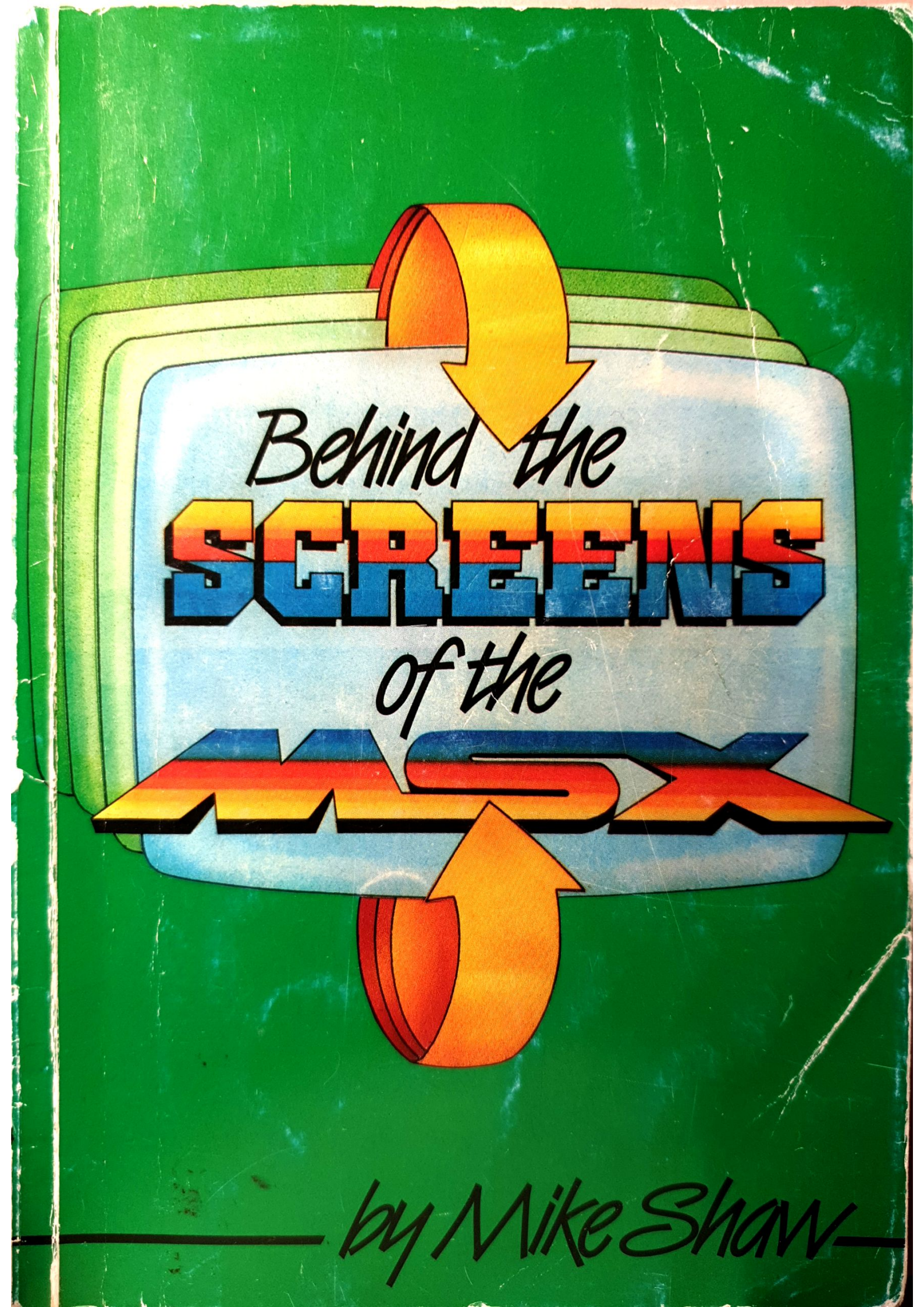

Behind the
SCREENS
of the
MSX
by Mike Shaw

# BEHIND THE SCREENS of the MSX

Published by KUMA COMPUTERS LTD.

Copyright (C) 1984 Mike Shaw

ALL RIGHTS RESERVED

No part of this Book may be reproduced by any means without prior permission of the publisher.

The only exceptions are the entry of programs contained herein onto a computer for the sole use of the owner of this book.

ISBN 07457 0008 X

First Published 1984

Published by

KUMA COMPUTERS LTD
Unit 12, Horseshoe Park
Horseshoe Road, Pangbourne
Berkshire RG8 7JW
Tel: 07357 4335

Written using WDPRO V2.37 software

# PREFACE

One of the most important features of practically every computer program is the screen display. The MSX, like so many other home computers today, has more than one screen mode of operation, so enabling programmers to choose the type of display most suited to their purposes.

This Book is all about that part of the MSX that produces the picture on the screen - the Video Display Processor, or 'VDP' for short. It discusses how the VDP works, how it is used by MSX Computers, and how the programmer can use its versatility in MSX Basic or machine code programs.

It has been written to satisfy both the newcomer to home computing and the more experienced programmer alike, giving the newcomer a deeper insight into what's going on 'behind the screens', and giving the more experienced programmer the information needed to save on midnight oil.

My appreciation and thanks go to Kuma Computers and to Sony for the loan of equipment, to Texas Instruments - manufacturers of the VDP used in MSX Computers - for allowing use of reference material in the preparation of this book (without it, we'd all probably be floundering in the dark!), and to the Microsoft Corporation, for information and data on the MSX' System.

Mike Shaw

November 1984

MSX is a Registered Trademark
of the Microsoft Corporation.

CONTENTS

LIST OF ILLUSTRATIONS

SECTION 1

# THE VDP EXPLAINED

In learning how to use any system to its best advantage, it often helps to understand first how the system actually works. This first Section, therefore, is devoted to a discussion on how the Video Display Processor operates in general, and within the MSX in particular.

## 1.1 PUTTING YOU IN THE PICTURE

### 1.1.1 The Computer Within Your Computer

Let us start by having a general look at the Video Display Processor, or VDP as we shall call it from now on. In many respects, this is like another small computer nestling inside your MSX. It is there to control the way anything and everything appears on the screen of your TV or Monitor. Like any computer, it needs an input of instructions to tell it what to do, and data for the instructions to act on. Given these, it provides an output in a suitable form for the screen display.

When using BASIC, the VDP gets its instructions from the routines that are resident in the ROM of your MSX. Machine code programmers can access the VDP via two Ports, or by calling ROM routines (the easiest and best way). The data for the VDP to act on is held in its own area of RAM - usually 16k bytes. This RAM is quite separate from the main memory RAM, and to differentiate it, we shall call it 'VRAM' - short for Video RAM.

When a Screen Mode is selected using BASIC, routines in ROM are called up to automatically fill certain areas of VRAM with specific data. In Screen Modes 0 and 1, for example, the complete character set is entered into a part of VRAM. MSX BASIC also allows you to enter data into the VRAM, either indirectly, using the variable 'SPRITE$', or directly using VPOKE.

For those who wish to write their programs in machine code, the ROM routines in the MSX can, of course, be used to enter data into VRAM (which is easier than writing your own routines!), and to let you give the VDP its instructions.

### 1.1.2 Four Modes Of Operation

The VDP provides the MSX with powerful visual display capabilities. For a start, it can be set to any one of four completely different modes of operation. These are:

(a) Text Mode, 40 x 24 (Screen 0)

(b) Text Mode, 32 x 24 (Screen 1)

(c) Hi-Resolution Mode (Screen 2)

(d) Multi-colour Mode (Screen 3)

Each mode has specific characteristics in the way information is presented to the screen. For example in the 40 x 24 Text Mode, only two colours can be used on the screen at a time, whereas in the other Modes, all the colours can be displayed.

As another example, in the 32 x 24 Text Mode, one VRAM 'Colour Address' controls the foreground and background colours for a sequence of eight characters, while in the Hi-Resolution Mode, one Colour Address controls just one of the eight horizontal 'lines' that go to make up a complete character position.

Details of these characteristics will be given more fully in the respective Mode Sections of this book: sufficient to know at this stage that each Screen Mode operates in a different manner.

### 1.1.3 A Splash Of Colour

The VDP gives you a palette of 15 colours, and 'transparent'. Unlike some computer systems which only 'map' the colour information to screen addresses (the Sharp MZ700, for example), MSX BASIC uses the VDP to either define the colour of a specific character (e.g. as in Screen 1), or to 'map' to a Screen address (e.g. as in Screen 2).

So if in Screen Mode 1 character 65 (the letter 'A') is coloured, say, red on yellow, then wherever that character appears it will be red on yellow. In Screen Mode 2, on the other hand, MSX BASIC arranges the VDP so that any specific point on the screen can be given a required foreground or background colour.

However, it is possible to achieve different colour schemes for the one character in Screen Mode 1, just as it is possible to define character colours in Screen Mode 2, by changing the data held in VRAM. This is one of the beauties of the VDP: you don't have to use it the way the MSX sets it up if you don't want to. We shall be examining ways to re-organise the VDP later on in the book.

The colours available are:

| | |
|---|---|
| 0 | Transparent (Border colour) |
| 1 | Black |
| 2 | Medium Green |
| 3 | Light Green |
| 4 | Dark Blue |
| 5 | Light Blue |
| 6 | Dark Red |
| 7 | Cyan |
| 8 | Medium Red |
| 9 | Light Red |
| 10 | Dark Yellow |
| 11 | Light Yellow |
| 12 | Dark Green |
| 13 | Magenta |
| 14 | Grey |
| 15 | White |

### 1.1.4 Sprites - The Animated Characters

Another of the features of the VDP is its ability to provide Sprites. A Sprite is a kind of animated character, controlled by the VDP in a different way to ordinary characters.

The VDP lets you have Sprites in one of four different forms - though only one form can be present on the screen at any given time. The Sprite pattern can occupy the space of one character or a block of four characters: both of these forms can be 'magnified' to occupy the space of a block of four characters or 16 characters respectively.

Sprites can be moved pixel by pixel in any direction across the screen to give smooth movement, and they can be made to pass in front of or behind each other, to give a 3D effect.

The VDP can detect when the patterns of any Sprites coincide - that is, occupy the same pixel area on the screen - and this feature is, quite naturally, used to provide useful commands in MSX BASIC. Detecting a 'collision' of Sprites is an essential feature of many Arcade type games.

Up to 32 Sprites can be placed on the screen (though only four can occupy one horizontal line at a time), and you can define a maximum of 64 or 256 different Sprite patterns, depending on the Sprite size you choose.

### 1.1.5 Screen Management

The VDP constantly 'refreshes' the screen display, examining the instructions it is given and the data it holds about 50 times a second. This refreshing process is quite invisible, and takes place virtually independently of any 'communication' that may be going on between the VDP and the MSX central processor unit.

Machine code programmers will be interested to know that the VDP produces an interrupt signal at the end of every screen refresh operation. The signal is detected by MSX ROM routines - and is accessible via a 'hook', which allows the programmer to insert his own interrupt-driven routines.

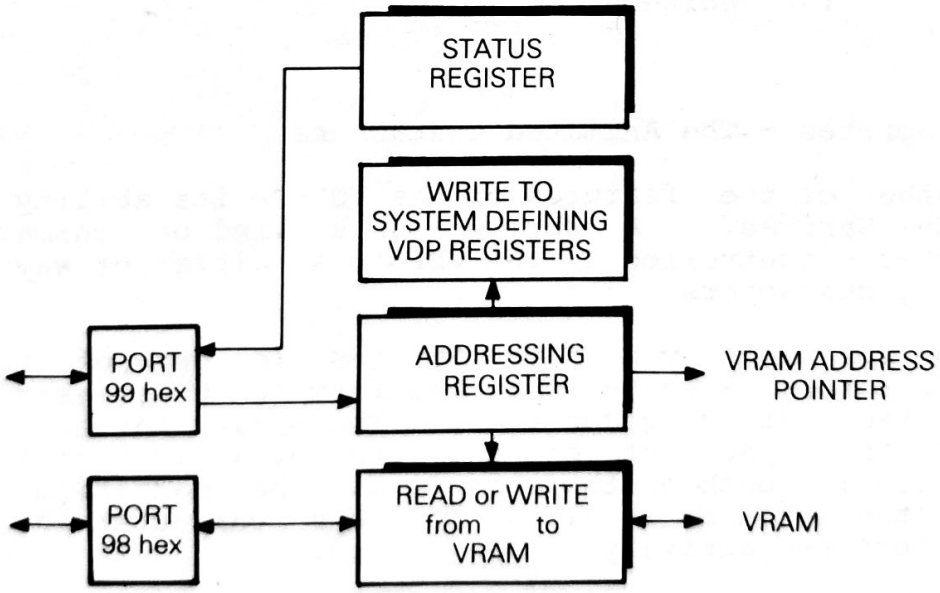



Figure 1 Input/Output to VDP

## 1.2 INSIDE THE VDP

### 1.2.1 The Ins And Outs

A greatly simplified block diagram of the interfacing between the MSX and the VDP is shown in Fig. 1. As you can see, the MSX 'communicates' with the VDP through Ports 99 hex and 98 hex. Note that these are not RAM addresses: Ports are like doorways to and from Input/Output devices. MSX BASIC lets you access Ports directly with the 'OUT' and 'INP' statements, but this will rarely be necessary in a BASIC program.

Through Ports 99H and 98H, the MSX can instruct the VDP to perform one of four operations. These are:

(a) Write bytes of data to VRAM

(b) Read bytes of data from VRAM

(c) Write data to one of the eight VDP System Defining Registers

(d) Read data from the VDP status Register.

For the first two of these operations, two bytes of data are routed separately through Port 99H to the Addressing Register. These bytes, along with information carried on three control lines (not shown in the Figure), set up the VDP address for the required operation. The Reading or Writing of data from or to VRAM is then achieved through Port 98H.

Before we examine these operations in closer detail, let us first look at the nature of the VRAM addresses. The maximum VRAM area that the VDP can deal with is 16k or, put another way, the maximum possible address in VRAM is 3FFF hex. In binary, this is 00111111 11111111: thus, as you can see, the top two bits of the High byte - bits 14 and 15 - will always be zeroes as far as the address is concerned, and only bits 0 to 13 have any addressing significance.

Bits 14 and 15 of the address data (i.e. bits 6 and 7 of the 'High' byte) are used by the VDP, along with information on the control lines, to determine the nature of the operation to be performed. Armed with this information, we can now take a look at the four operations in closer detail.

Writing to VRAM

When writing data to VRAM, first the required VRAM address is input through Port 99 hex, one byte at a time (in the order Low byte then High byte), and then the data to be written to that address is input to Port 98 hex. For a 'Write' operation, the High byte of the address data must have its bit 6 set to a '1'.

The address in VRAM having been set up for a write operation by the first two bytes, the data itself is then written to that address via Port 98 hex. There needs to be a very short delay between the input of the address and the input of the data - in MSX routines, this delay is achieved by performing an 'EX (SP),HL' twice.

You are about to ask a question...'Does this mean that for every byte of data transferred into VRAM it is necessary to enter first a two-byte address?'.

Glad you asked. The answer is no. The VDP is a bit cleverer than that: once the data byte has been transferred to VRAM, the address pointed to by the Addressing Register is automatically incremented to the next address. Thus, if say 100 bytes of data are to be transferred into sequential VRAM addresses, it is necessary to set up only the first (lowest) address, and then pump in the 100 data bytes through Port 98 hex.

It's worth mentioning the timing at this point. When the address is initially set up, the first data byte is transferred to that address from Port 98 hex in 2 to 3 microseconds: for subsequent sequential addresses, the VDP can take up to 8 microseconds to make the transfer. This is because the VDP must wait for a 'window' in its screen refreshing operations.

There are two occasions when the wait for an access window is effectively zero: when the screen has been 'disabled' (the entire screen will then only be showing the border color), and when the VDP is in its 'vertical refresh' mode. The 'vertical refresh' occurs after the active display has been refreshed, and lasts for some 4300 microseconds. It is possible to use the VDP interrupt signal - which occurs at the end of the active screen refresh - to indicate when data can be transferred to (or from) VRAM in the minimum time.

Programmers using BASIC can write data to the VRAM area by using the VPOKE command. Note that MSX BASIC differentiates between VPOKE, which is for Video Ram addresses, and POKE, which is for RAM addresses.

Unfortunately, using the VPOKE command means data has to be transferred one byte at a time. Where a considerable amount of data has to be transferred to VRAM (such as when defining a new character set), the

transfer can be speeded up by incorporating a machine code routine to be placed in 'safe' memory by the BASIC program. But this would consume quite a lot of RAM memory space, for the machine coding data and the data itself would be in memory twice - once in the BASIC program and again in the area allocated to the machine code - as well as, ultimately, in VRAM. If timing is not too important (most people are prepared to wait while 'initialising' takes place), the best way to transfer a lot of data is probably with a FOR-NEXT loop and READ statements.

For machine code programmers, here are some useful monitor routine addresses for Writing data to VRAM:

04DH Write one data byte to VRAM

IN: Data in Register A
VRAM address in HL
OUT: AF modified

056H Fill VRAM area with one data byte

IN: Data in Register A
VRAM address in HL
Length to be filled in BC
OUT: AF, BC modified

05CH Move block of data from RAM to VRAM

IN: RAM source start address in HL
VRAM destination start address in DE
Length of block in BC
OUT: All Registers modified

The Registers referred to in these routines are, of course, the Z80 Registers. Note that the MSX routine addresses given above are simply jumps to the actual routine performing the operation: it is best to use these jump addresses wherever possible, rather than the actual routine addresses, to ensure compatability with other MSX machines.

Reading From VRAM

The operation of reading data from VRAM is very similar to the Write to VRAM operation. The required address in VRAM to be read is first input as two bytes (one at a time, in the order Low byte, High byte) to Port 99 hex: in this instance, however, bits 14 and 15 of the address must both be zeroes.

This sets up the VDP for a read operation, and points it to the appropriate address. The data at that address can then be read at Port 98 hex.

As with the Write operation, after the data byte has been read at Port 98 hex, the VRAM address pointed to by the Addressing Register is automatically incremented, so that a further 'read' from the next address can take place. In this way, a whole block of data can be read from sequential VRAM addresses, once the initial starting address has been given. The timings for a Read from VRAM are the same as for a Write.

Programmers using BASIC can access the contents of a VRAM address by the VPEEK statement. For machine code programmers, here are a couple of useful ROM routine addresses:

04AH Read one byte from VRAM

IN: VRAM address in HL
OUT: Data in A

059H Move block of data from VRAM to RAM

IN: VRAM source start address in HL
RAM destination start address in DE
Length of block in BC
OUT: All Registers modified

Write to VDP Register

The VDP has eight Registers which define the way the system is to 'operate'. These Registers and their functions are explained in Section 1.2.2: here we are going to discuss how data in the Registers can be changed.

It should perhaps be mentioned at this point that one can only write to the VDP Registers - their contents cannot be read. MSX keeps a record of what's in the VDP Registers by storing the information written to them, in addresses F3DF hex (Register 0) to F3E6 hex (Register 7). However, it will only do this when its own ROM routine is used to change the Register data.

Two data bytes are required to write to a VDP Register. The first byte carries the actual data to be written, and is input through Port 99 hex. The second byte also goes to Port 99 hex, to tell the VDP which Register the data is to be written into. The most significant bit (bit 7) of this second byte must always be a '1', while bits 3, 4, 5 and 6 must be zeroes. Bits 0, 1 and 2 carry the Register number in the usual binary manner.

Note that the data byte is written to Port 98 hex before the addressing byte - unlike the VRAM Read and Write operations, which need the data byte to be written after the address has been set up.

If it is required to re-write data to a VDP Register after a data byte has been loaded through Port 98 hex, it is necessary to Read the Status Register first, to re-initialise the VDP's logic circuits. Otherwise, the VDP will be duped into thinking that the next data byte to appear is an address. This situation could occur in interrupt-driven situations, for example, and for this reason the MSX monitor routine performing a Write to a VDP Register disables the interrupts while the writing process is carried out.

It is, therefore, adviseable to always use the monitor routine (when programming in machine code) when it is desired to change the data in a VDP Register:

```
047H  Write to VDP Register

      IN:  Register number in C
           Data in B
     OUT:  Registers AF, BC modified
```

This monitor routine also updates the data stored at the appropriate address F3DF to F3E6 hex - another good reason for using it.

BASIC programmers can also change the data held in a VDP Register, by using the 'VDP(x)' variable. For a five-minute breather, try the following program...

```
10 SCREEN 0
20 PRINT "OLD COLOUR = ";HEX$(PEEK(&HF3E6))
30 VDP(7)=&HB6
40 PRINT "NEW COLOUR = ";HEX$(PEEK( &HF3E6))
```

As you will see later, in Screen Mode 0, VDP Register 7 holds the colour information for the background and foreground colours. When run, the MSX will initialise the screen into Mode 0 - usually with a white on blue display. This is recorded in VDP Register 7 as 'F4' hex, 'F' being 15 in decimal - which is the number for 'white', and 4 being the number for 'blue'.

Then, in line 30, we use the VDP(x) variable to write new data to Register 7. The data written, 'B6' hex, represents the colours Light Yellow ('B' hex or 11 decimal) and Dark Red (6). In line 40, we re-read the storage address for VDP Register 7 to find the data has been updated.

The screen, of course, changes colour at line 30. Try putting a 'FOR I=1 to 1000:NEXT' delay at line 25, and you'll see the moment of change. You can also try changing the data input to VDP(7) in line 30 - for different screen colourings. Very exciting! (Well, it makes a break, doesn't it?).

It may seem a trivial point to make, but you cannot change the contents of a VDP Register by POKEing the desired value into the corresponding storage address (F3DF - F3E6 hex).

Read VDP Status Register

The VDP provides a Register to report on the status of certain events during its operation. It is a Read-only Register - you cannot change its contents.

Details of the information provided by this Register are given in Section 1.2.2. To read the Status Register requires nothing more than 'collecting' the data at Port 99 hex.

There is a monitor routine, at 013E hex, to read the Status Register. This routine simply does an IN A,(99H) followed by a RETurn. It should be noted, however, that reading the Status Register has the effect of clearing the interrupt flag, which means the MSX's own interrupt handling system could miss it.

Consequently, if the machine code programmer wishes to detect when an interrupt has occured, or wishes to check the contents of the Status Register, he is best advised to use one of the 'hooks' made available by the MSX routines (see Section 3.3.6).

For programmers using BASIC, the contents of the Status Register can be read using the variable 'VDP(8)' in a 'PRINT VDP(8)' or a 'V=VDP(8)' type of statement.

However, it would seem that MSX BASIC finds the answer by looking at the data held in address F3E7 hex (the Status Register store) rather than performing a separate VDP read operation. Since this store is only updated when an interrupt occurs, bit 7 - which marks the occurence of an interrupt - will always be set, and it cannot therefore be used to detect the occurence of an interrupt.

### 1.2.2 The VDP Registers

There are nine accessible Registers within the VDP, each holding one byte (8 bits) of information. Eight of the Registers define how the system is to operate - which Screen Mode is to be used, what the Sprite size is to be, which parts of VRAM are to be devoted to what functions, and so on.

These are Write only Registers - that is to say, it is not possible to read what each Register contains. Such information is, at times, very useful to know. Consequently the MSX sets up a storage area in RAM (from F3DF hex for Register 0, to F3E6 hex for Register 7) where it holds the information contained in the Write only Registers. It does this by loading the appropriate memory store with the data at the same time as it loads the VDP Register.

It is adviseable when writing to a VDP Register, therefore, to use an MSX ROM routine.

The ninth VDP Register can only be read, not written to, and it contains certain information on the status of events that take place within the VDP. Section 1.2.1 discusses how this Register can be read both from BASIC or using a ROM routine. Its contents are also held in RAM, at address F3E7 hex.

It should be pointed out, however, that the Register information stores are only updated when a VDP interrupt occurs - which is about 50 times a second, at the end of each active-screen refresh.

The nature of the data held in all nine of the VDP stores is shown in Figure 2. The first two Registers - Register 0 and Register 1 - contain information that controls the system's mode of operation and its features. Registers 2 to 6 contain values which tell the VDP's inner circuitry where to start looking in VRAM for specific screen display data.

Register 7 is used to define the border colour and, for Screen Mode 0, the screen foreground and background colour.

Before discussing the specific function of each Register in turn, let us first take a broad look at what information the VDP needs in order to place a character, in colour, on the screen, and what information it needs to place a Sprite, in colour, on the screen.

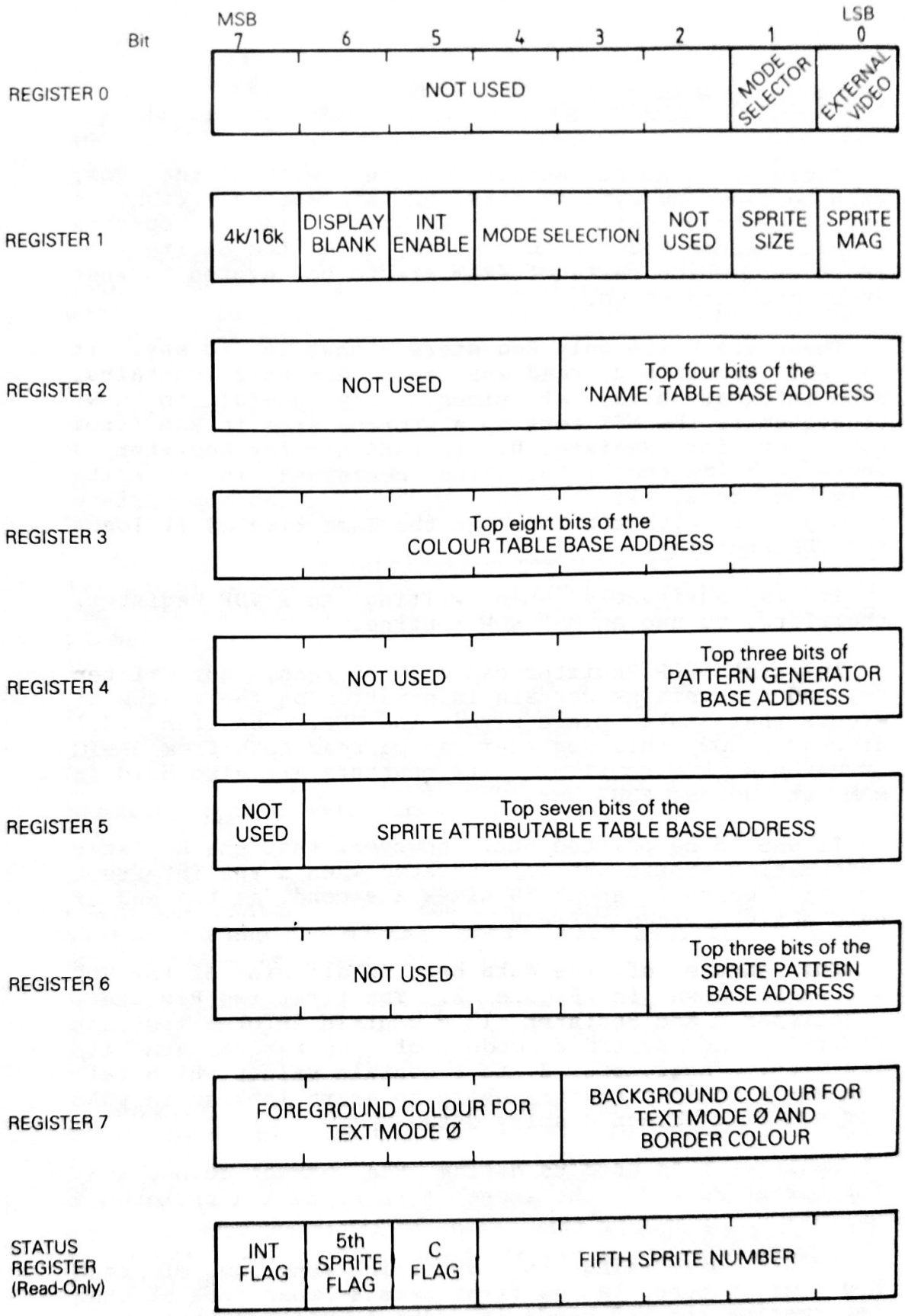



**Figure 2. The VDP Registers**

In order to place a character on the screen, the VDP needs to know

(a) The 'NAME' or 'number' of the character and its position on the screen

(b) The shape or pattern of the character

(c) The colour that is to be associated with the character.

Each Screen Mode operates in a different manner, but they all need this information in one form or another.

To find where on the screen it is to place a character, and which character is required at that place, it looks at a part of VRAM designated the 'NAME TABLE'. If the screen happens to be in the 32 X 24 'text' mode (Screen 1), then there will be 768 separate character positions (32x24), and the NAME Table will be 768 bytes long.

A value in any one of these bytes is recognised by the VDP to be a character number, or NAME. Thus, if the first byte of the NAME Table contained 41 hex - 65 decimal - then the VDP would say to itself "In the first Screen position for this Mode - the top left hand corner - I must place the character numbered 41 hex".

The VDP then looks at the PATTERN GENERATOR Table, to see what character 41 hex looks like - that's item (b) above. In the Text Mode we are discussing (Screen 1), the VDP requires 8 bytes of information to place a character on the screen (more about this in a later Section). So for a 256 character set, the VDP needs a PATTERN GENERATOR Table that's 2048 bytes long (256x8). It looks at this table to find the eight bytes associated with character number 41 hex. (If the ASCII character set has been loaded, then the eight bytes will, of course, form the letter 'A').

Then the VDP looks to the COLOUR Table, to see what colours it should make the background and the foreground for the selected character. If this information tells it that the background is to be Dark Red and the foreground is to be Yellow, then it proceeds to place the character, in Yellow on a Dark Red background, in the top left screen position.

This is a very simplistic view of the way the VDP places characters on the screen for Modes 0,1 and 2: further details are given in the relevant Sections of this book.

The process for placing a Sprite on the screen is rather different, although the VDP still needs to know where to find the necessary data.

The information required to produce a Sprite on the screen is:

(a) The Sprite´s attributes - screen location, colour, ´plane´ number

(b) The Sprite´s pattern or shape

(c) The Sprite´s size and magnification

Again, we will be discussing all of these features in detail later on. The point to be made at this stage is that Table addresses have to be set up in the VDP Registers so that it knows exactly where to look.

The beauty of the VDP is that the user can set up these addresses as required - although, as mentioned before, when operating from MSX BASIC the addresses are set to specific values, and should be changed with caution if it is intended to continue programming in BASIC.

We are now in a position to examine in more detail the function and contents of each of the VDP Registers.

> NOTE: Should you be unsure how ´bit´ positions´ (binary values) convert to hexadecimal and decimal numbers - and vice versa - please refer to Appendix A

VDP Register 0

This Register currently makes use of only the least significant two bits - bits 0 and 1. The other bits must always be zeroes (they are reserved for future developments on the VDP).

Bit 0

This is the external VDP enable/disable bit. It is only used when two VDP´s are connected in ´cascade´, to allow the display from the external VDP to be made visible on the screen. Since there is no ´external´ VDP used in the MSX, this bit should always be set to ´0´ - the ´disable external VDP´ condition. Setting it to a ´1´ makes the screen go haywire (to put it mildly!).

Try this, to see the effect:

(a) In Screen 0 or Screen 1, type in VDP(0) = 1, followed by 'CR'

(b) When you've had enough of the prancing screen, very carefully type in VDP(0)=0 (you won't see what you're typing on the screen)

(c) Make a note to be very careful about what you put into Register 0!

Bit 1

This bit is used along with bits 3 and 4 of VDP Register 1 to determine the Screen Mode: a description of the values for this bit is given under Bits 3 and 4 of Register 1

VDP Register 1

Seven of the eight bits of Register 1 select the operating options for the VDP. The remaining bit (Bit 2) is currently not used.

Bit 0

The least significant bit determines whether or not the Sprite patterns will be 'magnified'. The conditions are:

0 Selects unmagnified Sprites
1 Selects magnified Sprites

In the unmagnified condition, Sprites are represented on the screen by one pixel for each pattern position. In the magnified condition, each pattern position is represented by a block of 2x2 pixels.

Since this Bit controls the magnification for all Sprites, it is not possible to have unmagnified and magnified Sprites on the screen at the same time. Interesting effects can be achieved by setting and resetting this bit at short time intervals while Sprites are displayed on screen. This can be achieved from BASIC by using the 'VDP(1)=' construction - but be careful not to upset the other Bits in Register 1! (A sample program is given in Appendix B).

The contents of this Bit, along with the contents of Bit 1 of this Register, are set by the value of 's' in the BASIC command 'SCREEN m,s,k,cb,po'. ('m' is the Screen Mode required, 'k' is the keyclick on/off, 'cb' is the cassette baud rate and 'po' is the printer option: refer to your Owner's Manual for details of the last three functions as they are not relevant to this book).

Bits 0 and 1 are set according to the value of 's' as follows:

| 's' | Bit 1 | Bit 0 | Sprite Size |
|---|---|---|---|
| 0 | 0 | 0 | 8-byte unmagnified |
| 1 | 0 | 1 | 8-byte magnified |
| 2 | 1 | 0 | 32-byte unmagnified |
| 3 | 1 | 1 | 32-byte magnified |

Note that using the BASIC 'SCREEN' command to set Sprite size also re-initialises the screen and clears all the Sprite patterns, so it cannot be used in the same way as 'VDP(1)' to change the size of Sprites whilst on the screen.

### Bit 1

The pattern size of all Sprites is governed by this Bit. Sprite patterns can be created to cover unmagnified blocks of 8x8 pixels, or 16x16 pixels. The conditions are:

0 8-byte Sprite patterns
1 32-byte Sprite patterns

Generally speaking, care must be taken when changing this bit in a program, since the VDP looks to it to see how many bytes in the SPRITE PATTERN Table are required to create the pattern. Changing the size from 8-byte to 32-byte patterns will make the VDP pick up the first four Sprite characters to create the first 32-byte Sprite character.(See program 9, Appendix B),

The way this Bit is set by the BASIC 'SCREEN' command is explained in the discussion for Bit 0 above.

### Bit 2

This particular Bit is currently not used by the VDP.

Bits 3 and 4

These two bits, along with Bit 1 of VDP Register 0, determine the operating Screen Mode for the VDP. They are set by the BASIC 'SCREEN' command as follows:

| | Reg 1 Bit 4 | Reg 1 Bit 3 | Reg 0 Bit 1 | |
|---|---|---|---|---|
| Screen 0 | 1 | 0 | 0 | 40x24 Text Mode |
| Screen 1 | 0 | 0 | 0 | 32x24 Text Mode |
| Screen 2 | 0 | 0 | 1 | Hi-Resolution Mode |
| Screen 3 | 0 | 1 | 0 | Multicolour Mode |

It will not be necessary, normally, to select the Screen Mode by setting or resetting these Bits: if writing in BASIC, it is best to use the 'SCREEN' command, and if writing in machine code, it is best to call up the appropriate ROM routine. This is because other parameters also need to be set in order to initialise the selected screen properly - the VDP's Tables, for example.

If it is intended to re-organise the Table locations in VRAM, then of course, one could change the Screen Mode by setting these Bits accordingly. Programmers using BASIC should be careful, however, to ensure that the other Bits of VDP Registers 0 and 1 are not affected, or are properly set.

Machine code programmers have a number of ways to change the Screen Mode. To select a Mode and set it up the same way that BASIC sets it up, they can either

(a) Load (Z80) Register A with the required mode (0, 1, 2 or 3) and CALL 05FH

or

(a) CALL the appropriate 'initialising' routine direct -

06CH for Screen 0
06FH for Screen 1
072H for Screen 2
075H for Screen 3

To simply select a Mode - by appropriately setting the VDP Registers 0 and 1 - they can CALL

078H to set the VDP for Mode 0
07BH to set the VDP for Mode 1
07EH to set the VDP for Mode 2
081H to set the VDP for Mode 3

The advantage of using these routines over writing direct to the VDP Registers - using ROM routine 047H (as explained on page 9) - is that they ensure the other Bits of VDP Registers 0 and 1 are unaffected. They also record the new settings in the VDP Register 0 and 1 storage addresses, F3DF hex and F3E0 hex.

### Bit 5

This switches on or off the VDP´s own interrupt signal. The conditions are:

0 Interrupt disabled
1 Interrupt enabled

It should be noted that detection of the VDP´s interrupt signal is an integral feature of the hardware interrupt routines in the MSX: switching it off will therefore affect these routines and prevent MSX BASIC from functioning properly.

### Bit 6

This is the active screen enable/disable Bit. The conditions are:

0 Entire display shows border colour only.
1 Active display area enabled

Even though the screen is ´blank´ when disabled, data can still be entered into the VRAM areas although, obviously, you won´t see it on screen until it has been enabled again. Note that the ´blank´ colour is the same as the border colour.

### Bit 7

This Bit is set according to the amount of VRAM available to the VDP -

0 for 4k VRAMs
1 for 8/16k VRAMs

It should always be left in its initialised setting, invariably 16k on MSX machines.

VDP Register 2

The top four bits of this Register aren't used. The bottom four bits provide the information needed for the VDP to identify the start or 'Base' of the NAME Table.

The NAME Table is that area of VRAM where the VDP finds the NAME (or 'number') of a character to be placed on the screen, the address of the character's NAME within the Table identifying the position on the screen.

Figure 3 demonstrates this for Screen Mode 1, the 32x24 Text screen. Each address, starting with the NAME Table Base address, represents a screen position: thus, the 32nd byte represents the leftmost position on the screen, one row down from the top of the active area.

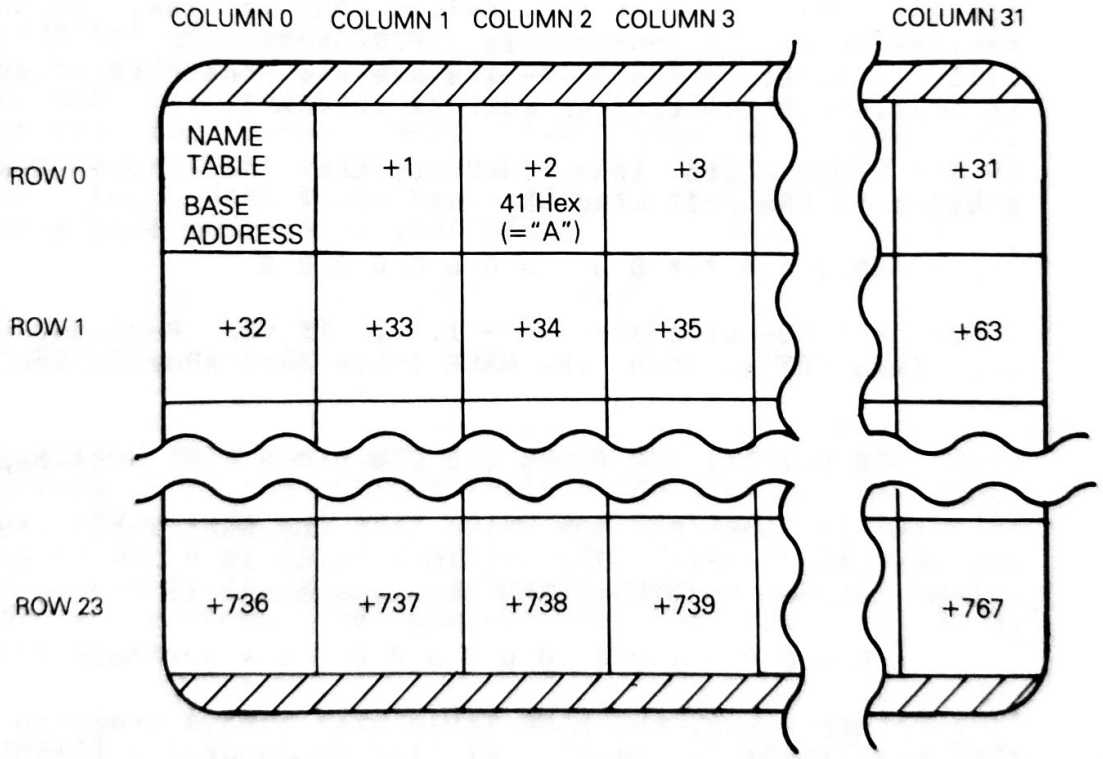



**Figure 3. Mapping the NAME Table to the screen. (Screen Mode 1)**

The VDP regards the value of the byte at each address in the Table as a character NAME, even if the value is zero. Since '0' is generally used in the Table to denote a blank space, Character 0 is generally made a blank, like the ASCII Character 20 hex (32 decimal - 'space').

In Figure 3, the NAME Table Base address plus two contains the data byte '41 hex' (decimal 65). This tells the VDP that the third column (labelled Column 2) at the top of the screen (Row 0) is to be filled with the character pattern designated by 41 hex: if ASCII is loaded into the PATTERN GENERATOR, this would, of course, be the letter 'A'.

Note that Columns and Rows are numbered from '0', the start point being the top left hand corner of the screen.

The relevant four Bits of VDP Register 2 represent the Most Significant four bits of the 14-bit NAME Table address (remember that the maximum VRAM is 16k, so the maximum possible address is 3FFF hex, or 00111111 11111111 binary - hence 14-bits are all that are needed to completely identify an address in VRAM).

The four bits thus identify the NAME Table Base address in the following way:

0 0 * * * * 0 0 0 0 0 0 0 0 0 0

If all the bits are '1' - i.e., if VDP Register 2 contained '0F', then the NAME Table Base address would be:

0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 = 3C00 hex

This is the maximum value that the NAME Table Base address can have. The minimum value is 0, and the 'step' between possible NAME Base addresses is

0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 = 400 hex

In other words, the NAME Table will always start on a 400 hex (1024 decimal, or '1k') boundary. A Table showing the NAME Table Base addresses corresponding to all possible values of VDP Register 2 is given in Appendix C.

The lower ten bits of the NAME address are formed from the 'row' and 'column' counters within the VDP: these are adjusted by the VDP as it searches through the NAME Table for bytes to be 'put on the screen' during the active display refresh period.

The NAME Table is 768 bytes long for Screen Modes 1, 2 and 3, and 960 bytes (40x24) long for Screen Mode 0.

The MSX initialises VDP Register NAME Table Base address for each Screen Mode as follows:

| Screen Mode | NAME Base Address | Register 2 Value |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 1800H | 6 |
| 2 | 1800H | 6 |
| 3 | 800H | 2 |

Because the Base address for the NAME Table can be switched by simply resetting the value in VDP Register 2, it is possible to switch from one screen display to another fairly quickly.

This is demonstrated in Program 3 of Appendix B, which loads VRAM addresses from 3C00 hex, then switches the NAME Base address back and forth between its BASIC setting for Screen Mode 1 (1800 hex) and 3C00 hex. (Notice that the values given to VDP Register 2 are 0F hex, for a NAME Table Base address of 3C00 hex, and 6 for a Base address of 1800 hex.)

VDP Register 3

All of this Register is used to define the Base address for the COLOUR Table - the area in VRAM where the VDP finds out what colours a particular character should be when it is operating in Screen Modes 1 and 2. No COLOUR Table is used in Screen Modes 0 and 3, and so for these Modes the content of VDP Register 3 is irrelevant.

In Mode 1, the eight bits of Register 3 form the most significant eight bits of the 14-bit COLOUR Table Base address. It thus defines the address as follows:

0 0 * * * * * * * * 0 0 0 0 0 0

The maximum value that Register 3 can hold is FF hex - 11111111 binary (255 decimal). Consequently, the maximum value the COLOUR Table Base address can have is

0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 = 3FC0 hex

The minimum value is 0, and the 'step' between possible COLOUR Table Base address locations is

0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 = 40 hex

Thus the COLOUR Table Base address can be set on any '40 hex' boundary in VRAM, from 0 to 3FC0 hex. The COLOUR Table Base addresses for all possible values of VDP Register 3 are given in Appendix C.

In Mode 2, the location of the COLOUR Table is determined by the top bit only of VDP Register 3: all the other bits are set to '1' (but see Section 6.3.4). The Table can consequently be located only at 0 or 2000 hex in VRAM.

The MSX initialises the COLOUR Table Base address at 2000 hex (8192 decimal) for Screen Modes 1 and 2.

The way the VDP selects the colours to be associated with a character pattern is different for every Screen Mode: for details of the way colours are produced, therefore, please refer to the appropriate MODE Section of this book.

VDP Register 4

The least significant three bits of this Register determine the Base address of the PATTERN GENERATOR: the top five bits are not used. The three relevant bits form the most significant three bits of the 14-bit PATTERN GENERATOR Base address:

0 0 * * * 0 0 0 0 0 0 0 0 0 0 0

The maximum value for these three bits is 7, or '111' binary, hence the maximum value for the PATTERN GENERATOR Base address is:

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 = 3800 hex

The miminum value is 0, and the 'step' between possible Base addresses is:

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 = 800 hex

A Table of the PATTERN GENERATOR Base addresses corresponding to each value of VDP Register 4 is given in Appendix C.

The PATTERN GENERATOR is where the VDP looks to find the pattern associated with a specific character NAME. The complete 14-bit address is derived for Modes 0,1 and 3 as shown in Figure 4.

Except for Mode 2, the top three bytes of the VRAM address are obtained from Register 4, as just described. The next eight bytes are derived from the Character's NAME - with values from 0 to 255 (0 - FF hex). The last three bytes, which can have a decimal value of 0 to 7, determine the row number within the specific pattern set, and in the Text Modes are incremented from 0 to 7 by the VDP in the process of placing the character on the screen. For Mode 2 only Bit 2 defines the address, which can be at 0 or 2000 hex, bits 0 and 1 being set to '1'. (But see Section 6.3.4)

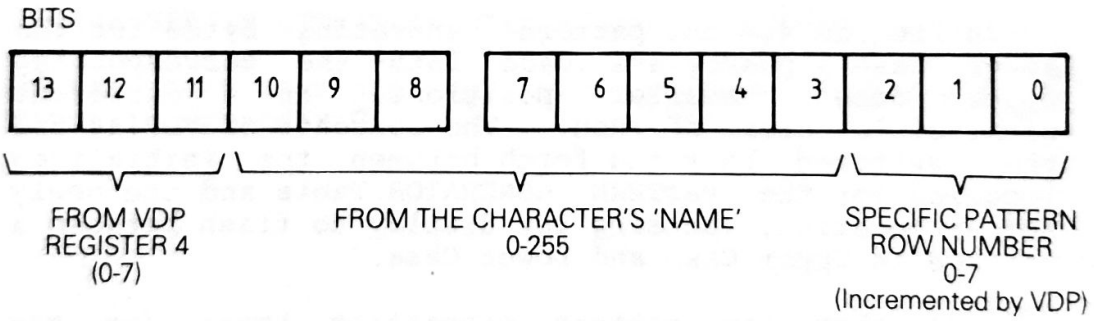



**Figure 4. Derivation of the PATTERN GENERATOR address. (except Mode 2)**

In the two 'Text' Modes (Screens 0 and 1), each character requires eight bytes to define its pattern, and so for a set of 256 characters, the PATTERN GENERATOR Table will be 2048 bytes long (256x8). In the Hi-Resolution Mode (Screen 2), 758 different characters can be specified, so the Table will be 6144 bytes long. In the Multicolour Mode (Screen 3), the PATTERN GENERATOR Table is used to define colour blocks rather than just patterns, eight bytes being used for each 'colour character' name. Consequently for a full set of 256 'colour characters', this Mode requires a Table 2048 bytes long. The MSX initialises PATTERN GENERATOR Base addresses as follows:

| Screen Mode | PAT GEN. Base Address | Register 4 Value |
|---|---|---|
| 0 | 800H | 1 |
| 1 | 0 | 0 |
| 2 | 0 | 0 |
| 3 | 0 | 0 |

As with the NAME Table, it is possible to switch from one PATTERN GENERATOR set to another by resetting the contents of Register 4. This is demonstrated in Program 4 of Appendix B.

In Program 4, the pattern generating bytes for the lower case alphabet are loaded into the corresponding upper case character positions, in a different permissible area of VRAM. The contents of VDP(4) are then switched back and forth between the initialised location for the PATTERN GENERATOR Table and the newly set up location, causing the display to flash between a message in Upper Case and Lower Case.

Note that the pattern generating bytes for the character set used by the MSX can be found in the 2048 bytes starting from address 1BBF hex (7103 decimal) in ROM. As each character's pattern takes 8 bytes, it is easy to deduce the starting byte for any particular character. Thus, the eight pattern bytes for character 97 (61 hex = 'a') start at decimal address 7879 (7103+(97*8)), or 1EC7 hex.

The area used for demonstration Program 4 is that normally reserved by the MSX for Sprite Patterns. This demonstrates a further point about the VDP: one can use 'areas' allocated to other functions, provided that the same VRAM addresses are NOT being used simultaneously for both functions. Thus, in the example Program, provided that Sprite Patterns do not occupy more than the first 520 bytes of the SPRITE PATTERN Table, both Sprite Patterns and a second character set (Character numbers 41 hex or 65 decimal upwards) can fill the VRAM block normally allocated entirely to Sprite Patterns.

VDP Register 5

The VDP looks to this Register to find the Base address for the SPRITE ATTRIBUTE Table. The most significant bit is not used: the rest form the top seven bits of the 14-bit address, thus:

0 0 * * * * * * * 0 0 0 0 0 0 0

The maximum value that Register 5 can hold is 7F hex (1111111 binary), and so the maximum value for the SPRITE GENERATOR Base address is:

0 0 1 1 1 1 1 1 1 0 0 0 0 0 0 0 = 3F80 hex

The minimum value is 0, and the 'step' between possible Base addresses is:

0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 = 80 hex

The SPRITE ATTRIBUTE Table can thus start on any 80 hex boundary (128 decimal) from 0 to 3F80 hex: a Table of the start addresses for every value of Register 5 is given in Appendix C.

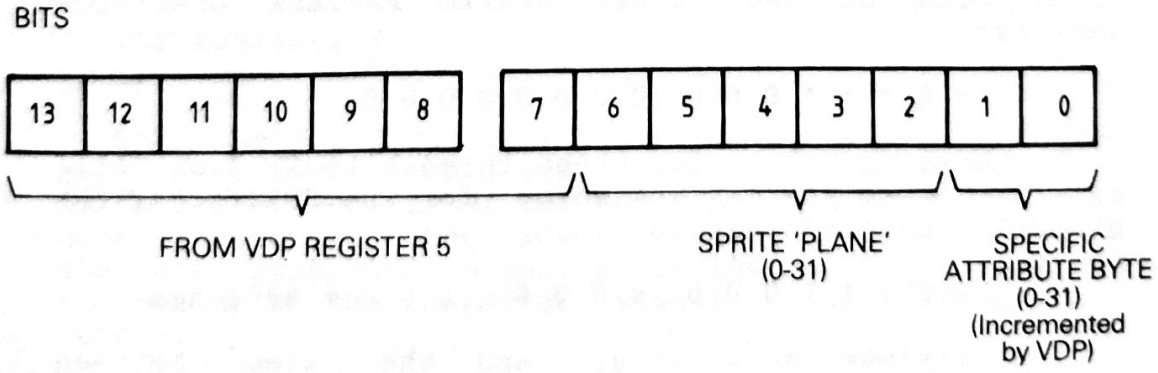



Figure 5. Derivation of the SPRITE ATTRIBUTE address

Figure 5 shows how the complete 14-bit address for the SPRITE ATTRIBUTE Table is obtained by the VDP. The top seven bits are derived from Register 5, as just described. The next five bits - which can provide values from 0 to 31 - are obtained from the Sprite Plane. The last two bits determine which byte in the set of four the VDP is to examine in the process of placing a Sprite on the screen, and are incremented from 0 to 3 during that process.

The MSX initialises the SPRITE ATTRIBUTE Base address to 1B00 hex (6912 decimal) for Screen Modes 1, 2 and 3, and so the value in VDP Register 5 will be 36 hex (54 decimal) in each case. Sprites are not possible in the Text Screen Mode 0.

Each entry in the SPRITE ATTRIBUTE Table is four bytes long, and there is one 4-byte entry for each of the 32 possible Sprite 'planes'. Thus the Table is always regarded by the VDP to be 128 bytes (32*4) long.

The VDP looks to the four data bytes in a SPRITE ATTRIBUTE entry to find where on the screen the Sprite should be placed, which 'plane' it is to be on, and what its colour is to be: these details are explained more fully in the Section on Sprites.

VDP Register 6

The lower three bits of this Register determine the Base address for the SPRITE PATTERN GENERATOR. The top five bits are not used. The three bits form the top three bits of the 14-bit SPRITE PATTERN GENERATOR address:

0 0 * * * 0 0 0 0 0 0 0 0 0 0

The maximum value for these three bits is 7 or '111' binary, hence the maximum value for the SPRITE PATTERN GENERATOR Base address is:

0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 = 3800 hex

The minimum value is 0, and the 'step' between possible Base addresses is:

0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 = 800 hex

A Table of the SPRITE PATTERN GENERATOR addresses for each value of VDP Register 6 is given in Appendix C.

The MSX initialises the SPRITE PATTERN GENERATOR Base address to 3800 hex (14336 decimal) for Screens 1,2 and

3, and so the contents of VDP Register 6 will be '7' for all of these Screen Modes.

The SPRITE PATTERN GENERATOR is where the VDP looks to find the pattern for a specific Sprite 'NAME' - very similar in fact, to the way the VDP looks for a character pattern according to the character's NAME.

The Table is 2048 bytes long - the same, in fact, as the character PATTERN GENERATOR Table. Indeed, you've probably noticed that both these Tables also have the same range of Base start addresses. The maximum number of Sprite patterns possible is 256 when the Sprite size is 8x8 pixels (8-byte patterns), and 64 when the Sprite size is 16x16 pixels (32-byte patterns).

One would not normally set the two PATTERN GENERATOR Tables to the same Base address - obviously! - but it is possible to do so, if you want. Program 5 in Appendix B demonstrates this possibility. It simply resets the Base address for the SPRITE PATTERN GENERATOR to that used for the character PATTERN GENERATOR. So if you want character patterns to become Sprites - that's one way to do it!.

The program also shows the 32-byte Sprite size - where 4 blocks of 8 bytes are required to form the Sprite pattern. When you try this Program, you'll notice that four blocks of 8-byte character patterns are also placed on screen - since in the 16x16 pixel Sprite size, the VDP expects to find 32 bytes defining the Sprite pattern. The on-screen positions of these characters demonstrates how the 32-byte Sprite is formed. This is discussed more fully in the Sprite section.

VDP Register 7

This Register serves two purposes. For all Screen Modes, the lower nibble (four bits) defines the 'border' or 'backdrop' colour of the screen. For the 40x24 Text Mode (Screen 0), the lower nibble also defines the character backgound colour, while the top nibble defines the colour of the character itself - the 'foreground' colour.

Each nibble can have a value of 0 to 15. Thus for Screen Modes 1,2 and 3, all 15 colours and 'transparent' are available for the border or backdrop colour, irrespective of the colours given to the characters themselves. In Screen Mode 0, the VDP looks to Register 7 for the character colours as well as the border colour: hence only two colours can be displayed on the screen at a time when in Mode 0.

The value held by the top or lower nibble corresponds to the colour's number (see page 3). If one uses hexadecimal notation, therefore, the colours for screen 1 can be specified thus:

```
VDP(7)=&HFB
```

where 'F' is the Foreground colour and 'B' is the background colour. In this instance, 'F' hex (15 decimal) is the number for the colour White, and 'B' hex (11 decimal) is the number for the colour Light Yellow. So entering VDP(7)=&HFB would, in Screen Mode 0, give White characters on a Light Yellow background, with a Light Yellow border...not the best of combinations!

## VDP Register 8 (Status)

This is a Read-Only Register that's used to report on specific events within the VDP. It is read regularly by MSX ROM routines, and can also be read from BASIC, by using for example a 'PRINT VDP(8)' statement.

### Bit 7

The most significant bit is the Interrupt Status Flag. It is set to a '1' at the end of the raster scan at the last line of the active display. It is reset to '0' after the Status Register is read.

The MSX reads the Status Register frame by frame, so that the interrupt flag is cleared and made ready for being set again at the end of the next refresh period. Provided that the Interrupt Enable bit (Bit 5) of VDP Register 1 is set to '1', the VDP will produce an interrupt signal whenever this bit of the Status Register is '1'. If the Interrupt Enable bit of VDP Register 1 is not set to '1', then interrupt signals will not be produced - and hardware interrupt routines within the MSX will not occur: that means the keyboard is effectively 'switched off' - with no way to redress the situation except by a 'reset'.

### Bit 6

The VDP won't allow more than four Sprites to occupy one horizontal pixel line at a time. (Program 6 demonstrates this). If five or more Sprites occur on a pixel line, this Bit of the Status Register is set to a '1'. It is reset whenever the Status Register is read - but will be set again if five or more Sprites still occupy a line.

### Bit 5

The VDP is able to detect when the pattern bits of any two or more Sprites coincide at the same pixel on the screen. When such an event occurs, this bit of the Status Register is set. It is reset when there are no 'coincidences'.

BASIC tests this Bit when the SPRITE ON function has been invoked during the running of a program: if it detects a coincidence, it passes program running control to the 'ON SPRITE GOSUB' line number. Whilst the subroutine is being executed, further tests for Sprite coincidence are suspended. Testing is re-introduced on the Return from the subroutine, unless within the routine there is a specific 'SPRITE OFF' statement.

If SPRITE STOP has been invoked in a BASIC program, testing continues to take place, but no action is taken until a 'SPRITE ON' is invoked: any coincidence that may have occured is remembered and acted upon.

If SPRITE OFF is invoked, testing does not take place. However, the programmer can make the test for himself, if he wishes, by examining this bit of the Status Register (preferably by examining address F3E7 hex): the VDP always reports a coincidence of Sprites for as long as the coincidence occurs.

### Bits 0 to 4

When five or more Sprites occur on a horizontal pixel line - as detected by Bit 6 of this Register - the number of the 'plane' carrying the fifth Sprite is placed in Bits 0 to 4. The content of these Bits is valid and meaningful ONLY when Bit 6 is set.

Thus, a program segment such as

```
IF VDP(8) AND &H40  THEN FS = VDP(8) AND &H1F
```

will test Bit 6 of the Status Register (ANDing with &H40 masks out all but Bit 6), and if it is '1', will make variable FS equal to the 'plane' number of the fifth Sprite - by masking out the unwanted top 3 Bits.

## 1.3 FORMING THE DISPLAY

### 1.3.1 Building Up The Picture

There is, obviously, considerably more within the VDP than described in Section 1.2. Apart from numerous other registers, counters and controllers, there is all the logic and colour decoding necessary to produce the signals for the display screen.

A description of such circuitry is beyond the scope of this book, since the user has no control over it other than as so far described. What is important, however, is the way that the picture is built up on the screen, and it is this aspect of the VDP's operation that we are going to examine in this Section.

Unlike earlier home computers, which treated the screen as one flat plane, the VDP is capable of producing picture elements at different 'depths' on the screen. The screen can, in fact, be regarded as a series of transparent sheets laid one over the other, with each carrying a part of the overall display (see Figure 6). It is in this way that your computer is able to create the three-dimensional effect of one object passing behind or in front of another.

### 1.3.2 The 'Backdrop'

Right at the very 'back' of the display is the layer known as the 'Backdrop'. This covers the entire screen - not just the 'active area' that you can create pictures on. Its single colour is determined by the lower four bits of VDP Register 7.

The background colour of the active display area can be different from the backdrop colour: when it is, the backdrop provides a distinct 'border' colour to the overall picture at the top and bottom (the sides are sometimes concealed from view on European system television sets). Indeed, in the BASIC 'COLOR x,y,z' statement, the 'z' parameter allows you to determine the 'border' colour. The colour specified for 'z' is loaded into the lower nibble of VDP Register 7.

In Screen Mode 0, the backdrop colour also becomes the background colour for the active display area - consequently in this Mode the entire screen has but one background colour.

When 'transparent' has been specified as a colour, the screen becomes transparent through to the backdrop. Thus, entering 'SCREEN 1: COLOR 15,4,7' would produce a backdrop (border) colour of Cyan (colour 7), with an active display of White (colour 15) on Dark Blue (colour 4). If, then, 'COLOR 0,4,7' is entered, the characters displayed on the screen in the 'Foreground' colour White would become transparent (colour 0) - and one would 'see' the backdrop colour through them: they would, in effect, take on the colour of the backdrop.

If transparent is specified for the backdrop colour, then it defaults to black: if the MSX had an 'external' VDP connected and it was enabled, the screen display generated by the 'external' VDP would be displayed instead of black.

### 1.3.3 The Pattern or Multicolour Plane

The next 'layer' over the backdrop is the Character Pattern or Multicolour plane. It occupies only the 'active screen' area, and it is the layer on which all the characters or, in Screen Mode 3, the multicolour blocks are displayed in the specified colours.

The size of the active display screen is 256x192 pixels, a pixel being the smallest point that can be defined on the screen. In Screen Mode 0, each character occupies a block of 6x8 pixels, and the 8 pixels on both the side edges are not accessed. Thus it is possible to have 40 characters to a row (240/6), and 24 rows (192/8).

In Screen Modes 1 and 2, the character sizes are 8x8 pixels, and so there can be a maximum of 32 characters to a row and 24 rows. It is usually recommended, however, that the number of characters to a row be limited to 29, to avoid loss of characters at the edges on some television receivers. The narrower width can be achieved by using the MSX BASIC 'WIDTH' command. For even numbers (e.g. WIDTH 28), the MSX centres the display: when an odd number is specified, the first character position on the left hand side is indented.

In Screen Mode 3, the multicolour blocks each occupy 4x4 pixels, and so it is possible to have 64 blocks to a row and 48 rows. Each individual block, however, can have only one colour, and cannot be given a character 'pattern' as with the other screens. It is possible to have all 15 colours and transparent displayed on screen, and the 4x4 pixel blocks can of course be used to create pictures.

### 1.3.4 The Sprite Planes

For Screen Modes 1, 2 and 3 there are 32 Sprite planes located over the Pattern or Multicolour layer. These are numbered 0 to 31, 31 being closest to the Pattern layer, and 0 being nearest to the 'front' of the screen.

Each of the Sprite planes can carry one Sprite only. Sprites can have basic patterns occupying 8x8 pixels or 16x16 pixels, and both these pattern sizes can be 'magnified' so that each pattern point occupies a block of 2x2 pixels instead of 1 pixel. Sprites on all planes have the same size and magnification: the sizes and magnifications cannot be intermixed.

Each Sprite can be given one specific colour only, the coloured part of the Sprite being associated with the defined pattern. The non-defined pattern areas are automatically transparent.

Since there are 32 Sprite planes, 32 Sprites can be positioned on the screen at any one time. However, there cannot be more than four Sprites on any .one horizontal line of pixels. Should more occur, the four Sprites nearest to the front (i.e. the four Sprites with the lowest plane numbers) will be displayed. The plane number of the fifth Sprite is recorded in the lower five bits of the VDP Status Register and the '5th Sprite Flag' - bit 6 of that Register - will be set to a '1'. Any further Sprites on the same horizontal line are ignored.

Sprites can occupy any position in the active display area, and they can be made to occupy the same area on the screen. When this happens, Sprites nearest to the front 'block out' the Sprites behind their pattern areas, but allow them to be seen through their transparent non-pattern areas. It is thus possible to define Sprites in such a way that they can be overlaid to produce a multicoloured pattern.

Also, since each Sprite occupies just one plane and can be moved anywhere over the active screen area, it is possible to make Sprites move in front of or behind each other, so giving a three-dimensional effect.

The VDP reports when the patterns of any two Sprites occupy the same pixel - by setting bit 5 of the Status Register. This coincidence of Sprites can be detected using the MSX BASIC 'ON SPRITE GOSUB' statement after a SPRITE ON command has been invoked. Machine code programmers would read the VDP's Status Register by calling the routine at 013E hex: this returns the value of the Status Register in Z80 register A.

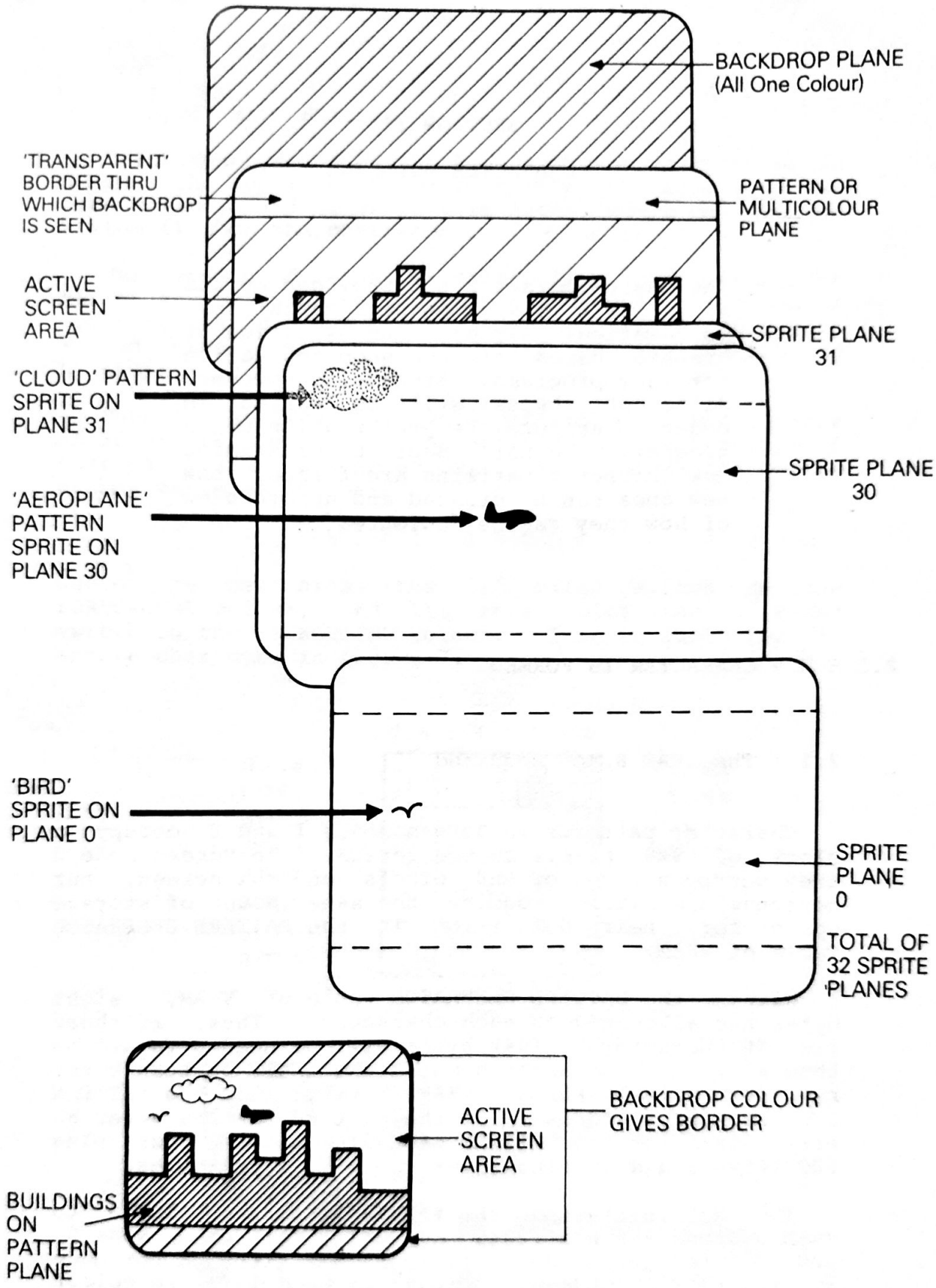



**Figure 6. Build up of the screen.**

SECTION 2

# CHARACTER BUILDING

The basic character set provided by the MSX, with its alphabet variations, mathematical symbols and selection of graphic shapes, gives a good foundation for many programs. Eventually however, different shapes will be wanted in order to enhance the presentation of a program. In this Section we examine how Character patterns are formed, how new ones can be created and an overview of how they can be 'coloured'

## 2.1 HOW A CHARACTER IS FORMED

### 2.1.1 The VRAM Space Required

Character patterns in Screen Modes 1 and 2 occupy a block of 8x8 pixels on the screen. In Screen Mode 0 they occupy a block of 6x8 pixels on the screen, but nevertheless still require the same amount of storage space for their definition in the PATTERN GENERATOR Table of VRAM.

Within the PATTERN GENERATOR Table of VRAM, eight bytes are allocated to each character. Thus, if there are 256 characters, 2048 bytes will be needed to define them all. The character numbers and their pattern bytes run sequentially through VRAM, starting at the PATTERN GENERATOR Base address. So the pattern for Character 65 will start at the PATTERN GENERATOR Base address plus 520 (65x8), and continue over the next seven bytes.

The MSX initialises the PATTERN GENERATOR Table to VRAM address 800 hex (2048 decimal) for Screen Mode 0, and to 0 for Screen Modes 1, 2 and 3 - although for Mode 3, the PATTERN GENERATOR Table is used to store colour pattern blocks rather than character patterns.

## 2.1.2 Bytes Make Patterns

The first byte of an 8-byte character pattern block defines the top line of the character, the next byte the second line, and so on to byte eight, which defines the bottom line of the character.

The pattern of each character line is determined by the binary value of the corresponding byte. (If you're not sure how to convert decimal or hexadecimal to binary - and vice versa - you'll find an explanation in Appendix A).

Thus if the first byte of a pattern block has a value of 5C hex (92 decimal), the pattern for the top line of that particular character will be determined by the binary value of 5C hex:

0 1 0 1 1 1 0 0

where the set bits (the '1' bits) switch on the FOREGROUND colour, and the reset bits (the '0' bits) switch on the BACKGROUND colour. This is shown for an entire character in Figure 7.

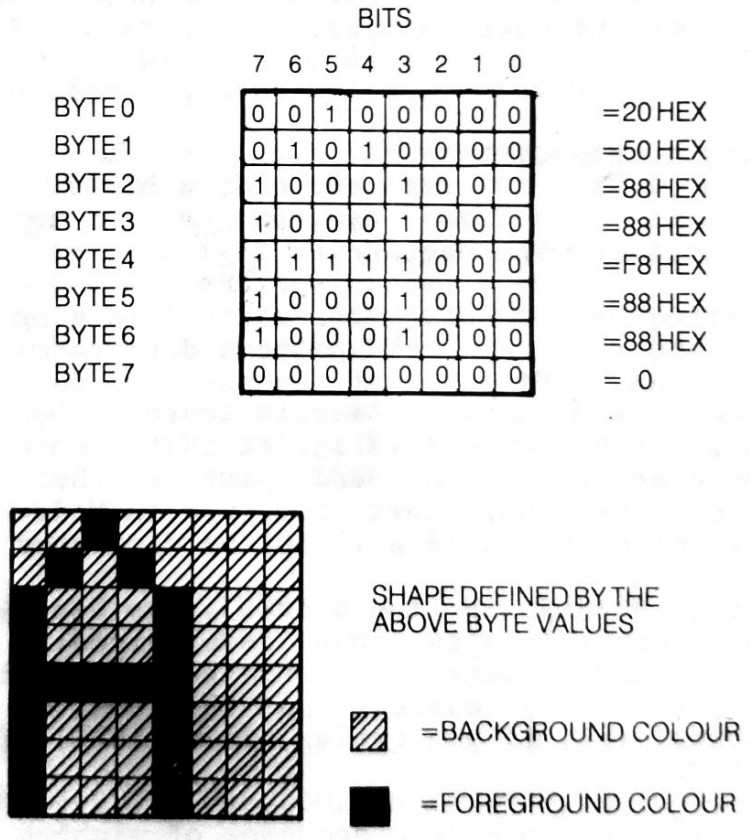



**Figure 7. How Characters are defined.**

### 2.1.3 The MSX Character Set

When Screen Modes 0 and 1 are selected, the MSX automatically loads a complete set of 256 characters into the 2048 bytes of VRAM allocated to the PATTERN GENERATOR, from a storage area within ROM. This storage area starts at 1BBF hex. (Program 7 of Appendix B enables you to display on the screen any character from the MSX set in large detail, with the pattern byte values).

In Screen Mode 2, the PATTERN GENERATOR Table is cleared to 'zeroes', and in Screen Mode 3 it is loaded with 44 hex throughout - for reasons which will be discussed later.

Consequently, any characters defined by the user whilst in one Screen Mode may be lost from VRAM when another Mode is selected.

Note that for Screen 0, the lowest two significant bits of each character pattern byte (bits 0 and 1) are ignored by the VDP and do NOT get reproduced on the screen: Mode 0 characters are 6x8, remember, the 6 pixels in a character row being controlled by the top six bits of the pattern byte.

## 2.2 CREATING A CHARACTER

### 2.2.1 Designing The Pattern

Creating a new character pattern is simply a matter of entering the required pattern data into the correct area of VRAM. First, of course, the pattern must be designed, and this is easily achieved by shading in blocks in an 8x8 grid (similar to that shown in Figure 7) wherever the 'Foreground' part of the pattern is required. Remember that for Screen Mode 0 the right hand two bits (0 and 1) will not be displayed on screen.

Having shaded in the blocks, the binary value for each horizontal line is then ascertained: the non-shaded blocks become '0' and the shaded blocks '1'. The 8-bit binary number can then be converted to hexadecimal (or, if you prefer, to decimal), for loading into VRAM.

You will end up with a sequence of eight byte values, the first byte representing the pattern for the top line, the second byte the pattern for the second line, and so on. Thus if you were constructing the 'A' shape shown in Figure 7, the eight bytes would be (hex) 20, 50, 88, 88, F8, 88, 88 and 0. Alternatively, in decimal the values would be 32, 80, 136, 136, 248, 136, 136, 0.

### 2.2.2 Loading The New Character

It now has to be decided which character number this new pattern is to become. Whatever the number, its 8-byte location in VRAM will start at:

PATTERN GENERATOR Base address + (8xCharacter Number)

So if the PATTERN GENERATOR Base address is ´0´ and the character number is to be 97, then the first byte of the pattern will need to be loaded into VRAM address 0+97x8, or 776 decimal (0308 hex), the second byte into address 777 decimal - and so on. A typical program to achieve this in BASIC would be:

```
10 FOR I=0 TO 7
20  READ D$
30  VPOKE 776+I, VAL("&H"+D$)
40 NEXT
50 REM *Now the Data*
60 DATA 20,50,88,88,F8,88,88,0
```

If you prefer to enter the data as decimal values then in Line 20 use a numeric variable such as ´D´ (or, better, ´D%´) instead of ´D$´, and change Line 30 to VPOKE 776, D (or D%). The data in line 60 must now be entered as decimal values, not hex, of course.

If a number of character patterns are to be entered, they can be VPOKEd within the same FOR-NEXT loop (hence the reason for running the loop from 0 to 7 rather than 776 to 783), in a similar way to Line 30. Alternatively, if the character numbers are consecutive, the FOR-NEXT loop can be extended to embrace the entire range of character patterns to be entered.

Machine code programmers can use the Block Shift ROM routine at address 5C hex: a typical Assembly language subroutine to load the pattern for an ´A´ as character 97 would be:

```
LOADA:LD   HL,CHRPATA
      LD   DE,0+97*8
      LD   BC,8       ;Bytes to shift
      CALL 5CH
      RET

CHRPATA:DB 20H,50H,88H,88H,F8H,88H,88H,0
```

It should be noted that, unlike some computer systems, if a character pattern is changed whilst the original character is being displayed on the screen, then the displayed character will take on the new pattern wherever it appears.

This feature could be put to use within a program: instead of defining a series of characters to provide, say, an animated sequence in one character position, the character itself can be redefined to provide the sequence. Program 8 of Appendix B demonstrates this feature.

### 2.2.3 Printing the Character

Once a character has been defined, it can be printed on the screen either by using the 'PRINT CHR$(number)' statement - suitable for characters with numbers above 32 decimal - or by selection of the appropriate keys.

For example, if character 65 - the letter 'A' - has been redefined, this can be accessed by pressing the 'A' and 'SHIFT' keys on the keyboard.

A Table showing the MSX characters, character numbers and how they are accessed from the keyboard is given in the Appendices.

Alternatively, for Screen Modes 0 and 1, the character number can be VPOKEd into the appropriate address in the VRAM NAME Table for the screen location. In Mode 1, for example, the NAME Table starts at 6144 decimal in VRAM (1800 hex), and the column width is 32. So to place, say, character 97 on the screen at a position three rows down and ten columns across, one would

```
VPOKE 6144+(3*32)+10,97
```

In Mode 0, the NAME Table starts at 0, and the column width is 40, so to place character 97 on the screen three rows down and ten columns across, one would

```
VPOKE 0+(40*3)+10,97
```

In Screen Mode 2, because of the way the MSX initialises the Mode, the entire character pattern will have to be VPOKEd to the PATTERN GENERATOR Table: see Sections 2.3.4 and 6.2.4. Alternatively, Mode 2 can be re-initialised - see Section 6.3.

## 2.3 CHARACTERS AND SCREEN MODES

### 2.3.1 All Change

The VDP and the way the MSX initialises it for BASIC affects the way characters are displayed on the screen in each Mode. The differences between Modes are discussed elsewhere, but are given here for reference purposes.

It is important to remember that when Screen Modes are changed using MSX BASIC, the PATTERN GENERATOR is completely re-initialised: any characters you may have defined in one Mode could be lost from the PATTERN GENERATOR when another Screen Mode is selected.

We will now examine the way the PATTERN GENERATOR Table is used for each Screen Mode.

### 2.3.2 Screen 0

In Screen Mode 0 the characters can be defined as described in Section 2.2, but only the leftmost 6 bits of each pattern line will be reproduced on screen.

This should be remembered when defining characters for Screen Mode 0: the lowest two bits - those on the extreme right of the pattern - will not be visible. If you examine the ASCII character set - and indeed most of the other characters - produced by the MSX, you will see that their patterns do not involve the lowest three bits. This is to enable the character set to be used for Screen 0 (with a one pixel space between the characters), as well as for Screen 1.

The MSX Character set is automatically loaded into VRAM whenever this Mode is selected.

### 2.3.3 Screen 1

Screen Mode 1 characters occupy the full 8x8 pixel block, and they can be defined and changed as described in Section 2.2. Whenever this Mode is selected, the MSX Character set is automatically loaded into the PATTERN GENERATOR Table, and so changes to character patterns should be made after the Mode has been selected.

### 2.3.4 Screen 2

In Screen Mode 2, the MSX initialises the VDP and loads VRAM in such a way that no character is defined within the PATTERN GENERATOR Table until it is to be displayed on the screen: it does this so that the Mode can be used for high resolution graphics. Full details of this initialisation are given in the Section on Screen Mode 2.

The point about this Mode is that, when PRINTing to the screen using MSX BASIC, you are limited to the MSX's own character set, and you can only PRINT to the screen after OPENing the GRP screen.

To give an example:

```
10 SCREEN 2
20 OPEN "GRP:" AS  1
30 PRESET (70,100)
40 PRINT  1,"Here we are!"
50 GOTO 50
```

Line 30 sets the position on the screen at which PRINTing is to occur: note that, unlike the Text Modes, you can position the characters with their top left corner at ANY pixel position on the screen. Line 50 prevents the MSX jumping to Screen 0 or 1 after the program has run - which would clear the displayed phrase from the screen.

You can display a character of your own design on the screen by loading its eight bytes into the PATTERN GENERATOR at the location related to the screen position required. The next short program shows how to display our 'own' character 'A' at screen position 110 (3 rows down, 14 columns across).

```
10 SCREEN 2
20 COLOR 14,4,11
30 FOR I=0 TO 7
40  READ D$
50  VPOKE I+(110*8), VAL("&H"+D$)
60  NEXT
70 DATA 20,50,88,88,F8,88,88,0
80 GOTO 80
```

Line 20 has been added to show you that, when placed on the screen this way, the character takes on the Border colour. One would have to VPOKE the corresponding addresses of the VRAM COLOUR Table to change the colour: this is discussed more fully later on.

You might like to note at this point that the VDP allows for a completely different character to be defined for every position on the screen in Mode 2 - a total of 768 completely different characters! This feature is dealt with in the Section on Mode 2.

You can re-initialise Mode 2 so that you can load a (768) character set 'permanently' into the PATTERN GENERATOR Table, and use the Mode the same way as you use Mode 2 - with direct access from the keyboard. Apart from the number of characters available, this has the advantage of higher resolution colouring - each pattern byte can be individually coloured.

### 2.3.5 Screen 3

In Screen Mode 3 (the Multicolour Mode) the PATTERN GENERATOR Table is not used to define characters as such, but to determine the colours of specific 4x4 pixel blocks. However, like Screen Mode 2, you can OPEN the GRP screen and PRINT from the BASIC character set to the screen, although in this instance, each pixel position in the original character pattern will be represented by a block of 4x4 pixels: try the program at the top of page 40, with Line 10 changed to SCREEN 3, to see the effect.

## 2.4 COLOURING CHARACTERS

### 2.4.1 Each Mode is Different...

The way that a character is coloured depends on the Screen Mode selected, and is discussed in detail in the Screen Mode Sections of this book. Here we are going to take a general look at the way you can change the character colours for Screen Modes 0, 1 and 2.

For these Modes, the VDP looks at the appropriate data in the COLOUR Table area of VRAM 'at the same time' as it looks at the character pattern data. Changing the colours of a specific character will, therefore, change that character's colours wherever it appears on the screen, even if it is already being displayed.

### 2.4.2 Screen 0 Colours

Only two colours are possible on this screen, consequently all characters will have the same Foreground and Background colour.

These colours can be defined by the BASIC 'COLOR x,y,z' statement, where 'x' gives the Foreground colour, 'y' the Background colour, and 'z' has no effect.

Alternatively, one can use the 'VDP(7)=&HFB' statement, where 'F' is the hexadecimal value of the required foreground colour, and 'B' is the hexadecimal value of the Background colour. Thus, for Dark Yellow (colour 10, or 'A' hex) characters on a Dark Red (colour 6) background, one would use 'VDP(7)=&HA6'.

### 2.4.3 Screen 1 Colours

Only 32 bytes in the COLOUR Table are used to define all the 256 characters. MSX initialises the COLOUR Table at 2000 hex (8192 decimal) for Screen Mode 1, and so only the bytes from 2000 hex to 201F hex (8192 to 8223 decimal) have any significance.

Each byte in the COLOUR Table determines the Foreground and Background colours for eight characters. The first byte - at 2000 hex - determines the colours for characters 0 to 7, the second byte for characters 8 to 15, and so on.

The top four bits of the byte determine the Foreground colour, and the bottom four bits of the byte determine the Background colour.

To find which byte defines the colour for a specific character, divide the character number by eight and ignore the remainder. Thus the colouring for the letter 'A', character 65, is determined by the eighth byte in the COLOUR Table - the byte at address 2008 hex (8200 decimal). To change the colour of this character (and also characters 64 to 71), simply VPOKE the required colour into VRAM address 2008 hex.

When you have changed the colours of a character, it will take on the new colours wherever it appears on the screen. If you want a character to appear in different colour combinations, then the character will have to be defined again in a different eight-character block, and the corresponding colour bytes in the COLOR Table changed accordingly.

### 2.4.4 Screen 2 Colours

In this Mode, every one of the eight bytes that go to make up a character pattern has an associated colour byte in the COLOR Table. The COLOR Table in this Mode is therefore the same length as the PATTERN GENERATOR Table - 6144 bytes. The Tables are synchronous: in other words, the 100th byte in the COLOR Table determines the colours for the 100th pattern line in the PATTERN GENERATOR Table.

However, remember that in this Mode, MSX BASIC does not load the character set into the PATTERN GENERATOR Table, but instead initialises the screen for graphic displays. Because of this, the COLOR Table for Mode 2 is primed to address screen locations. This is discussed more fully in the Section on Mode 2.

### 2.4.5 Screen 3 Colours

The VDP does not recognise a COLOR Table in this Mode: all colouring is determined by the contents of the PATTERN GENERATOR Table. See the Section on Screen Mode 3.

SECTION 3

SPRITES

In addition to 'ordinary' characters, the VDP is capable of producing 'Sprites' - special characters that can be moved pixel by pixel around the screen to create animated and three-dimensional displays. In this Section, we examine Sprites in detail.

## 3.1 HOW A SPRITE IS FORMED

### 3.1.1 Screens and Sizes

Sprites are available in Screen Modes 1, 2 and 3, and in all these Modes, they are formed the same way. Furthermore, the SPRITE PATTERN Table is initialised to the same Base address for all three Modes (3800 hex, 14336 decimal) and its contents are not changed when switching from Mode to Mode provided the Sprite size and magnification are left unspecified. This means Sprites can be defined in one Screen Mode, and used in another.

It has been mentioned in earlier Sections that Sprites can be formed in two different sizes, and each size can be unmagnified or magnified. The following table shows the number of pixels occupied by the different sizes of Sprite, and the number of bytes required to generate their patterns.

| SCREEN m,s 's' VALUE | VDP REG 1 Bit 1 | Bit 0 | PIXEL SIZE | BYTES PER PATTERN |
|---|---|---|---|---|
| 0 | 0 | 0 | 8x8 | 8 |
| 1 | 0 | 1 | 16x16 | 8 |
| 2 | 1 | 0 | 16x16 | 32 |
| 3 | 1 | 1 | 32x32 | 32 |

### 3.1.2 The 8 Byte Sprite

This size of Sprite pattern is formed in virtually the same way as the character pattern. Eight consecutive bytes in the SPRITE PATTERN Table define the pattern, the first byte providing the information for the top row of the Sprite, the second byte the second row, and so on to byte eight which defines the bottom row of the Sprite's pattern.

The pattern formation is, therefore, the same as that indicated in Figure 7 for a character pattern. The only difference is that the background colour is always 'transparent': thus, whatever colour lies behind the Sprite's 'background' areas will be seen, whether it is from another Sprite pattern, from a character pattern, or from the 'backdrop'.

The SPRITE PATTERN Table has a maximum size of 2048 bytes, and since it takes eight bytes to define the Sprite pattern, in this size one can have a maximum of 256 different Sprites. Only 32 of them can be displayed on the screen at a time, of course, since there can be only one Sprite on each of the 32 different Sprite planes.

When the Sprite is 'magnified' each bit of a pattern byte controls not one pixel, but a block of 2x2 pixels. Consequently the Sprite pattern occupies an area of 16x16 pixels although of course, it still needs only 8 bytes to define it.

The first Sprite pattern - Sprite number 0 - occupies the first eight bytes of the SPRITE PATTERN Table, the second Sprite - number 1 - occupies the next eight bytes, and so on up to Sprite number 255, which occupies the last eight bytes of the Table. So to find the start address in VRAM for a particular Sprite pattern, simply multiply its number by eight, and add the result to the SPRITE PATTERN Base address. When using the MSX as initialised by the ROM routines, the Base address is always 3800 hex (14336 decimal).

### 3.1.3 The 32 Byte Sprite

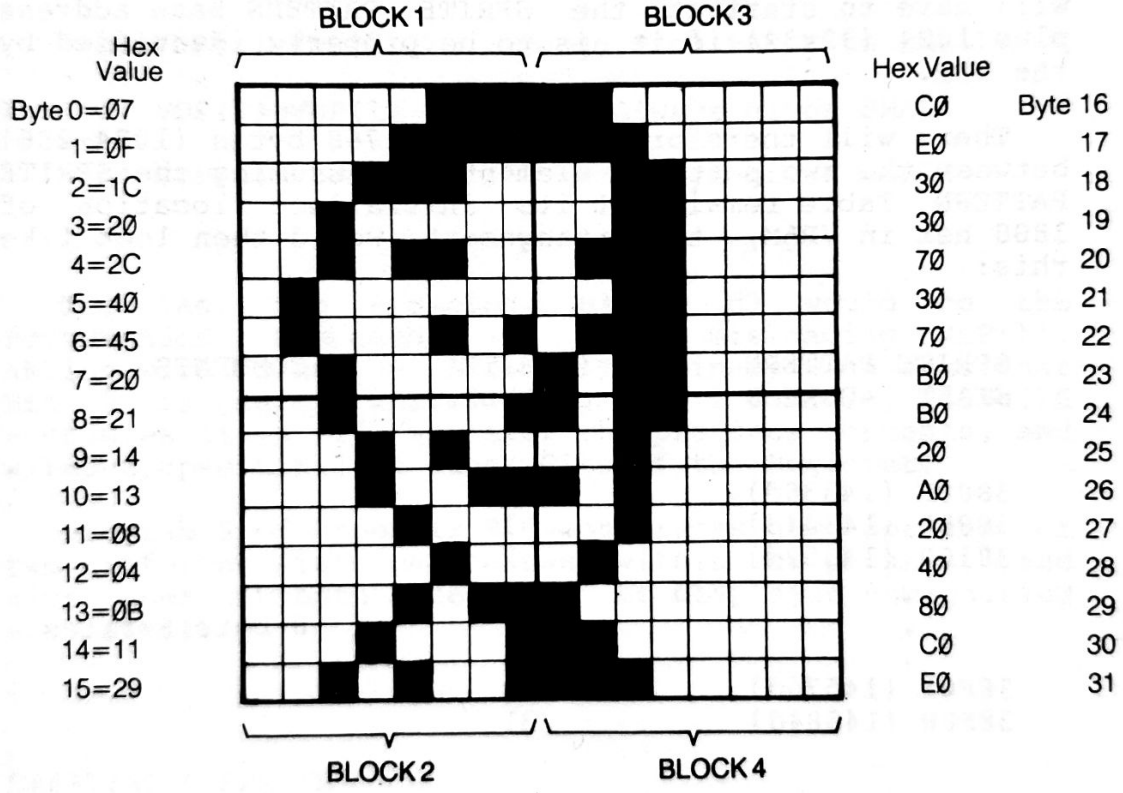
Sprites can also be made to have a pattern that occupies a 16x16 pixel area on the screen, unmagnified, or a 32x32 pixel area when magnified. This is achieved by giving 's' the value 2 or 3 in the BASIC 'SCREEN m,s' command, or by setting bit 1 of VDP Register 1.

These larger Sprites need 32 bytes to define their pattern. The first eight of these bytes relate to the top left block of the Sprite, the next eight bytes to the bottom left block, the third eight bytes to the top right block and the last eight bytes to the bottom right block. (See Figure 8).

In Figure 8, each shaded area represents a binary '1'

- which switches on the Sprite colour, and each non-shaded area represents a '0', which switches on 'transparent'. The hex values that would have to be loaded into the SPRITE PATTERN Table to create this particular Sprite are also shown in Figure 8.

It is important to remember how the pattern data is stored in VRAM - especially when creating a 32 byte Sprite. Since it takes 32 bytes to define this size of Sprite, the maximum number that can be defined is 64 (2048/32).




If this were sprite pattern number 1, then for the Sprite Pattern Table initialised by MSX, BYTE 0 would be at VRAM location 3800 hex plus 32=3820 hex, or 14368 decimal, BYTE 1 at 3821 hex (14369 decimal) etc.

**Figure 8. Defining a 32 Byte Sprite pattern.**

### 3.1.4 Mixing Sprite Sizes

The VDP knows, from bit 1 of VDP Register 1, what size of Sprite has been specified, and if the larger size is chosen, the VDP will always construct the Sprite using 32 bytes. Changing Sprite size during a program can be achieved, but the patterns for the two sizes should be kept apart in the Table.

For example, one may decide to use Sprite numbers 0 to 31 as 8-byte patterns, and Sprites 32 to 63 as 32 byte patterns. In this specific example, the first 32 8-byte Sprites will occupy an area of 256 bytes (8x32). The 32 byte Sprites will occupy an area of 1024 bytes (32x32) and, since the first of these is numbered 32, it will have to start at the SPRITE PATTERN Base address plus 1024 (32x32) if it is to be properly identified by the VDP.

There will therefore be a gap of 768 bytes (1024-256) between the two pattern elements. Assuming the SPRITE PATTERN Table remains at its initialised location of 3800 hex in VRAM, the arrangement would then look like this:

| SPRITE PATTERN START ADDRESS | SPRITE NUMBER | COMMENTS |
|---|---|---|
| 3800H (14336d) | 0 | |
| 3808H (14344d) | 1 | |
| 3810H (14352d) | 2 | |
| . | . | |
| . | . | 8-byte Sprites |
| . | . | |
| 38F0H (14576d) | 30 | |
| 38F8H (14584d) | 31 | |
| 3900H (14592d) | | Unused area |
| -3BFFH (15359d) | | of VRAM |
| 3C00H (15360d) | 32 | |
| 3C20H (15392d) | 33 | |
| 3C40H (15424d) | 34 | |
| . | . | |
| . | . | 32-byte Sprites |
| . | . | |
| 3FC0H (16320d) | 62 | |
| 3FE0H (16352d) | 63 | |

Obviously great care must be taken when using Sprites this way: with our example, calling the 8-byte Sprite numbered '4', say, whilst in the 32-byte Sprite size would yield a block of four of the 8-byte Sprite patterns - numbered 12, 13, 14 and 15.

During a program the Sprite size must be changed by setting or resetting bit 1 of VDP Register 1: to change Sprite size by using the 'SCREEN m,s' statement would re-initialise the entire display as well as clear the Sprite Tables! There are a number of ways of changing Sprite size using Register 1:

(a) VDP(1)=VDP(1)+2 To go from SMALL to LARGE

(b) VDP(1)=VDP(1)-2 To go from LARGE to SMALL

(c) VDP(1)=VDP(1) AND &HFD Always gives SMALL

(d) VDP(1)=VDP(1) OR 2 Always gives LARGE

The last two examples, c) and d) would be the recommended approach, to avoid misloading VDP(1). ANDing with FD hex - 11111101 in binary - ensures that Bit 1 is zero, whilst ORing with 2 - binary 00000010 - ensures it is '1', whatever the previous contents, and without upsetting the other Bits of the Register.

Program 9 of Appendix B demonstrates how Sprites of two different sizes can be used within one program, and also shows how both sizes can be displayed unmagnified and magnified.

## 3.2 CREATING A SPRITE

### 3.2.1 Designing 8-Byte Sprites

The process for creating an 8x8 Sprite pattern is the same as that for creating a character pattern. First the pattern is designed in a grid that is 8x8, then the pattern areas converted to '1's so that the binary value for each row of the pattern can be ascertained. This value is then loaded to the appropriate address in the SPRITE PATTERN Table.

Whereas character patterns have to be VPOKEd into the VRAM area, to enable you to enter Sprite data, MSX BASIC provides a statement variable - 'SPRITE$(x)', where 'x' is the Sprite pattern number.

This variable can be defined in various ways.

(a) SPRITE$(1) = CHR$(7)+CHR$(&HF)+CHR$(28)...etc

(b) SPRITE$(2) = "aAbBcCdD"

(c) SPRITE$(3) = B$ (B$ having been previously defined, of course).

(d) SPRITE$(4) = STRING$(8,255)

All these methods load the required data bytes into the appropriate area of VRAM for the Sprite number: if insufficient bytes are specified then the remaining bytes in the pattern block are made equal to zero. Thus, if one wrote 'SPRITE(8)=CHR$(&H27)', then the first byte of the pattern would be given the value 27 hex (39 decimal), and the remaining bytes - either 7 or 31 of them, depending on the Sprite SIZE - would be zero.

The method given in (b) for defining a Sprite pattern may need explaining. When a 'String variable' is stored by BASIC, the characters given in quotes are converted to their ASCII values - so they are stored within the MSX as numbers. The ASCII number for 'a' is 97, for 'A' it is 65, for 'b' it is 98, for 'B' it is 66 - and so on. So 'SPRITE$(2)="aBbBcCdD"' is a short way of writing 'SPRITE$(2)=CHR$(97)+CHR$(65)+....' and so on.

This method takes up less programming memory space - but does require that you know the character for a given number - and where to find it on the keyboard. You'll find a Table to help you do this in Appendix D.

Machine code programmers can use the ROM routine at 5C hex to load Sprite data: see the details under 'Writing to VRAM' in Section 1.2.1. There is also a routine in ROM which returns the start address for a Sprite pattern in the SPRITE PATTERN Table, which may be useful. The details are:

084H Find Sprite Pattern start address

IN: Sprite pattern number in Register A
OUT: Pattern start address in HL
AF,DE,HL modified

This routine checks whether 8 or 32-byte Sprites have been selected, and returns the correct start address accordingly.

### 3.2.2 Designing 32-Byte Sprites

The process for creating a 32-byte Sprite is similar to that for an 8-byte Sprite, but you must ensure that you get the four 8-byte segments in the correct order.

First the pattern is designed on a 16x16 grid - as shown in Figure 8. The grid is then divided into four quarters, so each quarter is an 8x8 block. The binary - or hex - values for each of the horizontal lines in each of the 8x8 blocks is then ascertained, and written alongside the grid.

When entering the data - using the SPRITE$(x) variable as for the 8x8 Sprite - the data for the top left block must come first, followed by the data for the bottom left block, the top right block and finally the bottom right block. In all, there should be 32 bytes of data: if there are less, the MSX will load zeroes into the pattern table to make up the difference. These will always be loaded at the end of your byte sequence.

Thus if you specify only 16 bytes of a 32-byte Sprite, the entire right hand side of the Sprite will be 'blank', since the pattern area for the top right and bottom right blocks will be filled with '0'.

## 3.3 MOVING SPRITES

### 3.3.1 How the VDP does it

For characters, there are three Tables in VRAM to define their pattern, colour and screen location. Sprites have only two - the SPRITE PATTERN Table, discussed in Sections 3.1 and 3.2, and the SPRITE ATTRIBUTE Table.

Each Sprite Plane has a block of four bytes within the ATTRIBUTE Table, to define the Sprite's location on the screen, to name the pattern required, and to specify the colour. As with characters, whenever the VDP does an active screen refresh - 50 times a second - it examines the SPRITE ATTRIBUTE Table to see what Sprite should be going where, and acts accordingly.

For all the Sprite Screen Modes (1, 2 and 3), the MSX initialises the SPRITE PATTERN Table to start at 1B00 hex (6912 decimal) in VRAM. Each of the 32 Sprite planes has one set of four attribute bytes, and so the Table is 128 bytes long (32x4).

The function of the four attribute bytes is shown in Figure 9, and will now be described in detail.

### The First Byte (Byte 0)

This byte determines the vertical pixel position of the top left corner of the Sprite pattern. It is defined so that a value of ´-1´ butts the top edge of the Sprite right against the top border of the screen.

Values of -2 to -8 will cause a Sprite occupying an 8x8 pixel block to apparently ´dissappear´ behind the border. Similarly, Sprites occupying blocks of 16x16 and 32x32 pixels can be made to gradually dissappear by increasing the negative value held by this byte to -16 and -32 respectively. Thus the Sprite can be made to apparently ´bleed´ off or onto the screen at the top.

Programmers not familiar with machine code techniques may wonder how a negative number - binary or otherwise - can be stored in one byte. The answer is the top bit - bit 7 - is used in this instance to indicate the sign of the number, and the rest of the number is ´two´s complemented´. The VDP evaluates the contents of the byte to ascertain whether it should be regarded as a two´s complement or not. For further information on this topic, please refer to a book on Z80 machine coding - such as ´Starting Machine Code on the MSX´, by Geoff Ridley.

Machine code programmers will have gathered from the last paragraph that to make a Sprite dissappear at the top of the screen, they should two´s complement for decimal values -1 to -32.

At the bottom of the screen, values approaching 191 decimal will make the Sprite similarly ´dissappear´ behind the border area. What actually happens, of course, is that both at the top and the bottom of the screen, the Sprites are ´leaving´ the active display area, but the effect is they are going behind the border.

Two very special values can be held by this Vertical position byte. They are 209 decimal and 208 decimal. 209 decimal tells the VDP to ´switch off this Sprite, so that it isn´t displayed at all´. 208 decimal does the same thing - but it also tells the VDP to ´switch off all the bytes with higher plane numbers too´. Thus, if the Sprite on plane 6, say, is given a vertical position attribute of 208, not only will the Sprite on plane 6 be switched off, but so will the Sprites on planes 7 to 31.

## The Second Byte (Byte 1)

The next byte in a SPRITE ATTRIBUTE block locates the horizontal pixel position of the SPRITE - again, using the Sprite's top left corner as the marker. The values can range from 0 to 255 decimal: values in the region of 255 cause the Sprite to 'disappear' at the right edge of the display.

A value of '0', however, butts the Sprite up against the left edge of the display: to allow Sprites to enter and disappear at the left edge, Bit 7 of the last byte (byte 3) in the ATTRIBUTE block is used. When this particular bit is set ('1'), the horizontal position of the Sprite is shifted left by 32 pixels. Values below 32 then placed in the horizontal position byte will cause the Sprite to become concealed behind the left edge of the display.

In BASIC, one would simply enter a negative value from -1 to -32 for the horizontal position, in order to achieve the entrance or dissapparing act at the left edge: the MSX ROM routines make the necessary entries in the ATTRIBUTE Table. Machine code programmers will have to set or reset Bit 7 of the fourth attribute byte according to their needs.

It should be noted that Sprites operate over the entire width of the active display screen, irrespective of the character setting given by the BASIC 'WIDTH' statement.

## The Third Byte (Byte 2)

This byte tells the VDP which Sprite pattern number should appear on the attribute's particular Plane. Thus, if the attribute is for Plane 4, say, and this byte holds the value 17 decimal, then Sprite pattern number 17 will be displayed on Plane 4.

## The Fourth Byte (Byte 3)

The last attribute byte holds in its lower four bits the number for the colour to be displayed. The most significant bit - Bit 7 - has already been discussed under the Second Byte: it shifts the Sprite pattern 32 pixels to the left. The remaining bits of this byte are not used.

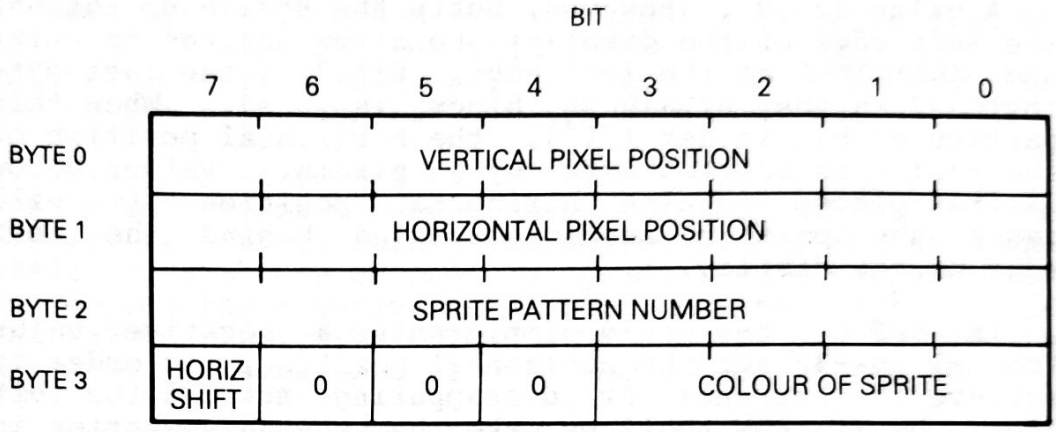



**Figure 9. The Four Attribute Bytes for a Sprite Plane**

### 3.3.2 Initial Settings

When the MSX is switched on, the four bytes of each attribute block in the SPRITE ATTRIBUTE Table are initialised the following way:

(a) The Vertical Position byte (byte 0) is set to 209 - which switches off the Sprite.

(b) The Horizontal Position byte value varies for each Plane of the attribute block - but is irrelevant since the Sprite is 'switched off'.

(c) The Pattern Number Byte is set to the same value as the Plane number: thus, Plane 0 is set to pattern 0, Plane 1 is set to pattern 1, and so on.

(d) The Colour byte is set to the initial Foreground colour, usually White (15 decimal).

Whenever the data in an attribute is changed, it retains the value given to it until changed again - even when Screen Modes are switched: in other words, it is not initialised again unless the MSX is 'reset'.

Thus if the pattern and colour is to remain the same once set, subsequent use of the BASIC statement

'PUTSPRITE' for that Plane need not include the colour or pattern data.

The following program demonstrates this feature:

```
10 SCREEN 1:VDP(6)=VDP(4)
20 PUTSPRITE 0,(10,209),1,42
30 FOR J=0 TO 150
40  PUTSPRITE 0,(10+J,10+J)
50 NEXT
60 GOTO 30
```

Line 10 simply sets the Sprite Patterns to the character set (after ensuring the machine is in a Mode to accept Sprites!). Line 20 sets up the Sprite data - in particular, the colour is set to Black ('1') and the pattern to '*' (Character 42), while leaving the Sprite still switched off (the vertical locator = 209). Line 30 then starts the loop to move the Sprite across the screen: note that in Line 40, it is not necessary to specify colour or pattern again.

The SPRITE PATTERN Table is usually initialised to zeroes throughout, so no Sprite patterns are present. However, you may find on some MSX machines one or two Sprite patterns have in fact been set - the Sony 'Hit Bit', for example, initialises the first few Sprite patterns on switch-on to 'markers' that it uses for its opening Menu display.

### 3.3.3 Making the Move

In MSX BASIC, Sprites are moved using a PUTSPRITE statement, or by VPOKEing the required data into the correct VRAM address.

#### PUTSPRITE

The PUTSPRITE statement has one of two formats:

(a) PUTSPRITE p,(x,y),c,n

(b) PUTSPRITE p,STEP(x,y,),c,n

where 'p' is the Plane number for the Sprite, 'x' is its horizontal pixel location, in the range -32 to 255, 'y' is its vertical pixel position in the range -32 to 192, 'c' is the colour and 'n' is the pattern number. Note that values for 'y' approaching 255 dupe the MSX into thinking that the Sprite should be located at the top of the screen (see Section 3.3.1).

As previously mentioned in Section 3.3.2, for both formats the values for 'c' and 'n' can be ignored once

they have been correctly set for the Plane. Initially they are set to the switch-on Foreground colour and the Plane number respectively.

If only the pattern is being changed, then the format would be:

```
PUTSPRITE p,(x,y),,n
```

The variables 'p, x and y' must always be present.

The first format, (a), gives an absolute pixel location for the top left corner of the Sprite, determined by the values of 'x' (horizontal) and 'y' (vertical). To move a Sprite across the screen using this format, one would use a program segment such as

```
70 FOR I=40 to 100
80 PUTSPRITE 1,(I,120)
90 NEXT
```

The values for 'x' and 'y' can also be expressions - try the following program, for example:

```
10 SCREEN 1:VDP(6)=VDP(4)
20 FOR I=1 TO 7.3 STEP .05
30 PUTSPRITE 0,(120+SIN(I)*50,90+COS(I)*50),1,42
40 NEXT
```

This moves a '*' around the screen in a circle, each location on the screen being defined in absolute terms by the expressions for 'x' and 'y'.

The second format, (b), uses 'x' and 'y' to move the Sprite to a position that is relative to its previous position. In other words, the value of 'x' is added to the previous horizontal position, and the value of 'y' is added to the previous vertical position. A program segment to move a Sprite across the screen using this format could be:

```
70 FOR I=1 to 50
80 PUTSPRITE 0,STEP(1,0)
90 NEXT
```

As you can see, in this instance the loop counter is not used in the determination of the Sprite's location - merely to count off how many times the Sprite must move relative to its last position. If the loop counter variable is used, then the Sprite will appear to accelerate very quickly indeed: try the last program, making the 'x' value 'I/5' instead of 'I', and you'll see the effect.

## VPOKEing a Sprite

A Sprite´s attributes can be changed by VPOKEing the required data into the correct location in VRAM. To find the attribute start address for the desired Plane, multiply the Plane number by 4, and add it to the start address for the SPRITE ATTRIBUTE Table - in the MSX, this is 6912 decimal (1B00 hex). This gives the address for the first byte of the attribute for that Plane - see Figure 9.

The following program demonstrates how the VPOKE can be used to move a Sprite across the screen on Plane 3:

```
10 SCREEN 1:VDP(6)=VDP(4)
20 VPOKE 6912+(3*4),100
30 VPOKE 6914+(3*4),42
40 VPOKE 6915+(3*4),1
50 FOR I=40 TO 200
60  VPOKE 6913+(3*4),I
70 NEXT
```

Line 10 should be familiar to you now - it sets the Sprite Patterns to the character patterns. In lines 20, 30 and 40 the ´y´, ´n´ and ´c´ attributes are set to put a Black ´*´ patterned Sprite, 101 pixels down (the top pixel line is ´-1´, remember) on Plane 3 (note the ´3*4´ added to 6912, to find the correct VRAM addresses). Then in line 60, the required ´x´ pixel position is VPOKED into the horizontal attribute byte, to move it across the screen.

Machine code programmers would use a similar process, entering data into the VRAM addresses using ROM routines as described in the first Section of this book. There is a handy ROM routine to identify the start address of the attributes for a specific Plane. The details are as follows:

**087H Find Sprite Attribute start address**

**IN:** Sprite Plane number in Register A
(F928 hex = Attribute Base address)

**OUT:** Attribute start address in HL
AF,DE,HL modified

It should be noted that in this routine, the MSX looks to the SPRITE ATTRIBUTE Base address it has stored at F928 hex: if the Base address is changed, its new location should be entered at F928 hex (low byte) and F929 hex (high byte) for the above routine to produce a valid result.

Machine code programmers won't need reminding of the speed of machine code: delays must be incorporated between successive Sprite moves - otherwise they occur too fast to be seen! One way of achieving a suitable 'timing' for Sprite movement is to use the Interrupt signal from the VDP itself - for further details see Section 3.3.6

### 3.3.4 Four to a line

The patterns of only four Sprites can be displayed on one horizontal pixel line: if more Sprites have a line of their pattern on the same line of pixels, only the four Sprites with the lowest Plane numbers (i.e those 'nearest the front' of the screen) will have that particular line of their pattern displayed.

The 'Fifth Sprite' Flag of the VDP Status Register (Bit 6, VDP Register 8) is set, and the next highest Plane number carrying an offending Sprite is loaded into the lower 5 Bits of the Status Register.

Thus if Sprites on Planes 3, 5, 7, 9, 12, 14 and 17 all have lines of their patterns on the same horizontal lines of pixels, then only Sprites 3, 5, 7 and 9 will be displayed in full. Sprites 12, 14 and 17 will lose those parts of their patterns on the common lines, the Fifth Sprite Flag will be set, and '12' will be loaded into the lower part of the VDP Status Register.

The Plane number of the Fifth Sprite can be found by a program segment such as:

```
IF VDP(8) AND &H40 THEN FS = VDP(8) AND &H1F
```

This tests the Fifth Sprite Flag, and if it is '1', returns the Plane number of the fifth Sprite in variable FS. Program 6 of Appendix B demonstrates the 'four in line' feature of MSX Sprites.

Machine code programmers should refer to Section 3.3.6 for information about reading the Status Register.

### 3.3.5 Collision Courses

When two (or more) Sprites have one active bit of their patterns occupying the same pixel on the screen, the Coincidence Flag of the VDP Status Register 8 is set. This is described under 'Bit 5' of VDP Register 8 in Section 1.2.2.

MSX BASIC provides Interrupt driven statements to allow such collisions to be acted upon. The statements are:

```
ON SPRITE GOSUB
SPRITE ON
SPRITE OFF
SPRITE STOP
```

The Coincidence Flag is examined every time the VDP provides an interrupt signal - 50 times a second. If it is set and the SPRITE ON statement has been made, program sequencing jumps to the subroutine specified by the ON SPRITE GOSUB statement. This subroutine should of course, provide a RETURN - the RETURN being made to that part of the program being processed when the interrupt occured.

While the subroutine is being processed, other program accessible interrupt-driven routines are temporarily inhibited, so that two cannot be operative simulataneously.

SPRITE OFF switches off the Sprite interrupt process altogether, as far as the program is concerned. When the SPRITE STOP statement has been made, a Sprite coincidence is 'remembered' until a SPRITE ON statement occurs in the program. The interrupt is then acted upon accordingly - with a jump to the ON SPRITE GOSUB routine.

As with all the interrupt-driven routines, the 'ON SPRITE GOSUB' statement can be made right at the beginning of a program, and invoked when desired in the program by the SPRITE ON statement. Since the MSX does all the testing in ROM, it saves the need for tests to be incorporated in the main program - and that means faster program execution. It also means that your MSX can apparently be doing several things at once - accepting an input from the keyboard, playing 'background' music and testing for a Sprite collision, to name but three - all whilst the main program is running.

### 3.3.6 Sprite Status for Machine Code Programmers

Machine code programmers can test for the fifth Sprite on a line and for Sprites coinciding by examining VDP Register 8, the Status Register. However, reading this Register clears the Interrupt Flag - which could adversely affect the machine´s own hardware interrupt-driven routines.

Furthermore, the Coincidence Flag is also reset when the Status Register has been read. On the face of it, this makes life a little tricky. If the Status Register is read prior to the MSX´s own read operation, it will not see a ´set´ interrupt flag, and so will not perform any of its hardware interrupt routines. If it´s read after the MSX does its own read, the Coincidence Flag will have been cleared, and Sprite ´crashes´ will go undetected.

There is, of course, a solution. The Status Register must be ´read´ at the same time as the MSX reads it.. In practice, this means ´saving´ the result of the MSX´s read for subsequent examination. Fortunately, the authors of MSX BASIC have done this for us: you´ll find the latest information on the VDP Status Register at F3E7 hex.

The MSX ROM hardware interrupt routine provides a number of useful features for machine code programmers, so let us look at the routine in a little more detail.

#### ROM Interrupt Routine

The routine is accessed whenever a ´RST 38H´ occurs - the normal address for the Z80´s Interrupt Mode 1. On such interrupts, the MSX pushes both Z80 Register sets to the stack, then CALLs a ´hook´ at FD9A hex.

This hook comprises a series of five ´RETurns´ - and since they are in RAM, they can be replaced by a jump to one´s own routine, the RETurn going back to the hardware interrupt process. This hook enables additional hardware interrupt procedures to be added.

On return from this hook, the MSX reads the VDP Status Register - and if the VDP´s Interrupt Flag - Bit 7 - is NOT set, it returns from the interrupt routine. (Hence the reason for being careful about reading the Status Register - which clears the VDP´s Interrupt Flag).

If the VDP´s Interrupt Flag IS set, there is an immediate CALL to another hook, at FD9F hex. Since VDP interrupts occur 50 times a second, this hook can provide an extremely useful ´clock´. It, too, consists of a series of five ´RETurns´, which can be replaced by a jump to the programmer´s own routines.

The hook can be used, for example, to increment a ´timer´ for Sprite movements, or to initiate Sprite movement direct. However, to traverse the screen - some 250 pixels - would take five seconds at 50 pixels per second: for faster speeds, it would be necessary to move the Sprite more than one pixel at a time, or to incorporate one´s own delay routine.

At this hook, the Z80 Register A contains the result of the VDP´s Status Register read - so it could be saved in a suitable place in memory in an ´own´ routine. Again, care should be taken here, because the MSX not only uses the information in Z80 Register A itself in subsequent operations but also the data held in other Z80 Registers. So their contents must be preserved - using PUSHes and POPs - in the user´s own routine.

On return from the hook, the contents of Z80 Register A are loaded into the Status Register store, at F3E7H. This address can be examined for details on the Fifth Sprite and the Coincidence Flag, without need to use the hook and without detriment to any other routines.

The ROM Interrupt routine then goes on to examine the status of the the various interrupt conditions for BASIC (ON KEY... ON STOP... ON INTERVAL... and so on) as well as the status of any PLAY statements and keyboard inputs etc. That´s how the MSX can play music while a program is running, and how it ´remembers´ which keys have been pressed so that it can display them when an input from the keyboard is called.

To achieve a ´clock´ signal - to time Sprite movement for example - the hook at FD9F hex should be used, this hook being ´activated´ 50 times a second while interrupts are ENABLED. When interrupts are DISABLED - which occurs, for example, during the brief moments it takes to set up the VDP for a Read or Write operation - then the FD9F hex hook is not accessed, but this should not seriously disrupt any general timing mechanisms.

To help newcomers to machine code programming, a simple example of how this hook can be used to perform a ´count´ is now given.

First, the hook must be loaded with a jump to the routine address, which we shall call 'COUNT' and which we will Assemble at E000 hex (having ensured space in RAM is reserved, using the BASIC 'CLEAR' statement, of course!). The hook is five bytes long, so there's plenty of room for the three byte jump command:

```
FD9F  C300E0          JP   COUNT
```

One could do a CALL, of course - followed by a RETurn. But this isn't really necessary, since our COUNT routine will have a RETurn which will send processing directly back to the ROM Interrupt routine, instead of via the hook.

The next step is to write the COUNT routine to update the value of our own counter, which we will call TICKER:

```
E000  E5       COUNT:PUSH HL
E001  2107E0         LD   HL,TICKER
E004  34             INC  (HL)
E005  E1             POP  HL
E006  C9             RET

E007  00       TICKER:DB   0
```

Notice the PUSH HL at the start of this routine and the POP HL at the end: that's to make sure that on the RETurn, Register pair HL contains the same data it came in with, so that the MSX isn't presented with spurious data. Whenever using a hook, always make sure the data in all Registers is returned the way it came in.

Every time the machine does a hardware interrupt, this routine will add one to the data in TICKER, at address E007 hex. So it will clock up '50' every second. Elsewhere in the program, TICKER can be examined to see if it's 'time for action' - move a Sprite, perhaps. If it is planned to move a Sprite every tenth of a second, then the move must be made when TICKER reaches '5', and a program segment could look like this:

```
MOVE?:LD   A,(TICKER)
      CP   5
      JP   NC,MOVEIT
```

The routine 'MOVEIT' would actually make the move, by updating the appropriate byte of the relevant attribute. MOVEIT in this case would be called whenever TICKER had clocked up '5' (or more): the MOVEIT routine should, of course, include a statement to reset TICKER to zero after the move has been made.

SECTION 4

# SCREEN MODE 0

## Text Only

This is the first of the four possible Screen Modes for the MSX: it is used mainly for text, offering a Screen size of 40 columns x 24 rows and only two colours at a time. Most models of the MSX enter this Mode on switch-on. Either this Mode or Mode 1 (whichever was previously in use) will also be selected when a program using Mode 2 or Mode 3 is terminated.

## 4.1 SCREEN MODE SPECIFICATION

### 4.1.1 Screen Parameters

| | |
|---|---|
| Screen Size (Max): | 40 columns x 24 rows |
| (Recommended): | 37 columns x 24 rows (to avoid loss at edges) |
| Character set (Max): | 256 |
| Character Size: | 6 pixels x 8 pixels deep (Right two bits of each Character is 'lost') |
| Characters re-definable: | Yes |
| Colours available: | Any two from range of 15 |
| Sprites: | Not available |

This Mode is selected when Bit 1 of VDP Register 0 and Bit 3 of VDP Register 1 are equal to '0', and Bit 4 of VDP Register 1 is equal to '1': VDP Registers 2 and 7 must also be initialised.

### 4.1.2 MSX Initialisation of Mode 0

When Screen Mode 0 is selected using BASIC, the VDP Registers are initialised by the MSX so that the Tables in VRAM are set as follows:

| | | |
|---|---|---|
| PATTERN NAME: | 0 | VDP(2)=0<br>BASE(0)=0 |
| PATTERN GENERATOR: | 800 hex<br>2048 dec | VDP(4)=1<br>BASE(2)=800 hex |

During the Screen initialisation process, the 256 MSX character set is loaded into the PATTERN GENERATOR Table in VRAM from its location in ROM, the line width is set to its default value - usually 37, as determined by the contents of address F3AE hex in RAM, the screen is cleared and the default colours set.

Screen colour for this Mode is derived from VDP Register 7: the top four bits define the Foreground colour, the bottom four bits define both the Background and the Border colour. The initial setting is usually F4 hex - that is, White on a Dark Blue Background.

VDP Registers 3 (COLOUR Table start address), 5 and 6 (SPRITE data) have no significance, since colour is derived from VDP Register 7, and Sprites are not available.

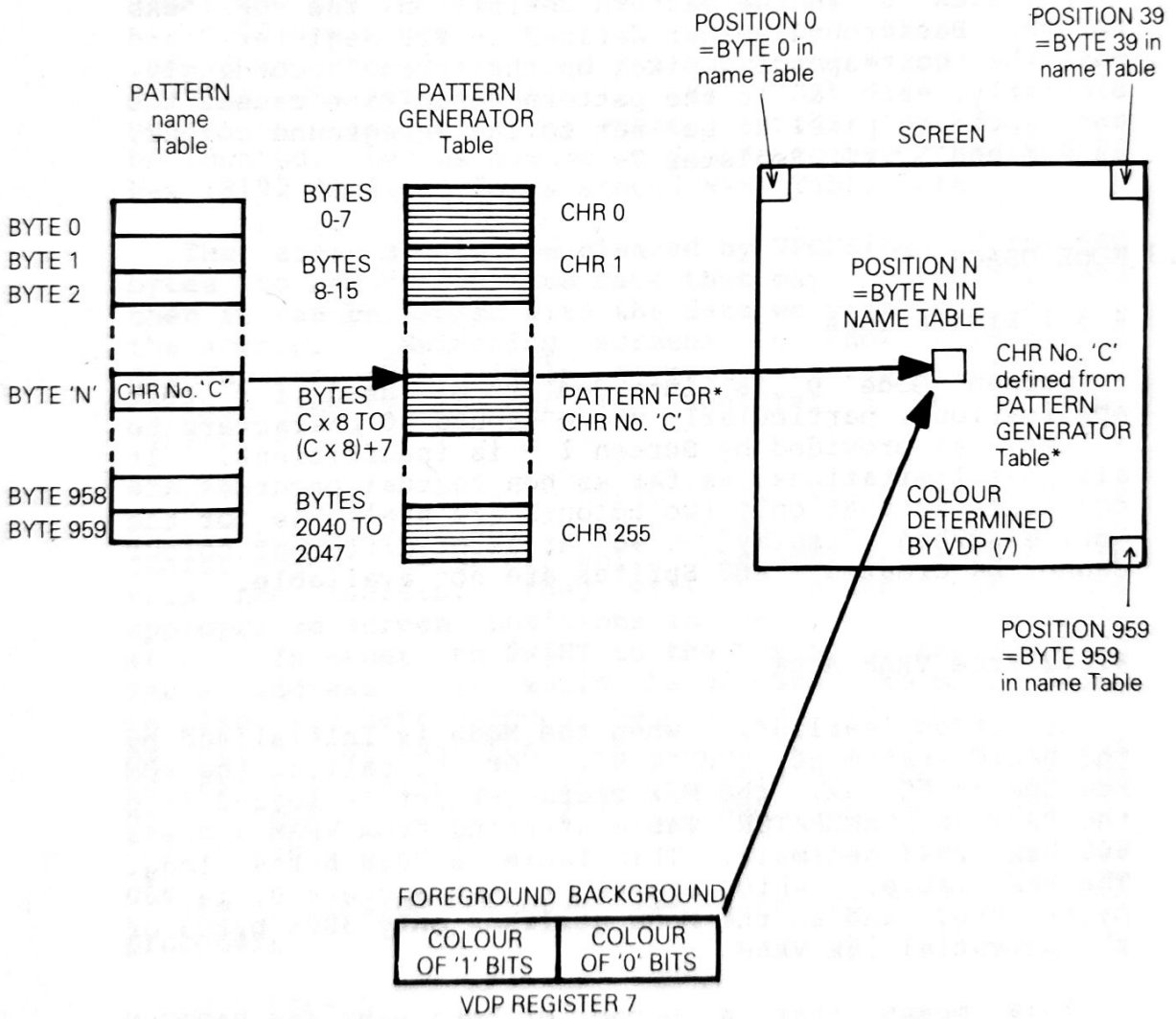
The MSX BASIC Graphic statements (DRAW, LINE, CIRCLE, PAINT, PSET, PRESET and POINT) are not operative in this Mode.

## 4.2 HOW MODE 0 OPERATES

The PATTERN NAME Table for Mode 0 is 960 bytes long, each byte 'mapping' to a screen location. The first byte of the Table determines the character displayed at the top left of the screen, the second byte determines the character in the column next to it, and so on. The arrangement is similar to that shown in Figure 3, except that for this Mode each row has 40 columns.

Each character position on the screen occupies a block of six horizontal pixels x 8 vertical pixels, while the character pattern is actually defined by eight bytes, each of which has eight bits. For this screen only, the VDP therefore ignores the least significant two bits of the each pattern defining byte: that is to say, the rightmost side of the pattern block is not reproduced on the screen.

For this reason, most of the characters are either defined within the top five bits, leaving one bit 'blank' to provide the space between characters, or they are defined within the top six bits. Characters defined using seven or eight bits per byte will have their right sides truncated.




*NOTE:– Lowest two bits of each pattern definition are ignored in this mode.

**Figure 10. Creating Mode 0 Screen display**

During the active-screen refresh period, the VDP examines each byte in the NAME Table in turn, to determine what should be placed in the corresponding location on the screen. Whatever value it finds in the NAME Table it takes to be a character number. It then looks to the PATTERN GENERATOR Table, locating the start of the pattern definition by its character number: if it found 65 decimal in the NAME Table, it would look to the pattern which started at 65x8 bytes from the beginning of the PATTERN GENERATOR Table.

For each '0' in the pattern definition, the VDP looks to the Background colour defined in VDP Register 7 and sets the corresponding pixel on the screen accordingly. Similarly, each '1' in the pattern definition causes the corresponding pixel to be set to the Foreground colour, as defined by VDP Register 7.

## 4.3 MODE USAGE

### 4.3.1 Limitations

Screen Mode 0 is intended to be used for text applications, particularly where around 30 characters to a line - as provided by Screen 1 - is insufficient. It also has limitations as far as non-textual programs are concerned in that only two colours are available for the entire screen display - so areas of different colour cannot be created - and Sprites are not available.

### 4.3.2 Free VRAM Area

As stated earlier, when the Mode is initialised by the BASIC statement 'SCREEN 0', or by calling the ROM routine at 6C hex, the MSX character set is loaded into the PATTERN GENERATOR Table starting from VRAM address 800 hex (2048 decimal). This Table is 2048 bytes long. The NAME Table, which starts at VRAM address 0, is 960 bytes long, and so the Mode utilises only 3008 bytes of the potential 16k VRAM.

This means that a number of both NAME and PATTERN GENERATOR Tables can be defined and, provided they are located at viable Base addresses, switching from one Table to another is simply a matter of resetting the appropriate VDP Register. Details of how to do this are given in the following paragraphs, while possible start addresses for the Tables are given in the Appendices.

This technique can provide the programmer with a very fast means of switching screen displays.

### 4.3.3 Switching NAME Tables

In a very simple example of switching the NAME Table address, one may decide to set up a separate screen with a 'Menu' or 'Help' information. The initialised Base for the NAME Table is at 0 in VRAM. The Table always 'sits' on 1k (400 hex) boundaries, and so other potential Base addresses are 400 hex, 800 hex, C00 hex, 1000 hex, 1400 hex 1800 hex, 1C00 hex - and so on, up to 3C00 hex maximum.

The PATTERN GENERATOR Table is initialised at 800 hex and runs for 2048 bytes (800 hex), and so occupies the area 800 hex to FFF hex. If we avoid this area (as indeed we should if we don't want to infringe into the PATTERN GENERATOR Table!), then that still leaves a number of potential areas where a further NAME Table can be located. Let us assume we decide to use address 2000 hex (8192 decimal) for a second NAME Table Base.

The area should be cleared by VPOKEing '0' for 960 bytes (to remove spurious data that may be there), and then it can be VPOKEd with the data we want to appear on the screen. Switching screens to show this second display is then simply a matter of resetting VDP(2) - the NAME Table defining Register - to 8 (i.e. 'VDP(2)=8').

Note that, because BASIC still expects to find the NAME Table at 0, any PRINT statements will not produce characters on the screen whilst the NAME Table is set to this new address: they will be entered into the appropriate screen positions in the original NAME Table at 0. In order to PRINT to the screen at the new NAME Table address, it would be necessary to make BASIC realise you have changed the address: this can be done by using the BASE function. BASE(0) gives the start address for the NAME Table, so this must be set to the new address ( BASE(0)=&H2000 ), and then the screen re-initialised.

The following program may help to clarify this procedure:

```
10 SCREEN 0
20 PRINT "THIS SCREEN, NAME TABLE AT 0"
30 GOSUB 100
40 BASE(0)=&H2000
50 VDP(2)=8: REM See Section 1
60 SCREEN 0
70 PRINT "THIS SCREEN, NAME TABLE AT &H2000"
80 GOSUB 100
90 STOP
100 FOR I=0 TO 2000:NEXT:RETURN
```

After running this program, do a 'PRINT VDP(2)' and you'll see it has the value '8'. Similarly, BASE(0) will have the value &h2000 (8192 decimal). You will be able to use the screen in the normal way - that is, entering commands direct from the keyboard - although it has now been completely re-initialised to a new NAME Table address.

To see what is in the 'original' NAME Table - at Base address 0 - enter VDP(2)=0 direct from the keyboard. You will then see the result of the PRINT statement in line 20. Now try entering other data from the keyboard, and you will see your efforts apparently unrewarded. Press the 'RETURN' key, to ensure you are on a fresh line. Then enter VDP(2)=8 very carefully (you won't see what you are doing!), and you will be returned to the screen that BASIC now accepts, to see the results of your previous keying-in. To return your MSX to its original state, simply list the program and change lines 40 and 50 so that BASE(0)=0 and VDP(2)=0 and RUN again. All will be restored to 'normal'.

### 4.3.4 Switching PATTERN GENERATOR Tables

Obviously before switching the PATTERN GENERATOR Table to a new address, it is necessary to load up the Table with the desired patterns. Creating patterns has already been discussed in Section 2: the important thing to remember is where your new PATTERN GENERATOR Table is going to start.

Program 4 of Appendix B demonstrates how the PATTERN GENERATOR can be re-addressed: although this Program operates in Screen Mode 1, it can also be used for demonstration purposes in Mode 0 provided VDP(4) is correctly reset to '1' on return to the initialised state. One can remain with the 'second' character set by adding a STOP in the Program after VDP(4) has been set for the new PATTERN GENERATOR Table address.

However, because the Program doesn't load a full character set into the new PATTERN GENERATOR Table area, keys for undefined characters will not produce a result on the screen: the only character numbers defined by this program are those for the CAPITAL alphabet letters. They have been defined as their lower case counterparts, and so you will get lower case letters when - and only when - typing the CAPITALS. After running this program, you can return to the original PATTERN GENERATOR Table by typing in VDP(4)=1.

For further information on switching PATTERN GENERATOR Table addresses, please refer to the discussion on VDP(4) in Section 1.2.2.

SECTION 5

# SCREEN MODE 1

## Text and limited Graphics

This Mode provides a screen format of 32 x 24 for text and graphic character applications, with Sprites and limited colour facilities. The MSX will always return to either this Mode or Mode 0 when BASIC enters the command level - i.e. for direct access to the screen from the keyboard. It is possible to obtain the features provided by the VDP for Mode 2 while 'in' this Mode: these are different from those provided by the MSX system when it initialises Mode 2 for its own use, and enable more characters to be defined, with very high colour resolution. Details are given in the Section on Mode 2: this Section deals only with the way the VDP operates in Mode 1, and the way the MSX initialises and uses the Mode.

## 5.1 SCREEN MODE SPECIFICATION

### 5.1.1 Screen Parameters

| | |
|---|---|
| Screen Size (Max): | 32 columns x 24 rows |
| (Recommended): | 29 columns x 24 rows (to avoid loss at edges) |
| Character set (Max): | 256 |
| Character size: | 8 pixels x 8 pixels |
| Characters redefinable: | Yes |
| Colours available: | All, but blocks of eight characters have the same Foreground and Background colours. |
| Sprites: | Available |

### 5.1.2 MSX Initialisation of Mode 1

When Screen Mode 1 is selected using BASIC or by calling the ROM routine at 6F hex, the VDP Registers are initialised by the MSX so that the Tables in VRAM are set as follows:

| | | |
|---|---|---|
| PATTERN NAME: | 1800 hex<br>6144 dec | VDP(2)=6<br>BASE(5)=1800 hex |
| COLOUR: | 2000 hex<br>8192 dec | VDP(3)=80 hex<br>BASE(6)=2000 hex |
| PATTERN GENERATOR: | 0 | VDP(4)=0<br>BASE(7)=0 |
| SPRITE ATTRIBUTE: | 1B00 hex<br>6912 dec | VDP(5)=36 hex<br>BASE(8)=1B00 hex |
| SPRITE PATTERN: | 3800 hex<br>14336 dec | VDP(6)=7<br>BASE(9)=3800 hex |

Additionally, Bit 1 of VDP Register 0 and Bits 3 and 4 of VDP Register 1 are reset to '0' in order to select VDP Mode 1. During the initialisation for Mode 1, the MSX:

(a) Loads its own 256 character set into the PATTERN GENERATOR Table in VRAM from 1BBF hex in ROM, as determined by the contents of F920 hex (low byte) and F921 hex (high byte).

(b) Sets the line width to its default value - usually 29.

(c) Loads the first 32 bytes of the COLOUR Table in VRAM with the default value, F4 hex - giving White (15, or 'F' hex) characters on a Dark Blue (4) background. VDP(7) is loaded with the Border colour default value.

(d) IF Sprite size/magnification is specified, the SPRITE ATTRIBUTE and SPRITE PATTERN Tables are initialised as described in Section 3.3.2. Otherwise they are left alone.

This initialisation process takes place whenever 'SCREEN 1' is called from MSX BASIC, or the ROM routine at 6F hex is called from a machine code program. Other areas of VRAM are left unaffected, so if this Mode is called immediately after Screen 0 has been called, for example, there will still be a character pattern set in the VRAM locations starting at 800 hex (2048 decimal) - this being where the PATTERN GENERATOR Table is initialised for Screen Mode 0.

The MSX BASIC Graphic statements (DRAW, LINE, CIRCLE, PAINT, PSET, PRESET and POINT) are not operative in this Mode.

## 5.2 HOW MODE 1 OPERATES

The PATTERN NAME Table for Mode 1 is 768 bytes long, each byte 'mapping' to a screen location. The first byte of the Table determines the character displayed at the top left of the screen display, the next byte determines the character in the column next to it, and so on. The arrangement is shown in Figure 3.

During the active screen refresh period, the VDP examines each byte in the NAME Table in turn, to determine what should be placed in the corresponding location on the screen. Whatever value it finds, it takes to be a Character number or 'NAME'.

The VDP then looks at the PATTERN GENERATOR Table, to determine the shape for the required character number. Each character is defined by eight bytes, so if the character number is 65, say, then the VDP would look to the eight pattern bytes starting at 65x8 bytes from the beginning of the PATTERN GENERATOR Table.

At the same time, the VDP looks to the COLOUR Table, to determine the colours for the displayed character. The COLOUR Table in this Mode is 32 bytes long, and the colours of a character for display are determined by the most significant five bytes of the character number. So if the character number is 65 (41 hex), say, the top five bytes would give a value of '8':

0 1 0 0 0 0 0 0 1

The VDP looks at the resulting byte number in the COLOUR Table - in our example, byte 8. The top four bits of this byte's value determine the colour for all '1s' in the character's pattern (i.e., the Foreground colour), while the bottom four bits determine the colour of all the '0s' in the character's pattern (i.e. the Background).

From this, it is evident that eight characters share the same COLOUR Table byte. The colours for character numbers 0 to 7 are determined by the first byte (byte 0) in the COLOUR Table, for character numbers 8 to 15 by the second byte (byte 1) in the COLOUR Table, and so on. The byte number in the COLOUR Table can be found by integer division of the character number by eight.

The 'mapping' to the screen for this Mode is illustrated in Figure 11.

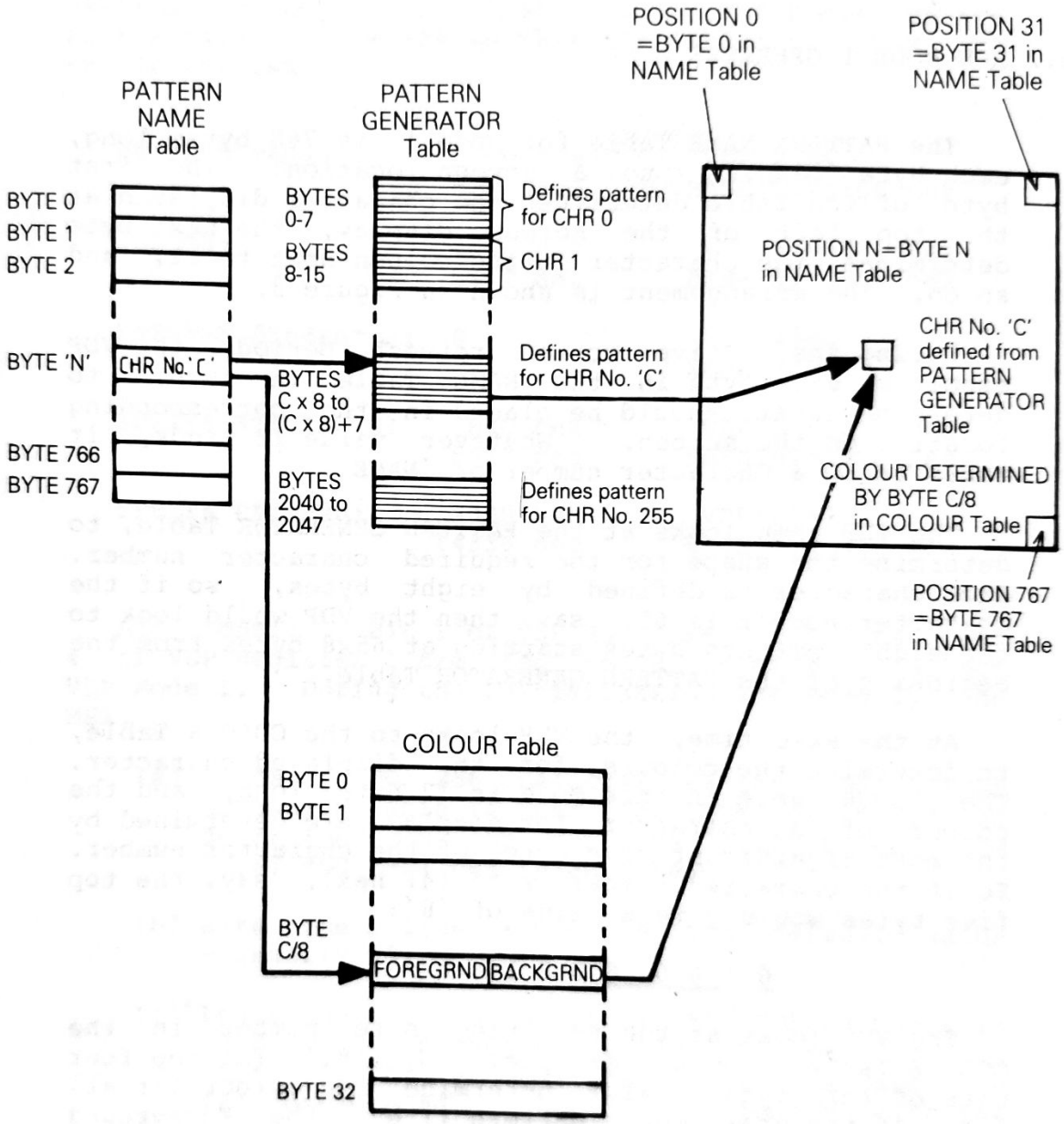



**Figure 11. Creating Mode 1 screen display**

## 5.3 MODE USAGE

### 5.3.1 Free VRAM Areas

In this Mode, the NAME Table occupies 768 bytes, the PATTERN GENERATOR Table 2048 bytes, the COLOUR Table 32 bytes, the SPRITE ATTRIBUTE Table 128 bytes, and the SPRITE PATTERN Table 2048 bytes. Bearing in mind where these Tables are initialised by the MSX, the arrangement in VRAM looks like this:

| | | |
|---|---|---|
| 0 - 2047 | decimal | PATTERN GENERATOR |
| 0 - 07FF | hex | TABLE |
| 2048 - 6143 | decimal | Not |
| 0800 - 17FF | hex | used |
| 6144 - 6911 | decimal | NAME |
| 1800 - 1AFF | hex | TABLE |
| 6912 - 7039 | decimal | SPRITE ATTRIBUTE |
| 1B00 - 1B7F | hex | TABLE |
| 7040 - 8191 | decimal | Not |
| 1B80 - 1FFF | hex | used |
| 8192 - 8223 | decimal | COLOUR |
| 2000 - 201F | hex | TABLE |
| 8224 - 14335 | decimal | Not |
| 2020 - 37FF | hex | used |
| 14336 - 16383 | decimal | SPRITE PATTERN |
| 3800 - 3FFF | hex | TABLE |

From this, you can see that there are areas within VRAM which are not used. If one chose not to use all the area allocated to Sprites, then this would release a further area within VRAM.

### 5.3.2 Switching NAME and PATTERN Tables

As with Mode 0, these 'free areas' can be used to set up alternative Tables. Switching from the initialised Table address to the alternative address is achieved by resetting the appropriate VDP Registers.

Note that each Table must 'sit' on a specific boundary address, as detailed in Section 1.2.2 on the VDP Registers and given in Appendix C.

A discussion on NAME and PATTERN Table switching is given in Sections 4.3.3 and 4.3.4 for Mode 0: the process for this Mode is similar, although of course the addresses of the initialised Tables are different. You'll also find simple examples of switching these Tables in Programs 3 and 4 of Appendix B.

Among the programs of Appendix B, you'll see several instances where the SPRITE PATTERN Table has be reset to the same address as the PATTERN GENERATOR Table for characters: this gives an easy way to set up an entire SPRITE PATTERN Table. This Table always 'sits' on an 800 hex boundary, and so a completely new Table could be defined in the Non-used areas of VRAM for this Mode, at 800 hex, 1000 hex, 1800 hex, 2800 hex or 3000 hex, without interfering with any of the other Tables. However, only 32 Sprites can be displayed on the screen at a time, and since a SPRITE PATTERN Table allows for 256 8-byte Sprite patterns or 64 32-byte Sprite patterns, one Sprite Table should always prove adequate.

### 5.3.3 Colour for Mode 1

The COLOUR Table in this Mode can sit on 40 hex boundaries, and since it is only 32 bytes long, a large number of COLOUR Tables can be set up within the free areas of VRAM. However, switching COLOUR Tables affects the colouring of all characters, and so a complete Table should be defined in every instance.

Also it should be remembered that the COLOUR Table controls the colours of characters, not screen locations, and so the value of switching COLOUR Tables is rather limited: it is probably easier to simply change the colours within the existing Table.

This can be achieved by VPOKEing the appropriate byte in the COLOUR Table with the desired colour data: if one uses hex notation, the most significant hex digit represents the foreground colour, and the least significant digit the background colour.

Thus, to change the colouring of the letter 'A' - character 65, we would first ascertain which byte we need to change in the COLOUR Table. The integer of 65 divided by 8 is 8, and so the byte required is at 8+8192 (the COLOUR Table start address), i.e. 8200. We would then VPOKE this address with the required data - for Dark Yellow (10, or A hex) on a Dark Red background (6), we would 'VPOKE 8200,&HA6'.

Every occurence of the letter 'A' on the screen would then be in the new colouring. So will the '@' character, and letters B, C, D, E, F, and G - since they all share the same colour information byte.

This makes life a little difficult if one wants to pick out certain words in a different colour - since any letters used in those words will be in the 'new' colour wherever they appear on the screen. One solution is to repeat the alphabet portions of the character set at, say, their 'relative' positions in the top section of the PATTERN GENERATOR Table: i.e., with the top bit set. Thus the pattern for 'A' - character 65 or 41 hex - would be repeated at character position 193 or C1 hex (65+128, or 41 hex + 80 hex).

If this were done for the entire alphabet (and punctuation marks), there would be two alphabet sets loaded into the PATTERN GENERATOR Table, each of which can be given its own colouring. To call the 'second' set, one would simply set the top bit high before printing to the screen. This could be achieved in a subroutine, with the line to be printed passed into the subroutine via a string variable.

The following program illustrates the whole process:

```
10  SCREEN 1
19  REM
20  REM  ** Load alphabet/punctuation marks **
21  REM
30  FOR I=32*8 TO 122*8
40   VPOKE I+(128*8),PEEK(&H1BBF+I)
50  NEXT
59  REM
60  REM  ** Now change the colour **
61  REM
70  FOR I=(128+32)/8 TO (128+122)/8
80   VPOKE 8192+I,&H16: REM Black on Dark Red
90  NEXT
99  REM
100 REM  ** Print line in original colour **
101 REM
110 LOCATE 5,5
120 PRINT "Original colours"
129 REM
130 REM  ** Now print in new colours **
131 REM
140 LOCATE 5,7
150 SC$="New colours!"
160 GOSUB 200
170 PRINT SC$
180 STOP
189 REM
190 REM  ** THE SUBROUTINE **
191 REM
200 FOR P=1 TO LEN(SC$)
210  MID$(SC$,P,1)=CHR$(ASC(MID$(SC$,P,1)) OR 128)
220 NEXT
230 RETURN
```

The various values in the program have been written so that the process can be easily understood. Lines 30 and 40 load from character 32 to character 122 into the VRAM pattern area for characters 160 to 250. Note that the character set in ROM is used rather than VPEEKing the set already existing in VRAM.

Lines 70 to 90 change the colour of the second alphabet set that has just been loaded, while lines 110 to 170 locate and print from the two sets onto the screen.

The subroutine at line 200 takes every character in the string variable SC$ in turn, and 'ORs' it with 128 (80 hex) - which is one way to set the top bit. Note that to do this, it is necessary to first obtain the ASCII value of the character - and then, having updated the character number, it is turned back to its character pattern within the string variable SC$. The string variable with its updated character patterns is then printed to the screen on return from the subroutine.

VDP Mode 2 provides more scope for those who wish to use different coloured characters. However, the way that the MSX initialises the Screen for Mode 2 inhibits its use in the same way as Mode 1: the good news is, you can re-initialise Screen 2 so that it operates in the same way as Mode 1 - with direct access to the screen from the keyboard, and with the potential to create 768 different character patterns. Also, each of the eight lines that form a character pattern can be displayed in two different colours - so one character can have a total of 15 different colours in its make-up. Details are given in Section 6 of this book.

### 5.3.4 Screen Width

When using MSX BASIC 'PRINT' statements, the number of characters that can be printed on one line of the display area is determined by the WIDTH statement. This is initialised to 29 columns on most MSX machines, thus leaving three character positions blank at the edges of the screen.

It is possible to display characters outside the area set by WIDTH by using the VPOKE command, since the PATTERN NAME Table is always addressing the entire active display area. Characters so placed will not affect any BASIC command statements made along the same line: BASIC operates only on commands made within the area set by WIDTH: anything outside this area is ignored and remains permanently on the screen until it is cleared either by 'CLS' or SHIFT-HOME.

SECTION 6

# SCREEN MODE 2

High Resolution Graphics
or
High Resolution Text

When initialised by the MSX, this Mode provides high resolution graphics with limited text facilities, and can be entered only during program or direct statement execution. The Mode can also be used in a similar manner to Screen Mode 1, to provide a larger range of characters and greater colour resolution than Mode 1. Used this way the screen can be accessed direct from the keyboard for program development, but the MSX BASIC graphic statements cannot be used. This Section deals with both methods of operation.

## 6.1 SCREEN MODE SPECIFICATION

### 6.1.1 One Mode, Two Displays

Normally, the MSX BASIC routines only allow this Mode to be entered during program execution, or the execution of direct statements from Modes 0 or 1. At the termination of the program and on return to the BASIC command level, the MSX returns automatically to the Text Mode that was in operation prior to entering Mode 2.

The reason for this lies in the way that the Mode is set up by the MSX. This is done in such a way that it is not possible to print characters direct to the screen from the keyboard: for program development, it is necessary to be in one of the Text Modes - hence the return to one of the Text Modes on exit from Mode 2.

The set up of Mode 2 by the MSX enables it to provide sophisticated graphics routines - such as the very powerful macro 'DRAW' statement, which enables a

complete, complex figure to be produced from one string variable.

It is possible to print text on the screen whilst in this Mode, but to do so using BASIC, it is necessary first to ´OPEN "GRP:" AS #1´ before using ´PRINT#1, string´ statements. It is also necessary to set the location for PRINTing to the screen by (for example) using the PRESET statement.

The interesting point here is that the text can be positioned to start at any pixel location on the screen - not necessarily at a character position. So in that respect, one has greater control of where PRINT statements are to be displayed. (The pixel location defines the position of the top left corner of the character).

Because of the way text is presented to the graphics screen, it is not possible to redefine the characters: you can only use the character set as provided by the MSX.

However, it is possible to initialise the VDP into its ´Mode 2´ - but without the use of the MSX graphics facilities. When used this way, one can define up to 768 different characters, and each character can have every one of its pattern lines in two different colours - so one character can be coloured with the entire colour range.

Furthermore, the screen can be accessed direct from the keyboard - for program development - as if it were a ´Text Mode´. . This is, in fact, the ´standard´ way of using the VDP: it is the designers of the MSX system that have adapted this usage to provide for high resolution graphics.

The programmer therefore has a choice of how Mode 2 is to be used: either in the manner it is set up by the MSX, with extensive graphic facilities, or in the manner of Mode 1, but with a potentially larger character set and much greater colouring facilities.

Both methods will be discussed, but first let us have a look at the way the VDP operates in Mode 2.

### 6.1.2 VDP Screen Parameters

| | |
|---|---|
| Screen size (max): | 32 columns x 24 rows |
| (Recommended): | 29 columns x 24 rows (to avoid loss at edges) |
| Character set (Max): | 768 |
| Character size: | 8 pixels x 8 pixels |
| Characters redefinable: | Yes (but not as initialised by the MSX) |
| Colours available: | All: each line of a character pattern can be individually defined for Foreground and Background colour (but not as initialised by the MSX). |
| Sprites: | Available |

### 6.1.3 How the VDP Operates in Mode 2

The VDP is set to Mode 2 when Bit 1 of VDP Register 0 is set to '1', and Bits 3 and 4 of VDP Register 1 are reset to '0'. When this happens, the VDP's system for evaluating the Base address of the PATTERN GENERATOR Table and the COLOUR Table is different. In this Mode, both of these Tables are 6144 bytes long, and both can sit only on an 8k boundary.

Whereas in the other Modes, three bits define the VRAM Base address for the PATTERN GENERATOR Table (see Section 1.2.2), in this Mode only one Bit - Bit 2 of VDP Register 4 - defines the Table's start point. The other two Bits - 0 and 1 - must be set to '1s' for the VDP to function fully in the Mode.

Similarly in Mode 2 only one Bit defines the VRAM Base address for the COLOUR Table - Bit 7 of VDP Register 3. The remaining Bits (0 to 6) must be set to '1s' for the VDP to function fully in the Mode.

Both Tables can have only one of two possible locations - at address 0 in VRAM, or at address 8192 (2000 hex), as selected by the Bits mentioned above. The MSX in fact initialises the PATTERN GENERATOR Table to address 0, and the COLOUR Table to address 8192 (2000 hex).

The PATTERN NAME Table is 768 bytes long, each byte 'mapping' to a screen location in exactly the same manner as Mode 1 (see Figure 3). Thus the first byte of the Table determines the character displayed at the top left position on the screen display, the second byte determines the character displayed in the column next to it, and so on.

So far, there would appear to be no difference between this Mode and Mode 1. However, in Mode 1 the PATTERN GENERATOR Table is only 2048 bytes long, catering for the 8-byte patterns of 256 characters. In this Mode, the PATTERN GENERATOR Table is 6144 bytes long, thus catering for 768 characters.

Obviously the maximum value that a byte in the NAME Table can have is 255 (FF hex), so the question must arise 'how does the NAME Table address a character with a value greater than 255'. The short answer is - it doesn't.

What in fact happens is the screen is divided into thirds. The top third of the screen - 256 locations (768/3) - is represented by the first 256 bytes of the NAME Table. Any character number called within this section of the NAME Table is defined by the character pattern in the first 2048 bytes of the PATTERN GENERATOR Table.

The 'middle' third of the screen, represented by bytes 256 to 511 in the NAME Table, calls up character patterns in the second 2048 bytes of the PATTERN GENERATOR Table. Similarly, the bottom 'third' of the screen , represented by bytes 512 to 767 in the NAME Table, calls up character patterns in the third block of 2048 bytes in the PATTERN GENERATOR Table. The three blocks of 2048 bytes in the PATTERN GENERATOR Table make up the total Table length of 6144 bytes.

Thus, if byte 0 in the NAME Table has the value '65', then the particular character pattern placed at the top left of the screen will be that defined by bytes 65x8 to (65x8)+7 in the PATTERN GENERATOR Table. If byte 256 in the NAME Table has the value '65', then the particular character pattern placed on the left edge of the screen eight lines down from the top (256/32) will be that defined by bytes 2048+(65x8) to 2048+(65x8)+7 of the PATTERN GENERATOR Table.

This, obviously, can be a completely different character pattern to that appearing in the top third of the screen. Similarly, yet a different character pattern again can be defined for the bottom third of the screen - yet called by the same number or Name, '65'. This time, the pattern is defined by bytes 65x8 to (65x8)+7 in the third section of the PATTERN GENERATOR Table - which starts at byte 4096 from the Table's Base address. (See Figure 12).

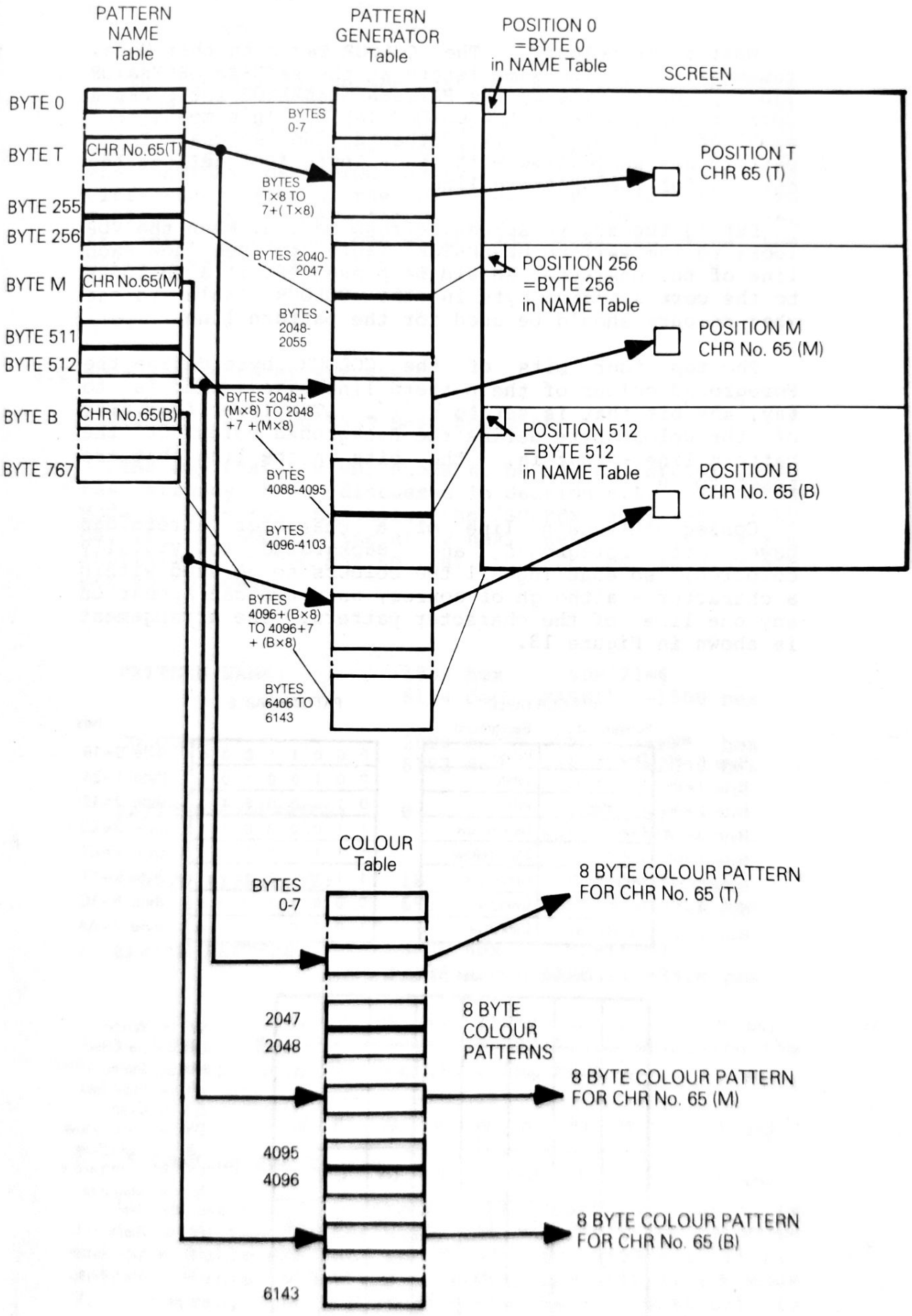



**Figure 12. How the VDP creates Mode 2 screen display**

What about colour? The COLOUR Table in this Mode, remember, is the same length as the PATTERN GENERATOR Table: every byte in the PATTERN GENERATOR Table has a corresponding byte in the COLOUR Table. This means that each of the eight bytes that define a character's pattern has associated with it a byte that defines the colours for that pattern line.

During the active screen refresh period, when the VDP looks to the PATTERN GENERATOR Table to see how each line of the character should be presented, it also looks to the corresponding byte in the COLOUR Table to see what colours should be used for the pattern line.

The top four bits of the COLOUR byte define the Foreground colour of the pattern line byte - that is to say, any bit that is set to a '1'. The lower four bits of the colour byte define the Background colour of the pattern line - that is, the bits in the line that are reset to '0'.

Consequently each line of a character pattern can have its Foreground and Background individually coloured, so enabling all the colours to be used within a character - although of course, only two can appear on any one line of the character pattern. The arrangement is shown in Figure 13.

COLOUR TABLE

| | hex | Foreground | Background |
|---|---|---|---|
| Byte 0 | 6F | Dark Red | White |
| Byte 1 | 8F | Medium Red | White |
| Byte 2 | 9F | Light Red | White |
| Byte 3 | 7A | Cyan | Dark Yellow |
| Byte 4 | 5B | Light Blue | Light Yellow |
| Byte 5 | 0E | Magenta | Grey |
| Byte 6 | 17 | Black | Cyan |
| Byte 7 | 34 | Light Green | Dark Blue |

PATTERN TABLE

| | | | | | | | | hex |
|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Byte 0=18 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | Byte 1=24 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | Byte 2=42 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | Byte 3=C3 |
| 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | Byte 4=E7 |
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 1 | Byte 5=F7 |
| 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | Byte 6=1C |
| 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | Byte 7=AA |

COLOURS OF CHARACTERS ON SCREEN

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Line 0 | W | W | W | DR | DR | W | W | W |
| Line 1 | W | W | MR | W | W | MR | W | W |
| Line 2 | W | LR | W | W | W | W | LR | W |
| Line 3 | C | C | DY | DY | DY | DY | C | C |
| Line 4 | LB | LB | LB | LY | LY | LB | LB | LB |
| Line 5 | M | M | M | M | G | M | M | M |
| Line 6 | C | C | C | B | B | C | C | C |
| Line 7 | LG | DB | LG | DB | LG | DB | LG | DB |

W = White
DR = Dark Red
MR = Medium Red
LR = Light Red
C = Cyan
DY = Dark Yellow
LB = Light Blue
LY = Light Yellow
M = Magenta
G = Grey
B = Black
LG = Light Green
DB = Dark Blue

**Figure 13. How characters are coloured, VDP Mode 2**

The correlation between the bytes in the COLOUR Table and the bytes in the PATTERN GENERATOR Table means that the colour byte for a given pattern line can be found by simply adding or subtracting 8192 decimal (2000 hex) to the PATTERN GENERATOR address - 8192 always being the difference between the Base addresses for the two Tables. In the MSX, the PATTERN GENERATOR Table is set at zero, and so one would add 8192 to find the corresponding colour byte.

## 6.2 MODE 2 AS USED BY MSX BASIC

### 6.2.1 Initialising VRAM Base Addresses

The way that the VDP needs to be set up for this Mode has already been discussed in Section 6.1. When the Mode is selected by using the 'SCREEN' statement or by calling the ROM routine at 72 hex, the VDP Registers are initialised by the MSX so that the Tables in VRAM are set as follows:

| | | |
|---|---|---|
| PATTERN NAME: | 1800 hex<br>6144 dec | VDP(2)=6<br>BASE(10)=1800 hex |
| COLOUR: | 2000 hex<br>8192 dec | VDP(3)=FF* hex<br>BASE(11)=2000 hex |
| PATTERN GENERATOR: | 0 | VDP(4)=3 *<br>BASE(12)=0 |
| SPRITE ATTRIBUTE: | 1B00 hex<br>6912 dec | VDP(5)=36 hex<br>BASE(13)=1B00 hex |
| SPRITE PATTERN: | 3800 hex<br>14336 dec | VDP(6)=7<br>BASE(14)=3800 hex |

* NOTE: See Sections 6.1 and 1.2.2 regarding the values in these VDP Registers for Mode 2.

### 6.2.2 Loading the Tables

In Modes 0 and 1, the PATTERN GENERATOR Table is loaded with the MSX character set, and all the bytes in the NAME Table are set to zero, so giving a 'blank' screen. This is NOT the case when MSX initialises Mode 2. Instead, the MSX loads the Tables with data in readiness for accepting and operating on the BASIC graphic statements - CIRCLE, DRAW, LINE PSET and so on.

During the initialisation process, the MSX:

(a) Fills the entire PATTERN GENERATOR Table (6144 bytes) with zeroes.

(b) Fills the entire COLOUR Table to give 'Transparent' (i.e. the Border colour) characters on the currently named Background colour. Thus, if the current colours are White Foreground, Dark Blue Background and Cyan Border, then every byte in the Table will have the value 04 hex, the top nibble '0' giving Transparent - or the Border colour, and the bottom nibble '4' giving Dark Blue.

(c) The first 256 bytes of the 768 byte NAME Table are loaded with the values 0 to 255 decimal respectively. Bytes 256 to 511 of the NAME Table are loaded with the values 0 to 255 respectively. Bytes 512 to 767 of the NAME Table are loaded with the values 0 to 255 respectively.

(d) The Sprite Tables are left in their current state provided that Sprite size/magnification is unspecified. Otherwise they're re-initialised as stated in Section 3.3.2.

### 6.2.3 How MSX BASIC uses Mode 2

The significance of the way the Tables are loaded will now be explained. You will see that the top, middle and bottom third areas of the NAME Table are each loaded with consecutive values 0 to 255. In the earlier part of this Section, it was explained that each third of the screen looks to its own 'area' within the PATTERN GENERATOR Table to determine the pattern for the character being 'NAMEd'.

From the way the NAME Table is loaded with data, it can be seen that the first byte in a screen area always points to character number (or 'name') 0, the second byte to character number 1, the third to character number 2 and so on through to the last byte in each 'one third' area of the screen, which points to character number 255.

The patterns for all characters are intially set to zero, consequently the screen is initially 'blank'. When a graphic command is made, the appropriate character pattern areas in the PATTERN GENERATOR Table are filled with the necessary data to produce the required graphics. Since these pattern areas are 'permanently' pointed to by the NAME Table, the 'patterns' then appear on the screen.

To show this effect, try the following two short programs.

```
10 SCREEN 2:COLOR 15,4,7
20 FOR I=6144+16 TO (6144+768)-16 STEP 32
30  VPOKE I,1
40 NEXT
50 LINE (20,70)-(250,74),10,BF
60 LINE (20,100)-(250,104),6,BF
100 GOTO 100
```

Enter and RUN this program, and you will see that two thick coloured lines are drawn across the screen, but both have a 'one character' gap in the middle. This is a result of lines 20 to 40. In these program lines, the 'centre' bytes all the way down the NAME Table are reset to the value '1'.

So they no longer 'point' to the correct area in the PATTERN GENERATOR Table for the thick coloured lines to be drawn by the MSX. For example, the centre byte in the top line - byte 16 in the NAME Table - should be pointing to the 16th set of character pattern bytes in the PATTERN GENERATOR Table. But it isn't - because it has been reset to point to the pattern bytes for 'character 1'.

Program lines 50 and 60 draw the 'boxes' across the screen of course (refer to your Owner's Manual for details on using this statement), while line 100 ensures that the MSX doesn't leave Screen Mode 2 at the end of the program and so lose the display.

Break the program by using 'CONTROL-STOP', then add the following line:

```
70 LINE (8,0)-(15,190),1,BF
```

This draws a 'block' line right down the screen, one character wide (8 pixels), starting at the NAME Table byte that's calling character number 1. But the centre line of bytes throughout the NAME Table has also been directed to character number 1 - so this new program line produces TWO vertical block lines on the screen.

If the first part of Line 70 is changed to read 'LINE (8,8)' instead of 'LINE (8,0)', then the top third of the screen doesn't carry the central block vertical line: character '1' in this third of the screen hasn't been addressed by the statement, and consequently the PATTERN TABLE for this third is not carrying data for character '1'.

No mention has been made so far about the COLOUR Table. The colour for each of the block lines that are drawn on the screen is specified within the 'LINE' statement. The MSX loads this colour data into the 'top' nibble of the COLOUR Table bytes that correspond to the PATTERN Table bytes. If no colour is specified, then the current 'Foreground' colour (as stored at F3E9 hex) is loaded into the top nibble.

The way that the MSX ROM routines select which bits of the pattern-line bytes have to be set for a graphic statement is quite complex. It is sufficient to know that the appropriate pattern lines and the bits are selected according to the 'character' locations on the screen, as addressed from the NAME Table set up.

Take, for example, a 'PSET (24,3),1' statement. This tells the MSX that the pixel 25 across from the left and four down from the top is to be set in the colour Black. (Remember that the start point at the top left corner is pixel 0,0, which is the top left corner of 'Character number 0'). Pixel 24,3 thus lies in the fourth character position along the top of the screen.

The fourth character position in the NAME Table is initialised by the MSX to point to the pattern for Character number 3 (the first position is Character 0). Pixel 24 is, in fact, the leftmost bit of the character pattern line. The '3' tells the MSX that the particular pattern line is the fourth one in the set of eight pattern bytes.

Thus, the leftmost bit (most significant) of the fourth pattern line of the pattern set for Character number 3 is set to a '1'. The corresponding COLOUR Table byte has its top nibble set to '1' - for Black.

If, subsequently, a 'PSET (25,3),15' statement is made - to set the pixel next to the previous one to White - then in the PATTERN GENERATOR Table, the TWO most significant bits of pattern line 4 for character number 3 would be set. Now however, the corresponding COLOUR Table byte would have its top nibble set to 15 (F hex) - for White, and since both the pixels that have just been set lie in the same pattern line, both pixels would turn 'White' on the current background colour.

When ALL the pixels for a character pattern are set, any further 'attempts' to set one of the pixels to a new colour results in the LOWER (Background) nibble of the corresponding COLOUR byte to be set to the new colour, and the appropriate bit in the pattern line to be reset to '0' to show that colour.

The following programming examples will help to demonstrate this feature of the way the MSX operates on the high resolution graphic screen.

First, enter and RUN the following short program:

```
10  SCREEN 2:COLOR 15,4,7
20  OPEN"GRP:" AS #1
30  LINE (82,0)-(95,0),6
40  GOSUB 500

.200 GOTO 200

500 PRESET(24,100)
510 FOR I=0 to 7
520  PRINT #1,HEX$(VPEEK(8192+80+I));" ";
530 NEXT:RETURN
```

Line 30 draws a Red line at the top of the screen from pixel '82' to pixel '95'. This means that the first pattern line for Character number 10 will have its lower six bits set, and the first pattern line for Character 11 will have all its bits set. The subroutine at Line 500 prints out the hex values for the COLOUR Table bytes that correspond to each line of the pattern for Character 10. As you will see when the program is run, the first pattern line has its set bits coloured Red ('6' - the Foreground nibble), and the reset or zero bits coloured Blue ('4' is the Background nibble).

Now break the program ('CTRL-STOP'), delete Line 40, and add Lines 50 and 60:

```
50  LINE (83,0)-(83,180),1
60  GOSUB 500
```

When RUN, this draws a vertical line down through Character 10 to near the bottom of the screen: notice that the Foreground nibble of the COLOUR bytes is now '1' - for Black, and that the top pattern line for Character 10 is no longer Red - but defined by the new colour, Black.

Now break the program and change Line 30 to read:

```
30 LINE (80,0)-(96,0),6
```

and RUN again. Now we have drawn the Red line (Line 30) from the beginning of the character position - i.e. all the bits of the first pattern line are set. This time, drawing the vertical Black line down the screen - in Line 50 - results in the COLOUR byte for the first pattern line of Character 10 holding the value 61 hex. The 'new' colour - Black - has been placed in the LOWER nibble of the COLOUR byte, and if you were to examine the first pattern line for Character 10, you'd see that one bit has been reset to zero - to display the Black as a Background colour. For all the other COLOUR bytes associated with Character 10, Black is still defined as the Foreground colour (14 hex).

While this description has covered the 'simplest' graphic statements, all operate in a similar manner. Once the necessary screen location calculations have been made by ROM routines (for the 'CIRCLE' statement, for example), the necessary pixels are turned on and their colour is defined by the top (Foreground) nibble of the associated COLOUR byte.

If they are ALL already turned on, any new requirement results in the COLOUR byte being 'amended' so that the new colour is displayed as a background colour, with the relevant bit(s) of the pattern byte being reset to '0' to display that colour.

The colour of a set pixel always changes to the last one specified for the particular pattern line. If no colour is specified, then it will be changed to the current foreground colour for the screen.

### 6.2.4 Text on MSX Screen 2

There are two ways that 'text' characters can be placed on Screen Mode 2, as it is initialised by the MSX. One way has already been mentioned: first OPEN the graphic screen as a 'file' - 'OPEN"GRP:" AS #1' - and then use 'PRINT #1,"string"' statements. The other method involves VPOKEing the required character pattern into the PATTERN GENERATOR Table. Let us first examine the 'PRINT' method.

The required text is placed on the screen at the currently addressed pixel location, this being the top left corner of the first character in the string. The location can, of course, be defined by the 'PSET' or PRESET' statements.

When this procedure is used, MSX ROM routines load each desired character in the string into a 'Work area', then transfer it to the current pixel location on the screen. Since this 'pixel location' may not necessarily coincide with the top left corner of a screen character position, a 'masking' system is used to SET the correct bits in the appropriate bytes of the PATTERN GENERATOR Table.

Bits already set to '1' in the PATTERN GENERATOR Table merely have their colour changed. Where there is a '0' bit in the Character pattern being 'printed' to the screen, the PATTERN GENERATOR bit is left unaltered: if it is set, it stays set. The corresponding COLOUR bytes have their top nibble set to the current foreground colour for the screen, so any previously set bits in an affected pattern line will take on the colour of the PRINTed Character.

This process means three things:

(a) Only the MSX's own Character set can be displayed on the screen in Mode 2 when using 'PRINT 1' statements.

(b) Text characters PRINTed to the screen cannot be cleared by overwriting with 'space' characters. These do not set any bits, and so leave the already set bits in the PATTERN GENERATOR Table as they are. This is done so that graphic patterns are not destroyed when PRINTing to the screen, but continue to appear through the 'gaps' in the text.

(c) If PRINTing a character to the screen results in all the bits of a pattern line being set, then those bits that were previously set will take on the colour of the PRINTed text - i.e., the current foreground colour for the screen.

Now let us look at the second way of setting text and character patterns on the screen, which involves VPOKEing.

For this, it is necessary to determine first the character position on the screen where it is desired to place the character pattern. From this, the location in the PATTERN GENERATOR Table can be deduced. For example, suppose it is desired to place three characters on the screen, all in the column number 6, but in rows 2, 12 and 20 respectively.

Bearing in mind the way the screen is mapped by the NAME Table, the first character will have to be defined in the PATTERN GENERATOR Table in the eight bytes starting at: 2x32 (the number of characters in the preceding rows) PLUS 6 (to give the column in the third row), multiplied by eight (the number of bytes in a character pattern definition). In other words, the pattern for the character to be placed at Column 6 of Row 2 will have to start at byte 560 in the PATTERN GENERATOR Table.

Similarly, the second and third characters will have to start at bytes 3120 - ((12x32)+6)x8 - and 5168 - ((20x32)+6)x8 - respectively from the start of the PATTERN GENERATOR Table.

The eight bytes making up each of the required character patterns are then VPOKEd into the eight addresses starting from their respective start locations. This means of course that programmers can either define their own characters, or use the characters in the MSX by PEEKing the appropriate section in ROM.

The ROM character set starts at address 1BBF hex (7103 decimal). Thus, to place the letter 'A' (Character 65 in the MSX set) in the three locations just discussed:

```
10 SCREEN 2:COLOR 15,4,10
20 FOR I=0 TO 7
30  D=PEEK(&H1BBF+(65*8)+I)
40  VPOKE 560+I,D
50  VPOKE 3120+I,D
60  VPOKE 5168+I,D
70 NEXT
80 GOTO 80
```

The characters so VPOKEd will have their Foreground set to Transparent (i.e., the Border colour) and Background as set on initialisation of Screen 2, or as changed by graphics statements. To colour the characters differently, the COLOUR Table bytes corresponding to the PATTERN GENERATOR Table bytes will have to be loaded with the required colour information. The appropriate COLOUR Table address can be found by adding 8192 to the PATTERN GENERATOR Table address.

Thus, to colour the 'A' at the top of the screen Red on Yellow, one would VPOKE VRAM addresses 560+8192 to 560+8192+7 with the value 6A hex. This can be done in the program above by adding

```
45  VPOKE 560+8192+I,&H6A
```

Since it takes eight bytes to define the entire colour for a character, the programmer can of course define the foreground and background colour of each line individually - and so get a beautifully multicoloured character pattern.

This procedure for getting characters onto the graphics screen consumes more program space than the 'PRINT #1' method - although space can be saved by judicious use of subroutines and multi-statement lines (not used in this book, for the sake of clarity). Furthermore, characters can be placed only in 'screen character' locations, and whatever is VPOKEd into VRAM will overwrite any graphics or colours currently being displayed - unless 'masking' routines are written!.

For example, to have a White character on an existing colour background, one could 'VPOKE ca,VPEEK(ca) AND &HF OR &HF0', where 'ca' is the COLOUR Table address. ANDing with 0F hex retains the lower nibble of the COLOUR byte, while ORing with F0 hex puts the code for White into the top nibble.

So, in return for the possible 'disadvantages' of this method, the programmer is able to define and colour characters as desired.

## 6.3 MODE 2 AS A TEXT SCREEN

### 6.3.1 How to Initialise the VDP

If you compare the VRAM Base addresses set up by the MSX for Mode 2 (page 81) with the addresses set up for Mode 1 (page 68), you will see that they are the same. VDP Registers 3 and 4 are not set to the same values, however. For Mode 2, only one bit of each of these Registers is significant in the determination of the Base addresses: the other 'selection' bits must be set to '1s' for the full implementation of the Mode.

Obviously the Mode setting bits of VDP Registers 0 and 1 will not have the same values for both Modes - but simply setting these to switch the VDP into Mode 2 is not sufficient to select the Mode.

The following program illustrates one method of setting the VDP to implement Mode 2 so that it operates in the same manner as Mode 1 - but with the additional features mentioned at the beginning of Section 6.

> Note that it will also be necessary to load VRAM with data before the Mode can be fully utilised: this will be discussed Section 6.3.2

```
10 SCREEN 0:SCREEN 1
20 VDP(0)=VDP(0) OR 2
30 VDP(1)=VDP(1) AND &HE7
40 VDP(3)=&HFF
50 VDP(4)=3
```

Machine code programmers can use a similar technique: for them, however, it would not be necessary to go through the procedure of first initialising Screens 0 and 1: only Screen 1 need be initialised.

Why are both Screens 0 and 1 initialised? (Line 10). This is to save a little time on initialising the 'new' Mode 2. You will recall that when Modes 0 and 1 are initialised by the MSX, the MSX Character set is loaded into the PATTERN GENERATOR Table.

For Mode 0, the set is loaded into the 2048 bytes starting from VRAM address 2048 - the location of the PATTERN GENERATOR Table Base for that Mode. For Mode 1, the set is loaded into the 2048 bytes starting from VRAM address 0, without overwriting or clearing the data held in bytes 2048 to 4095. Thus Line 10 usefully loads the MSX Character set twice into VRAM.

For Mode 2, it needs to be loaded three times - the PATTERN GENERATOR Table in Mode 2 is 6144 bytes long, remember. If there is no set in the last 'third' of the Table, it will not be possible to display characters sensibly in the lower third of the screen when it is initialised this way: it can be re-intialised differently to use only one or two character sets over the screen - see Section 6.3.4.

Enter and RUN the program, and you will see that when the cursor is moved to the middle third of the screen, it 'vanishes', although tapping keys produces characters on the screen. In the lower third of the screen, there will be no cursor, and no 'sensible' result from tapping the keys: the characters are not displayed since a proper character set is not in that part of the PATTERN GENERATOR Table - yet.

The cursor 'vanishes' from the screen since the MSX still 'thinks' it is in Mode 1. To create the cursor, the MSX 'images' the content of the actual character at the cursor location, and loads the resulting pattern into character number 255. For Mode 1, it expects character 255 to be at the end of the first 2048 bytes of the PATTERN GENERATOR Table. Character 255 in the middle and the bottom areas of the screen are, at this stage, simply 'blanks'.

The colours of characters displayed in the middle third of the screen when keys are pressed may well be varied, depending on the particular make of MSX. Equally, some characters in varied colours may be obtained in the lower third of the screen when keys are pressed. This will be the result of residual data entered into the PATTERN GENERATOR Table and COLOUR Table areas during the 'wake-up' display of the machine (the Sony Hit Bit, for example), or as a result of previous programming.

Program Lines 20 to 50 set the VDP so that it will operate in its 'full' Mode 2 state (see Section 6.3.4). As far as the MSX is concerned, however, it is operating under Mode 1 conditions. But more needs to be done...

### 6.3.2 Getting VRAM ready

The next stage in completely preparing the screen is to load a character set into the lower third section of the PATTERN GENERATOR Table, to render this area 'visible'. The following program lines will load the MSX character set (add them to the previous program, so that you can, if you wish, save it all for future use).

```
60 FOR I=0 TO 2047
70  VPOKE 4096+I,PEEK(&H1BBF+I)
80 NEXT
```

This is not the fastest of programs - when you RUN it, you will understand why Line 10 in the original setting-up program was introduced - it cuts the time to load the character set by about half.

Loading from the Character set in ROM also saves a little time over loading from a set already in VRAM. While the program is running, provided that you have not cleared the Key Function display from the bottom of the screen, you should see the Key words appear as the letter characters are entered into VRAM.

Machine code programmers can use the Block Data Move ROM routine at 5C hex to perform an almost instantaneous load (see 'Writing to VRAM' in Section 1.2.1). All three 'thirds' of the PATTERN GENERATOR Table can be filled in microseconds this way: hence there is no need to initialise Screens 0 and 1 to perform the task.

The MSX will now be in a position to display at any point on the screen data that is entered from the keyboard. However, a cursor will still be missing from the middle and bottom thirds of the screen. This is easily rectified:

```
90  FOR I=0 TO 7
100  VPOKE 4088+I,&HFF
110  VPOKE 6136+I,&HFF
120 NEXT
```

You can, of course, simply enter RUN 90 to save the whole program from being run just to enter this data. This last program segment creates a 'block' for a cursor: it will not be a true cursor as appears at the top of the screen - i.e. it won't image the character it is placed over. Nevertheless, it does enable you to see where the current cursor location is on the screen.

Now for the colour. The next program segment is written to demonstrate the three areas of the screen controlling the different character sets: you could, obviously, choose to load the entire screen with one colour.

```
130 FOR I=0 TO 2047
140  VPOKE 8192+I,&HF4
150  VPOKE 8192+2048+I,&HB6
160  VPOKE 8192+4096+I,&H1B
170 NEXT
```

As in previous programs, the VPOKE addresses are written so that you can see how they are derived. Machine code programmers can use the Block Fill ROM routine at 56 hex (see Section 1.2.1).

When this segment is RUN, you will see each character on the screen change colour as the COLOUR Table is being

filled with data. This, again, shows that it is characters and not screen locations that are coloured when the VDP is used this way.

The three colour bands that appear across the screen after this program has been run clearly show the three areas in which each character set - and associated colours - operate independently.

The screen is now initialised for use - although not, perhaps, the way you´d like it to be from the colour or character definition point of view. But now that you know how, you can experiment with your own colours and character patterns.

In early ´experiments´, it is probably wisest to leave the alphabet and punctuation parts of the character sets intact, so that you can ´see´ what you are doing. This isn´t necessary in order to enter program lines, however, as long as you know that you have pressed the right keys(!): the MSX looks at the character NUMBER, remember, not its shape. So if you decide to make the ´?´ character a different shape in, say, the bottom third of the screen, then pressing the ´?´ key and a "string" in the bottom third will result in the string being printed, even though the string is preceded by your new character.

### 6.3.3 Using the ´Text´ Mode 2

Initialised as described in the last few pages, the VDP operates as described in Section 6.1.3, which gives a good idea of how the Mode can be used. The MSX ´behaves´ as though it is in Screen 1 - in other words, the graphics commands are not available.

If you study the length of each VRAM Table and its Base address, you´ll see that there isn´t very much VRAM space available for ´Table switching´.

| | |
|---|---|
| 0 - 6143 dec | PATTERN GENERATOR Table |
| 6144 - 6911 dec | NAME Table |
| 6912 - 7039 dec | SPRITE ATTRIBUTE Table |
| 7040 - 8191 dec | Not used |
| 8192 - 14335 dec | COLOUR Table |
| 14335 - 16383 dec | SPRITE PATTERN Table |

A fairly busy VRAM! In any event, neither the PATTERN GENERATOR nor the COLOUR Tables can be moved to other VRAM locations (unless they are ´swapped´). The only Table that can sensibly be switched to another location in VRAM is the NAME Table: this could be switched to location 7168 dec (1C00 hex), by loading VDP Register (2) with ´7´. The comments given in Section 4.3.3 regarding the use of BASIC with the Table at a new address would also apply here.

With regard to creating your own characters - refer to Section 2. Remember that you can define the Foreground and Background colour for each line of a pattern - the address of the appropriate byte in VRAM being 8192 more than the PATTERN GENERATOR byte.

### 6.3.4 Other Initialisations for Mode 2

The initialisation procedure for the VDP, given in Section 6.3.1 gives a full character set with colour control over the entire screen. You can, however, set up the VDP differently to produce different effects.

These subtle variations on the full initialisation mean that you can effectively set up the Mode 2 character to suit your particular needs, so saving VRAM and programming space, and time.

#### One Character Set: Three Colour Sets

If you change Line 50 (program in Section 6.3.1) so that VDP(4)=0 instead of 3, then you will have but one character set for the entire screen, defined by the first 2048 bytes of the PATTERN GENERATOR Table. The cursor will be a true cursor wherever it appears on the screen, and you will be able to print anywhere on the screen immediately.

The COLOUR Table, however, will still be fully operative: the first 2048 bytes of the Table will define the colours for the characters that appear at the top of the screen, the second 2048 bytes will define the colours for the characters that appear in the middle of the screen, and the last 2048 bytes will define the colours of the characters at the bottom of the screen.

Thus, if 256 characters are sufficient, this would be the simplest way of initialising the Mode - yet still allowing you to have the same character in three different colourings, according to where it appears on the screen. You will also 'release' 4096 bytes of VRAM for other uses - such as 'Table switching'.

#### Two Character Sets: Three Colour Sets

If you make VDP(4)=1, then the character set in the first 2048 bytes of the PATTERN GENERATOR Table appears at the top AND bottom sections of the screen: the middle section is defined by the second 2048 bytes of the PATTERN GENERATOR Table.

The first, second and third blocks of 2048 bytes in the COLOUR Table still point to the top, middle and bottom of the screen respectively, so characters can be individually coloured in each area, as before.

Similarly, if VDP(4)=2, then the first 2048 bytes of the PATTERN GENERATOR Table will be used for the characters at the top and middle of the screen. The bottom of the screen uses characters defined by bytes 4096 to 6143 of the PATTERN GENERATOR Table. Again, the COLOUR Table still allows characters in each section of the screen to be individually coloured.

### One or Two Colour Sets

Just as the PATTERN GENERATOR Table can be reduced in size by giving VDP(4) different values, so the COLOUR Table can be reduced. The values for VDP(3) are as follows:

(a) VDP(3)=&H9F. This gives just one 'colour set' for the entire screen, the colours being determined by the first 2048 bytes of the COLOUR Table.

(b) VDP(3)=&HBF. This provides two colour sets for the entire screen. The first 2048 bytes of the COLOUR Table control the colours of characters in the top and bottom sections of the screen. The second block of 2048 bytes in the colour table control the middle section of the screen.

(c) VDP(3)=&HDF. This provides two colour sets for the entire screen. The first 2048 bytes of the COLOUR Table control the colours of characters in the top and middle sections of the screen. The last (third) block of 2048 bytes in the COLOUR Table control the colours of characters in the bottom section of the screen.

A word of warning: Bits 0 to 4 of VDP(3) must ALWAYS be set, otherwise only part of the character set will be recognised, and Bit 7 must always be set for the operation of Mode 2 in this way. If it is not set, the COLOUR Table will have the same Base as the PATTERN GENERATOR Table ... with weird results!

### Mix 'n Match

That's right - you can also mix the way you initialise the Table lengths for Mode 2. Thus, by making VDP(4)=2, and VDP(3)=&HDF, you will have one character set and one colour set (the first 2048 bytes of both Tables) controlling the top and middle sections of the screen, and one character set and one colour set (the last 2048 bytes of both Tables) controlling the bottom section of the screen. This flexibility gives you ample scope to create a screen display economically - and save useful VRAM space for other purposes.

SECTION 7

# SCREEN MODE 3

## Multicolour

This Mode provides multicoloured block graphics on a screen that is 64 blocks wide by 48 blocks deep. The MSX initialises the Screen like Screen 2 - that is, to accept graphics commands, with limited facilities for printing to the screen. All shapes are formed using the 4x4 pixel colour blocks, which can be in any of the 15 colours available. It is not possible to print characters direct from the keyboard onto the screen, and consequently when a program using this Mode has ended or is aborted, the MSX returns to Screen 1 or Screen 0 to accept BASIC commands.

## 7.1 SCREEN MODE SPECIFICATION

### 7.1.1 Screen Parameters

| | |
|---|---|
| Screen Size (Max): | 64 columns x 48 rows |
| Character set: | None inherent |
| Colour Block size: | 4 pixels x 4 pixels |
| Colours available: | All |
| Sprites: | Available |

### 7.1.2 How the VDP Operates in Mode 3

The VDP is set to Mode 3 when Bit 1 of VDP Register 0 and Bit 4 of VDP Register 1 are reset to '0', and Bit 3 of VDP Register 1 is set to '1'. The screen is then treated as an unrestricted 64 x 48 block display, each block being 4 x 4 pixels - the maximum definition that can be achieved on the screen. A group of four blocks (8x8 pixels) make up one 'character position', and are represented by one byte in the NAME Table.

Thus, in this Mode the NAME Table is the same as that for the Screen Modes 1 and 2 - that is, 768 bytes long, and 'mapping' to the screen as shown in Figure 3. However, in this Mode, the NAME Table doesn't 'look' to the PATTERN GENERATOR Table for a character pattern, but for a set of four 4x4 pixel colour blocks. Thus data in the PATTERN GENERATOR produces colours on the screen, not patterns. A COLOUR Table isn't used.

Each character position on the screen is made up of four 4 x 4 pixel blocks, and the colour for each pair of horizontal blocks is derived from one byte in the eight that go to make up a pattern definition. The two bytes providing the colours for a complete character block area are consecutive in the PATTERN GENERATOR Table: which two of the eight bytes of a pattern set provide the colours for a character block on the screen depends on the location of the character block on the screen.

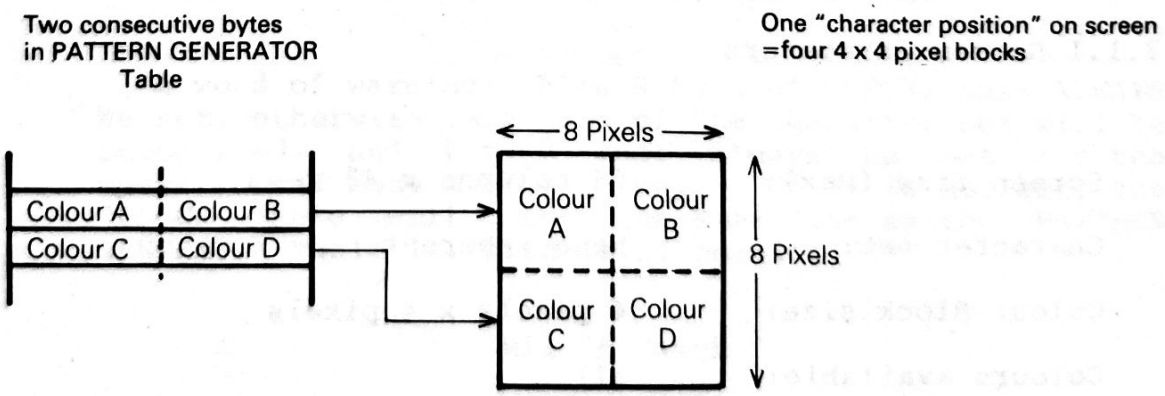



**Figure 14. Creating the Multicolour Character**

Figure 14 shows how the two bytes provide the colours for a character position on the screen. Exactly which two bytes of the PATTERN GENERATOR 'character set' are used depends on the location on the screen that the colour block is to appear - which is directly related to a location in the NAME Table.

For the very top Row - Row 0, the first two bytes of the 'character pattern' are used. For the second Row - Row 1 - the second pair of pattern bytes are used. For the third Row - Row 2 - the third pair of bytes are used, and for the fourth Row, the fourth pair of bytes are used. For Row 5 on the screen, the first pair of bytes are again the ones that determine the colours - and so on for each group of four rows right down the screen. Thus, the second pair of bytes in the 'character pattern' determine the colours when the character is called in Rows 1, 5, 9, 13, 17 or 21. This mapping to the screen is shown in Figure 15.

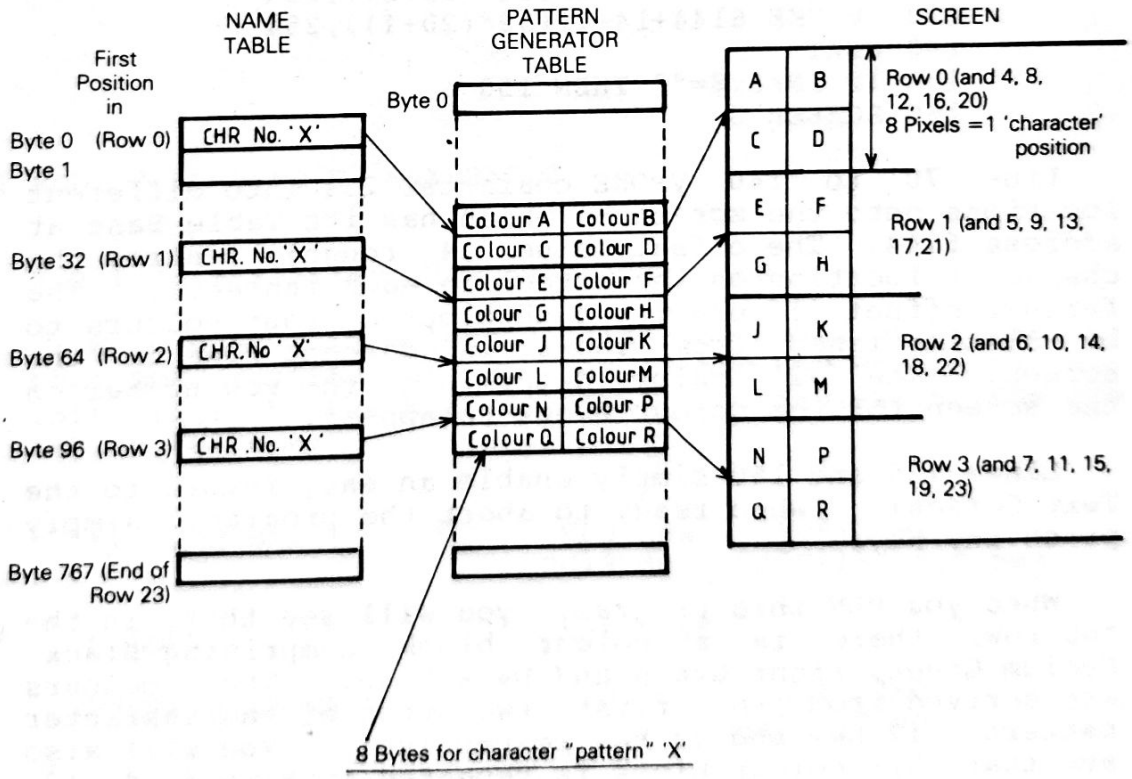



**Figure 15. Mapping to the Screen, Mode 3**

Probably the best way to see this is by an illustrative program such as the one that follows. Since the MSX initialises Mode 3 in a special way, it is necessary to set the VDP for Mode 3 by using the VDP Registers: this is done in Lines 10 and 20. We then load into character position 254 of the PATTERN GENERATOR Table (with its Base at address 0 in VRAM - as set when Screen 1 is called by Line 10) the colour set as consecutive values, for easy identification.

```
10  SCREEN 1:KEYOFF
20  VDP(1)=VDP(1) OR 8:CLS
30  FOR I=0 TO 7
40   READ D$:D=VAL("&H"+D$)
50   VPOKE(254*8)+I,D
60  NEXT
65  DATA 12,34,56,78,9A,BC,DE,F1
70  FOR I=0 TO 3
80   VPOKE 6144+14+I+(32*I),254
90   VPOKE 6144+14+I+(32*(4+I)),254
100  VPOKE 6144+14+I+(32*(8+I)),254
110  VPOKE 6144+14+I+(32*(12+I)),254
120  VPOKE 6144+14+I+(32*(16+I)),254
130  VPOKE 6144+14+I+(32*(20+I)),254
140 NEXT
150 IF INKEY$="" THEN 150
160 SCREEN 0
```

Line 70 to 140 VPOKE character 254 into different locations onto the screen - which has its Table Base at address 6144. The offset value '14' roughly centres the character location on the screen in each instance. The further offset 'I' enables each block of four colours to be distinguished from those in adjacent rows on the screen. The '32*' value determines the Row number on the screen for the colour block to appear.

Lines 150 and 160 simply enable an easy return to the Text Screen: when ready to abort the program, simply press any key.

When you RUN this program, you will see that, in the top row, there is a colour block comprising Black, Medium Green, Light Green and Dark Blue. These colours are derived from the first two bytes of the character pattern - 12 hex and 34 hex respectively. You will also see that this colour block is repeated in Rows 4, 8, 12, 16 and 20, showing that they too are coloured by the first two bytes of the character pattern.

The colours of the block in the second row (Row 1) are Light Blue, Dark Red, Cyan and Medium Red - derived from the second pair of bytes in the character pattern (56 hex and 78 hex respectively). These too are repeated at every fourth row. Similarly, you will be able to see the derivation of the colours in the colour blocks in Row 2 and Row 3 - and every fourth Row on.

Used in this way, the NAME Table is 768 bytes long and the PATTERN GENERATOR Table is 2048 bytes long - making a total of 2816 bytes. So there can be plenty of VRAM space available if required.

However, there is another way to use the Mode - with greater economy of VRAM space, and requiring less 'pattern definition'. Since each screen character location needs only two bytes to define all four colours in its colour block, and since there are 768 screen locations, then every location on the screen could be individually and separately coloured from 1536 bytes (768x2).

One would thus have a total of 192 character pattern sets (1536/8). Each character pattern set, remember, provides individual colouring for four different character pattern blocks on the screen, depending on their location: any more character patterns in the PATTERN GENERATOR are therefore superfluous.

The MSX uses this principle in its initialisation of Mode 3.

## 7.2 MODE 3 AS USED BY MSX BASIC

### 7.2.1 Initialising VRAM Base Addresses

When Mode 3 is selected using MSX BASIC or by calling the ROM routine at 75 hex, the VDP Registers are initialised by the MSX so that the Tables in VRAM are set as follows:

| | | |
|---|---|---|
| PATTERN NAME: | 800 hex<br>2048 dec | VDP(2)=2<br>BASE(15)=800 hex |
| COLOUR: | Not Used | |
| PATTERN GENERATOR: | 0 | VDP(4)=0<br>BASE(17)=0 |
| SPRITE ATTRIBUTE: | 1B00 hex<br>6912 dec | VDP(5)=36 hex<br>BASE(18)=1B00 hex |
| SPRITE PATTERN: | 3800 hex<br>14336 dec | VDP(6)=7<br>BASE(19)=3800 hex |

Note that the Sprite Table addresses are the same as for Screen Modes 1 and 2.

### 7.2.2 Loading the Tables

As indicated in Section 7.1.3, the MSX makes use of the fact that not every character number is required to provide colouring for every possible block on the multicolour screen. It does this by loading the NAME Table in a similar way to the way it loads Mode 2 - that is, with 'pointers' to the PATTERN GENERATOR Table.

The very first, leftmost character position in each of the first four Rows all derive their colours from the same character pattern set - albeit, from different pairs of bytes. Similarly with the second character position - and so on to the end of the Row.

The 32 bytes for each of the first four Rows of the NAME Table are therefore initialised to point to the first 32 character patterns in the PATTERN GENERATOR Table. That is, the first 32 bytes of the NAME Table are loaded with the values 0 to 31 (1F hex) respectively. Similarly, the second, third and fourth sets of 32 bytes in the NAME Table are loaded with the values 0 to 31 respectively.

The 32 bytes for each of the next four Rows - Rows 4 to 7 are loaded with the values 32 to 63 (20 hex to 3F hex) respectively, and so on throughout the NAME Table:

| Screen Row | Character Numbers Pointed To | Relevant Bytes of Chr Pattern |
|---|---|---|
| 0 | 0- 31 ( 0-1F hex) | 0 and 1 |
| 1 | 0- 31 ( 0-1F hex) | 2 and 3 |
| 2 | 0- 31 ( 0-1F hex) | 4 and 5 |
| 3 | 0- 31 ( 0-1F hex) | 6 and 7 |
| 4 | 32- 63 (20-3F hex) | 0 and 1 |
| 5 | 32- 63 (20-3F hex) | 2 and 3 |
| 6 | 32- 63 (20-3F hex) | 4 and 5 |
| 7 | 32- 63 (20-3F hex) | 6 and 7 |
| 8 | 64- 95 (40-5F hex) | 0 and 1 |
| 9 | 64- 95 (40-5F hex) | 2 and 3 |
| 10 | 64- 95 (40-5F hex) | 4 and 5 |
| 11 | 64- 95 (40-5F hex) | 6 and 7 |
| 12 | 96-127 (60-7F hex) | 0 and 1 |
| 13 | 96-127 (60-7F hex) | 2 and 3 |
| 14 | 96-127 (60-7F hex) | 4 and 5 |
| 15 | 96-127 (60-7F hex) | 6 and 7 |
| 16 | 128-159 (80-9F hex) | 0 and 1 |
| 17 | 128-159 (80-9F hex) | 2 and 3 |
| 18 | 128-159 (80-9F hex) | 4 and 5 |
| 19 | 128-159 (80-9F hex) | 6 and 7 |
| 20 | 160-191 (A0-BF hex) | 0 and 1 |
| 21 | 160-191 (A0-BF hex) | 2 and 3 |
| 22 | 160-191 (A0-BF hex) | 4 and 5 |
| 23 | 160-191 (A0-BF hex) | 6 and 7 |

Thus, the colour for every individual 4x4 pixel colour block can be specified by suitably loading the top or bottom nibble of the appropriate byte in the character pattern relating to the block's position on the screen.

An example will undoubtedly help to illustrate this. Suppose it is desired to colour Black the 4x4 pixel block that is fifth along from the left and second down from the top. This particular 4x4 block lies in the third 'character' position on the screen. To be precise, it is in the bottom right hand corner of the third character position.

The third byte in the NAME Table points to character pattern number 2, and since we are in the top 'character' row on the screen, it is the first two bytes of the character pattern set that control the four colours of the character block. We want the bottom left segment of the character block - which is specifically controlled by the bottom nibble of the second byte (refer to Figure 14).

Thus, the byte we need to change is the second of Character Pattern number 2 in the PATTERN GENERATOR Table - and we need to change only its bottom nibble. The start of the PATTERN GENERATOR Table is at VRAM address 0, so the second character will start at address 16 (0+(2x8)). The second byte of this pattern will therefore be at VRAM address 17.

The question now arises - how do you change the bottom nibble without upsetting the top nibble. Probably the easiest way is to use a simple 'masking' technique. Thus, if we get the value of this byte and AND it with &HF0, we will retain whatever is in the top nibble - without having to know its actual value. If, then, it is ORed with '1' (the code for the colour Black) and replaced (VPOKEd) back, the job will have been done:

```
10 SCREEN 3
20 VPOKE 17,VPEEK(17) AND &HF0 OR 1
30 IF INKEY$="" THEN 30
40 SCREEN 0
```

If one wanted to colour Black the fourth 4x4 pixel block along, second down, then the end of Line 20 would read 'VPEEK(17) AND &HF OR &H10' - to load '1' into the top nibble without affecting the bottom nibble.

The appropriate nibble of the appropriate byte in the character pattern table for any 4x4 pixel location on the screen can be found by a similar process.

All of the 1536 bytes of the PATTERN GENERATOR Table (for the eight byte 'character patterns' 0 to 191) are intialised to 44 hex by the MSX - i.e., to make every 4x4 pixel block on the screen Dark Blue in colour.

### 7.2.3 How MSX BASIC uses Mode 3

The Multicolour Screen is essentially for graphics, and it is initialised by the MSX in a similar manner to Screen Mode 2 so that all the BASIC graphics statements can be used.

The operation of the screen is, therefore, very similar to Screen Mode 2 - but for this screen, of course, the definition is not in single pixels, but in 4x4 pixel blocks.

When using the graphics statements, such as 'LINE' and 'CIRCLE', the parameters are still defined by pixel locations, even though an entire colour block is switched on by the statement.

Putting text on the screen involves the same procedure as that for Screen 2 - using the 'OPEN "GRP:" AS #1' and 'PRINT #1,"string" ' statements. The characters displayed on the screen will be large, occupying 8x8 colour blocks rather than 8x8 pixels, and they will have the last defined 'Foreground' colour (COLOR statement).

Like Screen 2, only the MSX character set can be used this way: to put other 'characters' on the screen, one would have to set the colours of the appropriate nibbles in the PATTERN GENERATOR Table ... an awesome task! It would undoubtedly be far easier to use the powerful graphics statements to create 'pictures' on the screen.

## 7.3 MODE USAGE

### 7.3.1 Text on Screen

The nature of the Multicolour display - with its large, easily read characters - lends itself to educational programs for young children, and to Titling for programs.

As with Screen 2, characters printed to the screen (using the PRINT 1 statement) can be individually coloured by preceding the 'PRINT' with a 'COLOR' statement that defines the Foreground colour only. To locate characters on the screen, the 'PRESET' statement can be used: remember that the graphic statements require locations to be defined as pixel positions, not colour block positions.

### 7.3.2 Free VRAM Areas

An extensive area of VRAM is free for Table Switching - from 2816 to 6911 decimal (0B00-1AFF hex), and from 7040 to 14335 decimal (1B80-37FF hex). This would enable separate PATTERN GENERATOR Tables to be set up, for rapid switching of displays: a discussion on Table Switching is given in Sections 4.3.3 and 4.3.4.

### 7.3.3 Sprite Patterns

Since the MSX doesn´t load a character pattern set into VRAM in this Mode, it is not possible to use the ´VDP(6)=VDP(4)´ technique to define all the Sprite Patterns as character patterns. Furthermore, the VRAM areas used for the PATTERN GENERATOR in Modes 1 and 2 are overwritten on initialisation of Mode 3. In order to create a Sprite Pattern set that is the same as the character set, it would be necessary to juggle around with the Table Base addresses during the initialisation process. The following program gives an example of how this could be achieved:

```
10 BASE(7)=&H2800
20 SCREEN1:SCREEN3
30 VDP(6)=5
35 REM
36 REM Sprite example
37 REM
40 PUTSPRITE 0,(100,100),15,65
50 GOTO 50
```

Line 10 tells the computer that you want BASE(7) - which is the Base address for the PATTERN GENERATOR Table in Screen Mode 1 - to start at VRAM address 2800 hex (10240 decimal), and not its normal address for that Mode, which is ´0´. Screen Mode 1 is then initialised (Line 20) so that the MSX character set will be loaded into the PATTERN GENERATOR Table at the new address, and then Mode 3 is initialised.

The Character set will still be in VRAM, and so all that is now needed is to reset the SPRITE PATTERN Table so that its Base is at 2800 hex. This is achieved by suitably loading the VDP Register controlling the SPRITE PATTERN Table Base address - VDP(6) - as in Line 30. Line 40 demonstrates that the system works (!) by placing Sprite Pattern 65 (the letter ´A´), in White on the screen.

This may seem like a good way to get small characters on the sceen for text purposes: as long as there are not more than four sprites to a line ... it is. One has to admit that four Srites to a line would be very limiting as far as text is concerned!

## Author´s Note

It is hoped that this book and its Appendices has provided you with a better insight into the operation of the Screen displays on your MSX Home Computer, and that it continues to be an invaluable source of reference when programming.

Every effort has been made to ensure the accuracy of its contents, and to present the information in a way that can be readily understood by all. Suggestions for additional information or improvements should be sent to the Publishers.

# APPENDIX A

## BINARY-HEX-DECIMAL CONVERSIONS

Newcomers to computing often find the concept of binary and hexadecimal numbering systems difficult to understand. For computer programming both these systems are more meaningful than decimal and, when grasped, can make the process of programming much easier and quicker. Indeed, for machine code programming, knowledge of binary and hexadecimal is essential. This short Appendix gives a broad overview of the systems, with details of how to convert from one to another.

### WHY BINARY?

Binary simply means 'two', and a binary counting system is one which has only two values - '0' and '1'. Decimal, by contrast, has ten values - 0 to 9. Binary is useful because it is easier to have devices with only two states - on or off, for example - than with ten states.

When counting in decimal, we start from zero and go up to 9. Then, with no more values to use, we add one to the column to the left to represent 'another ten', reset the 'units' to zero and start again. Binary works the same way. Only this time, we have only 0 and 1 to play with. The first few binary numbers look like this:

| | | |
|---|---|---|
| 0 0 0 | = | 0 in decimal |
| 0 0 1 | = | 1 |
| 0 1 0 | = | 2 |
| 0 1 1 | = | 3 |
| 1 0 0 | = | 4 |
| 1 0 1 | = | 5 |
| 1 1 0 | = | 6 |
| 1 1 1 | = | 7 |

If you were to write down a four digit decimal number - say 1234, you would know that this represented:

```
4 'units'
3 'tens'       = 3x10
2 'hundreds'   = 2x(10x10)     = 2x(10 to the power 2)
1 'thousands'  = 1x(10x10x10) = 1x(10 to the power 3)
```

The first two values - for the units and tens - can be written as

```
4x(10 to the power 0)
3x(10 to the power 1)
```

because any number to the 'power 0' is equal to 1, and any number to the 'power 1' is equal to itself.

In computers we deal with binary numbers eight digits at a time. Each digit is called a 'bit' - short for binary digit, and all eight together are called a 'byte'. If we call the least significant 'bit' Bit 0 and the most significant bit Bit 7, then a binary number such as '10011101' can be represented like this:

```
Bit Number    7 6 5 4 3 2 1 0

Value         1 0 0 1 1 1 0 1
```

Let us now 'translate' this binary number into decimal. In the decimal system each digit to the left in a number represents an increasing power of ten - see the example given above. The same is true in binary - only instead of being a power of ten, each position represents an increasing power of 2. So to convert a binary number to decimal, wherever a '1' occurs in the number we 'raise' two to the power of the corresponding 'bit' number. Thus, using the above example, we have:

```
1 for bit 0. 2 to the power 0 =   1
0 for bit 1.  Do nothing
1 for bit 2. 2 to the power 2 =   4 (2x2)
1 for bit 3. 2 to the power 3 =   8 (2x2x2)
1 for bit 4. 2 to the power 4 =  16 (2x2x2x2)
0 for bit 5.  Do nothing
0 for bit 6.  Do nothing
1 for bit 7. 2 to the power 7 =128 (2x2x2x2x2x2x2)
```

When these are all added up, we get 157 - which is the decimal equivalent of 10011101 in binary. Don't forget that any number to the power 0 is equal to '1', strange though it may seem.

Another way of looking at this is to write down the decimal value represented by each bit position:

| Bit Position | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| Decimal Value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |

So wherever a bit has a '1', simply add in to the total the decimal equivalent. You'll notice that, by adding up these decimal values in different combinations, you can get every number from 0 to 255 - the maximum that can be stored in eight binary digits.

How about converting decimal into binary? Probably the easiest way is to use the 'divide by two' method. For this, you take the decimal number, and successively divide it by two, writing down the REMAINDER in ascending order for the binary number. Thus, taking 157 for example:

```
157/2 = 78, remainder =                1
 78/2 = 39, remainder =               0
 39/2 = 19, remainder =              1
 19/2 =  9, remainder =             1
  9/2 =  4, remainder =            1
  4/2 =  2, remainder =           0
  2/2 =  1, remainder =          0
  1/2 =  0, remainder =         1

So the binary number is        10011101
```

## WHY HEXADECIMAL?

Hexadecimal is a counting system in which we have 16 values: we use the numbers 0 to 9, then for the values 'ten' to 'fifteen', we use the letters A to F respectively. If you look at the binary numbers again, you'll see that four binary bits can be used to count from 0 to 15 - and that's what makes hexadecimal so useful in computing.

One hexadecimal digit can be represented by four binary bits, so an eight bit binary number can be represented by two hexadecimal digits. The hexadecimal number enables us to tell at a glance what the value is for each 'group' of four bits - once you have mastered the fact that A=10, B=11 and so on, of course.

Each group of four bits is called a 'nibble' (don't ask!). To convert from binary to hexadecimal is simply a matter of taking the bits four at a time. Using our example of 10011101 binary again:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Nibble 'bit' number | 3 | 2 | 1 | 0 | 3 | 2 | 1 | 0 |
| Decimal value of bit | 8 | 4 | 2 | 1 | 8 | 4 | 2 | 1 |
| Binary value | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |

Adding the decimal values for the '1' bits in the top nibble, we get 8+1 = 9. Similarly, adding the decimal values for the '1' bits in the bottom nibble, we get 8+4+1 = 13, which in hexadecimal is represented by the letter 'D'. So the hexadecimal equivalent is 9D hex.

Converting '9D hex' to decimal is also straightforward. Remember that there are 16 values in hexadecimal, so the least significant digit can have values from 0 to F (15 in decimal). For 16, we 'add one' to the next column to the left - so this column represents the number of '16s'. So, to convert '9D hex' to decimal, multiply the 9x16, then add in the value of 'D' - which is 13. The answer is 157.

If we have four hexadecimal digits instead of two, we can get up to a value of 65535 in decimal (FFFF hex). Here, the 'base' is 16, so each position to the left represents an increasing power of 16 - just as in decimal, each position to the left represents an increasing power of 10, and in binary, an increasing power of 2.

Thus AB34 hex is:

```
4x(16 to the power 0) = 4x1          =    4
3x(16 to the power 1) = 3x16         =   48
Bx(16 to the power 2) =11x16x16      = 2816
Ax(16 to the power 3) =10x16x16x16 =40960

Adding the results together          =43828 decimal.
```

One final point. When a two-byte number (four hex digits, 16 binary digits) is stored in machine code on 'Z80' machines such as the MSX, the LEAST significant byte is always placed before the most significant byte. Thus, if 'AB34' hex were to be stored in the machine in two adjacent addresses, the first address would have the value 34 hex, and the next address would have the value AB hex. The reason is tied up with the way the Z80 processor works ... and that is another book altogether.

# APPENDIX B

## DEMONSTRATION PROGRAMS

The programs in this Appendix illustrate specific operating features of the MSX, and are given here rather than within the text of the book for easy reference to routines which can be adapted for the reader's own programs. The programs are not written for speed or space-saving, but so that they can be easily understood.

The reader is strongly urged to 'experiment' with the programs, since this can be an invaluable way to build knowledge and understanding of the machine's operation.

Machine code programs are not included here: programmers with knowledge of machine code will be able to 'translate' these programs using the ROM routines and useful addresses given in Appendices E and F. Programmers wishing to enter the fascinating world of machine coding are recommended to read 'Starting Machine Code on the MSX' by G. Ridley, and Published by Kuma Computers Ltd.

## THE PROGRAMS

1. Setting up Sprite Patterns
2. Changing Sprite Magnification by VDP (1)
3. Switching NAME Table addresses
4. Switching PATTERN GENERATOR Table addresses
5. Sprite Patterns as Character Patterns
6. Fifth Sprite demonstration
7. MSX Character patterns in detail
8. Changing Character shape whilst it is displayed
9. All Sprite sizes in one program

## PROGRAM 1

## SETTING UP SPRITE PATTERNS

This is a 'Sprite pattern loader' program, serving two purposes. It demonstrates how both 8-byte and 32-byte patterns can be loaded into the SPRITE PATTERN Table so that both sizes can be used in the same program. It is also used as a Subroutine for other programs in this Appendix.

Lines 1000 to 1040 load the data for eight 8-byte Sprites and into the first eight Sprite patterns - 0 to 7 - of the Sprite Table. The first four of these patterns overlay each other to produce a car shape: to see it, set SCREEN 1,0, enter 'GOSUB 1000', then enter:

```
PUTSPRITE 0,(100,100),15,0
PUTSPRITE 1,(108,100),15,1
PUTSPRITE 2,(100,100),1,2
PUTSPRITE 3,(108,100),1,3
```

Notice the displacement in the 'x' location for Sprites 1 and 3 - the entire pattern is two Sprites wide, and each Sprite is 8 pixels when not magnified. If SCREEN 1,1 is used (BEFORE loading the Sprite Patterns) - or Bit 0 of VDP(1) is set - the Sprites will be 16 pixels wide, and the '108' parameters will need to be '116'.

The next four Sprite patterns are a Dog, a face, and two arbitrary geometric shapes.

Lines 1050 to 1100 load the data for 32-byte Sprite patterns into the PATTERN GENERATOR Table. For this size of Sprite, only 64 can be defined altogether, and the patterns in this program are located in the Table such that, when 32-byte Sprites have been specified (SCREEN 1,2 for example), they are Pattern Numbers 32, 33 and 34. This has been done to allow both sets to be called from within the same program - see PROGRAM 10 of this Appendix, and Section 3.1.4.

Each of the 8-Byte Sprite DATA lines represents one Sprite pattern, making it easier for you to see how the shape is created. The DATA for each of the 32-byte Sprite patterns is split into two 16-byte DATA lines - the first gives the pattern for the left side of the Sprite, and the second the pattern for the right side. See Section 3.1.3.

Save this Program for use in later Demonstration Programs. Note: if you use SAVE"CAS:", you will be able to use MERGE "CAS:" to merge the saved program with a program already written - provided that you don't use the same Line numbers in both programs.

```
998 ' *** LOAD EIGHT 8-BYTE SPRITES ***
999 '
1000 RESTORE 1200
1010 FOR I=0 TO 63
1020  READ D$
1030  VPOKE &H3800+I,VAL("&H"+D$)
1040 NEXT
1047 '
1048 ' *** LOAD THREE 32-BYTE SPRITES ***
1049 '
1050 RESTORE 1300
1060 FOR I=0 TO 95
1070  READ D$
1080  VPOKE &H3C00+I,VAL("&H"+D$)
1090 NEXT
1100 RETURN
1197 '
1198 ' *** 8-BYTE SPRITE DATA ***
1199 '
1200 DATA 1F,8,88,FF,FF,FF,0,0
1205 DATA F0,98,8C,FC,FC,F8,0,0
1210 DATA 0,7,7,0,0,0,70,20
1215 DATA 0,60,70,0,0,0,70,20
1220 DATA 0,1,C2,FC,3C,24,24,0
1225 DATA 38,7C,BA,EE,7C,44,38,10
1230 DATA 10,28,44,82,41,22,14,8
1235 DATA 77,63,55,8,55,63,77,0
1297 '
1298 ' *** 32-BYTE SPRITE DATA ***
1299 '
1300 DATA 0,0,1,1,1,1,1,3F,3F,1,1,1,1,1,0,0
1301 DATA 0,0,80,80,80,80,80,FE,FE,80,80,80,80,80,0,0
1302 '
1305 DATA 0,0,0,18,1C,0E,7,3,3,7,0E,1C,18,0,0,0
1306 DATA 0,0,0,18,38,70,E0,C0,C0,E0,70,38,18,0,0,0
1307 '
1310 DATA 7,F,1C,20,2C,40,45,20,21,14,13,8,4,B,11,29
1311 DATA C0,E0,30,30,70,30,70,B0,B0,20,A0,20,40,80
     C0,E0
```

## PROGRAM 2

## CHANGING SPRITE MAGNIFICATION BY VDP (1)

This Program demonstrates how VDP Register 1 can be used to change the magnification of a Sprite displayed on the screen. This enables Sprite magnification to be changed without using the BASIC 'SCREEN n,m' statement - which not only clears the screen of any text present - but, because a magnification or size value is specified ('m'), also clears the Sprite Patterns! If the Screen Mode is changed without specifying the size or magnification factor, then the Sprite Pattern Table is left untouched.

This Program uses Program 1 as a Subroutine to generate Sprite patterns: if you have SAVEd Program 1 (using SAVE"CAS:"), you will be able to merge it (using MERGE"CAS:") after this Program has been entered. If, on the other hand, you CSAVEd Program 1, then load it first before entering this Program.

The 'BEEP' in Line 20 is to tell you when the Sprite patterns have been loaded, and will give you an idea of how long it takes. Lines 30 and 40 place two of the 8-byte Sprites on the screen, using Sprite Planes 1 and 2. Line 50 provides a short delay between the changes in Sprite magnification.

Line 60 obtains the value of Bit 0 of VDP Register 1 - the 'magnification Bit' - by masking out the other Bits, and places the value in variable 'A'. Lines 70 and 80 change the magnification. If Bit 0 of VDP Register 1 is set to a '1', Line 70 resets it. Conversely, Line 80 sets it if it was reset. Once the change has been made - in either Line, a jump is made back to the delay in Line 50.

Lines 70 and 80 can be combined in just one Program Line, and Line 60 deleted altogether, by:

```
IF (VDP(1)AND1)=1 THEN VDP(1)=VDP(1)AND&HFE:ELSE
VDP(1)=VDP(1)OR1:GOTO 50
```

The brackets round the 'VDP(1) AND 1' at the beginning of this Line are important.

Notice the technique used to set and reset the VDP Register Bit. This is better than adding or subtracting '1' from the Register contents, since this procedure could upset other Bits in the Register if things became 'out of phase' - as may occur in a full length program.

After you have entered and RUN the Program and Subroutine, Line 20 can be turned into a 'REM'.

For a demonstration of how the SCREEN statement can clear the SPRITE PATTERN Table, after the program has been RUN, change Line 10 to read 'SCREEN 1,0' and Line 20 to be a REM: now, when RUN, there will be no Sprite display at all. To restore the situation, make Line 10 'SCREEN 1' again, remove the REM from Line 20, and RUN.

```
10 SCREEN 1
20 GOSUB 1000:BEEP
30 PUTSPRITE 1,(120,100),6,4
40 PUTSPRITE 2,(120,120),1,5
50 FOR I=1 TO 200:NEXT
60 A=VDP(1) AND 1
70 IF A=1 THEN VDP(1)=VDP(1) AND &HFE:GOTO 50
80 IF A=0 THEN VDP(1)=VDP(1) OR 1:GOTO 50

1000 ' Subroutine - Program 1 - goes here
```

See Program 10 for the way to change Sprite Size (VDP Register 1, Bit 1)

## PROGRAM 3

## SWITCHING NAME TABLE ADDRESSES

This program shows how the PATTERN NAME Table Base can be switched to a new address in VRAM, to present a completely different screen display very quickly.

WARNING: BASIC expects to find the NAME Table at the address it was initialised at on entering the Screen Mode - i.e., the address defined by BASE(n), where 'n' is 0, 5, 10 or 15 for Modes 0 to 3 respectively. Always be sure, therefore, that at the end or on abortion of your program, the 'initialised' screen is re-entered.

The 'second' NAME Table is located at 3C00 hex in this program - on a 'legitimate' boundary (see pages 19-21 and Appendix C). This address is actually within the SPRITE PATTERN Table area, but we are not using Sprites in this program, so thats o.k.

Line 20 prints a message to the initialised screen display. Lines 40 to 60 fill the second NAME Table with 'colons', while Line 70 sets the colours for the colon character to Black on Dark Red (see Section 5.3.3).

Lines 80 to 100 place the message in M$ (Line 30) into the NAME Table, starting at the offset value 322 - 10 rows down, 2 columns along. These routines all take a four or five seconds to complete.

Then comes the actual Table switching segment, beginning with an ON STOP statement. This is so that when the program is aborted, conditions are returned to 'normal' for the MSX - including putting the original colours back into COLOUR Table address 8199. The Subroutine at Line 300 simply holds each screen display for about four seconds before making the switch.

As you will see when the program is RUN, switching from display to display this way is virtually instantaneous, with none of the visible build up that occurs when the screen is filled using normal BASIC statements.

```
10  SCREEN 1
14  '
15  ' *** Set up the two displays ***
16  '
20  LOCATE 5,10:PRINT "INITIALISED SCREEN"
30  M$="The SECOND Display Screen"
40  FOR I=0 TO 767
50   VPOKE &H3C00+I,&H3A
60  NEXT
70  VPOKE 8199,&H16
80  FOR I=1 TO LEN(M$)
90   VPOKE &H3C00+322+I,ASC(MID$(M$,I,1))
100 NEXT
104 '
105 ' *** Switch NAME Tables ***
106 '
110 ON STOP GOSUB 200: STOP ON
120 VDP(2)=&HF
130 GOSUB 300
140 VDP(2)=6
150 GOSUB 300
160 GOTO 120
194 '
195 ' *** Restoring to Normal ***
196 '
200 VDP(2)=6:VPOKE 8199,&HF4:STOP
294 '
295 ' *** Delay Subroutine ***
296 '
300 FOR I=0 TO 2000:NEXT:RETURN
```

## PROGRAM 4

## SWITCHING PATTERN GENERATOR TABLE ADDRESSES

In this program the PATTERN GENERATOR Table Base is switched to a new address in VRAM, by suitable adjustment to the value of VDP Register 4. This could be used to change the patterns displayed on the screen to alternative patterns, the contents of the NAME Table being unchanged.

For this demonstration, only a part of the 'second' PATTERN GENERATOR Table is loaded with patterns. The second Table is located at 3800 hex - normally the start of the SPRITE PATTERN Table. But as we are not going to use Sprites in this program, this is o.k.

The procedure used is as follows. The lower case character set is loaded into the second PATTERN GENERATOR Table, at locations that would normally be occupied by the upper case character set. Then the NAME Table is filled with the message 'UPPER TO LOWER CASE' (in capitals) - by simply using the 'PRINT' statement.

When the PATTERN GENERATOR Table Base address is switched, the NAME Table points to the new patterns for the characters - which have been defined as lower case letters. Consequently the displayed sentence switches from Capital letters to small letters.

Line 10 initialises the screen to Mode 1. Line 20 ensures that when the program is aborted, the NAME Table is pointing to the full PATTERN GENERATOR Table.

Lines 30 to 50 VPOKE into the new PATTERN GENERATOR Table, starting at the location for capital A (520=65x8), the lower case patterns for the alphabet. The MSX character set is in ROM, starting at address 1BBF hex. The pattern for the lower case letter 'a' - character 97 - therefore starts at 1BBF hex plus 97x8: each character pattern needs eight bytes, remember.

The FOR-NEXT loop (Line 30) is made long enough to just cover the 26 letters of the alphabet. The '-1' at the end of Line 30 is to ensure the exact number of 'loads' are made: there would be 209 operations between 0 and 208, the first being made when I=0.

After you have RUN the program, see the effect of putting a 'STOP' at the end of line 80. This will leave you with the NAME Table pointing to the second character set - in which only the CAPITAL letter characters have been defined. You will therefore be able to see what you're printing on the screen only when CAPITAL letters are used - and they will appear as lower case letters.

With a full character set in this second PATTERN GENERATOR Table, the character for every key will print to the screen. To return to the original PATTERN GENERATOR Table, carefully type in VDP(4)=0 and 'carriage return'.

The technique outlined in this program enables you to have more than one character set in the text Screen Modes, but of course, characters from one set only can be displayed at any given time.

```
10  SCREEN 1
20  ON STOP GOSUB 200:STOP ON
24  '
25  ' *** LOAD CHARACTERS a-z IN SECOND TABLE ***
26  '
30  FOR I=0 TO (26*8)-1
40   VPOKE &H3800+(65*8)+I,PEEK(&H1BBF+(97*8)+I)
50  NEXT
54  '
55  ' *** PRINT MESSAGE IN CAPITALS ***
56  '
60  LOCATE 5,10:PRINT "UPPER TO LOWER CASE"
64  '
65  ' *** TABLE-SWITCHING LOOP ***
66  '
70  GOSUB 300
80  VDP(4)=7
90  GOSUB 300
100 VDP(4)=0
110 GOTO 70
194 '
195 ' *** RESTORE ON PROGRAM ABORT ***
196 '
200 VDP(4)=0:STOP
294 '
295 ' *** DELAY SUBROUTINE ***
296 '
300 FOR I=1 TO 300:NEXT:RETURN
```

**For further information on VDP(4), refer to pages 22 to 24.**

## PROGRAM 5
## SPRITE PATTERNS AS CHARACTER PATTERNS

This program - in two stages - shows first how Sprites can be defined as MSX Character patterns, and secondly the formation of a 32-byte sprite.

First enter and RUN Part A. In this part, Line 20 sets the Base address for the SPRITE PATTERN Table to the same Base address used for the character PATTERN GENERATOR Table, by suitable adjustment of the value contained in VDP Register 6 (see Section 1.2.2 for details on the VDP Registers).

To make things slightly more interesting (!), you will be asked to 'Press any key' (Line 30), and in Line 60, the ASCII value for the key you pressed is stored in variable 'A'. Line 40 clears the input buffer, so that only the next key pressed will be accepted (try the program with this line temporarily as a REM, pressing as many keys as you can while the program is actually running). Note: you can of course use the 'greater-than' and 'less-than' symbols instead of 'NOT' and the '=' sign in this Line.

Line 80 puts the selected character on the screen as a Sprite - with a little movement as provided by the FOR-NEXT loop, Lines 70 and 90.

This program will enable you to see the 'characters' produced by keys such as those used for Cursor control. You can see the Sprite characters magnified by making Line 10 read SCREEN 1,1

Having RUN the program a few times, break out (by pressing 'CTRL-STOP'), and amend lines 10 and 80 as shown in Part B.

The new Line 10 sets the Sprites to 32-byte characters, magnified. Since each Sprite pattern now takes 32 bytes, there can be only 64 Sprite patterns in the 2048 byte SPRITE PATTERN Table. The amendment to Line 80 ensures that only Sprite patterns up to 64 are called.

To see how the 32 bytes form a large size Sprite, press the '0' key. You will see a Sprite Pattern comprising the numbers 0, 1, 2 and 3. That's because these patterns are consecutive in the PATTERN Table. Notice that the first two 8-byte segments, 0 and 1, form the left side of the Sprite, and the next two 8-byte segments form the right side of the Sprite.

This demonstrates quite clearly how the larger sized Sprite is formed from the consecutive block of 32 bytes in the PATTERN Table.

Part A

```
10  SCREEN 1,0
20  VDP(6)=VDP(4)
30  LOCATE 8,5:PRINT "PRESS ANY KEY"
40  IF NOT INKEY$="" THEN 40
50  A$=INKEY$:IF A$="" THEN 50
60  A=ASC(A$)
70  FORI=-8 TO 150
80   PUTSPRITE 1,(120,I),1,A
90  NEXT
100 GOTO 40
```

Part B

```
10  SCREEN 1,3
80  PUTSPRITE 1,(120,I),1,A/4
```

## PROGRAM 6

## FIFTH SPRITE DEMONSTRATION

This Program demonstrates what happens when five (or more) sprites are present on one horizontal pixel line. It will give you an opportunity to see how changing the planes on which Sprites are displayed can affect which Sprite 'vanishes' when four or more are on one pixel line.

Line 10 sets the MSX into Screen Mode 1, with magnified 8-byte Sprites: this will enable you to see what is happening quite clearly. Line 20 sets up VDP Register 6 so that the SPRITE PATTERN Table is at the same address as the character PATTERN GENERATOR Table: in other words, all the Sprite Patterns are set to character patterns.

Lines 30 to 60 put Sprite patterns 49 to 52 - the numbers 1 to 4 - on the screen in different colours, and slightly overlapping so that between them they occupy a small band of horizontal pixels. Notice that Sprite planes 2, 4, 6 and 8 are used: this is so that, later on, you can 'pass' a Sprite on an intermediate plane.

Lines 70 to 140 provide a loop to move a fifth Sprite (the number '5') down the screen, past the other four. At the top of the screen is printed the status of the 5th Sprite Flag (Bit 6 of VDP Register 8), and the 'Fifth Sprite Number'. The delay loop in line 130 slows the movement down, so that you can watch what is happening.

When RUN, the Fifth Sprite Flag will initially be zero, and will turn to a '1' as the moving Sprite coincides with the other four on the same horizontal pixel line. Part of the fifth Sprite will then 'vanish', and the Number of the Fifth Sprite plane will be indicated: any value indicated while the Flag is at '0' is spurious, and should be ignored.

After running the program a few times, change Line 80 to PUTSPRITE '1', instead of '9': you'll notice that now, the Fifth Sprite plane is different - and it is part of Sprite pattern '4', on plane 8, that 'vanishes' as Sprite pattern '5' passes by it. The four fully displayed Sprites are always those closest to the 'front' of the screen, i.e. those with the lowest plane numbers.

Try placing the Sprites on other planes and observe the effects: it will give a good insight into how the Fifth Sprite mechanism works.

```
10  SCREEN 1,1
20  VDP(6)=VDP(4)
30  PUTSPRITE 2,(100,100),15,49
40  PUTSPRITE 4,(108,102),7,50
50  PUTSPRITE 6,(116,104),10,51
60  PUTSPRITE 8,(124,106),1,52
70  FOR I=60 TO 130
80   PUTSPRITE 9,(132,I),9,53
90   LOCATE 5,5
100  PRINT "5th SPRITE FLAG  = ";(VDP(8) AND &H40)/&H40
110  LOCATE 5,6
120  PRINT "5th SPRITE Plane = ";VDP(8) AND &H1F
130  FOR J=1 TO 50:NEXT
140 NEXT
```

## PROGRAM 7

## PRINTING OUT THE CHARACTER PATTERNS

This program enables you to look at how any character pattern is constructed on the screen - and can help you when defining your own characters Line 20 sets variable BA to the Screen 1 PATTERN GENERATOR Table Base address. Lines 22 to 28 generate a new pattern for characters 250 and 219 - to give a more attractive display: they can be omitted, if you wish, but if you do, you should change the VPOKE in Line 40 to &H66 instead of &HF6.

Lines 30 and 40 set the colours for characters 250 and 219, and load the characters into variables D1$ and D2$ respectively for later on.

Lines 50 to 100 get the character number you wish to see displayed: Line 50 clears the input buffer, and Line 100 checks that a valid number has been INPUTed.

In Line 110, BL is made equal to the first byte of the pattern definition. Then comes a FOR-NEXT loop which examines each pattern line in turn. The print destination is set up (Line 130), and the pattern byte is VPEEKed (Line 140).

The byte value is converted to a BINary string and to a HEX string (Line 150), and the strings made the 'correct' length (Lines 160 and 170). A FOR-NEXT loop then prints out the binary strings in position - character 250 for a '0' bit, and 219 for a '1' bit (Line 190): watch the semi-colons in this Line.

Then the Hex value is printed out, so you can see how the byte value relates to the pattern line. Lines 230 simply enable you to continue inspecting character shapes: simply press 'Y' to view another, then enter the new character number.

One interesting character to view will be 255 - the cursor character. To do this, enter '255', then Backspace the cursor over one of the characters in the same line before making the Carriage Return: you will see how the cursor character always 'images' the character it is placed over.

```
4   '
5   ' *** SET UP SCREEN ***
6   '
10  SCREEN 1
20  BA=BASE(7)
22  FOR I=0 TO 7
24  IF I=0 THEN VPOKE250*8,&HFF:VPOKE219*8,&HFF:GOTO 28
26   VPOKE (250*8)+I,&H80:VPOKE (219*8)+I,&H80
28  NEXT I
30  VPOKE 8192+(250/8),&HF1:D1$=CHR$(250)
40  VPOKE 8192+(219/8),&HF6:D2$=CHR$(219)
44  '
45  ' *** GET REQUIRED CHARACTER ***
46  '
50  IF NOT INKEY$="" THEN 50
60  LOCATE 3,1:PRINT "WHICH CHARACTER NUMBER?"
70  LOCATE 10,3:PRINT "    "
80  LOCATE 10,3:INPUT Q$
90  Q=VAL(Q$)
100 IF Q<0 OR Q>255 THEN GOTO 70
104 '
105 ' *** PRINT SHAPE & DATA ***
106 '
110 BL=BA+(8*Q)
120 FOR I=0 TO 7
130  LOCATE 5,6+I
140  CP=VPEEK(BL+I)
150  CP$=BIN$(CP):HV$=HEX$(CP)
160  IF LEN(HV$)=1 THEN HV$="0"+HV$
170  CD$=LEFT$("00000000",8-LEN(CP$))+CP$
180  FOR J=1 TO 8:BV$=MID$(CD$,J,1)
190   IF BV$="0" THEN PRINT D1$;:ELSE PRINT D2$;
200  NEXT J
210  PRINT " = ";HV$;" HEX"
220 NEXT I
224 '
225 ' *** MORE? ***
226 '
230 LOCATE 0,17:PRINT "PRESS 'Y' FOR ANOTHER"
240 PRINT "PRESS 'N' TO STOP"
250 IF NOT INKEY$="" THEN 250
260 Q$=INKEY$
270 IF Q$="Y" OR Q$="y" THEN 50
280 IF Q$="N" OR Q$="n" THEN END
290 GOTO 260
```

## PROGRAM 8

## CHANGING CHARACTER SHAPE WHILST IT IS DISPLAYED

This program demonstrates how a character shape can be changed whilst it is on display. The program creates a pair of 'eyes', which blink - until you've had enough and CTRL-STOP it from running.

Lines 10 to 30 set up the display, and locate (twice) on the screen the character that will be changed to an 'eye' shape. The DATA is RESTOREd to its start, then the open-eye character created via the subroutine.

Line 60 provides a short delay between the 'blinks'. The pattern for the character is then changed by a series of calls to the subroutine, through a FOR-NEXT loop. The Subroutine simply reads the DATA and VPOKEs it into the PATTERN GENERATOR Table at the correct locations for the character being changed.

While this technique may seem space consuming in terms of the data bytes required for each of the character pattern changes, it does save on the number of different character numbers required. Also, if used in Screen Mode 2 as set for a Text Screen (see Section 6.3), only one set of COLOUR Table bytes need be defined. Using eight different characters in Screen 2 as a Text screen would require 64 (8x8) COLOUR Table bytes to be defined.

Programming space can be saved also, by using the 'string' technique for the DATA statements, as outlined in Section 3.2.1 (example (b) ) for Sprite definitions. The Subroutine would need to be changed, of course.

Assuming that a DATA line had the form "abcdefgh", the Subroutine could be:

```
200 READ D$
210 FOR I=1 TO 8
220 P=ASC(MID$(D$,I,1))
230 VPOKE (254*8)+I-1,P
240 NEXT:RETURN
```

Note the '-1' in the VPOKE address, Line 230: this is to ensure the values are VPOKEd into their correct addresses. See Appendix D for the way to access characters from the keyboard for string statements.

```
4   '
5   ' *-* SET UP THE SCREEN ***
6   '
10  SCREEN 1
20  LOCATE 11,12
30  PRINT CHR$(254);" ";CHR$(254)
34  '
35  ' *** OPEN EYE SHAPE ***
36  '
40  RESTORE 500
50  GOSUB 200
60  FOR I=1 TO 200:NEXT
64  '
65  ' *** CREATE 'BLINK' ***
66  '
70  FOR J=1 TO 7
80   GOSUB 200
90  NEXT
100 GOTO 40
194 '
195 ' *** SUBROUTINE TO CHANGE CHARACTER ***
196 '
200 FOR I=0 TO 7
210  READ D$
220  VPOKE (254*8)+I,VAL("&H"+D$)
230 NEXT:RETURN
494 '
495 ' *** DATA ***
496 '
500 DATA 18,66,99,99,66,18,0,0
510 DATA 0,7E,99,99,66,18,0,0
520 DATA 0,0,FF,99,66,18,0,0
530 DATA 0,0,E7,99,66,18,0,0
540 DATA 0,0,81,E7,66,18,0,0
550 DATA 0,0,0,81,7E,18,0,0
560 DATA 0,0,FF,99,66,18,0,0
570 DATA 0,7E,99,99,66,18,0,0
```

## PROGRAM 9

## ALL SPRITE SIZES IN ONE PROGRAM

When you specify Sprite Size and Magnification using the BASIC 'SCREEN' statement, not only is the Screen Mode completely re-initialised, but the SPRITE PATTERN Table is cleared too. (Simply specifying the 'SCREEN' number, without Sprite Size or Magnification, leaves the SPRITE Tables unaltered, but re-initialises the Mode).

This program shows how to change Sprite Size and Magnification whilst a program is running, using 'VDP(x)' statement, although it is not possible to obtain the different kinds of Sprite on the screen at the same time.

The program uses Program 1 as a Subroutine: either load Program 1 before entering this program, or MERGE"CAS:" load it after entering this program.

Lines 10 and 20 initialise the screen, ensure that the Sprite data in VDP(1) is set to 0, and load the Sprite Patterns via the Subroutine (Program 1). Lines 30 to 50 put three 8-byte Sprite patterns onto the screen - unmagnified.

After a short delay - through the subroutine at Line 300 - the Sprite Magnification bit of VDP(1) is set in Line 80 - to produce Magnified 8-byte Sprites. After another delay, all the Sprites are cleared from the screen - by entering '208' for the vertical position of the Sprite with the lowest Plane number - and VDP(1) is reset for 'no magnification' (Line 120).

The process is then repeated for the 32-byte Sprites - selection of this size being done in Line 140, by setting the Size bit of VDP(1). Then, after a short message, the whole program is repeated again. To break the program, enter 'CTRL-STOP'.

The reader is encouraged to experiment with this program. For example, try changing the magnification of the Sprites whilst they are on the move. Also, notice that the top left corner of each Sprite remains in the same position whatever its size or magnification. This is always the reference point for the location of a Sprite.

```
4   '
5   ' *** SET SCREEN & LOAD SPRITES ***
6   '
10  SCREEN 1:COLOR 15,1,1:VDP(1)=VDP(1) AND &HFC
20  GOSUB 1000
24  '
25  ' *** 8-BYTE SPRITES ***
26  '
30  PUTSPRITE 1,(100,30),8,5
40  PUTSPRITE 2,(100,50),7,6
50  PUTSPRITE 3,(100,80),3,7
60  LOCATE 0,20:PRINT "8-BYTE SPRITES, UNMAGNIFIED"
70  GOSUB 300
80  VDP(1)=VDP(1) OR 1
90  LOCATE 0,20:PRINT "8-BYTE SPRITES, MAGNIFIED  "
100 GOSUB 300
110 PUTSPRITE 1,(90,208)
114 '
115 ' *** 32-BYTE SPRITES ***
116 '
120 VDP(1)=VDP(1) AND &HFE
130 LOCATE 0,20:PRINT "32-BYTE SPRITES, UNMAGNIFIED"
140 VDP(1)=VDP(1) OR 2
150 PUTSPRITE 1,(100,30),13,32
160 PUTSPRITE 2,(100,50),10,33
170 PUTSPRITE 3,(100,80),8,34
180 GOSUB 300
190 LOCATE 0,20:PRINT "32-BYTE SPRITES, MAGNIFIED  "
200 VDP(1)=VDP(1) OR 1
210 GOSUB 300
220 PUTSPRITE 1,(30,208)
230 CLS:LOCATE 2,5:PRINT "ALL IN THE SAME PROGRAM"
240 GOSUB 300
250 VDP(1)=VDP(1) AND &HFC
260 CLS:GOTO30
294 '
295 ' *** DELAY ***
296 '
300 FOR I=1 TO 2000:NEXT:RETURN

1000 ' *** SPRITE LOADER  - PROGRAM 1 ***
```

# APPENDIX C

## VDP TABLES

The values in VDP Registers 2 to 6 determine the Base addresses of the various Tables in VRAM. The way these addresses are derived is discussed fully in Section 1.2.2: this Appendix provides Tables of all the possible values for the Registers.

## VDP REGISTER 2: Base Addresses Pattern Name Table

| HEX VALUE | START ADDRESS (HEX) |
|---|---|
| 00 | 0000 |
| 01 | 0400 |
| 02 | 0800 |
| 03 | 0C00 |
| 04 | 1000 |
| 05 | 1400 |
| 06 | 1800 |
| 07 | 1C00 |
| 08 | 2000 |
| 09 | 2400 |
| 0A | 2800 |
| 0B | 2C00 |
| 0C | 3000 |
| 0D | 3400 |
| 0E | 3800 |
| 0F | 3C00 |

## VDP REGISTER 4: Base Addresses for Pattern Generator Table

| HEX VALUE | START ADDRESS (HEX) |
|---|---|
| 00 | 0000 |
| 01 | 0800 |
| 02 | 1000 |
| 03 | 1800* |
| 04 | 2000 |
| 05 | 2800 |
| 06 | 3000 |
| 07 | 3800** |

* In mode 2, Start address=0

** In mode 2, Start address=2000 hex

# VDP REGISTER 3: Base Addresses for Colour Table

| HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) |
|---|---|---|---|---|---|---|---|---|---|
| 00 | 0000 | 34 | 0D00 | 68 | 1A00 | 9C | 2700 | D0 | 3400 |
| 01 | 0040 | 35 | 0D40 | 69 | 1A40 | 9D | 2740 | D1 | 3440 |
| 02 | 0080 | 36 | 0D80 | 6A | 1A80 | 9E | 2780 | D2 | 3480 |
| 03 | 00C0 | 37 | 0DC0 | 6B | 1AC0 | 9F | 27C0 | D3 | 34C0 |
| 04 | 0100 | 38 | 0E00 | 6C | 1B00 | A0 | 2800 | D4 | 3500 |
| 05 | 0140 | 39 | 0E40 | 6D | 1B40 | A1 | 2840 | D5 | 3540 |
| 06 | 0180 | 3A | 0E80 | 6E | 1B80 | A2 | 2880 | D6 | 3580 |
| 07 | 01C0 | 3B | 0EC0 | 6F | 1BC0 | A3 | 28C0 | D7 | 35C0 |
| 08 | 0200 | 3C | 0F00 | 70 | 1C00 | A4 | 2900 | D8 | 3600 |
| 09 | 0240 | 3D | 0F40 | 71 | 1C40 | A5 | 2940 | D9 | 3640 |
| 0A | 0280 | 3E | 0F80 | 72 | 1C80 | A6 | 2980 | DA | 3680 |
| 0B | 02C0 | 3F | 0FC0 | 73 | 1CC0 | A7 | 29C0 | DB | 36C0 |
| 0C | 0300 | 40 | 1000 | 74 | 1D00 | A8 | 2A00 | DC | 3700 |
| 0D | 0340 | 41 | 1040 | 75 | 1D40 | A9 | 2A40 | DD | 3740 |
| 0E | 0380 | 42 | 1080 | 76 | 1D80 | AA | 2A80 | DE | 3780 |
| 0F | 03C0 | 43 | 10C0 | 77 | 1DC0 | AB | 2AC0 | DF | 37C0 |
| 10 | 0400 | 44 | 1100 | 78 | 1E00 | AC | 2B00 | E0 | 3800 |
| 11 | 0440 | 45 | 1140 | 79 | 1E40 | AD | 2B40 | E1 | 3840 |
| 12 | 0480 | 46 | 1180 | 7A | 1E80 | AE | 2B80 | E2 | 3880 |
| 13 | 04C0 | 47 | 11C0 | 7B | 1EC0 | AF | 2BC0 | E3 | 38C0 |
| 14 | 0500 | 48 | 1200 | 7C | 1F00 | B0 | 2C00 | E4 | 3900 |
| 15 | 0540 | 49 | 1240 | 7D | 1F40 | B1 | 2C40 | E5 | 3940 |
| 16 | 0580 | 4A | 1280 | 7E | 1F80 | B2 | 2C80 | E6 | 3980 |
| 17 | 05C0 | 4B | 12C0 | 7F | 1FC0 | B3 | 2CC0 | E7 | 39C0 |
| 18 | 0600 | 4C | 1300 | 80 | 2000 | B4 | 2D00 | E8 | 3A00 |
| 19 | 0640 | 4D | 1340 | 81 | 2040 | B5 | 2D40 | E9 | 3A40 |
| 1A | 0680 | 4E | 1380 | 82 | 2080 | B6 | 2D80 | EA | 3A80 |
| 1B | 06C0 | 4F | 13C0 | 83 | 20C0 | B7 | 2DC0 | EB | 3AC0 |
| 1C | 0700 | 50 | 1400 | 84 | 2100 | B8 | 2E00 | EC | 3B00 |
| 1D | 0740 | 51 | 1440 | 85 | 2140 | B9 | 2E40 | ED | 3840 |
| 1E | 0780 | 52 | 1480 | 86 | 2180 | BA | 2E80 | EE | 3B80 |
| 1F | 07C9 | 53 | 14C0 | 87 | 21C0 | BB | 2EC0 | EF | 3BC0 |
| 20 | 0800 | 54 | 1500 | 88 | 2200 | BC | 2F00 | F0 | 3C00 |
| 21 | 0840 | 55 | 1540 | 89 | 2240 | BD | 2F40 | F1 | 3C40 |
| 22 | 0880 | 56 | 1580 | 8A | 2280 | BE | 2F80 | F2 | 3C80 |
| 23 | 08C0 | 57 | 15C0 | 8B | 22C0 | BF | 2FC0 | F3 | 3CC0 |
| 24 | 0900 | 58 | 1600 | 8C | 2300 | C0 | 3000 | F4 | 3D00 |
| 25 | 0940 | 59 | 1640 | 8D | 2340 | C1 | 3040 | F5 | 3D40 |
| 26 | 0980 | 5A | 1680 | 8E | 2380 | C1 | 3080 | F6 | 3D80 |
| 27 | 09C0 | 5B | 16C0 | 8F | 23C0 | C3 | 30C0 | F7 | 3DC0 |
| 28 | 0A00 | 5C | 1700 | 90 | 2400 | C4 | 3100 | F8 | 3E00 |
| 29 | 0A40 | 5D | 1740 | 91 | 2440 | C5 | 3140 | F9 | 3E40 |
| 2A | 0A80 | 5E | 1780 | 92 | 2480 | C6 | 3180 | FA | 3E80 |
| 2B | 0AC0 | 5F | 17C0 | 93 | 24C0 | C7 | 31C0 | FB | 3EC0 |
| 2C | 0B00 | 60 | 1800 | 94 | 2500 | C8 | 3200 | FC | 3F00 |
| 2D | 0B40 | 61 | 1840 | 95 | 2540 | C9 | 3240 | FD | 3F40 |
| 2E | 0B80 | 62 | 1880 | 96 | 2580 | CA | 3280 | FE | 3F80 |
| 2F | 0BC0 | 63 | 18C0 | 97 | 25C0 | CB | 32C0 | FF | 3FC0 |
| 30 | 0C00 | 64 | 1900 | 98 | 2600 | CC | 3300 | | |
| 31 | 0C40 | 65 | 1940 | 99 | 2640 | CD | 3340 | | |
| 32 | 0C80 | 66 | 1980 | 9A | 2680 | CE | 3380 | | |
| 33 | 0CC0 | 67 | 19C0 | 9B | 26C0 | CF | 33C0 | | |

## VDP REGISTER 5: Base Addresses for Sprite Attribute Table

| HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) | HEX VALUE | START ADDRESS (HEX) |
|---|---|---|---|---|---|---|---|
| 00 | 0000 | 20 | 1000 | 40 | 2000 | 60 | 3000 |
| 01 | 0080 | 21 | 1080 | 41 | 2080 | 61 | 3080 |
| 02 | 0100 | 22 | 1100 | 42 | 2100 | 62 | 3100 |
| 03 | 0180 | 23 | 1180 | 43 | 2180 | 63 | 3180 |
| 04 | 0200 | 24 | 1200 | 44 | 2200 | 64 | 3200 |
| 05 | 0280 | 25 | 1280 | 45 | 2280 | 65 | 3280 |
| 06 | 0300 | 26 | 1300 | 46 | 2300 | 66 | 3300 |
| 07 | 0380 | 27 | 1380 | 47 | 2380 | 67 | 3380 |
| 08 | 0400 | 28 | 1400 | 48 | 2400 | 68 | 3400 |
| 09 | 0480 | 29 | 1480 | 49 | 2480 | 69 | 3480 |
| 0A | 0500 | 2A | 1500 | 4A | 2500 | 6A | 3500 |
| 0B | 0580 | 2B | 1580 | 4B | 2580 | 6B | 3580 |
| 0C | 0600 | 2C | 1600 | 4C | 2600 | 6C | 3600 |
| 0D | 0680 | 2D | 1680 | 4D | 2680 | 6D | 3680 |
| 0E | 0700 | 2E | 1700 | 4E | 2700 | 6E | 3700 |
| 0F | 0780 | 2F | 1780 | 4F | 2780 | 6F | 3780 |
| 10 | 0800 | 30 | 1800 | 50 | 2800 | 70 | 3800 |
| 11 | 0880 | 31 | 1880 | 51 | 2880 | 71 | 3880 |
| 12 | 0900 | 32 | 1900 | 52 | 2900 | 72 | 3900 |
| 13 | 0980 | 33 | 1980 | 53 | 2980 | 73 | 3980 |
| 14 | 0A00 | 34 | 1A00 | 54 | 2A00 | 74 | 3A00 |
| 15 | 0A80 | 35 | 1A80 | 55 | 2A80 | 75 | 3A80 |
| 16 | 0B00 | 36 | 1B00 | 56 | 2B00 | 76 | 3B00 |
| 17 | 0B80 | 37 | 1B80 | 57 | 2B80 | 77 | 3B80 |
| 18 | 0C00 | 38 | 1C00 | 58 | 2C00 | 78 | 3C00 |
| 19 | 0C80 | 39 | 1C80 | 59 | 2C80 | 79 | 3C80 |
| 1A | 0D00 | 3A | 1D00 | 5A | 2D00 | 7A | 3D00 |
| 1B | 0D80 | 3B | 1D80 | 5B | 2D80 | 7B | 3D80 |
| 1C | 0E00 | 3C | 1E00 | 5C | 2E00 | 7C | 3E00 |
| 1D | 0E80 | 3D | 1E80 | 5D | 2E80 | 7D | 3E80 |
| 1E | 0F00 | 3E | 1F00 | 5E | 2F00 | 7E | 3F00 |
| 1F | 0F80 | 3F | 1F80 | 5F | 2F80 | 7F | 3F80 |

## VDP REGISTER 6: Base Addresses for Sprite Pattern Generator Table

| HEX VALUE | START ADDRESS (HEX) |
|---|---|
| 00 | 0000 |
| 01 | 0800 |
| 02 | 1000 |
| 03 | 1800 |
| 04 | 2000 |
| 05 | 2800 |
| 06 | 3000 |
| 07 | 3800 |

# APPENDIX D

## ACCESSING A CHARACTER

## FROM THE KEYBOARD

It is often useful to be able to access a character number direct from the Keyboard, particularly when generating Sprite shapes or new patterns for characters. It is easier, for example, to enter A$="AaBc" than it is to enter A$=CHR$(65)+CHR$(97)+CHR$(66)+CHR$(99).

This Appendix shows how all but three character numbers can be obtained from the Keyboard: the three unobtainable characters are 0 (null or blank), 127 (delete) and 255 (Cursor character).

Note: On European versions of the MSX, Characters 0-31 are formed by two bytes - the first "switching on" the lower set: these cannot therefore be used as the others in string elements to produce character numbers.

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS |
|---|---|---|---|---|
| 0 | 00 | Blank | NOT AVAILABLE | |
| 1 | 01 | ☺ | [ | Graph |
| 2 | 02 | ☻ | [ | Graph & Shift |
| 3 | 03 | ♥ | ' | Graph & Shift |
| 4 | 04 | ♦ | ; | Graph & Shift |
| 5 | 05 | ♣ | " | Graph |
| 6 | 06 | ♠ | ; | Graph |
| 7 | 07 | • | 9 | Graph |
| 8 | 08 | ◘ | 9 | Graph & Shift |
| 9 | 09 | ○ | 0 | Graph |
| 10 | 0A | ◙ | 0 | Graph & Shift |
| 11 | 0B | ♂ | M | Graph |
| 12 | 0C | ♀ | M | Graph & Shift |
| 13 | 0D | ♪ | ] | Graph |
| 14 | 0E | ♫ | ] | Graph & Shift |
| 15 | 0F | ☼ | Z | Graph |

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS |
|---|---|---|---|---|
| 16 | 10 |  | G | Graph & Shift |
| 17 | 11 |  | B | Graph |
| 18 | 12 |  | T | Graph |
| 19 | 13 |  | H | Graph |
| 20 | 14 |  | F | Graph |
| 21 | 15 |  | G | Graph |
| 22 | 16 |  | \ | Graph & Shift |
| 23 | 17 |  | - | Graph |
| 24 | 18 |  | R | Graph |
| 25 | 19 |  | Y | Graph |
| 26 | 1A |  | V | Graph |
| 27 | 1B |  | N | Graph |
| 28 | 1C |  | X | Graph |
| 29 | 1D |  | / | Graph |
| 30 | 1E |  | \ | Graph |
| 31 | 1F |  | - | Graph & Shift |
| 32 | 20 | Space | SPACE | |
| 33 | 21 | ! | 1 | Shift |
| 34 | 22 | " | ' | Shift |
| 35 | 23 | # | 3 | Shift |
| 36 | 24 | $ | 4 | Shift |
| 37 | 25 | % | 5 | Shift |
| 38 | 26 | & | 7 | Shift |
| 39 | 27 | ' | ' | |
| 40 | 28 | ( | 9 | Shift |
| 41 | 29 | ) | 0 | Shift |
| 42 | 2A | * | 8 | Shift |
| 43 | 2B | + | = | Shift |
| 44 | 2C | , | , | |
| 45 | 2D | - | - | |
| 46 | 2E | . | . | |
| 47 | 2F | / | / | |
| 48 | 30 | 0 | 0 | |
| 49 | 31 | 1 | 1 | |
| 50 | 32 | 2 | 2 | |
| 51 | 33 | 3 | 3 | |
| 52 | 34 | 4 | 4 | |
| 53 | 35 | 5 | 5 | |
| 54 | 36 | 6 | 6 | |
| 55 | 37 | 7 | 7 | |
| 56 | 38 | 8 | 8 | |
| 57 | 39 | 9 | 9 | |
| 58 | 3A | : | ; | Shift |
| 59 | 3B | ; | ; | |
| 60 | 3C | < | , | Shift |
| 61 | 3D | = | = | |
| 62 | 3E | > | . | Shift |
| 63 | 3F | ? | / | shift |

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS |
|---|---|---|---|---|
| 64 | 40 | @ | 2 | Shift |
| 65 | 41 | A | A | Shift |
| 66 | 42 | B | B | Shift |
| 67 | 43 | C | C | Shift |
| 68 | 44 | D | D | Shift |
| 69 | 45 | E | E | Shift |
| 70 | 46 | F | F | Shift |
| 71 | 47 | G | G | Shift |
| 72 | 48 | H | H | Shift |
| 73 | 49 | I | I | Shift |
| 74 | 4A | J | J | Shift |
| 75 | 4B | K | K | Shift |
| 76 | 4C | L | L | Shift |
| 77 | 4D | M | M | Shift |
| 78 | 4E | N | N | Shift |
| 79 | 4F | O | O | Shift |
| 80 | 50 | P | P | Shift |
| 81 | 51 | Q | Q | Shift |
| 82 | 52 | R | R | Shift |
| 83 | 53 | S | S | Shift |
| 84 | 54 | T | T | Shift |
| 85 | 55 | U | U | Shift |
| 86 | 56 | V | V | Shift |
| 87 | 57 | W | W | Shift |
| 88 | 58 | X | X | Shift |
| 89 | 59 | Y | Y | Shift |
| 90 | 5A | Z | Z | Shift |
| 91 | 5B | [ | [ | |
| 92 | 5C | \ | \ | |
| 93 | 5D | ] | ] | |
| 94 | 5E | ^ | 6 | Shift |
| 95 | 5F | _ | - | Shift |
| 96 | 60 | ` | ` | Code* |
| 97 | 61 | a | A | |
| 98 | 62 | b | B | |
| 99 | 63 | c | C | |
| 100 | 64 | d | D | |
| 101 | 65 | e | E | |
| 102 | 66 | f | F | |
| 103 | 67 | g | G | |
| 104 | 68 | h | H | |
| 105 | 69 | i | I | |
| 106 | 6A | j | J | |
| 107 | 6B | k | K | |
| 108 | 6C | l | L | |
| 109 | 6D | m | M | |
| 110 | 6E | n | N | |
| 111 | 6F | o | O | |

* Varies with model of MSX

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS | |
|---|---|---|---|---|---|
| 112 | 70 | p | P | | |
| 113 | 71 | q | Q | | |
| 114 | 72 | r | R | | |
| 115 | 73 | s | S | | |
| 116 | 74 | t | T | | |
| 117 | 75 | u | U | | |
| 118 | 76 | v | V | | |
| 119 | 77 | w | W | | |
| 120 | 78 | x | X | | |
| 121 | 79 | y | Y | | |
| 122 | 7A | z | Z | | |
| 123 | 7B | { | [ | Shift | |
| 124 | 7C | ¦ | \ | Shift | |
| 125 | 7D | } | ] | Shift | |
| 126 | 7E | ~ | | Shift | |
| 127 | 7F | | NOT AVAILABLE | | |
| 128 | 80 | Ç | 9 | Code | & Shift |
| 129 | 81 | ü | G | Code | |
| 130 | 82 | é | U | Code | |
| 131 | 83 | â | Q | Code | |
| 132 | 84 | ä | A | Code | |
| 133 | 85 | à | Z | Code | |
| 134 | 86 | á | , | Code | |
| 135 | 87 | ç | 9 | Code | |
| 136 | 88 | ê | W | Code | |
| 137 | 89 | ë | S | Code | |
| 138 | 8A | è | X | Code | |
| 139 | 8B | ï | D | Code | |
| 140 | 8C | î | E | Code | |
| 141 | 8D | ì | C | Code | |
| 142 | 8E | Ä | A | Code | & Shift |
| 143 | 8F | Å | , | Code | & Shift |
| 144 | 90 | É | U | Code | & Shift |
| 145 | 91 | æ | J | Code | |
| 146 | 92 | Æ | J | Code | & Shift |
| 147 | 93 | ô | R | Code | |
| 148 | 94 | ö | F | Code | |
| 149 | 95 | ò | V | Code | |
| 150 | 96 | û | T | Code | |
| 151 | 97 | ù | B | Code | |
| 152 | 98 | ÿ | 5 | Code | |
| 153 | 99 | Ö | F | Code | & Shift |
| 154 | 9A | Ü | G | Code | & Shift |
| 155 | 9B | ¢ | 4 | Code | |
| 156 | 9C | £ | 4 | Code | & Shift |
| 157 | 9D | ¥ | 5 | Code | & Shift |
| 158 | 9E | ₧ | 2 | Code | & Shift |
| 159 | 9F | ƒ | 1 | Code | |

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS |
|---|---|---|---|---|
| 160 | A0 | á | Y | Code |
| 161 | A1 | í | I | Code |
| 162 | A2 | ó | O | Code |
| 163 | A3 | ú | P | Code |
| 164 | A4 | ñ | N | Code |
| 165 | A5 | Ñ | N | Code & Shift |
| 166 | A6 | a | . | Code |
| 167 | A7 | o | / | Code |
| 168 | A8 | ¿ | / | Code & Shift |
| 169 | A9 | ⌐ | R | Graph & Shift |
| 170 | AA | ¬ | Y | Graph & Shift |
| 171 | AB | ½ | 2 | Graph |
| 172 | AC | ¼ | 1 | Graph |
| 173 | AD | ¡ | 1 | Code & Shift |
| 174 | AE | « | , | Graph & Shift |
| 175 | AF | » | . | Graph & Shift |
| 176 | B0 | Ã | H | Code & Shift |
| 177 | B1 | ã | H | Code |
| 178 | B2 | Ĩ | K | Code & Shift |
| 179 | B3 | ĩ | K | Code |
| 180 | B4 | Õ | L | Code & Shift |
| 181 | B5 | õ | L | Code |
| 182 | B6 | Ũ | ; | Code & Shift |
| 183 | B7 | ũ | ; | Code |
| 184 | B8 | Π | ' | Code & Shift |
| 185 | B9 | ij | ' | Code |
| 186 | BA | ¾ | 3 | Graph |
| 187 | BB | ~ | | Graph |
| 188 | BC | ◊ | C | Graph |
| 189 | BD | ‰ | 5 | Graph |
| 190 | BE | ¶ | 3 | Code & Shift |
| 191 | BF | § | 3 | Code |
| 192 | C0 |  | U | Graph |
| 193 | C1 |  | D | Graph & Shift |
| 194 | C2 |  | O | Graph |
| 195 | C3 |  | O | Graph & Shift |
| 196 | C4 |  | A | Graph |
| 197 | C5 |  | U | Graph & Shift |
| 198 | C6 |  | J | Graph |
| 199 | C7 |  | D | Graph |
| 200 | C8 |  | L | Graph |
| 201 | C9 |  | L | Graph & Shift |
| 202 | CA |  | J | Graph & Shift |
| 203 | CB |  | Q | Graph & Shift |
| 204 | CC |  | Q | Graph |
| 205 | CD |  | E | Graph |
| 206 | CE |  | E | Graph & Shift |
| 207 | CF |  | W | Graph |

| CHARACTER NUMBER. Dec | Hex | CHARACTER SHAPE (MSX Set) | KEY-BOARD CHARACTER to press | PLUS |
|---|---|---|---|---|
| 208 | D0 |  | W | Graph & Shift |
| 209 | D1 |  | S | Graph & Shift |
| 210 | D2 |  | S | Graph |
| 211 | D3 |  | N | Graph & Shift |
| 212 | D4 |  | F | Graph & Shift |
| 213 | D5 |  | V | Graph & Shift |
| 214 | D6 |  | H | Graph & Shift |
| 215 | D7 |  | P | Graph & Shift |
| 216 | D8 |  | 0 | Code & Shift |
| 217 | D9 | ∓ | 2 | Code |
| 218 | DA | ω | ] | Code |
| 219 | DB |  | P | Graph |
| 220 | DC |  | I | Graph |
| 221 | DD |  | K | Graph |
| 222 | DE |  | K | Graph & Shift |
| 223 | DF |  | I | Graph & Shift |
| 224 | E0 | α | 6 | Code |
| 225 | E1 | β | 7 | Code |
| 226 | E2 | Γ | 8 | Code & Shift |
| 227 | E3 | Π | P | Code & Shift |
| 228 | E4 | Σ | ` | Code & Shift* |
| 229 | E5 | σ | ` | Code * |
| 230 | E6 | μ | M | Code |
| 231 | E7 | τ | 8 | Code |
| 232 | E8 | Φ | [ | Code & Shift |
| 233 | E9 | Θ | = | Code |
| 234 | EA | Ω | ] | Code & Shift |
| 235 | EB | δ | 0 | Code |
| 236 | EC | ∞ | 8 | Graph |
| 237 | ED | φ | [ | Code |
| 238 | EE | ∈ | - | Code |
| 239 | EF | ∩ | 4 | Graph |
| 240 | F0 | ≡ | = | Graph & Shift |
| 241 | F1 | ± | = | Graph |
| 242 | F2 | ≥ | . | Graph |
| 243 | F3 | ≤ | , | Graph |
| 244 | F4 | ⌠ | 6 | Graph |
| 245 | F5 | ⌡ | 6 | Graph & Shift |
| 246 | F6 | ÷ | / | Graph & Shift |
| 247 | F7 | ≈ | ` | Graph & Shift* |
| 248 | F8 | ° | Z | Graph & Shift |
| 249 | F9 | • | X | Graph & Shift |
| 250 | FA | • | C | Graph & Shift |
| 251 | FB | √ | 7 | Graph |
| 252 | FC | n | 3 | Graph & Shift |
| 253 | FD | 2 | 2 | Graph & Shift |
| 254 | FE |  | A | Graph & Shift |
| 255 | FF | Cursor | NOT AVAILABLE | |

* On some models, use the "£" Key

# APPENDIX E

## USEFUL ROM ROUTINES

At the beginning of ROM in the MSX there is a Jump Table to a number of routines that can be of value to the machine code programmer. It is advised that the Jump Table is used rather than the actual routine address, since this would maintain compatibility should the actual routine addresses change in future versions.
This Appendix gives details of the routines that are associated with the VDP and the screen display.

0041H Disable screen display
IN: None
OUT: AF,BC Modified

0044H Enables screen display
IN: None
OUT: Modifies AF,BC

0047H Write to a VDP REGISTER
IN: B = Data
C = Register number
OUT: Modifies AF, BC

004AH Read VRAM address
IN: HL= Required VRAM address
OUT: A = Data. AF modified

004DH Write to VRAM address
IN: A = Data
HL= Required VRAM address
OUT: AF modified

0056H Fill VRAM area with specified data
IN: A = data byte
BC= Length of VRAM area to be filled
HL= Start address in VRAM
OUT: Modifies AF,BC

0059H Move block of memory from VRAM to RAM
IN: BC= Length of block to be moved
DE= Destination in RAM
HL= Source in VRAM
OUT: Modifies all

005CH Move block of memory from RAM to VRAM
IN: BC= Length of block to be moved
DE= Destination in VRAM
HL= Source in RAM
OUT: Modifies all

005FH Set VDP Mode according to Accumulator
IN: A = Required Mode (0...3)
OUT: Modifies all

0062H Change screen colours
IN: Foreground colour at F3E9H
Background colour at F3EAH
Border colour at F3EBH
OUT: Modifies all

0066H Initialise - reset - all Sprites
IN: FCAFH = 1,2 or 3 (Mode)
OUT: Modifies all

006CH Initialise Mode 0 (Text Screen)
IN: F3B3H = NAME Table Base address
F3B7H = PATTERN GENERATOR Base address
OUT: Modifies all

006FH Initialise Mode 1 (Screen 1)
IN: F3BDH = NAME Table Base address
F3BFH = COLOUR Table Base adddess
F3C1H = PATTERN GENERATOR Base address
F3C3H = SPRITE ATTRIBUTE Base address
F3C5H = SPRITE PATTERN Base address
OUT: Modifies all

0072H Initialise Mode 2 for BASIC graphics
IN: F3C7H = NAME Table Base address
F3C9H = COLOUR Table Base address
F3CBH = PATTERN GENERATOR Base address
F3CDH = SPRITE ATTRIBUTE Base address
F3CFH = SPRITE PATTERN Base address
OUT: Modifies all

0075H Initialise Mode 3 (Multicolour screen)
IN: F3D1H = NAME Table Base address
F3D5H = PATTERN GENERATOR Base address
F3D7H = SPRITE ATTRIBUTE Base address
F3D9H = SPRITE PATTERN Base address
OUT: Modifies all

0084H Find Sprite Attribute address
IN: A = Plane number for attribute
OUT: HL= Start address of attribute
Modifies AF,DE,HL

```
008AH   Find current Sprite size
            IN: None
           OUT: A = Bytes per Sprite
                Carry Flag set if 32 Byte Sprites

008DH   Print character to Graphic Screen
            IN: A = Character number
           OUT: No change

00A2H   Print character to screen, Modes 0,1
            IN: A = Character number
           OUT: No change

00C3H   Clear the screen
            IN: None
           OUT: Modifies AF,BC,DE

00C6H   Position cursor on screen, Modes 0,1
            IN: H = Column
                L = Row
           OUT: Modifies AF

00CCH   Erase Function key display
            IN: None
           OUT: Modifies all

00CFH   Display Function keys
            IN: None
           OUT: Modifies all

013EH   Read VDP Status Register
            IN: None
           OUT: A = data
```

# APPENDIX F

## USEFUL ADDRESSES AND HOOKS

The MSX uses the 'top' area of RAM to store data and to provide 'Hooks' for routines in ROM. The Hooks enable the system to be expanded, and also enable the user to 'interject' his own machine code routines. When using the Hooks in this way, it is recommended that the contents of those Z80 registers used are preserved for the return, since this data may be used by the ROM routines too. The Hooks comprise five bytes, each one initialised to contain a machine code 'RETURN'. Any added peripherals - such as discs - or system developments could result in some of the Hooks being used: it is wise to check the Hook data, therefore, before making any changes.

This Appendix details the Storage and Hook addresses that are associated with the display screens: the initialised contents are not given, since these can vary from machine to machine, and, obviously, during system use.

The addresses are given in HEX. Note that where data (such as an address) is stored in two consecutive bytes, the first byte contains the least significant byte of the data, and the second byte contains the most significant byte of the data. Thus, data such as F123 hex would be stored in the order 23, F1 hex.

## ADDRESSES

| | |
|---|---|
| F3AE | Initialising width for Text Screen Mode 0 |
| F3AF | Initialising width for Screen Mode 1 |
| F3B0 | Current line width |
| F3B3/4 | Mode 0 NAME Table Base address |
| F3B7/8 | Mode 0 PATTERN GENERATOR Base address |
| F3BD/E | Mode 1 NAME Table Base address |
| F3BF/C0 | Mode 1 COLOUR Table Base address |
| F3C1/2 | Mode 1 PATTERN GENERATOR Base address |
| F3C3/4 | Mode 1 SPRITE ATTRIBUTE Base address |
| F3C5/6 | Mode 1 SPRITE PATTERN Base address |
| F3C7/8 | Mode 2 NAME Table Base address |
| F3C9/A | Mode 2 COLOUR Table Base address |
| F3CB/C | Mode 2 PATTERN GENERATOR Base address |
| F3CD/E | Mode 2 SPRITE ATTRIBUTE Base address |
| F3CF/D0 | Mode 2 SPRITE PATTERN Base address |
| F3D1/2 | Mode 3 NAME Table Base address |
| F3D5/6 | Mode 3 PATTERN GENERATOR Base address |
| F3D7/8 | Mode 3 SPRITE ATTRIBUTE Base address |
| F3D9/A | SPRITE PATTERN Base address |
| F3DC | Cursor 'Y' position |
| F3DD | Cursor 'X' position |
| F3DE | Function key display flag |
| F3DF | VDP Register 0 store |
| F3E0 | VDP Register 1 store |
| F3E1 | VDP Register 2 store |
| F3E2 | VDP Register 3 store |
| F3E3 | VDP Register 4 store |
| F3E4 | VDP Register 5 store |
| F3E5 | VDP Register 6 store |
| F3E6 | VDP Register 7 store |
| F3E7 | VDP Status Register (8) store |
| F3E9 | Foreground colour |
| F3EA | Background colour |
| F3EB | Border/backdrop colour |
| F87F | Start of FUNCTION key storage: 160 bytes |
| F920/1 | Pointer to ROM character pattern set |
| F922/3 | Current NAME Table Base address |
| F924/5 | Current PATTERN GENERATOR Base address |
| F926/7 | Current SPRITE PATTERN Base address |
| F928/9 | Current SPRITE ATTRIBUTE Base address |
| FBCC | Code storage for cursor |
| FCAF | Current Screen Mode |
| FCB0 | Old Screen Mode |

## HOOKS

| | |
|---|---|
| FD9A | Start of Interrupt routines |
| FD9F | Start of timed Interrupt routines |
| FDC7 | Start of pattern initialising routine |
| FDE5 | Start of 'ON-GOTO' interrupt routines |
| FFBB | Start of 'SCREEN' statement |

[redacted]

This well written book takes an in-depth look at the very powerful graphics screen displays of the MSX microcomputer, revealing many hidden secrets and providing an absolute mine of invaluable information. Designed not only to give the newcomer to computing an insight into what's happening behind the MSX Screens, but also to provide the experienced programmer with the reference information needed to save on midnight oil. A must for the user who wishes to make the most of the outstanding MSX graphics capabilities.

8.95 net

*Published by*

Kuma MSX

Kuma Computers Ltd., Pang[illegible] Berkshire, England
Telephone 07357-4335 [illegible]62 TELFAC

£ 8.95