOTO 120
130 A=(X\2)*8+(YAND7)+(YAND248)*32
140 M=2^((7-X)AND7)
150 VPOKEA,C(XAND1)
160 RETURN
```

この実行結果を、写真 2.2.5.1. に示します。

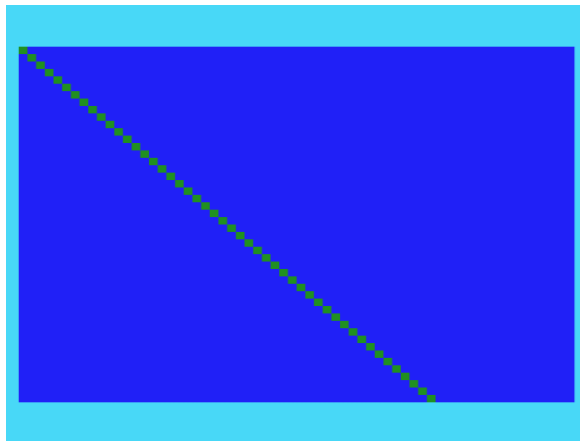


写真 2.2.5.1. (0,0)-(47,47) に &HC の直線を描画

100 行目の C(0) は X が偶数だった場合、つまり Pattern Generator Table の上位 4bit にターゲットの座標が存在する場合に VRAM に書き込む値です。今回、&HC の線を描きたいので、&HC? となるわけですが、COLOR15,4,7 と初期化しているため、描画領域は初期状態で &H4 で塗りつぶされています。これを壊さないために ? は 4 を入れておきます。従って C(0)=&HC4 です。C(1) は X が奇数だった場合、つまり Pattern Generator Table の下位 4bit にターゲットの座標が存在する場合に VRAM に書き込む値になるため、C(0) に対して上位と下位が入れ替わった値にしておきます。従って C(1)=&H4C となります。この数値の意味が分からなかった場合は、実際に値を変えたらどのように挙動が変化するか観察してみると良いと思います。

MSX-BASIC の SCREEN3 でのグラフィック表示は、基本的に Pattern Generator Table の値を書き換えるだけで実現するため、VPOKE 以外に Pattern Name Table を変更するような命令は存在しません。VPOKE 命令で VRAM に直接書き込めば Pattern Name Table も変わるため、ちょっとした変わったこともできます。

先ほどのサンプルプログラム(SC3LINE.BAS)の実行後の画面左上を拡大したイメージを図 2.2.5.3. に示します。

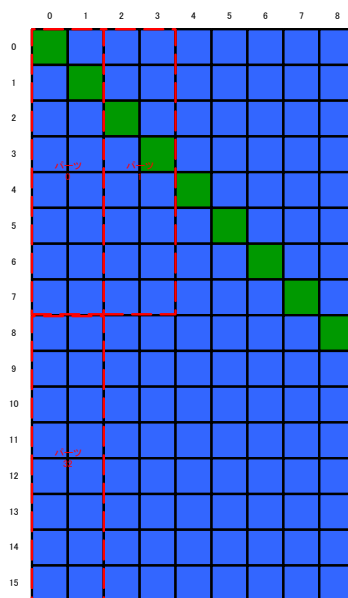


図 2.2.5.3. SC3LINE.BAS 実行後左上拡大イメージ

左上隅の水平2画素・垂直8画素の縦長の領域がパーツ0に対応しています。これは、Pattern Generator Table のパーツ0に記録されているビットマップを表示していて、かつ Pattern Name Table の先頭番地には0と書かれていることになります。

次にすぐ右隣の2×8の縦長の領域にはパーツ1が表示されています。これも Pattern Generator Table のパーツ1に記録されているビットマップを表示していて、かつ Pattern Name Table の2バイト目が1と書かれていることによります。

次にパーツ0のすぐ下は、同様にパーツ32に対応しています。

Pattern Name Table を書き換えて、パーツ1とパーツ32の位置にもパーツ0を配置してみましょう。そのサンプルプログラム(SC3LINE2.BAS)を下記に示します。

```

100 COLOR15,4,7:SCREEN3:DEFINT A-Z:C(0)=&HC4:C(1)=&H4C
110 FOR I=0 TO 47:X=I:Y=I:GOSUB 140:NEXT
120 VPOKE&H801,0:VPOKE&H820,0
130 GOTO 130
140 A=(X\2)*8+(YAND7)+(YAND248)*32
150 M=2^((7-X)AND7)
160 VPOKEA,C(XAND1)
170 RETURN

```

MSX Documents VDP

120 行目で Pattern Name Table を書き換えているわけですが、VPOKE&H801,0 は、もともとパーツ1の上端2x2が配置されている位置にパーツ0の上端2x2を表示せよという意味。VPOKE&H820,0 は、もともとパーツ0だった場所なので、この書き込みで表示は変化しません。

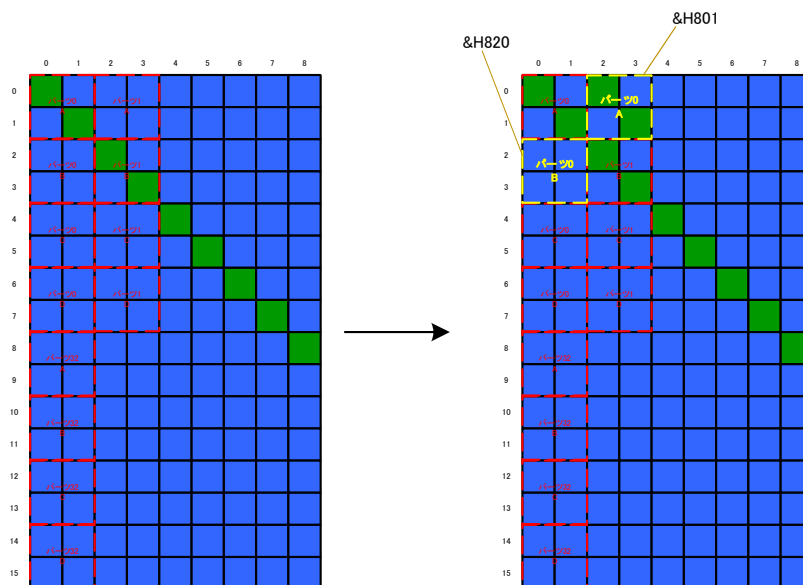


図 2.2.5.4. SC3LINE2.BAS 実行イメージ

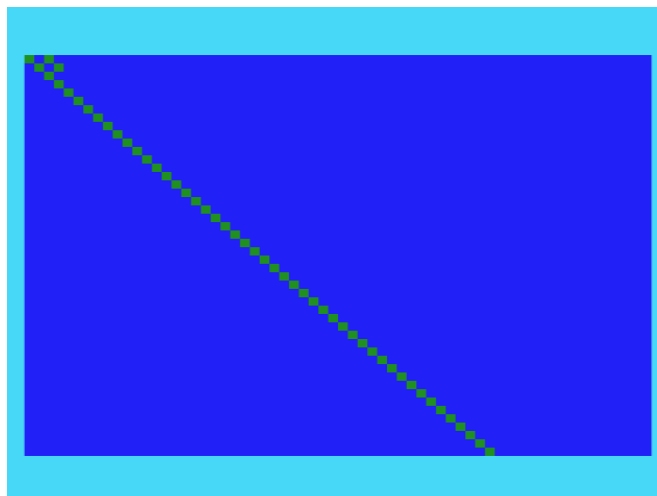


写真 2.2.5.2. VRAM の&H801, &H820 の2カ所に 0 を書き込んだ後

同じパーツ番号であっても、Y 座標によってパーツのどの部分が表出するかが変わるため、&H800, &H801 に対応する左右に並ぶ位置の (0,0)-(1,1) のブロックと、(2,0)-(3,1) のブロックとは同じ表示になりますが、Y 座標が異な

MSX Documents VDP

る (0,0)-(1,1) と (0,2)-(1,3) では、同じパーツ 0 でも、パーツ 0 の A,B,C,D のどこが表示するかが異なるため、同じ繰り返し表示にはなりません。

このように、ややこしい配置の画面モードのため、あまり活用されているケースを見かけませんが、MSX1(TMS9918) で周囲のドットの状況に影響されずにドットごとに好きな色を配置できるのはこのモードだけです。使い方次第では、面白いコンテンツを実現できるかもしれませんね。

次に MSX-BASIC での座標の取り扱いに関する注意点です。MSX-BASIC の SCREEN3 で PSET や LINE のようなグラフィック描画命令を使うときに使用する座標系は、画面全体を水平 256ドット・垂直 192ドットと見なした SCREEN 2 と同じ座標系になっていることに注意しなければなりません。PSET(0,0),15:PSET(1,0),2 とやると、画面左上の隅の同じドットに色 15 を描画した後に色 2 を描画してしまいます。画面全体を 水平 64ドット・垂直 48ドットの座標系の値は X と Y それぞれ 4 倍した値を指定しなければなりません。そのイメージが分かるサンプルプログラム(SC3PTN.BAS)を下記に示します。

```
100 COLOR15,4,7:SCREEN3:DEFINT A-Z
110 FOR I=0 TO 15:PSET((I AND 1)*4,(I\2)*4),I:PSET(((I AND 1)+2)*4,(7-(I\2))*4),I:NEXT I
120 GOTO 120
```

MSX Documents VDP

このサンプルの実行結果を写真 2.2.5.3.に示します。



写真 2.2.5.3. SC3PTN.BAS の実行結果

次に Pattern Name Table の理解を深めるために、このサンプル(SC3PTN.BAS)をいじって動作を見てみましょう。

Pattern Name Table の先頭の 32byte をすべて 0 にするプログラム(SC3PTN2.BAS)を下記に示します。

```
100 COLOR15,4,7:SCREEN3:DEFINT A-Z
110 FOR I=0 TO 15:PSET((I AND 1)*4,(I\2)*4),I:PSET(((I AND 1)+2)*4,(7-(I\2))*4),I:NEXT
120 FOR I=&H800 TO &H81F:VPOKE I,0:NEXT
130 GOTO 130
```

この実行結果を写真 2.2.5.4.に示します。



写真 2.2.5.4. SC3PTN2.BAS の実行結果

左上隅の2ドット×2ドットのブロックが、その右へ31個複製されたことが分かります。

次に Pattern Name Table の先頭 64byte を 0 フィルするように修正したプログラム(SC3PTN3.BAS)を下記に示します。

MSX Documents VDP

```
100 COLOR15,4,7:SCREEN3:DEFINT A-Z
110 FOR I=0 TO 15:PSET((I AND 1)*4,(I\2)*4),I:PSET(((I AND 1)+2)*4,(7-(I\2))*4),I:NEXT
120 FOR I=&H800 TO &H83F:VPOKE I,0:NEXT
130 GOTO 130
```

この実行結果を写真 2.2.5.5.に示します。

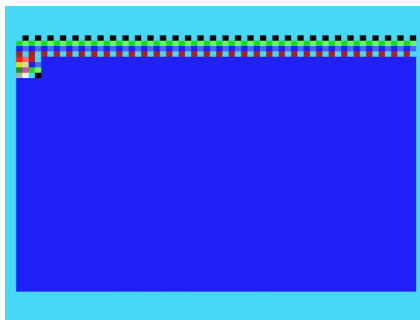


写真 2.2.5.5. SC3PTN3.BAS の実行結果

Pattern Name Table の先頭 32byte・その次の 32byte は、ともに 0 で埋められているのに、先頭 32byte に対応する領域は、左上隅の 2ドット×2ドット、その次の 32byte に対応する領域は、左上隅の 2ドット×2ドットのすぐ下の 2ドット×2ドットが複製された状態になっています。このように、同じパーツ番号であっても、描画対象となる Pattern Generator Table の対応位置が異なるのです。Pattern Generator Table の 1パーツ分の領域を切り出したイメージを図 2.2.5.5.に示します。

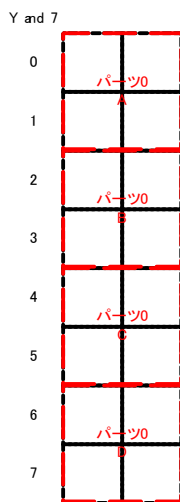


図 2.2.5.5. Pattern Generator Table の1パーツ分

MSX Documents VDP

Y and 7 で示される 0, 1, 2, ..., 7 は VRAM アドレスに相当します。これを連続する 2 個ペアにした 0,1 を A、2, 3 を B、4, 5 を C、6, 7 を D と名付けます。画面上の座標 Y (※画面全体を 64×48 と見なした座標系です) の下位 3bit である Y and 7 から A, B, C, D のどれに該当するか選択され、Pattern Name Table で示されるパーツ番号の A, B, C, D のいずれかが画面に表示されることになります。

従って、Pattern Name Table をすべて 0 フィルすると、垂直 8 画素を繰り返したイメージになります。その確認プログラム(SC3PTN4.BAS)を下記に示します。

```
100 COLOR15,4,7:SCREEN3:DEFINT A-Z
110 FOR I=0 TO 15:PSET((I AND 1)*4,(I\2)*4),I:PSET(((I AND 1)+2)*4,(7-(I\2))*4),I:NEXT I
120 FOR I=&H8000 TO &HFFFF:VPOKE I,0:NEXT I
130 GOTO 130
```

この実行イメージを写真 2.2.5.6. に示します。

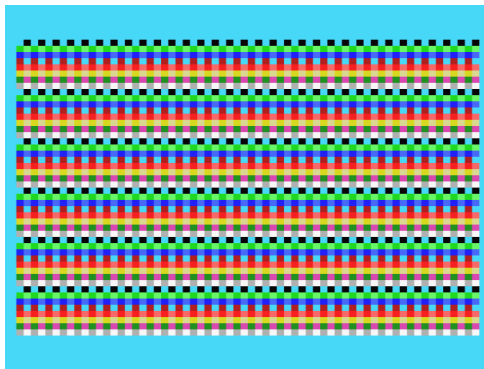


写真 2.2.5.6. SC3PTN4.BAS の実行結果

2.2.6. SCREEN4 (GRAPHIC3)

このモードは、TMS9918(MSX1)では利用できません。V9938/V9958 で利用可能です。ほぼ SCREEN2(GRAPHIC2)と同じですが、スプライトがスプライトモード 2 になっている点のみが異なります。そのため、Pattern Name Table, Pattern Generator Table, Color Table については 2.2.4. SCREEN2 (GRAPHIC2)を、スプライトについては 2.5.2. スプライトモード 2 の説明をご覧ください。

また、CPU からの VRAM アクセスにおけるアドレスオートインクリメントは、SCREEN0～3 では TMS9918 との互換性のため、03FFFH の次は 00000H に戻りますが、SCREEN4 では 03FFFH の次は 04000H に進む点が異なります。つまり、オートインクリメントによって R#14 で管理されている上位アドレスビットが影響を受けるかどうか異なります。

SCREEN2 のところで、SCREEN1.5 について説明しましたが、同様のことを SCREEN4 を使っても実現できます。SCREEN1.5 のような俗称を知らないなので、ここでは仮に SCREEN1.75 と呼ぶことにします。

Sprite Attribute Table のアドレスは、SCREEN1 と異なるのですが、BASE()システム変数により管理されているので、PUT SPRITE 命令、SPRITE\$()システム変数は問題なく使えます。しかしながら、色を指定する引数は無効になります。色は Sprite Color Table に書き込まねばなりませんが、MSX-BASIC は SCREEN1 (スプライトモード 1)だと思い込んでいるので、Sprite Color Table も存在していないものとして扱っています。そのため、Sprite Color Table を書き換える命令 (COLOR SPRITE() 等) は使えません。

```
; -----
;   SCREEN1.75 に切り替える
;   =====
;   2020/03/25   t.hara
;   -----
```

```
ROM_FONT      := 0x1BBF
```

```
FILVRM        := 0x0056
```

```
LDIRVM        := 0x005C
```

```
CHGMOD        := 0x005F
```

```
SCRMOD        := 0xFCAF
```

```
PGT1          := 0x0000
```

```
PGT2          := 0x0800
```

```
PGT3          := 0x1000
```

```
COLT          := 0x2000
```

```
PNT           := 0x1800
```

```
set_screen_1p5::
```


MSX Documents VDP

```
; SCREEN4 へ切り替える
ld      a, 4
call    CHGMOD

; MSX-BASIC を「SCREEN1」だと誤認させる
ld      a, 1
ld      [SCRMOD], a

; Pattern Generator Table にフォントを転送する
ld      hl, ROM_FONT
ld      de, PGT1
ld      bc, 8*256
call    LDIRVM

ld      hl, ROM_FONT
ld      de, PGT2
ld      bc, 8*256
call    LDIRVM

ld      hl, ROM_FONT
ld      de, PGT3
ld      bc, 8*256
call    LDIRVM

; Color Table に色を書き込む
ld      hl, COLT
ld      a, 0xF4          ; 前景： 15, 背景： 4
ld      bc, 0x800*3
call    FILVRM

; Pattern Name Table に ' ' を敷き詰める
ld      hl, PNT
ld      a, ' '
ld      bc, 32*24
call    FILVRM
ret
```

これを実行する BASIC プログラムを下記(SC1P75.BAS)に示します。

```
100 CLEAR100,&HD7FF:SCREEN1:DEFINT A-Z
110 FOR I=0 TO &H44:READ D:POKE &HD800+I,D:NEXT
120 DEFUSR0=&HD800:A=USR(0)
130 END
140 DATA &H3E,&H04,&HCD,&H5F,&H00,&H3E,&H01,&H32,&HAF,&HFC,&H21,&HBF,&H1B,&H11,&H00,&H00
150 DATA &H01,&H00,&H08,&HCD,&H5C,&H00,&H21,&HBF,&H1B,&H11,&H00,&H08,&H01,&H00,&H08,&HCD
160 DATA &H5C,&H00,&H21,&HBF,&H1B,&H11,&H00,&H10,&H01,&H00,&H08,&HCD,&H5C,&H00,&H21,&H00
```

MSX Documents VDP

```
170 DATA &H20,&H3E,&HF4,&H01,&H00,&H18,&HCD,&H56,&H00,&H21,&H00,&H18,&H3E,&H20,&H01,&H00
180 DATA &H03,&HCD,&H56,&H00,&HC9
```

この実行後に BASIC 命令のインライン実行で左上にスプライトを表示してみた画面を写真 2.2.6.1.に示します。

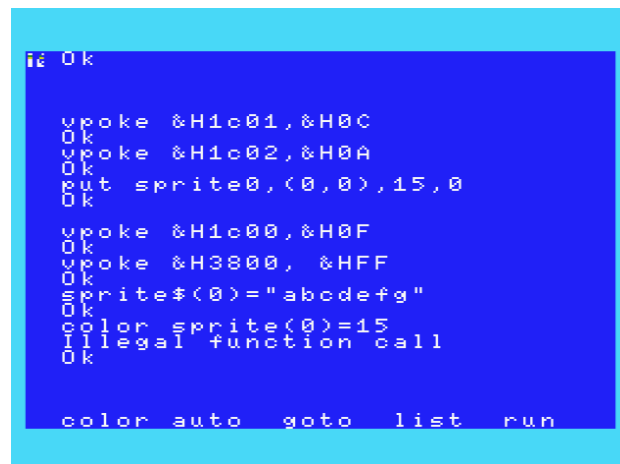


写真 2.2.6.1. SCREEN1.75 の動作イメージ

2.2.7. SCREEN5 (GRAPHIC4)

このモードは、TMS9918 では利用できません。V9938/V9958 で利用可能です。

V9938 から搭載されたビットマップグラフィックを扱うモードで、MSX2 以降のゲームで最も多く使われているモードだと思います。水平 256ドット、垂直 212ドットになります。

巨大な Pattern Name Table (0000h～)にビットマップ情報が格納されていて、水平2ドットペアが 1byte に収まっています。上位 4bit が左ドット、下位 4bit が右ドットに対応しています。左上から右に向かって、右端に到達すると、すぐ下のラインの左端から右へ・・・と素直な並びになっています。Pattern Name Table だけでグラフィック表示を構成しています。

VRAM を構成する他の要素は、Sprite Color Table (7400h～), Sprite Attribute Table (7600h～), Sprite Generator Table (7800h～) のスプライト系と、MSX-BASIC が用意する Palette Table (7680h～) のみです。スプライトは、スプライトモード2になりますので、2.5.2. スプライトモード 2 を参照ください。Palette Table に関しては、MSX-BASIC が COLOR=(n,r,g,b)の結果を書き出しているだけのデータ置き場なので、ここに書き込んだところでパレットが変化するわけではないのは他のモードと同じです。

ビットマップ表示によってリッチな表示ができるようになった反面、I/O アクセス経由の VRAM 書き込みでは VDP の最高スループットで書き込んだとしても限界があります。扱う情報量に対して I/O ポートのアクセス帯域が細すぎて、ゲームなどに十分なデータ転送を賄いきれません。これを補うために、VDP には VDP コマンドという機能が追加されています。VDP コマンドについては別途 2.6. VDP コマンドにて説明したいと思います。

ここでは、Pattern Name Table について説明します。Pattern Name Table と画面のドットの対応関係のイメージを図 2.2.7.1.に示します。

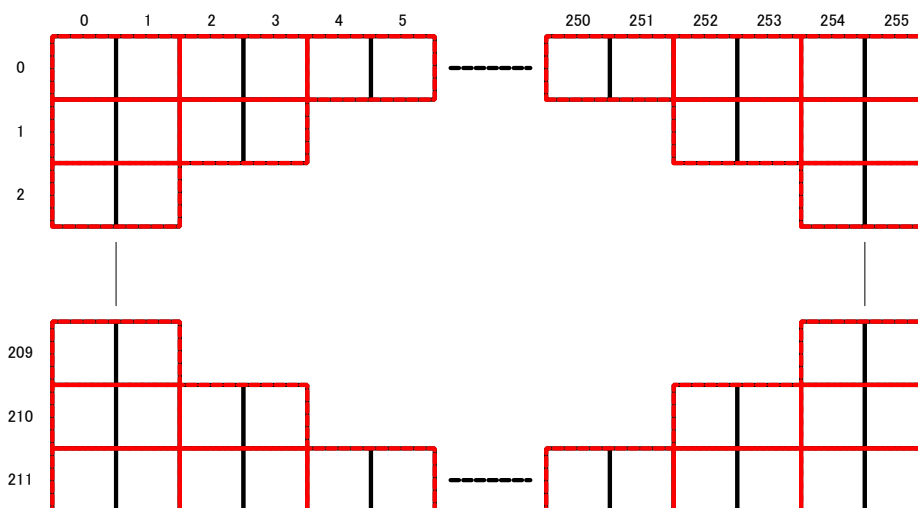


図 2.2.7.1. 画面上のドットと Pattern Name Table の対応関係

図 2.2.7.1.の赤い点線が Pattern Name Table の 1byte に相当します。黒い四角が1ドットに相当します。このように隣り合う2ドットが 1byte に対応しています。Pattern Name Table 上の 1byte の上位 4bit が左ドットの色を示すパレット番号、下位 4bit が右ドットの色を示すパレット番号になります。そのイメージを図 2.2.7.2.に示します。

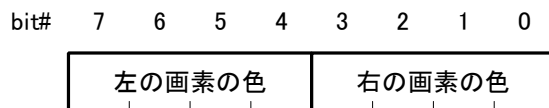


図 2.2.7.2. Pattern Name Table の 1byte のビットマップ

Pattern Name Table の先頭 1byte は、(0,0)と(1,0) のペアに対応しています。次の 1byte は、(2,0)と(3,0)のペア。次の 1byte は、(4,0)と(5,0)のペア、となります。(X,Y)に対応するアドレスは、下記の式で示されます。

$$(X \div 2) + (Y * 128)$$

SCREEN3 と比べれば非常にシンプルで分かりやすいですね。X and 1 が 0 なら上位 4bit, X and 1 が 1 なら下位 4 bit が対応しています。

MSX Documents VDP

では、実際に VRAM に書き込んでみましょう。左上から右下へ斜め線を描くサンプル(SC5LINE.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,4,7:SCREEN5:C(0)=&HC4:C(1)=&H4C
110 FOR Y=0 TO 211:X=Y\2:GOSUB130:NEXT
120 GOTO120
130 A=(X\2)+(Y*128):VPOKEA,C(XAND1):RETURN
```

写真 2.2.7.1. にこのサンプル(SC5LINE.BAS)を動かした結果の画面を示します。

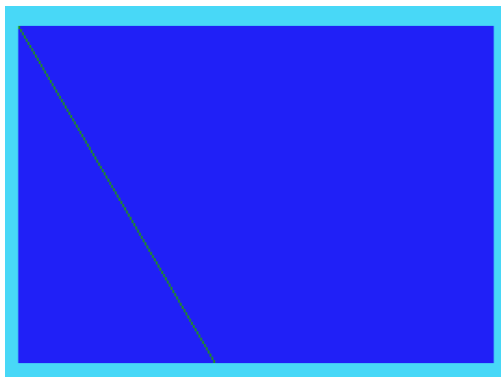


写真 2.2.7.1. SC5LINE.BAS の実行結果

MSX-BASIC では、SET PAGE 命令で表示や描画先を切り替えられます。VRAM64KB の機種では 2 ページ、128KB の機種では 4 ページから選択して指定できます。VDP の Pattern Name Table のアドレスは、実際に表示されている画面(つまり SET PAGE 命令では、第1引数の Display Page)に対応しており、これは VDP コントロールレジスタを書き換えることでアドレスを変更できます。また、描画先(同、第2引数の Active Page)は、何を表示しているかわからず、更新先なので VDP コマンドで所望の座標へ描画されるように指定するのみです。Pattern Name Table のアドレス指定の方法については、2.4.1. コントロールレジスタで説明します。

2.2.8. SCREEN6 (GRAPHIC5)

このモードは、V9938/V9958 のみで利用可能です。

VRAM を構成する要素の配置は、SCREEN5 と同じです。異なるのは画面の水平解像度が 512 に倍増して、代わりに1画素 2bit に半減している点です。1byte で水平 4ドットを構成しています。図 2.2.8.1.に 1byte の構成を示します。



図 2.2.8.1. Pattern Name Table の 1byte のビットマップ

画素値は 0～3 の4種類しかないので、カラーパレットも 0～3 の4種類しか使いません。4～15 のカラーパレットは未使用です。

Pattern Name Table のアドレス計算は、SCREEN5 に対して水平の画素数が増えたことと、1byte が 4ドットになったことで、X 座標の項のみ変化します。

$$(X \div 4) + (Y * 128)$$

では、実際に VRAM に書き込んでみましょう。左上から右下へ斜め線を描くサンプル(SC5LINE.BAS)を下記に示します。3 番の色で斜め線入れるサンプル(SC6LINE.BAS)です。

```

100 DEFINT A-Z:COLOR3,0,0:SCREEN6:C(0)=&H0:C(1)=&H30:C(2)=&H0C:C(3)=&H03
110 FOR Y=0 TO 211:X=Y\2:GOSUB 130:NEXT
120 GOTO 120
130 A=(X\4)+(Y*128):VPOKEA,C(XAND3):RETURN

```

MSX Documents VDP

写真 2.2.8.1. にこのサンプル(SC6LINE.BAS)を実行した結果を示します。

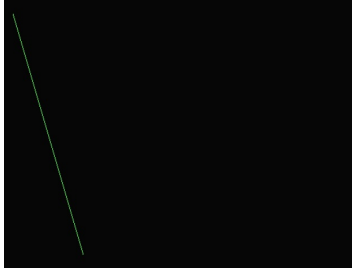


写真 2.2.8.1. SC6LINE.BAS の実行結果

2.2.9. SCREEN7 (GRAPHIC6)

このモードは、V9938/V9958 の VRAM128KB の機種のみで利用可能です。

この画面モードは、水平 512ドット、垂直 212ドットの高解像度グラフィックモードになります。1ドット 16 色使えるので、SCREEN5 と SCREEN6 のいいとこ取りのようなスペックですが、その分 VRAM の使用量が SCREEN5 や 6 と比べて倍増していることに注意しなければなりません。

また、SCREEN7 以上の画面モードでは、VRAM として使用している DRAM の利用方法が変化します。このあたりは、3. 解析で詳しく解説します。

Pattern Name Table の 1byte は、SCREEN5 と同じで、図 2.2.9.1. の構成になります。

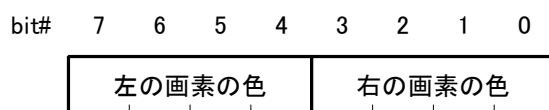


図 2.2.9.1. Pattern Name Table の 1byte のビットマップ

SCREEN5 は、水平 256ドットで、1byte が 2ドットを構成するので、1 ラインは 128byte でした。

SCREEN7 は、水平 512ドットで、1byte が 2ドットを構成するので、1 ラインは 256byte になります。従って、グラフィック座標 (X,Y) に対応する Pattern Name Address のアドレス計算は、下記のようになります。

$$(X \div 2) + (Y * 256)$$

MSX Documents VDP

下記に直線を描画するサンプル(SC7LINE.BAS)を示します。

```
100 DEFINT A-Z:COLOR15,4,7:SCREEN7:C(0)=&HC4:C(1)=&H4C
110 FOR Y=0 TO 211:X=Y\2:GOSUB130:NEXT
120 GOTO120
130 A!=(X\2)+(Y*256):VPOKEA!,C(XAND1):RETURN
```

SCREEN7 の1画面は 32767byte を越えているので、アドレス計算をすると 32768 以上の値が発生します。MSX-BASIC の整数型変数は -32768～32767 の範囲しか採れませんので、オーバーフローが発生ないように 130 行目の A は ! を付けて単精度実数型にしてあります。

写真 2.2.7.1. にこのサンプル(SC7LINE.BAS)を動かした結果の画面を示します。

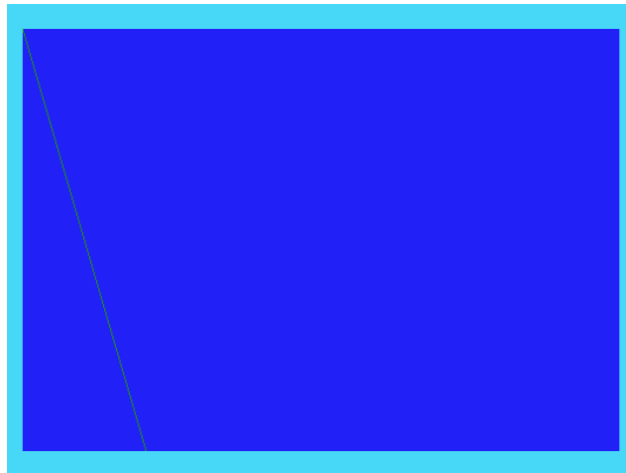


写真 2.2.7.1. SC7LINE.BAS の実行結果

2.2.10. SCREEN8 (GRAPHIC7)

このモードは、V9938/V9958 のみで利用可能です。

この画面モードは、水平 256ドット、垂直 212ドットのグラフィックモードになります。1ドット 256 色使えますが、SCREEN0～7と異なり、カラーパレットではなく固定の 256 色です。VRAM の構成としては SCREEN7と同じになります。

また、SCREEN7 以上の画面モードでは、VRAM として使用している DRAM の利用方法が変化します。このあたりは、3. 解析で詳しく解説します。

Pattern Name Table の 1byte は、1ドットに対応しており、図 2.2.10.1.の構成になります。

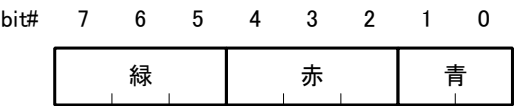


図 2.2.10.1. Pattern Name Table の 1byte のビットマップ

特徴的なのは、灰色が存在しないところです。赤と緑は 0～7 の 8 通り。青は 0～3 の 4 通りの階調を持っていて数値が大きい方が明るく、(R,G,B) = (7,7,3) が白、(R,G,B)=(0,0,0) が黒になります。赤・緑の階調と明るさの割合を表 2.2.10.1.に、青の階調の明るさの割合を表 2.2.10.2.に示します。

表 2.2.10.1. 赤・緑の、"階調"と"明るさの割合"の対応表

階調値	明るさの割合
0	0%
1	14.3%
2	28.6%
3	42.9%
4	57.1%
5	71.4%
6	85.7%
7	100%

表 2.2.10.2. 青の、"階調"と"明るさの割合"の対応表

階調値	明るさの割合
0	0%

MSX Documents VDP

1	33.3%
2	66.6%
3	100%

例えば、(R,G,B) = (2,2,1) の場合、明るさの割合としては (28.6%, 28.6%, 33.3%) となり、やや青みがかった灰色になります。

SCREEN8 は、水平 256ドットで、1byte が 1ドットを構成するので、1 ラインは 256byte になります。従って、グラフィック座標 (X,Y) に対応する Pattern Name Address のアドレス計算は、下記のようにになります。

$$X + (Y * 256)$$

下記に直線を描画するサンプル (SC8LINE.BAS) を示します。

```
100 DEFINT A-Z: SCREEN8: COLOR255, 0, 0: CLS
110 FOR Y=0 TO 211: X=Y\2: GOSUB 130: NEXT Y
120 GOTO 120
130 A!=X+(Y*256): VPOKEA!, 123: RETURN
```

この実行結果を写真 2.2.10.1. に示します。

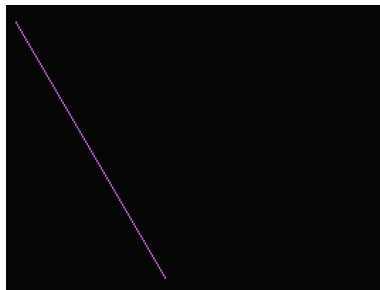


写真 2.2.10.1. SC8LINE.BAS の実行結果

2.2.11. SCREEN9

このモードは、ハングル文字を表示するモードらしいですが、日本向け MSX では使用できません。私は詳細を知らないで割愛します。VDP にはこのモードに対応する専用のモードは存在しないので、ハングル版 MSX-BASIC がソフトウェア的に追加している機能だと思います。

2.2.12. SCREEN10 (GRAPHIC7, YJK+RGB mode)

このモードは、V9958 のみで利用可能です。

この画面モードは、水平 256ドット、垂直 212ドットのグラフィックモードになります。Pattern Name Table サイズは SCREEN8と同じですが、連続する 4byte が組になっており、4byte で 水平連続 4ドットを表現します。

Pattern Name Table の 4byte 組は図 2.2.12.1.の構成になっています。

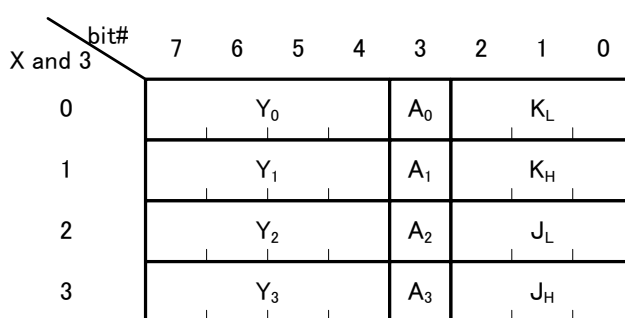


図 2.2.12.1. Pattern Name Table の 4byte 組のビットマップ

YJK 方式と呼ばれる VDP 固有の方式で色を表現します。YJK から RGB への変換方法については SCREEN12 のところで詳しく説明します。

bit3 の A0～A3 ですが、各ドットにおけるパレットフラグになります。ここが 1 になっているドットは、対応する Yn をカラーパレット番号と見なして、そのカラーパレットの色になります。0 になっているドットは、Yn は輝度値になります。bit7～bit3 の 5bit で 0～31 の輝度を表現しますが、bit3 は 0 なので実質 0,2,4, ..., 30 の 16 階調になります。

KH を上位 3bit, KL を下位 3bit とする 6bit の値 K と、JH を上位 3bit, JL を下位 3bit とする 6bit の値 J とで色相を表現しています。K と J は符号付きの値なので、それぞれの定義域は -32～31 の 64 通りになります。

16 階調×K が 64 通り×J が 64 通り = 65536 通り、65536 色も出るの?と思うかもしれませんが、残念ながらそうなりません。このあたりも、SCREEN12 のところで演算式を交えながら詳しく説明します。

MSX Documents VDP

【note】

一般に YUV 方式と呼ばれる色空間の表現方法が存在しますが、YUV→RGB の変換マトリクス演算では、変換係数にある程度の小数精度が必要になります。V9958 の時代は、除算回路は回路規模が大きすぎて使えない、乗算器も大きいので極力使いたくない、といった感じだと思います。変換マトリクスは定数行列を掛ける演算になるので、例えば小数8ビットの精度を持たせるのであれば、8ビット値を掛けて、右8ビットシフトするような演算になります。乗算器が必要になりますね。そこで、この「掛ける値」を2のN乗に近似してしまうことを考えたのでしょう。すると、乗算器は左シフトに置き換えられるようになります。左シフトは、回路上では配線だけなので追加回路は必要ありません。回路規模を抑えるための苦肉の策で生まれたのが YJK 方式なのではないかなと、私は想像しています。

2.2.13. SCREEN11 (GRAPHIC7, YJK+RGB mode)

このモードは、V9958 のみで利用可能です。

VDPとしては SCREEN10 と全く同じです。MSX-BASIC の挙動が異なります。

2.2.14. SCREEN12 (GRAPHIC7, YJK mode)

このモードは、V9958 のみで利用可能です。

この画面モードは、水平 256ドット、垂直 212ドットのグラフィックモードになります。Pattern Name Table サイズは SCREEN8と同じですが、連続する 4byte が組になっており、4byte で 水平連続 4ドットを表現します。

Pattern Name Table の 4byte 組は図 2.2.14.1.の構成になっています。

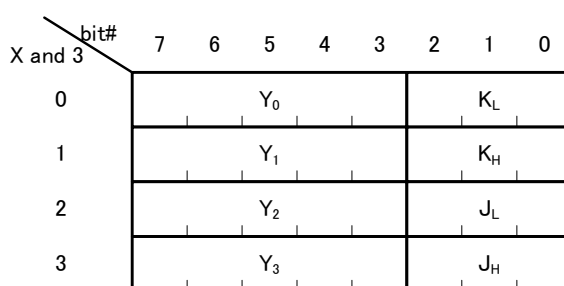


図 2.2.14.1. Pattern Name Table の 4byte 組のビットマップ

SCREEN10, 11 と同じ YJK 方式の画面モードになります。SCREEN10,11 と異なるのは、図 2.2.14.1.に記載の通り、A 0~A3 が無くなり、bit3 が Y のビットになっている点です。そのため、SCREEN12 では 32 階調になっています。

32 階調×K が 64 通り×J が 64 通り = 131072 通り、131072 色も使えるの?というと、残念ながら使えません。

使えない理由を理解するために、まず YJK から RGB への変換について説明します。

VDP は内部で YJK 値を RGB 値に変換して、同時に表示するスプライトや、SCREEN10,11 ならカラーパレット値と揃えて重ね合わせて、そこからモニターへの出力信号を生成しています。そのため、一端 RGB へ変換する課程を理解する必要があります。

YJK から RGB への変換は下記を用います。

$$R = Y + J$$

$$G = Y + K$$

$$B = 1.25 Y - 0.5 J - 0.25 K$$

MSX Documents VDP

この式を 符号付き右シフト演算子 \gg , 符号付き左シフト演算子 \ll , 下位 2bit 落として四捨五入する $\text{sround2}()$ を使って書き換えると、下記ようになります。

$$R = Y + J$$

$$G = Y + K$$

$$B = \text{sround2}((Y \ll 2) + Y) - ((J \ll 1) + K)$$

ここで定義域を考えてみます。Y は 0～31, J と K は -32～31 です。Y=0 が 0%の輝度、Y=31 が 100%の輝度になるので、RGB の値域は、0～31 になります。

そうすると、Y=0 は黒となり、色は無関係になります。Y=0 のときに J,K が 0 以外(つまり色指定がある)だと、上記計算式では RGB は値域をはみ出してしまいます。例えば、(Y,J,K)=(0,-32,-32) の場合を考えてみましょう。

$$R = 0 + (-31) = -31$$

$$G = 0 + (-31) = -31$$

$$B = \text{sround2}((0 \ll 2) + 0) - (((-32) \ll 1) + (-32)) = 24$$

R や G は -100%近い値、B に至っては 200%近い値になっています。当時の出力モニターはそういった指定を受けられないため、VDP は値域に収まるようにクリッピングします。クリッピング関数を $\text{CLIP}(n)$ とすると、

$$R = \text{CLIP}(Y + J)$$

$$G = \text{CLIP}(Y + K)$$

$$B = \text{CLIP}(\text{sround2}((Y \ll 2) + Y) - ((J \ll 1) + K))$$

となります。従って、(Y,J,K)=(0,-32,-32)の場合は、(R,G,B)=(0,0,24)となります。0 が 0%, 31 が 100% なので、 $24=24/31=77\%$ 。77%の明るさの青になるわけです。

クリッピング関数の入出力特性を図 2.2.14.2.に示します。

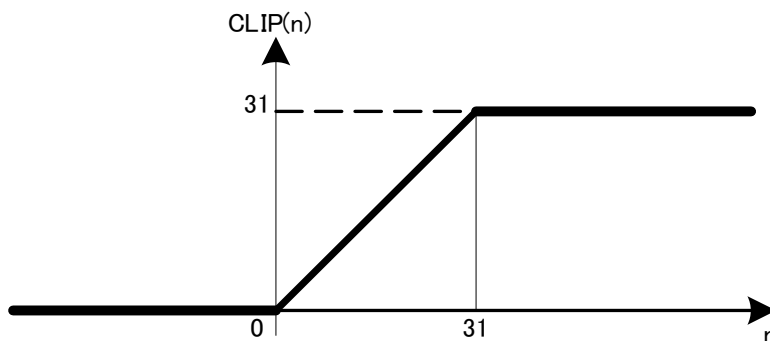


図 2.2.14.2. CLIP(n) の入出力特性

この CLIP() によって値が丸められることによって、複数の YJK が、一つの RGB に対応する組み合わせになるため、YJK の組み合わせの数よりも表現可能な色数が少なくなることを示しています。

また、RGB の値域が各要素 0～31 の 32 通りだとすると、この組み合わせは 32 の 3 乗で 32768 通りありますが、Y, J, K が整数であることから RGB の取り得る全組み合わせを網羅できず、これよりも少ない数になります。具体的には、上記右シフト演算 >> によって小数部にはみ出したビットはカットされます。

このように、演算精度や値域の関係で値の丸め処理が入ることによって表現可能な色数が減っており、SCREEN12 では 19268 色となっています。演算精度を上げればもう少し色数が増えたと思いますが、精度増加による回路規模増加のコストと、19268 色でも当時は自然画と呼べるくらいには多色だったことから、これで十分という判断が下されたのだと思います。

ちなみに、SCREEN10 では 12499 色+16 色パレットとなります。

現在のパソコンを使って色数を数える Python3 スクリプト(sc12colcnt.py)を下記に示します。

```
#!/usr/bin/env python3
# coding=utf-8
```

```
def clip( i ):
    if i < 0:
        return 0
    if i > 31:
        return 31
    return i
```

```
def sround2( i ):
```

MSX Documents VDP

```
if i < 0:
    return (i + 1) >> 2
return (i + 2) >> 2

color_ref = {}
for y in range( 0, 32 ):
    for j in range( -32, 32 ):
        for k in range( -32, 32 ):
            r = clip( y + j )
            g = clip( y + k )
            b = clip( sround2((y << 2) + y - ((j << 1) + k)) )
            color_ref[ ( r, g, b ) ] = True
print( "SCREEN12 COLOR NUM IS %d." % len( color_ref ) )

color_ref = {}
for y in range( 0, 32, 2 ):
    for j in range( -32, 32 ):
        for k in range( -32, 32 ):
            r = clip( y + j )
            g = clip( y + k )
            b = clip( sround2((y << 2) + y - ((j << 1) + k)) )
            color_ref[ ( r, g, b ) ] = True
print( "SCREEN10/11 COLOR NUM IS %d." % len( color_ref ) )
```

(Y,J,K) = (0,-32,-32) が 77%の明るさの青であることを確認するサンプル(SC12BLUE.BAS)を下記に示します。

```
100 SCREEN12:COLOR255,0,0:CLS
110 FORX=1TO255STEP2:LINE(X,0)-(X,211),4:NEXTX
120 GOTO 120
```

写真 2.2.14.1.に SC12BLUE.BAS の実行結果を示します。



写真 2.2.14.1. SC12BLUE.BAS の実行結果

J, K は 2 の補数表現による符号付き整数となっています。Z80 の符号付きの数値と同じですね。

MSX Documents VDP

なので、 $-32 = \&B100000$, $-31 = \&B1000001$, $-30 = \&B1000010$, ..., $-1 = \&B111111$, $0 = \&B000000$, $1 = \&B000001$, ..., $30 = \&B0111110$, $31 = \&B0111111$ という対応になっています。

-32 は $\&B100000$ なので、上位 3bit と下位 3bit は、それぞれ $\&B100$, $\&B000$ となります。これを Pattern Name Table のビットマップに当てはめると、図 2.2.14.2. のようになります。

X and 3	bit#							
	7	6	5	4	3	2	1	0
0	Y ₀				$\&B000$			
1	Y ₁				$\&B100$			
2	Y ₂				$\&B000$			
3	Y ₃				$\&B100$			

図 2.2.14.2. (J,K)=(-32,-32) のビットマップイメージ

Y=0 のケースを確認したいので、4ドットすべて Y=0、つまり Y₀=0, Y₁=0, Y₂=0, Y₃=0 にします。図 2.2.14.3. のようになりますね。

X and 3	bit#							
	7	6	5	4	3	2	1	0
0	$\&B000000$				$\&B000$			
1	$\&B000000$				$\&B100$			
2	$\&B000000$				$\&B000$			
3	$\&B000000$				$\&B100$			

図 2.2.14.3. (Y_n,J,K) = (0,-32,-32) のビットマップイメージ

$\&B00000000 = 0$, $\&B00000100 = 4$ なので、X 座標が偶数の列は 0, X 座標が奇数の列は 4 で画面を敷き詰めれば、画面全体が (Y,J,K)=(0,-32,-32) の色で塗りつぶされることになります。

ということで、サンプル(SC12BLUE.BAS)は、COLOR255,0,0:CLS で画面全体を all 0 でクリアした後に、X 座標が奇数の列にのみ LINE 命令で 4 の色の縦線を描いて (0,-32,-32) を作り出しています。

2.3. VDP へのアクセス詳細

本章では、VDP へのアクセス方法に関して 2.1. 章よりも詳細に述べる。

VDP へのアクセスは、I/O BUS 経由であることは先に述べたとおりです。CPU からは I/O 制御によって VDP にアクセスするわけですが、それには決まった手順があるので、その手順について説明します。

2.3.1. I/O 制御によるアクセス

I/O 制御によるアクセスには表 2.3.1.1. に示す種類が存在します。目的に応じて適切な種類のアクセス方法を選択してください。また、MSX-BASIC の OUT 命令・INP 関数によるアクセスではうまくいかないことが多いです。その理由については、2.3.2. MSX-BASIC からのアクセスにて説明します。

表 2.3.1.1. I/O 制御によるアクセスの種類

アクセスの種類	対応VDP	概要
コントロールレジスタライト	TMS9918/V9938/V9958	VDPのコントロールレジスタに値を書き込む手段。
VRAMライト	TMS9918/V9938/V9958	VDPを介してVRAMへ書き込む手段。
VRAMリード	TMS9918/V9938/V9958	VDPを介してVRAMを読み出す手段。
ステータスレジスタリード	TMS9918/V9938/V9958	VDPのステータスレジスタを読み出す手段。
カラーパレットライト	V9938/V9958	VDPのカラーパレットレジスタへ書き込む手段。

VDP の I/O ポートは、一般的に 98h～9Bh になります。これがそれぞれ Port#0～Port#3 に対応します。しかし、MSX バージョンアップアダプタを介した MSX1 上の V9938 の I/O ポートは本体 VDP との I/O 競合を避けるために別のアドレスに割り当てられています。BIOS を使った動作(CHGMOD で SCREEN を変えるなど)は、これらの違いを吸収してくれます。画面表示に関しては速度を要することが多いので I/O を直接扱うことが許可されていますが、バージョンアップアダプタのようなケースに対応できるようにするために、MAIN-ROM の 6 番地・7 番地に VDP の I/O アドレスが格納されています。

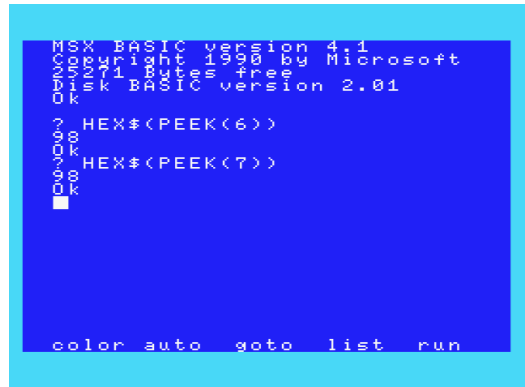


写真 2.3.1.1. MAIN-ROM の 6 番地・7 番地の内容

写真 2.3.1.1.は、MSX-BASIC から MAIN-ROM の 6 番地・7 番地を読んだイメージです。通常、このように 6 番地と 7 番地に 98h が書き込まれており、Port#0～Port#3 は 98h～9Bh に対応しています。

ちなみに、MSX バージョンアップアダプタを取り付けた場合、アダプタ内に MSX2 相当の MAIN-ROM を搭載して機能アップしています。この MAIN-ROM 6 番地・7 番地には 88h になっているようです。MSX 規格上のルールとして、Read ポートは 6 番地に書かれているアドレス、Write ポートは 7 番地に書かれているアドレスとされているようですが、実際に 6 番地と 7 番地に異なる内容が書かれている市販デバイスは無さそうです。MSX バージョンアップアダプタを無視すれば、VDP は 98h～9Bh と決め打ちアドレスで問題ないかと思います。

各 Port と、その I/O アドレスの求め方 (具体的なアドレス) と、用途を表 2.3.1.1.にまとめておきます。

VDP I/O アドレス名	具体的なアドレス	用途
Port#0 (Read)	PEEK(6)	VRAM読み出し
Port#0 (Write)	PEEK(7)	VRAM書き込み
Port#1 (Read)	PEEK(6)+1	ステータスレジスタ読み出し
Port#1 (Write)	PEEK(7)+1	コントロールレジスタ書き込み
Port#2 (Write)	PEEK(7)+2	パレットレジスタへの書き込み
Port#3 (Write)	PEEK(7)+3	間接指定のコントロールレジスタ書き込み

表 2.3.1.1. VDP の I/O アドレス

2.3.1.1. コントロールレジスタライト

VDPはコントロールレジスタと呼ばれる Write Only なレジスタを持っています。Write Only なので**書き込んだ値を読み出すことはできません**。

コントロールレジスタへの書き込みは、VDP に対して VDP の挙動を指示するものであり、例えば SCREEN モードを変更するのもこのコントロールレジスタへの書き込みで行われます。

コントロールレジスタへの書き込みは、図 2.3.1.1-1.に示す手順で行います。

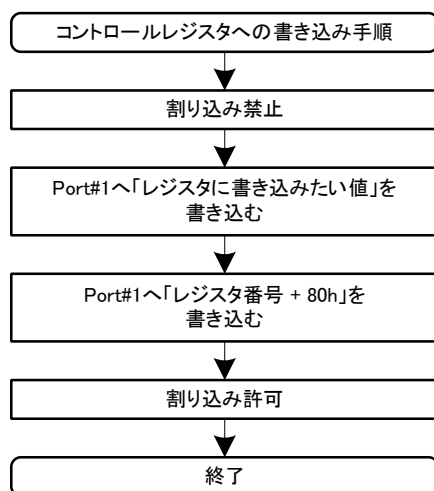


図 2.3.1.1-1. コントロールレジスタへの書き込み手順

通常、この I/O ポートに関して、VDP はアクセス待ちの状態になっています。Port#1 に何か書き込まれると、VDP は「2 つ目の書き込みを待機する状態」に遷移します。この状態でもう一度 Port#1 に書き込むとその内容に応じて挙動を決めます。2 つ目の Port#1 の書き込みの上位 2bit が 2 進数で 10 だった場合に「コントロールレジスタへの書き込みだ」と解釈されます。

このように、Port#1 へのアクセスで VDP の内部状態が変化するわけですが、MSX-BIOS は割り込み処理ルーチンの中で VDP にアクセスしています。たまたま上記手順の Port#1 へのアクセスの 1 回目と 2 回目の間に割り込まれて VDP にアクセスされると、割り込み処理ルーチンから戻ってきた後に実行される「2 回目の書き込み」の時には VDP の状態が期待する状態から別の状態に遷移していることになり、おかしい挙動になってきます。このような理由から、この 2 連続のアクセスは割り込み禁止して間に割り込まれないように保護する必要があります。

以下にコントロールレジスタへの書き込みサンプルを提示します。

MSX Documents VDP

```
; -----  
; 入力  
;   E ..... 書き込み先の VDP コントロールレジスタ番号  
;   D ..... 書き込むデータ  
; 出力  
;   なし  
; 破壊  
;   AF, BC  
; 備考  
;   割り込み禁止・割り込み解除を実施します  
; -----  
write_vdp_control_register::  
    ld    a, [0x0007]      ; 書き込み用 Port#0 を取得  
    inc   a                ; 書き込み用 Port#1 を取得  
    ld    c, a  
  
    ld    a, e  
    or    a, 0x80          ; a = 0x80 | VDP コントロールレジスタ番号  
  
    di  
    out   [c], d  
    out   [c], a  
    ei  
    ret
```

MSX バージョンアップアダプタを無視する場合は、7 番地を読む処理を省略できます。例えば、MSX2+以上向けのソフトであれば MSX バージョンアップアダプタを無視しても問題ないでしょう。そのようなケースのサンプルを下記に提示します。

```
; -----  
; 入力  
;   A ..... 書き込み先の VDP コントロールレジスタ番号  
;   D ..... 書き込むデータ  
; 出力  
;   なし  
; 破壊  
;   AF, C  
; 備考  
;   割り込み禁止・割り込み解除を実施します  
; -----
```


MSX Documents VDP

```
write_vdp_control_register2::  
    ld        c, 0x99  
    or        a, 0x80          ; a = 0x80 | VDP コントロールレジスタ番号  
    di  
    out       [c], d  
    out       [c], a  
    ei  
    ret
```

例えば、垂直スクロールレジスタ R#23 に 100 を書き込むのであれば、上記ルーチンを使って下記のように実行します。

```
set_vdp_r23::  
    ld        a, 23  
    ld        d, 100  
    call      write_vdp_control_register2  
    ret
```

BIOS では、VDP のコントロールレジスタに書き込んだ値を BIOS ワークエリア内に保持しています。VDP のコントロールレジスタがすべて write only のため、実際に設定された値を読み出すことができません。しかしながら、8bit の設定値のうち変更したいのが一部の bit のみの場合、書き換える直前の値が何だったのかを知りたいケースがあります。知りたくなった場合、BIOS ワークに保持している「最後に書き込んだ値」を読み取って加工後書き戻すという使い方になります。

注意すべきは、上記に用意した write_vdp_control_register 及び同 2 は、この BIOS ワークを変更しません。MSX-BASIC に戻ると実際に設定されている値と、保持している値との間に矛盾が出てきます。処理後に MSX-BASIC へ戻るアプリの場合、そのあたりに留意すべきです。

コントロールレジスタの種類については、2.4.1. コントロールレジスタで詳細を説明します。

2.3.1.2. ステータスレジスタリード

VDPはステータスレジスタと呼ばれる Read Only なレジスタを持っています。VDP の状態を示す値が読み出せるレジスタで、書き込みはできません。

TMS9918 は、ステータスレジスタが 1 つのみだったため、MSX1 では Port#1 を読み出すことは、唯一あるステータスレジスタを読み出すことに一致していましたが、V9938/V9958 ではステータスレジスタが 10 個に増加しています。ステータスレジスタは S#0～S#9 の 10 個あり、TMS9918 のステータスレジスタは S#0 に相当します。

ステータスレジスタ#0 の読み出し手順を図 2.3.1.2-1 に示します。

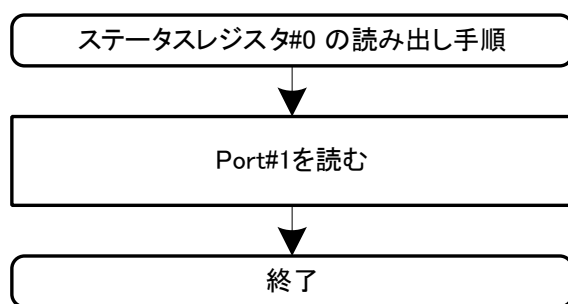


図 2.3.1.2-1. S#0 の読み出し手順

具体的なステータスレジスタ S#0 の読み出し方は下記の通り。

```

; -----
; 入力
;   なし
; 出力
;   A ..... 読み出した値
; 破壊
;   AF, BC
; -----
read_vdp_status_register0::
    ld     a, [0x0006]      ; 読み出し用 Port#0 を取得
    inc    a                ; 読み出し用 Port#1 を取得
    ld     c, a
    in     a, [c]
    ret
  
```

MSX バージョンアップアダプタを無視するなら、下記の通り。

MSX Documents VDP

```
; -----  
; 入力  
;   なし  
; 出力  
;   A ..... 読み出した値  
; 破壊  
;   AF, BC  
; -----  
read_vdp_status_register0::  
    in     a, [0x99]  
    ret
```

もはや、サブルーチンにするのももったいないくらい短いコードですね。

V9938/V9958 から増設された S#1～S#9 を読み出す場合は、コントロールレジスタ R#15 に読み出したいステータスレジスタの番号を設定してから Port#1 を読み出すことで実現します。

しかし、その実行前に BIOS の構造を知っておく必要があります。

MSX は、VDP の垂直帰線割込を受けると 0038h 番地に飛びます。どの割り込みでも 0038h に来るため、BIOS は VDP の垂直帰線割込かどうかを知るために、VDP S#0 を読み出しています。この読み出しがまさに上記 S#0 の読み出し方と同じになっています。ここからは想像ですが、垂直帰線割込は 1 秒間に約 60 回(NTSC)または 50 回(PAL)の割り込みを発生させます。MSX にとっては結構頻繁に入る割り込みですね。MSX1 の時には `in a, [Port#1]` すればよかったものが、V9938 になったときに R#15 への設定も追加してしまうと処理が重くなってしまうので MSX2 以降も `in a,[Port#1]` のままになっています。R#15 は Port#1 から読めるステータスレジスタの番号を指定するレジスタですが、BIOS との共存のため、R#15 を 0 以外にしている間、割り込み禁止を維持して、必要なステータスレジスタ読み出しが終わったら R#15 を 0 に戻してから割り込み許可にする必要があります。MSX-DOS の場合でも上記に特別な対策を打っていない限りは、同様です。

これらを踏まえて、ステータスレジスタの読み出し手順を図 2.3.2.1-2. に示します。

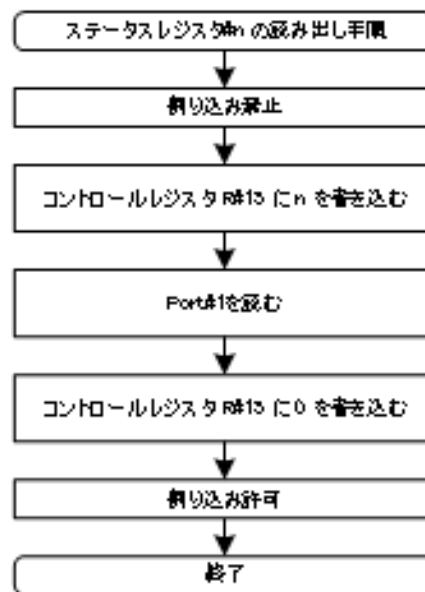


図 2.3.2.1-2. S#n の読み出し手順

具体的なステータスレジスタ S#n の読み出し方は下記の通り。

```

; -----
; 入力
;   B ..... 読みたいステータスレジスタの番号
; 出力
;   A ..... 読み出した値
; 破壊
;   AF, BC
; -----
read_vdp_status_register0::
    ld     a, [0x0007]      ; 書き込み用 Port#0 を取得
    inc    a
    ld     c, a
    ld     a, 15 | 0x80
    di
    out    [c], b
    out    [c], a

    ld     a, [0x0006]      ; 読み出し用 Port#0 を取得
    inc    a                ; 読み出し用 Port#1 を取得
    ld     c, a
    in     b, [c]
  
```

MSX Documents VDP

```
ld      a, [0x0007]      ; 書き込み用 Port#0 を取得
inc     a
ld      c, a
xor     a, a
out     [c], b
out     [c], a
ei
ld      a, b
ret
```

MSX バージョンアップアダプタを無視するなら、下記の通り。

```
; -----
; 入力
;   B ..... 読みたいステータスレジスタの番号
; 出力
;   A ..... 読み出した値
; 破壊
;   AF, BC
; -----
read_vdp_status_register0::
ld      c, 0x99
ld      a, 15 | 0x80
di
out     [c], b
out     [c], a
in      b, [c]
xor     a, a
out     [c], a
ld      a, 15 | 0x80
out     [c], a
ei
ret
```

ステータスレジスタの種類については、2.4.2. ステータスレジスタで詳細を説明します。

【note1】

MSX-DOS の場合、CPU メモリの Page0 が DRAM になっており、0038h 番地を書き換えることができます。割り込み処理をすべて自分で行う前提で、0038h を乗っ取ってしまうと、S#0 を読む前に R#15 を 0 に戻す処理を追加することもできます。ただ、割り込み処理内でモーター制御している外付けFDDなどもあるので、うかつな乗っ取り方をするとそのような一部の環境で異常な動作をするケースがあるので、あまりお勧めできません。

MSX Documents VDP

【note2】

0038h から実行される BIOS ルーチンは、真っ先に H.KEYI を呼びます。そこから戻ってくると、S#0 を読んで垂直帰線割込であれば H.TIMI を呼ぶ構造になっています。H.KEYI をフックして、R#15 を 0 に戻す処理を入れてしまえば、メインルーチン内で R#15 を 0 に戻す処理は必要ありませんが、ステータスレジスタを呼んでいる間割込禁止にしなければならない状況はわかりません。(R#15 を所望の値に変えてから、ステータスレジスタを読む前に割り込まれて R#15 に戻されてしまうかもしれませんから)。

2.3.1.3. パレットレジスタ

V9938/V9958 にはパレットレジスタと呼ばれる内部レジスタが存在します。この機能は TMS9918 には存在しません。

SCREEN0～7, 10, 11 では背景やスプライトに 16 色利用できますが、この 16 色の色を決定するのがパレットレジスタになります。光の三原色で色を指定する方式で、赤・緑・青それぞれ 0～7 の 8 段階指定することができます。

MSX-BASIC の COLOR=(パレット番号, 赤, 緑, 青) がこれに該当します。試しにパレット番号 4 を、(R,G,B)=(2,3,4) に変更してみたのが写真 2.3.1.3-1. です。

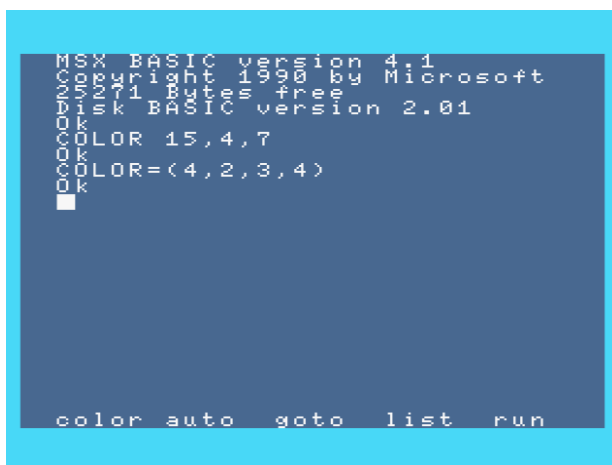


写真 2.3.1.3-1. パレット番号 4 のパレットを (2,3,4) に変更

I/O アクセスによるパレット書き換えは、まずコントロールレジスタ R#16 にパレット値を書き換えたいパレット番号を指定します。次に bit6-4 に赤レベル、bit2-0 に青レベルを指定した値を書き込み用 Port#2 に書き込みます。さらに続けて bit2-0 に緑レベルを指定した値を書き込み用 Port#2 に書き込みます。このシーケンスを図 2.3.1.3-1. に示します。

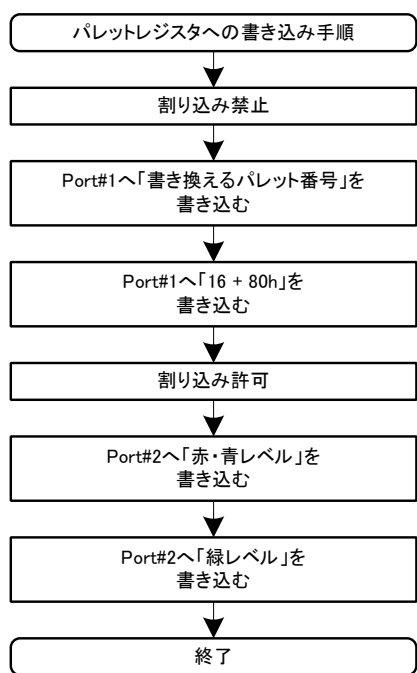


図 2.3.1.3-1. パレットレジスタ書き込みシーケンス

Port#2 へ書き込むたびに内部カウンタはインクリメントしていますので、すぐ次のパレットを変更する場合は R#16 への書き込みは省略してもかまいません。図 2.3.1.3-2.に複数のパレット書き換えシーケンスを示します。

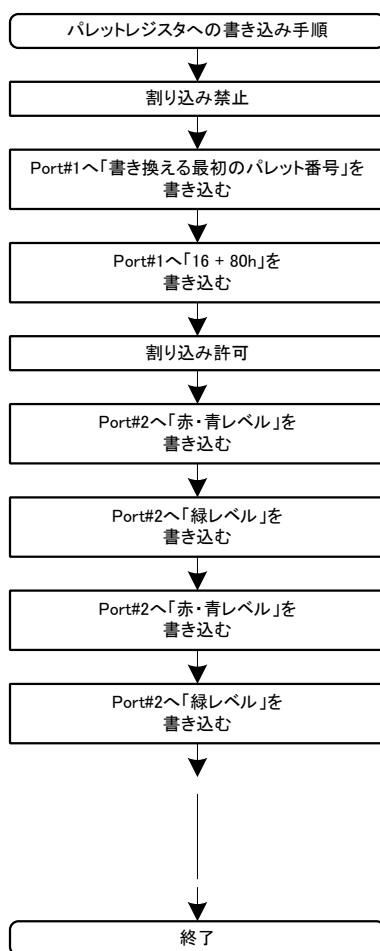


図 2.3.1.3-2. 複数のパレットを連続書き換え

これによりパレットを変更できます。

MSX-BASIC の COLOR=(パレット番号,赤,緑,青) を実行すると、VRAM の Palette Table と呼ばれる領域に、パレットの値が書き込まれますが、これは MSX-BASIC が COLOR=() 命令を実行するときに書き込んでいるだけです。そのため、上記シーケンスにより I/O 制御した場合、Palette Table は更新されません。Palette Table から値を取り直してパレットレジスタに設定する命令 COLOR=RESTORE で元の色に戻ってしまうのを確認できます。

では、MSX-BASIC から I/O アクセスによるパレット更新も試してみましょう。詳しくは 2.3.2. で説明しますが、VDP コントロールレジスタ R#16 に対応するシステム変数は VDP(17) になります。(※VDP(16)でないことに注意！)

VDP(17)=4:OUT&H9A,&H24:OUT&H9A,&H03

MSX Documents VDP

これを実行すると、写真 2.3.1.3-2.のようになります。

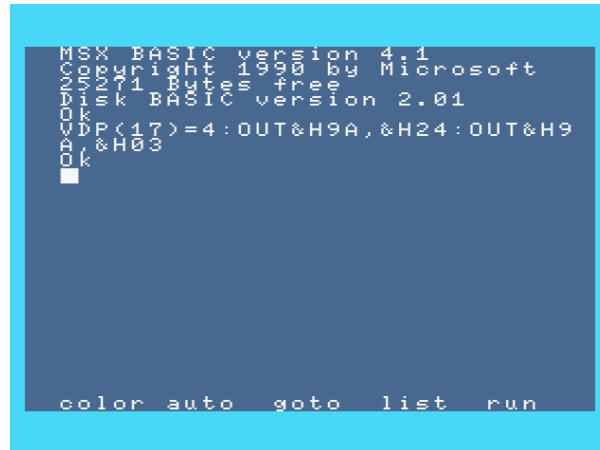


写真 2.3.1.3-2. MSX-BASIC から I/O アクセスによるパレット変更

当然これは VRAM の Palette Table を書き換えないので、COLOR=RESTORE で元の色に戻ります。

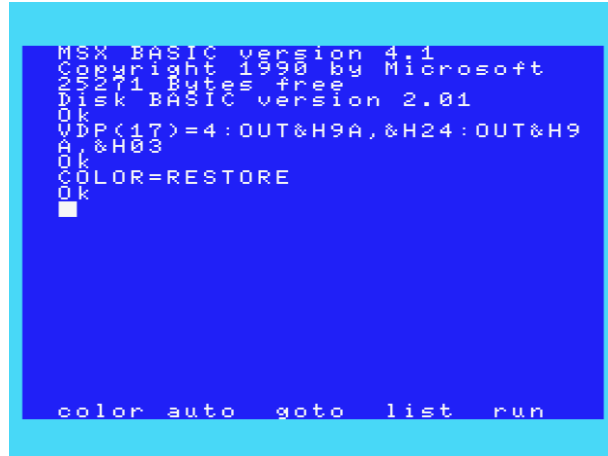


写真 2.3.1.3-3. 続けて COLOR=RESTORE を実行

【note】

VDP へのアクセスは、割込禁止で実行しなければダメじゃ無かったのか?と疑問に思われた方も居るかもしれませんが、補足しておきます。

割込禁止にする理由は、処理の途中に割り込まれることにより、Port#1 に対する VDP 内部のステートマシンの状態変数が割り込み処理内で変えられると処理シーケンスが崩れて期待通りに処理できないから、です。そして、初期状態で BIOS が割り込み処理内で VDP にアクセスするのは、「S#0 の読み出しと見なした Port#1 の読み出し」のみです。Port#2 に対する書き込

MSX Documents VDP

みアクセスは行われません。従って、Port#2 に対する書き込みは割込禁止にする必要はありません。しかし、同様の理由で、「割込フックを使って自身の割り込み処理を追加して、その中で Port#2 へ書き込んでいる場合」は割込禁止にする必要があります。

VDP()への書き込みアクセスは、BIOS の VDP レジスタ更新ルーチンを介してアクセスされます。当然ながら BIOS の VDP レジスタ更新ルーチンは適切な割込禁止を実施するので問題が起きないわけです。

2.3.1.4. VRAM への書き込み

VRAM への書き込みには、アドレス指定→値書き込みの順で行います。TMS9918 の場合はアドレスは 14bit、V9938 /V9958 の場合は 17bit+1bit 存在します。アドレスのうち一部はコントロールレジスタへ、残りは Port#1 への書き込みで VDP へ通知します。アドレス指定を終えた後に Port#0 へ書き込むと、指定のアドレスへ書き込まれ、アドレスは自動的にインクリメントされます。連続するアドレスへの書き込みであれば、Port#0 へ連続して書き込むことでアドレス指定を省略することができます。処理のフローを図 2.3.1.4-1.に示します。

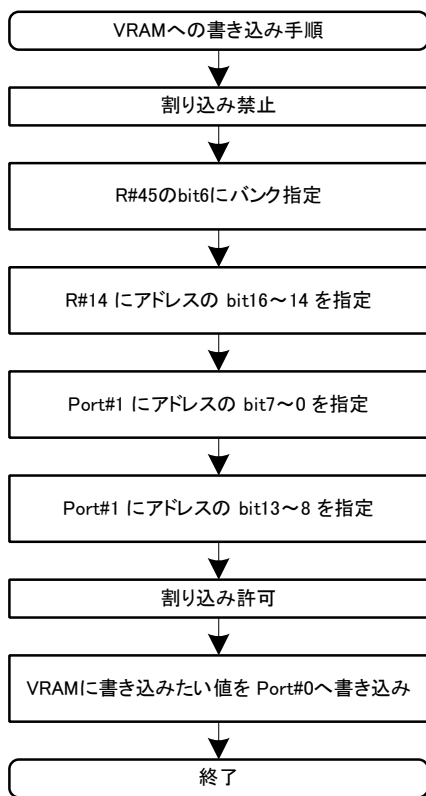


図 2.3.1.4-1. VRAM への書き込みフロー

R#45 への書き込みですが、これは通常省略します。V9938/9958 は 128KB の VRAM の前半の 64KB の裏にもう一組 VRAM を搭載することができます。いわば、R#45 の bit6 に指定するのはアドレスの bit17 に相当する値です。しかしながら、この裏 VRAM を搭載した MSX2/2+/turboR 製品は、私の知る限りでは存在していませんので、0 固定で使うのが一般的です。

【note】

MSX Documents VDP

改造により、DRAMを増設して裏 VRAM を使えるようにしている人もいて、ごく一部のソフトがこの裏 VRAM に対応していたりします。VDP コマンドで、表 VRAM から裏 VRAM への転送などが可能であればアドレスの bit17 と言えたのでしょうか、実際は表と裏の間でやりとりする転送などには非対応で、まるごと VRAM をすり替える(バンク切り替える)機能となっていて、使い勝手がいいまいです。

R#14 には、TMS9918 には存在しないアドレスビット(bit16~14)を指定します。これも値が保持されるので、前回のアクセス時から変更のない場合は、R#14 への書き込みを省略して構いません。ただし、アドレスオートインクリメントはこのアドレスビットにも作用するのでご注意ください。

次に Port#1 へ最下位 8bit と、上位 6bit を続けざまに書きます。上位 6bit を書き込むときに bit6 を立てる必要があります。bit6 を 0 にすると読み出し、1 にすると書き込みの意味になるためです。TMS9918 の場合は、R#45 も R#14 も存在しませんので、Port#1 のアクセスだけでアドレス指定が完結します。

TMS9918 では、図 2.3.1.4-2.のシーケンスになります。

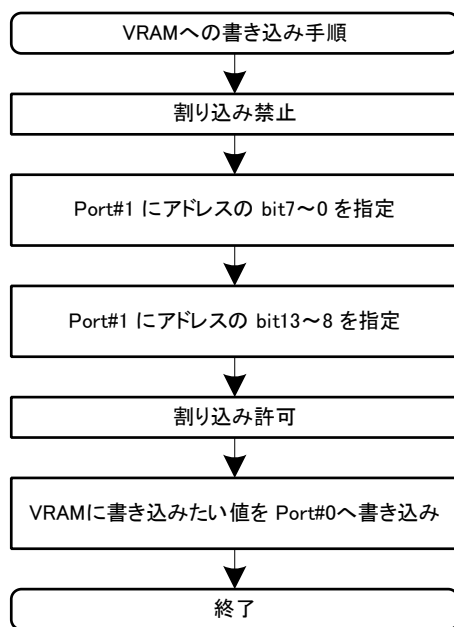


図 2.3.1.4-2. TMS9918 の VRAM 書き込みシーケンス

図 2.3.1.4-2.のプログラム例を下記に示します。

```
;-----  
;  
; WRTVRAM  
; HL ..... 書き込む VRAM アドレス  
; A ..... 書き込む値  
; 破壊レジスタ: AF, BC
```

MSX Documents VDP

```
;-----  
WRTVRM::  
    LD  B,A  
    LD  A,[0x0007]  
    INC A  
    LD  C,A  
    LD  A,H  
    OR  A,0x40  
    DI  
    ; Port#1 に A7～A0 を設定する  
    OUT [C],L  
    ; Port#1 に A13～A8 を設定する  
    OUT [C],A  
    EI  
    DEC C  
    OUT [C],B  
    RET
```

MSX バージョンアップアダプタを無視するのであれば、下記で OK です。

```
;-----  
; WRTVRAM  
; HL ..... 書き込む VRAM アドレス  
; A ..... 書き込む値  
; 破壊レジスタ: AF, BC  
;-----  
WRTVRM::  
    LD  B,A  
    LD  C,0x99  
    LD  A,H  
    OR  A,0x40  
    DI  
    ; Port#1 に A7～A0 を設定する  
    OUT [C],L  
    ; Port#1 に A13～A8 を設定する  
    OUT [C],A  
    EI  
    DEC C  
    OUT [C],B  
    RET
```

MSX の BIOS には、WRTVRM (004Dh) というルーチンが用意されていますので、特に処理速度などを要求しない場合は BIOS を使う方が安全でしょう。というのも、VDP の応答速度に対して、Z80 が微妙に速いために、あまり詰めた

MSX Documents VDP

アクセスをすると VDP 側が取りこぼすケースがあります。BIOS はその当たりも考慮されていますので取りこぼすことはありません。

【note】

MSXturboR の場合、システム IC(S1990)が I/O アクセスを監視していて R800 モード時に VDP にアクセスしに行くと、R800 に対して強制的に一定時間の WAIT を入れるので取りこぼさなかったりしますが、WAIT がやや過剰であることが知られています。そのような事情から「R800 で取りこぼさなかったから Z80 なら大丈夫だろう」というと、実はそうではなく Z80 で VDP アクセスした方が速かったりします。概ね VDP へのアクセスは、VDP 側は 21cycle@3.579MHz 程度、SCREEN0 では 22cycle@3.579MHz 程度かかるらしいです。S1990 は、この 22cycle を考慮した固定長 WAIT を入れるため、大半の 21cycle で良いケースでも 22cycle 待ってしまっていて遅いのではないか、という考察をしている方もいらっしゃいます。R800 の WAIT 入力端子をオシロスコープなどで観測すれば真偽のほどが分かるかもしれませんが、貴重な turboR に観測用線出しの半田付けをするのはリスクが大きいので未確認です。

次に、V9938/V9958 の場合、TMS9918 互換モード時には、上記 TMS9918 と同じシーケンスで問題ありません。その代わり、VRAM 前半の 16KB にしかアクセスできません。VRAM の 04000h~1FFFFh にアクセスするためには、図 2. 3.1.4-1.のシーケンスを採ることになります。そのプログラム例を下記に示します。

```
;-----  
; WRTVRAM  
; BHL .... 書き込む VRAM アドレス  
; A ..... 書き込む値  
; 破壊レジスタ: AF, BC, D  
;-----  
WRTVRM::  
    LD  D,A  
    LD  A,[0x0007]  
    INC A  
    LD  C,A  
    ; R#14 に A16~A14 を設定する  
    LD  A,H  
    AND A,0xC0  
    OR  A,B  
    RLCA  
    RLCA  
    DI  
    OUT [C],A  
    LD  A,14 | 0x80  
    OUT [C],A  
    LD  A,H  
    AND A,0x3F  
    OR  A,0x40  
    ; Port#1 に A7~A0 を設定する  
    OUT [C],L  
    ; Port#1 に A13~A8 を設定する  
    OUT [C],A  
    EI
```

MSX Documents VDP

```
DEC C
OUT [C],D
RET
```

MSX バージョンアップアダプタを無視するのであれば、下記で OK。

```
;-----
; WRTVRAM
; BHL .... 書き込む VRAM アドレス
; A ..... 書き込む値
; 破壊レジスタ: AF, BC, D
;-----
WRTVRM::
    LD D,A
    LD C,0x99
    ; R#14 に A16~A14 を設定する
    LD A,H
    AND A,0xC0
    OR A,B
    RLCA
    RLCA
    DI
    OUT [C],A
    LD A,14 | 0x80
    OUT [C],A
    LD A,H
    AND A,0x3F
    OR A,0x40
    ; Port#1 に A7~A0 を設定する
    OUT [C],L
    ; Port#1 に A13~A8 を設定する
    OUT [C],A
    EI
    DEC C
    OUT [C],D
    RET
```

Port#0 に対して連続的に書き込めば、連続アドレスの VRAM に書き込めるので、OUTI 命令や OTIR 命令を使うことで高速にまとめて転送することができます。

【note】

OUTI 命令は、16cycle 程度です。一方で VDP は概ね 1 回の VRAM アクセスに 21cycle を要します。そのため、高速動作を目的として OUTI を多数並べて処理すると、速すぎて VDP が取りこぼします。OTIR 命令もループ中は 21cycle なので問題ありませんが、256byte 以上転送しようとして OTIR, OTIR と並べて書くと 1 個目の最後の書き込みから 2 個目の最初の書き込

MSX Documents VDP

みまでが 16cycle 程度になって取りこぼす恐れがあります。厄介なのは、VDP は内部的に DRAM コントローラーを持っていて、そのコントローラーの状態に応じて反応できる速度が変化することです。実行タイミングによっては、今は動いたけど、次は取りこぼした、なんてことが起こり得ます。

昨今、優秀な MSX エミュレーターによって、高度なデバッグ付きの開発が可能になっていますが、この「VDP の取りこぼし」までエミュレートしているエミュレータは、2020 年 3 月現在、見たことがありません。エミュレーターでは、既存のソフトで問題が出ない程度の互換性があることを目標にしているものが多く、「実機で動くものを、エミュレータでも動くようにする」が基準になっています。「実機で動かないもの」がどうなるかまでエミュレートターゲットに含まれていないので、しばしば「実機で動かないものが動いてしまう」という問題に直面します。エミュレータだけで開発して、実機に持って行ったら動かなかった、なんてことがよく起こります。VDP の取りこぼしも、この「実機に持って行ったら動かなかった」という代表例なので、エミュレータ上で開発する方は注意しておくべきかと思います。

MSX Documents VDP

2.3.1.5. VRAM からの読み出し

2.3.1.6. 間接的なコントロールレジスタへの書き込み(インクリメントあり)

この機能は V9938 で追加になった機能です。TMS9918 では使えません。

コントロールレジスタ R#17 に、書き込みたいコントロールレジスタ番号を書き込むと、Port#3 がそのレジスタ番号のコントロールレジスタに接続されます。そして Port#3 に書き込むたびに、そのコントロールレジスタ番号がインクリメントされます。

つまり、連続するコントロールレジスタへ、まとめて書き込みたい場合にこの機能を使うことで、余計な I/O アクセスを削減することができます。よく使うのは、VDP コマンドのレジスタ群への設定ですので、VDP コマンドを使う場合は、このアクセス方法を覚えておくと役立ちます。

図 2.3.1.6-1.にこのアクセス方法の処理フローを示します。

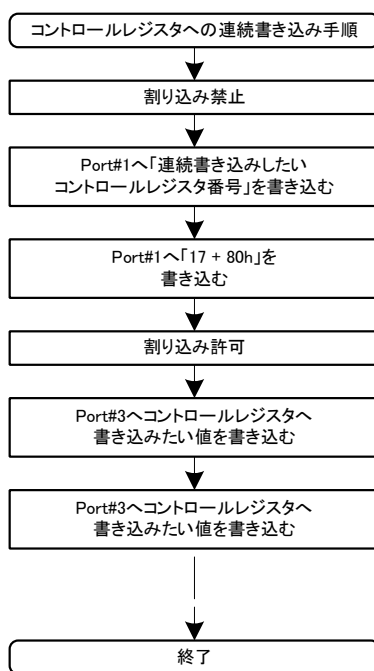


図 2.3.1.6-1. コントロールレジスタへの連続書き込み(オートインクリメント)処理手順

このフローを見ればわかるように、最初に R#17 へ書き込む手間は発生しますが、その後はレジスタ番号を設定する手間を省けるので、多数の連続レジスタへのアクセスでは、直接コントロールレジスタへ書き込む方法よりも実行サイクル数が少なく済みます。DRAM 上に書き込む値が並べてあれば、OTIR 命令でまとめて書き込むこともできます。

MSX Documents VDP

この機能を利用して、VDP コマンドを発行するルーチンの例を下記に示します。VDP コマンドの詳細については、2.6. VDP コマンドを参照ください。

```
IO_VDP_PORT1 = 0x99
```

```
IO_VDP_PORT3 = 0x9B
```

```
; -----  
; run_vdp_command  
; input)  
; hl ... VDP コントロールレジスタ R#32~R#45 に設定する値が格納されたアドレス  
; output)  
; なし  
; break)  
; af, bc, hl  
; -----  
run_vdp_command::  
; VDP R#17 = 32  
ld a, 32  
di  
out [IO_VDP_PORT1], a  
ld a, 0x80 | 17  
out [IO_VDP_PORT1], a  
; R#32~46 (15 registers)  
ld bc, (15 << 8) | IO_VDP_PORT3  
otir  
ei  
ret
```

この例では、OTIR は割込禁止中になっていますが、割り込み処理(H.TIMI や H.KEYI)の中で R#17 や Port#3 にアクセスするプログラムを配置していなければ、OTIR の前に EI しても問題ありません。

2.3.1.7. 間接的なコントロールレジスタへの書き込み(インクリメントなし)

この機能は V9938 で追加になった機能です。TMS9918 では使えません。

間接的なコントロールレジスタへの書き込み(インクリメントあり)と、処理フローは同じです。ただし、R#17 にレジスタ番号を設定するときに、MSB(bit7) を 1 にするとインクリメントなしになります。図 2.3.1.7-1. にフローを示します。

同じコントロールレジスタに対して、連続的にアクセスする場合にしか使えないため、VDP コマンドの CPU→VRAM 転送(HMMC, LMMC)用に作られた機能だと思います。もちろん、それ以外のレジスタにも使用することはできます。

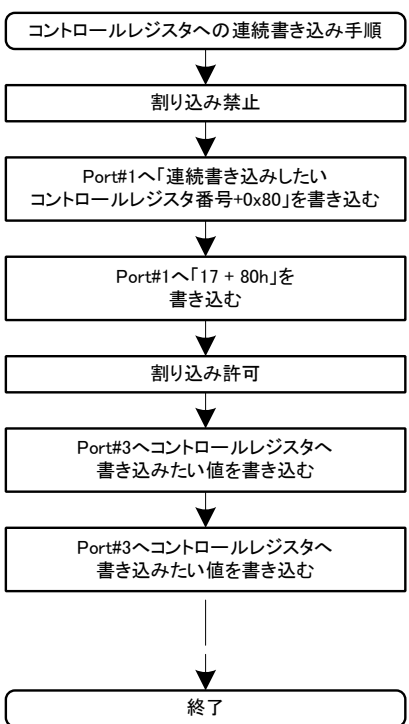


図 2.3.1.7-1. コントロールレジスタへの連続書き込み(非オートインクリメント)処理手順

この機能は、同一のコントロールレジスタへの連続書き込みをサポートするだけの機能になるため、その他の待ち処理は含まれていません。例えば、HMMC/LMMC のように VDP ステータスレジスタ S#2 の TR ビットが 1 になるのを待ってから次の 1byte を書き込む必要のある機能では、OTIR ではなく OUT 命令 や OUTI 命令で 1byte ずつ送り、ステータスレジスタのチェックを間に挟む必要があることに注意してください。

2.3.2. MSX-BASIC からのアクセス

当然のことながら、MSX-BASIC には割り込み禁止・割り込み許可をする命令は存在しません。そのため、OUT 命令・INP 関数を使って各種 VDP ポートへのアクセスは厳禁です。その代わりに、安全に VDP へアクセスできるように、システム変数の VDP()・BASE() が用意されています。

VDP() は、MSX-BASIC から VDP のレジスタを読み書きする際に利用するシステム変数です。VDP のレジスタと、VDP() の対応関係を表 2.3.2.1. にまとめます。

【note1】

R#8～R#46 は、V9938/V9958 で追加になったレジスタですが、MSX-BASIC 上では VDP(9)～VDP(47) に対応しています。MSX1 は TMS9918 なので、R#0～R#7 までしか無いので VDP(0)～VDP(7) に対応させたわけですが、S#0 の読み出しを VDP(8) に割り当てていました。MSX2 を出す時点で V9938 に変わったわけですが、VDP(8) を R#8 にしてしまうと、VDP(8) を読み出して使ってる MSX-BASIC プログラムが動かなくなってしまう互換性上の問題があります。そのため、R#8 は VDP(9) に、以降 1 つずつズレた番号に対応させる対応がなされました。

S#1～S#9 も V9938/V9958 から追加になったレジスタですが、これらは VDP(-1)～VDP(-9) に対応させることで、今後拡張されたとしても R#n の対応関係がズレないように配慮されたわけです。そのため、V9958 から追加になった R#25, 26, 27 は、VDP(26), VDP(27), VDP(28) に対応しており、「1 つずれているだけ」を維持しています。

【note2】

ステータスレジスタ S#n に対応する VDP() は、ワークエリアでは無く、実際に読み出した値が返ってきます。当然ながら書き込みはできないので VDP(8)=0 のような記述はエラーになります。

【note3】

VDP コマンド用のコントロールレジスタは、対応するワークエリアがありません。VDP コマンド用コントロールレジスタは、書き込むこと自体が実行のアクションに繋がっており、VDP 内部で値を維持しているわけでは無く、VDP コマンド実行とともに変化してしまうレジスタであるためワークエリアにバックアップを保存しても意味が無いからだと思います。そのため、VDP(33)～VDP(47) は書き込み専用で、PRINT VDP(33) のような読み出しアクセスはエラーになります。

表 2.3.2.1. VDPレジスタとVDP()とワークエリアの対応関係

Control Register	VDP(n)	ワークエリア名	ワークエリアアドレス
R#0	VDP(0)	REG0SAV	0xF3DF
R#1	VDP(1)	REG1SAV	0xF3E0
R#2	VDP(2)	REG2SAV	0xF3E1
R#3	VDP(3)	REG3SAV	0xF3E2
R#4	VDP(4)	REG4SAV	0xF3E3
R#5	VDP(5)	REG5SAV	0xF3E4
R#6	VDP(6)	REG6SAV	0xF3E5
R#7	VDP(7)	REG7SAV	0xF3E6
R#8	VDP(9)	REG8SAV	0xFFE7
R#9	VDP(10)	REG9SAV	0xFFE8
R#10	VDP(11)	REG10SAV	0xFFE9
R#11	VDP(12)	REG11SAV	0xFFEA
R#12	VDP(13)	REG12SAV	0xFFEB
R#13	VDP(14)	REG13SAV	0xFFEC
R#14	VDP(15)	REG14SAV	0xFFED
R#15	VDP(16)	REG15SAV	0xFFEE
R#16	VDP(17)	REG16SAV	0xFFEF
R#17	VDP(18)	REG17SAV	0xFFFF0
R#18	VDP(19)	REG18SAV	0xFFFF1
R#19	VDP(20)	REG19SAV	0xFFFF2
R#20	VDP(21)	REG20SAV	0xFFFF3
R#21	VDP(22)	REG21SAV	0xFFFF4
R#22	VDP(23)	REG22SAV	0xFFFF5
R#23	VDP(24)	REG23SAV	0xFFFF6
R#25	VDP(26)	REG25SAV	0xFFFF8
R#26	VDP(27)	REG26SAV	0xFFFF9
R#27	VDP(28)	REG27SAV	0xFFFFA
R#32	VDP(33)	無し	無し
R#33	VDP(34)	無し	無し
R#34	VDP(35)	無し	無し
R#35	VDP(36)	無し	無し
R#36	VDP(37)	無し	無し
R#37	VDP(38)	無し	無し
R#38	VDP(39)	無し	無し
R#39	VDP(40)	無し	無し
R#40	VDP(41)	無し	無し
R#41	VDP(42)	無し	無し
R#42	VDP(43)	無し	無し
R#43	VDP(44)	無し	無し
R#44	VDP(45)	無し	無し
R#45	VDP(46)	無し	無し
R#46	VDP(47)	無し	無し
S#0	VDP(8)	無し	無し
S#1	VDP(-1)	無し	無し
S#2	VDP(-2)	無し	無し
S#3	VDP(-3)	無し	無し
S#4	VDP(-4)	無し	無し
S#5	VDP(-5)	無し	無し
S#6	VDP(-6)	無し	無し
S#7	VDP(-7)	無し	無し
S#8	VDP(-8)	無し	無し
S#9	VDP(-9)	無し	無し

MSX Documents VDP

先にしたように、VDP のコントロールレジスタは読み出せません。MSX-BASIC では 一部を除くコントロールレジスタに対応するワークエリアを用意しており、読み出しはこのワークエリアに対して行われます。BIOS を介した VDP コントロールレジスタへの書き込みの際には、対応するワークエリアへの書き込みも行われています。

2.3.1. で I/O ポート直接アクセスによるレジスタ更新方法を示しましたが、これによるレジスタ書き込みは当然ながらワークエリアの更新は行われません。MSX-BASIC の VDP()読み出しと混在するような機械語プログラムでは、BIOS を介してアクセスするか、対応するワークエリアの更新も行う方が安全です。

コントロールレジスタによってアドレスを変更可能な各種 VRAM 要素ですが、MSX-BASIC では BASE()システム変数によってアドレスを管理しています。複数のコントロールレジスタに分断されているアドレス指定もあるため、VDP()で値を取得して計算すると処理時間がかかってしまいますが、アドレスを知りたいのであれば BASE()を使えば計算の手間を省けます。表 2.3.2.2.に BASE()とテーブルアドレスの対応関係を示します。

表 2.3.2.2. BASE()システム変数と VRAM 要素の対応関係

BASE(n)のn	初期値	対応するSCREENモード	意味
0	00000H	0	Pattern Name Table Address
1	00000H	0	Blink Table Address (Width80 only)
2	00800H	0	Pattern Generator Table Address
3	00000H	0	N/A
4	00000H	0	N/A
5	01800H	1	Pattern Name Table Address
6	02000H	1	Color Table Address
7	00000H	1	Pattern Generator Table Address
8	01B00H	1	Sprite Attribute Table Address
9	03800H	1	Sprite Generator Table Address
10	01800H	2	Pattern Name Table Address
11	02000H	2	Color Table Address
12	00000H	2	Pattern Generator Table Address
13	01B00H	2	Sprite Attribute Table Address
14	03800H	2	Sprite Generator Table Address
15	00800H	3	Pattern Name Table Address
16	00000H	3	N/A
17	00000H	3	Pattern Generator Table Address
18	01B00H	3	Sprite Attribute Table Address
19	03800H	3	Sprite Generator Table Address
20	01800H	4	Pattern Name Table Address
21	02000H	4	Color Table Address
22	00000H	4	Pattern Generator Table Address
23	01E00H	4	Sprite Attribute Table Address
24	03800H	4	Sprite Generator Table Address
25	00000H	5	Pattern Name Table Address
26	00000H	5	N/A
27	00000H	5	N/A
28	07600H	5	Sprite Attribute Table Address
29	07800H	5	Sprite Generator Table Address
30	00000H	6	Pattern Name Table Address
31	00000H	6	N/A
32	00000H	6	N/A
33	07600H	6	Sprite Attribute Table Address
34	07800H	6	Sprite Generator Table Address
35	00000H	7	Pattern Name Table Address
36	00000H	7	N/A
37	00000H	7	N/A
38	0FA00H	7	Sprite Attribute Table Address
39	0F000H	7	Sprite Generator Table Address
40	00000H	8	Pattern Name Table Address
41	00000H	8	N/A
42	00000H	8	N/A
43	0FA00H	8	Sprite Attribute Table Address
44	0F000H	8	Sprite Generator Table Address
50	00000H	10	Pattern Name Table Address
51	00000H	10	N/A
52	00000H	10	N/A
53	0FA00H	10	Sprite Attribute Table Address
54	0F000H	10	Sprite Generator Table Address
55	00000H	11	Pattern Name Table Address
56	00000H	11	N/A
57	00000H	11	N/A
58	0FA00H	11	Sprite Attribute Table Address
59	0F000H	11	Sprite Generator Table Address
60	00000H	12	Pattern Name Table Address
61	00000H	12	N/A
62	00000H	12	N/A
63	0FA00H	12	Sprite Attribute Table Address
64	0F000H	12	Sprite Generator Table Address

MSX Documents VDP

各 SCREEN モードによって BASE(n)の n が分けられています。SCREEN モード番号を n とすると、

BASE(n*5) ~ BASE(n*5+4) がその画面モードの VRAM 要素のアドレスを示しています。

BASE(n*5) は Pattern Name Table のアドレス、BASE(n*5+1)は Color Table のアドレス、BASE(n*5+2)は Pattern Generator Table のアドレス、BASE(n*5+3)は Sprite Attribute Table, BASE(n*5+4)は Sprite Generator Table のアドレスになっています。

コントロールレジスタでアドレスを変更できると言っても、1byte 単位でアドレスを変更できる領域は存在しません。BASE(x)=y とすることで、アドレスを変更してくれるのですが、設定不可能な値を指定すると Illegal Function Call のエラーが発生するので注意してください。

変更できることは SCREEN1 で BASE(5)=&H2000 などと実行してみれば分かります。

設定不可能な値を指定するとエラーになることは SCREEN1 で BASE(5)=&H1801 などと実行してみれば分かります。

2.4. VDP のレジスタ

VDP のレジスタに対するアクセス方法に関しては、2.3.にて説明しました。ここでは、アクセスすることによってどのような作用があるのか説明します。

レジスタのビットマップ表記（各ビットが何に割り当てられているかを示す表記）では、アドレス表現の値を A と記載します。例えば、アドレス値の bit16 に対応するビットは A16 と記載します。また、白=TMS9918/V9938/V9958, 赤=V9938/V9958, 黄=V9958, 青=V9938, 灰=無効として色分け表示します。

2.4.1. コントロールレジスタ

この章では、VDP コントロールレジスタについて説明します。どのコントロールレジスタも 8bit の幅を持っていて、書き込み専用です。

2.4.1.1. Mode Register 0 (R#0)

画面モードを指定するレジスタです。各ビットの意味を図 2.4.1.1-1. に示します。

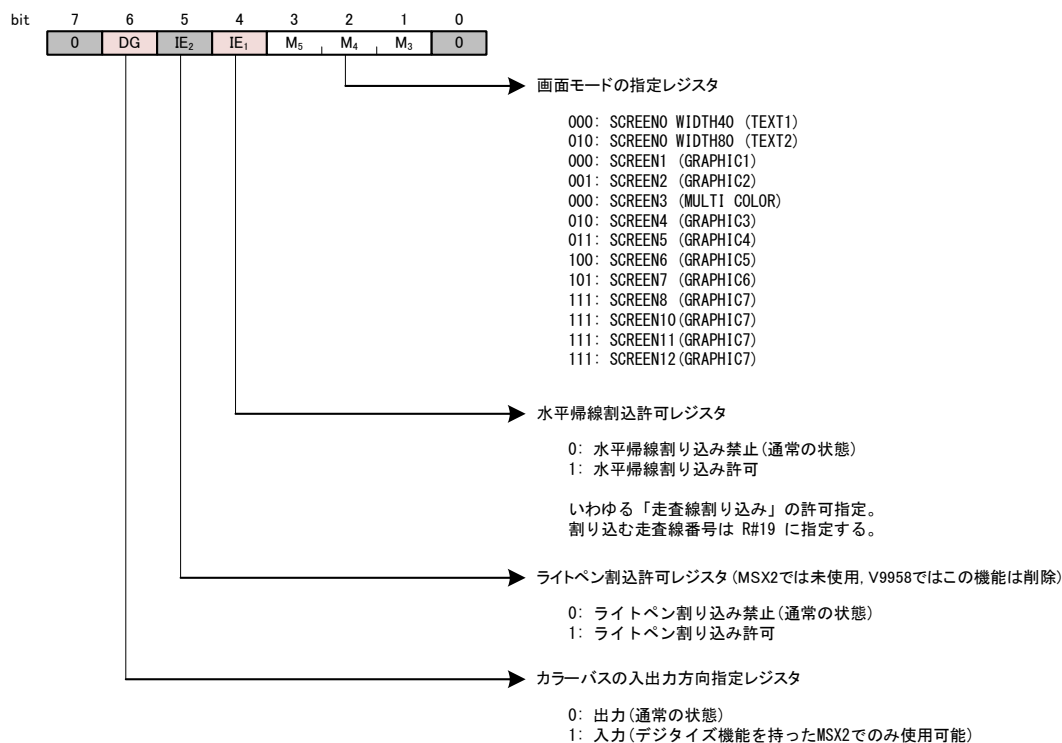


図 2.4.1.1-1. R#0 のビットマップ

bit0 は無効です。

bit3～bit1 は、画面モード指定が割り当てられています。R#1 にある同様のビットとの組み合わせで意味が決まるため、2.4.1.2. Mode Register 1 (R#1) でまとめて説明します。

bit4 は、TMS9918 では無効です。V9938/V9958 では水平帰線割込の許可ビットになっています。デフォルトでは 0 (無効) となっており、1 にすると有効になります。水平帰線割込については 2.4.1.20. Interrupt Line Register (R#19) でまとめて説明します。

bit5 は、TMS9918 及び V9958 では無効です。V9938 ではライトペン向けにライトペン割込の有効・無効の設定が追加されているのですが、MSX2 規格ではこの機能は利用されていないようです。V9958 ではこの機能は削除されています。

MSX Documents VDP

bit6は、TMS9918では無効です。デジタイザ搭載 MSX2 で、カラーバスの入力方向を指定します。通常は 0 で、出力方向。1 にすると入力方向になります。

bit7は無効です。

無効な bit には、0 を指定してください。

2.4.1.2. Mode Register 1 (R#1)

画面モードを指定するレジスタです。各ビットの意味を図 2.4.1.2-1.に示します。

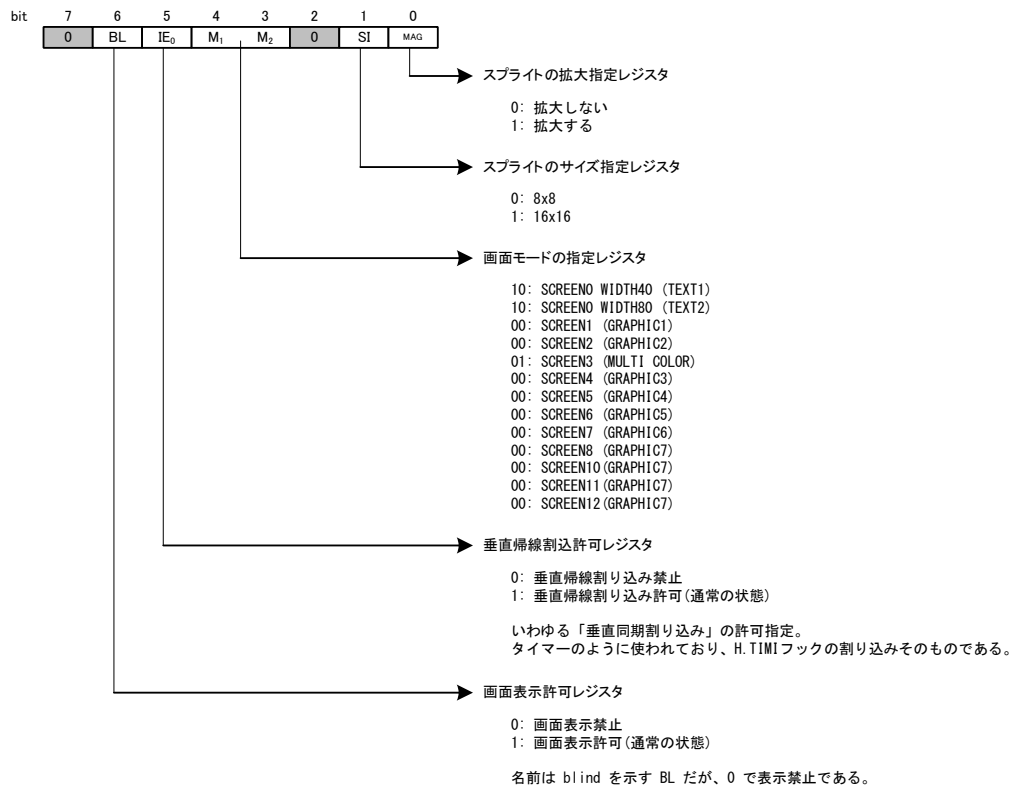


図 2.4.1.2-1. R#1 のビットマップ

bit1, bit0 の 2bit は、MSX-BASIC の SCREEN 命令の第2引数と一致します。表 2.4.1.2-1.に設定値をまとめます。スプライトのサイズ指定になりますが、スプライトモード1とモード2ともに同じ動作になります。

表 2.4.1.2-1. R#1 bit1,bit0 の設定値

設定値		意味
bit1	bit0	
0	0	8ドット×8ドットサイズ。拡大無し。
0	1	8ドット×8ドットサイズ。拡大あり。
1	0	16ドット×16ドットサイズ。拡大無し。
1	1	16ドット×16ドットサイズ。拡大あり。

MSX Documents VDP

これを使ってスプライトのサイズを、動的に変更させるサンプル(R1SPRITE.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,4,7:SCREEN1:WT=2000
110 FOR I=0 TO 30 STEP 2:VPOKE&H3800+I,&HAA:VPOKE&H3801+I,&H55:NEXT
120 V1=VDP(1)AND&HFC:PUTSPRITE0,(0,0),15,0
130 '
140 VDP(1)=V1+0:GOSUB180
150 VDP(1)=V1+1:GOSUB180
160 VDP(1)=V1+2:GOSUB180
170 VDP(1)=V1+3:GOSUB180:GOTO140
180 FOR I=0 TO WT:NEXT:RETURN
```

このサンプルは、Z80での動作を想定しています。turboRの場合は1キーを押しながら立ち上げるなどしてZ80モードにして頂くか、あるいは180行目のFOR~NEXTを_PAUSE(1000)などに置き換えてください。

このサンプル(R1SPRITE.BAS)を実行した結果を写真2.4.1.2-1.に示します。



写真 2.4.1.2-1. R1SPRITE.BAS 実行結果

bit2は何も機能が割り当てられていません。

bit4, bit3にはモード設定が割り当てられています。これについては、R#0とも関連があります。R#0のbit3~bit1の3bitと、R#1のbit4, bit3の2bitの合計5bitでモード指定をします。画面の表示方法を決定するのみで、VRAM上の各種要素(Pattern Name Table 等)のアドレスは、別途レジスタに設定する必要があります。設定値の一覧を表2.4.1.2-2.にまとめます。

表 2.4.1.2-2. R#0/R#1 M5～M1 の設定値

R#0			R#1		意味	互換性		
M5	M4	M3	M2	M1		TMS9918	V9938	V9958
0	0	0	0	0	SCREEN1(GRAPHIC1)	○	○	○
0	0	0	0	1	SCREEN3(MULTI COLOR)	○	○	○
0	0	0	1	0	SCREEN0:WIDTH40(TEXT1)	○	○	○
0	0	0	1	1	未定義	-	-	-
0	0	1	0	0	SCREEN2(GRAPHIC2)	○	○	○
0	0	1	0	1	未定義	-	-	-
0	0	1	1	0	未定義	-	-	-
0	0	1	1	1	未定義	-	-	-
0	1	0	0	0	SCREEN4(GRAPHIC3)	×	○	○
0	1	0	0	1	未定義	-	-	-
0	1	0	1	0	SCREEN0:WIDTH80(TEXT2)	×	○	○
0	1	0	1	1	未定義	-	-	-
0	1	1	0	0	SCREEN5(GRAPHIC4)	×	○	○
0	1	1	0	1	未定義	-	-	-
0	1	1	1	0	未定義	-	-	-
0	1	1	1	1	未定義	-	-	-
1	0	0	0	0	SCREEN6(GRAPHIC5)	×	○	○
1	0	0	0	1	未定義	-	-	-
1	0	0	1	0	未定義	-	-	-
1	0	0	1	1	未定義	-	-	-
1	0	1	0	0	SCREEN7(GRAPHIC6)	×	○	○
1	0	1	0	1	未定義	-	-	-
1	0	1	1	0	未定義	-	-	-
1	0	1	1	1	未定義	-	-	-
1	1	0	0	0	未定義	-	-	-
1	1	0	0	1	未定義	-	-	-
1	1	0	1	0	未定義	-	-	-
1	1	0	1	1	未定義	-	-	-
1	1	1	0	0	SCREEN8(GRAPHIC7)	×	○	○
1	1	1	0	1	未定義	-	-	-
1	1	1	1	0	未定義	-	-	-
1	1	1	1	1	未定義	-	-	-

モードの種類に対して 5bit もあり、かなり冗長ですが、V9958 ではこの未定義は活用されておらず、SCREEN10～12 は SCREEN8 と同じ扱いになっています。

bit5 には、垂直帰線割込の ON/OFF 指定が割り当てられています。VDP が 1 画面分をモニター出力から出力して、次の画面を出力し始めるまでの待ち時間を垂直ブランキング期間と呼びますが、その垂直ブランキング期間の始まりのタイミングで発生する割込です。一般的な MSX では、割込と言えば VDP からしか受けていないことがほとんどで、その中でも必ず使っている VDP 割込がこの割込になります。MSX には標準ではタイマー回路が搭載されていませんが、タイマーの代わりを成す仕組みとして利用されています。

ただし、出力先のモニターの規格に合わせるため、NTSC モニター接続の場合は 59.94Hz、PAL モニター接続の場合は 50.0Hz の周期で発生します。

TMS9918 は、IC の型番によって NTSC 出力用・PAL 出力用、と接続先モニターが固定になっていて切り替えられません。TMS9918 は、接続する DRAM の種類でも型番が異なっていたり、互換品があったりしますが、本書ではその点については省略させていただきます。

V9938/V9958 では、R#9 で NTSC/PAL を切り替えることができます。切替については R#9 の項目を参照ください。

MSX では、タイマーの代わりにこの垂直帰線割込を利用していることもあって、むやみに止めてしまうと不具合が発生する場合があるのでご注意ください。この割込を使ってモーターを停止させるタイミングをはかる FDD があるとの情報を見たことがありますので、通常はこのビットは 1 のまま利用するのが望ましいです。

bit6 の BL は、通常 1 になっています。0 にすると画面が非表示になります。非表示というのは、VDP がモニターへの出力を止めてしまうわけではなく、周辺色で塗りつぶされた画面表示に固定したモニター出力になります。特に V9938/V9958 では、VRAM へのアクセスを管理している DRAM コントローラーが、タイミング必須のアクセス(背景表示

MSX Documents VDP

やスプライト表示)の隙間で VDP コマンドのための VRAM アクセスを行います。BL=0 にして背景・スプライトを非表示にすることで、「タイミング必須のアクセス」が存在しなくなり、VDP コマンドによるアクセスの優先度が上がります。その影響で、VDP コマンドの動作が速くなります。画面を真っ黒表示にしても良いようなケースで、裏画面に何らかの描画処理を行っているのであれば、このビットを 0 にすることでその待ち時間を短縮できる可能性があります。MSX-BASIC の SCREEN5 以降の LINE/COPY 等も VDP コマンドで動作していますので、それらの高速化にも効果があります。

2.4.1.3. Pattern Name Table Address (R#2)

Pattern Name Table の開始アドレスを指定するレジスタです。各ビットの意味を図 2.4.1.3-1.に示します。

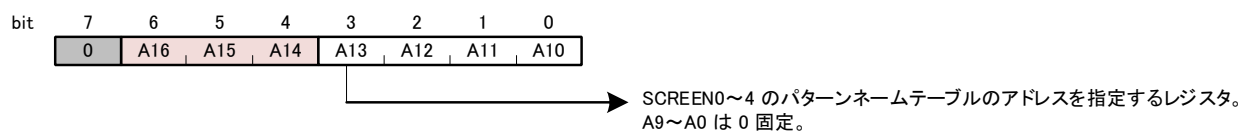


図 2.4.1.3-1. R#2 のビットマップ

bit6～0 で、Pattern Name Table のアドレスの bit16～bit10 を指定します。ただし、TMS9918 では、A16～A14 は無効なため、bit6～4 には 0 を指定します。

SCREEN0～4 では、A16～A10 に任意の値を指定できます。TMS9918 では 0000h, 0400h, 0800h, 0C00h, 1000h, 1400h, 1800h, 1C00h, 2000h, 2400h, 2800h, 2C00h, 3000h, 3400h, 3800h, 3C00h の 16 通りのアドレスのいずれかを指定できます。V9938 で VRAM64KB の機種では A16 は 0 を指定しなければなりません。VRAM128KB の機種では、多いので列挙しませんが 2 の 7 乗の組み合わせで 128 通りのアドレス指定が可能です。

SCREEN5～6 では、A16～A15 で MSX-BASIC の SET PAGE の表示ページにあたる 0～3 を指定します。ただし、bit 4～0 はすべて 1 にしなければ期待通りに動作しません。また、bit4～0 をすべて 1 に指定していますが、A14～A10 は、すべて 0 と見なされます。bit4～0 に all 1 以外の指定をすると未定義の動作になります。未定義の動作については 3. 解析で確認してみます。

SCREEN5～6 では、表示ページ 0 なら 1Fh, 表示ページ 1 なら 3Fh, 表示ページ 2 なら 5Fh, 表示ページ 3 なら 7Fh を R#2 に書き込みます。

SCREEN7～8, SCREEN10～12 では A16～A14 で MSX-BASIC の SET PAGE の表示ページにあたる 0～1 を指定します。ただし、bit5～0 はすべて 1 にしなければ期待通りに動作しません。また、bit5～0 をすべて 1 に指定していますが、A15～A10 は、すべて 0 と見なされます。bit5～0 に all 1 以外の指定をすると未定義の動作になります。

SCREEN7～8, SCREEN10～12 では、表示ページ 0 なら 3Fh, 表示ページ 1 なら 7Fh を R#2 に書き込みます。

では、MSX-BASIC の SET PAGE 命令が、SCREEN5 と 6 とで表示ページ指定に対して R#2 にどのような値を設定しているか確認するサンプル(R2SC56.BAS)を下記に示します。

```
100 DEFINT A-Z: DIM R2(1,3)
```

MSX Documents VDP

```
110 SCREEN5
120 SETPAGE0:R2(0,0)=VDP(2)
130 SETPAGE1:R2(0,1)=VDP(2)
140 SETPAGE2:R2(0,2)=VDP(2)
150 SETPAGE3:R2(0,3)=VDP(2)
160 SCREEN6
170 SETPAGE0:R2(1,0)=VDP(2)
180 SETPAGE1:R2(1,1)=VDP(2)
190 SETPAGE2:R2(1,2)=VDP(2)
200 SETPAGE3:R2(1,3)=VDP(2)
210 SCREEN1:WIDTH32
220 FORS=0TO1:FORP=0TO3
230 PRINT "SCREEN";(S+5);" PAGE";P;"=&H";HEX$(R2(S,P))
240 NEXTP,S
```

この実行結果を、写真 2.4.1.3-1.に示します。

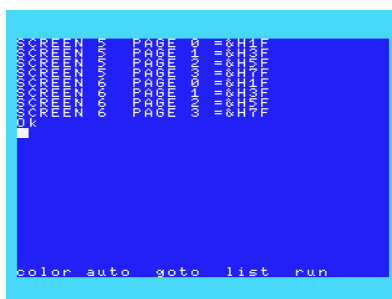


写真 2.4.1.3-1. R2SC56.BAS の実行結果

MSX Documents VDP

同様に SCREEN7, 8 で確認するサンプル(R2SC78.BAS)を下記に示します。

```
100 DEFINT A-Z: DIM R2(1,1)
110 SCREEN7
120 SETPAGE0: R2(0,0)=VDP(2)
130 SETPAGE1: R2(0,1)=VDP(2)
160 SCREEN8
170 SETPAGE0: R2(1,0)=VDP(2)
180 SETPAGE1: R2(1,1)=VDP(2)
210 SCREEN1: WIDTH32
220 FOR S=0 TO 1: FOR P=0 TO 1
230 PRINT "SCREEN"; (S+7); " PAGE"; P; "=" & H"; HEX$(R2(S,P))
240 NEXT P, S
```

この実行結果を、写真 2.4.1.3-2. に示します。



写真 2.4.1.3-2. R2SC78.BAS の実行結果

MSX Documents VDP

同様に SCREEN10～12 について確認するサンプル(R2SCAC.BAS)を下記に示します。

```
100 DEFINT A-Z: DIM R2(2,1)
110 SCREEN10
120 SETPAGE0: R2(0,0)=VDP(2)
130 SETPAGE1: R2(0,1)=VDP(2)
140 SCREEN11
150 SETPAGE0: R2(1,0)=VDP(2)
160 SETPAGE1: R2(1,1)=VDP(2)
170 SCREEN12
180 SETPAGE0: R2(2,0)=VDP(2)
190 SETPAGE1: R2(2,1)=VDP(2)
200 SCREEN1: WIDTH32
210 FOR S=0 TO 2: FOR P=0 TO 1
220 PRINT "SCREEN"; (S+10); " PAGE"; P; "=" & H"; HEX$(R2(S,P))
230 NEXT P, S
```

この実行結果を、写真 2.4.1.3-3. に示します。

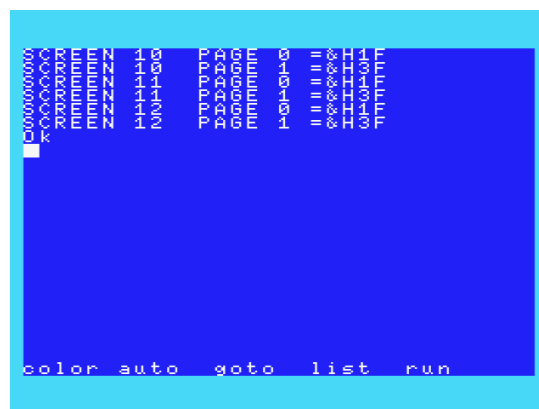


写真 2.4.1.3-3. R2SCAC.BAS の実行結果

2.4.1.4. Color Table Base Address Register Low (R#3)

SCREEN0(WIDTH80)の Blink Table 及び SCREEN1、2、4 の Color Table の開始アドレスを指定するレジスタです。

V9938/V9958 では、さらに R#10 で、より上位のビットも指定できます。TMS9918 には R#10 は存在せず、R#3 のみで Color Table のアドレスを指定します。R#10 については 2.4.1.11. Color Table Base Address Register High (R#10)を参照ください。

各ビットの意味を図 2.4.1.4-1., 図 2.4.1.4-2. 及び図 2.4.1.4-3. に示します。

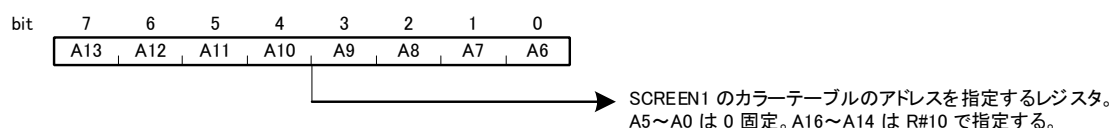


図 2.4.1.4-1. R#3 のビットマップ(SCREEN1: Color Table)

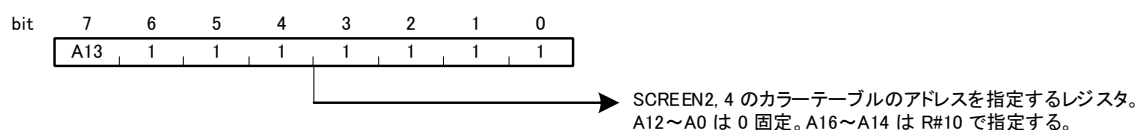


図 2.4.1.4-2. R#3 のビットマップ(SCREEN2,4: Color Table)

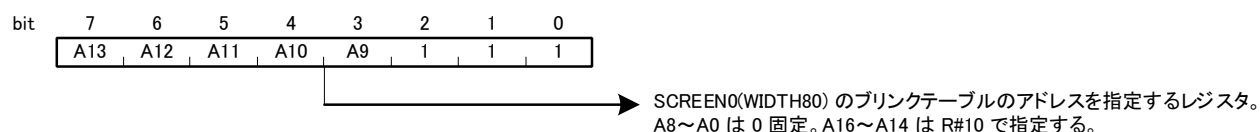


図 2.4.1.4-3. R#3 のビットマップ(Blink Table)

SCREEN1 の Color Table のアドレスは、下位は A6 まで指定できるので、64 単位で指定できます。0000h, 0040h, 0080h, 00C0h, 0100h, ... , 3F80h, 3FC0h と、かなり柔軟に選べます。

SCREEN1 では、1byte で 8 パーツの色を指定するので、256 種類のパーツの色つけに 32byte のサイズになっています。

SCREEN2 及び 4 の Color Table のアドレスは、下位は A13 までしか指定できません。8192 単位ですね。TMS9918 では 0000h か 2000h の 2 択です。V9938/V9958 では R#10 にさらに上位のアドレスがあるので、もう少し選択の幅があります。

SCREEN2 及び 4 では、8byte で 1 パーツの色を指定するので、256 種類のパーツの色つけに 2048byte のサイズ、これが上中下にそれぞれ存在するので 3 倍で 6144byte になっています。

MSX Documents VDP

Blink Table のアドレスは、下位は A9 までしか指定できません。Color Table と違って A8～A6 は 0 固定です。指定可能なアドレスは 512 単位ですね。

SCREEN0(WIDTH80)は、1bit で画面上の 1 文字の点滅有無を指定するので、 $80 \times 24 / 8 = 240$ byte のサイズがあります。

【note】

TMS9918 が登場した当時、乗算器はおろか加算器でさえ「大きくて高価な回路」だったと思います。R#3 の下位に &B0000000 を付与して、そこに現在の表示位置に必要な相対アドレス値を加算すれば、どのモードでも R#3 の全ビットが有効になるわけですが、コスト増を避けるために大きくて高価な回路を入れたくない。そこで、&B0000000 の代わりに &B1111111 を下位に付与して、計算した相対アドレス値の上位にも 1 の bit を必要数付与して、and を採っているものと思われます。and ゲートであれば 1bit あたりトランジスタ 2 個程度。加算器を使う場合に比べて 1/3 以下の回路規模で実現できます。TMS9918 は VRAM 4KB にも対応していますが、設計当初は高価な DRAM の量を柔軟に選択できるようにして、4KB 搭載機と 16KB 搭載機とでレジスタ設定値を変えるだけで対応できることを目的としてアドレス指定がレジスタになっているのだと思います。ゲームなどを作る上では、領域を 2 力所以上用意して、表示用と書き換え用のダブルバンクを垂直帰線割込に同期してバタバタ切り替えることで、書き換え途中の状態を見えないようにするといった使い方が出来るので、柔軟に設定できると格段に使い勝手が良くなるのですが、おそらく当時は高価な DRAM をダブルバンクで使うなんてことは、贅沢すぎてあまり考慮に入っていなかったのでは無いかと想像できます。しかも柔軟に設定できるようにするためにはアドレスの加算器が必要。今でこそこの程度の加算器であれば柔軟性確保の方が遙かに有意ですが、当時の設計としては非常に妥当かつ合理的かと思います。実際に and を使っているかどうかは、3. 解析のところで確認したいと思います。

2.4.1.5. Pattern Generator Table Base Address Register (R#4)

Pattern Generator Table の開始アドレスを指定するレジスタです。SCREEN0～4 で有効です。

TMS9918 では、A13 までしか存在しませんので、A16～A14 は 0 を指定します。

各ビットの意味を 図 2.4.1.5-1. に示します。

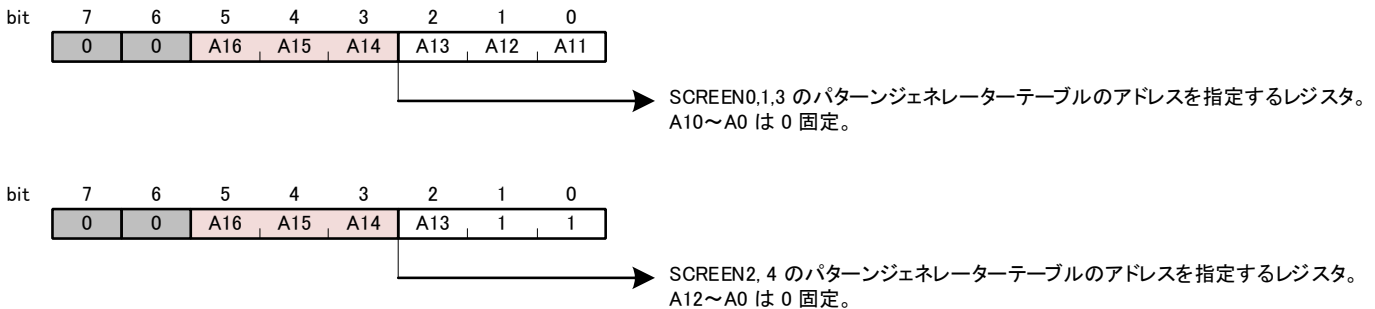


図 2.4.1.5-1. R#4 のビットマップ

SCREEN0(WIDTH40), SCREEN0(WIDTH80), SCREEN1, SCREEN3 では、Pattern Generator Table のアドレスのうち、bit16～bit11 を指定できます。これを R#4 の bit5～bit0 に書き込むことによりアドレスを指定のアドレスに変更できます。アドレスの bit10～bit0 は 0 の扱いになります。

SCREEN2, SCREEN4 では、Pattern Generator Table のアドレスのうち bit16～bit13 を指定できます。これを R#4 の bit5～bit2 に、R#4 の bit1 と bit0 は 1 にすることによりアドレスを指定のアドレスに変更できます。アドレスの bit12～bit0 は 0 の扱いになります。

各画面モードにおける Pattern Generator Table のサイズを表 2.4.1.5-1.にまとめます。

表 2.4.1.5-1. Pattern Generator Table のサイズ一覧

SCREENモード	Pattern Generator Table のサイズ(byte)
0(WIDTH40)	2048
0(WIDTH80)	2048
1	2048
2	6144
3	2048
4	6144

SCREEN5～12 では、Pattern Generator Table は存在しないため、R#4 の設定値は無視されます。

MSX Documents VDP

Pattern Generator Table は、表示するパーツの形状を定義するテーブルです。VDP は画面表示の際に、現在出力モニターに出力する位置に対応する Pattern Name Table の内容を読み出して表示するパーツ番号を得ます。次に表示するパーツ番号とY座標の値から、表示するパーツの形状を Pattern Generator Table から読み出します。

このような読み出し処理を、水平 8ドット (SCREEN8(WIDTH80) では 6ドット) に1度行い、連続する 8ドット (同 6ドット) の組は、読み出した結果により表示色が決定されます。

そのため、Pattern Generator Table のアドレスを変更すると、瞬時に画面上のパーツ形状が切り替わります。

VRAM 上に複数の Pattern Generator Table を用意しておき、1つを表示用、もう一つを書き替え用にして、垂直帰線割込に同期してその役割を入れ替えてやることにより、画面全体のパーツの表示形状を一瞬で変化させ、アニメーションさせることができます。

V9938/V9958 搭載機では、VRAM は 64KB ないし 128KB 搭載していますので、2面といわずたくさん用意することで、もっと複雑なアニメーションも可能になります。

アドレスの切替だけによるアニメーションであれば、R#4 を書き替えるだけなので MSX-BASIC からでも高速に処理できます。

工夫次第で面白い表現が出来るかもしれませんね。

2.4.1.6. Sprite Attribute Table Base Address Register Low (R#5)

Sprite Attribute Table の開始アドレスを指定するレジスタです。スプライトモード2では Sprite Color Table の終了アドレスも兼ねていることにご注意ください。

TMS9918 では、A13～A0しかありませんので、A14は0を指定してください。

V9938/V9958 では、残りの A16～A15 を R#11 で指定します。

各ビットの意味を図 2.4.1.6-1. に示します。

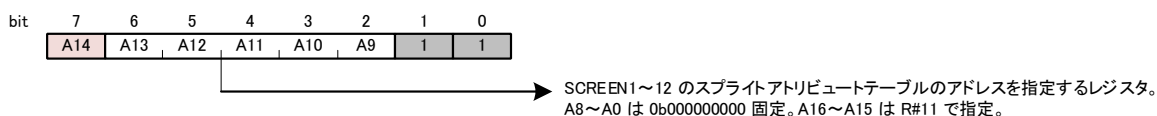


図 2.4.1.6-1. R#5 のビットマップ

例えば、SCREEN4 で Sprite Attribute Table の先頭アドレスを 01E00h にしたいとします。これを 2 進数で表現すると、

0 0001 1110 0000 0000

となります。この A14～A9 (bit14～bit9) を青にすると、

00001 1110 0000 0000

この青で示した 6bit を R#5 の上位 6bit に、R#5 の下位 2bit は 1 を指定するので、R#5 に書き込む値は

001 111 11

となります。MSX-BASIC で書くなら、

VDP(5)=&B00111111

ですね。ちなみに、上に緑色で表記したアドレス最上位の 00 に 00 以外を指定したい場合は、R#11 に設定します。

SCREEN4 は、スプライトモード2なので、このアドレスは Sprite Color Table の最後のアドレスの次のアドレスに相当します。Sprite Color Table の先頭アドレスを指定するレジスタが別途用意されておらず、R#5 で指定したアドレス値 - 512 が Sprite Color Table のアドレスとなるためです。図 2.4.1.6-2. に Sprite Color Table と Sprite Attribute Table の位置関係を示します。

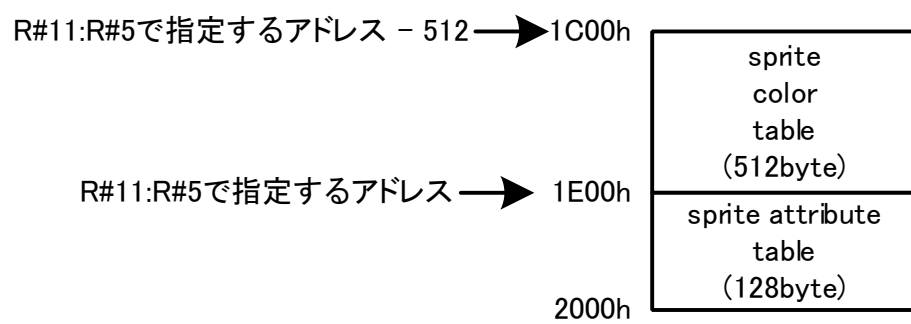


図 2.4.1.6-2. Sprite Color Table と Sprite Attribute Table の位置関係

2.4.1.7. Sprite Pattern Generator Table Base Address Register (R#6)

Sprite Pattern Generator Table の開始アドレスを指定するレジスタです。

TMS9918 では、A13～A0 しかありませんので、A16, A15, A14 は 0 を指定してください。

各ビットの意味を図 2.4.1.7-1. に示します。

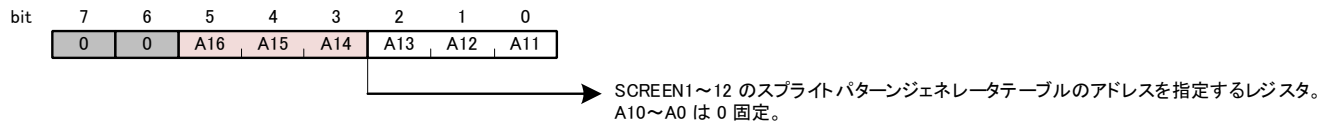


図 2.4.1.7-1. R#6 のビットマップ

Sprite Pattern Generator Table は、スプライトの形状を定義するテーブルなので、これを複数箇所に用意して、R#6 を順番に切り替えることによって、MSX-BASIC からでもスプライト形状のアニメーションを高速に実現することが出来ます。しかし、TMS9918(MSX1)の環境では、アドレスとして指定できるのが A13, A12, A11 の 3bit しかないため、表 2.4.1.7-1. に示した 8 通りしか選択肢がありません。

表 2.4.1.7-1. R#6 (TMS9918) で指定できる値の組み合わせ

Sprite Pattern Generator Table のアドレス	R#6 設定値
0000h	00h
0800h	01h
1000h	02h
1800h	03h
2000h	04h
2800h	05h
3000h	06h
3800h	07h

V9938/V9958 では、表 2.4.1.7-1. の値は当然指定可能です。それに加えて、さらに上位 3bit を指定できるので、64 通りの値を指定できます。例えば、08h を指定すると Sprite Pattern Generator Table の先頭アドレスは 04000h になります。R#6 の設定値 * 0800h が Sprite Pattern Generator Table の先頭アドレスになり、R#6 は、0～63 が指定できることになります。

MSX Documents VDP

Sprite Pattern Generator Table のアドレスを MSX-BASIC で変更するときは、下記の BASE()システム変数を使う方が簡単です。

BASE(スクリーン番号 * 5 + 4)

BASE()システム変数には、VRAM のアドレスそのものを書き込むことで、アドレスを変更できます。より正確に書くと、現在のスクリーンモードが 1 のときに $\text{BASE}(1 * 5 + 4) = \&\text{H}3000$ とすると、ここから R#6 に設定する値を内部で計算して R#6 に設定してくれます。その分遅くなりますが、直感的で分かりやすくなります。

SCREEN1 の Pattern Generator Table は、背景のパターン形状定義のテーブルになりますが、Sprite Pattern Generator Table も同じ構造になっているので、これらを同じアドレスに設定すると、背景とスプライトで同じ形状になります。その実験プログラム(SC1PGT.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN1,0
110 BASE(9)=0
120 FOR I=0 TO 31:PUTSPRITE I,(I*8,I*2),2,ASC("0")+I:NEXT
```

この実行結果を写真 2.4.1.7-1.に示します。



写真 2.4.1.7-1. SC1PGT.BAS の実行結果

100 行目は、SCREEN1 で、スプライトは 8ドット x8ドットサイズで拡大無しに設定。

110 行目は、SCREEN1 の Sprite Pattern Generator Table のアドレスである BASE(9) を 0000h に書換。

MSX Documents VDP

もともと SCREEN1 の VRAM 0000h には Pattern Generator Table が存在するので、MSX-BASIC が文字の形状定義を書き込んでいる場所です。Sprite Pattern Generator Table のアドレスも同じアドレスにするので、この1行で一瞬にして文字形状と同じスプライトパターン形状が定義されたことになります。

120 行目で、32 枚表示可能なスプライトを全部使って緑色の文字(スプライト)を表示させています。スプライトパターン番号と、背景のパターン番号=ASCII コードとが一致しているので、ASC("0")で "0" の ASCII コードを指定することで "0" の形のスプライトパターンを選択できます。

【note】

この状態で `SPRITE$(ASC("0"))="xxxx"` などとしたらどうなるのか?という、何も表示に変化はありません。残念ながら、`SPRITE$()` は `BASE()` システム変数に連動しておらず、SCREEN1 なら「3800h に Sprite Pattern Generator Table が存在している」と決め打って処理されるので、0000h に移動した新しい Sprite Pattern Generator Table には書き込んでくれません。もし連動してくれれば、背景の Pattern Generator Table を `SPRITE$()` を使って書き替えられるようになるわけですが、残念ながら連動しません。

2.4.1.8. Text Color/Back Drop Color Register (R#7)

前景色と周辺色を指定するレジスタです。各ビットの意味を図 2.4.1.8-1.に示します。

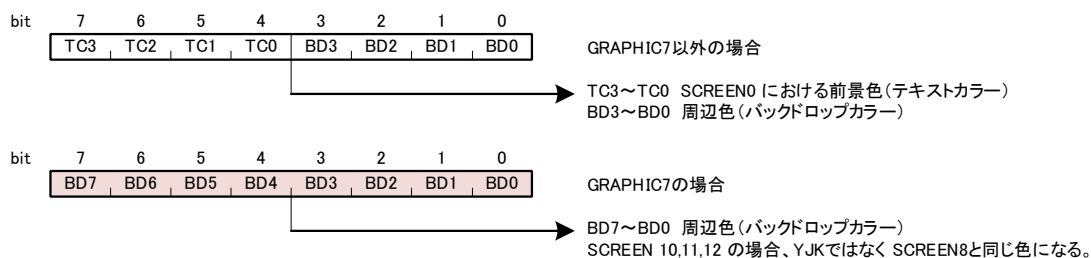


図 2.4.1.8-1. R#7 のビットマップ

各画面モードで若干意味が異なりますので、画面モードごとに説明します。

SCREEN0(WIDTH40, WIDTH80 ともに)の場合、図 2.4.1.8-1. では上の段に記載のビットマップになります。TC3～TC0 の 4bit で文字の前景色を指定します。SCREEN0 は Color Table を持ちませんが、このレジスタで色を指定する仕組みになっています。BD3～BD0 は背景色を指定します。

SCREEN0(WIDTH40)で、R#7 に 3Ch を指定した場合の画面表示を写真 2.4.1.8-1.に示します。



写真 2.4.1.8-1. SCREEN0(WIDTH40)で R#7=3Ch にした結果

前景色が 3, 背景色が 12(=0Ch)になっています。3 は黄緑。12 は濃い緑。

MSX Documents VDP

次に、SCREEN0(WIDTH80)で、R#7 に 3Ch を指定した場合の画面表示を写真 2.4.1.8-2.に示します。



写真 2.4.1.8-2. SCREEN0(WIDTH80)で R#7=3Ch にした結果

次に SCREEN1～4 の場合ですが、こちらも図 2.4.1.8-1.では上の段に記載のビットマップになります。ただし、TC3～TC0 の 4bit は未使用です。

COLOR15,4,7 のときの SCREEN1 で、R#7 に 3Ch を指定した場合の画面表示を写真 2.4.1.8-3.に示します。

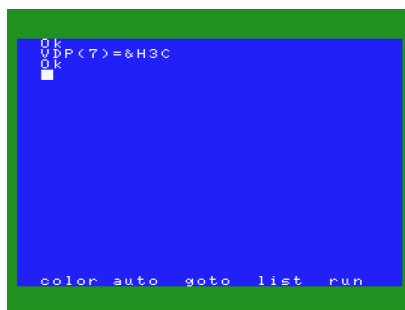


写真 2.4.1.8-3. SCREEN1 で R#7=3Ch にした結果

SCREEN1 には Color Table があり、画面に表示しているパーツそのものが背景色を持っています。写真 2.4.1.8-3. の場合、32byte の Color Table が F4h で敷き詰められているため、前景が 15(0Fh)の白、背景が 4(04h)の濃い青、となっています。周辺を取り囲む領域の色(周辺色)が R#7=3Ch の 12(0Ch)の濃い緑になって現れているのが確認できます。

この周辺色、日本語では周辺色ですが、もともとはバックドロップカラーです。パーツを表示している領域でも、色コード 0 が指定された部分にはこの周辺色が現れることになります。

試しに、画面全体を覆っているスペースのパーツの背景を 0 にしてみましょう。スペースのパーツ番号は 32 で、SCREEN1 の Color Table は、2000h なので、 $2000h + 32/8 = 2004h$ がスペースに対応するカラー情報のアドレスです。ここに、F0h (前景は 15 の白、背景は 0 の透明)を指定してみた画面を写真 2.4.1.8-4.に示します。



写真 2.4.1.8-4. VRAM の 2004h に F0h を書き込んだ結果

記号 & の背景まで緑色になっているのは、スペース(32)と &(38) は、32～39 のグループで 2004h の色を共有しているため連動して変化しています。SCREEN2, 4 もライン単位で色を指定できる点が異なるのみで、前景または背景をカラーコード 0 にすると、周辺色が現れるという点で同様です。

ただし、V9938/V9958 では R#8 の TP(bit5) を 1 にすると、カラーコード 0 を透明ではなくパレット 0 の値にすることが出来ます。その場合、周辺色では無く、パレット 0 に指定している色が背景部分に表示されます。R#8 (MSX-BASIC では VDP(9))の TP を 1 にしてみた結果を写真 2.4.1.8-5. に示します。



写真 2.4.1.8-5. R#8 bit5 を 1 にした場合

パレット 0 は、デフォルトでは COLOR=(0,0,0,0) に設定されているため黒です。R#8 の TP を 1 にすると不透明になり、そのパレット色で表示されるので、写真 2.4.1.8-5. では背景が黒くなったわけです。

TMS9918 では、R#8 は存在しないのでカラーコード 0 は「透明」のみです。ただし、BD3～BD0 にカラーコード 0 を指定した場合は、黒の扱いになります。

SCREEN3 の場合ですが、こちらも図 2.4.1.8-1. では上の段に記載のビットマップになります。ただし、TC3～TC0 の 4 bit は未使用です。

MSX Documents VDP

SCREEN3 は、BD3～BD0 が画面の周辺色になることと、カラーコード 0 で描画した部分の色が BD3～BD0 で指定したカラーコードになります。確認のためのプログラム(R7SC3.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,4,7:SCREEN3
110 LINE(0,0)-(255,191),0
120 VDP(7)=&HF5:FORI=0TO1000:NEXT
130 VDP(7)=&HF7:FORI=0TO1000:NEXT
140 VDP(7)=&HF4:FORI=0TO1000:NEXT
150 GOTO120
```

この実行結果を写真 2.4.1.8-6.に示します。

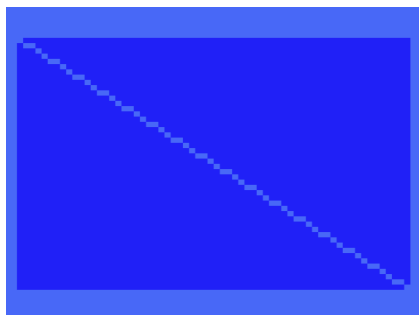


写真 2.4.1.8-6. R7SC3.BAS の実行結果

100 行目で SCREEN3 に変更して、110 行目で左上から右下への斜め線をカラーコード 0 で描画しています。

120 行目で周辺色を 5 (青)に指定して、少し待機。

130 行目で周辺色を 7(シアン)に指定して、少し待機。

140 行目で周辺色を 4(濃い青)に指定して、少し待機。

150 行目で 120 行目に戻る。

実行すると、周辺色だけでなく、斜め線部分の色も変化するのを確認できます。

コントロールレジスタ R#7 を書き替えるだけで、画面全体に及ぼす色を変更できるので、例えばフラッシュアニメーションなどに使うことが出来ます(※くれぐれも激しい点滅をさせてポケ○ンショックを引き起こさないようにご注意ください(笑))。

SCREEN5, 6, 7 も基本的に SCREEN3 と同じです。

MSX Documents VDP

SCREEN8 の場合ですが、こちらは図 2.4.1.8-1. では下の段に記載のビットマップになります。グラフィックの描画に使われる 256 色と同じ色を指定できます。色コードも同様の指定であり、BD7～5 が緑、BD4～2 が赤、BD1～0 が青の輝度となります。また、SCREEN3,5,6,7 と異なり、透明色は存在しません。確認プログラム(R7SC8.BAS)を下記に示します。

```
100 DEFINT A-Z: SCREEN8: COLOR255,128,0:CLS
110 LINE(0,0)-(255,191),0
120 VDP(7)=&HE0:FOR I=0 TO 1000:NEXT
130 VDP(7)=&H1C:FOR I=0 TO 1000:NEXT
140 VDP(7)=&H03:FOR I=0 TO 1000:NEXT
150 GOTO 120
```

この実行結果を写真 2.4.1.8-7. に示します。

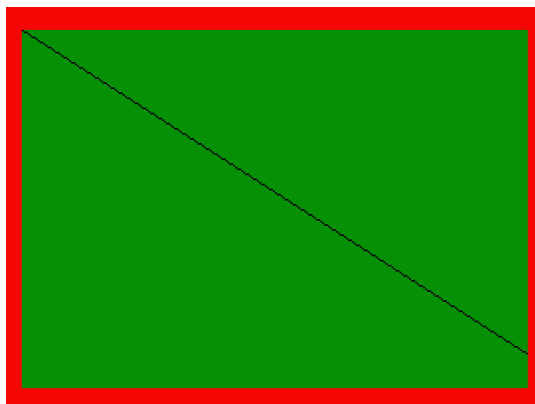


写真 2.4.1.8-7. R7SC8.BAS の実行結果

100 行目で SCREEN8 に初期化して表示クリア。128 は緑色。

110 行目でカラーコード 0 (黒) で左上から右下への斜め線を描画。

120 行目で、周辺色に 緑 100%, 赤 0%, 青 0% を指定して少し待機。

130 行目で、周辺色に 緑 0%, 赤 100%, 青 0% を指定して少し待機。

140 行目で、周辺色に 緑 0%, 赤 0%, 青 100% を指定して少し待機。

150 行目で 120 行目に戻る。

カラーコード 0 で描画した斜め線は黒のまま、周辺色には影響されないことを確認できます。

MSX Documents VDP

SCREEN10,11,12 の場合ですが、こちらは SCREEN8 と全く同じになります。グラフィックの方は YJK+16 色カラーパレットか、YJK による多色表示ですが、連続する 4byte の情報を用いて色を決定しています。R#7 は 1byte の領域しかないで 4byte 収まりません。そのため、これらのモードでは周辺色に限り、SCREEN8 のグラフィックと同じ色指定をすることになっています。確認のためのプログラム(R7SC12.BAS)を下記に示します。

```
100 DEFINT A-Z: SCREEN12: COLOR255, 128, 0: CLS
110 BLOAD "IMAGE1.SCC", S
120 VDP(7) = &HE0: FOR I = 0 TO 1000: NEXT
130 VDP(7) = &H1C: FOR I = 0 TO 1000: NEXT
140 VDP(7) = &H03: FOR I = 0 TO 1000: NEXT
150 GOTO 120
```

この実行結果を写真 2.4.1.8-8. に示します。



写真 2.4.1.8-8. R7SC12.BAS の実行結果

100 行目で SCREEN12 に初期化して表示クリア。

110 行目で画像ファイル IMAGE1.SCC を読み込み。

120 行目で、周辺色に 緑 100%, 赤 0%, 青 0% を指定して少し待機。

130 行目で、周辺色に 緑 0%, 赤 100%, 青 0% を指定して少し待機。

140 行目で、周辺色に 緑 0%, 赤 0%, 青 100% を指定して少し待機。

150 行目で 120 行目に戻る。

R7SC8.BAS の挙動と見比べて頂ければ分かりますが、周辺色の色の変化が全く同じですね。

2.4.1.9. Mode Register 2 (R#8)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

MSX-BASIC では、VDP(9)に対応します。VDP(8)ではないのでご注意ください。

動作モードを指定するレジスタです。各ビットの意味を図 2.4.1.9-1.に示します。

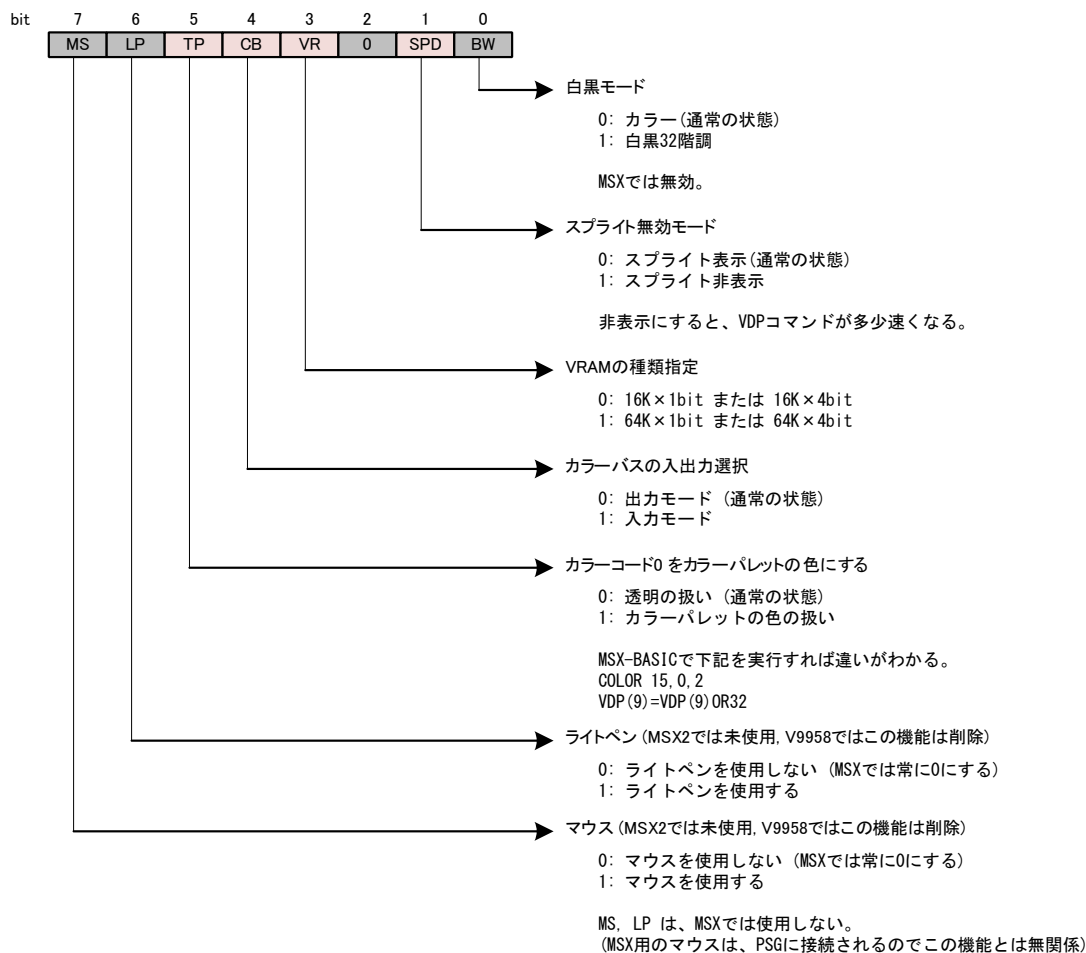


図 2.4.1.9-1. R#8 のビットマップ

BW(bit0)は、MSX では無効です。V9938 の Technical Data Book には、1 にすると 32 階調モノクロモードになると書かれていますが、実際に MSX で 1 を書き込んでも表示に変化は出ませんでした。(※FS-A1GT で確認)

BW に関するより詳細な情報は、3.2. RGB 出力とコンポジット出力を参照ください。

SPD(bit1)は、Sprite Disable ビットになります。1 にするとスプライトを非表示に、0 にするとスプライトを表示します。スプライトが使える画面モードでのみ有効です。Sprite Attribute Table の内容にかかわらず全スプライトを非表示

MSX Documents VDP

に出来ることと、非表示中はスプライト用の DRAM 帯域を VDP コマンドに割り振って VDP コマンドを高速化する役目もあります。

VR(bit3)は、VDPに VRAMとして接続されている DRAM の IC の種類を指定するレジスタです。この値を書き替える则表示が出なくなりますので書き替えてはいけません。0 は 16Kword * 1bit または 16Kword * 4bit の DRAM、1 は 64Kword * 1bit または 64Kword * 4bit の DRAM を示します。FS-A1GT では 64Kword * 4bit の DRAM IC が 4 つ接続されているので 1 になっています。下記 note 参照。

CB(bit4)は、カラーバスの入出力方向を指定します。デジタイズ機能に使うレジスタです。通常は 0 の出力モードとなっています。1 にすると入力モードに切り替わります。カラーバスとは、VDP の 12~19 ピンの C0~C7 のことを示しています。11 ピンの CBDR は、R#8 の CB(bit4)によって CB=0 (出力) なら Low, CB=1 (入力) なら High が出力されます。デジタイズ機能は、アナログ信号を "カラーバスへ入力するデジタルデータ" へ変換する A/D コンバータが必要で、かつ取り込む元となるアナログ信号入力端子も必要となるため、「デジタイズ機能搭載」と銘打っている機種でのみ利用可能です。一般的な MSX2 ではそのような入力が付いていないため、この bit だけ 1 にしても正常に機能しないのでご注意ください。

TP(bit5)は、カラーパレット 0 の扱いを指定します。0 はカラーパレット 0 を透明として扱うモード、1 はカラーパレット 0 を他のカラーパレットと同様に扱うモードとなります。透明とした場合は、カラーパレット 0 (カラーコード 0)で描画した部分は、周辺色と同じ色にすり替わります。スプライトでカラーパレット 0 が指定されている部分も、背景がそのまま表示されます。一方で、他のカラーパレットと同様に扱うモードでは、透き通らずにそのまま指定の色(デフォルトでは黒)が表示されます。SCREEN8,10,11,12 ではカラーパレットではないので TP(bit5)は無効です。2.4.1.8. Text Color/Back Drop Color Register (R#7)に TP(bit5)を 1 にするサンプルがありますので、併せて参照ください。

MS(bit7), LP(bit6) は、VDP がサポートするマウス・ライトペンの機能を利用する場合のレジスタになります。MSX 用のマウスは、ジョイスティックポート(PSG)に接続する仕様になっていますので、この機能は利用していません。MS=1 にした場合は、カラーバスを入力(CB=1)として、カラーバスからマウス信号を入力するようです。MSX ではそのような配線になっていませんので、利用できません。

MSX Documents VDP

【note】

FS-A1STとFS-A1GTの内部写真を、写真2.4.1.9-1と写真2.4.1.9-2に掲載しておきます。この2機種のメインボードは非常に似ていますが、全く同じではありません。しかし、VDP周りは同じようです。両者とも、写真に向かって右側に写っている大きなICがV9958です。左下に4個並んでいるのが64Kword * 4bitのDRAMで、合計128KBになります。

V9938のTechnical Data Bookには、VDPに接続するDRAMとして16KB、32KB、64KB、128KBの選択肢が掲載されています。16Kword * 1bitまたは16Kword * 4bitの設定は、この中の16KBや32KBの場合の選択肢になります。MSX2規格では64KBか128KB、MSX2+/turboRでは128KBのみなので、VR(bit3)を0にすることはありません。16KBはTMS9918との互換のために用意されているモードだと思います。32KBは、64KBに対する128KBと同じ関係にあり、特に用途は無いけども副産物として存在するモードだと思います。16Kword DRAMを使って32KB構成にする場合、前半16KBと後半16KBのアドレスが連続するようにVRAMを構成するとすると、64KB DRAMとは若干アドレスビットの並べ替えが必要になるため、その並べ替えを考慮するためのbitがVRだと思いますが、そうだとすると副産物である32KBモードのためにわざわざVRというレジスタを追加して、かつそれに応じたMUXを搭載しているので、32KBを使った製品を何かしら想定していたのかもしれませんが、32KBあればSCREEN5の1画面分は表現できますからね。実際どういう思想で16KB/32KBの接続を想定していたのかは、中の人でないの分かりません。



写真 2.4.1.9-1. FS-A1ST メインボードのVDP 周辺



写真 2.4.1.9-2. FS-A1GT メインボードのVDP 周辺

2.4.1.10. Mode Register 3 (R#9)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

MSX-BASIC では、VDP(10)に対応します。VDP(9)ではないのでご注意ください。

動作モードを指定するレジスタです。各ビットの意味を図 2.4.1.10-1.に示します。

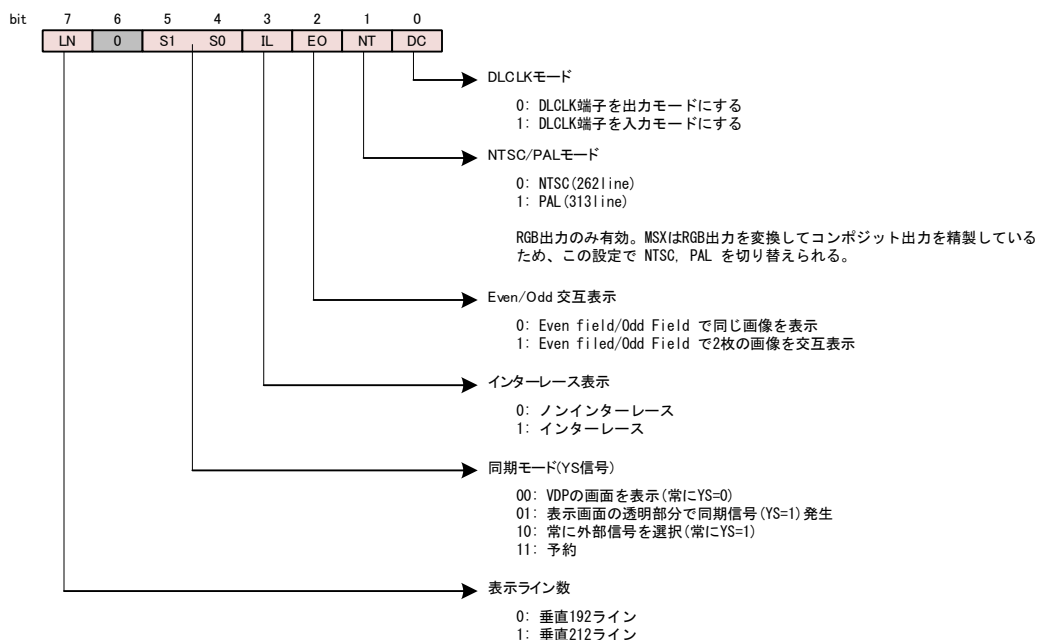


図 2.4.1.10-1. R#9 のビットマップ

DC(bit0) は、VDP の DLCLK 端子の入出力方向を指定するレジスタです。0 を指定すると出力、1 を指定すると入力となります。通常は 0 になります。外部に対して同期用のクロック信号 (5.37MHz) を出力しています。入力モードにすると、外部から 5.37MHz を入れなければ動作しなくなります。通常の MSX で、VDP(10)=1 などと書き込んでしまうと、その途端に VDP が停止して、垂直帰線割込も入らなくなるため、キー入力も受け付けなくなります。それどころか、おそらく VDP に対するレジスタ書き込みも機能しなくなるかと思います。電源を入れ直すまで復帰できなくなりますのでご注意ください。

NT(bit1)は、映像出力のタイミングが NTSC なのか PAL なのかを指定します。NTSC の場合は 0 を、PAL の場合は 1 を指定してください。VDP の 21 ピンのコンポジットビデオ出力だけでなく、RGB 出力にも影響があります。6 ピンの CSYNC 信号がこの設定に従ったタイミング信号になっています。試しに NTSC モニターに接続している状態で、VDP

MSX Documents VDP

(10)=2と打ってみてください。PAL 出力に変化するため、VDPとして出力する1フレーム分のライン数が増加=CSY NCの間隔が間延びするので、画面が垂直方向にグルグルまわるような表示に変わるはずですが。最近の液晶モニターに接続している場合、液晶モニター側が頻繁に同期チェックしているため、グルグル回る表示だけでなく色がおかしくなって止まったり、乱れたり、とても見づらい表示になったりします。50Hz出力に対応しているモニターだと正常に表示される可能性もあるので、ゲームなどで「あたかもMSXが壊れたかのような演出」として使うのも厳しいですね。ちなみにTMS9918では、NTSCやPALといった出力先に対応したICが個別に出ています。つまり、レジスタによってNTSCとPALを切り替えることが出来ません。例えば、TMS9918はNTSC出力・TMS9929はPAL出力といった具合です。

NTSCだと、垂直帰線割込の周期は59.94Hzとなります。PALだと、50Hzとなります。MSXの場合、垂直帰線割込がタイマー割込を兼ねているので、BGM演奏やゲーム速度タイミング調整をこの割込で実現しているソフトが多数あります。そのため、割込周期が変化すると、BGMの演奏テンポが変化したり、ゲームの速度が変化してしまったりするので、それぞれの出力に併せてプログラム側で調整する必要があるのが難点です。

EO(bit2)は、2画面交互表示の有効化レジスタです。0では交互表示は行わない。1では交互表示を行うという意味になります。これはSCREEN5以上でのみ有効で、Pattern Name Tableを奇数ページにしておかねばなりません。下記に2画面交互表示のサンプル(R9EO.BAS)を示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5
110 SETPAGE0,0:BLOAD"IMAGE1.SC5",S
120 SETPAGE1,1:BLOAD"IMAGE2.SC5",S
130 VDP(10)=VDP(10)OR4
140 GOTO140
```

この実行結果を、写真2.4.1.10-1.に示します。

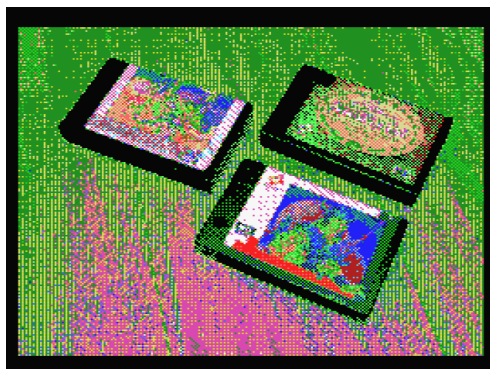


写真 2.4.1.10-1. R9EO.BAS の実行結果

MSX Documents VDP

100 行目で、画面初期化。

110 行目でページ 0 (偶数ページ) に切り替えて、1 枚目の画像 IMAGE1.SC5 を読み込み。

120 行目でページ 1 (奇数ページ) に切り替えて、2 枚目の画像 IMAGE2.SC5 を読み込み。

130 行目で R#9 の EO に 1 を立てる。

140 行目で待機。

奇数ページで無いと EO=1 による交互切替が発動しないので、試しに 125 行目に下記を追加(R9EO2.BAS)してみてください。交互表示しなくなったのを確認できます。

125 SETPAGE0,0

IL(bit3) は、ノンインターレース・インターレースを切り替えるレジスタです。0 でノンインターレース(通常表示)、1 でインターレース(偶数フレームと奇数フレームを垂直方向に半ドットずらす)になります。ブラウン管モニターでは、走査線と走査線の間に隙間がありますが、インターレースにするとこの隙間が詰まったような表示になり、激しく上下に動くので若干チラついたような映像になります。このビットは全画面モードで有効です。

ここで、IL(bit3)と EO(bit2)とを合わせることによって、垂直方向に擬似的に2倍の解像度を持ったような画像を表示できます。サンプル(R9IL.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN7
110 SETPAGE0,0:BLOAD"IMAGE1A.SC7",S
120 SETPAGE1,1:BLOAD"IMAGE1B.SC7",S:COLOR=RESTORE
130 VDP(10)=VDP(10)OR12
140 GOTO140
```

100 行目で SCREEN7 に初期化。

110 行目で偶数ラインの画像を PAGE0 に読み込む。

120 行目で奇数ラインの画像を PAGE1 に読み込み、カラーパレットを設定する。

130 行目で (IL, EO) = (1,1) を設定する。

140 行目で待機。

MSX Documents VDP

この実行結果を写真 2.4.1.10-2. に示します。



写真 2.4.1.10-2. R9IL.BAS の実行結果

S1, S0 (bit5, 4) は、スーパーインポーズのための VDP10pin の/YS 信号を制御するモードを選択します。(S1,S0)=(0,0)の場合 /YS は常に Low。(S1, S0)=(0,1) の場合 /YS はカラーコード 0 が表示されるドットの表示タイミングで /YS が High。(S1, S0)=(1,0)の場合 /YS は常に High。

スーパーインポーズ対応の MSX であれば、/YS 信号が High のタイミングは外部の映像信号を出力するように選択する回路が入っていますが、一般的な非対応の MSX では、この/YS 信号は未使用のため、効果を成しません。たとえば、FS-A1GT の V9958 の/YS は、Open になっており何も繋がっていません。

LN(bit7)は、表示ライン数を選択するレジスタです。0 の場合は 192 ライン(TMS9918 互換)、1 の場合は 212 ラインとなります。

MSX-BASIC で SCREEN0～4 は 192 ライン表示で、SCREEN5 以上は 212 ライン表示になっていますが、このライン数の違いは LN の設定値で変化します。従って、SCREEN0～4 でも LN=1 にすると 212 ラインになりますし、SCREEN5 以上でも LN=0 にすると 192 ラインになります。

2.4.1.11. Color Table Base Address Register High (R#10)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

SCREEN0(WIDTH80)の Blink Table 及び SCREEN1, 2, 4 で利用する Color Table のアドレスの上位ビットを指定するレジスタです。各ビットの意味を図 2.4.1.11-1. に示します。

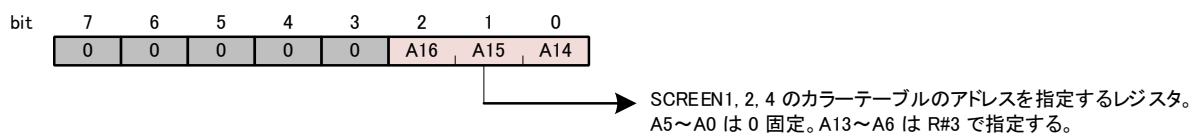


図 2.4.1.11-1. R#10 のビットマップ

意味は、2.4.1.4. Color Table Base Address Register Low (R#3)と同じです。

V9938/V9958 では VRAM は 64KB か 128KB 存在しています。SCREEN1, 2, 4 でも 16KB を越えたアドレスに Color Table をマッピングすることが出来ます。128KB もあるので、Color Table をたくさん用意しておいてアドレス切替によって画面全体の色を瞬時に切り替えてアニメーションさせるといったことが実現できます。レジスタの書換だけなので、MSX-BASIC からでも高速に切り替えられるのです。

MSX-BASIC では、BASE(SCREEN 番号 * 5 + 1) が Color Table のアドレスに対応しています。残念ながら BASE() システム変数は整数値しか書き込めないのので、例えば BASE(6)=65536 と指定するとオーバーフローエラーになってしまいます。VPOKE もアドレス値に整数を指定するので、VPOKE 65536,0 でオーバーフローエラーになります。切替自体は VDP(3), VDP(11) を使い、VRAM の 64KB 越えアドレスには機械語から書き込むのが最もシンプルになるかと思います。

例えば、SCREEN1 で Color Table を 12000h に指定したい場合、下記のように考えます。

(1) 2 進数表記に変換

&H12000 = &B 1 0010 0000 0000 0000

(2) R#10 は A16～A14, R#3 は A13～A6 を指定するので、対応するビットをそれぞれ赤・青で分別

&B 1 0010 0000 0000 0000

MSX Documents VDP

(3) R#10, R#3 とともにビットを右詰め

```
R#10 = &B100
```

```
R#3 = &B10 0000 00
```

(4) これを 16 進数表記に戻し、R#10 は VDP(11), R#3 は VDP(3)に対応するので MSX-BASIC 風に記述する

```
VDP(11) = &H04
```

```
VDP(3) = &H80
```

R#10 の初期値は 0 です。Color Table のアドレスの bit16～14 が 0 で良いなら、R#10 を書き替える必要はありません。つまり、R#10 と R#3 の間に設定順序のシーケンスは存在しないということです。どちらを先に設定してもそれぞれ即時反映されます。

MSX-BASIC で Color Table 切替アニメーションに使うのだとすれば、VPOKE で書き込みが出来る 64KB の範囲で使うのが現実的だと思います。では実際に試してみたサンプル (R10SC1.BAS) を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN1
110 FOR I=0 TO 767:VPOKE &H1800+I,I:AND 255:NEXT
120 FOR I=0 TO 15:FOR J=0 TO 31:VPOKE &H4000+I*32+J,(I+J*16)AND 255:NEXT J,I
130 VDP(10)=1:R3=0
140 VDP(3)=R3:R3=(R3+1)AND 15:FOR I=0 TO 1000:NEXT:GOTO 140
```

この実行結果を写真 2.4.1.11-1.に示します。

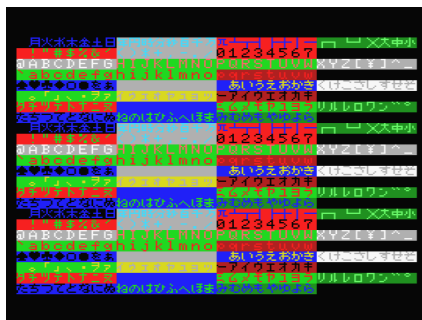


写真 2.4.1.11-1. R10SC1.BAS の実行結果

100 行目で SCREEN1 に初期化。

110 行目で Pattern Name Table を書き替えて画面全体に文字を敷き詰める。

MSX Documents VDP

120 行目で16個の Color Table (I のループ)に対して、32 個(1つの Color Table のサイズが 32byte) のカラー値をある程度値がばらけるように計算して設定 (J のループ)。

130 行目で Color Table の (A16,A15,A14) に (0,0,1) を指定する。R3 に設定する値を保持する変数を 0 に初期化。

140 行目で R#3 に R3 を設定して、R3 を次の Color Table のアドレスに変更し、少し待って繰り返す。

ゆっくり切り替わっているように見えるかもしれませんが、それは 140 行目の FOR 文で待っているからです。この待ちを入れないと激しく切り替わりすぎて目に悪いので入れました。注目すべきは、256 個のキャラクタの色は 32byte の Color Table で決められますが、BASIC から 32byte 書き替えると、それだけで書換を目で追える程度には遅いです。このアドレスを切り替える方法だと、256 個の文字の色がまとめて一度に切り替わっていることが確認できます。

実際に、R#10 と R#3 書換によるアドレス切替ではなく、Color Table を全部書き替える方法を MSX-BASIC からやったらどのくらい遅いのかを確認する比較用のサンプル(R10SC1C.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN1
110 FOR I=0 TO 767:VPOKE&H1800+I,I AND 255:NEXT
120 I=0
130 FOR J=0 TO 31:VPOKE&H2000+J,(I+J*16) AND 255:NEXT
140 I=(I+1) AND 15:FOR J=0 TO 1000:NEXT:GOTO 130
```

実行すると、上からヌルヌルヌルっと書き換わっている状況を目で追えてしまうのが分かりますよね。瞬時に切り替わらないデメリットがあるわけです。それだけでなく、「それだけ時間が掛かる」わけですから、ゲームの演出などに使うと、その時間だけ他の処理が止まってモッサリ動作になってしまうわけです。

SCREEN2, 4 でもアドレスの指定可能範囲は限定 (R#3 の下位ビットが固定のため) されますが、同様に瞬時に切り替えられます。SCREEN2, 4 の場合 Color Table は 6144byte もありますので、もはや BASIC で全書換は現実的ではありません。ただ、TMS9918 の場合は、VRAM が 16KB しかなく、アドレスの選択肢も 0000h と 2000h の2択で Pattern Generator Table も共存していることから、複数の Color Table を確保するだけの VRAM 容量がありません。SCREEN 2, 4 でアドレス切替による色アニメーションを実現するとすれば、実質的に MSX2 以降となります。

2.4.1.12. Sprite Attribute Table Base Address Register High (R#11)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

2.4.1.13. Text Color/Back Color Register (R#12)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

2.4.1.14. Blinking Period Register (R#13)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

2.4.1.15. VRAM Access Base Address Register (R#14)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

2.4.1.16. Status Register Pointer (R#15)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

TMS9918 ではステータスレジスタは S#0 の1つしか存在しないため、ステータスレジスタを選択するレジスタ自体が存在していません。しかし、V9938 では 10 本のステータスレジスタがありますので、そのどれを読み出すのかをこのレジスタで指定することになります。MSX では、MSX1 用に作成されたソフトウェアが問題なく動作するように、通常は R#15 = 0 を維持するように利用し、S#0 以外のステータスレジスタを読む場合は、割込禁止して処理するのが一般的です。BIOS の処理ルーチンも R#15 を変更せずに S#0 を読むので、割込禁止せずに R#15 に 0 以外を設定すると、まず暴走しますのでご注意ください。(定期的に垂直帰線割込が入りますが、この割込要因は S#0 を読み出すことでクリアされます。R#15 が 0 以外にされた状態で割込が発生すると、BIOS は S#0 を読むつもりで異なるステータスレジスタを読み、処理を継続します。すると実際には S#0 は読まれていないため、割込要因がクリアされておらず、割り込み処理ルーチンから抜けようとした瞬間に、再度割込が発生して・・を無限繰り返すようになり、戻ってこなくなります。)

2.4.1.17. Color Palette Address Register (R#16)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

2.4.1.18. Register Pointer (R#17)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

間接指定によるコントロールレジスタへの書き込みにおける書き込み対象のレジスタ番号を指定するためのレジスタになります。

2.4.1.19. Display Adjust Register (R#18)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

画面位置調整機能です。MSX-BASIC でいうところの SET ADJUST 命令に相当します。MSX-BASIC では、これを画面の位置調整に使っており、この設定値は RTC(リアルタイムクロック IC)の中にある記憶メモリで保持しています。MSX-BASIC は、起動時にこの RTC のメモリ内容を読み取って、R#18 にセットすることで画面位置調整を再現するようにしています。

昔のブラウン管テレビでは、ブラウン管の周囲に外装が被っているテレビがあって、その外装によって周辺の映像が隠れてしまうことがありました。アナログ製品なので、表示位置も少しずれたりすることもありました。これを位置調整によって見えるようにしようという機能になります。

しかしながら、水平垂直にそれぞれ-8～+7 の位置ずらしができるので、これをドット単位のハードウェアスクロール機能として利用したゲームも存在していました。(スペースマンボウやサイコワールドなどが有名ですね。)

R#18 のビットマップを図 2.4.1.19-1. に示します。

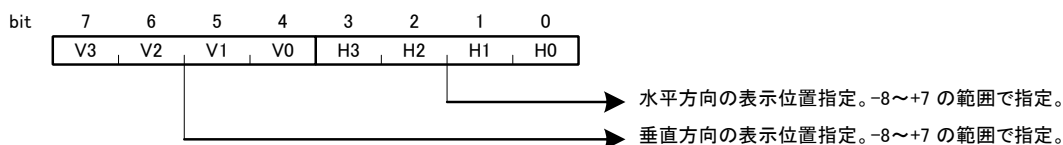


図 2.4.1.19-1. R#18 のビットマップ

上位 4bit に垂直方向の表示位置指定を 2 の補数形式の数値 -8～+7 で指定。下位 4bit に水平方向の表示位置指定を 2 の補数形式の数値 -8～+7 で指定します。

-8 = &B1000, -7 = &B1001, -6 = &B1010, -5 = &B1011, ..., -1 = &B1111, 0 = &B0000, 1 = &B0001, ..., 7 = &B0111 ですね。

では、水平を+方向に調整したら、画面表示は右へ動くの?左へ動くの?というのがこの情報だけでは分かりません。実際に試してみましょう。

水平に -8 した画面を写真 2.4.1.19-1. に、水平に +7 した画面を 2.4.1.19-2. に示します。

MSX Documents VDP

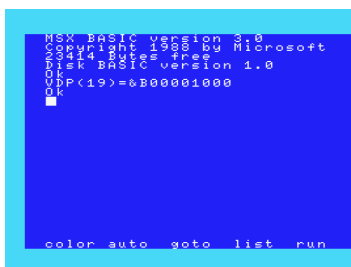


写真 2.4.1.19-1. R#18=&B00001000 (水平-8) の実行結果

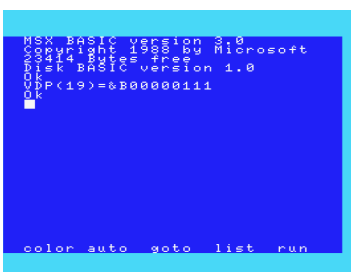


写真 2.4.1.19-2. R#18=&B00000111 (水平+7) の実行結果

マイナス方向は向かって右ヘシフトし、プラス方向は向かって左ヘシフトすることがわかります。垂直も確認してみましょう。垂直-8を写真 2.4.1.19-3.に、垂直+7を写真 2.4.1.19-4.に示します。

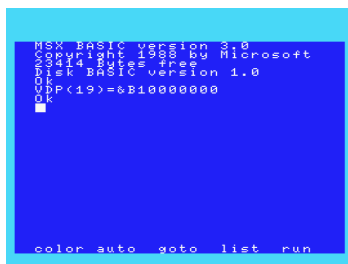


写真 2.4.1.19-3. R#18=&B10000000 (垂直-8) の実行結果

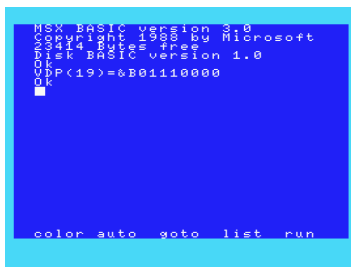


写真 2.4.1.19-4. R#18=&B01110000 (垂直+7) の実行結果

マイナス方向は向かって下ヘシフトし、プラス方向は向かって上ヘシフトすることが分かります。

2.4.1.20. Interrupt Line Register (R#19)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

水平帰線割込の割込走査線番号を指定するレジスタです。2.4.1.1. Mode Register 0 (R#0)の IE1(bit4) を 1 にすると、R#19 で指定したラインの表示タイミングで割込が発生するようになります。

R#19 で指定するライン番号は、あくまでライン番号であり、画面上の走査線番号ではありません。通常は、ライン番号＝走査線番号で一致していますが、2.4.1.24. Display Offset Register (R#23)に 0 以外が設定されていると、その分ズレます。ライン番号はY座標のような意味合いの値です。走査線番号は画面の上端が 0 で、下へ向かって増加する値です。R#23=100 の時に、一番上のラインで水平帰線割込(走査線割込)が発生させる場合は、R#19 に 100 を指定する必要があります。

この機能を使うことによって、画面を上と下に分断し、それぞれで異なる画面モード(R#0/R#1)・異なる画面位置調整(R#18)・異なる垂直スクロール(R#23)・異なるパレットなどを実現することができます。

例えば、縦スクロールシューティングゲームで、画面の上部はスコア表示でスクロールしない領域。画面の下部は垂直スクロールするなんて画面のゲームがありますが、これはこの水平帰線割込を使って R#23 を書き換えることで実現できます。市販ゲームだとアレスタとかゼビウスとかいろいろあります。

上部がスクロールして下部が停止している画面のゲームが少ないのは、上部と下部の切り替わり目のライン番号が上部のスクロールに使う R#23 の設定値によって変化してしまうため、やや処理が面倒(=CPU負荷がかかる)からだと思います。

他にも、スプライトアトリビュートテーブルを画面の中央付近でアドレス変更してしまえば、スプライトを上と下それぞれで32枚同時表示できたりします(これをスプライトダブラーと呼んでいる人もいます)。

パレットを切り替えて上と下で異なるパレットを使うゲームもありますね。幻影都市の背景パレットとキャラ表示パレットがこのような処理でそれぞれ16色使っていました。

一部の機種に搭載されている PAUSE ボタンを押すと、CPU だけ停止して割込が処理されなくなるので、この画面分割の片方だけに固定されて確認できます。

直接的には VDP ではなく Z80/R800 と BIOS の挙動の話になりますが、割り込み処理について簡単に説明しておきたいと思います。VDP のように割り込を発生させるデバイスは、割り込要求を出すときに Z80 の入力ピンである /INT 信号を L に落とします。図 2.4.1.20-1. のような回路になっています。

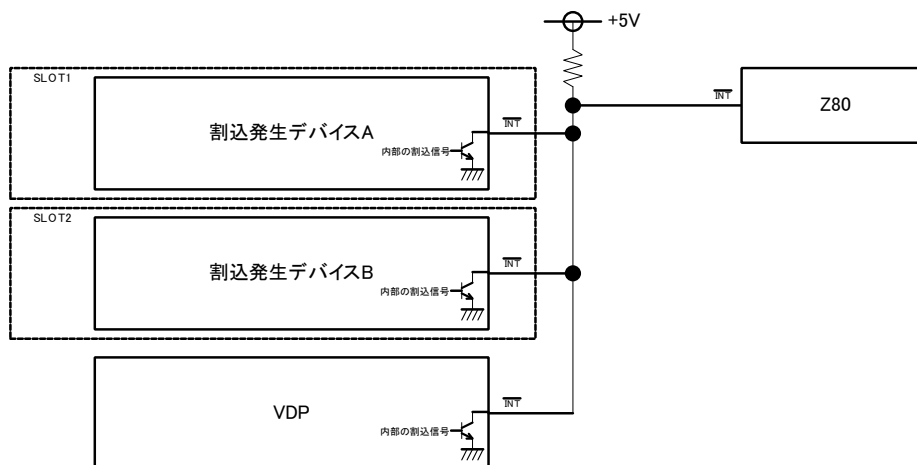


図 2.4.20.1-1. /INT 周りの回路

割込発生デバイス A と B は、カートリッジスロットに装着する何らかのデバイスを想定しています。ゲームカートリッジなどは割込を発生させませんので、/INT 端子は未接続になっています。例えば、turboR 用の MSX-MIDI インターフェースカートリッジである μ PACK は、タイマー IC (i8254) を内蔵していて、タイマー割込を発生させます。

VDP を含む割込発生デバイスの内部の割込信号は、デフォルトで L レベルです。オープンコレクタ接続で /INT が出るので、プルアップ抵抗によって /INT ラインは H レベルに維持されています。

割込発生デバイスは、割込要求をする場合に、内部の割込信号を H レベルにします。/INT は L に落ちます。複数の割込発生デバイスから同時に割込要求が発生することはあり得ます。

Z80 は、ATOMIC (処理の単位) の切れ目のタイミングで /INT=L を検出すると、割込シーケンスに移ります。

割込処理ルーチンで、どの割込デバイスからの割込であるかを判定し、対応する割込デバイスの割込要因クリアの制御を行うと、それを受けた割込デバイスは内部の割込信号を L レベルに落とします。割込要求を出していたすべての割込発生デバイスの内部の割込信号が L レベルになると、/INT ラインは H レベルに戻ります。

割り込み処理ルーチンは、それらの処理を終えると通常処理に戻ります。

このようなハードウェア構成になっています。

MSX Documents VDP

次にソフトウェア面でも見てみましょう。図 2.4.20.1-2. に BIOS の割り込み処理シーケンスを示します。

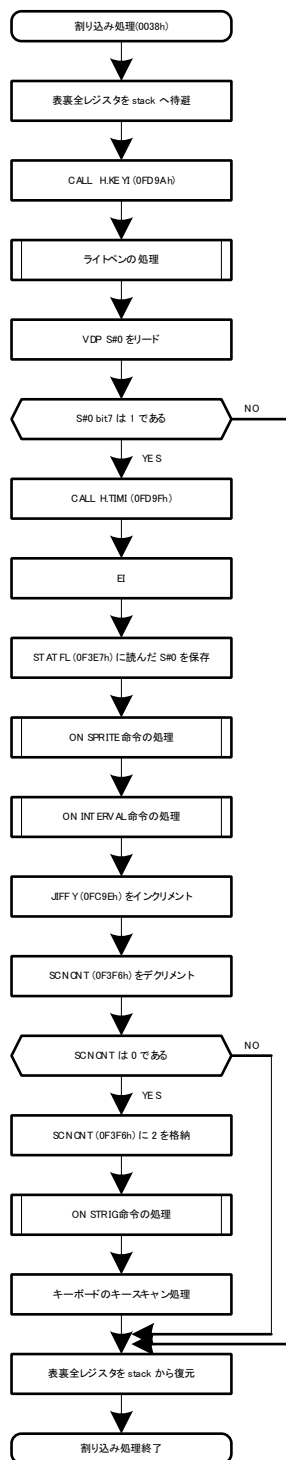


図 2.4.20.1-2. 割り込み処理ルーチンのシーケンス

MSX Documents VDP

Z80/R800 は、/INT 端子に L が投入されたのを検知すると、PCレジスタの値をスタックに積み、0038h 番地へジャンプします。ここには、通常は MAIN-ROM が存在しています。MSX-DOS の稼働時には DRAM になっていますが、0038h 番地から MAIN-ROM の割り込み処理ルーチンへジャンプするコードが置かれています。

MAIN-ROM の割り込み処理ルーチンは、全レジスタを待避してから H.KEYI を呼び出します。"H." は hook の略で、機能をすり替えられるように DRAM 上に配置された 5byte の領域の名前の頭に付けられます。hook 自体は DRAM なので、ここは実行時に書き換えられます。

H.KEYI から戻ってくると、いろいろ処理をした後に VDP S#0 を読み出して、垂直帰線割込であるかどうかをチェックします。垂直帰線割込であれば、H.TIMI を呼び出します。

戻ってきた後は、割込許可をした後、キースキャンなどを実行して戻っていきます。

このようなシーケンスになっているため、VDP の垂直帰線割込以外の割込は H.KEYI で処理する必要があります。

(※FS-A1GT は、MSX-MIDI 関連の割込フック H.MDIN, H.MDTM が存在しており、割込シーケンス内にそれらへの分岐も含まれていますが、VDP とは直接関係の無い部分なのでここでは説明を割愛します。)

ということで、水平帰線割込(走査線割込)は、垂直帰線割込と同じ VDP 発生割込ですが、H.KEYI で処理することになります。H.KEYI は垂直帰線割込も含むすべての割込のタイミングで呼ばれるので、H.KEYI をフックして実行するルーチンは、動いて良いタイミング判断する処理が必要です。

水平帰線割込のタイミングかどうかは、Status Register 1 (S#1) の bit0 をみて判断します。0 であれば他の割込み。1 であれば水平帰線割込です。この bit は、割込要因(図 2.4.20.1-1.の内部の割込信号)にもなっていて、かつ読み出すと 0 にクリアされます。つまり、水平帰線割込を使う設定をしておいて、水平帰線割込によって H.KEYI に入ってきたところで S#1 を読まなかった場合、割り込み処理シーケンスの中の EI をした瞬間にまた割り込んでスタックメモリをどんどん消費して、いずれプログラムの置いてあるメモリを破壊して暴走します。プログラムが ROM 上だったとしても、メモリを一巡して、スロットバンクセクタである 0FFFFh に何かを書き込んでスロットが切り替わり暴走します。従って、忘れずに S#1 を1度だけ読んでください。読むとクリアされるレジスタなので1度だけです。2度読むはいけません。

H.KEYI のフックは、他のフックと同様、他のプログラムがすでに利用中の可能性があるので、5byte すべてコピーして保存しておき、自身の H.KEYI 用ルーチンを実行した後にそちらを呼ぶようにして、他のプログラムを邪魔しないようにします。MSX の場合、普通は1つのプログラムしか動作しませんが、ドライバの類(外付けFDDの制御プログラムなど)は裏で動いていたりするので、それらを妨害しないための対策です。

H.KEYI のフックの仕方の例を下記に示します。

MSX Documents VDP

H_KEYI = 0xFD9A

```
; -----
; H.KEYI にフックする
; input)
;  無し
; output)
;  無し
; break)
;   AF, BC, DE, HL
; comment)
;   2 回呼んではいけません。
; -----
set_hkeyi_hook::
    ld    hl, H_KEYI
    ld    de, previous_hkeyi_hook
    ld    bc, 5
    di                                ; H_KEYI を他の割込が書き換えないように割禁
    ldir

    ld    hl, hkeyi_routine_entry    ; H_KEYI 書き換えてる途中で割り込まないように割禁維持
    ld    [ H_KEYI + 1 ], hl
    ld    a, 0xC3
    ld    [ H_KEYI ], a              ; H_KEYI には JP hkeyi_routine_entry を書く
    ei                                ; 書き換え完了したので割込許可
    ret

; -----
hkeyi_routine_entry::
    ; ここに H_KEYI の処理ルーチンを書く

previous_hkeyi_hook::
    db    0xC9, 0xC9, 0xC9, 0xC9, 0xC9
```

MSX Documents VDP

次に H.KEYI に配置する水平帰線割込処理ルーチンの例を示します。(※VDP の I/O ポートを定数にしているため、このサンプルは MSX バージョンアップアダプタには対応していません。)

VDP_IO_PORT1 = 0x99

```
; -----  
; H.KEYI 割込処理ルーチン (水平帰線割込用)  
; input)  
;  無し  
; output)  
;  無し  
; break)  
;  全レジスタ  
; -----  
hkeyi_routine_entry::  
    ; S#1 を読む、割禁状態で来るので割禁にする必要なし  
    ld    c, VDP_IO_PORT1  
    ld    a, 1  
    out   [c], a  
    ld    a, 15 | 0x80  
    out   [c], a          ; R#15 ← 1  
    in    a, [c]          ; a ← S#1  
    ld    b, 0  
    out   [c], b  
    ld    b, 15 | 0x80    ; R#15 ← 0  
    out   [c], b  
    ; 水平帰線割込かどうか調べる  
    and   a, 0x01  
    jp    z, previous_hkeyi_hook    ; 水平帰線割込でなければ前の hook へ飛ぶ  
  
    ; ここに 水平帰線割込でやりたい処理を書く、終わったらそのまま previous_hkeyi_hook へ  
  
previous_hkeyi_hook::  
    db    0xC9, 0xC9, 0xC9, 0xC9, 0xC9
```

このようになります。割り込み処理ルーチンですが、レジスタの待避・復元は BIOS の方でやってくれるので、フックしたルーチン内でやる必要はありません。

MSX Documents VDP

画面に何らかの効果を与える処理を実施する場合、垂直帰線割込と組み合わせると便利です。垂直帰線割込で画面の上側の設定を行い、水平帰線割込で画面下側の設定を行うといった具合です。垂直帰線割込は、H.TIMIを使う点と、割込要因のチェックは BIOS 側でやってくれるので自身でステータスレジスタの読み出しを行わなくて良い点が違うのみです。

これらを踏まえて簡単なサンプル(R19VSCR.ASM, R19VSCR.BAS)を下記に示します。SCREEN5 を使用して、画面の Page0 に画像 1(IMAGE1.SC5), Page1 に画像 2(IMAGE2.SC5)を表示。水平帰線割込をライン 106 に指定して、画面の上半分を Page0, 下半分を Page1 にして、それぞれの画像に合わせたパレット設定と、下半分の垂直スクロールをします。写真 2.4.1.20-1.にこの実行イメージを示します。



写真 2.4.1.20-1. R19VSCR.BAS の実行イメージ

上半分と下半分で異なるパレットになっているため、同時に32色出ているわけですが、パレットの全切替には CPU から VDP へ 32byte の転送が必要です。この転送時間は、VDP が数ライン表示処理を進めてしまうほどの時間がかかるため、上と下の画面の継ぎ目の部分で、上の画面の色が少し下の画面を少し浸食しているような表示になっています。

市販ゲームなどでは、この切り替わり目の部分は黒帯などにして、黒を表現するパレットは同じ番号にするなどの工夫で、数ラインかけたパレットの変更が表示上では見えない・目立たないようにしていることが多いです。

また、BIOS を経由した割込処理は、無条件に裏レジスタを含む全レジスタをスタックに待避するコードが動いたりするので、それだけで VDP が数ライン処理を進めてしまいます。H.KEYI に入ってくるタイミングは、指定したラインからおよそ3ライン程度進んだタイミングで到達します。随時 R#19 を書き換えれば、1画面中何度も割込を入れることは可能ですが、100 ライン目に入れて、すぐ 101 ライン目に・・という短い間隔で入れることは不可能です。

MSX Documents VDP

F1 Spirit 3D Special のようなゲームでライン単位で水平スクロールレジスタを制御しているように見えるソフトウェアがありますが、ライン単位の水平スクロールの一番上に相当する部分のみ R#19 による割込で検知して、それ以降はステータスレジスタ S#2 の HR(bit5) をポーリングしてタイミングをとっていると思います。

では、具体的に R19VSCR.ASM の中身の説明に移ります。

```
; -----  
;   R#19 sample program  
; =====  
;   2020/04/01   t.hara  
; -----  
  
RG0SAV          = 0xF3DF  
H_KEYI          = 0xFD9A  
H_TIMI          = 0xFD9F  
VDP_IO_PORT0    = 0x98  
VDP_IO_PORT1    = 0x99  
PAGE0_PALTBL    = 0x7680  
PAGE1_PALTBL    = 0xF680  
LINE_NO         = 106           ; 水平帰線割込を発生させるライン番号
```

ここでは、各種定数を定義しています。表 2.4.1.20-1.に内容をまとめます。

表 2.4.1.20-1. 定数の意味

定数名	
RG0SAV	BIOS が保持している「R#0 に書き込んだ値」のバックアップ値。 R#0 は読み出せないので、BIOS は最後に書いた値をここに保存している。
H_KEYI	すべての割り込み処理の入り口で呼ばれるフック。
H_TIMI	VDP の垂直帰線割込の時に呼ばれるフック。
VDP_IO_PORT0	VDP の Port#0 の I/O アドレス。
VDP_IO_PORT1	VDP の Port#1 の I/O アドレス。
PAGE0_PALTBL	SCREEN5 の画面の PAGE0 に対応する Palette Table のアドレス。 IMAGE1.SC5 には、BASIC から COLOR=RESTORE でパレット設定できるように、Palette Table の位置にパレット情報が記録してある。
PAGE1_PALTBL	SCREEN5 の画面の PAGE1 に対応する Palette Table のアドレス。

IMAGE2.SC5 には、BASIC から COLOR=RESTORE でパレット設定できるように、Palette Table の位置にパレット情報が記録してある。

LINE_NO

水平帰線割込を発生させるライン番号。

続き。

```
; BSAVE file header
db      0xFE
dw      start_address, end_address, start_address
```

BSAVE ファイルフォーマットの機械語ファイルを生成するための、BSAVE ファイルヘッダ定義になります。

最初の 1byte は 0FEh で、BSAVE ファイルですよ、というシグネチャになっています。

次の 2byte は、機械語コードをどの番地に配置するかを示す先頭アドレスをリトルエンディアンで格納しています。

次の 2byte は、機械語コードの終了アドレスをリトルエンディアンで格納しています。

次の 2byte は、BLOAD"xxx",R でロード後すぐに実行する場合、どこから実行開始するかを示す実行開始アドレスをリトルエンディアンで格納しています。

続き。

```
; -----
; entry point
; -----
    org      0xC000
start_address::
    call     initialize
main_loop::
    nop
    jp      main_loop
```

プログラムコードは、0C000H 番地に配置するので、ORG で 0xC000 を指定しています。

これにより、start_address で示されるアドレスは 0C000H になります。

initialize は、後述の初期化ルーチンのアドレスです。

MSX Documents VDP

main_loop は、何もせずに待機するだけの無限ループになっています。今回のプログラムは、割込が主役なので、非割込ルーチンは、何もしません。

続き。

```
; -----  
; 読みだし用に VRAM アドレスをセットする  
; input)  
;   HL ... VRAM アドレス  
;   DE ... 読み出した値を格納する DRAM アドレス  
;   B .... 読み出すバイト数  
; output)  
;   無し  
; break)  
;   AF, BC  
; -----  
read_vram_block::  
    ; VRAM アドレス設定  
    ld     c, VDP_IO_PORT1  
    ld     a, h  
    rlca  
    rlca  
    and    a, 0x03  
    out    [c], a  
    ld     a, 14 | 0x80  
    out    [c], a      ; R#14 ← [0, 0, 0, 0, 0, A16=0, A15, A14]  
    ld     a, h  
    out    [c], l      ; Port#1 ← [A7, A6, ... , A1, A0]  
    and    a, 0x3F  
    out    [c], a      ; Port#1 ← [0, 0, A13, A12, ... , A8]  
  
    ; 読み出す  
    ex     de, hl  
    dec    c  
    inir  
    ret
```

read_vram_block は、MSX-BASIC で書くと下記のような処理になります。

```
FOR I=1 TO B: POKE DE+I, VPEEK( HL+I ):NEXT I
```

MSX Documents VDP

続き。

```
; -----  
; 割込フックの初期化と水平帰線割込の許可  
; input)  
; 無し  
; output)  
; 無し  
; break)  
; AF, BC, DE, HL  
; comment)  
; 2 回呼んではいけません。  
; -----  
initialize::  
    di  
    ; H.KEYI をバックアップ  
    ld    hl, H_KEYI  
    ld    de, previous_hkeyi_hook  
    ld    bc, 5  
    ldir  
  
    ; H.TIMI をバックアップ  
    ld    hl, H_TIMI  
    ld    de, previous_htimi_hook  
    ld    bc, 5  
    ldir
```

先ほど call initialize として呼ばれていた初期化関数の中身です。長いので分解して説明します。

上に取り上げた部分、まず DI して割込を禁止しています。これから割込フックに関する処理を実施するわけですが、バックアップをとっている最中に割込が発生して、割り込み処理内でフックの内容を書き換えるようなプログラムが動いていると、期待したとおりにバックアップできないため割込禁止しておきます。

まず最初に、H.KEYI をバックアップします。5byte の領域なので BC=5、バックアップの保存先として previous_hkeyi_hook を指定しています。

次に、H.TIMI をバックアップします。こちらも 5byte の領域なので H.KEYI と同じようにバックアップします。バックアップの保存先は previous_htimi_hook になります。

H.KEYI と H.TIMI は連続する 10byte なので、バックアップをとるだけであればまとめて LDIR しても問題ありません。

続き。

MSX Documents VDP

```
; このプログラムの H.KEYI 処理ルーチンへのジャンプ命令を H.KEYI に書き込む
ld    hl, hkeyi_routine_entry ; H_KEYI 書き換えてる途中で割り込まないように割禁維持
ld    [ H_KEYI + 1 ], hl
ld    a, 0xC3
ld    [ H_KEYI ], a           ; H_KEYI には JP hkeyi_routine_entry を書く

; このプログラムの H.TIMI 処理ルーチンへのジャンプ命令を H.TIMI に書き込む
ld    hl, htimi_routine_entry ; H_KEYI 書き換えてる途中で割り込まないように割禁維持
ld    [ H_TIMI + 1 ], hl
ld    a, 0xC3
ld    [ H_TIMI ], a           ; H_KEYI には JP hkeyi_routine_entry を書く
```

ここでは、自分の処理ルーチンへジャンプするようにフックに新しい内容を書き込んでいます。当然ながら、割込禁止は維持したままです。5byteしかない領域なので、JP 命令か、CALL 命令+RET 命令か RST 命令+RET 命令を配置するのがほとんどです。どこかに新しいフックの内容を書いておいて、それを LDIR で転送しても構いませんが、今回は 0C3h, XX, XX をそのまま書き込んでいます。

続き。

```
; R#19 に LINE_NO をセットする
ld    c, VDP_IO_PORT1
ld    a, LINE_NO
out    [c], a
ld    a, 19 | 0x80
out    [c], a
```

ここでは、水平帰線割込が発生してほしいライン番号を R#19 に書き込んでいます。今回は画面中央付近の 106 にしています。

続き。

```
; R#0 の IE1:bit4 を 1 にする
ld    a, [ RG0SAV ]           ; BIOS が R#0 に書き込んだ内容が保存されているワーク
or    a, 1 << 4               ; bit4 を 1 にする
out    [c], a
ld    a, 0 | 0x80
out    [c], a                 ; R#0 に求めた値を書き込む
```

MSX Documents VDP

R#19 にライン番号を書き込んだだけでは水平帰線割込は有効になりません。有効にするためには R#0 の IE1(bit 4)を 1 にする必要があります。他の bit は画面モード設定などなので、書き換えてはいけません。bit4 だけ書き込むことはできないので、BIOS が保持している「最後に R#0 に書き込んだ値」である RG0SAV を読み出して、その bit4 を 1 にした値を R#0 に書き込むことで bit4 だけ書き換えたような状態を作り出しています。

続き。

```
; VRAM上の Palette Table を読み出してワークエリアにコピーする
ld    hl, PAGE0_PALTBL
ld    de, page0_color_palette
ld    b, 32
call  read_vram_block

ld    hl, PAGE1_PALTBL
ld    de, page1_color_palette
ld    b, 32
call  read_vram_block
```

先ほど説明したサブルーチン read_vram_block を使って、VRAM 上に存在する Palette Table の値を読み取って DRAM (page0_color_palette, page1_color_palette) へコピーしています。

続き。

```
ei          ;書き換え完了したので割込許可
ret
```

初期化処理がすべて終わったので戻ります。戻る前に、割込許可に戻しておきます。

MSX Documents VDP

続き。

```
; -----  
; H.TIMI 割込処理ルーチン (垂直帰線割込用)  
; input)  
;   無し  
; output)  
;   無し  
; break)  
;   全レジスタ  
; -----  
htimi_routine_entry::  
    ; 垂直スクロールレジスタ (R#23) へ 0 を設定する  
    ld      c, VDP_IO_PORT1  
    xor     a, a  
    out     [c], a  
    ld      a, 23 | 0x80  
    out     [c], a
```

ここからは、垂直帰線割込になります。垂直ブランキング期間の開始タイミングで発生する割込ですので、表示画面の下端を表示し終えたタイミングと思えばいいですね。その後は画面の上から表示し始めるので、ここには上側の画面の設定を詰め込みます。

画面を上下に分割して上側の画面をスクロール固定して表示するので、R#23 に 0 を書き込んで定位置に止めています。

続き。

```
; 水平帰線割込を ON にする  
ld      a, LINE_NO  
out     [c], a  
ld      a, 19 | 0x80  
out     [c], a
```

R#23 をいじらない水平帰線割込利用であれば、初期化時に1回設定すれば良いのですが、今回は毎フレーム設定しています。その理由については後述します。ここではライン 106 で水平帰線割込が発生するように設定していると覚えておいてください。

MSX Documents VDP

続き。

```
; 表示ページを 0 に切り替える (Pattern Name Table を 0x000000 にする: R#2 ← 0x1F)
ld      a, 0x1F                                ; [0, A16=0, A15=0, 1, 1, 1, 1, 1]
out      [c], a
ld      a, 2 | 0x80
out      [c], a
```

R#2 に 0x1F を書き込んでいます。これは MSX-BASIC の SET PAGE 0 と同じような意味です。上下分割の上側は page0 を表示したいのでこのようにしています。

続き。

```
; Page0 のパレットをセットする
xor      a, a
out      [c], a
ld      a, 16 | 0x80
out      [c], a

inc      c
ld      b, 32
ld      hl, page0_color_palette
otir

; 前の H.TIMI の処理 (0xC9 は ret 命令)
previous_htimi_hook::
db      0xC9, 0xC9, 0xC9, 0xC9, 0xC9
```

さらに、page0 に読み込んだ IMAGE1.SC5 用のパレットをパレットレジスタに書き込んで設定します。Palette Table は、パレットレジスタと同じフォーマットになっているので、そのまま書き込むだけで OK です。

それを終わったら、バックアップしていた「以前の H.TIMI フックの内容」をそのまま実行します。previous_htimi_hook にバックアップしてるので、ジャンプ命令無しで実行されますね。

MSX Documents VDP

続き。

```
; -----  
; H.KEYI 割込処理ルーチン（水平帰線割込用）  
; input)  
;     無し  
; output)  
;     無し  
; break)  
;     全レジスタ  
; -----  
hkeyi_routine_entry::  
    ; S#1 を読む、割禁状態で来るので割禁にする必要なし  
    ld      c, VDP_IO_PORT1  
    ld      a, 1  
    out     [c], a  
    ld      a, 15 | 0x80  
    out     [c], a                ; R#15 ← 1  
    in      a, [c]                ; a ← S#1
```

ここは、H.KEYI の割り込み処理ルーチンです。

水平帰線割込による割込なのか判定しなければならないため、まず S#1 を読みます。

S#1 を読むには、R#15 に 1 を設定し、VDP Port#1 から読み出せば OK です。

続き。

```
    ; 水平帰線割込かどうか調べる  
    and     a, 0x01  
    jp      z, finalize          ; 水平帰線割込でなければ前の hook へ飛ぶ
```

S#1 の bit0 が水平帰線割込フラグになるため、ここが 1 になっているかどうかを判定しています。

0 の場合、他の割込なので、finalize へ飛びます。

S#1 を読んだタイミングで、自動的に S#1 の bit0 は 0 にクリアされて、VDP から CPU に対して出力されている /INT 信号は Hレベルへ戻ります。

続き。

```

; 垂直スクロールレジスタへ設定する値を更新する
ld      a, [vertical_scroll]
inc     a
ld      [vertical_scroll], a

; 垂直スクロールレジスタ (R#23) に設定する
out     [c], a
ld      a, 23 | 0x80
out     [c], a                      ; R#23 ← vertical_scroll

```

下画面は、垂直方向にスクロールさせます。そのために R#23 に設定する値 `vertical_scroll` をインクリメントしています。インクリメントした値を R#23 にそのまま設定しています。ここで注意が必要です。R#19 に指定する値は、出力モニター上の上を 0 とした Y 座標ではなく、あくまで VRAM 上の Y 座標になります。R#23 によって表示位置が垂直方向にずれると、R#19 で指定した値に対応するライン位置も一緒にずれます。そのイメージを図 2.4.20.1-3. に示します。

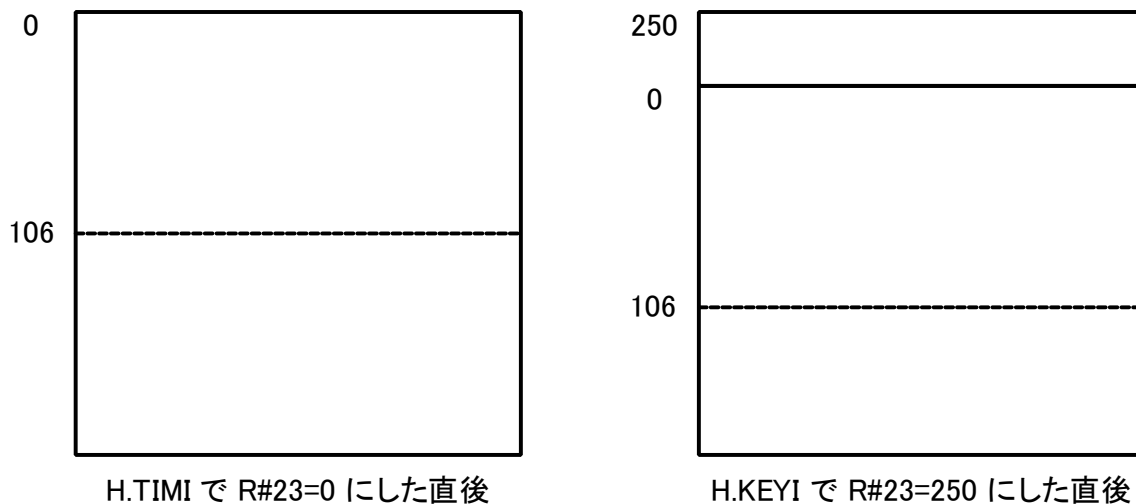


図 2.4.20.1-3. R#23 の設定値による R#19 の割込発生位置ずれイメージ

まず、R#19 = 106 としています。ライン 106 が表示されるときに水平帰線割込が入ることを期待しています。垂直帰線割込で R#23 = 0 に設定しているの、画面表示更新が上部のタイミングでは画面表示の上から 106 ライン目が水平帰線割込のタイミングになります。図の左側の点線位置ですね。

画面表示が図の左側 106 ライン目まで到達すると、割込が発生し H.KEYI が呼ばれ、上記 `hkeyi_routine_entry` に入ってきます。

MSX Documents VDP

水平帰線割込で入ってきているので、当然 S#1 の bit0 は 1 ですね。R#23 を更新に行きます。ここで、vertical_scroll のインクリメントの結果が、例えば 250 だった場合、図の右側のような状態に切り替わります。

すると、R#23 を更新する前に ライン 106 だったラインが、下へ少しずれた位置に現れます。

この「ずれたライン 106」に表示が到達すると、そのタイミングでまた水平帰線割込が発生してしまいます。VDP としては、S#1 を読んだ時点で、先ほどの水平帰線割込は処理されたものとして bit0 を 0 に戻していますが、「ずれたライン 106」のタイミングでまた S#1 bit0 を 1 にして、/INT 信号を L レベルにしてしまいます。

CPU 側は、この後の処理を終えて割込を抜けるわけですが、割り込み処理を抜けたタイミングで /INT 信号が L になっていることに反応して、すぐさままた割り込み処理ルーチンへ飛び、また hkeyi_routine_entry を実行してしまいます。

hkeyi_routine_entry の中で vertical_scroll をインクリメントしていますから、この「1画面の中で複数回 hkeyi_routine_entry が呼ばれること」の対策を入れておかないと、この複数回呼ばれる垂直スクロール位置だけスクロール速度が速くなったような動きになってしまいます。

これを避けるための対策として、R#23 を変更した直後に、R#19 に R#23 と同じ値を設定してしまいます。R#23 に設定する値は、画面の上端のライン番号になるわけですが、hkeyi_routine_entry が呼ばれるのは画面の上から 106 ライン目の位置なので、R#19 = R#23 にすることで、「次の H.TIMI 発生までの間に水平帰線割込を発生させない」という状況を作り出せます。

他のやり方として、hkeyi_routine_entry が来たら R#0 の IE1 を 0 にして水平帰線割込を OFF にして、H.TIMI で ON に戻すという方法でもできます。しかし、R#0 は VDP のモードレジスタで、他の機能の bit も含まれるため、私は R#19 = R#23 の方法が手軽かと思います。モードレジスタは、モードが切り替わると、VRAM である DRAM のコントローラーに対してアクセスタイミングを変える指示をするらしく、VDP コマンド稼働中にいじると、VDP コマンドによる VRAM アクセスを妨害することがあります。そういった意味でも、R#19 = R#23 の方法が無難です。

この対策が下記の処理になります。

```
； 垂直スクロールの影響で、即座に水平帰線割込が入ってしまわないように R#19 をずらしておく
ld      a, [vertical_scroll]
out      [c], a
ld      a, 19 | 0x80
out      [c], a

； R#23 を変更してから R#19 の変更を終えるまでの間に 次の水平帰線割込が発生してしまう
； ラインがあるので、それをキャンセルするために S#1 を空読み
in      a, [c] ; a ← S#1
```

MSX Documents VDP

R#19 に R#23 となるように vertical_scroll の内容を書き込んでいます。

R#23 設定→R#19 設定の順で処理していますが、1つずつ処理するため、これら2つのレジスタ設定には若干のタイムラグがあり、並列して動いている VDP は、この間に「ずれたライン 106」を検知してしまう可能性があります。

検知してしまった場合は、S#1 bit0 が 1 になっているので、この不要な割込要因をクリアするために S#1 を空読みします。この空読みを簡単にするために、hkeyi_routine_entry 冒頭の R#15 = 1 は、R#15 = 0 に戻す処理を入れていません。R#15 = 0 は、BIOS へ戻る前までにやれば問題ありません。

続き。

```
; 表示ページを 1 に切り替える (Pattern Name Table を 0x08000 にする: R#2 ← 0x3F)
ld      a, 0x3F                                ; [0, A16=0, A15=1, 1, 1, 1, 1, 1]
out     [c], a
ld      a, 2 | 0x80
out     [c], a

; Page0 のパレットをセットする
xor     a, a
out     [c], a
ld      a, 16 | 0x80
out     [c], a

inc     c
ld      b, 32
ld      hl, page1_color_palette
otir
dec     c
```

R#2 (Pattern Name Table Base Address) に Pattern Name Table が 8000h になるように 3Fh を設定しています
下の画面用のパレットも設定しています。

続き。

```
; R#15 を S#0 を指すように戻す
finalize::
xor     a, a
out     [c], a
ld      a, 15 | 0x80                            ; R#15 ← 0
out     [c], a

; 前の H.KEYI の処理 (0xC9 は ret 命令)
previous_hkeyi_hook::
db      0xC9, 0xC9, 0xC9, 0xC9, 0xC9
```

R#15 は、S#0 を示すように 0 を設定しておきます。

MSX Documents VDP

previous_hkeyi_hook には、H.KEYI に元々入っていた 5byte が格納されているので、そちらへ処理をボタンタッチします。

【note】

FS-A1GT では、内蔵の MSX-MIDI だけ特別扱いをしており、内蔵 MSX-MIDI が発生させるタイマー割込や MIDI-IN 割込を真っ先に処理するように割り込み処理ルーチン内で判断しています。これは、特に MIDI-IN が最速で 5μsec 間隔で割込を発生させるため、R800 でも処理速度的にシビアであることが理由かと思います。そのため、CPU が割込を認識して 0038h が呼ばれてから、H.KEYI が呼ばれるまでの処理時間は少し遅くなります。しかし、FS-A1ST+μPACK だと、普通に H.KEYI 使って MSX-MIDI 割込を処理することになるので、実際は H.KEYI でも間に合うのかもしれませんが。わざわざ内蔵と外付けで MSX-MIDI の割込フックを分けたのは、規格策定を吟味する時間が無かったのではないかなと、勝手に想像しています。

```
; -----  
; ワークエリア  
; -----  
vertical_scroll::  
    db          0  
page0_color_palette::  
    ds          " " * 32          ; 32byte の領域を確保  
page1_color_palette::  
    ds          " " * 32          ; 32byte の領域を確保  
end_address::
```

vertical_scroll は、R#23 に書き込む値を保持している領域。1byte です。

page0_color_palette は、IMAGE1.SC5 の Palette Table の値をコピーした 32byte の領域。

page1_color_palette は、IMAGE2.SC5 の Palette Table の値をコピーした 32byte の領域。

【note】

一般的なアセンブラは、ds は Define Space ですが、ZMA は Define String となっています。スペースを確保する疑似命令がないので、32byte の文字列を配置することで 32byte の領域確保としています。文字列に対する *32 は、その文字列を 32 回繰り返した文字列、という意味の演算子になります。

2.4.1.21. Color Burst Register 1 (R#20)

2.4.1.22. Color Burst Register 2 (R#21)

2.4.1.23. Color Burst Register 3 (R#22)

R#20～R#22 は V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

21 ピンのコンポジットビデオ出力のカラーバースト信号を作るためのパラメータを指定するレジスタですが、MSX では 21 ピンのコンポジットビデオ出力は使っていない(使っている機種があるかは不明。使っていない機種が複数存在することは確認済み。)ため、このレジスタは MSX では無効です。

この3つのレジスタにすべて 0 を書き込むと、21 ピンの出力は出なくなります。

ちなみに、R#20=00h, R#21=3Bh, R#22=05h が初期値で、21 ピン出力を利用するなら初期値のまま利用するものようです。

V9958 では、21 ピンには VCC が割り当てられており、コンポジットビデオ出力自体が存在しませんので、これらレジスタも存在しないと思います。書き込み専用レジスタで、効果を成す出力ピンがありません。

図 2.4.1.23.1 にレジスタのビットマップを示します。

R#20: Color burst register 1									
bit	7	6	5	4	3	2	1	0	
	0	0	0	0	0	0	0	0	VDPのコンポジット出力の設定レジスタ。 MSXでは、VDPのコンポジット出力は使っていないことが多いので書き換えてはならない。
R#21: Color burst register 2									
bit	7	6	5	4	3	2	1	0	
	0	0	1	1	1	0	1	1	VDPのコンポジット出力の設定レジスタ。 MSXでは、VDPのコンポジット出力は使っていないことが多いので書き換えてはならない。
R#22: Color burst register 3									
bit	7	6	5	4	3	2	1	0	
	0	0	0	0	0	1	0	1	VDPのコンポジット出力の設定レジスタ。 MSXでは、VDPのコンポジット出力は使っていないことが多いので書き換えてはならない。

図 2.4.1.23.1. R#20～22 のビットマップ

詳しくは、3.2. RGB 出力とコンポジット出力も参照ください。

2.4.1.24. Display Offset Register (R#23)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

垂直スクロールレジスタです。0～255 のライン番号を指定します。

Pattern Name Table で作られる画面表示ですが、Pattern Name Table として指定されているアドレスを左上隅とする画面イメージを図 2.4.1.24-1. に示します。

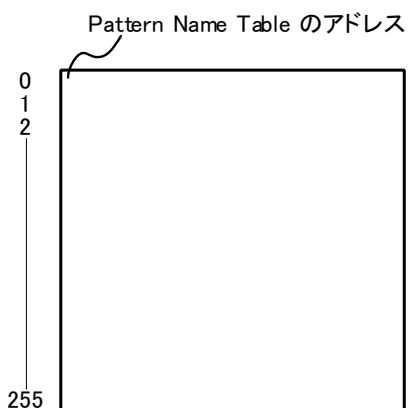


図 2.4.1.24-1. Pattern Name Table と画面の対応関係

デフォルトの状態では R#23 には 0 が設定されており、その状態では図 2.4.1.24-1. の左端に書いてある Y 座標のうち 0～191 または 0～211 の部分が画面に現れています。実際に画面に現れる部分は、 $R\#23 \sim ((R\#23+191) \text{ and } 255)$ または $R\#23 \sim ((R\#23+211) \text{ and } 255)$ となります。このように画面上の Y 座標に対して、Pattern Name Table アドレスからの Y 座標オフセットがいくつなのかを指定するレジスタであるため、Display Offset Register という名前になっています。

従って、R#23 をインクリメントすると、画像は上にシフトしていく動作になります。

通常、192～255 ラインまたは 212～255 ラインのところには、Pattern Name Table 以外の Sprite Attribute Table 等が配置されているので、それらがゴミのように見えます。R#23 を使って垂直スクロールを使う場合は、それらを見えない別のアドレスに変更しておくのが良いでしょう。

MSX-BASIC 3.0 以上では、SET SCROLL という命令が追加されており、これによる垂直スクロールは R#23 を制御します。MSX-BASIC 2.0 は V9938 搭載機であるにもかかわらずスクロール関連の命令は存在していません。垂直スクロールを BASIC で利用したい場合は VDP(24) が R#23 に対応しているので、ここに書き込むことになります。

MSX Documents VDP

VDP(24)に書き込むサンプル(R23VSCR.BAS)に示します。

```
100 DEFINT A-Z:SCREEN5:COLOR15,0,0:CLS
110 BLOAD"IMAGE1.SC5",S:COLOR=RESTORE
120 Y=0
130 VDP(24)=Y:Y=(Y+1)AND255:GOTO130
```

この実行結果を写真 2.4.24-1.に示します。

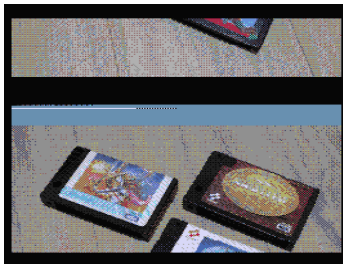


写真 2.4.24-1. R23VSCR.BAS の実行結果

2.4.1.25. N/A (R#24)

欠番です。R#24 は存在しません。

2.4.1.26. Mode Register 4 (R#25)

V9958 で新設されたレジスタです。TMS9918/V9938 にはこのレジスタは存在しません。

2.4.1.27. Horizontal Scroll Register High (R#26)

V9958 で新設されたレジスタです。TMS9918/V9938 にはこのレジスタは存在しません。

2.4.1.28. Horizontal Scroll Register Low (R#27)

V9958 で新設されたレジスタです。TMS9918/V9938 にはこのレジスタは存在しません。

2.4.1.29. VDP コマンド制御レジスタ (R#32～R#46)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

これらのレジスタについては、別途 2.6. VDP コマンドにて説明します。そちらを参照ください。

2.4.2. ステータスレジスタ

この章では、ステータスレジスタの個々の機能について説明します。

2.4.2.1. Status Register 0 (S#0)

xxx

2.4.2.2. Status Register 1 (S#1)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.3. Status Register 2 (S#2)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.4. Column Register Low (S#3)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.5. Column Register High (S#4)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.6. Row Register Low (S#5)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.7. Row Register High (S#6)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.8. Color Register (S#7)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.9. Border Register Low (S#8)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.4.2.10. Border Register High (S#9)

V9938 で新設されたレジスタです。TMS9918 にはこのレジスタは存在しません。

xxx

2.5. スプライト

TMS9918にはスプライト機能が搭載されています。スプライト機能とは、背景情報を破壊せずに、背景にオーバーラップして表示されるグラフィックのことです。ただし、背景に比べて大きな制限付きのグラフィックになります。

スプライトには2種類のモードがあり、TMS9918/V9938/V9958で共通に使えるモード1と、V9938/V9958から使えるようになったモード2の2種類になります。これ以降、それぞれスプライトモード1、スプライトモード2と表記します。

スプライトモード1とスプライトモード2の違いを下記の表 2.5.1. にまとめます。

表 2.5.1. スプライトモード1とスプライトモード2の差分

機能	スプライトモード1	スプライトモード2
水平に並べられる数	4枚	8枚
重ね合わせ機能(CCビット)	なし	あり
衝突判定無効化(ICビット)	なし	あり

スプライトモードは、画面モードに紐付いて決まります。各画面モードにおけるスプライトモードを表 2.5.2.にまとめます

表 2.5.2. 各画面モードにおけるスプライトモード

画面モード	スプライトモード
SCREEN0	スプライト無し
SCREEN1, 2, 3	スプライトモード1
SCREEN4, 5, 6, 7, 8, 10, 11, 12	スプライトモード2

VDPにおけるスプライトは、背景表示用の情報収集のためのVRAMアクセスが発生しない水平ブランキング期間中にVRAMから情報収集して、表示する情報を選別して内部レジスタに保持。映像期間中はその保持した情報に基づいて、背景より優先的に表示することで、背景にオーバーラップした形で表示されるグラフィックのことを指します。先に述べた「背景に比べて大きな制限」が付くのは、このような構造と、回路コストの都合から発生しています。

内部レジスタ(IC内部のSRAMか、FlipFlopで構成)は、非常に高速である反面、高価です。昨今のCPU等でL1キャッシュメモリが32KBとか64KB程度で、DRAMが8GBや16GBなどのように容量に落差があるのは、L1キャッシュがまさに高価なSRAMで構成されているからです。

MSX Documents VDP

容量がすくなく記憶素子と、限られた演算性能で実現するために、表示する情報を選別するわけですが、この選別が「優先度の上位4つ(8つ)」という選別なので、水平に5つ以上並べると5つ目が表示されない状況が発生します。

各水平ライン単位でこの選別は行われるため、少し垂直にずれて5つ並んでいないラインは表示されます。

ライン単位で処理するので、MSX の VDP に搭載されているスプライトは、一般的にはラインバッファ方式のスプライトと呼ばれているようです。

2.5.1. スプライトモード 1

SCREEN1, 2, 3 で利用できるスプライトのモードです。TMS9918/V9938/V9958 で互換性があります。

スプライトモード 1 には、2 種類のサイズモードがあります。8ドット×8ドットモードと、16ドット×16ドットモードの2種類です。さらに、2倍拡大表示の有無も選択できます。これは、R#1 の bit1, bit0 で選択します。

スプライトは、1画面に同時に 32 枚表示することが出来ます。これをスプライトプレーン#0～#31と呼ぶことにします。

表示したスプライトは水平に最大4枚並べることができ、5 枚目以降は表示されません。表示の優先順位は、若い番号の方が優先です。水平最大4枚のイメージを図 2.5.1.1.に示します。

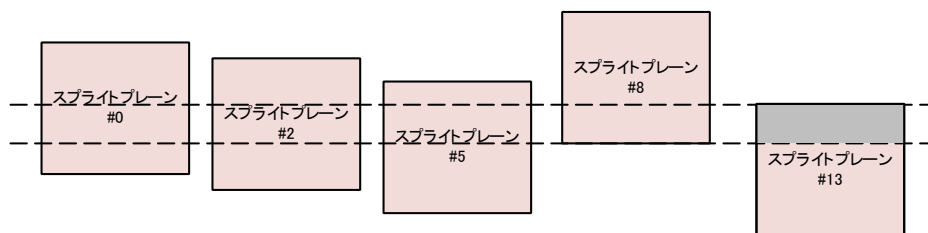


図 2.5.1.1. 水平最大4枚表示

ランダムに抽出したスプライトプレーン#0, #2, #5, #8, #13 を垂直方向に少しずらして、水平に並べて表示したイメージです。点線で挟まれたラインは、5枚並んでいます。このラインは、この5枚のプレーンのうち優先度の高い#0, #2, #5, #8 が表示され、それらよりも優先度の低い#13は表示されません。図ではグレーになっている部分が表示されません。

MSX Documents VDP

1枚のスプライトプレーンには、1つのアトリビュートが紐づけられています。アトリビュートとは、スプライトプレーンの表示情報を詰めた 4byte の属性テーブルで、VRAM 上の Sprite Attribute Table に格納されています。MSX-BASIC の PUT SPRITE 命令は、このアトリビュートを書き替えることで、スプライトの表示位置等を決めています。アトリビュートの構造を図 2.5.1.2. に示します。

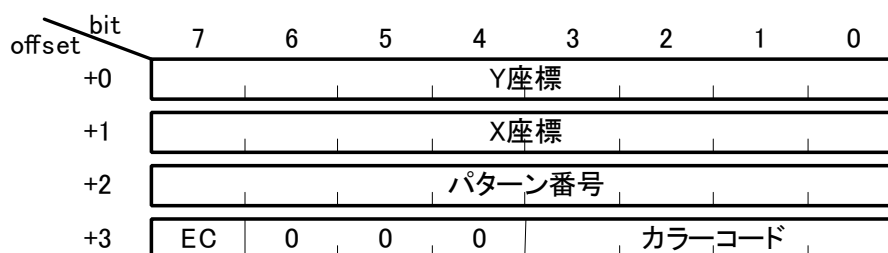


図 2.5.1.2. スプライトアトリビュート

Sprite Attribute Table の先頭 4byte はスプライトプレーン#0 のアトリビュート、次の 4byte はスプライトプレーン#1 のアトリビュート、、、最後の 4byte はスプライトプレーン#31 のアトリビュートとなっています。スプライトプレーン#n のアトリビュートの座標は下記のように求められます。

<Sprite Attribute Table の先頭アドレス> + <スプライトプレーン番号> * 4

アトリビュートは、最初の 2byte は座標ですが、Y 座標が先であることに注意してください。また、一番上端に表示するには Y = 0 ではなく Y = 255 にしなければなりません。回路構成の都合上、1 ラインずれています。

パターン番号とは、Sprite Generator Table で定義されるスプライト形状パターンの番号を指定する領域です。8 ドット x 8 ドットモードの場合は 0 ~ 255 の 256 通りが指定でき、16 ドット x 16 ドットモードの場合は、bit1, bit0 は無視され 4 の倍数の 64 通りを指定することが出来ます。

Sprite Generator Table は、8 ドット x 8 ドットの形状定義を 256 個定義することが出来ます。1byte で水平 8 ドットを表現しており、MSB 側が左・LSB 側が右に対応する 1bit=1 ドットの二値画像となっています。

スプライトアトリビュートの +4 にあるカラーコードは、Sprite Generator Table で 1 になっている bit に対応するドットの色を指定します。つまり1枚のスプライトは単色となります。かなり厳しい制約ではありますが、TMS9918 が登場した当初は、そのような制約があっても十分実用的でした。

この確認のためのサンプルプログラム(SC1SPR1.BAS)を下記に示します。

MSX Documents VDP

```
100 DEFINT A-Z: SCREEN1, 0: COLOR15, 4, 7
110 VPOKE &H3800, &B00111100
120 VPOKE &H3801, &B01111110
130 VPOKE &H3802, &B11111111
140 VPOKE &H3803, &B11111101
150 VPOKE &H3804, &B11111101
160 VPOKE &H3805, &B11111001
170 VPOKE &H3806, &B01100010
180 VPOKE &H3807, &B00111100
190 VPOKE &H1B00, 0
200 VPOKE &H1B01, 0
210 VPOKE &H1B02, 0
220 VPOKE &H1B03, 15
230 VDP(1) = VDP(1) AND &HFC: FOR I = 0 TO 1000: NEXT
240 VDP(1) = VDP(1) OR &H01: FOR I = 0 TO 1000: NEXT
250 GOTO 230
```

この実行結果を写真 2.5.1.1. に示します。



写真 2.5.1.1. SC1SPR1.BAS の実行結果

100 行目、SCREEN1 にして、スプライトを 8x8 モード・拡大無しで初期化。

110～180 行目は、Sprite Generator Table にビットパターンを書き込んで、パターン番号 0 の形状を定義。

190 行目、スプライトプレーン#0 の Y 座標に 0 を指定。

200 行目、スプライトプレーン#0 の X 座標に 0 を指定。

210 行目、スプライトプレーン#0 のパターン番号に 0 を指定。

220 行目、スプライトプレーン#0 の色をカラーコード 15 に指定。

230 行目、スプライトの「拡大無し」を設定して少し待機。

240 行目、スプライトの「拡大あり」を設定して少し待機。

MSX Documents VDP

250 行目、230 行目以降を反復する。

110～180 行目の記述は、SPRITE\$(0)を使っても実現できる内容です。

190～220 行目の記述は、SET SPRITE 0,(0,0),0,15 としても実現できる内容です。

230, 240 行目の記述は、SCREEN 命令の第2引数でも実現できる内容ですが、SCREEN 命令を実行するとその他の初期化処理まで実施してしまうので、途中で拡大有無を切り替えたい場合は、R#1 に直接アクセスしなければなりません。

このサンプルは、8ドット x8ドットモードを使うサンプルでした。

MSX Documents VDP

次に 16ドット x16ドットモードを使うサンプル(SC1SCR2.BAS)も提示しておきます。

```
100 DEFINT A-Z:SCREEN1,2:COLOR15,4,7
110 VPOKE&H3800,&B00000111
120 VPOKE&H3801,&B00011111
130 VPOKE&H3802,&B00111111
140 VPOKE&H3803,&B01111111
150 VPOKE&H3804,&B01111111
160 VPOKE&H3805,&B11111111
170 VPOKE&H3806,&B11111111
180 VPOKE&H3807,&B11111111
190 VPOKE&H3808,&B11111111
200 VPOKE&H3809,&B11111111
210 VPOKE&H380A,&B11111111
220 VPOKE&H380B,&B01111111
230 VPOKE&H380C,&B01111111
240 VPOKE&H380D,&B00111111
250 VPOKE&H380E,&B00011111
260 VPOKE&H380F,&B00000111
270 VPOKE&H3810,&B11100000
280 VPOKE&H3811,&B11111000
290 VPOKE&H3812,&B11111100
300 VPOKE&H3813,&B11111110
310 VPOKE&H3814,&B11111110
320 VPOKE&H3815,&B11111111
330 VPOKE&H3816,&B11111111
340 VPOKE&H3817,&B11111111
350 VPOKE&H3818,&B11111101
360 VPOKE&H3819,&B11111101
370 VPOKE&H381A,&B11111101
380 VPOKE&H381B,&B11111010
390 VPOKE&H381C,&B11111010
400 VPOKE&H381D,&B11100100
410 VPOKE&H381E,&B00011000
420 VPOKE&H381F,&B11100000
430 VPOKE&H1B00,0
440 VPOKE&H1B01,0
450 VPOKE&H1B02,0
460 VPOKE&H1B03,15
470 VDP(1)=VDP(1)AND&HFE:FORI=0TO1000:NEXT
480 VDP(1)=VDP(1)OR&H01:FORI=0TO1000:NEXT
490 GOTO470
```

2.5.2. スプライトモード 2

SCREEN4～12 で利用できるスプライトのモードです。V9938/V9958 で互換性があります。

スプライトモード 1 と異なるのは、下記 4 点です。

- (1) 水平最大 8 枚並べられる
- (2) スプライトプレーンにはライン単位で色を付けられる
- (3) ライン単位で重ね合わせの指定が出来る
- (4) ライン単位で衝突判定の有無を指定できる

それぞれ以下で個別に説明します。

(1) 水平最大 8 枚並べられる

スプライトは、1画面に同時に 32 枚表示することが出来ることは、スプライトモード 1 と同じです。これをスプライトプレーン#0～#31 と呼ぶことにします。

表示したスプライトは水平に最大 8 枚並べることができ、9 枚目以降は表示されません。表示の優先順位は、若い番号の方が優先です。水平最大 8 枚のイメージを図 2.5.2.1. に示します。

MSX Documents VDP

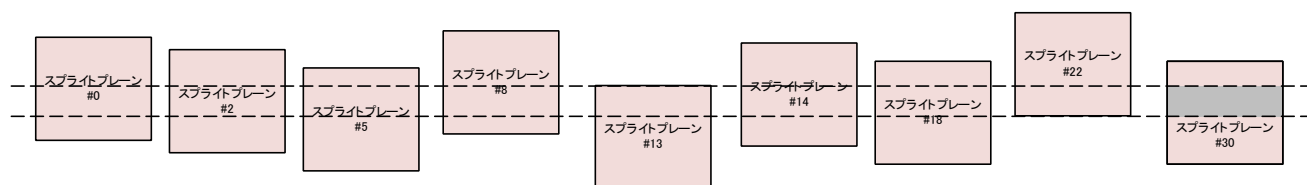


図 2.5.2.1. 水平最大 8 枚表示

水平に 9 枚の四角い形状のスプライトを表示しているイメージですが、グレーの部分が表示されません。スプライトプレーン#30 が中抜けのような状態で表示されます。

ピンク色の部分に形状定義があるかどうかは無関係です。直前のラインを表示しているときに、VDP 内部のスプライトバッファに表示するスプライトプレーン番号を格納していくわけですが、このときに Y 座標が現在の表示ラインに重なっているかどうかで判定しています。従ってスプライトの表示が 8x8, 16x16, 32x32 (16x16 の 2 倍拡大モード) のいずれかによって非表示になる範囲が変化します。スプライトパターン定義が 9 枚目の表示に影響しないことを確認できるサンプルプログラム (SP2OVMP.BAS) を用意しました。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5,2
110 VPOKE&H7800,&B00111100
120 VPOKE&H7801,&B01000010
130 VPOKE&H7802,&B10000001
140 VPOKE&H7803,&B10000001
150 VPOKE&H7804,&B10000001
160 VPOKE&H7805,&B10000001
170 VPOKE&H7806,&B01000010
180 VPOKE&H7807,&B00111100
190 FOR I=8 TO 31:VPOKE&H7800+I,0:NEXT I
200 PUT SPRITE0,(0,10),15,0
210 PUT SPRITE1,(8,10),15,0
220 PUT SPRITE2,(16,10),15,0
230 PUT SPRITE3,(24,10),15,0
240 PUT SPRITE4,(32,10),15,0
250 PUT SPRITE5,(40,10),15,0
260 PUT SPRITE6,(48,10),15,0
270 PUT SPRITE7,(56,10),15,0
280 Y=8:VY=1
290 PUT SPRITE8,(64,Y),15,0
300 Y=Y+VY
310 IF Y>32 OR Y<4 THEN VY=-VY
320 FOR I=0 TO 100:NEXT I:GOTO 290
```

順番に解説します。

MSX Documents VDP

100 行目：変数はデフォルトで整数型である宣言。前景色・背景色の指定。SCREEN5 へ変更して、スプライトを 16 ドット x16ドットの等倍サイズに指定。

110～190 行目：スプライトの形状定義です。図 2.5.2.2.のような形状を定義しています。

[illegible]

図 2.5.2.2. SP2OVMP.BAS のスプライト形状定義

スプライトパターン#0 に形状を指定します。16ドット x16ドットの場合、パターンジェネレーターテーブルは、1 ライン当たり 2byte、16line あるので全部で 32byte の領域を使って定義します。SCREEN5 のデフォルトのスプライトジェネレーターテーブルは、&H7800～ なので、スプライトパターン#0 の当該領域は、(&H7800 + 0 * 32)～(&H7800 + 0 * 32 + 31) の範囲となります。赤字の 0 は、スプライトパターン#0 の 0 です。

パターンジェネレーターテーブル上の 32byte のアドレスの並びは図 2.5.2.3.のようになっています。

				+0										+16				
				+1										+17				
				+2										+18				
				+3										+19				
				+4										+20				
				+5										+21				
				+6										+22				
				+7										+23				
				+8										+24				
				+9										+25				
				+10										+26				
				+11										+27				
				+12										+28				
				+13										+29				
				+14										+30				
				+15										+31				

図 2.5.2.3. スプライトパターン上のドットの位置とアドレスの位置関係

1byte 中の各ビット値がドットに対応しています。上位ビットが左、下位ビットが右になっているため、二進数表記するとそのまま見た目と一致した並びになります。0 のビットが透明、1 のビットが"ドット有り"を示しています。

したがって、110～180 行目は、スプライトの左上 8x8 の形状定義をダイレクトに VRAM へ書き込んでいることになります。

MSX Documents VDP

残りの +8～+31 の部分は、あえて「すべて透明」にしているので、値としては 0 です。190 行目の FOR 文でまとめて書き込んでいます。

200～270 行目：スプライトプレーン#0～#7 の 8 枚を水平に 8ドット間隔で並べています。

280～320 行目：スプライトプレーン#8 (9 枚目)をさらに右側に並べています。ただ並べると消えてしまうので、ゆっくり上下に動かしています。

実行イメージを写真 2.5.2.1.に示します。



写真 2.5.2.1. SP2OVMP.BAS の動作イメージ

表示しているすべてのスプライトプレーン (スプライトプレーン#0～#8) は、16ドット×16ドットの左上 8ドット×8ドットにしか形状定義が無く、右上・右下・左下は透明です。しかし、この形状定義が透明か否かは「水平最大 8 枚」のカウンタには影響せず、透明であってもカウントされます。そのため、9 枚目である一番右で動いているスプライトが消えている位置は、他のスプライトの 16x16 に重なるラインすべてで消えているのを確認できます。

右上・右下・左下にもパターンを入れて、かつ重ならないように位置を調節したサンプル(SP2OVMP2.BAS)を動かしてみたイメージを写真 2.5.2.2.に示します。



写真 2.5.2.2. SP2OVMP2.BAS の動作イメージ

MSX Documents VDP

9枚目が消えている部分は、他の8枚が並んでいるラインだけであることが確認できました。

SP2OVMP2.BASのソースリストは下記になります。SP2OVMP.BASからの改変部分を赤字にしてあります。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5,2
110 VPOKE&H7800,&B00111100
120 VPOKE&H7801,&B01000010
130 VPOKE&H7802,&B10000001
140 VPOKE&H7803,&B10000001
150 VPOKE&H7804,&B10000001
160 VPOKE&H7805,&B10000001
170 VPOKE&H7806,&B01000010
180 VPOKE&H7807,&B00111100
190 FOR I=8 TO 31:VPOKE&H7800+I,255:NEXT I
200 PUT SPRITE0,( 0,10),15,0
210 PUT SPRITE1,( 16,10),15,0
220 PUT SPRITE2,( 32,10),15,0
230 PUT SPRITE3,( 48,10),15,0
240 PUT SPRITE4,( 64,10),15,0
250 PUT SPRITE5,( 80,10),15,0
260 PUT SPRITE6,( 96,10),15,0
270 PUT SPRITE7,(112,10),15,0
280 Y=8:VY=1
290 PUT SPRITE8,(128, Y),15,0
300 Y=Y+VY
310 IF Y>32 OR Y<4 THEN VY=-VY
320 FOR I=0 TO 100:NEXT I:GOTO 290
```

さらに、図 2.5.2.1.のようにスプライトプレーンが中抜けするケースを再現するサンプル(SP2OVMP3.BAS)を動かしてみたイメージを写真 2.5.2.3.に示します。



写真 2.5.2.3. SP2OVMP3.BAS の実行イメージ

MSX Documents VDP

SP2OVMP3.BAS のソースリストは下記になります。SP2OVMP2.BAS からの改変部分を赤字にしてあります。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5,2
110 VPOKE&H7800,&B00111100
120 VPOKE&H7801,&B01000010
130 VPOKE&H7802,&B10000001
140 VPOKE&H7803,&B10000001
150 VPOKE&H7804,&B10000001
160 VPOKE&H7805,&B10000001
170 VPOKE&H7806,&B01000010
180 VPOKE&H7807,&B00111100
190 FOR I=8 TO 31:VPOKE&H7800+I,255:NEXT I
200 PUT SPRITE0,( 0,10),15,0
210 PUT SPRITE1,( 16,13),15,0
220 PUT SPRITE2,( 32,16),15,0
230 PUT SPRITE3,( 48, 8),15,0
240 PUT SPRITE4,( 64,19),15,0
250 PUT SPRITE5,( 80,13),15,0
260 PUT SPRITE6,( 96,15),15,0
270 PUT SPRITE7,(112, 6),15,0
280 Y=8:VY=1
290 PUT SPRITE8,(128, Y),15,0
300 Y=Y+VY
310 IF Y>32 OR Y<4 THEN VY=-VY
320 FOR I=0 TO 100:NEXT I:GOTO 290
```

スプライトの表示が水平 8 枚までなのは、VDP のハードウェア構造から発生する制約ですが、これを演出などにうまく使っているゲームソフトもあるようです。スプライトジェネレーターテーブルを書き替えてライン単位で消すと、同じ形状を選択している他のスプライトプレーンまで影響を受けますが、一部を消したいスプライトプレーンよりも若い番号のスプライトを並べておくと、その消したいスプライトプレーンだけを消すことが出来ます。

(2) スプライトプレーンにはライン単位で色を付けられる

スプライトモード 2 では、ライン単位に色を付けることが出来ます。

SCREEN2 や 4 の背景グラフィックに似たイメージですが、あちらは背景のパターンの方に色を付けましたが、スプライトモード 2 の方はスプライトプレーンの方に色を付けます。

ここで言葉の定義をおさらいしておきます。

MSX Documents VDP

スプライトパターン#0～#255

スプライトジェネレーターテーブルによって形状を定義できますが、この形状 1 つ 1 つの番号ですね。8ドット×8ドットモードの場合は #0～#255 の 256 種類、16ドット×16ドットモードの場合 #0～#63 の 64 種類のスプライトパターンを定義できます。

スプライトプレーン#0～#31

実際に表示されるスプライトのことを指します。32 枚まで同時に表示できますが、水平に並べられるのは 8 枚までです。MSX-BASIC の「PUT SPRITE <スプライトプレーン番号>, (<X 座標>, <Y 座標>), <カラーパレット番号>, <スプライトパターン番号>」では、指定のスプライトプレーンに紐つけるスプライトパターン番号を指定します。紐つけることによって、スプライトプレーンの表示形状が決まるわけです。

ライン単位の色指定の話に戻りますが、スプライトモード 2 では、スプライトプレーンごとに色を指定します。VRAM は図 2.5.2.4 のような構成になっています。スプライトプレーン 1 つあたり 16byte で、スプライトプレーン#0～#31 の色情報が連続して並んでいるイメージです。1byte の中の下位 4bit が色番号(カラーパレット番号)ですね。

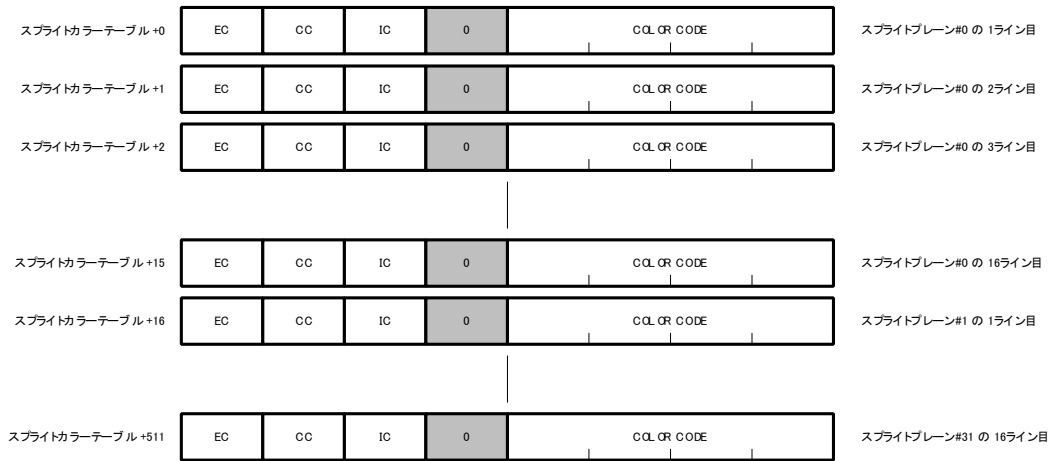


図 2.5.2.4. スプライトカラーテーブルの内容

従って、スプライトプレーン#0 と スプライトプレーン#1 の両方に、スプライトパターン#0 を指定していたとしても、スプライトプレーン#0 と スプライトプレーン#1 には別の色を付けることができます。

MSX Documents VDP

スプライトパターンの形状定義が二値画像になっていて色情報を含んでいないので、その二値の「ドットがある部分」の色を、別途指定します。貴重な VRAM の消費量とアクセス帯域を減らすための工夫ですね。

スプライトプレーン番号を SP, スプライトプレーン内の相対 Y 座標を PY, スプライトカラーテーブルの先頭アドレスを CA とすると、その PY に対応するラインの色を決める VRAM アドレス AD は下記の式で求められます。

$$AD = CA + SP * 16 + PY$$

注意すべきは、上記式の 16 は、8ドット×8ドットモードでも 16 である点です。1 スプライトプレーン当たり 16byte 割り当てられており、8ドット×8ドットモードでは、その若い番地の 8byte のみ使われ、残りの 8byte は無効になります。

16ドット×16ドットは、16byte 全部使って 16 ラインの色を指定します。

では、8ドット×8ドットモードで動作を見るサンプルプログラム(SP2COL1.BAS)を下記に示します。

```
100 DEFINT A-Z: COLOR 15, 0, 0: SCREEN 5, 1
110 VPOKE &H7800, &B00111100
120 VPOKE &H7801, &B01000010
130 VPOKE &H7802, &B10000001
140 VPOKE &H7803, &B10000001
150 VPOKE &H7804, &B10000001
160 VPOKE &H7805, &B10000001
170 VPOKE &H7806, &B01000010
180 VPOKE &H7807, &B00111100
190 PUTSPRITE 0, (10, 10), , 0
200 PUTSPRITE 1, (30, 10), , 0
210 VPOKE &H7400, &H0F
220 VPOKE &H7401, &H0E
230 VPOKE &H7402, &H0F
240 VPOKE &H7403, &H0E
250 VPOKE &H7404, &H0F
260 VPOKE &H7405, &H0E
270 VPOKE &H7406, &H0F
280 VPOKE &H7407, &H0E
290 VPOKE &H7408, &H02
300 VPOKE &H7409, &H03
310 VPOKE &H740A, &H02
320 VPOKE &H740B, &H03
```


MSX Documents VDP

```
330 VPOKE&H740C,&H02
340 VPOKE&H740D,&H03
350 VPOKE&H740E,&H02
360 VPOKE&H740F,&H03
370 VPOKE&H7410,&H04
380 VPOKE&H7411,&H05
390 VPOKE&H7412,&H04
400 VPOKE&H7413,&H05
410 VPOKE&H7414,&H04
420 VPOKE&H7415,&H05
430 VPOKE&H7416,&H04
440 VPOKE&H7417,&H05
450 VPOKE&H7418,&H04
460 VPOKE&H7419,&H05
470 VPOKE&H741A,&H04
480 VPOKE&H741B,&H05
490 VPOKE&H741C,&H04
500 VPOKE&H741D,&H05
510 VPOKE&H741E,&H04
520 VPOKE&H741F,&H05
530 GOTO 530
```

この動作イメージを写真 2.5.2.4. に示します。



写真 2.5.2.4. SP2COL1.BAS の実行イメージ

210～280 行目: 白・灰色・白・灰色・白・灰色・白・灰色をスプライトカラーテーブルの先頭 8byte に書き込んでいます。

290～360 行目: 緑・黄緑・緑・黄緑・緑・黄緑・緑・黄緑を次の 8byte に書き込んでいます。

370～520 行目: 青・明るい青・青・明るい青・青・明るい青・青・明るい青・青・明るい青・青・明るい青・青・明るい青・青・明るい青を次の 16byte に書き込んでいます。

表示されている丸いスプライトは、向かって左がスプライトプレーン#0、右がスプライトプレーン#1になります。

#0 は、210～280 行目で書き込んだ色になっています。

#1 は、290～360 行目で書き込んだ値ではなく、370～440 行目で書き込んだ色になっています。

これは、8ドット×8ドットモードであっても、スプライトカラーテーブルは 16byte で 1 プレーン分に相当することを示しています。

(3) ライン単位で重ね合わせの指定が出来る

スプライトモード 2 では、スプライトの重ね合わせによる合成が出来ます。重ね合わせの指定は、スプライトカラーテーブルに指定します。図 2.5.2.5. にスプライトカラーテーブルの 1byte の構成を示します。

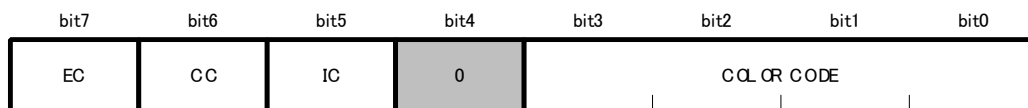


図 2.5.2.5. スプライトカラーテーブルの 1byte のビットアサイン

重ね合わせの指定は CC (bit6) になります。スプライトカラーテーブルなので、当然ながら 1 ライン単位で指定できます。

CC = 1 のスプライトラインは、"そのラインの COLOR CODE"と"1 つ優先度が高いスプライトラインの COLOR CODE"とを、ORしたカラーコードで表示されます。さらに、"1 つ優先度が高いスプライトラインの CC ビットが 1" だった場合は、さらに 1 つ優先度が高いスプライトラインと OR されます。このように何枚でも重ねられます。といっても、1 ラインに 8 枚までしか表示できないので最大 8 枚。カラーコードは 4bit しかないので、1,2,4,8 のカラーコードを持つ 4 枚のスプライトを重ねれば、全色表現できることを考えると、最大 4 枚重ねれば十分だと言えます。

優先度が最も高いスプライトプレーン#0 の CC ビットを 1 にしても、何もおきません。

MSX テクニカルハンドブックなどには、下記のように説明されています。

CC:CC ビットが“1”の場合、「このスプライトよりも優先順位が高く、かつ CC ビットが“0”で、最もこのスプライト面に近い」スプライトと等しい優先順位が得られる。等しい優先順位を持つスプライトが重なった場合には、その両者の色コードの OR(論理輪)をとったものが表示される。この場合、重なっても衝突は発生しない

MSX Documents VDP

「このスプライトよりも優先順位が高く」というのは「1 つ若い番号のスプライト」です。ただし、その「1 つ若い番号のスプライト」も CC=1 だった場合は、「さらに 1 つ若い番号のスプライト」となります。CC=0 かスプライトプレーン#0 になるまで辿っていくことになります。これをサラッと「このスプライトよりも優先順位が高く、かつ CC ビットが"0"」とかかれています。スプライトプレーン#1,#2,#3 の CC ビットがすべて 1 だと、スプライトプレーン#1, #2, #3 は、スプライトプレーン#0 と同じ優先順位となって、#0, #1, #2, #3 の OR をとった表示となります。

ここでいう優先順位は、表示の前後関係(他のスプライトプレーンより手前(見える)か、奥(隠れる)か)と衝突判定だけで、最大 8 枚表示の表示有無の優先順位には影響しないことに注意してください。CC = 1 にしたところで、最大 8 枚が増えるわけではありません。

CC=1 にして 2 枚重ねによる水平 3 色表示するサンプル(SP2OR1.BAS)を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5,3
110 FORI=0TO63:READD:VPOKE&H7800+I,D:NEXTI
120 FORI=0TO31:READD:VPOKE&H7400+I,D:NEXTI
130 FORI=0TO31:READD:VPOKE&H7680+I,D:NEXTI:COLOR=RESTORE
140 PUTSPRITE 0,(32,32),,0
150 'MAIN LOOP
160 Y=0:VY=1
170 PUTSPRITE1,(32,Y),,1
180 IFY=32THENFORI=0TO1000:NEXTI
190 Y=Y+VY:IFY<10RY>62THENVY=-VY
200 FORI=0TO100:NEXTI
210 GOTO170
220 'PATTERN DATA
230 DATA &H01,&H03,&H03,&H03,&H0F,&H03,&H0E,&H1F
240 DATA &H2F,&H5F,&HBF,&HBE,&HBF,&HBF,&HBB,&H23
250 DATA &H80,&HC0,&HC0,&HC0,&HF0,&HC0,&H70,&HF8
260 DATA &HF4,&HFA,&HFD,&H7D,&HFD,&HFD,&HDD,&HC4
270 DATA &H01,&H03,&H02,&H02,&H0E,&H0E,&H0F,&H1F
280 DATA &H3F,&H7F,&HFB,&HFB,&HEB,&HEB,&HE8,&H23
290 DATA &H80,&HC0,&H40,&H40,&H50,&H70,&HD0,&HD0
300 DATA &HDC,&HD6,&HD7,&HD7,&HD7,&HD7,&H17,&HC4
310 'COLOR DATA
320 DATA &H03,&H03,&H03,&H03,&H03,&H03,&H03,&H03
330 DATA &H03,&H03,&H03,&H03,&H03,&H03,&H03,&H03
340 DATA &H46,&H46,&H46,&H46,&H46,&H46,&H44,&H46
350 DATA &H46,&H46,&H46,&H46,&H46,&H46,&H46,&H46
360 'PALETTE DATA
370 DATA &H00,&H03,&H70,&H07,&H53,&H05,&H11,&H01
380 DATA &H30,&H00,&H33,&H03,&H17,&H03,&H66,&H06
390 DATA &H54,&H06,&H06,&H04,&H77,&H07,&H21,&H02
400 DATA &H00,&H00,&H43,&H04,&H65,&H06,&H54,&H05
```

使用しているスプライトパターンのデータイメージを図 2.5.2.6.に示します。

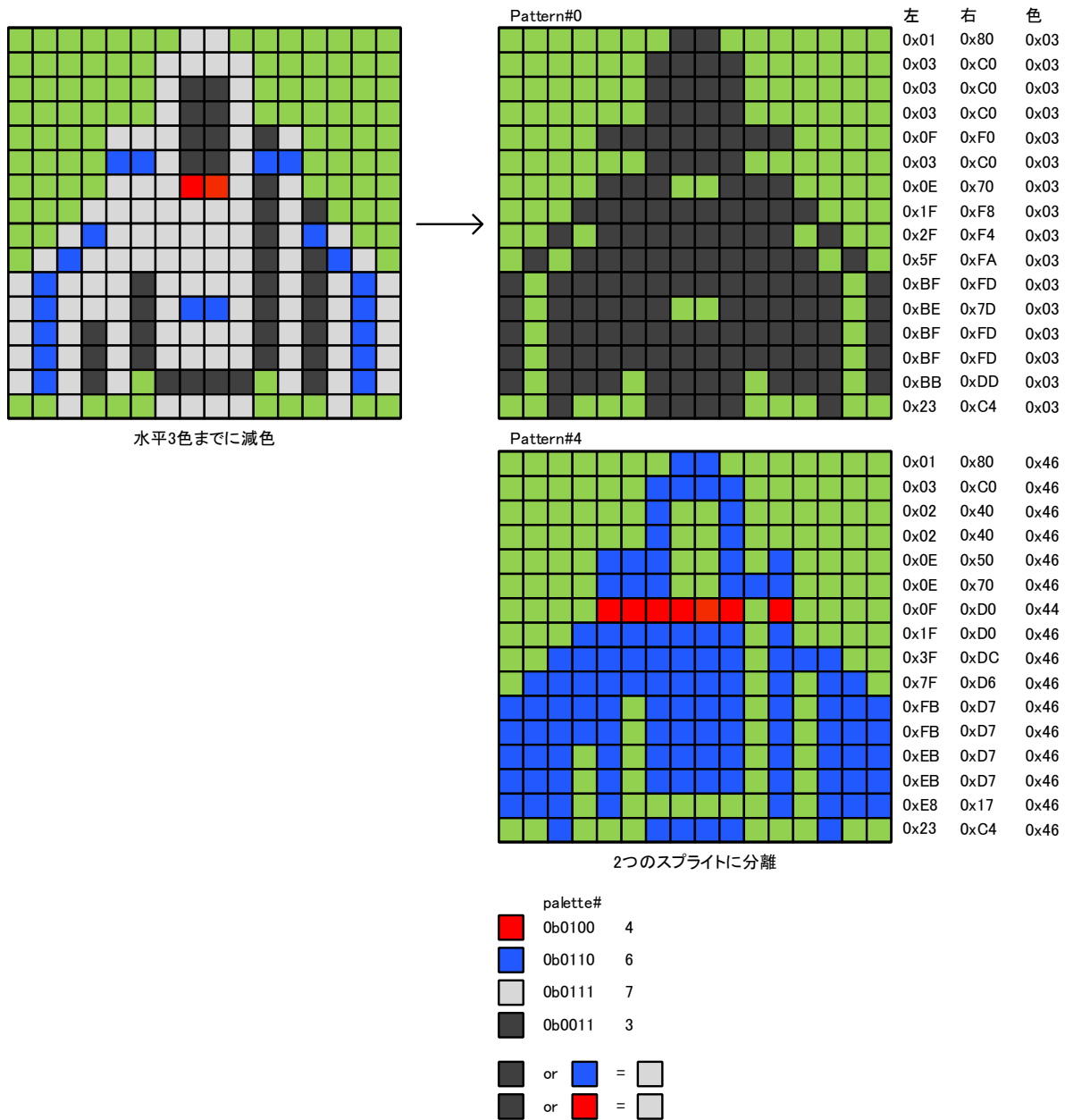


図 2.5.2.6. SP1OR1.BAS で使用しているパターンデータ

水平 3 色使えるといっても、3 色目は OR をとった値なので、なかなかイメージが難しいと感じる人も居ると思いますが、パラーパレットで色は自由に入れ替えられることを考えれば、あとは「多くのキャラクタで使う色を 1 や 2 や 4 や 8 に割り当てる」という方法で、割とやりやすくなります。

例えば、カラーパレット#1 を黒にしておきます。偶数番のカラーパレットに明るい色、奇数番のカラーパレットに暗い色（例えば、2 は明るい緑・3 は暗い緑）としておきます。スプライトパターン#0 は カラーパレット#1 を、スプライトパター

MSX Documents VDP

ン#1 はカラーパレット#2 を指定すると、OR をとったときに カラーパレット#3 が使えるので、使いやすくなります。これはライン単位で個別に指定できるので、うまく使えば 2 枚重ねでも結構豪華な表示が出来るのです。

CC=1 にしたラインは、親となる CC=0 のスプライトと同じラインでしか表示できません。SP2OR1.BAS でスプライトプレーン#1 の方を上下にずらしていますが、上下はスプライトプレーン#0 の表示されている範囲でしか表示されていないのが確認できます。ただし、ずれて「重ならない部分」の CC を 0 にしておけば表示されるので、一部分だけ重ね合わせて縦長のキャラクタを作る、といったことは可能です。

(4) ライン単位で衝突判定の有無を指定できる

2.6. VDP コマンド

V9938/V9958 にはビットマップを扱うグラフィックモードが追加になっていますが、これを CPU からの VRAM 読み書きのみで更新すると、現実的な処理速度では実現できません。そのような問題が発生しないように、VDP コマンドという機能が追加になっています。VDP コマンドは、端的に言えばよく使う画像処理を CPU の代わりに VDP が請け負ってくれる機能です。VDP が VDP コマンドを実行中も CPU は別の演算を実施できるのが強みです。また、VDP コマンド実行中にも VRAM 読み書きは可能です。

VDP コマンドには表 2.6.1.に示す種類があります。

表 2.6.1-1. VDP コマンドの種類

コマンド名	転送先	転送元	転送単位	ニーモニック	R#46の上位4bit			
High speed move	VRAM	CPU	1byte	HMMC	1	1	1	1
	VRAM	VRAM	1byte	YMMM	1	1	1	0
	VRAM	VRAM	1byte	HMMM	1	1	0	1
	VRAM	VDPLレジスタ	1byte	HMMV	1	1	0	0
Logical move	VRAM	CPU	1dot	LMMC	1	0	1	1
	CPU	VRAM	1dot	LMCM	1	0	1	0
	VRAM	VRAM	1dot	LMMM	1	0	0	1
	VRAM	VDPLレジスタ	1dot	LMMV	1	0	0	0
Line	VRAM	VDPLレジスタ	1dot	LINE	0	1	1	1
Search	VDPLレジスタ	VRAM	1dot	SRCH	0	1	1	0
Pset	VRAM	VDPLレジスタ	1dot	PSET	0	1	0	1
Point	VDPLレジスタ	VRAM	1dot	POINT	0	1	0	0
未使用	-	-	-	-	0	0	1	1
	-	-	-	-	0	0	1	0
	-	-	-	-	0	0	0	1
Stop	-	-	-	STOP	0	0	0	0

Hogh speed moveとLogical moveは、画像のブロック転送になります。画像上の矩形領域を別の場所へ複写する機能ですね。High speed moveの方は、Byte単位の転送で、後述のロジカルオペレーションは使えません。Logical moveの方は、dot単位の転送で、ロジカルオペレーションを利用可能です。

Lineは直線描画。Searchは指定色の探索。Psetは点描画。Pointは指定位置の色取得。Stopは稼働中のVDPコマンドを停止。LineとPsetはロジカルオペレーションを利用可能です。

ロジカルオペレーションとは、描画先に元々描かれている色と描画色とを所定の演算で合成して新しい画素値とする論理演算操作のことです。演算内容として選択可能なものの一覧を表2.6.1-2.にまとめます。

表 2.6.1-2. ロジカルオペレーション一覧

ロジカルオペレーション	演算内容	R#46の下位4bit			
IMP	DC = SC	0	0	0	0
AND	DC = SC and DC	0	0	0	1
OR	DC = SC or DC	0	0	1	0
EOR	DC = SC xor DC	0	0	1	1
NOT	DC = not SC	0	1	0	0
-		0	1	0	1
-		0	1	1	0
-		0	1	1	1
TIMP	if SC = 0 then DC = DC else DC = SC	1	0	0	0
TAND	if SC = 0 then DC = DC else DC = SC and DC	1	0	0	1
TOR	if SC = 0 then DC = DC else DC = SC or DC	1	0	1	0
TEOR	if SC = 0 then DC = DC else DC = SC xor DC	1	0	1	1
TNOT	if SC = 0 then DC = DC else DC = not SC	1	1	0	0
-		1	1	0	1
-		1	1	1	0
-		1	1	1	1

表 2.6.1-2.の演算内容で SC は転送元のカラーコード (Source Color Code)、DC は転送先のカラーコード (Destination Color Code) です。ロジカルオペレーションが T で始まるものは、転送元が 0 の場合は何もせず、0 以外の場合だけ演算を実施します。つまり、0 を透明扱いして、0 以外の部分だけ転送するわけですね。

High speed move では、ロジカルオペレーションには対応していません。これは、Logical move が、SC と DC を VRAM から読み出してからロジカルオペレーション演算を実施して DC へ書き込みを行うため 1 ドットあたり 3 回の VRAM アクセスを要します。一方で、High speed move では、SC を読み出して DC へ書き込むため 2 回のアクセスに削減されます、それに加えてドット単位ではなく Byte 単位のため、SCREEN5～7 のように 1Byte に複数のドットを含む画面モードでは、それだけ Logical move よりも高速に処理されます。

VDP コマンドでは、画像処理になりますので画面座標を意識した位置指定となります。このとき、MSX-BASIC における SET PAGE の Page 番号の若い画面から順に、垂直方向に並んだような状態で結合された縦長の座標空間となります。例えば、SCREEN5 では図 2.6.1-1.に示す座標空間となります。

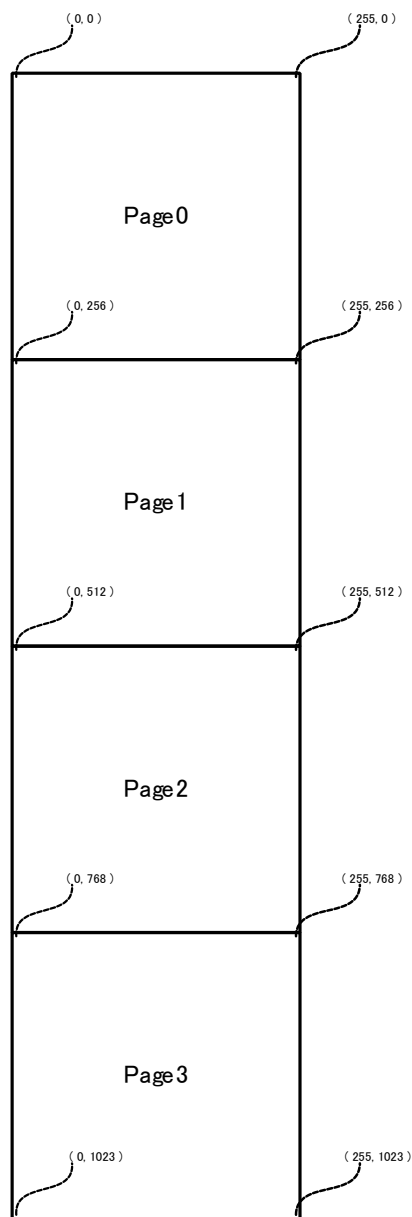


図 2.6.1-1. SCREEN5 における VDP コマンドの座標空間

Y 座標値が 10bit あって、上位 2bit が Page 番号に対応すると解釈すると近いですが、VDP コマンドにとって Page の境界に何ら障壁がない点が異なります。MSX-BASIC では BASIC インタプリタが自動的に 1 画面内で収まるように座標をクリッピングしてしましますが、VDP コマンドとしては、LINE(0,0)-(255,1023),15 のような直線描画も行えるのです。

また、画面表示は Y 座標は 0～211 の範囲となりますが、1Page あたり 0～255 の Y 座標空間を持っています。212～255 の非表示の領域には、Sprite Attribute Table や Sprite Pattern Generator Table 等が配置されているた

MSX Documents VDP

め、それらを VDP コマンドで塗りつぶすようなことも可能になっています。実際、R#23 を使った垂直スクロールを行うと、この非表示の領域が画面上に見える位置へ移動してきます。縦スクロールゲーム等でこの位置にも背景を描画する必要があるので、VDP コマンドがこの普段非表示の領域に描画可能なのも意味があるわけです。

VDP コマンドは、コントロールレジスタ R#32～R#46 を使うことでコマンドパラメータ設定とコマンド実行開始指示を出します。R#32～R#46 の各ビットの意味を図 2.6.1-2. に示します。

MSX Documents VDP

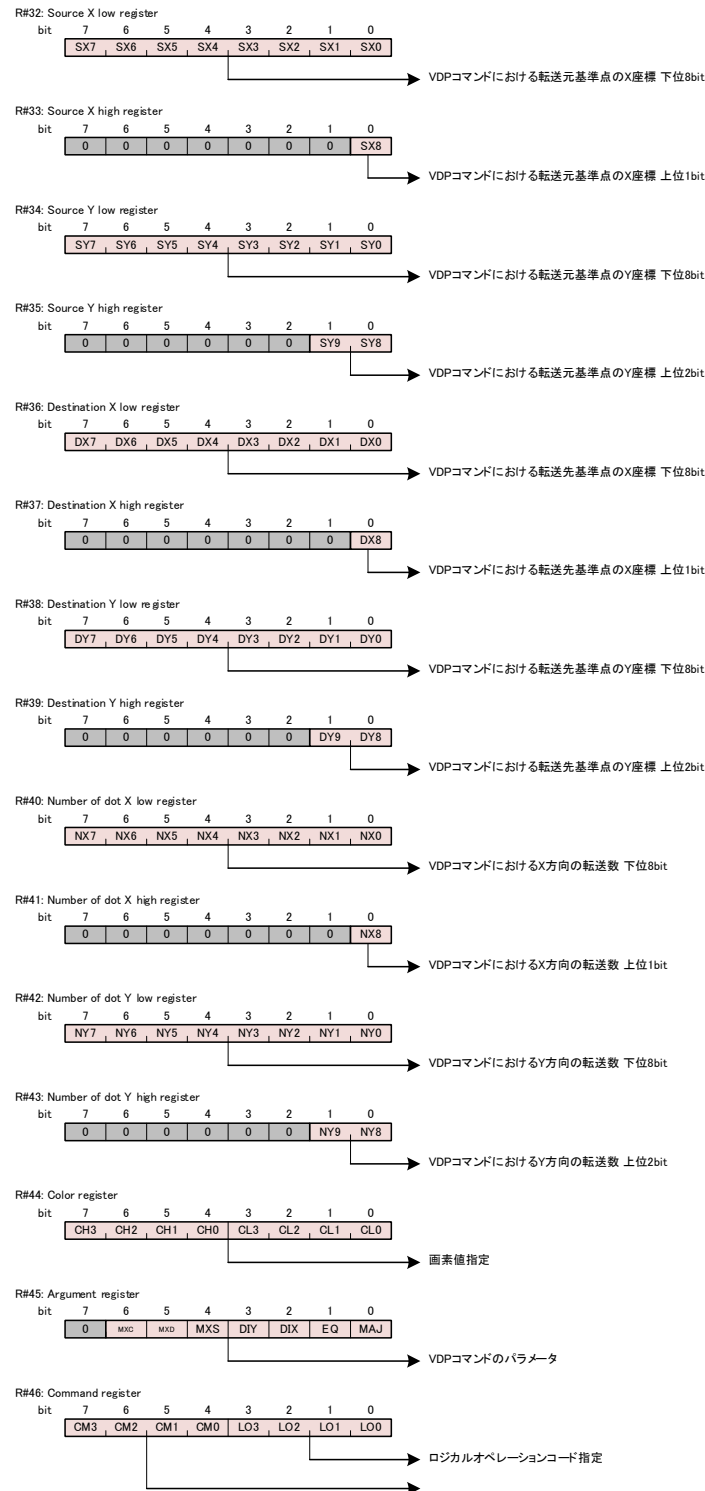


図 2.6.1-2. R#32～R#46 のビットマップ

では、各 VDP コマンドについて個別に説明していきます。

2.6.1. HMMC (High-speed Move CPU to VRAM)

高速に CPU から VRAM へ書き込むコマンドです。

VDP の転送完了チェックをしながら 1byte ずつ書き込むため、直接 VRAM へ書き込むより遅くなりますが、バイトアラインに沿った矩形領域への書き込みが可能な点が、アドレスインクリメントにしか対応していない直接 VRAM 書き込みよりもインテリジェントになっています。

挙動としては、CPU から VDP レジスタに書き込む内容を、(DX, DY)-STEP(NX,NY) に描画します。転送先ページは D Y に含まれています。

バイトアライン制約があるため、SCREEN5 及び SCREEN7 では X 座標(DX)と X 幅(NX)は偶数、SCREEN6 では4の倍数である必要があります。SCREEN8,10～12 は 1ドット = 1 バイトに対応しているため、任意のドット位置を指定できます。

動作のイメージを図 2.6.1-1.に示します。

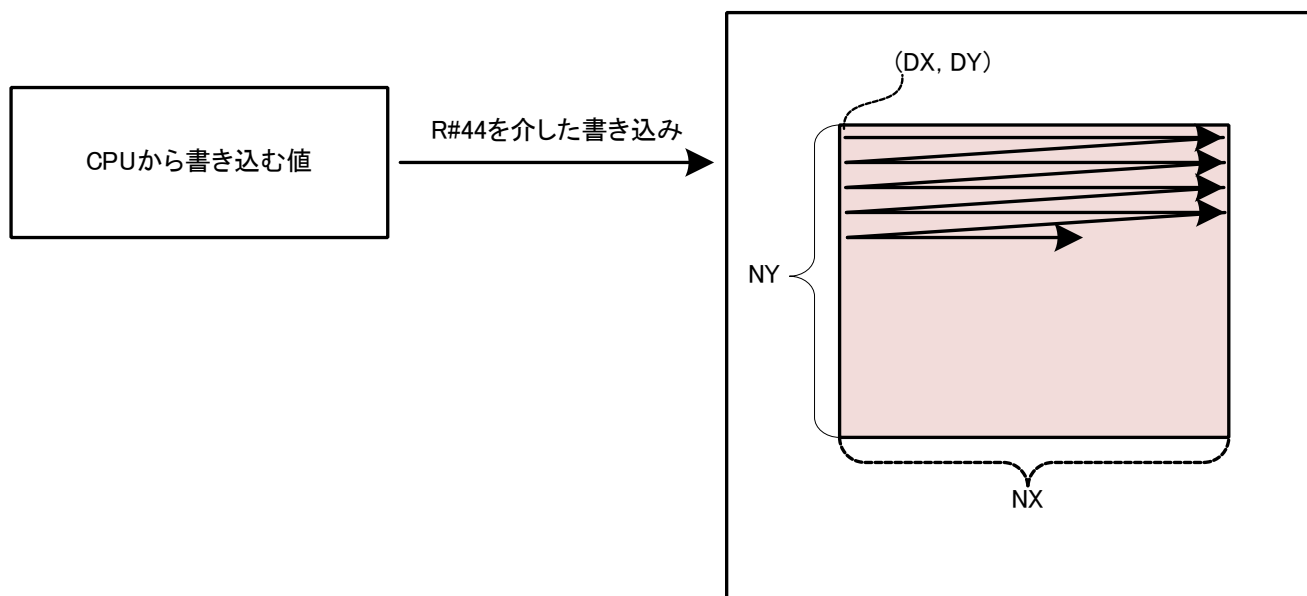


図 2.6.1-1. HMMC 動作イメージ

図 2.6.1-1.の左の四角(CPU から書き込む値)は、計算で求めても良いし、CPU のメモリ上にデータとして持っても良い画像データです。右の大きい太枠は VRAM 全体だと思ってください。この VRAM 上の書き込み位置を指定するのが (DX, DY) で、座標で指定します。座標としてはドット単位の値になりますが、DX はバイトアライン制約があります。たとえば SCREEN5 なら最下位ビットは無視されます。次に、(NX, NY) で幅と高さをドット単位で指定します。NX は DX と同じバイトアライン制約があります。

MSX Documents VDP

R#32～R#35 は、HMMC では未使用です。R#36・R#37 には DX を、R#38・R#39 には DY を、R#40・R#41 には NX を、R#42・R#43 には NY を書き込みます。

R#44 には CPU から転送する最初の 1byte を書き込んでおく必要があります。

R#45 には細かい動作指定を書き込みます。

MXC(bit6) は、HMMC では未使用です。

MXD(bit5) は、転送先メモリの選択を指定します。0 は VRAM, 1 は裏 VRAM。通常は 0 を指定。

MXS(bit4) は、HMMC では未使用です。

DIY(bit3) は、垂直描画方向を指定します。(DX,DY)から下方向へ描画していく場合 0、上方向へ描画していく場合 1。

DIX(bit2) は、水平描画方向を指定します。(DX,DY)から右方向へ描画していく場合 0、左方向へ描画していく場合 1。

EQ(bit1) は、HMMC では未使用です。

MAJ(bit0) は、HMMC では未使用です。

図 2.6.1-1.では、(DX,DY) を始点として、右方向・下方向へ描画していくイメージになっていますが、DIX, DIY の指定によって、左方向や上方向を選択することが出来ます。

R#46 には、CM(bit7～bit4)に HMMC を示す 1111 を書き込むことで HMMC を発動できます。ロジカルオペレーション指定 LO(bit3～bit0)の設定値は無視されます。

HMMC コマンドは、CPU から VRAM への転送になりますので、複数の byte 値で構成される画像データを流し込む必要があります。それには R#44 を使いますが、単純に連続して書き込んではいけません。VDP が次の byte 値を受け取れるようになったか確認しながら書き込む必要があります。1byte 目はコマンド発行時に R#44 に設定していた値が使われ、2byte 目以降は図 2.6.1-2. のシーケンスに則って設定する必要があります。

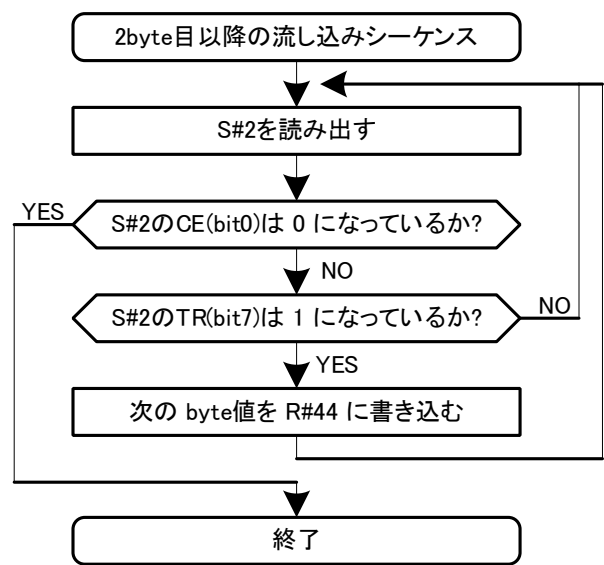


図 2.6.1-2. 2 バイト目以降の流し込みシーケンス

例えば、SCREEN5 で 8ドット×8ドットの画像を描画してみましょう。描画する画像は図 2.6.1-3.の画像にします。ドットの中に記載の数字はカラーコード(16 進数)です。

4	4	4	1	1	4	4	4
4	4	1	2	C	1	4	4
4	1	3	3	2	C	1	4
1	2	3	3	2	C	C	1
1	C	2	2	2	C	C	1
4	1	C	C	C	C	1	4
4	4	1	C	C	1	4	4
4	4	4	1	1	4	4	4

図 2.6.1-3. 描画したい画像

これを 1byte 単位に左上から右下まで、水平方向を優先的に順番にオフセット番号を振ったものを図 2.6.1-4.に示します。

4+0 4	4+1 1	1+2 4	4+3 4
4+4 4	1+5 2	0+6 1	4+7 4
4+8 1	3+9 3	2+10 0	1+11 4
1+12 2	3+13 3	2+14 0	0+15 1
1+16 0	2+17 2	2+18 0	0+19 1
4+20 1	0+21 0	0+22 0	1+23 4
4+24 4	1+25 0	0+26 1	4+27 4
4+28 4	4+29 1	1+30 4	4+31 4

図 2.6.1-4. 描画したい画像の byte 単位につけたオフセット番号

例えば、オフセット番号 +0 の位置は、左画素=4, 右画素=4 なので、転送する byte 値としては 44h になります。

オフセット番号 +1 の位置は、左画素=4, 右画素=1 なので、転送する byte 値としては 41h になります。

このように順番にコード化すると、下記ようになります。

44h, 41h, 14h, 44h, 44h, 12h, C1h, 44h, 41h, 33h, 2Ch, 14h, 12h, 33h, 2Ch, C1h
1Ch, 22h, 2Ch, C1h, 41h, CCh, CCh, 14h, 44h, 1Ch, C1h, 44h, 44h, 41h, 14h, 44h

R#44 に書き込む byte 値はこの値を順次書き込むことになります。これを HMMC コマンドにより (100,100)を左上とする位置に描画することにします。

では、サンプルプログラム(VCHMMC1.ASM, VCHMMC1.BAS)を下記に示します。

まずは、VCHMMC1.BAS の方から。

```
100 CLEAR200,&HBFFF:COLOR15,4,7:SCREEN5
110 BLOAD"VCHMMC1.BIN",R
120 GOTO120
```

100 行目でメモリ確保と SCREEN5 に画面を初期化する。

110 行目で VCHMMC1.ASM のアセンブル結果である VCHMMC1.BIN を読み込んで実行。

120 行目で待機。

次に、VCHMMC1.ASM を少しずつ分解しながら説明します。

MSX Documents VDP

```
VDP_IO_PORT1      = 0x99
VDP_IO_PORT2      = 0x9A
VDP_IO_PORT3      = 0x9B
```

まず、冒頭の VDP_IO_PORT* の定数定義ですが、VDP にアクセスするための I/O ポート番号に名前を付けています。ご覧の通り、99h, 9Ah, 9Bh に決め打ちになっているので、MSX バージョンアップアダプターには非対応のサンプルとなっています。通常の MSX2 と、バージョンアップアダプターによる MSX2 の両方に対応させるには、MAIN-ROM の 0006h 番地, 0007h 番地 の内容を読む必要があります。興味のある方は両方に対応できるようにサンプルを修正してみてください。

```
; BSAVE header
db          0xFE
dw          start_address
dw          end_address
dw          start_address
```

次の db, dw ですが、db は Define Byte。dw は Define Word。データを配置する疑似命令です。

BLOAD でロードされることを期待したプログラムなので、BLOAD 用の 7byte のヘッダになります。

```
; Program body
org          0xC000

start_address::
di
ld          c, VDP_IO_PORT1

; R#17 = 36 (R#36 からの間接連続書き込み設定)
ld          a, 36
out         [c], a
ld          a, 17 | 0x80
out         [c], a

; R#36~R#46 に dx~cmr をまとめて書き込む
ld          bc, (11 << 8) | VDP_IO_PORT3; R#36~R#46 は 11 個のレジスタ
ld          hl, dx
otir                     ; この命令の終了時点で HL = target_image になっている
```

ここからがプログラム本体です。org 疑似命令で、これ以降 C000h 番地からに配置される想定でアドレス計算するようにアセンブラに指示しています。なので、start_address は C000h になります。

MSX Documents VDP

途中で割り込まれるとマズい処理が多数続くので DI して割込禁止にしています。

HMMC コマンドを実行するために、R#36～R#46 の 11 個のレジスタに連続書き込みするので、R#17 を用いた間接的なコントロールレジスタへの連続書き込み(2.3.1.6. 間接的なコントロールレジスタへの書き込み(インクリメントあり))を利用します。そのために R#17 へ 36 を書き込んでいます。

間接的なコントロールレジスタへの連続書き込みは、port#3 に書き込むので B レジスタには 11 個、C レジスタには VDP_IO_PORT3 をまとめて設定するために LD BC, (11 << 8) | VDP_IO_PORT3 としています。

書き込む内容は、ソースの下の方にある dx というラベルからになるので、HL に dx を設定して OTIR を実行し、まとめて書き込んでいます。

```
ld      c, VDP_IO_PORT1

; 2 バイト目以降の書き込みシーケンス
; R#17 = 44 | 0x80 (R#44 への連続書き込み)
ld      a, 44 | 0x80
out     [c], a
ld      a, 17 | 0x80
out     [c], a
```

ここからが 2 バイト目以降の書き込みシーケンスになります。

2 バイト目以降の書き込みは R#44 に対して連続的行われるので、R#17 を使った間接的なコントロールレジスタへの連続書き込み(2.3.1.7. 間接的なコントロールレジスタへの書き込み(インクリメントなし))を利用します。今回はインクリメント無しですね。

これで Port#3 への書き込みは R#44 への書き込みになります。

```
; R#15 = 2 (S#2 を読むための設定)
ld      a, 2
out     [c], a
ld      a, 15 | 0x80
out     [c], a
```

次に、2 バイト目以降はステータスレジスタ S#2 の CE ビット(bit0)と TR ビット(bit7)を確認しながら R#44 へ書き込まねばなりません。そのため、ステータスレジスタ選択レジスタである R#15 に 2 を設定しておきます。

これで Port#1 からの読み出しは S#2 の読み出しとなります。

MSX Documents VDP

```
wait_tr_flag:
    in          a, [VDP_IO_PORT1]
    rrca                          ; Cy = CE bit
    jr          nc, exit_hmmc_loop
    and         a, 0x40           ; Zf = TR bit
    jr          z, wait_tr_flag
    outi                          ; 次のデータを書き込み
    jr          wait_tr_flag
```

まず、in 命令で Port#1 を読んでいますが、これは先ほど書いたように S#2 の読み出しになります。CE ビットと TR ビットをチェックすることになりますが、まず先に CE ビットをチェックします。bit0 なので RRCA で右 1bit シフトして bit0 を Cy フラグに入れています。

CE ビットの名前は、Command Execution Flag の略です。VDP コマンド実行中は 1 になり、停止中は 0 になります。最後の Byte 値を転送し終えた後は 0 になり、TR ビットは無効になるため、CE ビットを先にチェックするのが妥当です。0 が停止中、つまり「HMMC コマンドが終わったこと」を示すので、JR NC, exit_hmmc_loop で、ループから抜けます。

HMMC コマンドが動作中であれば、まだ全部のデータを転送し終わっていないので、次のデータを R#44 へ書き込んで良いか TR ビットで確認します。

TR ビットの名前は、Transfer Ready Flag の略です。HMMC 等の VDP と通信しながら処理を進めていくためのハンドシェイク用のビットですね。0 は、まだ転送処理中。1 は、転送完了したから次送って OK。という意味です。

TR ビットは bit7 に存在しますが、さきほど CE ビット確認用に RRCA して右 1 ビットシフトしているので、A レジスタの内容としては bit6 の位置に移動しています。bit6 だけを抽出するために AND 40h しています。

0 の場合はまだ転送中なので、JR Z, WAIT_TR_FLAG で S#2 の読み出しからやり直しにしています。

1 の場合は、次の byte 値を書き込んで良いので OUTI しています。

HL レジスタは、先ほど OTIR した時に target_image を指す位置で停止しています。target_image は 2 バイト目以降の byte 値を格納している領域です。C レジスタは、このループに入る前に VDP_IO_PORT3 にしています。そのため、Port#3 への書き込みは R#44 になるようにさきほど設定していますので、HL が指す target_image の 1 バイトを R#44 に書き込み、さらに HL を次へ遷移させなさいという意味になります。

全部のデータを転送し終わると、先ほどの CE ビットのところで説明したように、JR NC, exit_hmmc_loop で、ループから抜けます。

```
exit_hmmc_loop:
    ; R#15 = 0 (S#0 を読むための設定に戻す)
```

MSX Documents VDP

```
xor      a, a
out      [c], a
ld       a, 15 | 0x80
out      [c], a
ei
ret
```

この時点で HMMC コマンドの実行は終わって、画面にも画像が現れているのですが、R#15 = 2 のまま割込許可(EI)してしまうと、BIOS の割り込み処理ルーチンが暴走するので、そうならないように EI する前に R#15 = 0 に戻すのを忘れないでください。

```
dx:
    dw      100          ; R#36, R#37
dy:
    dw      100          ; R#38, R#39
nx:
    dw      8            ; R#40, R#41
ny:
    dw      8            ; R#42, R#43
clr:
    db      0x44         ; R#44
arg:
    db      0            ; R#45
cmr:
    db      0b1111_0000  ; R#46

target_image:
    db      0x41, 0x14, 0x44, 0x44, 0x12, 0xC1, 0x44
    db      0x41, 0x33, 0x2C, 0x14, 0x12, 0x33, 0x2C, 0xC1
    db      0x1C, 0x22, 0x2C, 0xC1, 0x41, 0xCC, 0xCC, 0x14
    db      0x44, 0x1C, 0xC1, 0x44, 0x44, 0x41, 0x14, 0x44
end_address::
```

ここはデータ列です。dx~cmr の dw, db は、OTIR でまとめて書き込むレジスタ設定値ですね。右側にコメントで対応するレジスタ番号を記入しておきました。

target_image は、OUTI で書き込む2バイト目以降のバイト値です。(1バイト目は clr に記載の 0x44)

今回の転送サイズは 8ドット×8ドットと小さかったので、全体を割込禁止にしました。

MSX Documents VDP

しかし、画面全体を転送しつつ BGM 演奏するとか、ディスクから読み出しながら転送といった場合は、全体を割込禁止にすることはできません。(前者は BGM の演奏に使われる H.TIMI が期待する間隔で入らず音が間延びしたり、後者はそもそもディスクアクセスルーチンが EI してしまったりする)

そういった場合は、R#44 への書き込みアクセスは単発のアクセスと同じ 2.3.1.1. コントロールレジスタライトを利用し、R#15 も S#2 を読み出したらずぐに 0 に戻して EI する待機ルーチンにすることになります。

TRビット待ちは、それほど長い時間ではないので、例えば 64 回までは割込禁止で処理するとか、工夫の余地はあると思います。

上記サンプルの実行結果を写真 2.6.1-1. に示します。

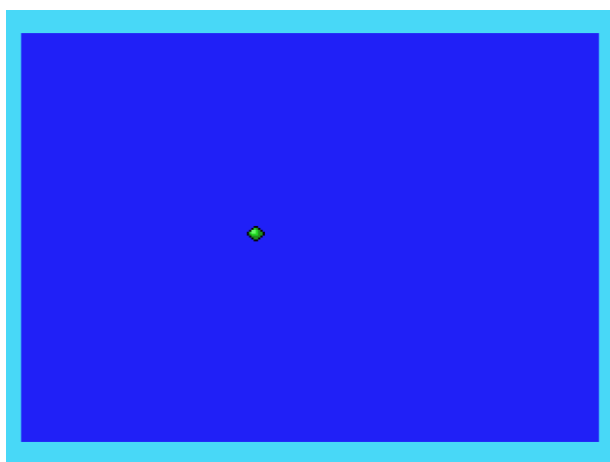


写真 2.6.1-1. VCHMMC1.BAS の実行結果

VCHMMC1.ASM は、DIX 及び DIY は 0 の場合のみでした。これを (DIX, DIY) = (0,0), (1,0), (0,1), (1,1) の 4 通りについてすべて試してみるサンプル VCHMMC2 を用意しました。R#44 に流し込む byte 値の順序は同じままで、DIX, DIY を違う値にするとどうなるか確認するサンプルになります。プログラムコードの中身は VCHMMC1 とほぼ同等なので説明は省略します。実行結果を写真 2.6.1-2. に示します。

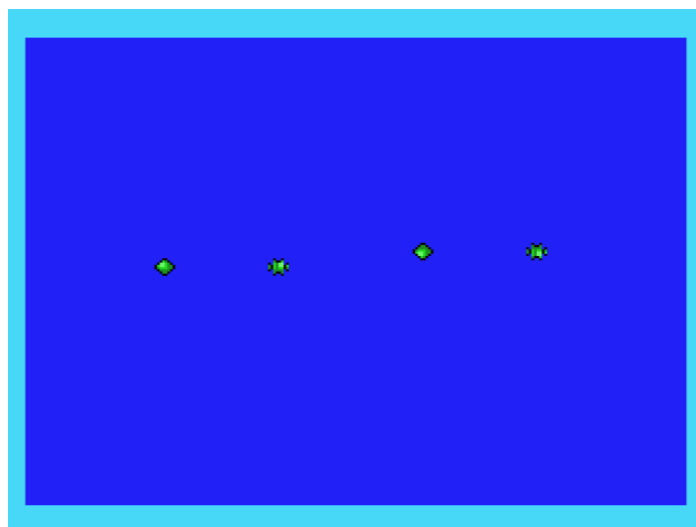


写真 2.6.1-2. VCHMMC2.BAS の実行結果

4つのパーツが描画されています。左から $(DIX, DIY) = (0,0), (1,0), (0,1), (1,1)$ の順になります。

描画先座標 (DX, DY) は、左から $(50,100), (100,100), (150,100), (200,100)$ になります。

$(DIX, DIY) = (0,0)$ の場合、R#44 の最初の値である 44h は、 $(50, 100)$ と $(51, 100)$ のペアに書き込まれます。

次の値は $(52, 100), (53,100)$ のペア、と順次右($DIX=0$)へ進み、 $(56,100), (57,100)$ のペアを描いた時点で下($DIY=0$)へ進み $(50,101), (51,101)$ のペアへ。といった具合に左から右へ、上から下への方向の順序で描画されていきます。

$(DIX, DIY) = (1,0)$ の場合、R#44 の最初の値である 44h は、 $(100,100), (101,100)$ のペアに書き込まれます。

次の値は $(98, 100), (99,100)$ のペア、と順次左($DIX=1$)へ進み、 $(94, 100), (95, 100)$ のペアを描いた時点で下($DIY=0$)へ進み $(50,101), (51,101)$ のペアへ。といった具合に右から左へ、上から下への方向の順序で描画されていきます。

$(DIX, DIY) = (0,1)$ の場合、R#44 の最初の値である 44h は、 $(150,100), (151,100)$ のペアに書き込まれます。

次の値は $(152, 100), (153,100)$ のペア、と順次右($DIX=0$)へ進み、 $(156, 100), (157, 100)$ のペアを描いた時点で上($DIY=1$)へ進み $(150,99), (151,99)$ のペアへ。といった具合に左から右へ、下から上への方向の順序で描画されていきます。

$(DIX, DIY) = (1,1)$ の場合、R#44 の最初の値である 44h は、 $(200,100), (201,100)$ のペアに書き込まれます。

MSX Documents VDP

次の値は (198, 100), (199,100) のペア、と順次左(DIX=1)へ進み、(194, 100), (195, 100) のペアを描いた時点で上(DIY=1)へ進み (200,99), (201,99) のペアへ。といった具合に右から左へ、下から上へ方向の順序で描画されていきます。

用意した画像は、(DIX, DIY) の順序を想定した画像であるため、乱れたり、逆さまになったりしています。

2.6.2. YMMM (High-speed Move VRAM to VRAM, y only)

高速に矩形領域を別の位置へコピーするコマンドです。

ただし、移動先は X 座標が同じである必要があり、かつ転送開始座標から画面の右端まで(または左端まで)まとめてコピーする用途にしか使えません。制約が多い反面、ブロック転送系コマンドでは最速となっています。

転送のイメージを図 2.6.2.1. に示します。

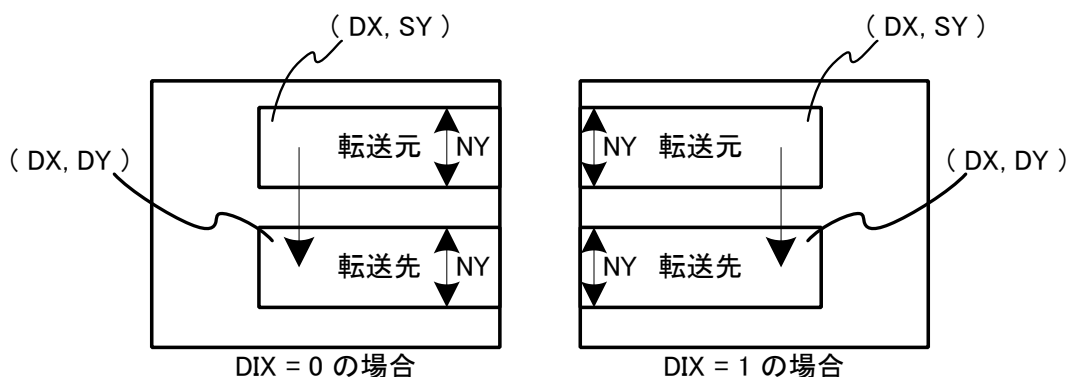


図 2.6.2.1. YMMM コマンド実行イメージ

転送元開始座標は、DX を使いますので SX は使いません。

転送サイズは画面の右端または左端までとなりますので NX は使いません。

ブロック転送なので CLR (R#44) も使いません。

R#45 は、MXD(bit5: 転送先メモリ選択), DIY(bit3: 転送方向 Y), DIX(bit2: 転送方向 X) のみ有効です。

R#46 の LO (bit3~bit0)は使いません。ロジカルオペレーションは使用できません。

DX 及び NX は、バイトアラインにのっている必要があります。例えば、SCREEN5 では 1byte に 2 画素詰められているため、DX および NX は偶数のみ指定可能で、奇数を指定しても bit0 が無視されます。

以下にサンプルプログラム(VCYMMM1.BAS)を示します。

まず動作イメージを説明します。初期化時に page1 に画像を読み込み、図 2.6.2.2. の状態にします。画面の左側を垂直スクロールさせる動作となります。R#23 による垂直スクロールと違って画像を書き替えているので重くなりますが、それでも YMMM コマンドが高速なので、MSX には速いと感じるかもしれません。このとき、R#23 と違って右側は動きません。それを分かりやすくするために、page1 に読み込んだ画像を page0 にもコピーしておきます。

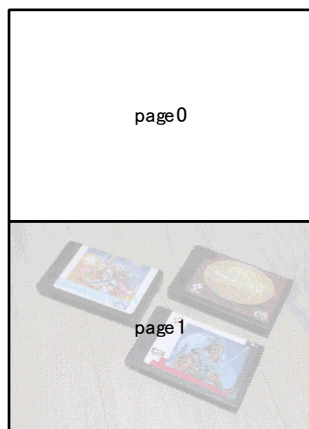


図 2.6.2.2. VCYMMM1 の初期化時

page1 の左側を 変数 top_line で上下に分断します。top_line は、0～211 を循環する数値で、8 ずつ減る数値です
0, 204, 196, ... , 4, 208, 200, 192, ... , 8, 0 といった具合で変化させます。上下に分断するイメージを図 2.6.2.3.に示
します。

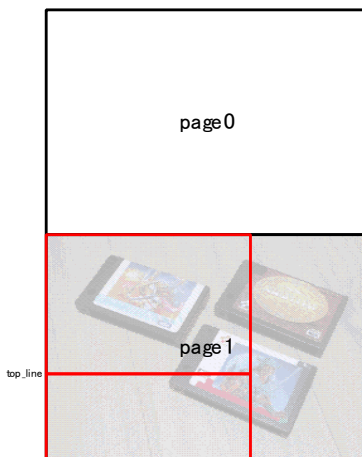


図 2.6.2.3. page1 の左側を top_line で上下に分断

この分断した "page1 の左側" の "下部" を、page0 の左上に転送するのに YMMM コマンドを利用します。
次に、もう一度 "page1 の左側" の "上部" を、page0 の左下に転送するのに YMMM コマンドを利用します。
ただし、例外的に top_line = 0 の時は上下に分断されないで1回の YMMM コマンドで転送を終えることにします。
このイメージを図 2.6.2.4.に示します。

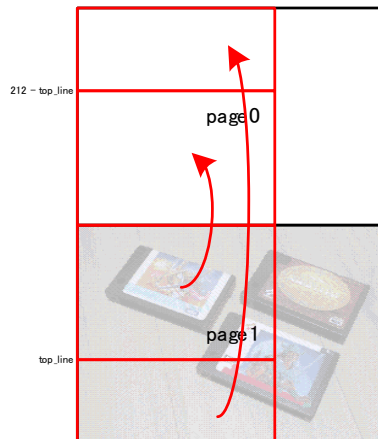


図 2.6.2.4. YMMM コマンドによる転送元と転送先

では、アセンブラコードの解説に移りたいと思います。

```
VDP_IO_PORT1    = 0x99
VDP_IO_PORT2    = 0x9A
VDP_IO_PORT3    = 0x9B
```

```
; BSAVE header
db          0xFE
dw          start_address
dw          end_address
dw          start_address
```

この部分は、VDP の I/O ポート定義と、BSAVE ヘッダの定義部分です。BSAVE ヘッダをアセンブラに書き出させることによって、MSX-BASIC から BLOAD 出来るようにしています。

```
start_address::
```

```
main_loop:
; 上部の転送 COPY( 160, top_line )-step(-160, 212-top_line ),1 to ( 160, 0 ),0
xor        a, a
ld         [dy], a
ld         a, [top_line]
ld         [sy], a
sub        a, 212
neg
ld         [ny], a
call       exec_ymmm
```

この部分は、2分割したブロックのうち、転送先(page0)で上側に来る矩形ブロックの転送処理になります。

この部分が着目している画像上の領域のイメージを図 2.6.2.5. に示します。

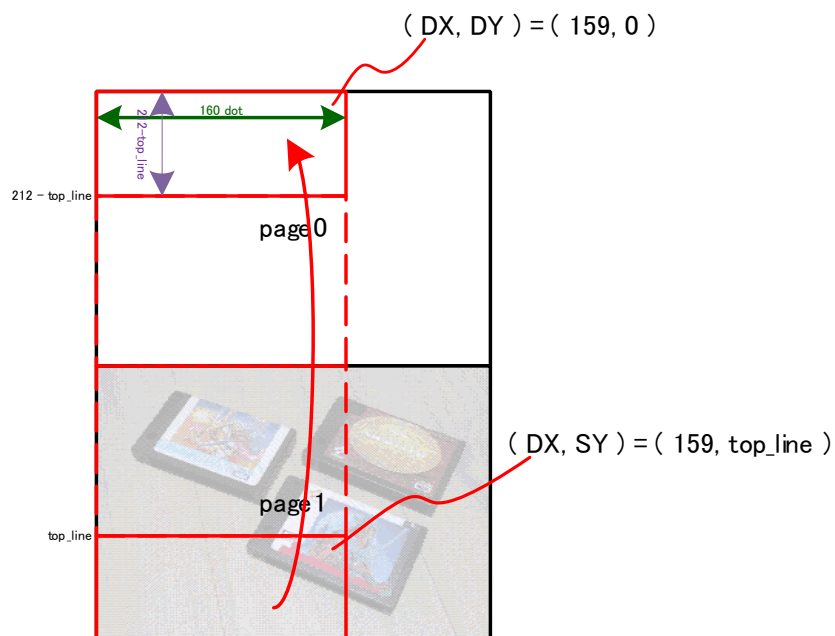


図 2.6.2.5. 上部の転送領域の座標

画面中央付近から左端までの矩形領域を転送するので、DIX = 1 にします。DIX = 1 で DIY = 0 の場合、基準は矩形領域の右上になることに注意してください。右から左へ順次転送していく処理になります。

ちなみに DIX = 1 で DIY = 1 にすると基準は矩形領域の右下になります。

【note】

DIX, DIY は転送方向を指定することになります。VDP は 1 単位 (高速転送なら 1byte, 論理転送なら 1dot) で処理します。YMMM の場合、DIX によって右端までなのか左端までなのかという選択の意味合いが強いですが、VDP のブロック転送系コマンドでは基準点 (SX,SY) からの走査順序を指定することになります。

転送の順序を指定する意味はあるのか? という疑問がわくかもしれませんが、転送元と転送先が重なっている場合に、転送順序によって大きな差異が生じます。ABCD の 4 文字を 左から B の位置にコピーすることを考えてみてください。A を B の位置に書き込むと AACD になります。次に B だった位置をコピーしようとしても、そこにはすでに A が書き込まれているので、それを読み取って C に上書きして AAAD になります。これを繰り返して AAAAA になります。

Z80 の LDIR と LDDR の関係と似ていますね。

あえて左から右へ走査させ、右側が重なっている状態にすることで同じグラフィックパーツで敷き詰めることが出来ます。HMMC 等で書き込むのは最初的小ブロックだけで済み、1 回の HMMM で大量に複製することが出来るわけです。それも VDP が処理してくれるので、その間 Z80 は別のことを実施できるわけですね。全部 HMMC で書いたり、HMMM をたくさん実行したりすると、Z80 の負荷が高くなってしまうので、DIX・DIY をうまく使うことで高速化に貢献できるわけです。これは MSX-BASIC の COPY 文でも同様に実行できます。覚えておいて損はないですね。

まず図 2.6.2.5.を見てもうと、転送先の (DX, DY) は (159, 0) という固定値になります。DX も DY も 2byte の領域ですが、今回は SCREEN5 なので X 座標は 0~255 の範囲であるため DX の上位 byte は 0 固定で使います。Y 座

MSX Documents VDP

標は 1page は垂直 256ドットのため、DY の2バイトの領域のうち下位 byte はページ内の Y 座標、上位 byte はページ番号と見なすことができます。転送先は、かならず page0 なので、DY の上位 byte は 0 固定。

次に転送元の (SX, SY) ですが、YMMM コマンドでは例外的に (DX, SY) となります。DX は先ほどの固定値 159。SY は、top_line の値そのものとなります。SY の上位 byte は page1 を示す 1 固定ですね。

転送サイズを示す NY は、212 - top_line となるので、その計算をしています。SY に top_line の値を格納するために Aレジスタに top_line の値を取り込んでいるので、これを再利用するために 212-top_line を -(top_line-212) と変形して、Aレジスタから 212 引いてから neg で符号反転して NY に格納しています。

exec_ymmm は、YMMM コマンドを実行するルーチンです。これについては後ほど説明します。

```
        ; 下部の転送 COPY( 160, 0 )-step(-160, top_line ),1 to ( 160, 212-top_line ),0
        ld      a, [top_line]
        or      a, a
        jr      z, lower_is_not_transfer
        ld      b, a
        ld      a, [ny]
        ld      [dy], a
        ld      a, b
        ld      [ny], a
        xor     a, a
        ld      [sy], a
        call    exec_ymmm
lower_is_not_transfer:
```

ここはもう一つのブロックの転送になります。top_line = 0 のときは2つに分断されていないので、2つ目の転送は不要。コメントを除く最初の3行はその判定処理になっています。

図 2.6.2.6.に2ブロック目の転送イメージを示します。

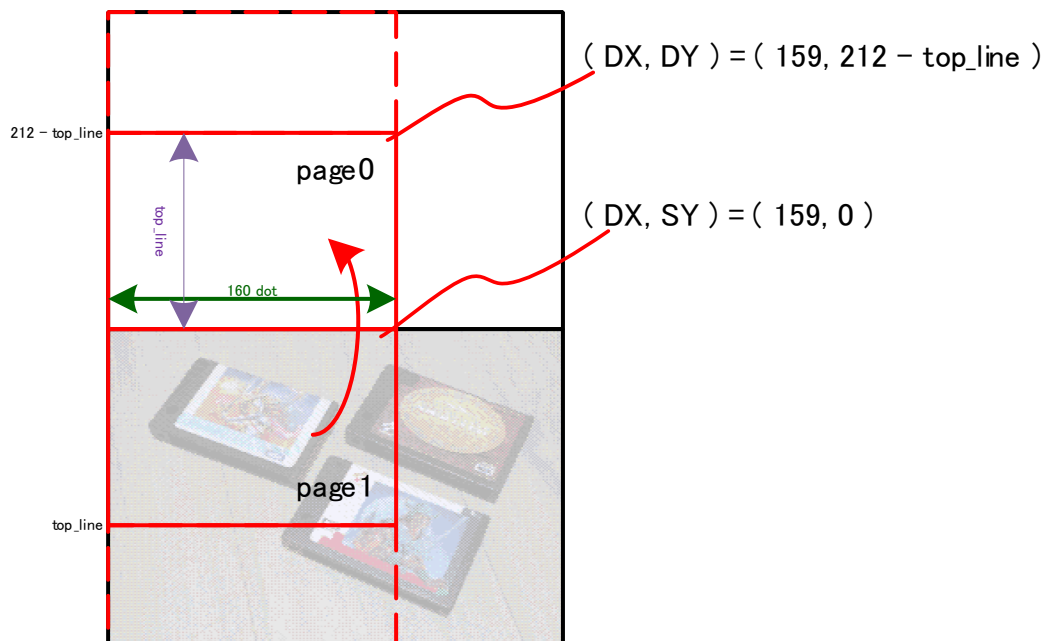


図 2.6.2.6. 下部の転送領域の座標

転送に使われる値は、上部の転送と同じ値なので、上部の転送に使った値の格納先を入れ替えることで対応しています。

```

; top_line をシフトする
ld      a, [top_line]
sub     a, 8
jp      nc, shift_end
add     a, 212
shift_end:
ld      [top_line], a

jp      main_loop

top_line:
db      0

```

ここは、変数 top_line の値を 8 ずつ減らしていく処理を実施しています。減らした後に負の値になったら 212 を加算して正の値に戻しています。これによりこの部分を実行するたびに 8 ずつ減って、いずれ 212 付近に戻るようになっています。

```
exec_ymmm::
```

MSX Documents VDP

```
        ; 転送完了待ち
        ld      c, VDP_IO_PORT1
        ld      de, 15 | 0x80      ; d = 0, e = 15 | 0x80
wait_ce_flag:
        ; R#15 = 2 (S#2 を読むための設定)
        di
        ld      a, 2
        out     [c], a
        out     [c], e
        ; a = S#2
        in      a, [c]
        ; R#15 = 0
        out     [c], d
        out     [c], e
        ei
        rrca    ; Cy = CE bit
        jr      c, wait_ce_flag
```

ここでは、VDP コマンドが実行中であれば、終了するまで待つ処理を記述しています。

YMMM コマンド実行前に終了待ち!?!と疑問に思うかもしれませんが、VDP コマンド開始指示から終了までの間は、(HMMC や LMMC のような CPU からの手番を含むコマンドを除き、)VDP が時間を掛けて処理する期間になります。

その間、CPU は別の処理を実施した方が良いので、VDP コマンド開始指示をしてから他の処理ルーチンへ戻り、次の VDP コマンド開始指示をしたいタイミングの直前で終了判定をして、終了していなければ終了するまで待つことで処理効率を上げています。

VDP コマンドを実行していなければ、S#2 の CE ビットは 0 になっているので、このルーチンはスルッと抜けてきます。

次に、DI と EI の関係ですが、HMMC のサンプルとは異なり、ここでは VDP コマンド完了待ちの間、一瞬割込許可期間を作るようにしています。VDP コマンドによる大きな領域のブロック転送は数十 msec かかることもあります。そのため、その間ずっと割込禁止にしてしまうのは好ましくありません。割り込み処理を邪魔しないために割込許可期間を含めています。割込許可を言うことは、BIOS が「垂直帰線割込か?の判定処理」のために S#0 を読みます。それは R#15 = 0 の設定を含まず、Port#1 の読み出しが S#0 の読み出しであると決め打ったコードになっています。それが誤って S#2 を読んでしまわないように、EI 前に R#15 = 0 を挟んでいます。

```
        ; R#17 = 34 (R#34 からの間接連続書き込み設定)
        ld      a, 34
        di
        out     [c], a
        ld      a, 17 | 0x80
```

MSX Documents VDP

```
out      [c], a
ei
```

VDP コマンドである YMMM を実行するに当たり、R#17 を使った間接的なレジスタ連続書き込みを利用しています。最初のレジスタ番号は R#34 なので、R#17 = 34 としています。これを直接的なレジスタアクセスの機能を使って書き込んでいます。

```
    ; R#34 ~ R#46 に sy ~ cmr をまとめて書き込む
    ld      bc, (13 << 8) | VDP_IO_PORT3; R#34 ~ R#46 は 13 個のレジスタ
    ld      hl, sy
    otir
    ret
```

BIOS の割り込み処理ルーチンは、R#17 や、Port#3 へのアクセスはしないため、ここは割込禁止にする必要はありません。

間接的なレジスタアクセスによって R#34 ~ R#46 へ書き込みをします。R#46 への書き込みは VDP コマンド処理開始のため、この OTIR 実行完了とともに YMMM コマンドが始動します。

```
sy:
    db      0          ; R#34
    db      1          ; R#35 常に転送元は page1 なのであらかじめ 1 を書いておく。
dx:
    dw      158        ; R#36, R#37
dy:
    db      0          ; R#38
    db      0          ; R#39 常に転送先は page0 なのであらかじめ 0 を書いておく。
nx:
    dw      0          ; R#40, R#41 無効だが OTIR で書き込むためのダミー
ny:
    dw      0          ; R#42, R#43
clr:
    db      0          ; R#44 無効だが OTIR で書き込むためのダミー
arg:
    db      0b0000_0100 ; R#45 DIY = 0, DIX = 1
cmr:
    db      0b1110_0000 ; R#46
end_address::
```

MSX Documents VDP

ここは、データ領域ですね。

DX の値は、先ほど 159 と書きましたが、ここで設定されているのは 158 です。

実は、X 座標が 158 と 159 の 2 ドットがペアになって VRAM 上の 1byte を構成しています。そのため、0～159 の 160 ドットを転送したいので 159 にしても良いのですが、SCREEN5 では byte アライン制約によって最下位ビットは無視されます。そのため、158 と 159 は同じ意味になります。

2.6.3. HMMM (High-speed move VRAM to VRAM)

高速に矩形領域を別の位置へコピーするコマンドです。最もよく使うコマンドだと思います。

転送のイメージを図 2.6.3.1. に示します。

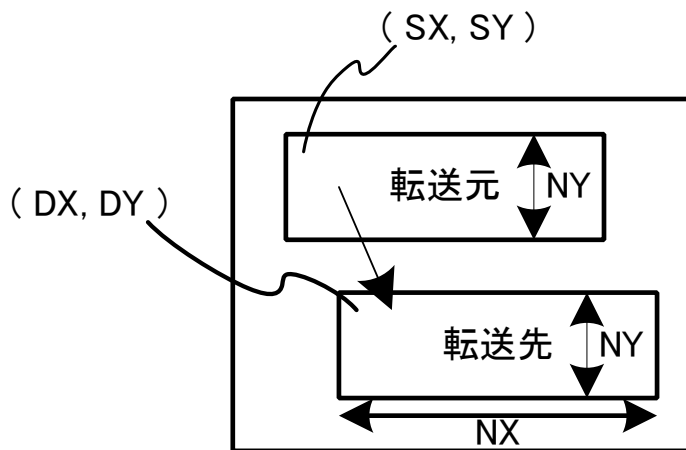


図 2.6.3.1. HMMM コマンド実行イメージ

MSX-BASIC の $\text{COPY}(\text{SX}, \text{SY})\text{-STEP}(\text{NX}, \text{NY}) \text{ TO } (\text{DX}, \text{DY})$ に相当します。ただし、byte 単位高速転送なので SX, NX, DX は byte アライン制約がかかります。SCREEN5 と 7 なら 2 の倍数、SCREEN6 なら 4 の倍数、SCREEN8 以上は制約無し。

また、HMMM はロジカルオペレーションは利用できません。ロジカルオペレーションは、転送先との論理演算を実施するために、転送先の値を VRAM 一度から読み取って、書き込む値と論理演算してから書き戻すことになります。そのため、「転送先の値を読み取る分」だけ VRAM のアクセス帯域を消費します。このロジカルオペレーションの処理を省略することで高速に動けることになります。論理演算を必要としない場合は、LMMM よりも HMMM を使った方が圧倒的に高速となります。

1byte ずつ転送するため、転送元と転送先が重なっている場合には、DIX と DIY による転送方向指定が大きな意味を持てきます。同じ領域転送の DIX が異なる例を図 2.6.3.2. に示します。

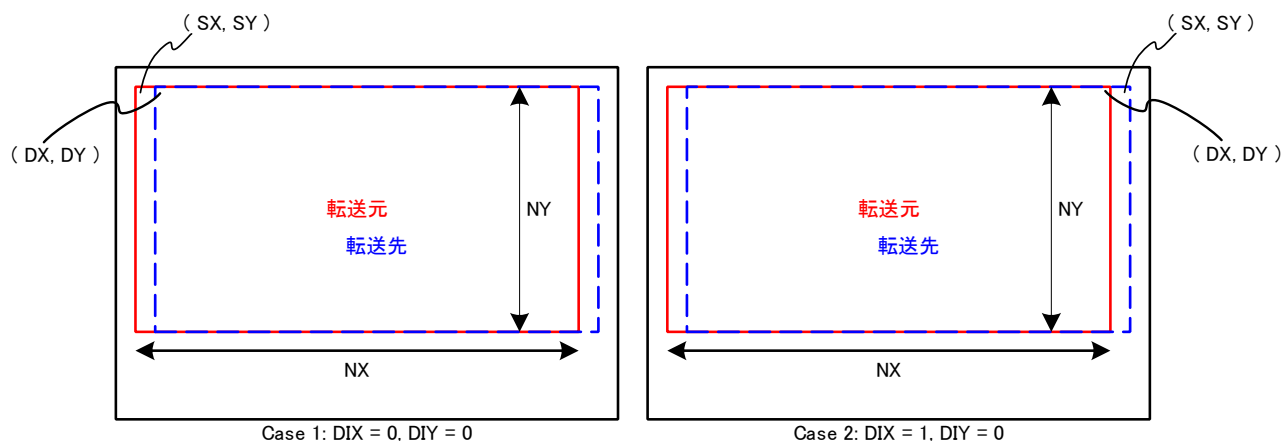


図 2.6.3.2. 同一領域のコピーの DIX が異なる2パターンの例

この Case 1 と Case 2 は、転送元 (赤い四角) と転送先 (青い点線の四角) の領域が一致していますが、処理の結果は異なります。DIX=0 の場合は左から右へ順次処理してゆきます。Case 1 の左上を拡大したイメージを図 2.6.3.3. に示します。

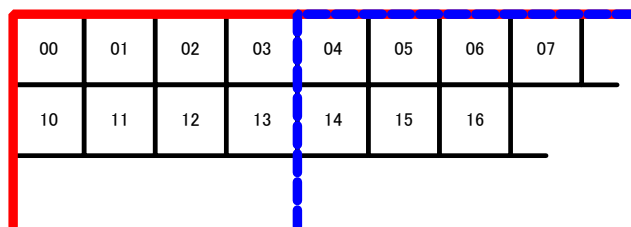


図 2.6.3.3. Case 1 の左上拡大図

具体的な数値が入っている方が分かりやすいので、 $(SX, SY) = (4, 4)$, $(DX, DY) = (8, 4)$ とします。SCREEN5 の想定 (1byte に 2 画素) とします。(4,4) の画素は 00 です。Case 1 の設定で HMMM を実行すると、次の順序で処理されます。

転送元最初の 1byte (00, 01 の画素値) を読み出して、読み出した値を転送先最初の 1byte (04, 05 の位置) へ書き込みます。図 2.6.3.4. のような状態になります。まだ読み出していない (04, 05) をかき潰してしまいます。

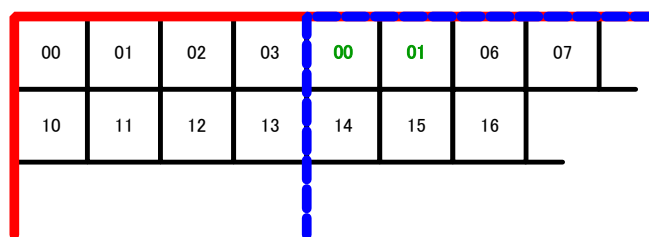


図 2.6.3.4. Case 1 の設定で最初の 1byte の転送完了時点

次に、転送元の 2byte 目の値 (02,03 の画素値)を読み出し、転送先の 2byte 目の位置(06,07 の位置)へ書き込みます。図 2.6.3.5.のような状態になります。まだ読み出していない (06, 07) をかき潰してしまいます。

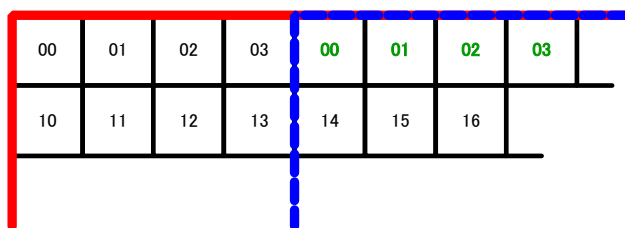


図 2.6.3.5. Case 1 の設定で 2byte の転送が完了した時点

次に、転送元の 3byte 目の値を読み出すわけですが、すでに 1byte 目の転送の時に書き込み済みの値(01,02 の画素値)となり、これを転送して図 2.6.3.6.のような状態になります。元々 (04, 05)だった位置を読み出すわけですが、先ほど書き潰してしまったので、(00,01)に変化しています。そのため、読み出し結果も (00,01)になります。これを、まだ読み出していないさらに先の領域に上書きしていきます。

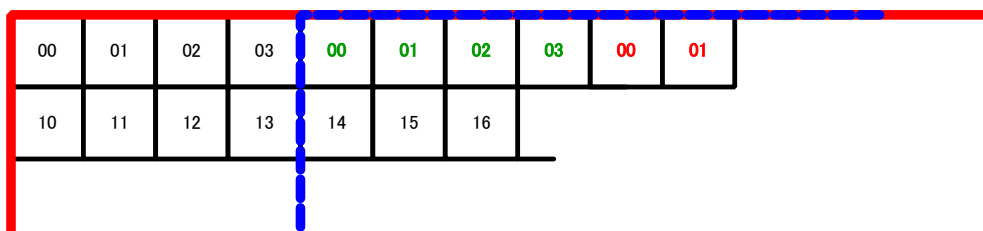


図 2.6.3.6. Case 1 の設定で 3byte の転送が完了した時点

これを繰り返すと、左 4 画素を繰り返したような画像になることが分かります。

MSX Documents VDP

この確認のサンプルプログラム(VCHMMM1.BAS)を動かしたイメージが写真 2.6.3.1.です。

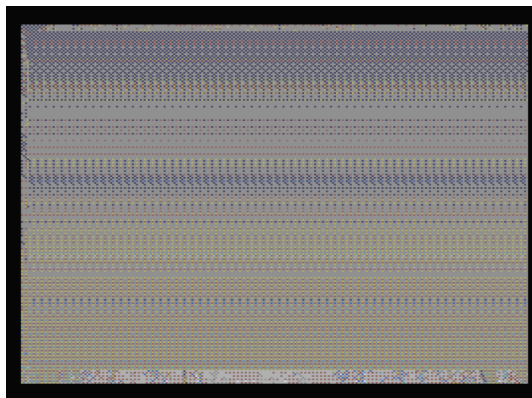


写真 2.6.3.1. サンプル VCHMMM1.BAS を実行して [RETURN] を押した後

VCHMMM1.BAS は、画像を読み込んだ後、キー入力待ちになります。[RETURN]を押すと、HMMM コマンドを実行して写真 2.6.3.1.のように画像が崩れるのを確認できます。

これを利用して左縦一列だけ描画して、画面全体に敷き詰めるのは HMMM で VDP にやらせる、という使い方もあります。

次に、Case 2 の DIX = 1 の場合について説明します。DIX = 1 の場合、右から左へ走査していくため、右上に着目します。右上部分を拡大したイメージを図 2.6.3.7.に示します。

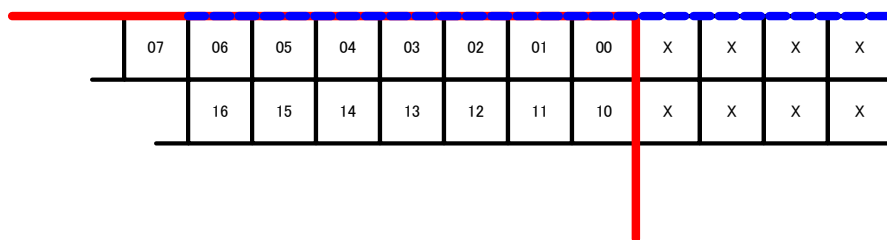


図 2.6.3.7. Case 2 の左上拡大図

まず、最初の転送画素は、(01,00)になるため、これを読み出して転送先に上書きします。上書き後のイメージを図 2.6.3.8.に示します。

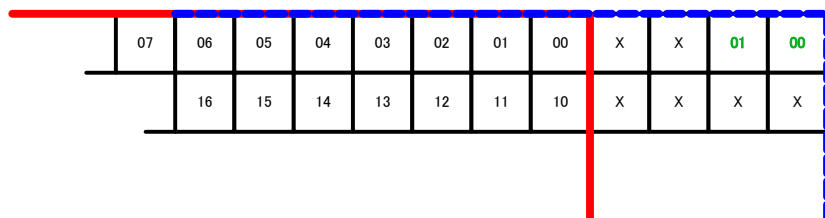


図 2.6.3.8. Case 2 の設定で 1byte の転送が完了した時点

01, 00 を上書きしますが、上書きする位置は、この転送の中では読まない領域になります。

続けて (03, 02) を転送します。

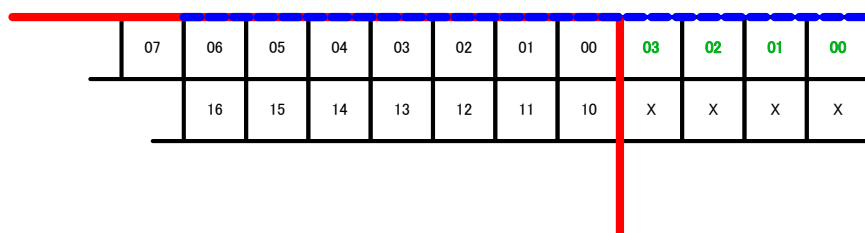


図 2.6.3.9. Case2 の設定で 2byte の転送が完了した時点

03, 02 を上書きしますが、上書きする位置は、この転送の中では読まない領域です。

続けて (05, 04) を転送します。

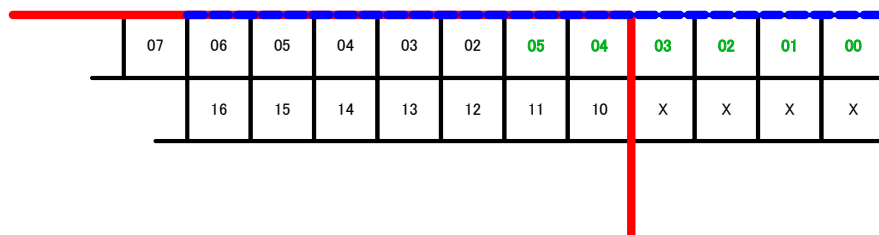


図 2.6.3.10. Case2 の設定で 3byte の転送が完了した時点

05, 04 を上書きしますが、上書きする位置は、この転送の中では読み出し済みで、もう読まない領域です。

このように、「これから読む部分を書き潰す」という問題が発生しないため、綺麗に転送されます。

この確認のサンプルプログラム(VCHMMM2.BAS)を動かしたイメージが写真 2.6.3.2.です。



写真 2.6.3.2. サンプル VCHMMM2.BAS を実行して [RETURN] を押した後

画像が繰り返しパターンではなく、期待通り右へ移動したのを確認できたと思います。

では、VCHMMM1.ASM, VCHMMM2.ASM のプログラム内容について説明します。この2者は、データが違うのみで、プログラムは全く一緒です。そのため、一緒に説明したいと思います。

```
VDP_IO_PORT1    = 0x99
VDP_IO_PORT2    = 0x9A
VDP_IO_PORT3    = 0x9B
```

```
; BSAVE header
db          0xFE
dw          start_address
dw          end_address
dw          start_address
```

ここは、VDP の I/O ポート定義と、BSAVE ファイルヘッダの記述です。

```
; Program body
org          0xC000
```

start_address::

```
; COPY( 4, 4 )-step( 248, 200 ),0 to ( 8, 4 ),0
call exec_hmmm
ret
```

exec_hmmm というサブルーチンを呼んで終わるだけのプログラムになっています。

MSX Documents VDP

VCHMMM1, VCHMMM2 でコメントが異なりますが、このコメントは exec_hmmm の中で実際にどのようなコピーが行われるかを MSX-BASIC 風に書いてみたものです。コメントなので挙動に影響はありません。

```
; R#17 = 32 (R#32からの間接連続書き込み設定)
ld      c, VDP_IO_PORT1
ld      a, 32
di
out     [c], a
ld      a, 17 | 0x80
out     [c], a
ei

; R#32~R#46 に sx~cmr をまとめて書き込む
ld      bc, (15 << 8) | VDP_IO_PORT3 ; R#32~R#46 は 15個のレジスタ
ld      hl, sx
otir
```

間接的なレジスタアクセス (R#17) を使って、VDP コマンドレジスタへの値をまとめて書き込み。

```
; 転送完了待ち
ld      c, VDP_IO_PORT1
ld      de, 15 | 0x80 ; d = 0, e = 15 | 0x80
```

wait_ce_flag:

```
; R#15 = 2 (S#2 を読むための設定)
di
ld      a, 2
out     [c], a
out     [c], e
; a = S#2
in      a, [c]
; R#15 = 0
out     [c], d
out     [c], e
ei
rrca    c, wait_ce_flag ; Cy = CE bit
jr      c, wait_ce_flag

ret
```

転送の完了待ち処理。

大きな矩形の転送なので時間が掛かる想定で、待ちの最中でも割込禁止・解除を繰り返して、割込処理が入れるようにして待っています。

MSX Documents VDP

```

sx:
    dw      4                ; R#32, R#33
sy:
    dw      4                ; R#34, R#35
dx:
    dw      8                ; R#36, R#37
dy:
    dw      4                ; R#38, R#39
nx:
    dw     248                ; R#40, R#41
ny:
    dw     200                ; R#42, R#43
clr:
    db      0                ; R#44 無効だが OTIR で書き込むためのダミー
arg:
    db     0b0000_0000       ; R#45 DIY = 0, DIX = 0
cmr:
    db     0b1101_0000       ; R#46
end_address::
```

ここは、実際に転送する座標・サイズ・VDP コマンドが書かれたデータテーブル。VCHMMM1.ASM と VCHMMM2.ASM が異なる部分です。

2.6.4. HMMV (High-speed move VDP to VRAM)

高速に VDP レジスタ CLR に設定した値で、指定の矩形領域を塗りつぶすコマンドです。

MSX-BASIC の LINE (dx,dy)-step(nx,ny),clr,BF に近いですが、BYTE 単位である点とロジカルオペレーションが使えない点が異なります。(MSX-BASIC の LINE (dx,dy)-step(nx,ny),clr,BF と等価なのは 2462.6.8. LMMV (Logical move VDP to VRAM) の方です。)

動作のイメージを図 2.6.4.1.に示します。

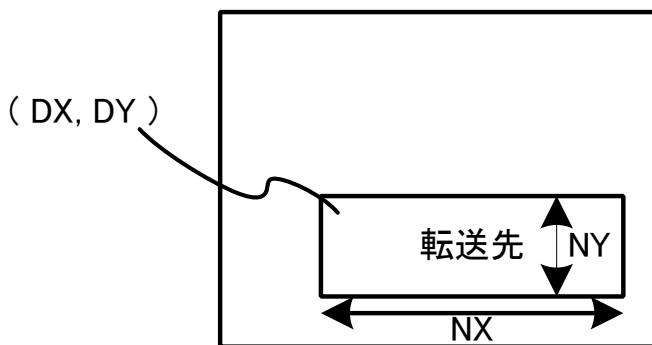


図 2.6.4.1. HMMV コマンドの実行イメージ

図には転送先とありますが、転送元は R#44 (CLR) になります。R#44 に書いてある値を、転送先に敷き詰めます。HMMV コマンド実行中に R#44 を書き替えると、途中からその値に変わりますが、VDP は自分のタイミングでどんどん進んでしまう上に、VDP コマンド自体が VRAM へのアクセス待機によって一定速度では動いていないため、CPU から R#44 を一定時間間隔で書き替えても、転送先が綺麗に格子状に変化するわけではないのでご注意ください。CPU から値を転送したい場合は、2.6.1. HMMC (High-speed Move CPU to VRAM) をご利用ください。つまり、HMMC が 1byte 転送する度に停止して R#44 に書き込まれるのを待ってくれるのに対し、HMMV は待たずにどんどん進むという点だけが異なります。

MSX-BASIC の LINE 文とは異なり、1byte 単位の転送になるため、SCREEN5～7 では R#44 (CLR) の上位 4bit と下位 4bit を異なるパレット番号にして、縦縞の縞々模様で塗りつぶすことも出来ます。

SCREEN5 で上位 4bit に &H2 (緑)を、下位 4bit に &HB (黄色) を指定して、(10,10)-STEP(236,190) の範囲を塗りつぶすサンプル(VCHMMV1.BAS) を実行したイメージを写真 2.6.4.1.に示します。

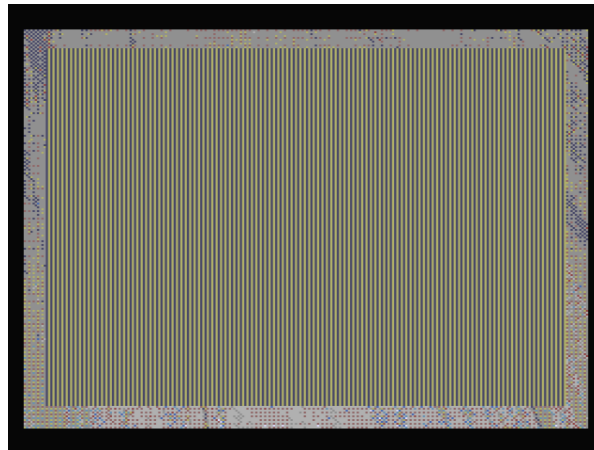


写真 2.6.4.1. VCHMMV1.BAS を実行したイメージ

VCHMMV1.BAS は、背景画像を読み込んでから、"VCHMMV1.ASM をアセンブルした機械語プログラム VCHMMV1.BIN" を読み込んで実行するだけのプログラムです。VCHMMV1.ASM のみ説明します。

```
VDP_IO_PORT1      = 0x99
VDP_IO_PORT2      = 0x9A
VDP_IO_PORT3      = 0x9B
```

```
    ; BSAVE header
    db          0xFE
    dw          start_address
    dw          end_address
    dw          start_address

    ; Program body
    org         0xC000
```

例によって VDP のポート番号定義と、BSAVE 形式のファイルヘッダです。プログラム自体は C000H 番地から始まります。

```
start_address::
```

```
    ; LINE ( 10, 10 ) - STEP( 236, 190 ), &H2B, BF
    call exec_hmmv
```


MSX Documents VDP

ret

exec_hmmv に VDP コマンド HMMV を実行する実体を入れているので、それを呼び出してから BASIC へ戻る ret です。

```
; -----  
;      HMMV  
; -----  
exec_hmmv::  
    ; R#17 = 36 (R#36からの間接連続書き込み設定)  
    ld      c, VDP_IO_PORT1  
    ld      a, 36  
    di  
    out     [c], a  
    ld      a, 17 | 0x80  
    out     [c], a  
    ei  
  
    ; R#36~R#46 に sx~cmr をまとめて書き込む  
    ld      bc, (11 << 8) | VDP_IO_PORT3 ; R#36~R#46 は 11 個のレジスタ  
    ld      hl, dx  
    otir
```

間接的なレジスタ連続書き込みを実施するために、R#17 に 36 を書き込み、PORT#3 を使って連続的に書き込んでいます。ここまでの HMMV は発動します。

```
    ; 転送完了待ち  
    ld      c, VDP_IO_PORT1  
    ld      de, 15 | 0x80 ; d = 0, e = 15 | 0x80  
  
wait_ce_flag:  
    ; R#15 = 2 (S#2 を読むための設定)  
    di  
    ld      a, 2  
    out     [c], a  
    out     [c], e  
    ; a = S#2  
    in      a, [c]  
    ; R#15 = 0  
    out     [c], d  
    out     [c], e
```

MSX Documents VDP

```
ei
rrca                ; Cy = CE bit
jr      c, wait_ce_flag

ret
```

ここで HMMV コマンドが完了するのを待っています。

律儀に待っていますが、この間に CPU は別の演算をさせると、VDP の HMMV コマンドと CPU の別演算を並列動作させることが出来ます。並列動作になる部分をふやすことで MSX のパフォーマンスを最大限に引き出せるわけです。

VDP コマンドのレジスタはダブルバッファにはなっておらず、動作中に R#32～R#46 を書き替えると、動作に影響が出てしまいますのでご注意ください。

```
dx:
    dw      10                ; R#36, R#37
dy:
    dw      10                ; R#38, R#39
nx:
    dw      236               ; R#40, R#41
ny:
    dw      190               ; R#42, R#43
clr:
    db      0x2B              ; R#44 無効だが OTIR で書き込むためのダミー
arg:
    db      0b0000_0000      ; R#45 DIY = 0, DIX = 0
cmr:
    db      0b1100_0000      ; R#46
end_address::
```

LINE (10,10)-STEP(236,190),&H2B,BF に相当するレジスタ設定値テーブルです。※MSX-BASIC では SCREEN5 で色番号 &H2B を指定することは出来ません。あくまでイメージです。

2.6.5. LMMC (Logical move CPU to VRAM)

論理的に CPU から VRAM へ書き込むコマンドです。

MSX-BASIC の「COPY 配列変数 to 転送先座標」に相当します。

2.6.6. LMCM (Logical move VRAM to CPU)

論理的に VRAM から CPU へ読み出すコマンドです。

MSX-BASIC の「COPY 転送元領域指定 to 配列変数」に相当します。

2.6.7. LMMM (Logical move VRAM to VRAM)

論理的に VRAM 上の矩形領域を別の矩形領域へ転送するコマンドです。

MSX-BASIC の「COPY 転送元領域指定 to 転送先座標指定」に相当します。

2.6.8. LMMV (Logical move VDP to VRAM)

論理的に指定のカラーコードで指定の領域を塗りつぶすコマンドです。

MSX-BASIC の LINE (dx,dy)-step(nx,ny),clr,BF に相当します。ロジカルオペレーションを使わず、BYTE 単位の位置/サイズ指定で構わない場合は、2.6.4. HMMV (High-speed move VDP to VRAM)を使った方が高速になります。用途に応じて使い分けると良いでしょう。

2.6.9. LINE

直線を描画するコマンドです。

MSX-BASIC の「LINE 始点-STEP サイズ, カラーコード」に相当します。

2.6.10. SRCH

指定のカラーコード、または指定のカラーコード以外のカラーコードが見つかるまで探索するコマンドです。

MSX-BASIC の PAINT 命令の処理の一部に相当します。

2.6.11. PSET

点を描画するコマンドです。

MSX-BASIC の「PSET 座標指定, カラーコード」に相当します。

2.6.12. POINT

指定の座標のカラーコードを読み取るコマンドです。

MSX-BASIC の「POINT(座標指定)」に相当します。

2.6.13. STOP

稼働中の VDP コマンドを停止させるコマンドです。

3. 解析

各種レジスタには、やや冗長な設定があり、動作が未定義とされている設定値が存在します。

通常の IC 設計では、下記の可能性が考えられます。

- (1) Don't care 扱いで、回路規模削減に寄与するように、機能としては意味が無いかもしれない構成の挙動を割り付ける
- (2) 実際は回路のテストなどを行うための隠し機能が割り付けられており、それを公開したくないので未定義としている
- (3) 実際はある種の機能を割り付ける予定だったが、IC が出来上がってからバグが見つかり、機能をカットし、未定義扱いにしている

いずれの場合も、未定義の設定値を書き込むと、興味深い動作をする場合があります。

この興味深い動作を探るための解析結果をここに記載したいと思います。一方で、TMS9918 に関しては、かなりの下図の互換 IC が出ています。TI 社製だけでも TMS9118 とか TMS9929 とか、微妙に違うものが多数出ており、TI 社以外からも互換 IC がリリースされており、それを搭載した MSX も存在しています。本解析では、CASIO の MX-101 (TMS9118 搭載) と、PANASONIC の FS-A1GT と、エミュレーター BlueMSX と OpenMSX の挙動を見ながら解析しています。それ以外の機種で同様に動作する保証はありません。

3.1. SCREEN6 以下と SCREEN7 以上の DRAM 使用方法について

V9938 は、搭載 VRAM 量を 64KB と 128KB の 2 種類から選択できるようになっている。MSX ハードメーカーは、どちらにするか決定して、同じ VDP に対して 64KB の DRAM を 1 組または 2 組搭載します。

SCREEN7 以上は VRAM 128KB 以上搭載機でしか使えない。容量的には 64KB あれば 1 画面分は確保できますが、DRAM のアクセス帯域の関係で 64KB の DRAM が 2 組必要となります。

DRAM アクセス帯域とは、データバスが 8bit で接続されている DRAM に対しては、最高速度で 1clock あたり 8bit しか読み書きできません。約 21MHz で動作する VDP は、どんなに頑張っても秒間 21MB しか流せないこととなります。DRAM は、アドレス指定などに所定のシーケンスを守らないと読み書きできません。読むためのシーケンスにも時間がかかるため、その分のオーバーヘッドがかかります。21MB のうち 5 割くらいがオーバーヘッドだとすれば、10.5MB しかアクセスできないこととなります。このようなアクセス速度の制限のことを DRAM アクセス帯域と呼びます。

説明の便宜上、ここでは DRAM の 1 組目の 64KB を VRAM-A、2 組目の 64KB を VRAM-B と表記します。

SCREEN6 以下では、VRAM-A が VRAM の 00000h～0FFFFh、VRAM-B が 10000h～1FFFFh に対応しています。

SCREEN7 以上では、VRAM-A が VRAM の 00000h～1FFFEh の偶数番地、VRAM-B が VRAM の 00001h～1FFFFh の奇数番地に対応しています。

このアドレスの組み替えは、VDP 内部でハード的に行われており、MSX-BASIC の VPOKE/VPEEK だけでなく、機械語プログラムからの Port#0 を使ったアクセスでもこのアドレス組み替えの差を意識せずにアドレス指定できます。1 バイトずつ読んだり書いたりしている分には、DRAM のどの番地が、VRAM 上のどの番地に対応しているのか意識する必要はありません。

しかし、R#0, R#1 の M5～M1 を直接書き替えて画面切り替えする場合には、意識しなければマズい場面が出てきます。例えば、水平帰線割込を使って、画面の上半分は SCREEN5、下半分は SCREEN7 にするような制御を行った場合、上半分の表示と下半分の表示とで、VRAM アドレスが DRAM 上のどの番地として認識しているか意識して回避するコードにしなければ、上下画面の一方が他方の画像を壊してしまいます。

VRAM の並びが変わっていることの確認プログラム(SC5SC7A.BAS, SC5SC7B.BAS) を下記に示します。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN5
110 BLOAD"IMAGE1.SC5",S:COLOR=RESTORE
120 VDP(0)=(VDP(0)AND&HF1)OR&H0A
130 GOTO 130
```

100 行目で SCREEN5 に切り替え。

MSX Documents VDP

110 行目で SCREEN5 用の画像ファイルを読み込んでいます。

120 行目で R#0 を書き替えて、VDP としては SCREEN7 に相当するモードに切り替えています。この切替では、VDP のモードレジスタしか変更しないため、VRAM の内容を破壊しません。

130 行目は、画面が消えてしまわないように無限ループして待機しています。

この動作確認画面を写真 3.1.1. に示します。



写真 3.1.1. SC5SC7A.BAS の実行結果

多少ゴミのようなものが出ていますが、これはスプライトです。

次に、普通に SCREEN7 のところに、SCREEN5 用の画像データを強引に読み込むプログラム(SC5SC7B.BAS)。

```
100 DEFINT A-Z:COLOR15,0,0:SCREEN7
110 BLOAD"IMAGE1.SC5",S
120 GOTO 120
```

100 行目で SCREEN7 に切り替え。

110 行目で SCREEN5 用の VRAM データを気にせず読み込み。

120 行目で待機。

この動作結果を写真 3.1.2. に示します。

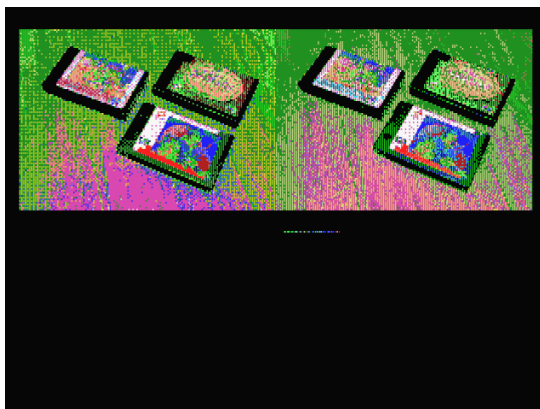


写真 3.1.2. SC5SC7B.BAS の実行結果

同じ VRAM データであっても、SCREEN5 の状態でロードした結果と、SCREEN7 の状態でロードした結果が明らかに異なることが分かります。

VDP は、SCREEN7 以上の画面モードが指定されると、2 組ある DRAM を同時にアクセスすることで DRAM 帯域を倍増させています。画面表示の場合、左から右へ、上から下へ、と表示のために送出する順番が決まっているため、SCREEN7 以降は、2 組の DRAM に対して同時にアクセスさせ、DRAM 制御を簡略化しています。

【note】

VRAM64KB の機種はあまり発売されなかったようなので、SCREEN6 以下でも SCREEN7 以上と同じアクセスにした方が、このようなややこしい仕様にはしなかったと思われますが、当時の DRAM は高価こともあったので、このような回避策に至ったのかもしれませんが。

3.2. RGB 出力とコンポジット出力

V9938/V9958 には、コンポジットビデオ出力と、RGB 出力の2種類の出力が出ています。図 3.2.1.に V9938 のピンアサインを示します。

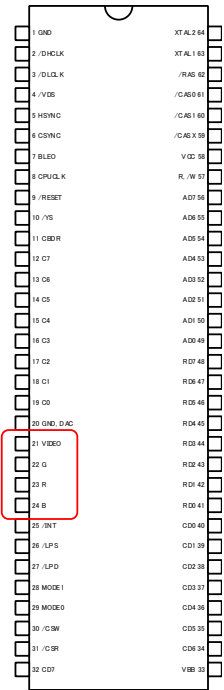


図 3.2.1. V9938 ピンアサイン

図の中で赤枠で囲ってある部分がコンポジットビデオ出力と RGB ビデオ出力になります。次に、図 3.2.2.に V9958 のピンアサインを示します。

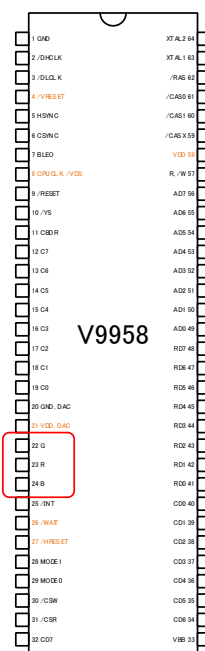


図 3.2.2. V9958 ピンアサイン

図 3.2.2. で赤文字のピン名は V9938 から変更になった名前になります。

R#8 の BW(bit0)、R#20、R#21、R#22 は、コンポジットビデオ出力用のレジスタなので、V9958 では内部的には削除されているものと思います。

V9938 を搭載している MSX2 であっても、ほとんどの機種種のコンポジットビデオ出力は、VDP の 21 ピンではなく、22 ～24 の RGB 出力からエンコーダー IC を利用して生成した信号になっているため、R#8 の BW(bit0)、R#20、R#21、R#22 は実質的に出力に影響を及ぼさないレジスタになっています。写真 3.2.1. と写真 3.2.2. に実験の結果を示します。

MSX Documents VDP

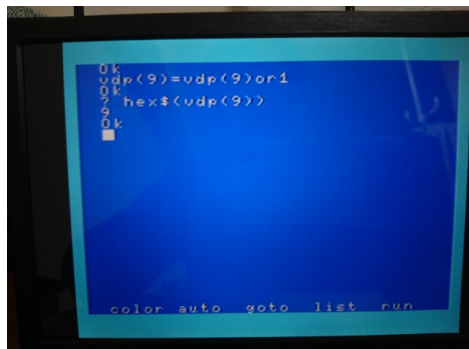


写真 3.2.1. FS-A1GT で R#8 の bit0 に 1 を立てても表示に変化無し



写真 3.2.2. FS-A1F で R#8 の bit0 に 1 を立てても表示に変化無し

【note】

ほとんどの機種と書きましたが、VDP の 21 ピン出力を MSX2 本体のコンポジットビデオ出力端子に繋げている機種があるかどうかは未確認。RGB 出力を持たない機種であればあるかもしれませんが、MSX テクニカルハンドブックや MSX DataPack 等には R#8 BW, R#20~22 は、「MSX では使用しない」と書かれているため、もしかすると MSX2 規格として、VDP21 ピンは使わず 22~24 ピンの出力のみを使用するよう規定されているのかもしれませんが。MSX2 の規格書のようなものを確認できないので、これもあくまで推測です。実際、いくつかの MSX2 本体所有者の方に、R#8 の bit0 に 1 を書き込んでもらうテストをして頂いたところ、みな変化無しとの回答を頂きました。FS-A1F, HB-F500, HB-F1XDmk2, PHC-23 で確認して頂いています。

3.3. 192 ライン/212 ラインモード

R#9 の bit7 には LN というライン数指定のレジスタが存在しています。SCREEN0～4 であっても LN=1 にすると 212 ライン表示されるのですが、増えた 20 ラインに何が表示されるのでしょうか？

Pattern Name Table は、増えた分だけサイズが増えます。SCREEN0, 1, 3 の場合はそれだけで解決しますね。

SCREEN2 と 4 の場合、上段・中段・下段の区分けがあり、Y=192～211 は、下段のさらに下になります。Pattern Generator Table 及び Color Table が拡張されるのでしょうか？このあたり、実際に実験して確認してみたいと思います。

<ToDo: 確認コードを作成してここでせつめいする>

3.4. 存在しないレジスタへのアクセス

R#17 による VDP レジスタアドレス指定は 6bit の範囲を指定できます。6bit で表現できる範囲は 0～63 ですが、24, 28～31, 47～63 にはレジスタが存在していません。ここに書き込んだらどうなるのでしょうか？

結論を言えば、「何もおきません」です。確認のために、FS-A1GT を使って実験してみました。

まず、下記のルーチンを用意しました。

```

        scope write_vdp_register
write_vdp_register::
        ld      c, VDP_IO_PORT1
        out     [c], b
        or      a, 0x80
        out     [c], a
        ret
endscope

```

write_vdp_register は、Aレジスタに指定した番号の VDP レジスタに、Bレジスタに指定した値を書き込むルーチンです。

MSX Documents VDP

```
        scope write_increment_datas
write_increment_datas::
        ld            b, 0
loop1:
        call write_vdp_register
        inc          b
        jp           nz, loop1
        ret
        endscope
```

write_increment_datas は、Aレジスタで指定した VDPレジスタに対して 0,1,2, ... , 255 とインクリメント値を書き込むルーチンです。

下記が上記サブルーチンを利用して作った実験プログラムです。

```
        scope test001
test001::
        ld            de, message_test001
        call puts

        di
        ld            a, 24
        call write_increment_datas
        ld            a, 28
        call write_increment_datas
        ld            a, 29
        call write_increment_datas
        ld            a, 30
        call write_increment_datas
        ld            a, 31
        call write_increment_datas
        ld            a, 47
        call write_increment_datas
        ld            a, 48
        call write_increment_datas
        ld            a, 49
        call write_increment_datas
        ld            a, 50
        call write_increment_datas
        ld            a, 51
        call write_increment_datas
        ld            a, 52
```

```

call write_increment_datas
ld      a, 53
call write_increment_datas
ld      a, 54
call write_increment_datas
ld      a, 55
call write_increment_datas
ld      a, 56
call write_increment_datas
ld      a, 57
call write_increment_datas
ld      a, 58
call write_increment_datas
ld      a, 59
call write_increment_datas
ld      a, 60
call write_increment_datas
ld      a, 61
call write_increment_datas
ld      a, 62
call write_increment_datas
ld      a, 63
call write_increment_datas
ei

ld      de, message_ok
call puts
call chget
endscope

```

VDPの中を観測できないので、厳密に「何も起きていない」と判定することは出来ませんが、少なくとも画面表示に何も変化が無いことは確認できました。

sample プログラムの VDPRTTEST です。

3.5. アドレスオートインクリメントのカウンタの挙動

R#17 を指定した VDP レジスタへの連続書き込みは、オートインクリメントモードを備えています。

0～63 の範囲でインクリメントして、63 の次は 0 に戻ります。

その確認コードを下記に示します。3.4. で作ったルーチンを利用しています。

```

scope test004
test004::
    ld        de, message_test004
    call     puts

    di
    ld        a, 17
    ld        b, 47
    call     write_vdp_register          ; R#17 = 47

    ld        c, VDP_IO_PORT3
    ld        a, 255
    out       [c], a                    ; R#47 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#48 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#49 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#50 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#51 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#52 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#53 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#54 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#55 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#56 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#57 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#58 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#59 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#60 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#61 = 255 (N/A)
    ld        a, 255
    out       [c], a                    ; R#62 = 255 (N/A)
    ld        a, 255

```

```

        out        [c], a                ; R#63 = 255 (N/A)
        ld         a, 0
        out        [c], a                ; R#0  = 0
(SCREEN0 WIDTH40)
        ld         a, 0x70
        out        [c], a                ; R#1  = 0x70
(SCREEN0 WIDTH40)
        ei

        ld         de, message_ok
        call puts
        call chget

        di
        ld         a, 17
        ld         b, 47
        call write_vdp_register          ; R#17 = 47

        ld         c, VDP_IO_PORT3
        ld         a, 255
        out        [c], a                ; R#47 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#48 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#49 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#50 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#51 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#52 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#53 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#54 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#55 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#56 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#57 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#58 = 255 (N/A)
        ld         a, 255
        out        [c], a                ; R#59 = 255 (N/A)
        ld         a, 255

```

	out	[c], a	; R#60 = 255 (N/A)
	ld	a, 255	
	out	[c], a	; R#61 = 255 (N/A)
	ld	a, 255	
	out	[c], a	; R#62 = 255 (N/A)
	ld	a, 255	
	out	[c], a	; R#63 = 255 (N/A)
	ld	a, 0	
(SCREEN1)	out	[c], a	; R#0 = 0
	ld	a, 0x60	
(SCREEN1)	out	[c], a	; R#1 = 0x60
	ei		
	endscope		

SCREEN1 にしてから上記ルーチンを実行して、R#47, ... , R#63 にダミーの値を書き込んだ後、R#0, R#1 に続けて書き込んでいきます。

R#0, R#1 には SCREEN0 WIDTH40 のモード設定を書き込みます。すると、画面が変化します。

キー入力待ちを経て、同じように今度は SCREEN1 のモード設定を書き込みます。すると画面が元に戻ります。

R#0, R#1 に書き込んでいるのは明らかですので、63 のつぎは 0 に戻るというシンプルな造りになっていることが分かりました。

sample プログラムの VDPRTTEST です。

3.6. 透明スプライトの衝突判定

TMS9918, V9938, V9958 のスプライト機能には衝突判定を行う機能があります。これはスプライトのパターンが存在するドット同士が重なっていた場合にステータスレジスタに重なり判定結果を出す機能です。

しかしこの判定、TMS9918 と V9938/V9958 とでは、0 番(透明)の色の扱いに差異があります。

TMS9918 の場合、0 番の色を指定されていても衝突判定されます。

V9938/V9958 の場合、0 番の色を指定されていると衝突判定されません。

MSX Documents VDP

<<ToDo>>

★R#0,R#1 の M5～M1 の未定義設定。

★SCREEN5 で pattern name table で「必ず 1 たてること」となってるビットに 0 を書いてみる。

→実際にやると、画面表示に興味深い影響が出る。その辺を詳しく書く。

★VDP コマンド稼働中に書き換えると VDP コマンドの邪魔をするコントロールレジスタ(R#9,R#18 はコマンド妨害を確認済み)。

★OpenMSX チームの VRAM アクセスタイミング解析の情報

4. 逆引

4.1. アドレス指定

Pattern Name Table のアドレス指定

2.4.1.3. Pattern Name Table Address (R#2)

Pattern Generator Table のアドレス指定

2.4.1.5. Pattern Generator Table Base Address Register (R#4)

Color Table のアドレス指定

2.4.1.4. Color Table Base Address Register Low (R#3)

2.4.1.11. Color Table Base Address Register High (R#10)

Sprite Attribute Table のアドレス指定

2.4.1.6. Sprite Attribute Table Base Address Register Low (R#5)

2.4.1.12. Sprite Attribute Table Base Address Register High (R#11)

Sprite Pattern Generator Table のアドレス指定

2.4.1.7. Sprite Pattern Generator Table Base Address Register (R#6)

Sprite Color Table のアドレス指定

2.4.1.6. Sprite Attribute Table Base Address Register Low (R#5)

2.4.1.12. Sprite Attribute Table Base Address Register High (R#11)

4.2. サンプルプログラム

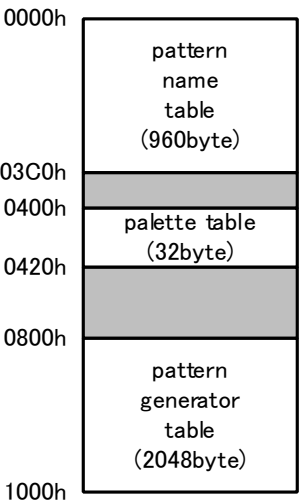
TMS9918/R1SPRITE.BAS

2.4.1.2. Mode Register 1 (R#1)

付録. VRAM マップ

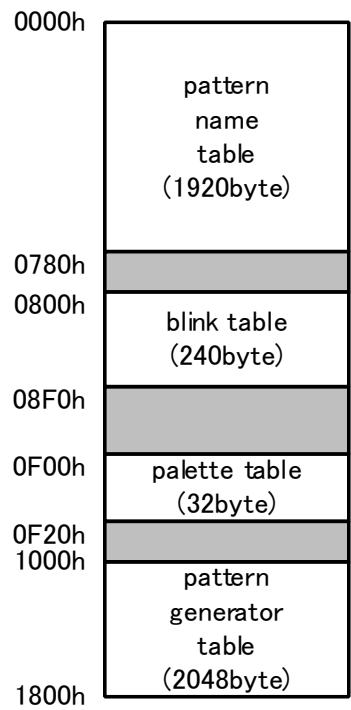
MSX-BASIC が各スクリーンモードに変更したときに設定される各テーブルのアドレスをまとめたマップです。グレーの領域は未使用・アドレス後半の図示していない部分も未使用ですが、テーブルアドレスを変更することにより使用することもできますし、VDP とは無関係なデータ置き場メモリとしても使うことができます。

SCREEN0 (WIDTH40) : TEXT1: TMS9918/V9938/V9958



※ 01000h～1FFFFh の領域は未使用。

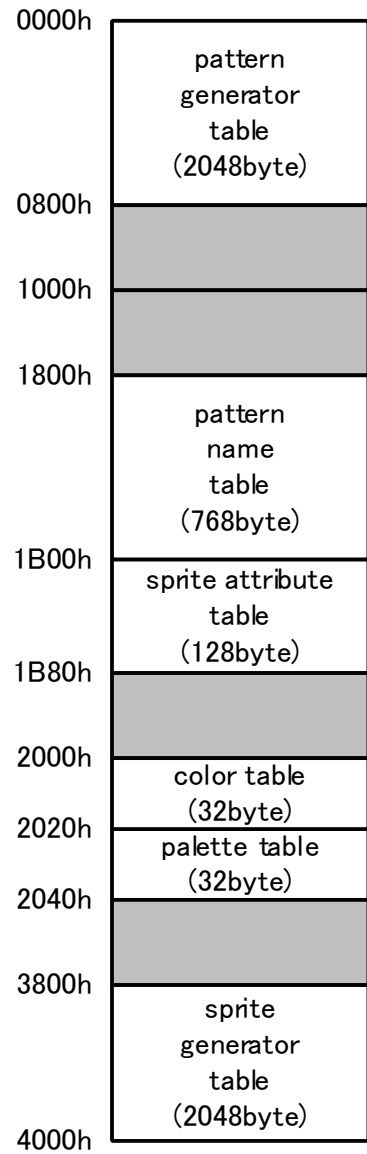
SCREEN0 (WIDTH80) : TEXT2: V9938/V9958



※TMS9918 では利用できません。

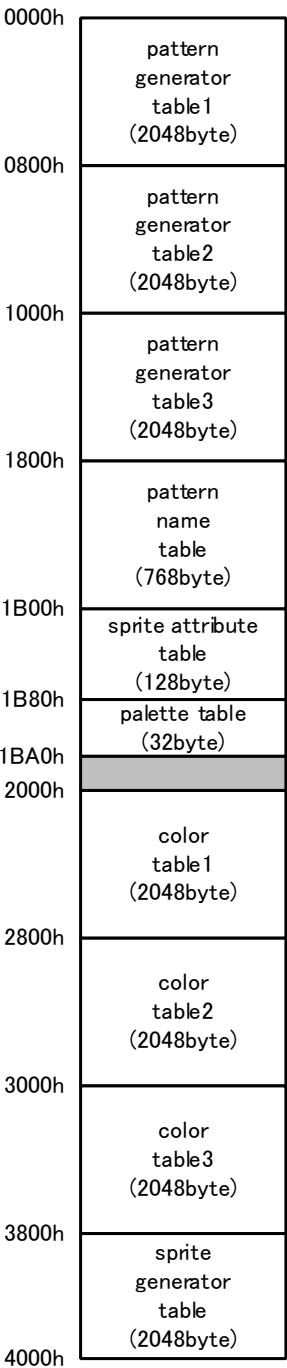
※ 01800h～1FFFFh の領域は未使用。

SCREEN1 : GRAPHIC1: TMS9918/V9938/V9958



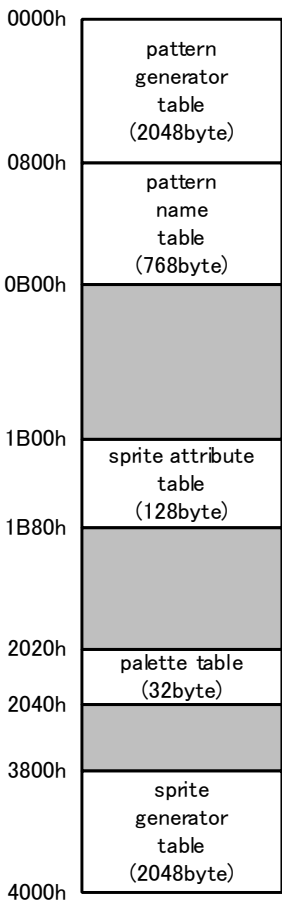
※ 04000h～1FFFFh の領域は未使用。

SCREEN2 : GRAPHIC2: TMS9918/V9938/V9958



※ 04000h～1FFFFh の領域は未使用。

SCREEN3 : MULTI COLOR: TMS9918/V9938/V9958



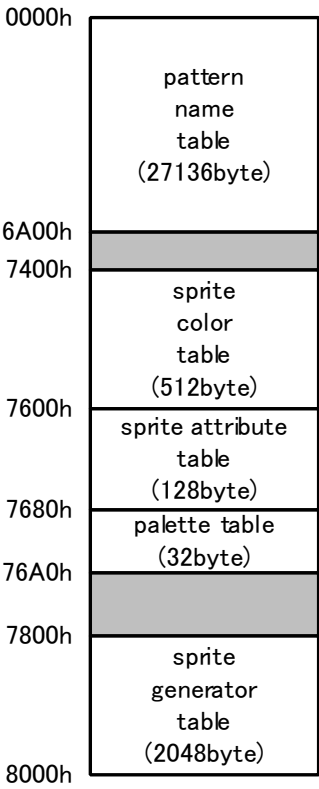
※ 04000h～1FFFFh の領域は未使用。

SCREEN4 : GRAPHIC3: V9938/V9958

0000h	pattern generator table1 (2048byte)
0800h	pattern generator table2 (2048byte)
1000h	pattern generator table3 (2048byte)
1800h	pattern name table (768byte)
1B00h	
1BA0h	palette table (32byte)
1BC0h	
1C00h	sprite color table (512byte)
1E00h	sprite attribute table (128byte)
2000h	pattern generator table1 (2048byte)
2800h	pattern generator table2 (2048byte)
3000h	pattern generator table3 (2048byte)
3800h	sprite generator table (2048byte)
4000h	

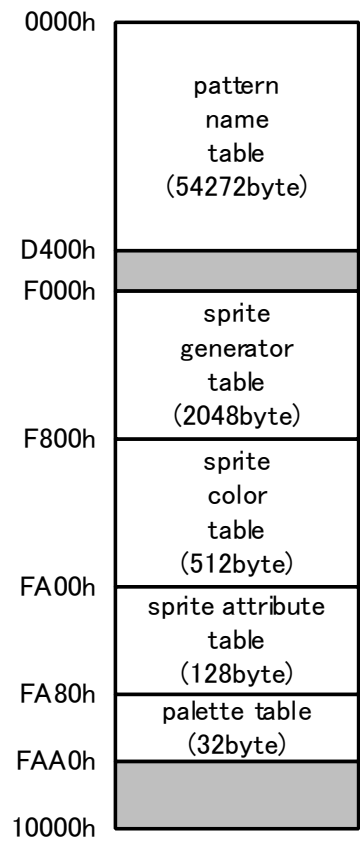
※ 04000h～1FFFFh の領域は未使用。

SCREEN5, 6 : GRAPHIC4, 5: V9938/V9958



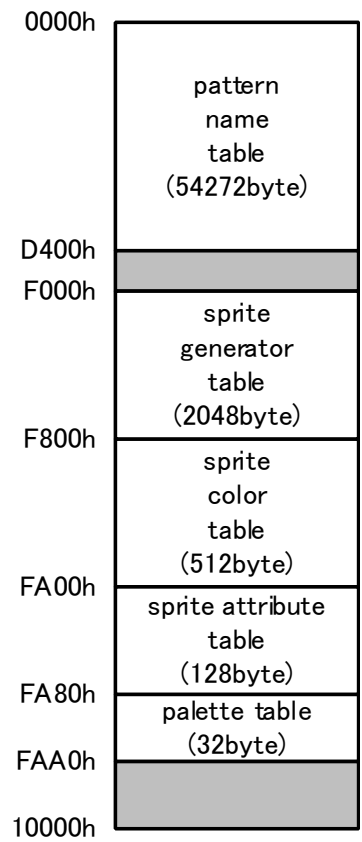
※MSX-BASIC では、SET PAGE 命令で 4 種類のアドレスを切り替えられる。具体的にはディスプレイページ指定×8 000h が、上記アドレスに加算されたアドレスになる。

SCREEN7, 8 : GRAPHIC6, 7: V9938(VRAM128KB 以上)/V9958



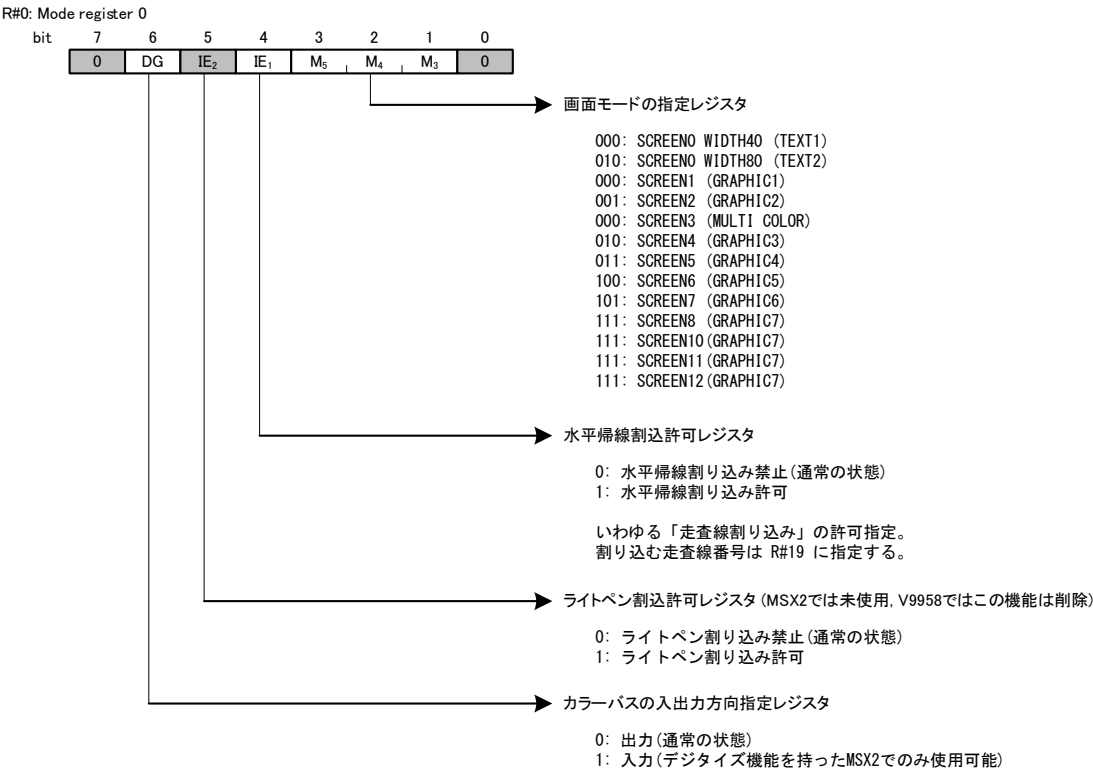
※MSX-BASIC では、SET PAGE 命令で 2 種類のアドレスを切り替えられる。具体的にはディスプレイページ指定×1 0000h が、上記アドレスに加算されたアドレスになる。

SCREEN10,11,12 : GRAPHIC7: V9958



※MSX-BASIC では、SET PAGE 命令で 2 種類のアドレスを切り替えられる。具体的にはディスプレイページ指定×1 0000h が、上記アドレスに加算されたアドレスになる。

付録. コントロールレジスタ



MSX Documents VDP

R#1: Mode register 1

bit	7	6	5	4	3	2	1	0
	0	BL	IE ₀	M ₁	M ₂	0	SI	MAG

→ スプライトの拡大指定レジスタ

0: 拡大しない
1: 拡大する

→ スプライトのサイズ指定レジスタ

0: 8x8
1: 16x16

→ 画面モードの指定レジスタ

10: SCREEN0 WIDTH40 (TEXT1)
10: SCREEN0 WIDTH80 (TEXT2)
00: SCREEN1 (GRAPHIC1)
00: SCREEN2 (GRAPHIC2)
01: SCREEN3 (MULTI COLOR)
00: SCREEN4 (GRAPHIC3)
00: SCREEN5 (GRAPHIC4)
00: SCREEN6 (GRAPHIC5)
00: SCREEN7 (GRAPHIC6)
00: SCREEN8 (GRAPHIC7)
00: SCREEN10 (GRAPHIC7)
00: SCREEN11 (GRAPHIC7)
00: SCREEN12 (GRAPHIC7)

→ 垂直帰線割込許可レジスタ

0: 垂直帰線割込み禁止
1: 垂直帰線割込み許可 (通常の状態)

いわゆる「垂直同期割込み」の許可指定。
タイマーのように使われており、H.TIMIフックの割込みそのものである。

→ 画面表示許可レジスタ

0: 画面表示禁止
1: 画面表示許可 (通常の状態)

名前は blind を示す BL だが、0 で表示禁止である。

R#2: Pattern name table base address register

bit	7	6	5	4	3	2	1	0
	0	A16	A15	A14	A13	A12	A11	A10

→ SCREEN0~4 のパターンネームテーブルのアドレスを指定するレジスタ。
A9~A0 は 0 固定。

R#3: Color table base address register low

bit	7	6	5	4	3	2	1	0
	A13	A12	A11	A10	A9	A8	A7	A6

→ SCREEN2, 4 のカラーテーブルのアドレスを指定するレジスタ。
A5~A0 は 0 固定。A16~A14 は R#10 で指定する。

R#4: Pattern generator table base address register

bit	7	6	5	4	3	2	1	0
	0	0	A16	A15	A14	A13	A12	A11

→ SCREEN0~4 のパターンジェネレーターテーブルのアドレスを指定するレジスタ。
A10~A0 は 0 固定。

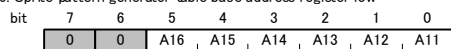
R#5: Sprite attribute table base address register low

bit	7	6	5	4	3	2	1	0
	A14	A13	A12	A11	A10	A9	1	1

→ SCREEN1~12 のスプライトアトリビュートテーブルのアドレスを指定するレジスタ。
A8~A0 は 0b110000000 固定。A16~A15 は R#11 で指定。

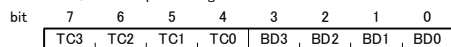
MSX Documents VDP

R#6: Sprite pattern generator table base address register low



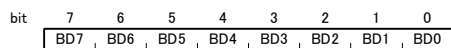
SCREEN1~12のsprayパターンジェネレータテーブルのアドレスを指定するレジスタ。
A10~A0は0固定。

R#7: Text color/Back drop color register



GRAPHIC7以外の場合

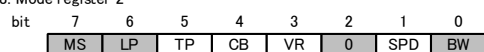
TC3~TC0 SCREEN0におけるテキストの色
BD3~BD0 バックドロップの色



GRAPHIC7の場合

BD7~BD0 バックドロップの色
SCREEN 10,11,12の場合、YJKではなくSCREEN8と同じ色になる。

R#8: Mode register 2



白黒モード

0: カラー (通常の状態)
1: 白黒32階調

MSXでは無効。

spray無効モード

0: spray表示 (通常の状態)
1: spray非表示

非表示にすると、VDPコマンドが多少速くなる。

VRAMの種類指定

0: 16K×1bit または 16K×4bit
1: 64K×1bit または 64K×4bit

カラーバスの入出力選択

0: 出力モード (通常の状態)
1: 入力モード

カラーコード0をカラーパレットの色にする

0: 透明の扱い (通常の状態)
1: カラーパレットの色の扱い

MSX-BASICで下記を実行すれば違いがわかる。
COLOR 15, 0, 2
VDP (9)=VDP (9) OR 32

ライトペン (MSX2では未使用, V9958ではこの機能は削除)

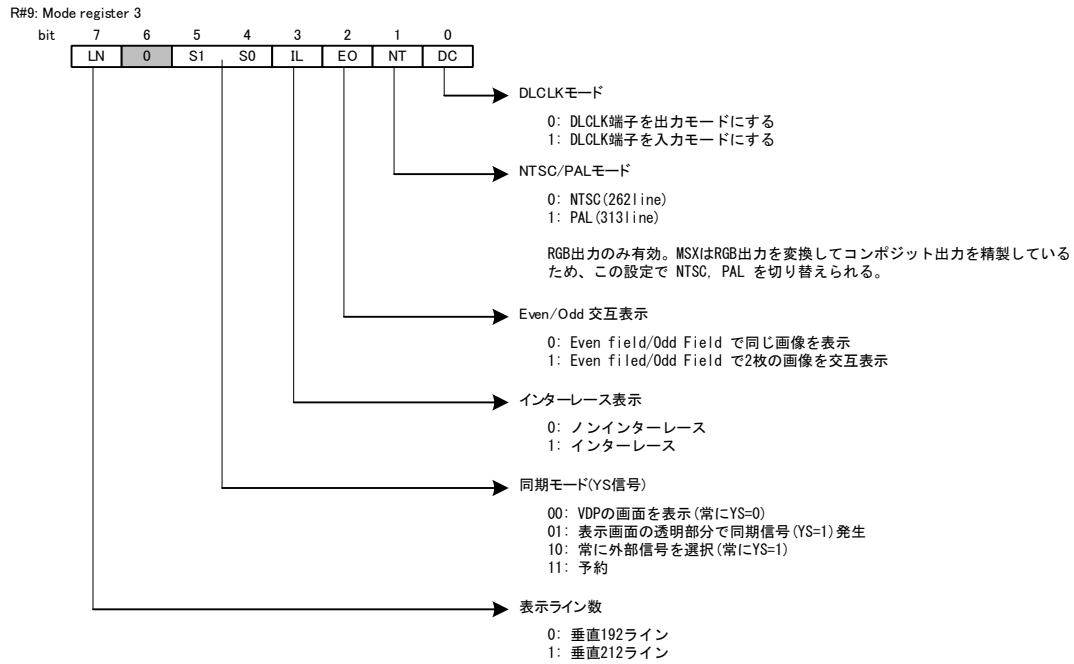
0: ライトペンを使用しない (MSXでは常に0にする)
1: ライトペンを使用する

マウス (MSX2では未使用, V9958ではこの機能は削除)

0: マウスを使用しない (MSXでは常に0にする)
1: マウスを使用する

MS, LP は、MSXでは使用しない。
(MSX用のマウスは、PSGIに接続されるのでこの機能とは無関係)

MSX Documents VDP



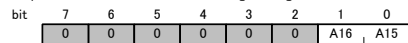
MSX Documents VDP

R#10: Color table base address register high



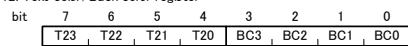
SCREEN2, 4 のカラーテーブルのアドレスを指定するレジスタ。
A5~A0 は 0 固定。A13~A6 は R#3 で指定する。

R#11: Sprite attribute table base address register high



SCREEN1~12 のスプライトアトリビュートテーブルのアドレスを指定するレジスタ。
A8~A0 は 0b00000000 固定。A14~A9 は R#5 で指定。

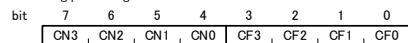
R#12: Text color/Back color register



パターンジェネレータテーブルでドットに対応する bit の値が 0 の部分の色を指定
パターンジェネレータテーブルでドットに対応する bit の値が 1 の部分の色を指定

SCREEN0 WIDTH 80 (TEXT2) のモードにおいて、パターンにプリンク属性が付いているときは、このレジスタで指定された色と R#7 で指定された色が交互に表示される。

R#13: Blinking period register

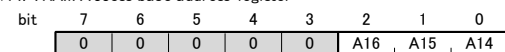


偶数ページの表示時間 (GRAPHIC4~7 の場合)
R#7 の色の表示時間 (TEXT2 の場合)

奇数ページの表示時間 (GRAPHIC4~7 の場合)
R#12 の色の表示時間 (TEXT2 の場合)

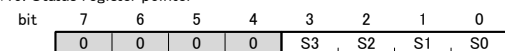
SCREEN0 WIDTH 80 (TEXT2) のモードではプリンク属性が付いているパターンの R#7, R#12 の色の切り替え時間指定。
GRAPHIC4~7 のモードでは偶数ページ・奇数ページの交互切り替え時間指定。

R#14: VRAM Access base address register



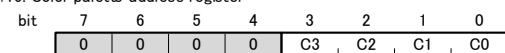
VRAMにアクセスするときに、アドレスの上位3bitをこのレジスタにセットする。
この値は、SCREEN0~3 の時には、VRAMアクセスによるアドレスインクリメントの影響を受けない。(TMS9918との互換のため)
SCREEN4以上では A13 のキャリーを受けて自動的にインクリメントする。

R#15: Status register pointer



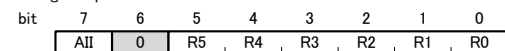
VDPのステータスレジスタ(S#0~S#9)を読みだす際に、読みだすステータスレジスタの番号を指定するレジスタである。0~9 の範囲で指定する。

R#16: Color palette address register



カラーパレットにアクセスする際に、アクセスするカラーパレット番号を指定するレジスタである。0~15 の範囲で指定する。

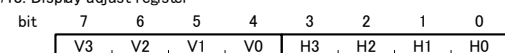
R#17: Register pointer



VDPのレジスタに対して連続アクセスするときの、レジスタ番号を指定する。
0~46 の範囲で指定する。

アクセスの際に、レジスタ番号をインクリメントするか否かを指定する。
0: オートインクリメント
1: オートインクリメント禁止

R#18: Display adjust register



水平方向の表示位置指定。-8~+7 の範囲で指定。

垂直方向の表示位置指定。-8~+7 の範囲で指定。

MSX Documents VDP

R#19: Interrupt line register

bit	7	6	5	4	3	2	1	0
	IL7	IL6	IL5	IL4	IL3	IL2	IL1	IL0

R#0 の bit4 (IE_i) が 1 の場合に、割り込みが発生するライン番号を指定する。
画面表示上のライン番号ではなく、VRAM上のライン番号を指定するため、
画面上のどのあたりで割り込みが発生するかは、R#23 に設定している内容に影響
されて上下することに注意すること。

R#20: Color burst register 1

bit	7	6	5	4	3	2	1	0
	0	0	0	0	0	0	0	0

VDPのコンボジット出力の設定レジスタ。
MSXでは、VDPのコンボジット出力は使っていないことが多いので書き換えてはならない。

R#21: Color burst register 2

bit	7	6	5	4	3	2	1	0
	0	0	1	1	1	0	1	1

VDPのコンボジット出力の設定レジスタ。
MSXでは、VDPのコンボジット出力は使っていないことが多いので書き換えてはならない。

R#22: Color burst register 3

bit	7	6	5	4	3	2	1	0
	0	0	0	0	0	1	0	1

VDPのコンボジット出力の設定レジスタ。
MSXでは、VDPのコンボジット出力は使っていないことが多いので書き換えてはならない。

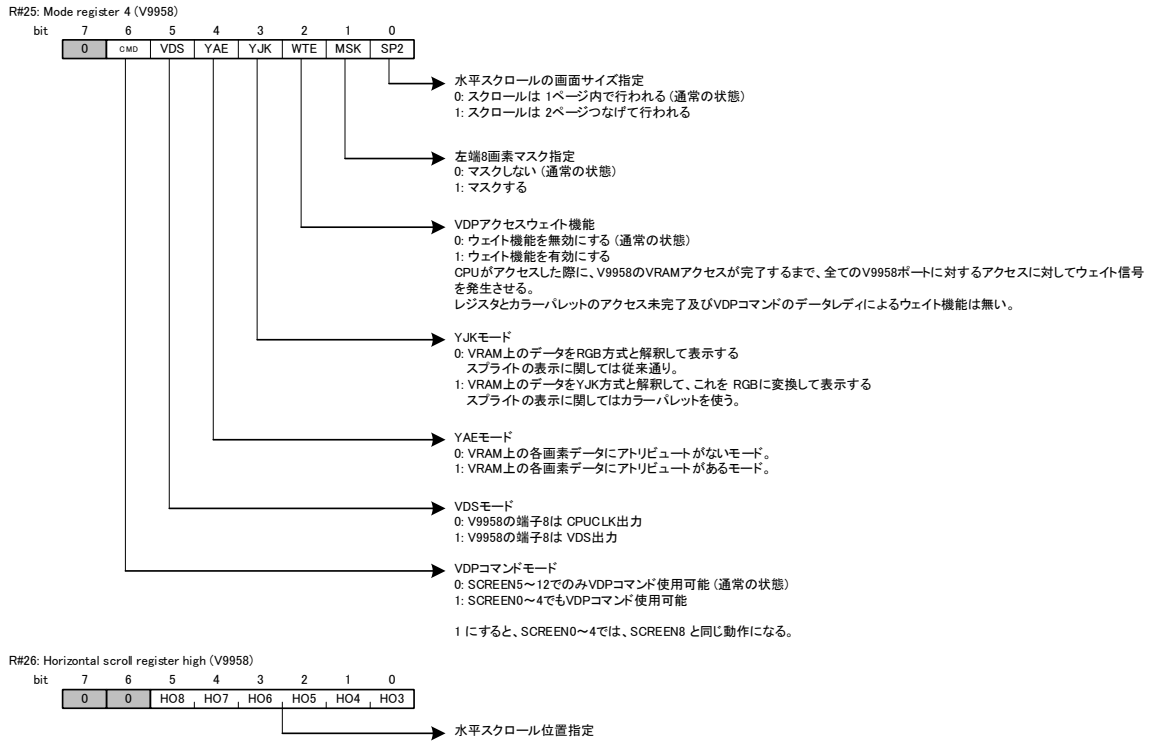
R#23: Display offset register

bit	7	6	5	4	3	2	1	0
	DO7	DO6	DO5	DO4	DO3	DO2	DO1	DO0

表示開始のライン番号を指定する。

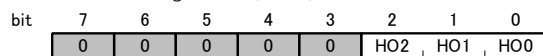
R#24: 欠番

MSX Documents VDP



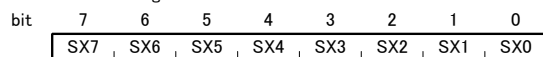
MSX Documents VDP

R#27: Horizontal scroll register low (V9958)



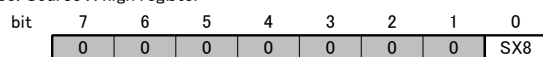
→ 水平スクロール位置指定

R#32: Source X low register



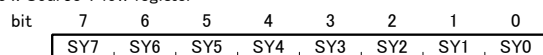
→ VDPコマンドにおける転送元基準点のX座標 下位8bit

R#33: Source X high register



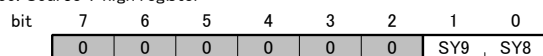
→ VDPコマンドにおける転送元基準点のX座標 上位1bit

R#34: Source Y low register



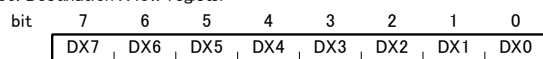
→ VDPコマンドにおける転送元基準点のY座標 下位8bit

R#35: Source Y high register



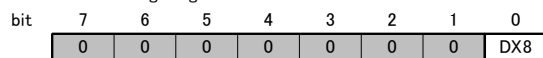
→ VDPコマンドにおける転送元基準点のY座標 上位2bit

R#36: Destination X low register



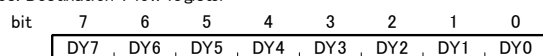
→ VDPコマンドにおける転送先基準点のX座標 下位8bit

R#37: Destination X high register



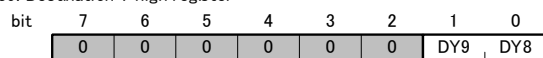
→ VDPコマンドにおける転送先基準点のX座標 上位1bit

R#38: Destination Y low register



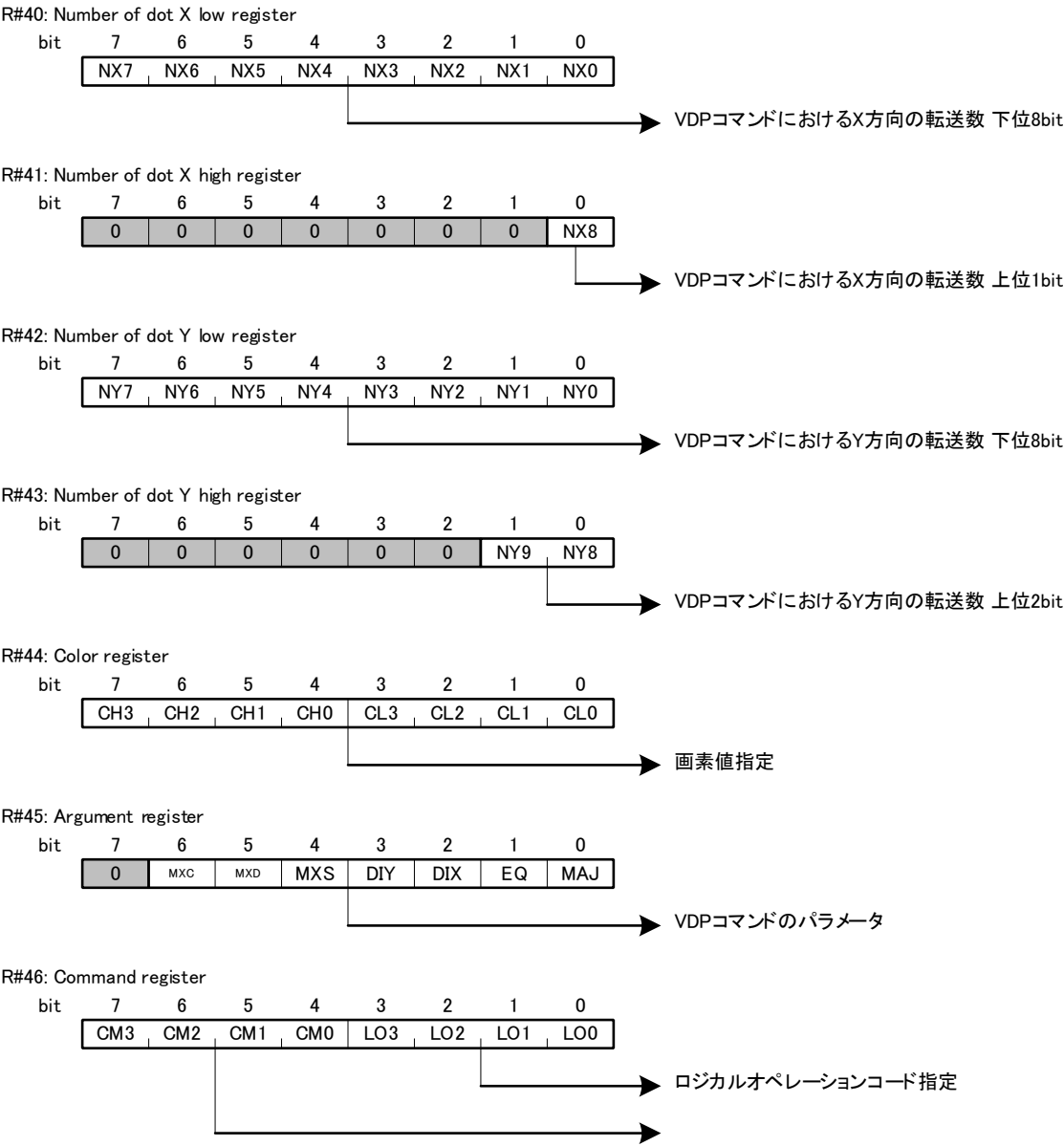
→ VDPコマンドにおける転送先基準点のY座標 下位8bit

R#39: Destination Y high register

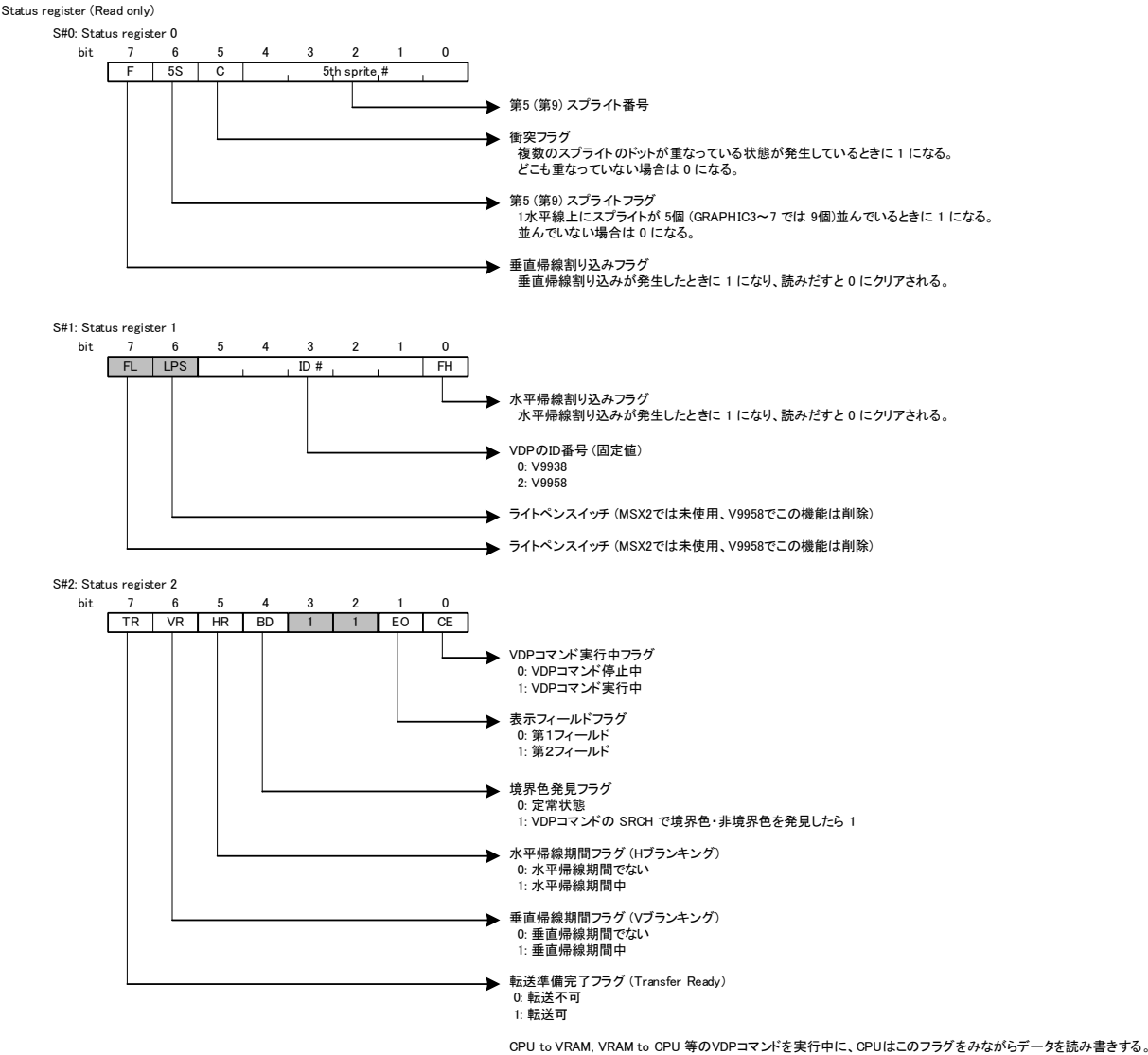


→ VDPコマンドにおける転送先基準点のY座標 上位2bit

MSX Documents VDP

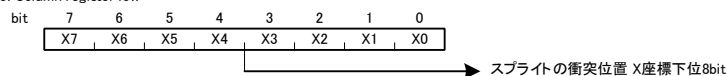


付録. ステータスレジスタ

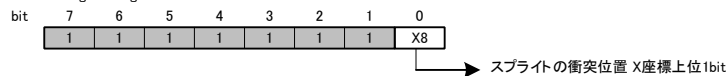


MSX Documents VDP

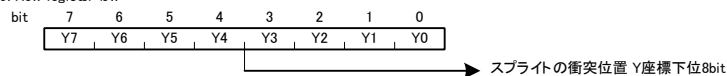
S#3: Column register low



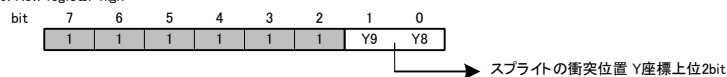
S#4: Column register high



S#5: Row register low



S#6: Row register high



復S#3, 4, 5, 6 は、スプライトの衝突座標・ライトペンの検出座標・マウスの検出座標を示す。MSX2ではライトペン/マウスは接続できないので、実質スプライトの衝突座標のみの利用になる。

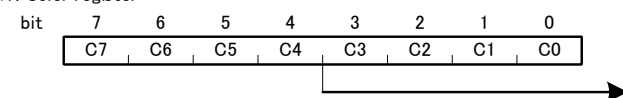
復S#3, 4, 5, 6 は、S#5 を読みだすとリセットされるので読みだし順序に注意。

復Y9 は、実質常に0。ライトペン/マウスの座標の時にEOフラグ(偶数・奇数のどちらのページか)として働くビット。

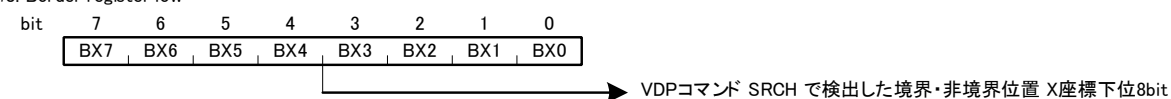
復X座標, Y座標ともに実際の衝突位置を (XC, YC) とした場合に、下記のオフセットが加算されているので注意。

(X, Y) = (XC+12, YC+8)

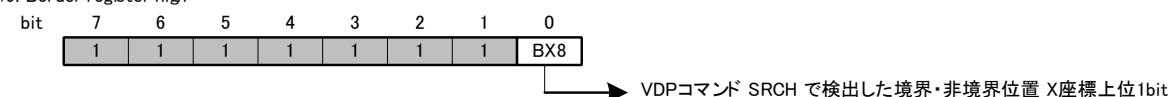
S#7: Color register



S#8: Border register low



S#9: Border register high



付録. 参考文献

名称	アドレス
MSX テクハン Wiki	http://ngs.no.coocan.jp/doc/wiki.cgi/TechHan
MSX データパック Wiki	http://ngs.no.coocan.jp/doc/wiki.cgi/datapack
MSX Assembly Page	http://map.grauw.nl/
TEXAS INSTRUMENTS 9900	-
TMS9918A/TMS9928A/TMS9929A Video Display Processors	-
ASCII/NIHON GAKKI V9938 MSX-VIDEO Technical Data Book	-
YAMAHA V9958 MSX-VIDEO Technical Data Book	-

本書は、HRA!が自身の知識・ネット上の情報・MSX 実機での実験結果などを収集して記述したものです。MSX 公式の資料ではありませんので、誤記があるかもしれません。HRA!は、内容について一切保証しません。

2021 年 6 月 5 日 HRA!