
SIGMA
PRESS
MSX
OPEN SWAP VPOKE VPEEK SCREEN COLOUR PLAY CIRCLE DRAW MAXFILES TRON VARPTR STICK ERA INTERVAL
PAINT ELSE DELETE PAD PRESET SOUND MOTOR KEY FOR TO READ POKE VAL NEXT DEFUSR
ON STRIG LOCATE GOTO USR PRINT BIN$ CHR$ MID$ GOSUB INKEY$ A$C KEYOFF STOP PUT
A PROGRAMMER'S GUIDE TO THE MSX SYSTEM
C.I.BURKINSHAW & R.GOODLEY

A Programmer's Guide to the MSX System

C. I. Burkinshaw
and
R. Goodley

Σ
SIGMA
PRESS

ISBN: 1 85058 015 4

Published by:
SIGMA PRESS
5 Alton Road
Wilmslow
Cheshire
UK.

Distributors:
UK, Europe, Africa:
JOHN WILEY & SONS LIMITED
Baffins Lane, Chichester,
West Sussex, England.

Australia:
JOHN WILEY & SONS INC.,
GPO Box 859, Brisbane,
Queensland 40001, Australia.

Acknowledgment
MSX is the registered trademark of Microsoft Corp., U.S.A.

Printed in Great Britain
by J. W. Arrowsmith Ltd., Bristol

FOREWORD

The video display processor originally incorporated in the MSX specification was Texas Instrument's 9929A. This unit has been replaced by the enhanced 9129. For programming purposes, the two processors may be regarded as identical.

The VDP constructs the display image from several data tables located in 16K of dedicated video RAM. The table locations in each display mode are returned by the system variable BASE(X). We believe these to be consistent between all models with version 1 BASIC. These table locations, given in Appendix C, are assumed by the relevant examples.

MSX BASIC permits program files to be handled in several formats. An undocumented instruction: RUN"filename" may be used to load and run ASCII program files (in place of LOAD"filename",R). At the time of going to press we had been unable to confirm the status of this instruction and no further reference has been made to it.

To our parents,
without whom it would not have been possible

CONTENTS

CHAPTER 1

INTRODUCTION

Overview. Memory Organisation. The tape interface. Display modes. V.D.P. and Sound chips.

Overview

The MSX standard is a specification introduced by Microsoft Inc. in mid 1983. It includes the BASIC language, operating system and external connectors (including R.O.M. cartridge & joystick port). Software is interchangable between all MSX machines (assuming adequate memory is available). All models (except the Spectravideo SVI-728 and Yamaha CX5M) feature a built in power supply. Facilities that are not required in the basic system are also rigidly specified: parallel 'centronics type' printer port, second joystick port, RS-232C interface and second cartridge slot.

The disc operating system (MSX-DOS) uses the same disc format as MS-DOS but no physical disc size has been agreed. MSX DOS requires a minimum of 64K of RAM.

The central processing unit is a Z-80A or equivalent running at 3.58MHz supported by the Texas Instruments TMS 9129 (Europe) video display processor. The VDP has 16K of dedicated video RAM and will support up to 32 sprites, each of a single colour. All sprites appear in front of the pattern display plane.

The VDP also generates the system interrupt. This occurs on completion of each display scan (approximately 1/50th second UK), and is used for keyboard analysis, for example.

The Z-80 has sufficient addressing capability to access a memory space of 64K bytes. On power up the system R.O.M. occupies the lower 32K of memory. An additional minimum of 8K of user RAM is provided. In practice, the method of memory management will limit the minimum RAM to 16K with almost all systems providing either 32K or 64K.

The BASIC interpreter is only able to utilize 32K of RAM - irrespective of additional on-board R.A.M. - with some 28K being available for program storage. MSX BASIC is Microsoft standard BASIC (version 4.5) with extensions primarily for graphics and sound manipulation.

Principle features include a full screen editor, interrupt driven commands and 14 digit accuracy with the default variable type.

The sound facilities are provided by a General Instruments AY-3-8910 programmable sound generator (PSG) which provides 3 voices over a range of 8 octaves. The PSG also handles joystick input; the minimum MSX specification requires a single standard D type joystick port.

An 8255 Parallel I/O PPI (Programmable peripheral interface) is used to select the memory banks which will appear to the processor in four 16K 'slots' (see later), and for keyboard scanning. If a printer port is fitted (a standardized extension) this is also serviced by the 8255.

A second optional standardised extension is an RS-232C interface. The LSI components required for this cartridge are the 8251 communication interface chip and an 8253 programmable interval timer.

One cartridge slot is obligatory with two the standard. At the time of going to press the following models had been announced for the UK market:

Hitachi MB-H80
Toshiba HX-10
Sony HB-75 (48K ROM)
Sanyo MPC 100
JVC HC-7GB
Yashica YC-64
Sega - Yeno DPH-64
Goldstar FC-20
Canon V-20
Yamaha CX5M (synthesiser + MSX computer)
Mitsubishi MLF-48
Mitsubishi MLF-80
Panasonic CF2800

Memory Organisation

The Z80 may read and write data to and from a memory area of up to sixty-four kilobytes. The MSX design permits this 64K to be selected from a larger memory bank. The bank is composed of four primary 'slots', each of which may hold up to 64K. Memory selection is handled in blocks or pages of 16K. The memory area addressable by the Z80 contains four 16K pages.

Port A of the 8255 PPI is used to determine which slot provides each 16K block. The two least significant bits specifing the slot which will provide the 0-16K block and so on.

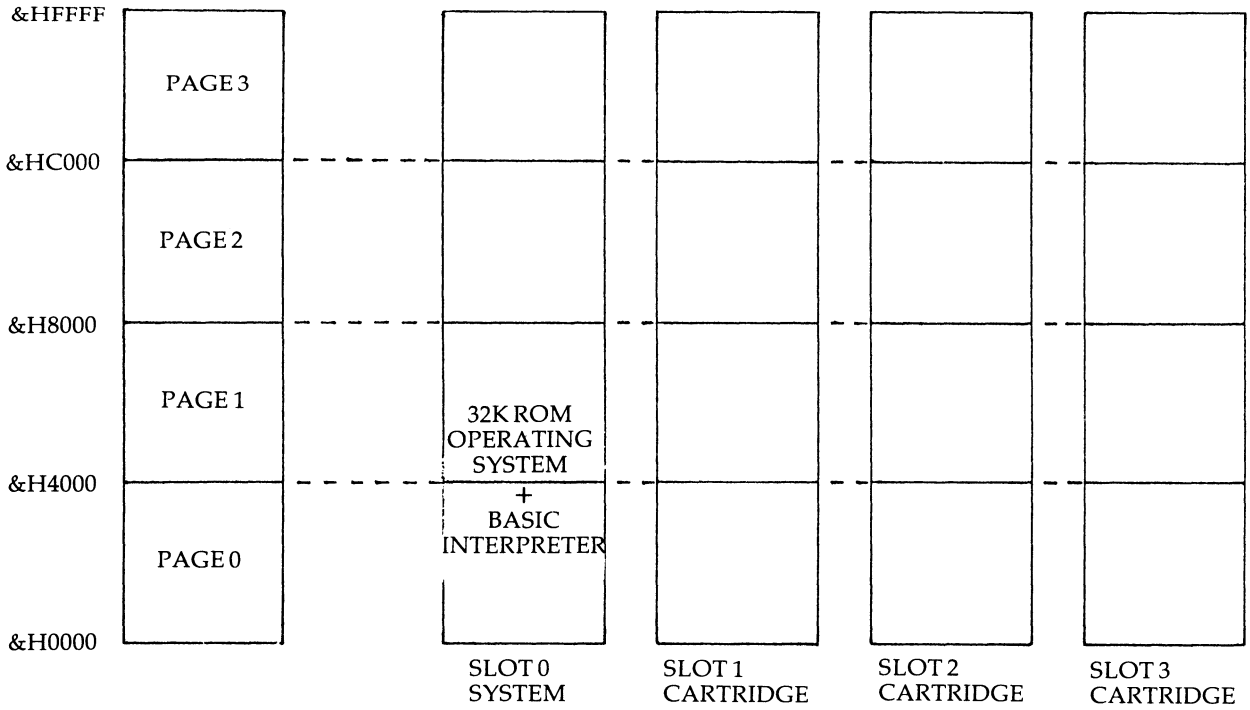


FIGURE 1.1 MEMORY MAP

It should be noted that a block may only appear to the processor in the same location that it occupies in the primary slot i.e. the block of slot four from 48-64K may only appear to the processor in that position and not from 0-16, 16-32 or 32-48K.

The primary slots are numbered 0 to 3. Slot 0 holds the 32K ROM Operating system / Interpreter. The majority of MSX computers possess 64K of RAM. This is normally placed in one slot. For example, the Sanyo MPC-100 uses slot 3 to house the RAM whilst the Toshiba HX-10 uses slot 2. One of the remaining slots is for cartridge use whilst the other may be configured as either another cartridge slot or an expansion bus. BASIC program storage starts from location 32768.

Input/Output Ports

In addition to being able to address 64K of memory, the Z80 may input or output to 256 eight bit ports. Ports 128 to 255 (&H80- &HFF) are assigned to system components and extensions: RS-232C, printer, display processor, sound generator, light pen etc. The remainder are reserved. The NMI (Non Maskable Interrupt) is not useable as the vector location 66H is used by MSX DOS.

Input / output should be handled using the resident ROM routines as system locations are not guaranteed to be consistant between machines. An exception is made in the case of the display processor where rapid data transfer may be necessary.

Tape Interface

A dedicated tape unit is not required. The default transfer rate is 1200 baud (roughly bits per second) and may operate at rates up to 2400 baud. Detection and adjustment to a non-standard rate is automatic. The external connection is an eight pin DIN socket with provision for tape motor control. Modulation is by frequency shift keying under software control. The optimum recording & playback levels will be determined by the tape recorder; however a near maximum setting is often necessary. A BASIC command (MOTOR) is available for tape motor control.

Three categories of file are supported:

1. Program files to cassette. The commands are CLOAD, CSAVE and CLOAD? to verify.

2. ASCII files using SAVE and LOAD. No verify command is available. ASCII program files may be merged.

3. Memory image using the BSAVE and BLOAD commands. Again no verify command is available.

Display Modes

The TMS 9129A is capable of displaying a pattern plane with up to 15 colours (plus transparent) independent of any sprites.

Four display modes are supported, two text and two graphic:

1. 40 * 24 cell text mode, 2 colours and no sprites. 8*6 pixel character cells. 256 cell character set.

2. 32 * 24 cell text mode, 16 colours. 8*8 character cells. 256 cell character set.

3. High resolution 32 * 24 cell Graphic mode, 16 colours. Essentially the 32*24 text mode with a 768 cell character set in order to allow a 'bit-mapped' screen - plus extra colour information.

4. Multicolour mode, 16 colours - each 4*4 pixel block may be specified to be one colour. No 'characters' are available.

The display mode is selected using the BASIC 'SCREEN' command or by amending the three mode bits in two of the 8 VDP write only registers (M3 - bit 6 of register 0, M1 & M2 - bits 3 and 4 of register 1). In all modes except the 40 column text mode, the border colour may be independently set.

The VDP scans the 16K of dedicated video R.A.M. This R.A.M. is independent of the Z80 memory area and may only be written to or read from by the CPU via the VDP. The VDP incorporates no facilities for scrolling the screen.

The BASIC clear screen command (CLS) is slightly unusual in that it operates in all display modes.

The VDP and Sound chips

Texas TMS 9129A VDP (Video Display Processor)
This is a standard 40 pin dual in line package (D.I.L.I.P.) that uses 16K of dynamic R.A.M. Only the VDP may directly access this video R.A.M. using a unidirectional 8-bit data bus. The VDP is controlled using three lines : RAS, CAS and R/W which derive from the address bus and processor control lines. The VDP is also connected to the system data bus.

A read or write to video RAM may be achieved from BASIC using the VPOKE and VPEEK commands or at the assembly language level via the appropriate ROM routines.

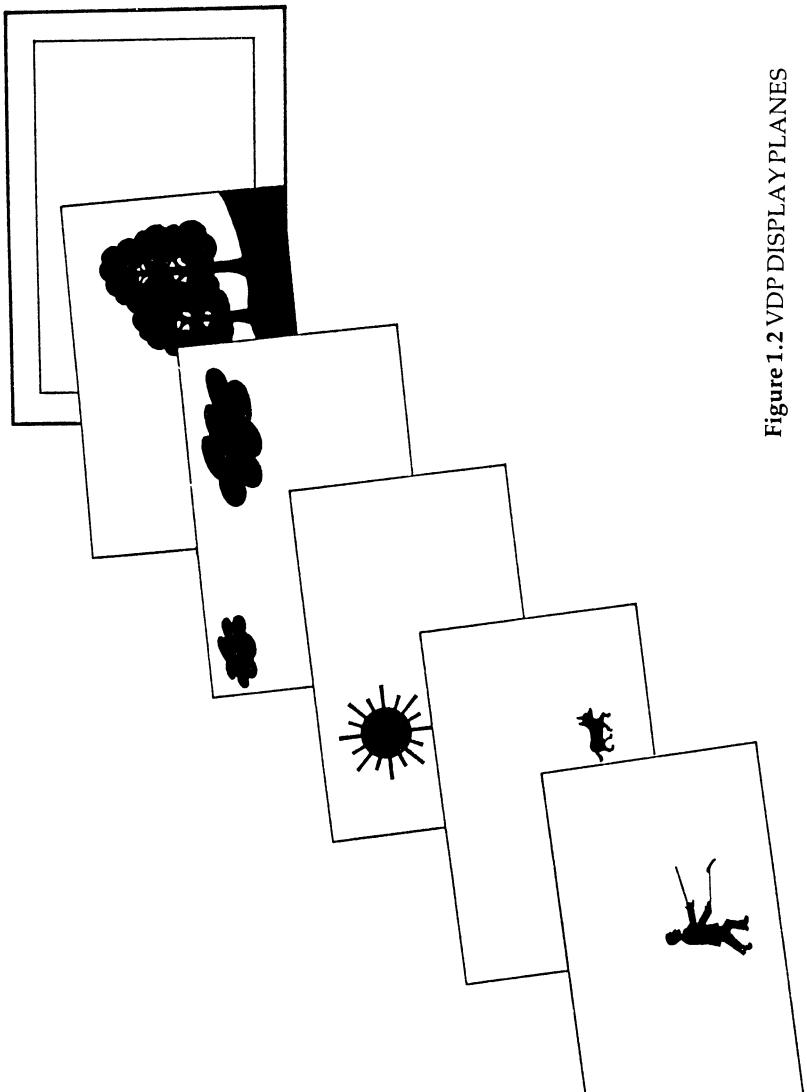


Figure 1.2 VDP DISPLAY PLANES

VDP Display Structure

In all modes the VDP builds up the display primarily from two tables in video RAM:

1. Pattern generator table

2. Pattern name table

The pattern generator table holds the definitions of each character that may be displayed (e.g. the pixel definitions of each printed character). The starting position of the generator table in each display mode is as follows:

Mode 0 : Text 40 : 2048
1 : Text 32 : 0
2 : HRG : 0
3 : Multicolour : 0

In all but the multicolour mode, each character is defined in the pattern generator table using eight bytes to specify the pixel pattern - all standard characters being left justified with the two right hand column and the bottom row blank.

This is illustrated in Figure 1.3

Upto 256 character definitions may be used in the 32 column text mode, in which case the table is 2048 bytes in length. The HRG mode allows upto 768 characters to be defined producing a table of length 6144 bytes. The 40 column text mode allows 256 character definitions, again each of 8 bytes, however the two right hand columns of each character are not displayed. This allows the 40 column display to be produced as each character cell is 8*6 and not 8*8 pixels as in the HRG and 32 column text modes.

In the multicolour mode, the pattern generator table contains 192 entries, each of eight bytes consisting of four pairs of two bytes. Each pair does not define a character but the colour of the four 4*4 cells in a 8*8 character cell.

The layout of the pattern generator is shown in Figure 1.4

Example: In 32 column text mode the generator table starts at VRAM location 0, so the character with character code 65 - an 'A' - is defined from locations 65*8 = 520 to 527.

The pattern name table determines which character appears in each character position (in the multicolour mode the colours are determined).

In the 32 column text mode there are 32 rows of 24 columns and therefore 32*24 = 768 character positions. Each may contain one of the 256 possible characters and consequently the name table is 768 bytes long - each byte specifying the character that will appear in a particular position.

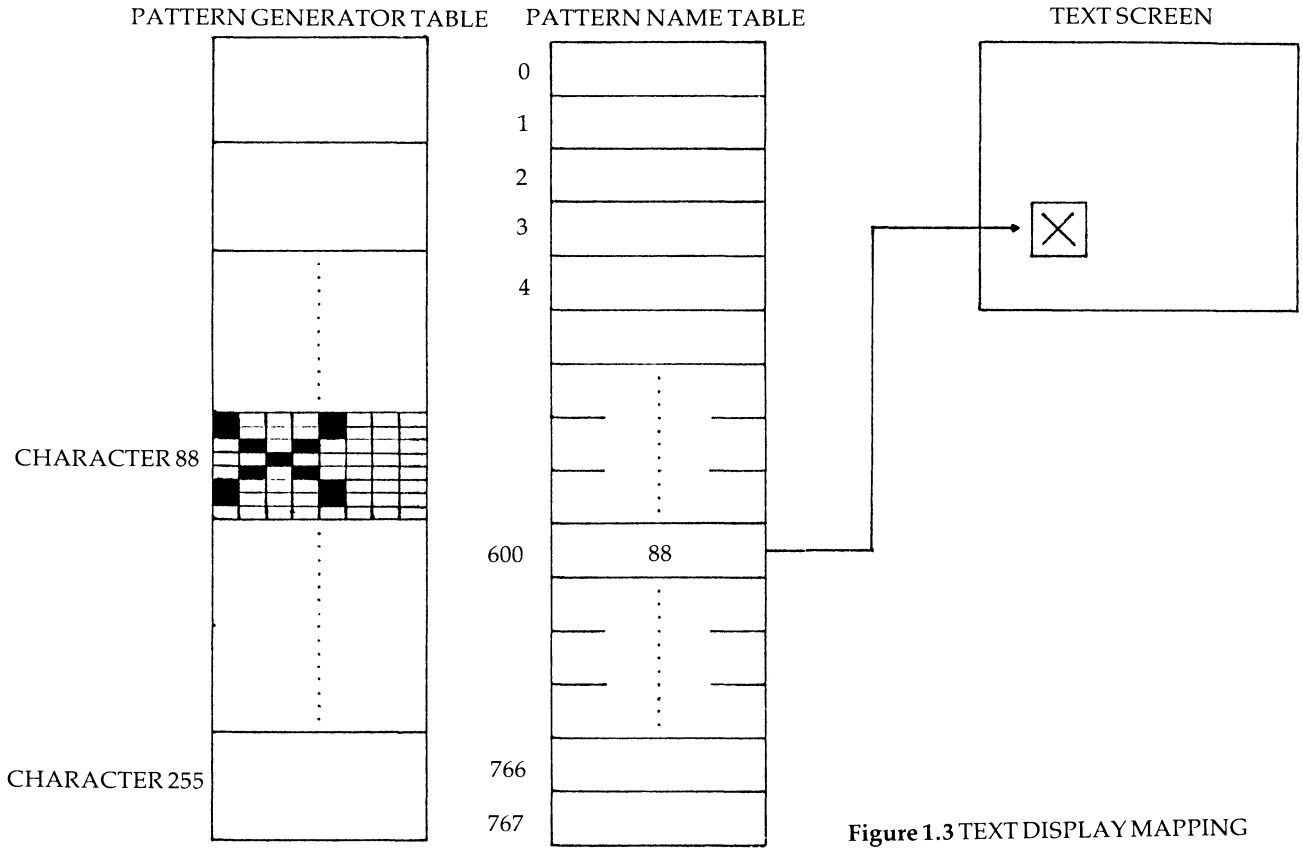


Figure 1.3 TEXT DISPLAY MAPPING

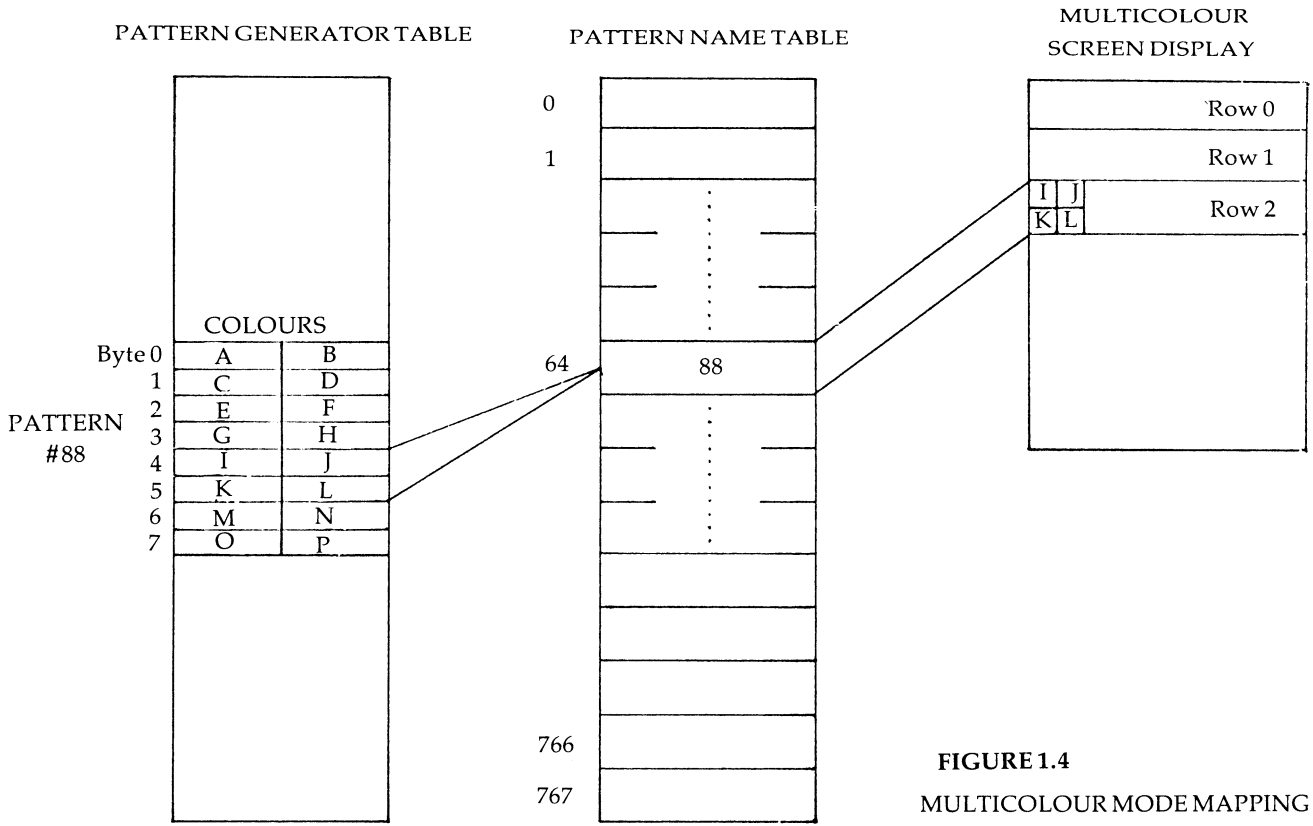


FIGURE 1.4

MULTICOLOUR MODE MAPPING

In the 40 column text mode, the screen name table is 40 * 24 =960 bytes. Again, each specifies the pattern displayed in one screen position. As in the 32 column text mode, the choice is from the 256 pattern variants contained in the pattern generator table.

High Resolution Graphics

In HRG mode there are again 768 bytes in the name table, however as in this mode, there are 768 character definitions available, how does one byte perform the selection?

The screen is divided into three layers and the first 256 entries in the name table select from characters 0-255, the second 256 entries select from characters 256-511 and the final 256 entries select from characters 512-768.

The multicolour name table is 768 bytes in length. The value held in each byte is a reference to an eight byte block in the multicolour generator table.

Two bytes of this block specify the colours taken by the four 4*4 blocks which form the character cell. Which of the four pairs is referenced depends on the screen row of the character cell. If this lies in the top row then the first two bytes are used, if the second row then bytes 3 and 4 are referenced with the subsequent two pairs used for the next two rows. Thus rows 0,4,8,12,16,20 will all reference the first two bytes.

Colour

Although the TMS 9129 VDP is capable of displaying 15 different colours plus transparent (we include black and white as colours), in the 40 column text mode only two colours maybe used, with the border taking the background colour.

From BASIC these are set (as for any mode) using the COLOR command, whilst in machine language, the VDP write only register 7 determines both colours. This is best amended using the appropriate ROM routine.

In the 32 column text mode, the 2K pattern generator table is divided into sets of eight character definitions. For each set, the foreground and background colour is held by one byte in a colour generator table. Thus the colour table is 32 bytes long.

The foreground colour is held in the high order nibble and the background colour in the low order nibble. A consequence of this is that in order to display a letter twice but in seperate colours, the character definition must be duplicated in another position of the pattern table. It may then be allocated a separate colour. In the HRG mode, the foreground / background combination may be specified for each of the 8 'lines' of each character cell. This requires 768*8 = 6144 bytes and is organised in the same format as the 6144 byte pattern generator table. This method of colour allocation renders a scroll procedure problematical.

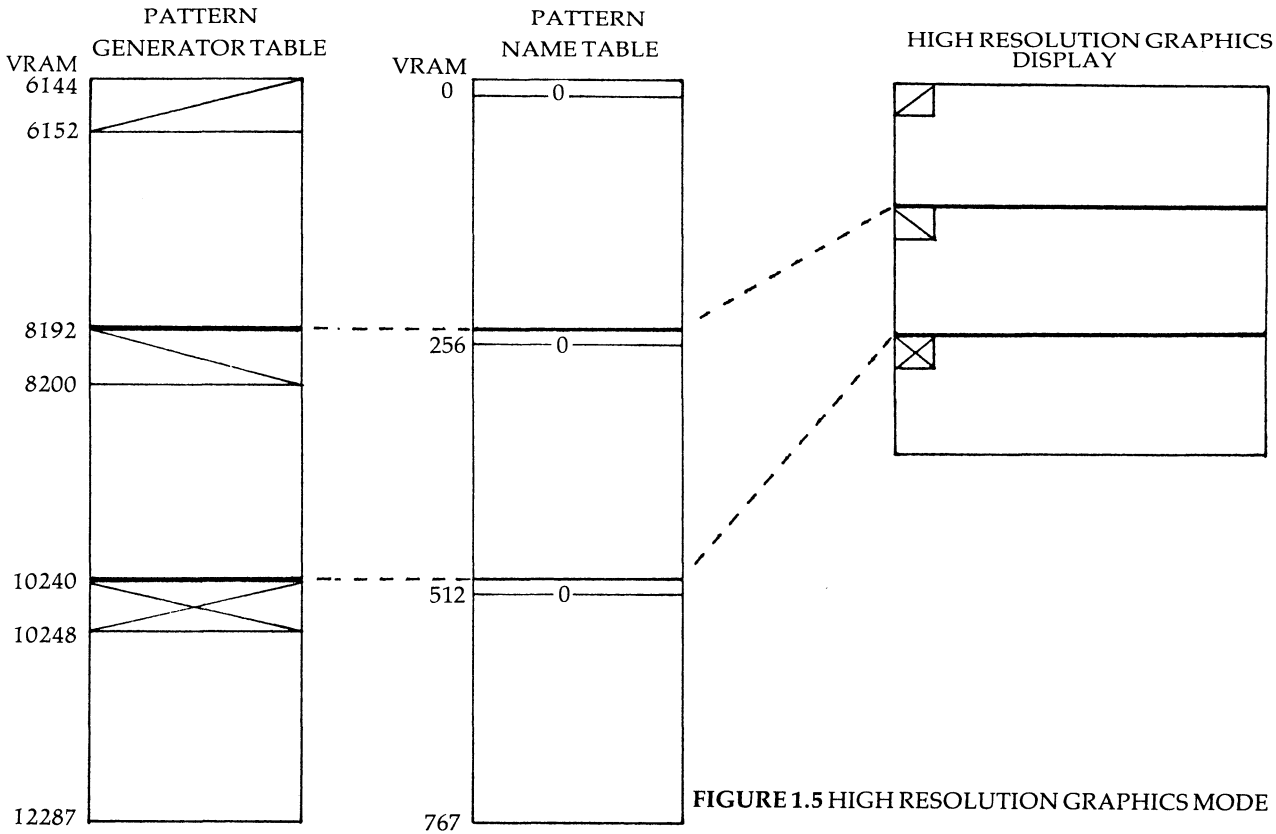


FIGURE 1.5 HIGH RESOLUTION GRAPHICS MODE

Sprites

A sprite is a pattern block which may be either 1, 2 or 4 character cells square. It is independent of the pattern plane and is moved by updating the X & Y coordinates allocated to it. It may be switched on or off at will, and additionally, any sprite to sprite collision is automatically detected. Each sprite has a priority and if a non transparent section of a higher priority sprite passes in front of another, it overlays the part covered. Sprites allow advanced effects i.e. three dimensional simulations, to be easily obtained.

Sprites are not available in the 40 column text mode. In all other modes upto 32 sprites may be placed on screen with a maximum of 4 on any screen line. If this number is exceeded, only the four highest priority sprites are displayed on the line. Additionally, the 'fifth sprite' flag is set and the fifth sprite number is placed in the VDP's status register.

Each sprite may only take one colour and its priority may not be altered. From BASIC two commands are used for sprite control. The first, SPRITE$(n), is used to define a bank of sprite patterns and the second, PUT SPRITE, sets the position, colour and pattern number of each sprite displayed.

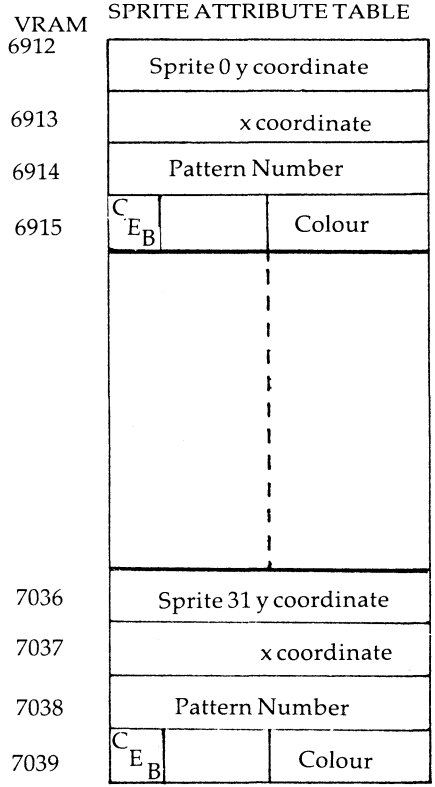
These commands update the two tables in video RAM that contain all the sprite display data. The sprite pattern table, which starts from VRAM location 14336 - irrespective of display mode - contains the shape data for all sprites. The sprite attribute table holds the X and Y coordinates, colour and pattern of each sprite.

Sprites may be either 8*8 pixels or 16*16 pixels in size. In addition they may be displayed at twice normal size - thus the largest sprite available is 32*32 pixels. Sprites are positioned relative to the top left hand corner of the display and may be moved one pixel at a time.

The sprite pattern table will accomodate upto 256 blocks of eight bytes. If sprites are to be 8*8 pixels (8 and 16 pixel sprites cannot be mixed) then 256 seperate sprite patterns may be defined - pattern 0 occupies locations 14336-14343, pattern 1 runs from 14344-14351 and so on. If 16 pixel sprites are selected, each is defined using four consecutive blocks of eight bytes. Each block defines a quarter section of the sprite, in the following sequence -

1 3

2 4



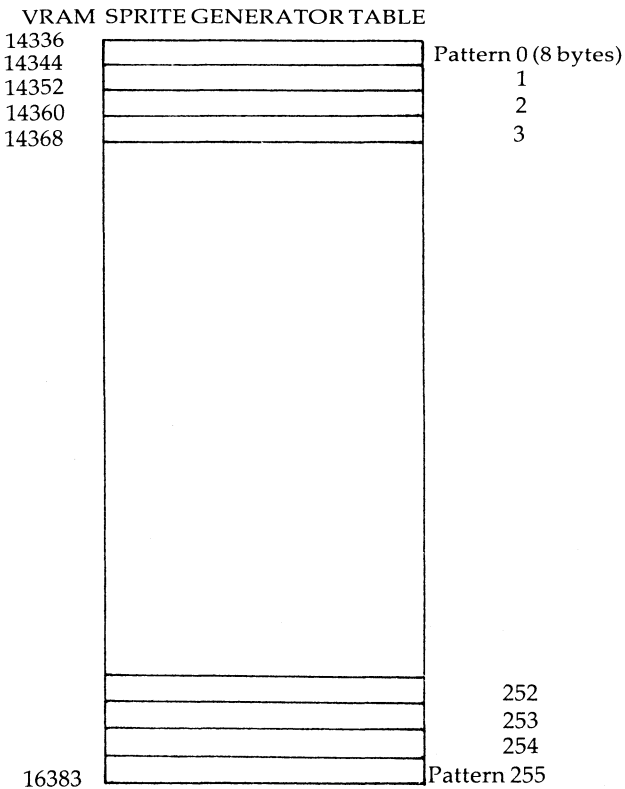


FIGURE 1.6 SPRITE TABLES

After setting up the sprites shape, it is placed on screen by an entry in the sprite attribute table. In all display modes with a sprite capability, the table begins at the same location in VRAM : 6912 and has a maximum length of 32 sprites * 4 bytes = 128 bytes. Each entry specifies the position, colour and pattern number for a single sprite. The first two bytes give the Y and X coordinates and the third identifies the shape by selecting a block or blocks in the sprite pattern table. The final byte contains in the low order nibble, the sprites foreground colour. In addition the most significant bit - the 'clock early bit' - will, if set, shift the sprite position 32 pixels left.

Sprite to sprite collisions are handled in BASIC using the SPRITE ON and ON SPRITE statements.

Figure 1.2 shows the combined display planes produced by the VDP.

General Instruments AY-3-8910 PSG (Programmable Sound Generator)

This is a register orientated programmable sound generator providing three voices over eight octaves. In addition joystick input is handled via two independent, on-board, I/O ports.

Each voice has separate volume control and may produce tone and/or noise. However, the same noise frequency and envelope shape / cycle is used for all channels - if two channels produce noise, then they produce the SAME noise; only the amplitude may differ.

The PSG has 16 internal read/write registers, two of which function as data storage registers for the two joystick ports. The remaining registers may be divided into 5 groups on the basis of function:

1. R0 - R5 : Channel frequency control.

2. R6 : Noise generator frequency.

3. R7 : Channel tone and/or noise select register. The two most significant bits set the data transfer direction of the two I/O ports.

4. R10 - R12 (Octal): Envelope controlled or fixed tone select registers.

5. R13 - R15 (Octal): Envelope pattern and period select registers.

The two input/output ports are completely independent of the PSG's sound generation functions.

MSX BASIC includes a comprehensive array of music commands for which most parameters are preset. In combination with a timed interrupt capability, this permits complex sound effects to be easily realised.

CHAPTER 2

MSX BASIC

Outline. Variables & Functions. Graphics commands. Sound. Program storage.

MSX BASIC is Microsoft standard BASIC 4.5 with several powerful extensions primarily in the graphics and sound areas. Principle omissions include Procedures, the `While / Wend` or `Repeat / Until` forms and full keyword abbreviation.

A full screen editor is featured which allows any line on screen to be simply amended. Line re-entry occurs if the RETURN key is depressed with the cursor over any character in the line. No error checking occurs on entry.

A line may be upto 255 characters in length and may contain multiple statements. Multistatement lines use the colon as a statement separator. A REM statement forces execution to continue from the following line. Line numbers must lie in the range 0-65529 inclusive.

Spaces are not necessary and are ignored by the interpreter (except as part of a string). A consequence of this is that keywords may not be embedded in variable names (of which only the first two characters are recognised). Upper and lower case letters are distinguished. Keywords are accepted in either upper or lower case.

Apart from the main keyboard there are two groups of keys. On the right is the editing cluster - the arrowed cursor keys and insert/delete options. On the left is a set of five function keys. On power up the function keys are preset to ten functions. The unshifted set is displayed on the bottom line of the screen - this key display may be switched off with the functions maintained using the command KEY OFF.

To re-define the function of any key, the KEYx,"string" instruction is used. For example to make key 1 run a program, the following could be used:

```
KEY1,"RUN"+CHR$(13)
```

A program listing or run may be paused by pressing the STOP key once. A second push restarts the option. To terminate either, the stop key is pressed with the control key held down.

For those unfamiliar with Microsoft BASIC, several features are worth highlighting:

1. Debugging is simplified by the use of the `TRON` and `TROFF` statements. If the `TRON` statement is made within a program or directly, then as each line is executed the line number is printed. This usually overwrites any display.

2. Blocks of lines may be deleted using the `DELETEx,y` command and the program renumbered with the `RENUMx,y,I` command. This is also useful for checking for branches to non existent lines. In such a case, at run time MSX BASIC will not flag an error. Instead a branch is made to the first line with a higher number than the one specified.

3. All variables beginning with a particular character or set of characters may be declared to be a particular type : integer:`DEFINT`, single precision:`DEFSNG`, double precision:`DEFDBL` or string:`DEFSTR`.

4. Arrays may be deleted using the `ERASE` command.

5. On power up, the space allocated for string storage is 200 bytes. Further allocation requires use of the `CLEAR` command. For example to reserve 1000 bytes use `CLEAR 1000`

6. The function `FRE(0)` will return the amount of remaining memory available for program storage.

7. The string function `MID$` may be used to replace a character or characters in an expression.

8. The `GOSUB / RETURN` structure allows a return address to be specified: `RETURN linenumber`

Variables and Functions

Variable names can be of any length but must start with a letter. Only the first two characters are significant. Lower case letters are distinguished from upper case. Variables do not need to be initialized to a value - a value of zero is assumed if non has been declared. Variables do not need to be dimensioned unless the number of elements exceeds eleven i.e. S(0) to S(10) would not need dimensioning.

A variable which is not declared to be of a particular type is assumed to be a double precision real number and is stored with 14 digits of precision. Variables declared as single precision are stored with upto 6 digits of precision. Integers must fall in the range -32768 to 32767.

A single precision variable is declared by an affix of a shreik (!) character i.e. SP! and an integer type with the percent sign (%). If it is necessary to declare a variable as double precision, the hash (#) symbol is used.

As mentioned above that variants of the DEF statement may be used to globally define variable types. If the statement DEFINT A was used then all variables beginning with 'A' would be handled as integers. It is also worth noting that the variable 'A' would not be distinguished from 'A%'.

Dimensioning variables is straightforward, for example `DIMX(20,40),Y(80,160)`. However there is a limit of 255 dimensions, although the number of elements is limited only by memory requirements.

As any variable need not be dimensioned unless more than 11 elements are to be used, economical use of memory is made if those variables with less than 11 elements are dimensioned.

MSX BASIC includes a command `SWAPx,y` which is self explanatory. However, the variables must be of the same type. Another related command is `VARPTRx` which returns the element storage location of the variable specified - if it has been declared.

MSX system variables are:

1. `TIME` : The system 'clock' - incremented each 50th of a second in the UK.

2. `BASE(n)` : Returns the first location of the table specified in video R.A.M. Independent of display mode.

3. `VDP(n)` : Returns the value in the specified VDP write only register or the status register.

4. `SPRITE$(pattern #)` : Used to define each of the 256 possible 8*8 or 64 possible 16*16 sprite shapes. It is set equal to a string of upto 32 characters. The character code of each character sets the bit pattern for one definition byte.

Functions

All the old favourites are supported:

XYZ is an arbitrary variable here.

`ASC(XYZ$), CHR$(XYZ), LEN(XYZ$), INT(XYZ), EXP(XYZ), ABS(XYZ), LOG(XYZ), HEX$(XYZ), OCT$(XYZ), BIN$(XYZ), SGN(XYZ), RND(XYZ), MID$(XYZ$,N,n), RIGHT$(XYZ$,n), LEFT$(XYZ$,n), STR$(XYZ), VAL(XYZ$), STRING$(n,XYZ$), INSTR(XYZ$,xyz$), ATN(XYZ), COS(XYZ), SIN(XYZ), SQR(XYZ), TAN(X), TAB(n), SPC(n), INKEY$, INPUT$(n), PEEK(XYZ), POKEXYZ, FRE(Ø), FRE"", USRXYZ(n), VARPTR(XYZ), VARPTR(#XYZ), CINT(XYZ), POS(n)` and `LPOS(n)`.

Extensions include :

1. `CDBL(XYZ)` and `CSNG(XYZ)` which convert XYZ to double precision and single precision respectively.

2. VPEEK(n) and VPOKEn,x are the video R.A.M. equivalents of the PEEK and POKE commands.

3. STICK(n) and STRIG(n) return the direction or trigger status of a joystick (or cursor keys and space bar).

4. POINT(X,Y) returns the colour of the specified pixel.

5. PLAY(channel) gives the status of one or all of the music queues.

6. EOF(file #) returns -1 if the end of the sequential file has been reached, otherwise 0.

7. PAD(n) : Here n determines which parameter of the touch pad status is returned.

8. PDL(paddle #) returns the value of a paddle.

Graphics commands

The full palette of 15 colours and transparent is available in all modes (see appendix B). The graphics instructions may be divided into three sections:

1. General formatting and colour

2. Sprites

3. High resolution

General purpose commands

The screen mode is set using the SCREEN command :

SCREEN Ø Text mode 40*24
SCREEN 1 Text mode 32*24
SCREEN 2 High resolution graphic mode (HRG)
SCREEN 3 Multicolour mode

This command may also be used to set the sprite size, key click off/on, cassette transfer rate and printer option. When used to select a text mode, the character set is copied from ROM to VRAM. The command should not be used if the user needs to access any redefined characters.

Text modes

The width of either the 40 or the 32 column text screens may be set to any value from one column upwards using WIDTH i.e. WIDTH2Ø. It is noted that in both text modes, the two lefthand and the rightmost columns are unused.

The majority of television receivers do not display the lefthand column.

In either mode, characters may be placed in a reserved column by poking the name table in video RAM. For example, to place a capital A in the top row of the two left hand columns of the 40 column text screen, `VPOKE` locations 0 and 1 with 65.

The global colours for any mode are set with the `COLOR` command. The format is:

```
COLOR foreground,background,border
```

The default setting is 15,4,4. In the 40 column text mode the border colour may not be specified seperately and takes the background colour. On screens 1,2 or 3, a change to black background and border with gray ink could be achieved using : `COLOR14,1,1`

Note that in the graphics modes, the new background colour is taken only after a `CLS` instruction.

In the 40 column text mode, the VDP does not allow any other colours to be used. No sprite capability is supported. In the 32 column text mode, the VDP will display text in colours other than those set by the `COLOR` instruction. However, there are no specific BASIC commands for this purpose. Thus for more than one foreground or background colour to be on screen requires changes be made to the colour table in V.R.A.M. This is achieved using the `VPOKE` command. A similar operation must be performed if it is necessary to re-define characters.

In both text modes, the cursor position is set by the `LOCATE X,Y` instruction. The 40 column text mode permits upto 37 characters to be printed on each line. Although column positions upto 255 are accepted, only those in the inclusive range 0 to 36 are active. For the 32 column text mode, the active column range is 0 to 28.

An alternative to LOCATE0,0 is to print CHR$(11). The locate command also controls the cursor display :

`LOCATE2,4,0` Switches display off

`LOCATE2,4,1` Enables the cursor display

The `PRINT` command, which may be entered as ?, uses standard formatting characters:

1. The semicolon ';' causes the omission of the return character - `CHR$(13)` - after the final print item specified. Thus the print position is not moved to the first column of a new line.

2. A comma moves the print position to the start of the next tabulation zone. Each tabulation zone is 14 columns wide.

3. The '+' sign is used to concatenate strings. For example : `A$=B$ + "XYZ"`

`TAB(n)` and `SPC(n)` follow the `PRINT` instruction i.e.
`PRINTSPC(4);"YET`
Notice that the final delimiter (") may be omitted.

The PRINT USING variant allows tables of strings or numerics to be printed in a specified format. The four principle control characters are :

1. The shriek (!) : This causes only the first character of a string to be printed i.e.

`PRINTUSING"!";"NEWORD"` would give 'N'

2. The back-slash (\): This is used in the form "\ \" to specify the number of characters of the string that are to be printed. The number of characters printed is two plus the number of spaces between the two signs, i.e.

`PRINTUSING"\ \";"NEWORD"` would give 'NEW'

If the field length exceeds the string length then spaces are added.

3. The ampersand (&) sign : This inserts a specified string in place of an & sign in another string i.e.

`Q$="NEWORD":?USING"ON &";Q$` would yield: ON NEWORD

Note that if more than one sub-string is given, the sequence is repeated for each. For example

`Q$="W":Z$="R":?USING"ON &";Q$,Z$` will print: ON WON R

4. The HASH (#) sign : Allows numerics to be printed with a specified number of digits before and after the decimal point. Additional 0's are inserted if the value is of insufficient length. For example:

`?USING"###.##";2,4.684` would give 2.00 4.68

If the field is too narrow for the data, either a % sign is printed infront of the value or it is rounded. For example:

`?USING"##";224444` gives %224444
`?USING"##.##";22.1234` gives 22.12

If the field is too large for the data, either the value is right justified or zeros are added. For example:

`?USING"##.##";22.2` gives 22.20
`?USING"####";22` gives 22

The numeric must not be more than 24 digits in length otherwise an error is flagged.

Additional numeric control characters used in conjunction with the '#' sign are: **, ££, coma, +, - and ^^^^ (carat).

A plus sign to the left or the right of the hash symbols will cause the sign of the number to be printed in front of or to the right of the value, respectively.

Similarly, a negative sign after the final hash sign will cause a negative sign to be printed after the value if it has a negative argument.

`?USING"##-";-2` will print 2–

The double asterisk ** combination is used on the left of the hash symbols and fills the leading spaces with askerisks i.e.

`?USING"**#.#";4.2` gives **4.2
`?USING"**##.#";4.2` gives ***4.2

The double pound sign prints a single pound sign in front of the value. This control character cannot be used in conjunction with the carat character.

If a comma is placed to the left of the decimal point, the number is printed with a comma to the left of every third digit i.e.

`?USING"####,.#";2222.2` gives 2,222.2

The quadruplet of carats is used to print the value in the exponential form, in order to allow room for the E+XX i.e.

`?USING"##.#^^^^";22.4` yields 2.2E+01

The screen may be cleared in all display modes with the `CLS` command. In both text modes, an alternative is to print `CHR$(12)`. The cursor's Y coordinate is returned by `CSRLIN` and the X coordinate by `POS(x)` (where x is a dummy argument and any value may be used).

Video RAM Manipulation

To obtain characters in colours other than those set globally by the COLOR command in the 32 column text mode, the appropriate byte in the colour table in VRAM is poked with the new foreground / background combination.

As mentioned, each of the 32 bytes in the colour table determines the colours of a set of 8 of the 256 available characters. As a result, to print text in other than the default colours, it is necessary to copy part of the character set to another section of the pattern generator table.

Pattern Generator table 0 - 2047
Colour table 8192 - 8223
Sprite attribute table 6912 - 7039
Name table 6144 - 6911
Sprite pattern table 14336 - 16383

Table 2.1 VRAM layout in the 32 column text mode

Note that the graphics characters occupy the first 32 definition blocks in the pattern table. This is because the first 32 character codes are non printing. Graphics characters may be placed on screen by printing `CHR$(1) + CHR$(65)` to `(95)`. Another non-standard character has the code 255. This character prints as the inverse of the character covered by the cursor. It is updated each 1/50th of a second.

Example: Multicoloured text

To have the option of upper case text with yellow foreground and black background colours, the following stages are necessary:

Copy the 26 character definitions to another position in the pattern generator table. The character set is defined using 8 bytes for each character. Characters 65 – 90 follow the sequence given in appendix B.

In the 32 column text mode the pattern table extends from VRAM location 0 to 2048. As each character requires eight definition bytes, the first byte to be copied is at location :

```
ASC("A")*8 = 520
```

and the final character definition byte is at`ASC("Z")*8 + 7 = 727`

The 'new' upper case characters will overwrite part of the existing character set. We will replace characters 145 – 170 which are defined from 145*8 = 1160 to 170*8+7 = 1367, as follows:

```
10 SCREEN 1:FOR X=0 TO 207:VPOKE 1160+X,VPEEK(520+X):NEXT
```

The four bytes in the colour table that determine the colours taken by the 32 characters 144–176 are resident in locations 8210–3. The value held by the most significant four bits in each sets the foreground colour, with the background colour determined by the least significant four bits.

FOREGROUND	BACKGROUND	
Yellow : 10	Black : 1	= 10*16 + 1 = 161

Table 2.2 Colour table entry

So the next line of the routine (see Table 2.2) is :

```
20 FOR X=0 TO 3:VPOKE X+8210,161:NEXT
```

If `CHR$(145)` is now printed, a yellow 'A' on a black background should result. Note that if the `COLOR` command is used, all 32 bytes in the colour table are filled with the specified colours. To obtain the alternative colours again, line 20 must be repeated.

The characters are left justified, so when printed with a background colour that contrasts with the screen colour the left hand character may be found to be indistinct.

This method of determining the screen colours, although cumbersome in some ways, allows many effective displays to be achieved quite simply. For instance, a display consisting predominantly of 4-8 characters can be 'animated' or flashed by manipulation of just two colour table bytes.

The following program allows the entire character set to be redesigned and saved or loaded from tape. The cursor keys are used to move about the matrix with the space bar 'flipping' the setting of the respective pixel.

CHARACTER SET EXAMPLE PROGRAM

```
10 '************************************
20 '****  CHARACTER SET UTILITY   ****
30 '****  WITH SAVE, LOAD AND     ****
40 '****  COPY FACILITIES.        ****
50 '************************************
60 '
70 'USE THE CURSOR KEYS TO MOVE ABOUT
    THE DEFINITION MATRIX.
80 'TO TOGGLE THE TARGET BIT, HIT THE
    SPACE BAR.
90 'NOTE THAT THE SCREEN COMMAND
    RESETS VIDEO RAM.
100 '
110 FOR X=0TO12:READA$,B$:POKE38000!+
X,VAL("&H"+A$):POKE38200!+X,VAL("&H"+
```

```
B$):NEXT
120 DEFUSR=38000!:DEFUSR2=38200!
130 DATA 21,21,00,40,00,9C,11,11,40,0
0,9C,00,01,01,00,00,08,08,CD,CD,59,5C
,00,00,C9,C9
140 ON KEY GOSUB 240,270,290,430:KEY(
1) ON:KEY(2) ON:KEY(3) ON:KEY(4) ON
150 ON STRIG GOSUB 400:STRIG(0) ON
160 KEY OFF:COLOR 14,1,1:SCREEN 1,0:C
LS
170 LOCATE2,1:PRINT"F1..SAVE  F3..CHA
NGE CHR$":LOCATE2,3:PRINT"F2..LOAD  F
4..COPY CHR$"
180 FORX=0TO7:VPOKE14336+X,VPEEK(224+
X):NEXT:PUT SPRITE 0,(144,47),8,0
190 FORX=373TO375:VPOKEX,0:NEXT:VPOKE
371,24:VPOKE372,24
200 A$=STRING$(8,46):FORX=0TO7:LOCATE
16,X+6:PRINTA$:NEXT
210 GOSUB 290:X1=1:Y1=1
220 X2=STICK(0):IF X2=0 THEN 220
230 ON X2 GOTO 350,220,360,220,370,22
0,380
240 Z=USR(2):LOCATE2,20:PRINT"RECORD
THEN RETURN"
250 A$=INPUT$(1):IF ASC(A$)<>13 THEN
250 ELSE BSAVE"CAS:",40000!,42047!
260 LOCATE 2,20:PRINTSTRING$(28,32):R
ETURN
270 LOCATE 2,20:PRINT"LOADING..":BLOA
D"CAS:":Z=USR2(2)
280 LOCATE 2,20:PRINTSTRING$(28,32):R
ETURN
290 LOCATE0,20:PRINT"ENTER # OF CHR$
<RET>":GOSUB 470:CN=XX
300 FOR X=0TO7:A$(X)=BIN$(VPEEK(CN*8+
X)):NEXT
310 FORX=0TO7:IF LEN(A$(X))<8 THEN A$
(X)=STRING$(8-LEN(A$(X)),79)+A$(X)
320 FOR BT=1 TO 8:LOCATE15+BT,X+6:IF
MID$(A$(X),BT,1)="1"THEN PRINTCHR$(21
```

```
9) ELSE PRINTCHR$(46)
330 NEXT:NEXT:LOCATE 2,11:PRINT"CHARA
CTER";CN:LOCATE 7,13:PRINT CHR$(CN)
340 RETURN
350 IF Y1=1 THEN 220 ELSE Y1=Y1-1:GOT
0390
360 IF X1=8 THEN 220 ELSE X1=X1+1:GOT
0390
370 IF Y1=8 THEN 220 ELSE Y1=Y1+1:GOT
0 390
380 IF X1=1 THEN 220 ELSE X1=X1-1:GOT
0 390
390 PUT SPRITE 0,(136+X1*8,39+Y1*8),8
,0:FORX=1TO80:NEXT:GOTO220
400 LOCATE 15+X1,5+Y1:IF MID$(A$(Y1-1
),X1,1)="1" THEN MID$(A$(Y1-1),X1,1)=
"0":PRINTCHR$(46) ELSE MID$(A$(Y1-1),
X1,1)="1":PRINTCHR$(219)
410 NN=CN*8+Y1-1:IF MID$(A$(Y1-1),X1,
1)="1" THEN VPOKENN,VPEEK(NN)OR(2^(8-
X1)) ELSE VPOKENN,VPEEK(NN)AND(255-(2
^(8-X1)))
420 RETURN
430 LOCATE0,20:PRINT"CHR$ TO COPY <RE
T>":GOSUB 470:C1=XX
440 LOCATE0,20:PRINT"CHR$ TO BE REPLA
CED":GOSUB 470:C2=XX
450 FORX=0TO7:VPOKEC2*8+X,VPEEK(C1*8+
X):NEXT
460 LOCATE0,20:PRINTSTRING$(28,32):RE
TURN
470 C1$=""
480 X$=INKEY$:IF X$="" THEN480
490 IF ASC(X$)<32 AND ASC(X$)>27 THEN
 480
500 IF ASC(X$)<>13 THEN C1$=C1$+X$:GO
TO 480ELSE XX=VAL(C1$)
510 IF XX>255 OR XX<0 THEN 470
520 LOCATE0,20:PRINTSTRING$(28,32):RE
TURN
```

Sprites

The BASIC statements and variables associated with sprite manipulation are:

1. `SPRITE$(Pattern number)=XYZ$`

2. `PUT SPRITE n,(X,Y),colour,pattern number`

3. `PUT SPRITE n,STEP(x,y),colour,pattern number`

4. `SCREEN Mode,Sprite size`

5. `SPRITE ON/OFF/STOP`

6. `ON SPRITE GOSUB`

Sprites can be 8 by 8 or 16 by 16 pixels in size. Additionally, either size may be used magnified by two. The `SCREEN` statement sets the size used :

0 = 8 * 8
1 = 8 * 8 Magnified
2 = 16 * 16
3 = 16 * 16 Magnified

i.e. `SCREEN 2,2` would enter the HRG mode with all sprites being 16 * 16 pixels.

An 8*8 pixel sprite is defined in the same way as a character using 8 bytes. The 16 * 16 sprite pattern is defined using four blocks of eight bytes:

```
1        17
|        |
8        24

25       9
|        |
32       16
```

These are defined using the SPRITE$ statement. Each sprite pattern is entered as a string of either 8 or 32 characters, each character number representing one byte of the pattern.

For instance to define an 8 by 8 sprite as a '–' sign use :

```
A$=CHR$(Ø):B$=CHR$(126)

SPRITE$(Ø)=A$+A$+A$+B$+B$+A$+A$+A$
```

Here the value of 126 is obtained as the value of the bits set :

128	64	32	16	8	4	2	1
	1	1	1	1	1	1	

An alternative to using the SPRITE$ statement would be to VPOKE the first eight bytes of the sprite pattern table :

```
FORX=14336 TO 14343:VPOKEX,Ø:NEXT:VPOKE14339,126:VPOKE14340,126
```

If 8 by 8 sprites are in use, then 256 sprite patterns may be defined, otherwise with 16 by 16 sprites only 256/4=64 patterns are permitted.

The ability to define sprites of such small size facilitates the use of sprite based character sets. Used in tandem with the magnification option, double sized multicoloured text is easily generated. Although there is a limitation of four letters on any screen line, this approach avoids the problems associated with the left justification of the resident character set.

A sprite may be placed on screen or moved using one of the two forms of the PUT SPRITE instruction. The screen is 'numbered' from the top left hand corner which has coordinates 0,0, to the bottom left hand corner which has coordinates 0,190. The top right hand corner has coordinates 255,0.

Any coordinate given specifies the top left hand pixel of the sprite. In order that a sprite may be brought slowly into view both the vertical and horizontal coordinates range from -32. Thus sprites may be positioned "off screen".

Coordinates specified outside these parameters within the range –32768 - 32767 will not produce an error. Instead the sprite will take a position at coordinate MOD 256.

The one exception to this is when a Y coordinate of 208 is given. After the VDP has encountered this, the sprite in question together with all lower prioity sprites, are removed from the display. This provides an efficient method of flashing sprites. The standard form of the PUT SPRITE statement is :

```
PUT SPRITE plane,(X,Y),colour,pattern number
```

Plane 0 is the highest priority plane and consequently any non- transparent section of a sprite in that plane will cover any part of the display it overlays. The lowest priority plane is 31. Only one sprite is permitted on any particular plane.

The same palette of colours that is used with the COLOR command is available. Note that only one colour may be defined for each sprite. If a colour is not specified, the current foreground colour is used. If the sprite coordinates are omitted, the sprite is displayed at location 209,0. The exception to this is if a sprite had previously been displayed on that plane. In which case the previous coordinates are used.

For example: `PUT SPRITE0,(40,40),10,0:PUT SPRITE0,,10,2` will update sprite pattern 0 with sprite pattern 2 on plane 0.

The alternative form of the `PUT SPRITE` statement :

```
PUT SPRITE Plane,STEP(x,y),colour,pattern number
```

allows the sprite coordinates to be updated relative to it's current position.

Sprites have priority over the character plane. Any non- transparent section of a sprite will occlude the underlying display. As with the limitation of one colour to each sprite, the priority limitation is intrinsic to the VDP.

The `PUT SPRITE` statement updates a four byte entry in the sprite attribute table in VRAM. In all modes which support sprites, this starts at location 6912 and is 32*4 = 128 bytes in length. The four bytes are allocated:

Byte 1 Y coordinate
2 X coordinate
3 Pattern number
4 Clock early bit / Colour

In BASIC, the Y coordinate may fall in the range –32 to 209. However, as an unsigned byte may only contain values in the range 0-255, the table uses the two's complement notation for negative coordinates (for an explanation of two's complement see the section on Z-80 assembly language).

This is not the case with the X coordinate due to the 'clock early' bit. If this is set, the sprite position is shifted left 32 pixels.

For example, to position a blue sprite on plane 0 with pattern 1 at coordinates 100,100 the following could be used as an alternative to the `PUT SPRITE` command:

```
VPOKE 6912,100:VPOKE 6913,100

VPOKE 6914,1:VPOKE 6915,4
```

To move the sprite left 32 positions either:

```
VPOKE 6913,68
```

or: `VPOKE 6915,132` could be used.

All the sprites may be removed from the display, by the following:

```
VPOKE 6912,208
```

Our next example copies the upper case character set into the sprite pattern

area to allow non-standard text:

```
10 SCREEN1,1:REM 8*8 expanded sprites
20 FOR X=0 TO 207
30 VPOKE 14856+X,VPEEK(520+X)
40 NEXT
```

Pattern numbers 65 onwards have been deliberately used to allow a direct correspondence between the character code and the pattern number. This allows strings to be 'printed' easily:

```
10 A$="MSX":FOR X=1 TO 3
20 PUT SPRITE X,(18*X,40),11,ASC(MID$(A$,X,1))
30 NEXT
```

The two remaining sprite commands:

1. ON SPRITE GOSUB

2. SPRITE ON

enable collision detection and handling. A sprite-sprite collision occurs when two non-transparent sprite sections overlap.

There are two limitations on the information available on any collision:

1 Only sprite-sprite and not sprite-background collisions are detected.

2 The planes of the sprites involved are not returned by either a BASIC function or VDP register.

The SPRITE ON statement activates the checking procedure which then occurs after each BASIC statement is executed. No checks occur in the direct mode. When a collision occurs, a GOSUB is performed to the routine specified by the ON SPRITE GOSUB statement.

The SPRITE OFF statement cancels collision checking. The SPRITE STOP statement suspends the actual call to the subroutine but maintains the collision checks. If a collision occurs, it is 'remembered' until a SPRITE ON statement is executed when the subroutine call is made.

A SPRITE STOP statement is automatically executed on entry to the subroutine. This is subsequently cleared by a SPRITE ON statement when the RETURN command is encountered - unless the handling routine contains a SPRITE OFF statement. It is important to note that the CLEAR instruction performs the SPRITE OFF statement.

Often, the movement of sprites is of a fixed, repetitive nature. This only requires the position be incremented/decremented at fixed intervals. MSX BASIC incorporates a powerful set of commands which cause program execution to branch to a subroutine at a regular interval. The time period is specified in units of 1/50th of a second (i.e. the VDP system interrupt frequency).

The format is similar to the sprite collision handling commands:

1. `INTERVAL ON`

2. `ON INTERVAL=X GOSUB`

Again, there are the variants INTERVAL OFF and INTERVAL STOP, the latter of which is executed on entry to the handler routine.

A slight variation in the period may become noticeable with short intervals, this is due to the check occuring after the execution of each BASIC instruction.

Two sets of commands which are similar to these are:

1. `ON STOP GOSUB` and `STOP ON/OFF/STOP`

2. `ON KEY GOSUB line x,line y ....and KEY(x) ON/OFF/STOP`

The `ON STOP GOSUB / STOP ON` structure must be used with caution as it can negate any return from the program mode to the direct entry mode. For example:

```
10 ON STOP GOSUB 40:STOP ON
20 PRINT"NOSTOP":GOTO 20
40 RETURN
```

cannot be aborted. If it is intended that a program run is not to be interrupted then the statements in line ten should appear after any `CLEAR` statement.

SPRITE DESIGNER EXAMPLE PROGRAM

A useful program is listed below. It enables you to design sprites of your own choosing.

```
10 '************************************
20 '****      SPRITE DESIGNER       ****
30 '************************************
40 '
50 ' USE THE CURSOR KEYS TO MOVE
     AROUND THE DEFINITION MATRIX.
60 '
70 ' SPACE BAR TO ALTER THE BIT TO
     THE LEFT OF THE CURSOR.
     CTRL/STOP TO FINISH AND PRINT
     OUT BYTE VALUES.
80 '
90 KEYOFF:FORX=0TO32:VPOKEX+14336,0:N
EXT
```

```
100 ON STOP GOSUB300:STOP ON
110 SCREEN1,2:COLOR14,1,1:CLS
120 A$=STRING$(16,CHR$(250)):FORX=4TO
19:LOCATE11,X:PRINTA$:NEXT
130 X=12:Y=4:LOCATEX,Y,1:PUT SPRITE0,
(20,100),14,0
140 Q=0:ON STRIG GOSUB 250:STRIG(0) O
N
150 FORL=1 TO 60:NEXT:D=STICK(0):ON D
 GOTO 170,150,190,150,210,150,230
160 GOTO 150
170 IF Y<>4 THEN Y=Y-1:LOCATEX,Y,1 EL
SE 150
180 IF Y<>11 THEN 150 ELSE Q=Q-1:GOTO
 150
190 IF X=27 THEN 150 ELSE X=X+1:LOCAT
EX,Y,1
200 IF X<>20 THEN 150 ELSE Q=Q+2:GOTO
 150
210 IF Y=19 THEN 150 ELSE Y=Y+1:LOCAT
EX,Y,1
220 IF Y<>12 THEN 150 ELSE Q=Q+1:GOTO
 150
230 IF X=12 THEN 150 ELSE X=X-1:LOCAT
EX,Y,1
240 IF X<>19 THEN 150 ELSE Q=Q-2:GOTO
 150
250 IF VPEEK(6145+32*Y+X)=250 THEN VP
OKE 6145+32*Y+X,219:AA=2 ELSE VPOKE 6
145+32*Y+X,250:AA=4
260 BY=14336+Q*8:IF Q=0 OR Q=2 THEN B
Y=BY+Y-4 ELSE BY=BY+Y-12
270 IF Q=0 OR Q=1 THEN BI=19-X ELSE B
I=27-X
280 VL=2^BI:IF AA=2 THEN VPOKEBY,(VPE
EK(BY) OR VL) ELSE VPOKEBY,(VPEEK(BY)
 AND (255-VL))
290 RETURN
300 CLS:FORX=0TO15:LOCATE2,X+2:PRINTV
PEEK(14336+X):LOCATE8,X+2:PRINTHEX$(V
PEEK(14336+X)):LOCATE12,X+2:PRINTVPEE
```

```
K(14352+X):LOCATE18,X+2:PRINTHEX$(VPE
EK(14352+X)):NEXT
```

High Resolution Graphics

MSX BASIC has seven instructions which may only be used in the two graphics modes:

1) CIRCLE 2) DRAW 3) LINE 4) PAINT

5) PSET 6) PRESET 7) POINT

However, these instructions produce rather clumsy results in the multicolour mode which has a maximum resolution of 4*4 pixel blocks.

It is not possible to print characters to a graphics screen in the normal manner. Instead, characters must be output to the display as file data:

```
1. OPEN"GRP:" FOR OUTPUT AS #1

2. PRESET(40,40)

3. PRINT#1,"Graphics text"
```

Note that unless more than one file has been declared with the `MAXFILES` statement, the file number must be one. In the multicolour mode, each letter is approximately 2.5 cm's square.

The workhorse of the graphics instructions is `DRAW` with the format:

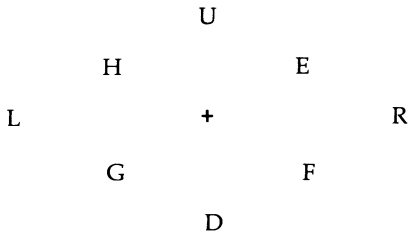
```
DRAW "single letter commands"
```

The 13 commands available constitute what Microsoft call a 'graphics micro language'. They allow lines to be drawn in specified steps either up, down, left, right or diagonally from the current reference point. Any of the 16 colours may be used. The statement is best explained by example:

```
DRAW "D40R40U40L40"
```

This draws a square proceeding down 40 units (D40), right 40 units (R40),

up forty units (U40) and left 40 units (L40). The principle direction commands are:



with `Cx` setting the colour used.

To draw a line to a position specified by absolute coordinates, use `M X,Y`. Alternatively, to draw to a location given relative to the current position, prefix the X or Y value with either a '+' or a '–' sign.

The units of movement are set by the command `Sn` where n is in the range 0-255 (default = 4). This is divided by four to give the number of pixels to each unit.

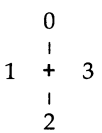
If a command specifies a position off screen but within the range -32768 to 32767 then no error report is produced and the most extreme position possible is taken.

There are two optional variants of each movement command:

1. A prefix of 'B' will cause the graphics cursor to move the specified distance with no line drawn.

2. A prefix of 'N' causes the graphics cursor to return to the initial position on completion of the subcommand.

The direction frame may be rotated anticlockwise through multiples of ninety degrees by the An command. The numeric must lie in the integer range 0-3 as follows:



Thus `DRAW "A3L40"` will cause a line to be drawn right 40 units.

The final sub-command, X; allows a string variable to be included in the command string.

This prefixes the variable, which is followed by a semi-colon; for example:

```
ST$="M-20,+40":DRAW"XST$;"
```

This command is unnecessary if no delimiters are used: `DRAW ST$`. A numeric variable may also be used with a command string. Here the variable is prefixed with an equate sign and followed by a semi-colon: `DRAW "U=Y;"`.

The line command format is:

```
LINE(X1,Y1)-(X2,Y2),colour
```

which draws a line between the two coordinates. Each coordinate may, alternatively, be specified using the relative STEP variant.

If an additional ',B' is appended to the command, a rectangle is produced or if ',BF' is added, a rectangle is produced and filled.

The circle command allows quite advanced diagrams to be produced simply. Any ellipse specified may be produced in its entirety or a 'slice' drawn. The 'base' form of the command is:

```
CIRCLE(X,Y),radius
```

with the options:

```
,colour,start angle,end angle,aspect ratio
```

The coordinates X and Y specify the circle's centre with the STEP(X,Y) form allowing the point to be defined relative to the reference position. If used, the start and end angles take arguments between 0 and 2*PI. A negative value will be accepted but additional lines are drawn between the centre of the ellipse and the start and end points. The aspect ratio is the ratio of the vertical and horizontal radii.

Again, the `PAINT` command is almost self-explanatory, and has the form:

```
PAINT (X,Y) or PAINT (X,Y),colour
```

It will fill any enclosed figure from the coordinate specified. If no colour is specified, the current foreground colour is used. Note that in the HRG mode the colour of both the borderline and the paint must be the same, otherwise the whole screen is recoloured. In the multicolour mode the form:

```
PAINT(X,Y),Fill colour, borderline colour
```

may be used.

Example: *To produce a solid red circle in the HRG mode use:*

```
SCREEN2:COLOR8,1,1:CIRCLE(80,80),40:paint(80,80)
```

The two remaining commands:

```
PSET(X,Y),colour
```

```
PRESET(X,Y),colour
```

are identical in that both set the pixel specified to the given colour. The colour specification may be omitted, in which case `PSET` defaults to the foreground colour and `PRESET` to the background colour.

SKETCH PAD EXAMPLE PROGRAM

The following program illustrates the use of High Resolution Graphics

```
10 '**********************************
20 '****        SKETCH-PAD          ****
30 '**********************************
40 '
50 ' JOYSTICK IN PORT 1. DRAW BY
     PRESSING THE FIRE BUTTON.
60 '
70  'F1 TO CHANGE COLOUR; FOLLOW WITH
     COLOUR NUMBER - TWO CHARACTERS
80  'F2 TO DRAW A CIRCLE; FOLLOW WITH
     THE RADIUS - TWO CHARACTERS
90  'F3 TO FILL A SHAPE; IT MUST BE
     ENCLOSED & BE OF THE SAME COLOUR
     AS THE CURRENT INK
100 'F4 TO SAVE THE PICTURE;
     2 MINUTES TO SAVE.
110 'F5 TO LOAD A SCREEN FROM TAPE;
     PRESS PLAY AFTER F5
     TWO MINUTES TO LOAD.
120 '
130 FOR X=38000!TO38024!:READA$:A=VAL
("&H"+A$):POKEX,A:NEXT:DEFUSR=38000!
140 DATA 21,00,00,11,40,9C,01,00,18,C
D,59,00,21,00,20,11,40,B4,01,00,18,CD
,59,00,C9
150 FOR X=38200!TO38224!:READA$:A=VAL
("&H"+A$):POKEX,A:NEXT:DEFUSR2=38200!
160 DATA 21,40,9C,11,00,00,01,00,18,C
```

```
D,5C,00,21,40,B4,11,00,20,01,00,18,CD
,5C,00,C9
170 ON KEY GOSUB 350,360,370,380,410:
KEY(1) ON:KEY(2) ON:KEY(3) ON:KEY(4)
ON:KEY(5) ON
180 ON STRIG GOSUB 180,240:STRIG(1) O
N
190 COLOR 10,1,10:SCREEN 2,0:CLS:X=12
5:Y=95
200 FOR Z=14336 TO 14343:READ ZZ:VPOK
EZ,ZZ:NEXT:C=10
210 DATA 0,32,32,248,32,32,0,0,0
220 PUT SPRITE0,(X-2,Y-4),10,0:IF A=1
 THEN RETURN ELSE A=1
230 GOTO 230
240 DN=STICK(1):IF DN=0 THEN 240
250 ON DN GOTO 260,270,280,290,300,31
0,320,330
260 IF Y=0 THEN 240 ELSE Y=Y-1:GOTO 3
40
270 IF X=255 OR Y=0 THEN 240 ELSE X=X
+1:Y=Y-1:GOTO 340
280 IF X=255 THEN 240 ELSE X=X+1:GOTO
 340
290 IF X=255 OR Y=191 THEN 240 ELSE X
=X+1:Y=Y+1:GOTO 340
300 IF Y=191 THEN 240 ELSE Y=Y+1:GOTO
 340
310 IF X=0 OR Y=191 THEN 240 ELSE X=X
-1:Y=Y+1:GOTO 340
320 IFX=0 THEN 240 ELSE X=X-1:GOTO 34
0
330 IF Y=0 OR X=0 THEN 240 ELSE Y=Y-1
:X=X-1
340 IF C<>0 THEN PSET(X,Y),C:GOTO 220
 ELSE 220
350 PUT SPRITE 2,(20,20),10,0:PUT SPR
ITE 4,(28,20),10,0:A$=INPUT$(1):PUT S
PRITE 2,(0,0),,0:B$=INPUT$(1):PUT SPR
ITE 4,(0,0),,0:C=VAL(A$+B$):RETURN
360 PUT SPRITE 2,(20,80),10,0:PUT SPR
```

```
ITE 4,(28,80),10,0:A$=INPUT$(1):PUT S
PRITE 2,(0,0),,0:B$=INPUT$(1):PUT SPR
ITE 4,(0,0),,0:R=VAL(A$+B$):CIRCLE(X,
Y),R,C:RETURN
370 PAINT (X,Y),C,C:RETURN
380 Z=USR(2)
390 SCREEN1:CLS:LOCATE2,2:PRINT"PRESS
 PLAY & RECORD THEN      RETURN"
400 A$=INPUT$(1):IF ASC(A$)<>13 THEN
400 ELSE BSAVE"CAS:",40000!,52288!:ST
OP
410 PUT SPRITE 2,(20,20),10,0:BLOAD"C
AS:":Z=USR2(2):PUT SPRITE 2,(0,0),,0:
RETURN
```

Sound

A simple beep may be produced by the BEEP statement or by printing CHR$(7). More complex sounds are generated either using the PLAY statement and macro language or by directly amending the PSG's registers with the SOUND command. The latter has the form:

```
SOUND register #, value
```

For details of the register functions see the section on the AY-3-8910.

The PLAY statement is followed by up to three strings, each of which is a sequence of one letter commands exclusively for one 'voice' (the format is similar to the DRAW command).

For example to play a single note using voice 2, the following would be used:

```
PLAY "","N20"
```

As default values are set for most parameters, it is not necessary to go through a lengthy initialization sequence.

There are two methods of specifying the note played:

1. Nx where x is in the range 0-96. If 0 is used a rest is 'played'.

2. By setting the octave Ox (x must lie in the range 1-8, 4 is default) and note A-G. For example:

PLAY"O6ABCD" will play a four note sequence in octave six.

If this was followed by:

`PLAY"BCD"` the notes will also be taken from octave 6.

Flats and sharps (only those available on piano) are produced by using # or + after the note for a sharp and – for a flat.

The default volume is eight on a scale from 0-15 and is set with the `V`x command.

The length of each note is set using Lx where x is in the inclusive range 1-64. A value of one will produce a whole note, a value of 4, a quarter note and so on. To obtain a note longer than 1 the tempo needs to be reset using the Tx command (default 120). The x may take a value from 32-255 and determines the number of quarter notes played in a minute. For example, to make all notes twice the default length use T60.

If it is only necessary to change the length of a single note, then the note reciprocal without the 'L' can just be placed after the note. For example B#8 or D32.

Note that with all the single letter commands, a variable may be used with the format:

```
PLAY"N=X;L=Y;N=X;"
```

To simplify transcription, Microsoft incorporated the ability to append a note with a period (.) to increase its length by one half. An alternative to using N0 to produce a pause is the Rx command (x default = 4). This produces a rest of the length specified by x which must lie in the range 1-64 and is interpreted in the same way as the parameter in the L command.

The two remaining single letter sub-commands, Mx and Sx, allow more advanced sound effects to be generated by altering the envelope of the sound produced.

This allows the volume to be varied in a preset manner over the duration of the note. Each envelope shape commences with one of two sequences:

1. The volume increases from zero to the maximum. Here, the envelope has an 'attack' value of 4.

2. The volume decreases from the maximum to zero. The envelope has an attack value of 0.

The remainder of the envelope sequence is determined by three other parameters. The required setting for each is added to that of the initial attack stage. The total is used with the S command to select the sound shape required:

1. Continue: If the sound is to terminate after the first cycle, this is set to zero, otherwise it is set to 8.

2. Hold: If set to 1, the attack sequence is continually repeated for each cycle. If set to zero, the volume is held at the setting at the end of the first cycle.

3. Alternate: A setting of 2 flips the volume setting at the end of each cycle. No change occurs if a value of zero is used.

Each of the above parameters has two settings, which allow, in total, 16 variants of envelope shape/cycle operation. However, there is duplication and there are only eight distinct patterns possible. These are given together with the parameter settings on page five of appendix E. The default settings are:

Shape: 1
Modulation: 255

For example, to produce notes which rise to a maximum volume and hold there for their duration, use the following parameter settings:

1. Attack: 4
2. Continue: 8
3. Hold: 1
4. Alternate: 0

Total = 13: `PLAY"S13....`

The duration of the first and subsequent cycles is set by the M command. This will accept integer values in the range 1 to 65535 inclusive.

One final point on the PLAY command is that (like the `DRAW` command) predefined strings may be executed with the X command:

```
A$="N40N20N40":PLAY "XA$;"
```

As a variable is used, it must be followed by a semicolon. The status of one or all the channels can be found with the `PLAY` function:

```
PLAY (x)
```

Here x must be in the range 0 to 3. If channel 1, 2 or 3 is checked, then if the channel is active -1 is returned otherwise zero. If `PLAY(0)` is used, all three statuses are logically ORed and returned.

Program storage

In all MSX computers with at least 32K of RAM, BASIC program storage commences from location 32768 (&H8000). Each program line is stored not as it is entered, but in a condensed form. Keywords are replaced by 'tokens', whilst variable names and symbols are stored directly. In addition, each line is prefaced by two pairs of bytes. The second pair holds the line number and the first the address of the following line. Lines are divided by a null byte with the program end indicated by a further two bytes with a value

of zero.

Just above the program is a variable table. This consists of storage for each non-array variable used in the program. Finally, above the simple variable table is an array table.

For example, if the following program is entered:

```
10 FOR X=32768! TO 40000!
20 PRINTX,PEEK(X)
40 NEXT
```

storage is organised as follows:

Location	*Value*	*Comment*
32768	0	
32769	23	Address of the next
32770	128	line.
32771	10	Line number
32772	0	
32773	130	Token for FOR
32774	32	Space code
32775	88	X
32776	239	Token for =
32777	29	S.P. Constant
32778	69	S.P. Exponent byte
32779	50	
32780	118	S.P. Mantissa
32781	128	
32782	32	Space code
32783	217	Token for TO
32784	32	Space code
32785	29	S.P. Constant
32786	69	S.P. Exponent byte
32787	64	
32788	0	S.P. Mantissa
32789	0	
32790	0	End of line marker
32791	36	Address of the next
32792	128	line.
32793	20	Line number
32794	0	
32795	145	Token for PRINT
32796	88	X
32797	44	comma
32798	255	Token for PEEK
32799	151	
32800	40	(
32801	88	X
32802	41	)
32803	0	End of line marker

32804	42	Address of the net
32805	128	line.
32806	40	Line number
32807	0	
32808	131	Token for NEXT
32809	0	End of line marker
32810	0	End of program
32811	0	marker.

Notice that constants are stored in the relevant format. Here the parameters of the `FOR NEXT` loop are stored in single precision.

The simple variable table is moved whenever program lines are added or deleted. The variables are stored in the order in which they occur in the program. The first byte of each indicates the variable type:

8 : Double precision
4 : Single precision
3 : String
2 : Integer

The following two bytes are the first two character codes of the variable name. The remaining section differs for each variable type:

1. Integer: The actual value is held in reversed, signed two byte binary form.

2. String: Three bytes are used to store string data. The first specifies the number of characters in the string. The remaining byte pair holds the address of the actual string. The address has the low and high order bytes reversed.

3. Single precision: The value is represented by one byte for the exponent and a six digit, three byte binary coded decimal mantissa. The m.s.b. of the mantissa flags the sign of the value: 1 = negative.

4. Double precision: As for a single precision variable, except a seven byte mantissa is used.

Note that the exponent byte, the location of which is returned by the VARPTR function, has &H40 added.

Each entry in the subsequent arary table again starts with three bytes holding the variable type and name. However, prior the the data elements is a header. This has three sections:

1. A two byte leader containing the remaining number of bytes in the array.

2. A single byte which holds the number of dimensions.

3. A sequence of two byte values sprcifying the size of each dimension.

CHAPTER 3

MSX BASIC VOCABULARY

`ABS(Y)`

A function which returns the absolute value of the numeric expression Y.

Example: `ABS(-4)` returns 4

`ASC(Y$)`

Returns the character code of the first character of the string Y$. A graphics symbol returns 1. A null string will cause an error report. For a complete listing of the character codes see appendix A.

Example: `ASC("ABC")` returns 65

`ATN(Y)`

A function which returns the arctangent of the expression Y. The result is in radians in the range –PI/2 to PI/2. If Y is a variable, it may be of any type, however the evaluation is performed in double precision.

Example: `C=ATN(40)` returns 0.67474094222354

`AUTO`

Variants:
```
AUTO
AUTO line number
AUTO ,increment
AUTO line number,increment
```

Omission default: Line number .. 0
Increment .. 10
Increment and line number .. 10

The subsequent line number is printed after each carriage return. Options

allow the initial line number and increment to be specified. The sequence is terminated by either CTRL C or CTRL STOP. An asterisk is printed after any line number which already exists. Pressing return, without any addition, will leave the resident line unaltered.

Example: `AUTO 200,20` will provide the sequence 200, 220, 240 ....

`BASE(Y)`

A special variable which returns the locations in video R.A.M. of the look up tables used by the VDP to produce the display. Values of Y between 0 and 19 return the start position of one table for a particular mode. See appendix C for further details. Each display mode has a name table and pattern generator table. These hold data on the pattern plane.

With the exception of the 40 column text mode, all modes have a sprite pattern table and a sprite attribute table. The position of each sprite table does not vary between modes:

Sprite attribute table: 6912
Sprite pattern table: 14336

Example: `BASE(0)` returns 0: the base of the name table in the text mode.

`BEEP`

A beep sound is generated.

`BIN$(Y)`

A function which returns a string of the binary equivalent of the decimal expression Y.

Y must lie in the range -32768 to 65535. A negative value is expressed in the two's complement notation.

Example: `BIN$(-1)  RETURN:"1111111111111111"`

`BLOAD`

Variants:
```
BLOAD"CAS:"
BLOAD"CAS:file name"
BLOAD"CAS:",R
BLOAD"CAS:",offset
```

Loads a machine language program from cassette (The only device supported

by version 1 BASIC) to the location from which it was saved.

The R option executes a call to the location specified at BSAVE on completion of the load. The offset option displaces the position of the program from which it was saved.

Example: BLOAD"CAS:",R,&H20 will load and call the next machine language program found. The program will be loaded 32 locations up in memory from the position from which it was saved.

```
BSAVE"CAS:",start,end
```

Variants:
```
BSAVE"CAS:filename",start,end
BSAVE"CAS:",start,end,call location
```

Used to save a block of memory to cassette. An option is to specify the location to be called if the program is loaded with the instruction: BLOAD"CAS:",R.

If BLOAD,"CAS:",R is used without a call location having been specified when the program was saved, the call is made to the first location occupied.

Example: BSAVE"CAS:",34000,38000,36000 saves the 4K block between 34000 and 38000 inclusive. If the program is then loaded using the BLOAD"CAS:",R option, then on completion of input a call is made to location 36000.

```
CALL statement name
```

Variants:
```
CALL statement name (argument)
CALL statement name (argument,argument ....)
```

Note: CALL may be entered as an underscore '_'

Call is used to execute an extended command provided by a ROM cartridge.

Example: CALL SPSET(176,32)

```
CDBL(Y)
```

Y is converted to a double precision number. Y may be of any numeric type.

```
CHR$(Y)
```

A single character string with the character code Y is returned. To print a graphics symbol, first print CHR$(1).

Example: CHR$(65) returns A

```
CINT(Y)
```

Converts Y to an integer number by truncation. An error is flagged if Y is not in the range -32768 to 32767.

Example: `Y=4.8:?CINT(Y)` gives 4

```
CIRCLE(X,Y),radius
```

Variants:
```
CIRCLE(X,Y),radius,colour
CIRCLE(X,Y),radius,colour,start angle,end angle
CIRCLE(X,Y),radius,,start angle
CIRCLE(X,Y),radius,,,end angle
CIRCLE(X,Y),radius,colour,s.a,e.a,aspect ratio
```

Default start angle: 0 End angle: 2PI

The centre of the circle, specified in absolute coordinates by X & Y, may also be given relative to the current reference position by `STEP(X,Y)`.

If no colour is specified, the current foreground colour is used. If the start and end angle options are used, the parameter must be given in radians in the range –2PI to 2PI. The aspect ratio determines the horizontal : vertical ratio of the radii of the ellipse.

Example: `CIRCLE(20,20),10,10,3.142,2` produces a half ellipse from PI to 2PI.

```
CLEAR
```

Variants:
```
CLEAR
CLEAR string space
CLEAR string space,memtop
```

Memtop default: &HF380

All open files are closed, numeric variables are set to zero and strings to null.

Default string space is 200 bytes. Memtop is the highest memory location for use by BASIC. Memtop may be lowered to allow a machine language program to be placed above the BASIC area. This ensures that the code will not be overwritten by BASIC data.

Note that all `ON ... GOTO / GOSUB` statements are also cleared by this instruction.

Example: `CLEAR 400,44000` resets the space available for string variables to 400 bytes and alters the BASIC ceiling to 44000.

CLOAD

Variant: CLOAD"filename"

CLOAD closes all program files and clears all variables before loading the next program file from cassette. If a file name is given, only the first six characters are significant.

CLOAD?

Variant: CLOAD?"filename"

The program file in memory is compared with the next program file on cassette. A mismatch produces the report "Verify error".

CLOSE

Variants: CLOSE
CLOSE #Y
CLOSE #Y,#Z

Note: The hash symbols are optional.

Unless a file number is specified, all open files are closed and any associated buffers released. Any data contained by an output buffer is first output.

Example: CLOSE 4 will close and release any buffers associated with channel 4.

CLS

Clears the screen to the background colour specified by the COLOR statement. In a text mode, the cursor is also homed to the top left corner. CLS operates in all modes.

COLOR foreground,background,border

Variants: COLOR,background
COLOR ,background,border
COLOR ,,border
COLOR foreground,,border

COLOR is valid in all display modes. In either text mode, the background colour is assumed immediately. The high resolution graphic and multicolour modes require a CLS instruction to implement the change. In the 40 column text mode, the border colour takes the background colour. The complete VRAM colour table is updated in all modes.

Example: `COLOR4,10,14` will set the border to grey (14), the pattern plane to dark yellow (10) and the ink to dark blue (4).

`CONT`

Continues program execution after a break or stop.

`COS(Y)`

Returns the cosine, in radians, of the angle given by the expression Y.

`CSAVE"file name"`

Variant: `CSAVE"filename",baud rate`

The current program file is saved to cassette. Only the first six characters of the file name are significant. The file may be saved at either 1200 (default) or 2400 baud:

```
CSAVE "name",1 .... 1200 baud
CSAVE "name",2 .... 2400 baud
```

`CSNG(Y)`

Y is converted to a single precision number.

`CSRLIN`

Returns the vertical position of the text cursor from 0 to 23.

Example: `ROW=CSRLIN`

`DATA`

A storehouse of data that is accessed by the `READ` statement. Numeric and string data may be mixed in one line. Data items must be separated by a comma. Strings do not need quotation marks unless they contain commas, colons, semi-colons or significant leading or trailing spaces.

Example: `DATA 1.4,6,8.0,,,,4,2`

`DEF FN Y(A) = Expression`

Variant: DEF FN Y(A,B,C) = Expression

A function is defined by an expression that typically contains the bracketed parameter(s). When evaluated, the value(s) given in brackets are substituted into the expression. These must be of the same type as the parameter(s) in the definition. A maximum of eight parameters may be used within the definition.

Examples:
`DEF FN Y(S) = 2*S:T = 4: W = FN Y(T)` yields 8.

`DEF FN Y(S,A) = 2*S + (A+4):T = 2: R = 4: N = FN Y(T,R)` yields 2*2 + (4+4)

```
DEFDBL Y
DEFINT Y
DEFSNG Y
DEFSTR Y
```

Variants: `DEFINT W-Z`
`DEFDBL A,B-S`

Each declares any variable (both simple and array) which starts with the letter given, to be of that particular variable type. This global allocation may be over-ridden for an individual variable by a subsequent type declaration.

Example: `DEFINT A,B` would define all variables starting with either A or B to be integer variables.

`DEFUSRx = address`

`DEFUSR` specifies the start address of a machine code routine. The routine is called with the `USR` function. The integer x may be in the range 0 to 9. If omitted, a value of zero is assumed.

Example: `DEFUSR2 = &H4000`

`DELETE line number-line number`

Variants: `DELETE line number`
`DELETE -line number`

Either line number may be ommitted, but if given, an actual line of that number must exist.

`DIM Y(x)`

Variants: `DIM Y(x),Z(n)`

If more than eleven elements of a variable are to be used then space must

be reserved in memory by the use of the DIM statement. A string array will allocate 255 bytes for each element.

Example: `DIM Y(20),S$(12)` will allow the elements Y(0) - Y(20) and S$(0) - S$(12) to be used.

`DRAW "string of subcommands"`

The principle graphics command with a subset of 13 single letter commands. See chapter 2 for details.

`END`

Terminates program execution and closes all open files. No break message is printed.

`EOF(file number)`

Used to test for the file end when inputing data from a sequential file. Returns zero except at the file end when -1 is given.

Example: `IF EOF(8)=-1 THEN 200`

`ERASE array variable`

Variant: `ERASE array variable,array variable,....`

Deletes the array(s) which start with the specified letter(s). To be re-used, the array must be re-dimensioned.

Example: `ERASE X,Y` will erase `XC(16)` and `YJ(14)`.

`ERL`

On occurrence of an error `ERL` contains the line number in which the error arose. If an error occurred in the direct mode, `ERL` contains 65535.

`ERR`

On the occurence of an error, `ERR` is set to the respective error code.

`ERROR x`

Used to force the specified error. The x must be an integer in the range 1

to 255. BASIC uses errors in the range 1-59, which, if forced print the relevant error message. The message "Unprintable error" is produced if an error is forced using the ON ERROR GOTO instruction, for which no definition has been made.

Example:
```
10 ON ERROR GOTO 200
.
.
100 ERROR 100
.
.
200 IF ERR=100 THEN END
```

EXP(Y)

The value e^y is returned where e is the Napierian constant. The expression y must evaluate in the range –147.3654459516 to 145.06286085862

FIX(Y)

Returns the integer of the expression Y. This function differs from that of INT in the treatment of negative values. Whilst FIX will simply truncate the fractional section of the value, INT rounds the value down.

Examples: FIX(-2.8) returns –2
INT(-2.8) returns –3

FOR Y=x1 TO x2 STEP x3

All subsequent instructions upto the appropriate NEXT statement are repeated for the inclusive values of Y from x1 and x2 in increments of x3. If STEP x3 is omitted, the step size defaults to one. If it is set to zero the loop is repeated.If x2 is <= x1 with a positive step size, the instructions are executed once.

FRE("")

Returns the remaining number of bytes available for string storage.

FRE(x)

A function which returns the remaining number of bytes available for program storage. Any value may be given to x as it is a dummy argument & is not used in the evaluation.

Example: Z=FRE(2)

```
GOSUB / RETURN
```

Variant: `RETURN linenumber`

A specialized form of the `GOTO` statement. A return back to the statement following the `GOSUB` instruction occurs when a `RETURN` instruction is encountered.

```
GOTO
```

A statement which forces a branch to the specified line.

```
HEX$(Y)
```

A function which returns the hexadecimal equivalent of the integer Y. The latter must lie in the range –32768 to 65535.

Example: `HEX$(20)` returns 14

```
IF condition(s) THEN
```

Variants:
```
IF..THEN..ELSE
IF..THEN..IF..THEN..ELSE
IF..GOTO
IF..GOTO..ELSE
```

Allows sets of statements to be executed if certain criteria are satisfied. If a branch is to be made, it is not necessary to use `GOTO` after `THEN` and `ELSE`. The variant `IF..THEN..IF..THEN..ELSE` will continue program execution from the following line if the initial condition is not true.

Example: `10 X=0: IF X=1 THEN IF Y=2 THEN 40 ELSE 80`
will continue with line 20.

```
INP(port address)
```

A single byte of data is read from the port specified.

```
INPUT Y
```

Variants:
```
INPUT "prompt string"; Y
INPUT #file number, Y
INPUT$(x)
INPUT$(x,file number)
```

The statement prints a question mark to prompt the user to enter data. This is then allocated to the variable(s) specified.

If the variable has been declared prior to the INPUT statement and the user enters no data before pressing the RETURN key, the value is unaltered - otherwise it is set to zero.

Variables may be simple or array and either string or numeric. More than one variable may be used if separated by a comma in both the statement and input.

Example: INPUT "X,Y$ <ENTER>";X,Y$ will print: X,Y$ <ENTER> ?

The second variant is similar except that a file number is specified as the data source. Variants 3 & 4 both return a string of x characters - variant 3 from the keyboard and variant four from the file specified. In either instance, it is not necessarry to terminate input with a carriage return.

INKEY$

A function that returns the first character in the keyboard queue or, if that is empty, a null string.

```
Example: 10 A$=INKEY$:IF A$=""THEN 10
```
Note that this may be achieved more efficiently by using A$=INPUT$(1).

INSTR(A$,B$)

Variant: INSTR(x,A$,B$)

Returns the position of B$ in A$. If B$ is null or is not contained in A$ then zero is returned. The variant allows the search to commence x characters from the left of A$.

Example: A=INSTR(2,"ABCD","D") sets A to 4.

INT(Y)

Returns the integer of Y. See FIX for comparison.

Example: INT(4.8) returns 4

INTERVAL ON / OFF / STOP

Activates, deactivates or suspends the call to a subroutine at the interval specified by the ON INTERVAL statement. See chapter two for detail.

KEY function key number,"string"

Allocates the given string to the specified function key. The string may be

up to 15 characters in length.

Example: `KEY 2,"RUN"+CHR$(13)`

`KEY LIST`

Lists, in order, the strings allocated to the function keys.

`KEY ON / OFF`

Switches on or off the function key display on the 24th text screen row.

`KEY (function key number) ON / OFF / STOP`

Activates, deactivates or suspends the call to a subroutine specified by the `ON KEY GOSUB` statement. If activated, a check to see if the key has been pressed is made after each BASIC statement has been executed. See chapter two for detail.

`LEFT$(Y$,x)`

Returns the x leftmost characters of Y$. A graphic character occupies two positions. If x is zero a null string is returned. If x exceeds the number of characters in Y$, only Y$ is returned - spaces are not added.

Example: `? LEFT$("NEWORD",2)` prints NE.

`LEN(Y$)`

A function which returns the number of characters - including control and graphic characters - in Y$. A graphic character is a composite of `CHR$(1)` and the character code.

`LET Variable=Expression`

Used to assign the value of the expression to the given variable.
NOTE: Optional in MSX BASIC

`LINE (x1,y1)-(x2,y2)`

Variants:
```
LINE (x1,y1)-(x2,y2),colour
LINE (x1,y1)-(x2,y2),colour,B
LINE (x1,y1)-(x2,y2),colour,BF
LINE -(x2,y2),colour
```

The STEP(X,Y) form may replace either absolute coordinate specifier. In either position, the STEP form is relative to the initial reference point.

A graphics mode instruction to draw a line between the given coordinates. The B option draws a rectangle and the BF option draws and fills a rectangle. See chapter 2 for detail.

```
LINE INPUT Y$
```

Variant: `LINE INPUT "prompt"; Y$`

Obtains a line of input from the keyboard. The instruction may not be used in either the HRG or multicolour display modes. The number of characters entered must not exceed 254 with input assigned when the RETURN key is pressed.

```
LINE INPUT# file number, Y$
```

Inputs and assigns a line (up to 254 characters) from the specified sequential file to the given string.

```
LIST
```

Variants:
```
LIST line number
LIST line number-
LIST line number-line number
LIST -line number
LIST .
```

A command to list all or part of a program. A full stop (period) may be used to list either:

1. The last line previously listed.

2. The line that caused program execution to be aborted.

```
LLIST
```

Variants: As for LIST

LLIST outputs all or part of the current program file to a line printer.

```
LOAD "file name"
```

Variants:
```
LOAD"device reference"
LOAD"device reference:file name"
LOAD"device reference:file name",R
```

The LOAD command closes all open files and deletes the current program from memory prior to loading the speciifed ASCII file. If the R option is taken, no files are closed and a run is initiated after the load sequence. For a list of device references see OPEN.

```
LOCATE column,row
```

Variants:
```
LOCATE ,row
LOCATE ,,cursorswitch
LOCATE column,row,cursor switch
```

The LOCATE statement may only be used in a text mode and moves the cursor to the column and row specified. In the 40 column mode, the range is:

Columns 0 - 36
Rows 0 - 22/23 (dependent on the cursor key display)

In the 32 column mode:
Columns 0 - 28
Rows 0 - 22/23 (dependent on the cursor key display)

The unused columns may be utilized by poking into VRAM - see chapter two for further details. The cursor switch may take one of two values:

0 : disables cursor display.
1 : enables the cursor display.

```
LOG (Y) Y>Ø
```

Returns the natural logarithm of the expression Y. Y must be greater than zero.

```
MAXFILES=Y
```

A statement to specify the maximum number of files which may be open at one time. The integer expression Y must evaluate in the inclusive range 0 to 15. This statement is required if more than one file is to be open at any time.

```
MERGE
```

Variants:
```
MERGE"filename"
MERGE"device reference"
MERGE"device reference,filename"
```

MERGE will load and combine the next ASCII program file on tape with the program in memory. If the line numbers overlap, the initial lines are replaced by those in the second program.

Example: To load and merge the next program on tape use: `MERGE"CAS:"`

`MID$(A$,x)=character`

Variant: `MID$(A$,x,n)=n characters`

An instruction used to replace the xth character of `A$`. Note that all graphic characters have a header of `CHR$(1)`. The variant allows a sequence of n characters to be updated. A null string is returned if x is greater than the length of A$.

Example: `A$="WAS":MID$(A$,1,2)=" I":?A$` gives " IS".

`MOTOR`

Variants: `MOTOR ON` (default condition)
`MOTOR OFF`

Unless a state is specified, the cassette motor switch is toggled.

Example: If the cassette motor is stopped then `MOTOR` will reverse the setting, switching the motor on.

`NEW`

The current program is deleted from memory and all variables are reset.

`OCT$(Y)`

Returns a string of the octal value of the decimal expression Y.

`ON Y GOTO x1,x2,..xn / ON Y GOSUB x1,x2,..xn`

If Y=1 a jump or subroutine call is made to the first line in the list: x1. If Y=2, a jump or call is made to the second line number and so on. If the expression Y is either zero or greater than the number of lines in the list, then program execution continues from the next line.

`ON ERROR GOTO x`

If an error arises, a jump is forced to line x. If line 0 is specified, error handling is normalized. The error handling routine is terminated with a `RESUME` instruction.

`ON INTERVAL = Y GOSUB X`

Defines the subroutine executed after each interval (length Y * 1/50th of a second) has elapsed. The sequence is initiated by the INTERVAL ON statement.

```
ON KEY GOSUB x1, x2, .. xn
```

Initiates a call to the given subroutine when a particular function key is pressed. The first line number corresponds to key 1, the second to key 2 etc. These are called if the relevant key is pressed after a KEY (x) ON statement.

Example ON KEY GOSUB 20,,,40 keys two and three not set up.

```
ON SPRITE GOSUB Y
```

Allocates the subroutine to which trapping occurs after a sprite to sprite collision. Enabled by the SPRITE ON statement.

```
ON STOP GOSUB Y
```

Initiates a call to the subroutine at line Y if the STOP and CTRL keys are pressed simulaneously. This action only occurs after a STOP ON statement has been made.

```
ON STRIG GOSUB x
```

Variants: ON STRIG GOSUB x1, x2, x3, x4

Identifies the subroutine to which trapping takes place if the space bar is pressed. Trapping is enabled by the STRIG ON(Y) statement.

The variant is used to allow either trigger 1 or trigger 2 of either joystick to initiate a trap. The subroutines must be given in the following order:

1. Space bar
2. Trigger 1, joystick 1
3. Trigger 1, joystick 2
4. Trigger 2, joystick 1
5. Trigger 2, joystick 2

Example: STRIG(2) ON:ON STRIG GOSUB 200,400,600 will initiate a call to the subroutine at line 600 if trigger 1 of joystick 2 is pressed.

```
OPEN "devicereference:" AS file number
```

Variants:
```
OPEN "devicereference:file name" AS file number
OPEN "devicereference:file name" FOR mode AS file No.
```

This statement opens a channel to a device and allocates an I/O buffer. A file must be opened before an instruction requiring a file number may be used. For example PRINT #, INPUT #, PUT or GET. Four device references may be used:

1. CAS: Cassette
2. LPT: Line printer
3. CRT: Screen
4. GRP: Graphic screen

The file number is used by other input/output instructions to refer to the file and must be in the inclusive range 0-Y where Y is specified by the MAXIFILES instruction.

The variant 'FOR mode' sets the type of data transfer:

'INPUT' : Sequential input
'OUTPUT' : Sequential output
'APPEND' : Sequential append

```
OUT port number, data byte
```

Transfers a data byte to the specified port. Both the port number and data must be in the inclusive integer range 0 to 255. The standard MSX configuration does not support I/O to more than 256 ports.

```
PAD(Y)
```

Returns the status of a touch pad. Y may take a value in the inclusive range 0-7:

0 - 3 Touch pad connected to Port 1

4 - 7 Touch pad connected to Port 2

Of the four values which may be selected for each port, 0-3 and 4-7, each returns a different parameter:

0 and 4 : Return = –1 if pad pressed, 0 if released.

1 and 5 : Return = X coordinate of point pressed

2 and 6 : Return = Y coordinate

3 and 7 : Return = –1 if switch pressed, 0 if not.

```
PAINT (X,Y)
```

```
Variants: PAINT STEP (X,Y)
          PAINT (X,Y), colour
          PAINT (X,Y), colour, borderline colour
```

An instruction which may be used in either the multicolour or high resolution modes. The enclosed object is filled with the foreground colour from the position specified. Note that in the HRG mode the border line colour must always be the same as the paint colour. See chapter two for detail.

```
PDL(Y)
```

Returns the status of a paddle connected to either terminal A or terminal B. The expression Y must be an integer in the inclusive range 1 to 12. If it evaluates to an odd number, input is read from terminal A otherwise terminal B. The value returned is in the range 0 to 255.

```
PEEK(Y)
```

Returns the integer value held by the location specified. The numeric expression Y must evaluate in the inclusive integer range –32768 to 65535.

```
PLAY "subcommand string"

Variants PLAY "string","string","string"
         PLAY "string","","string"
         PLAY Y$,Z$,X$
         PLAY "Y$;string","string","string"
```

The principle instruction for the generation of sound. Each of the three strings specifies the output of one sound channel. See chapter two for detail.

```
PLAY(Y)
```

A function which returns the status of the specified music channel(s). Y must evaluate to an integer in the inclusive range 0 to 3. If there is data in the buffer specified, -1 is returned. If the channel is not active, zero is returned. `PLAY(Ø)` gives the status of all three channels.

```
POINT(X,Y)
```

Returns the colour code of the specified pixel. The instruction is active only in the two graphics modes. A value of –1 is returned if either coordinate is out of range.

```
POKE X,Y
```

Places the value Y in location X. Y must be in the range 0 to 255 inclusive. The address range is –32768 to 65535. A negative address is first added to 65535.

Example: `POKE 40000,254` places 254 in location 40000.

`POS(Y)`

Returns the horizontal position of the cursor in either text mode. Y is a dummy argument and is not used in the evaluation.

`PRESET (X,Y)`

Variants:
```
PRESET (X,Y), colour
PRESET STEP(X,Y)
```

The specified graphic screen pixel is set to the background colour. The variant allows the colour to be specified.

`PRINT`

Variants:
```
PRINT "string"
PRINT "string";
PRINT A$
PRINT Y
PRINT ,X$
PRINT CHR$(Y)+A$
```

NOTE: `PRINT` may be entered as ?

If no expression(s) follow the keyword then a single line feed is printed. A carriage return is printed after the final item unless the item is followed by a semi-colon or comma.

A semi-colon causes the next item to be printed immediately after the last and a comma advances the print position to the next tabulation zone. Tabulation zones are 14 characters wide.

Control characters may be printed using the `CHR$(Y)` function.

`PRINT # file number, expression`

As for `PRINT` albeit the data is directed to the channel specified. A carriage return and line feed follow the final item.

`PRINT USING`

Variant: `PRINT # file number, USING`

Permits items to be printed in a specified format by the use of control characters. See chapter 2 for detail.

```
PSET (X,Y), colour
```

Variant: `PSET STEP(X,Y), colour`

The pixel specified, on a graphics screen, is set to the given colour.

```
PUT SPRITE sprite plane number
```

Variants:
```
PUT SPRITE plane number, (X,Y)
PUT SPRITE plane number, STEP (x,y)
PUT SPRITE plane number, (X,Y), colour
PUT SPRITE plane number, (X,Y), colour, pattern No.
```

The principle sprite display command. `PUT SPRITE` sets the position, colour and pattern for one sprite. The sprite cursor is moved to the sprite's top left-hand corner.

Only one sprite may be placed on any plane but any number of sprites may take a particular pattern. See chapter 2 and `SPRITE$` for detail.

Example: `PUT SPRITE 0,(40,40),4,2`

```
READ Y
```

Variant: `READ Y,X,Z,....`

The `READ` statement is used to allocate the constants in `DATA` statements to the given variables. The data read must agree with the variables specified.

```
REM
```

A statement used to annotate a program listing. Any following statements on the same logical line are ignored at run time with execution proceeding from the next line. `REM` may be entered as a single quotation mark '.

```
RENUM
```

Variants:
```
RENUM new number
RENUM new number,old number
RENUM new number,old number,increment
RENUM old line number
```

Default new number and increment: 10

All program lines are renumbered inclusive of references following GOTO, GOSUB, THEN, ELSE, ON..GOTO, ON..GOSUB, IF THEN and ERL statements.

The resulting program commences with the line number 10 (or if specified) the new number. The variant 'old number' causes only part of the program, from the old number onwards, to be renumbered.

RESTORE

Variant: RESTORE line number

Resets the data pointer to the first item in the initial DATA line. The variant allows the DATA line to be specified.

Example: DATA 200 would cause the next READ statement to take data from the DATA statement in or following line 200.

RESUME

Variants: RESUME NEXT
RESUME line number

The RESUME instruction is used to end an error handling routine that is trapped to by the ON ERROR GOTO statement. It causes program execution to continue with the statement that produced the error. The RESUME NEXT statement is similar except execution recommences with the statement after that which caused the original error.

RIGHT$(A$,x)

A function which returns a substring consisting of the x rightmost characters of A$.

If x is greater than the number of characters in A$ then A$ is returned. If x=0, a null string is returned. A graphics character always includes a prefix of CHR$(1). Example: RIGHT$("LEFT",20) returns LEFT

RND(Y)

Returns a random number in the exclusive range 0 to 1.

If Y > 0 the next number in the sequence is generated.

If Y = 0 the number returned is equal to the previous number.

If Y < 0 the random generator is reseeded for that Y.

Note that the sequence produced after `RND(-2)` is the same as that produced by a subsequent `RND(-2)`. To produce a 'new' sequence use `RND(-TIME)`.

`RUN line number`

Executes the current BASIC program in memory from the line specified. If no line number is given, execution commences with the first program line.

`SAVE "file name"`

Variants: `SAVE "device reference:"`
`SAVE "device reference:file name"`

Outputs a BASIC program file in ASCII format to the device specified. Control Z equals EOF. Note: it is not possible to verify a file. The transfer rate is specified by the `SCREEN` statement.

Example: `SAVE "CAS:PROG4"`

`SCREEN mode, sprite size, key click switch, cassette baud rate, printer option`

The `SCREEN` statement is used to assign the six options as follows:

Mode: 0 40 * 24 text mode
1 32 * 24 text mode
2 High resolution mode
3 Multi-colour mode

Sprite size:

0 8 * 8 unmagnified
1 8 * 8 magnified
2 16 * 16 unmagnified
3 16 * 16 magnified

Key click switch:

0 Switch on
1 Switch off

Cassette baud rate:

1 1200 baud
2 2400 baud

- for both the program and ASCII formats.

Printer option:

0 MSX printer
‹0› No MSX graphics facilities

If the latter option is taken, all graphics symbols are output as spaces.

`SGN (Y)`

If Y > 0 then `SGN` returns 1, if Y = 0 then zero otherwise –1.

`SIN (Y)`

`SIN` returns the sine of the expression Y in radians.

`SOUND register, Y`

The `SOUND` instruction writes the value of expression Y to the specified PSG register. Y must be in the inclusive range 0 to 255. See the chapter on the AY-3-8910 for more detail.

Example: `SOUND 4,4`

`SPACE$ (Y)`

Returns a string of Y spaces. Y must be in the inclusive range 0 to 255.

`SPC (Y)`

Used in conjunction with either the `PRINT` or `LPRINT` statements to produce Y spaces. Y must be in the range 0 to 255 inclusive.

Example: `?SPC(4);"OVERWRITES"`

`SPRITE$ (Y) = "Definition string"`

A system variable for sprite pattern definition. The character code of each character of the string allocated, is the pattern of one byte of the sprite. The binary form permits straightforward entry:

`CHR$(&B10101011)`

If 16 pixel square sprites are in use, 64 sprite patterns may be defined, and Y must be in the inclusive range 0 to 63. Eight pixel square sprites allow upto 255 pattern definitions. See chapter 2 for detail.

SPRITE ON / OFF / STOP

Activates, deactivates or suspends the call to a subroutine in the event of a sprite to sprite collision. The ON SPRITE GOSUB statement must first be used to specify the subroutine called.

Example: SPRITE STOP will suspend calls to the subroutine in the event of a sprite to sprite collision. The call is made when trapping has been enabled by a SPRITE ON statement.

SQU (Y)

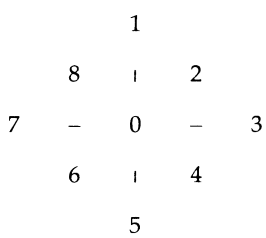
Returns the square root of the expression Y which must be equal to or greater than zero.

STICK (Y)

Y may take a value from 0 to 2:

0 Cursor key
1 Joystick - Port 1
2 Joystick - Port 2

The direction is returned:



STOP

A statement which terminates program execution and causes the message:

Break in ...

to be printed.

STOP ON / OFF / STOP

Activates, deactivates or suspends the call to a specified subroutine in the

event of the CTRL and STOP keys being depressed. See chapter 2 for detail.

`STRIG (Y)`

The integer expression Y must be in the range 0 to 4:

0 Space bar
1 Joystick 1, Trigger 1
2 Joystick 2, Trigger 1
3 Joystick 1, Trigger 2
4 Joystick 2, Trigger 2

If the 'trigger' is pressed then –1 is returned, otherwise 0.

`STRIG (Y) ON / OFF / STOP`

Activates, deactivates or suspends the trapping of a particular 'trigger' button. Y may be in the range 0 to 4 as given above. The `ON STRIG GOSUB` statement must first be used to define the subroutine called.

Example: `STRIG (Ø) ON` enables trapping in the event of the space bar being depressed.

`STRING$ (x,Y)`

Variant: `STRING$ (x,Y$)`

Returns a string of length x containing only the character with character code Y or the first character of Y$. If a graphics character occurs first, a string of character 1 is returned.

Example: `STRING$(4,65)` returns AAAA

`STR$ (Y)`

Returns the string representation of the numeric value Y. This provides a convenient conversion method from the hexdecimal, octal or E forms.

`SWAP Y,X`

Exchanges the values allocated to each variable. Both variables must be of the same type.

Example: `A$="w":B$="e":SWAP A$,B$:? A$` prints e

`TAB (Y)`

A function which may only be used after either the PRINT or LPRINT statements. TAB moves the print position to column Y - unless it is already to the right of that column, when no action is taken. Y must be in the inclusive range 0 to 255.

Example: `PRINT TAB(4);"ON 4th COLUMN"`

`TAN (Y)`

A trigonometric function that returns the tangent of Y in radians.

`TIME`

A special integer variable that is set to zero on power up and incremented every 1/50th of a second (UK). The update occurs during the VDP system interrupt which is suspended during tape input/output.

`TRON / TROFF`

The TRON statement causes each line number of the program to be printed as it is executed. The TRON statement is accepted in either the direct or indirect modes and is negated by the TROFF command.

Example:
```
10 TRON
20 REM
30 REM
40 TROFF
```

will print `[20][30][40]`

`Z=USR (Y)`

Variant: `Z=USR x (Y)`

USR x (Y) executes a call to the location specified by the DEFUSRx instruction. The value of the expression Y is passed to the routine. If no argument is to be transfered, a dummy value for Y must be given.

On completion of the machine language routine, the value passed back, is assigned to Z.

At both the call and the return, the data element of the exchange value starts at &HF7F6. The data type is indicated by the value in the accumulator:

Integer 2

String 3

Single precision 4

Double precision 8

The location of a string element is communicated by the DE register pair. The element consists of three bytes:

1. The number of characters in the string.

2 & 3. String storage location.

Example:

```
10 DEFUSR4=&HD000
20 Y=USR 4 (X)
```

`VAL (Y$)`

Returns the numeric value of the string Y$. Any preceeding spaces, tabs or linefeeds are ignored.

`VARPTR (Y)`

Variant: `VARPTR (# file number)`

A function which returns the first location of the data table element of the variable specified.

Any type of variable may be given. If the address returned is negative then 65536 must be addded to give the correct location.

See the section 'Program storage' at the end of chapter 2 for more detail.

`VARPTR (# file number)` returns the first location of the file control block.

`VDP (Y)`

A special variable where Y must lie in the inclusive range 0 to 8. Values 0 to 7 return the value held by that VDP write only register. VDP(8) yields the value in the VDP's status register. See the section on the VDP for more detail.

`VPEEK (address)`

Returns the value held in the specified VRAM location. The address must lie in the range 0 to 16383 inclusive.

`VPOKE X,Y`

A function that places the value Y in VRAM location X. Y must lie in the range 0 to 255 inclusive.

```
WAIT port, byte value
```

Variant: `WAIT port, byte value, mask`

An instruction which pauses program execution until a byte is input from the port specified. The byte must have a bit set in the same position as has the byte specified in the statement.

The variant first logically ORes the input byte with the mask byte, then it is AND'ed with the byte value.

All expressions must be integers in the range 0 to 255 inclusive.

```
WIDTH text screen width
```

An instruction which sets the width (in columns) of the screen in either text display mode.

The value allocated must be in the inclusive ranges:

1 - 40 in 40 * 24 text mode
1 - 32 in 32 * 24 text mode

Example: `WIDTH 20`

Chapter 4

Z-80 Machine Language

Microprocessors. The Z-80 Architecture. Instruction set. Address Modes.

Microprocessors

At the heart of each MSX system computer lies a Z-80A microprocessor. This unit is programmed directly using machine code and has three principle capabilities:

i. Simple arithmetic.

ii. The movement of data between locations.

iii. Logic operations.

A machine code program called the BASIC interpreter is held in R.O.M. (Read Only Memory). This compiles BASIC instructions into machine code instructions which are then executed by the Z-80.

The ommission of this 'translation' stage permits a machine code program to execute more rapidly than an equivalent BASIC routine. Hence, the description of BASIC as a 'high level language' which supports commands that are not intrinsic to the microprocessor.

So, speed is the main advantage of using machine language - but what are the drawbacks? Whereas BASIC was designed to allow the computer to be used and programmed with as little knowledge of the machine as was practical, working at the machine language level requires a little more awareness of what's actually going on and a familiarity with the binary and hexadecimal number systems - a small inconvenience for the dazzling effects that may be realised.

Right, we know that there is a Z-80 in there supported by a 9129 VDP and AY-3-8910 sound chip, but how does it all interconnect?

Communication between the units is via two main thoughfares: the address and data buses. The address bus is sixteen lines or wires running between the Z-80 and the ROM and RAM memory arrays. The data bus runs parallel

but has only eight lines.

Using the 16 line address bus, the Z-80 can specify any location in the range 0 to 65535. Each location - whether RAM or ROM - can hold a number between 0 and 255.

When the Z-80 outputs an address on to the bus and indicates to memory that a read and not a write is necessary, the number held by the address is automatically placed onto the data bus. The processor then gates this into an internal register.

Plainly, the system needs to be exactly synchronised and a timing reference - a 3.57 MHz clock - is used. The clock is external to the Z-80.

The VDP also has access to the data bus to allow the CPU to pass data and instructions. In addition, the VDP has 16K of RAM, which only it may access directly. For this it has a 'private' bidirectional eight bit data bus.

From the mode instructions given by the processor and the contents of the VRAM, the VDP builds up a video display. This is output directly to a monitor or modulated to drive a domestic television.

On power up, the definitions of the character sets held in ROM are transfered by the CPU (Central processing unit, the Z-80) to the VDP and from there into VRAM.

System Organisation

In dealing with memory areas of such size, it is convenient to break them down into manageable blocks. A common division is into 'pages' each holding 256 addresses. The opening page is known as page zero and contains locations 0 to 255. The term 'page' may also be used to refer to a 16K block of memory. The time taken by the CPU to transfer data to and from page zero is less than when a page boundery needs to be crossed. As a consequence, most small computers reserve zero page for system use.

Instructions to the microprocessor take the form of either one or two bytes. If either data or an address need to be specified, one or two more bytes may also be necessary.

Thus a machine language program takes the form of a continuous sequence of instruction and data bytes.

Certain internal memory locations on the processor are accessible to the programmer. The most useful include:

1. Program counter

2. Stack pointer

3. 2 Index registers: IY and IX

4. The accumulator or A register

5. Six general registers B to L

6. Flag register F

7. Interrupt and refresh registers I and R

8. An alternate register set A' to L'.

The registers from 1-3 are 16 bits wide and may hold any number in the range 0 to 65535. The remainder are eight bits wide and are limited to values in the inclusive range 0 to 255.

Plainly, the processor must know where its instructions are located, and this is the function of the program counter register, which holds the address of the next instruction.

The flag register holds information on both systems status and the results of certain operations. For example, if a subtraction resulted in a negative value, the sign bit (S) would be set.

The flag register is used by the branch options in the instruction menu. These reset the program counter to point to a new location if one of the flags is either set or clear.

The two index registers IY and IX may each be loaded with any value the programmer wishes - as long as it is in range. These registers are primairily intended to hold the address of data or routines.

The six general registers: B,C,D,E,H,L, may also be loaded with any value in range. Alternatively, they can be paired - BC, DE, HL - to be used as 16 bit registers.

The A register or accumulator has a special importance which results from the system architecture of the Z-80. Most 8 bit arithmetic and logical operations require one of the data values to be initially loaded into the accumulator. After the operation, the result is placed in the accumulator.

Although this allows such operations to execute quickly, separate instructions are usually required to first load and then tranfer data from the accumulator. The Z-80 is also capable of 16 bit addition or subtraction. Here, the HL register pair replaces the A register as the accumulator. The alternate register set can be 'switched in' to replace registers A,B,C,D,E,F,H,L with A',B',C',D',E',F',H',L'. Thus two register sets are available for immediate data storage/handling. In practice, however, the second set is infrequently used - usually to enable interrupts to be rapidly serviced.

Binary and Hexadecimal Representation

Before you skip this section try these two questions:

1. What are the binary representations of –34 and 0.02?

2. What is the decimal equivalent of &H0FDE ?

Microprocessors were not designed to use the decimal number system, and to program in machine language it is necessary to be fluent in both the hexadecimal and binary number systems.

In decimal there are ten possible digits 0-9, in binary there are only two: 0 and 1, so any binary number is a string of 1's and 0's.

The other main difference is that instead of each column representing a value ten times that of the column to its right, in binary it is double.

Decimal	Binary representation 128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	
2	0	0	0	0	0	0	1	0	
3	0	0	0	0	0	0	1	1	
4	0	0	0	0	0	1	0	0	
5	0	0	0	0	0	1	0	1	
6	0	0	0	0	0	1	1	0	
7	0	0	0	0	0	1	1	1	
8	0	0	0	0	1	0	0	0	= 8*1 + 4*0 + 2*0 + 1*1
9	0	0	0	0	1	0	0	1	
100	0	1	1	0	0	1	0	0	
200	1	1	0	0	1	0	0	0	= 128 + 64 + 8

Figure 4.1 Binary column values

To convert a binary number to the decimal equivalent, it is necessary to add the values represented by the columns in which the number has 1's. In figure 1.1, the binary equivalent of 100 is seen as 64 + 32 + 4. Each 1 or 0 is referred to as a bit, a number with eight bits being termed a byte.

Clearly, the largest number that can be represented by an eight bit number is 128+64+32+16+8+4+2+1=255. It is conventional to include 0's to the full eight bits for those values less than 255.

The right hand bit is called the least significant bit (l.s.b.) and the left-most digit the most significant bit (m.s.b.). These abbreviations should not be

confused with M.S.B (Most Significant Byte) and L.S.B. (Least Significicant Byte). For example, in a sixteen bit address:

1111111100000010

the l.s.b. is 0, the m.s.b is 1 and the M.S.B. is 11111111.

In the following chapters we will often need to refer to a specific bit within a byte. It is conventional to number the bits as follows:

76543210

Binary addition and subtraction are straightforward with one exception - the representation of a negative number. As no equivalent of the minus sign is available, negative numbers are formed in a different manner. The method of coding used is two's complement notation. This is obtained by taking the absolute value of the negative number and converting it to it's binary form. Each digit is then inverted and finally one is added.

For example, to obtain the two's complement of -12, take the binary representation of it's absolute value (12) 00001100. Invert it: 11110011, and then add 1 = 11110100. As the processor must be able to differentiate this from 244 decimal, which has the same representation, a limit is imposed on the range of values: –128 to 127. This prevents any overlap occuring.

Operation	Example
Negative value	–121
Absolute value	121 / 01111001
Inverted form	10000110
Increment	00000001
Two's complement	10000101

Figure 4.2 Representation of negative values in signed binary

The division by two of a binary value is achieved by moving each bit one position right. Therefore, to obtain the binary equivalent of 1/2, the byte 00000001 is shifted to give 0.1. Figure 4.3 shows the column values after the binary point.

Decimal	fraction Binary
0.5	0.1
0.25	0.01
0.125	0.001
0.0625	0.0001
0.03125	0.00001

Figure 4.3 Binary fraction equivalents

Any eight bit number may be split to give two four bit 'nibbles' (Half a byte). In isolation, each represents a value in the range 0 - 15. This is the basis of a useful shorthand for binary data. Clearly, 16 different digits are needed, and 0 - 9 are supplemented with the first letters of the alphabet representing the values 10 - 15.

Decimal	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F 10 11

Figure 4.4 Hexadecimal notation

This method of representation is the Hexadecimal number system. For example, to convert 56 decimal to hexadecimal, first take the binary equivalent: 00111000, and extract the value of each nibble in isolation: 0011 and 1000 = 3 and 8. The hexadecimal form is therefore 38.

This also illustrates the simplicity achieved over binary groups. To convert two digit hexadecimal values to decimal, just multiply the left digit by 16 and add it to the righthand value.

You may have noticed that hexadecimal is a number system to the base 16. To convert from decimal to hexadecimal, it is only necessary to break the value down into the component powers of 16.

Hexadecimal	4	3	A	4	
Column value	4096	256	16	Actual	
Evaluation	4*4096+	3*256+	10*16	+4	= 17316

Figure 4.5 Hexadecimal conversion to decimal notation

Logical operations

Consider the following problem:

Given two 8 bit binary numbers, you are required to form another byte with 1's only in those positions in which both the original bytes have 1's.

The answer? Simple, first load one value into the A register, then perform the logical AND operation between it and the other byte. On completion the result - the byte required - is returned in the A register.

The AND operation is one of three logical operations the Z-80 will perform, which compare each bit of two bytes, & depending on the basis of that comparison, form a new value in the accumulator. As we have seen, the AND operation will place 0 in each bit position unless both comparison data also contain a 1 in that column. The inverse operation to this is the OR instruction, which only returns a 1 in any position if one or both of the initial data contains a 1. The third instruction, XOR, is a cross between the previous two. It requires both data to have dissimilar digits in order that a 1 result in the accumulator.

To the newcomer to machine code, these operations may seem unimportant - not so - they are often required for certain calculations and the control of particular functions.

Operation	AND	XOR	OR
Condition for a 1 to result	Both data hold a 1 in the column	Both data have dissimilar digits in the column	One or both data is a 1

Figure 4.6 The comparison basis of the logical operations.

The Z-80 Architecture

Zilog manufacture several versions of the Z-80 which differ in the maximum speed at which they can run. For the Z-80A in the MSX system, the maximum clock speed is 4MHz (the Z-80B may run at up to 6MHz).

In structural configuration the processor is a typical 40 pin Dual In Line Package (D.I.L.P). The address bus requires 16 lines and the data bus eight. Principle of the remaining pins are:

1. Power supply

2. Reset and interrupt pins: RESET, INT and NMI.

3. WAIT - pauses the processor, typically to allow memory to retrieve the necessary data.

4. BUSREQ and BUSAK - control lines to allow other processors access to the data and address buses.

5. RFSH - indicates that the lower 7 address lines are being used to refresh part of memory.

6. Clock input pins.

7. Read and write (RD & WR) pins.

8. MREQ and IORQ pins.

In addition to being able to transfer data to and from memory, the processor may input or output data to or from up to 256 8 bit ports. The control registers of the principle system components are accessed in this way. This permits most input/output operations to be initiated by the CPU simply writing to the relevant port.

To gain the attention of the processor in order to carry out an urgent function, a component may cause an 'interrupt'. The Z-80 then completes the current instruction and branches to a handler routine. After this has been completed, control reverts to the original program.

It is often not necessary to use the interrupt procedures if only use of the address and data busses is required. The Z-80 is able to effectively isolate itself from the address and data busses. The BUSREQ line requests use of the busses from the CPU which acknowledges it is isolated using the BUSAK line.

The MREQ line indicates that the processor is to perform a memory read or write and has put an address on the address bus. IORQ goes low to either acknowledge an interrupt or to indicate an I/O operation is taking place.

Internally the Z-80 comprises an Arithmetic Logic Unit (A.L.U.), control unit, registers and read only memory. Their operation is best shown by looking at a sample operation:

i. The address pointed to by the program counter is accessed and loaded into the processor's instruction register. In our illustration it is 01111100 or decimal 124. This instructs the processor to load the accumulator (A register) with the contents of the H register.

ii. The CPU increments the program counter to point to the next instruction.

iii. The byte in the H register is copied into the A register. This will overwrite any previous value held.

iv. The next instruction is loaded into the processor ....

The entire operation completes in 4 T cycles (external clock periods). This gives an indication of the tempo of the system - it also becomes clear why program errors are often difficult to trace! Although, in comparison with many 8 bit

processors the Z-80 is well equipped with on-board registers, they are frequently inadequate for storing all the information we need on hand. Happily, the Z-80 provides a larger store - the stack.

The stack is a section of memory that may be used to store two byte elements of data. The data stored is from any 16 bit register or register pair with the exception of the program counter.

The stack is aptly named except that it is upside down! As each byte pair is added or pushed onto the stack, the 'top' of the stack moves two locations down in memory. The location of the last byte of the pair pushed is held by the stack pointer register.

A drawback to the stack is that byte pairs have to be retrieved or 'pulled' on a last in first out basis.

Example: The contents of the 16 bit IY register are pushed onto the stack followed by the value in the BC register pair. To retrieve the first pair pushed, it is necessary to first pull the BC pair. The stack is also used to store subroutine and interrupt return addresses.

Before we introduce the instruction set, a few final words on the 256 I/O ports. It is possible to read or write a single data byte to any port using the relevant instruction. The processor uses the eight low order address lines A0 to A7 to select the port (a memory read or write does not occur as the IORQ line, not the MREQ line, is asserted). The data byte is then placed onto the data bus and is latched by either the Z-80 (READ) or the port (WRITE).

The Z-80 Instruction Set

The Z-80 has 158 types of instruction which Zilog group into 11 categories. Before considering these, it is necessary to look at the main methods used to input or write machine language programmes.

A quick look in the computer press will reveal that there are two common methods of documenting machine language routines. The most popular is simply to list the component hexadecimal byte values. The user simply pokes these into memory. The program is then run by calling the first location with the BASIC `USR` instruction.

The bytes are known as the object code. In this format, the sequence may be directly processed by the Z-80.

However, to manually poke each byte into memory is both a laborious and error prone process. A simple solution is to write a short BASIC routine which accepts two digit hexadecimal numbers. The program then automatically pokes the decimal equivalent into a sequence of locations. This is known as a hex-

loader, an example of which is given below:

HEX-LOADER

```
200 SCREEN 1:COLOR 10,1,1:CLS
210 LOCATE 2,2:?"HEX LOADER FI TO FINISH":?
220 INPUT"START LOCATION";ST
230 SW=ST-1:?
240 SW=SW+1:?SW;" ";HEX$(SW);" ??";
250 ?CHR$(29);CHR$(29);
260 A$=INPUT$(2)
270 IF A$="FI" OR A$="fi" THEN 320
280 A$(1)=MID$(A$,1,1):A$(2)=MID$(A$,2,1)
290 IF INSTR("0123456789ABCDEFabcdef",A$(1))=0 OR
INSTR("0123456789ABCDEFabcdef",A$(2))=0 THEN 260 ELSE ?A$;
300 A=VAL("&H"+A$):?" ";A
310 POKE SW,A:?:GOTO 240
320 ?:?:?:?"START ";ST;" END ";SW-1
```

If the machine language program is of significant length, the process of looking up the byte code(s) for each instruction is not practical and it becomes necessary to use an Assembler.

An assember program enables a routine to be written using mnemonics for the operations and if necessary, strings for addresses. The source file is then converted into object code by the assember. Once a familiarity with the relevant mnemonics has been gained, the production of machine code is almost as simple as writing a BASIC program. The source file usually consists of numbered lines each containing one or more machine language instructions. All respectable assemblers allow the screen editor to be used and feature single letter commands.

Almost all commercial assemblers consist of three packages:

1. Assembler

2. Monitor

3. Disassembler

Once the source file is complete, it is assembled into code at the position specified. The file is then saved as it may contain errors which can necessitate the computer being switched off before it can be reused. Finally, the monitor is entered to run the program - one instruction at a time, if necessary. The disassember is used to examine and alter areas of memory and is usually accessed from the monitor.

In addition to these principle operations a set of secondary functions will be provided, typically moving/saving/loading blocks of memory, search operations for strings etc.

The package chosen will be determined by the programming task to be attempted. The price range is considerable - from a few pounds upwards.

For casual use a tape based system will suffice, but for the more serious applications a cartridge based package is almost obligatory. The main reason is that almost all machine language programs will initially contain errors, and reloading the package after each unrecoverable crash can become a source of some irritation.

Although there are 158 instruction types supported by the processor, there are 666 individual opcodes. This arises since most instruction categories contain a range of operations, each of which performs an identical function but specify the data address in different ways.

The Z-80 supports ten addressing modes which we will examine more closely in the next section. The instruction menu may be divided into the following categories:

1. Eight and sixteen bit load operations.

2. Eight and sixteen bit arithmetic operations.

3. Eight bit logical instructions.

4. Rotate and shift operations.

5. Bit instructions.

6. Branch and subroutine operations.

7. Block transfer and search operations.

8. CPU control and I/O instructions.

Eight and sixteen bit load operations

An eight bit load instruction will copy the contents of one register or memory location into another register or memory location. The sixteen bit instructions perform an equivalent operation between a 16 bit register and memory or another register.

Note that when a sixteen bit value is stored from a register to location X, the high order byte is placed in location X+1 and the low order byte in location X. Similarly, if a 16 bit register is loaded with the two bytes from location X, then the value in location X+1 is copied into the most significant 8 bits of the register and location X into the least significant eight bits.

The Zilog assembler mnemonic for this operation is LD X,Y where Y is the byte which is copied into location or register X.

Examples:

`LD A,B` Load the A register with the contents of register B.

`LD H,A` Load the H register with the contents of register A.

`LD B,40` Load the B register with 40.

If an operand is enclosed by brackets, it refers to the address of the data to be used.

`LD A,(40)` would load the accumulator with the value held in location 40.

Similarly, `LD B,(HL)` would load the B register with the byte in the location held by the HL register pair. Finally, two examples of 16 bit loads:

`LD HL,(40)` which copies the value held by location 40 into the L register and that held by location 41 into the H register.

`LD IY,22` loads 22 into index register IY.

Eight and sixteen bit arithmetic instructions

The simplest of these are the increment and decrement instructions. These either increase or decrease the data specified by one. The respective mnemonics are INC and DEC.

Examples:

`INC A` will add one to the value in the accumultor.

`DEC IX` decreases the value held by the index register IX by one.

There are two mnemonics for both addition and subtraction: `ADD`, `ADC`, `SUB` and `SBC`. `ADD` and `SUB` are fairly straightforward. The A register is used to accumulate the result of 8 bit operations and the HL register pair for 16 bit operations. For example to add 5 and 4, the A register is first loaded with one value:

```
LD A,4
```

and then added to 5:

```
ADD A,5
```

The result is returned back in the A register. As we mentioned earlier, for either 8 bit addition or subtraction, it is obligotory to use the A register. Similarly for 16 bit arithmetic, the HL register pair must be loaded with one of the original values and returns the result (the exception which proves the rule is that an index register may be used as the accumulator for one category of 16 bit addition).

Example: `ADD HL,BC` adds the contents of BC to the value in the HL register pair. The result is placed in HL.

The mnemonics ADC and SBC stand for AdD with Carry and SuBtract with Carry. The 'carry' is a bit or flag in the Flag register. The flag is set if an addition produced a result too large to be held by the accumulator or a subtraction operation required a carry.

When an ADC is executed, the data specified, plus the carry is added to that in the accumulator. For example ADC A,4 would, if the carry flag was set, add five to the value in the A register.

The instruction set includes two operations which directly amend the carry bit:

1. CCF : The flag is complemented

2. SCF : The flag is set to 1.

Note that there is no SUB operation provided for 16 bit data. Therefore, it is not necessary to specify the A register when using the 8 bit SUB mnemonic.

Example: SUB 4 decrements the accumulator by 4.

Eight bit logical operations

All six types require one of the two data values to be placed in the accumulator prior to the operation. The result is also returned in the accumulator.

Three of the operations were introduced earlier in the chapter:

1. AND

2. OR

3. XOR

The other three are:

1. CP : Compare

2. CPL: One's complement

3. NEG: Two's complement

CPL simply inverts each bit in the accumulator. NEG also inverts each digit but increments the outcome to give the two's complement of the original value.

CP is unusual in that the data given is subtracted from that held by the accumulator. However, the result is not returned. Instead, the value in the accumulator is left unchanged, and several flags in the F register are updated:

1. The ZERO bit is set if the value in the accumulator is equal to the data specified. For example CP 4 will set the Z flag (zero) if the accumulator holds 4, otherwise the flag is cleared.

2. The sign bit (S) is set if the comparison data has a larger value than that held by the accumulator. Additionally, the N flag is set and the H, P/V and CY flags are updated.

Examples:

`AND 128` will logically `AND` the accumulator with 10000000.

`CPL` inverts each digit in the accumulator.

Rotate and Shift operations

If a binary value is shifted one column left, its value is doubled. This shifting or rotating of a byte is the only form of multiplication or division which the processor is capable.

The Z-80 has three shift and eight rotate operations, all of which operate only on eight bit data. When a byte has each bit shifted in one direction, one end bit is displaced and another is left vacant. The carry bit is used to replace and/or store the displaced or vacant bits depending on the instruction used.

The eight rotate operations consist of four instructions which only rotate the byte in the accumulator, and a duplicate set that perform the same operations on any register or memory location:

1. `RL` and `RLA`: Perform an identical operation albeit the RLA operation is specific to the accumulator. Each bit is moved one position left. The vacant l.s.b. is filled with the carry bit and the displaced m.s.b. is placed in the carry bit.

Examples:
```
RL B
RL (HL)
RLA
```

2. `RR` and `RRA`: As for `RL` and `RLA` except the rotation is to the right.

Examples:
```
RR H
RR (HL)
RRA
```

3. `RLC` and `RLCA`: Here the displaced m.s.b. is placed in both the carrry and the l.s.b.

Examples:
```
RLC (HL)
RLC A
RLCA
```

Note that both `RLC A` and `RLCA` perform the same operation. However, as with many operations, the result updates several bits in the Flag register. `RLCA` and `RLC A` do not update the same flags.

4. `RRC` and `RRCA`: As for `RLC` and `RLCA` except the rotation is to the right.

The three shift operations are:

1. SLA: Shift left arithmetic

Each bit is shifted left with the displaced m.s.b. placed in the carry and the l.s.b. set low.

Examples:
```
SLA A
SLA (HL)
```

2. SRL: Shift right logical

Each bit is shifted right with the l.s.b. placed in the carry and the m.s.b. set low.

Example:
```
SRL D
```

3. SRA: Shift right arithmetic

This is a true arithmetic shift in that the m.s. sign bit is not altered. The byte is shifted one position left with the displaced l.s.b. placed in the carry. The m.s.b. is unaltered.

Examples:
```
SRA A
SRA (HL)
```

In addition, there are two binary-coded decimal (B.C.D.) instructions RLD and RRD.

Bit Operations

There are two of these, SET and RES, which set or clear a specific bit in a given location.

Examples: SET 4,A sets to 1, bit 4 of the accumulator.

RES 6,(HL) resets to 0 bit 6 of the location held by register HL.

The BIT instruction is used to test a specific bit in a register or location. The Z flag is set to the complement of the target bit which is not affected.

Branch and Subroutine operations

Six of the bits in the flag register are used as flags, the remaining two bits

are vacant:

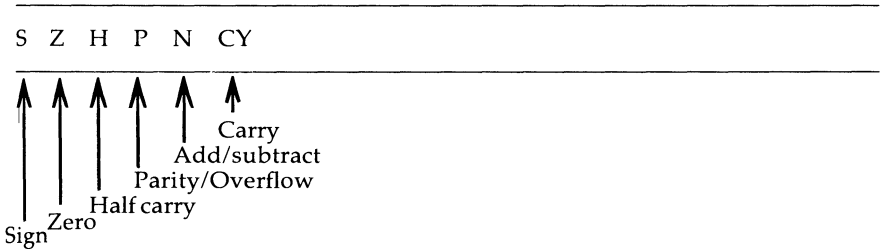


Figure 4.7 The Flag register

Many of the operations that the Z-80 will execute, set or reset certain flags according to the result.

The sign flag, S, is a copy of the most significant bit of a result. This is set when a negative value (two's complement) occurs. The zero flag is only set when a result is zero. If an operation is given by appendix D as one which updates the zero flag, then a non zero result will clear the flag.

The half carry and add/subtract flags are used in operations on B.C.D. values and are of little interest to the non-specialized programmer.

The parity/overflow flag is unusual in that it is set or reset by certain instructions according to one criteria whilst another set of instructions use an alternative criteria.

If the flag is updated according to the parity of a result, it is reset if the number of ones contained is odd, otherwise it is set. As an overflow flag, it is set if an addition or subtraction operation caused bit 7 to be changed erroniously. In signed binary, all negative values have the m.s.b. set. An out of range result may cause this bit to be incorrectly inverted. The overflow flag enables such a situation to be detected.

As we mentioned earlier, the flag register is used by the conditional jump instructions. If a particular flag is either set or clear then the jump is made, otherwise program execution continues with the next instruction.

The two principle jump instructions are JR (Jump Relative) and JP (JumP). Both may be conditional or forced. The JR instruction is accompanied by a displacement byte containing the number of bytes to be jumped from the address held by program counter. The displacement may be upto 127 locations forward or 128 locations back. In the case of a negative displacement, the two's complement form is used.

Examples: JP 40000 forces a jump to location 40000 by simply loading that address into the program counter. JR -120 updates the program counter to point to a location 120 bytes back from the instruction that would next have been executed.

The condition codes which may be used with either type of instruction are:

1. Z and NZ: JP Z,40000 would force a branch if the Z flag was set. JR NZ,20 would, if the zero flag was clear, force the relative jump.

2. C and NC: The branch is made if the carry flag is set or clear, respectively. The JR instruction may only use the Z, NZ, C and NC conditions.

3. PO and PE: The jump is made if the parity/overflow flag is set or clear, respectively. This condition may only be used with the JP instruction.

4. P and M : Again these may only be used by the JP instruction, and force the branch if the sign flag, S, is either clear or set, respectively.

Examples: JP P,22000 forces a jump to the location if the sign flag is clear. JP PO,12000 executes the jump if the overflow flag is set.

One final branch instruction is DJNZ which decrements the B register and makes the jump if it is not zero. For example DJNZ - 20.

The Z flag may be set or reset by using the BIT instruction. This tests a specific bit in a register or location and if it is clear, the Z flag is set, otherwise the flag is cleared. For example BIT 4,A will set the Z flag if bit 4 in the accumulator is clear.

Subroutines are implemented using the CALL and RET instructions. Both operations may be made conditional by the use of any of the conditions utilised by the JP instruction.

Examples: CALL 20000 forces a jump to the subroutine at 20000 until a RET instruction is encountered. The return address is kept on the stack. A conditional call takes the form CALL NZ,12000 and a conditional return: RET Z.

A variant of the CALL instruction is RST x where x is a multiple of eight in the inclusive range 0 to 56. This simply performs a subroutine call to location x.

For example RST 24 performs a call to location 24.

Block transfer and search operations.

There are four tranfer and four search instructions. The transfer instructions allow blocks of data to be moved with no effect on the accumulator:

1. LDIR : The first location of the block of data to be moved is placed in register pair HL and the number of bytes in the block in BC. Finally, before the

instruction is used, the destination address is placed in DE. As each byte is transferred, BC is decremented and HL and DE are incremented. When BC is equal to zero the transfer is terminated.

2. `LDDR` : Essentially the same as `LDIR` except that the addresses placed in HL and DE are decremented after each transfer.

3. `LDI` : As for `LDIR` except only one byte is transfered.

4. `LDD` : As for `LDDR` except only one byte is transfered.

Example: To copy a block of twenty bytes at 12000 to 14000:

```
LD HL,12000
LD BC,20
LD DE,14000
LDIR
```

This could also be achieved using the `LDDR` instruction by loading HL with 12019 and DE with 14019.

The search instructions check through a block of data until a match is made with the value in the accumulator:

1. `CPIR` : BC is loaded with the block length and HL with address of the first byte in the block. Finally, the comparison value is placed in the accumulator. The search terminates when BC is zero or when a match has occurred. The zero flag indicates which:

If set : A match was made between the accumulator and data byte.

If clear : BC had been decremented to zero.

2. `CPDR` : As for `CPIR` except the search procedes down in memory from the address in HL rather than upwards.

3. `CPI` : As for `CPIR` albeit only one byte is checked.

4. `CPD` : As for `CPDR` only one byte is checked.

Note that in all block move and transfer operations, the P/V flag indicates if the count has reached zero:

If BC=0 the P/V flag is cleared.

If BC<>0 the P/V flag is set.

Example: To search for a byte with value 4 in the 120 byte block from location 42000.

```
LD BC,120
LD HL,42000
LD A,4
CPIR
```

If on completion the zero flag is set then the HL register pair contains the address of the target byte.

CPU Control and I/O instructions

The CPU control group may be divided into four sections:

1. Exchange operations.

2. The NOP and HALT instructions.

3. Stack operations.

4. Interrupt and I/O operations.

1. Exchange operations.

There are two exchange instructions. EXX swaps the contents of the three register pairs BC,DE and HL with their alternate set equivalents BC',DE' and HL'. The second, EX, is used to exchange the value held by two sixteen bit registers. It has four variants:

i. EX AF,AF' .. exchanges the A and Flag registers with A' and F'.

ii. EX DE,HL .. simply swaps the value in the DE register pair with that in the HL pair.

iii. EX (SP),IY & EX (SP),IX .. exchange the index register with the two top bytes of the stack.

iv. EX (SP),HL .. as for iii. except that the HL register pair takes the place of the index register.

2. The NOP and HALT instructions

The NOP (NO oPeration) instruction does nothing for four T states. It is often used to either replace redundant code or to fill an area that may be required for future expansion.

The HALT operation causes the processor to continuously execute NOP's until the occurence of an interrupt or RESET. NOP's are performed to allow the Z-80

to continue to refresh dynamic RAM. The refresh register is automatically incremented after each instruction has been fetched from memory. The value held constitutes the low order byte of the section of memory to be refreshed.

3. Stack operations

All stack instructions operate on 16 bit data. The contents of any register pair may be either PUSHed onto the stack or loaded with the top two bytes POPped from the stack. The PUSH operation does not alter the contents of a register.

After the operation has completed, the stack pointer register holds the address of the byte on the top of the stack. Hence, if the stack pointer holds X, the high order byte of the last register pushed was placed in location X–1 and the low order byte in location X.

Examples:

```
PUSH BC,POP BC,PUSH IY,POP IY.
```

The stack pointer register may be manipulated directly using INC, DEC or LD.

Example: `INC SP:INC SP` will update the pointer two locations. As the stack moves down in memory, the top 16 bits of the stack are therefore lost.

4. Interrupt and I/O operations.

Interrupt operations: An external component may temporarily interrupt the processor's normal execution activities. This is achieved using any one of four of the Z-80's control lines:

1. BUSRQ

2. RESET

3. NMI

4. INT

1 - 3 are seldom, if ever, used by the programmer. BUSRQ, introduced earlier in the chapter, permits another processor to use the system address and data busses. RESET is used to initialise the system on power up. The I and R registers are set to zero prior to the commencement of program execution from location 0.

The Non Maskable Interrupt forces the processor to make a special subroutine call to location &H66. In the MSX configuration this location is allocated for use by the disc operating system and consequently the NMI will rarely be implemented.

The INT line may initiate any one of three sequences. The mode is selected by the programmer with the appropriate instruction: IM 0, IM 1 or IM 2.

In each case, the contents of the program counter register are saved too the stack and a jump made to a specified location. When a RETI (RETurn from Interrupt) instruction is encountered, the program counter is loaded with the return location and execution of the interrupted routine is re-commenced.

The INT input is checked by the Z-80 after each instruction executed. The input is level - not edge - sensitive. Thus the interrupting device must not continue to hold the line low, unless another interrupt is necessary.

The processor has two internal flip-flops which feature in the interrupt process: IFF1 and IFF2. If IFF1 is set to a logical 0, any INT request to the processor is ignored. If it is set to a logical 1, the maskable interrupts are enabled.

The programmer may set both the IFF1 and IFF2 states using the EI (Enable Interrrupt) or the DI (Disable Interrupt) instructions. IFF2 is used to store the state of IFF1 during a NMI, when IFF1 is set to disallow any INT requests.

The address of the routine executed is selected by different means for each of the three INT modes:

Mode 0 Interrupt : Enabled by the IM 0 instruction.

1. IFF1 and IFF2 are set to a logical 0. This negates any re- entrancy problems.

2. The Z-80 acknowledges the interrupt on the next clock pulse by taking two control lines low: IORQ and M1.

3. The component places a one byte opcode onto the data bus. This may be either a RST or a CALL instruction.

4. If a RST opcode is given, the contents of the program counter register are placed on the stack and a jump is made to the relevant zero page location.

5. After a call opcode, the component must place two further data bytes onto the data bus. The routine at that address is then executed after the PC register has been placed onto the stack.

6. When a RETI instruction is encountered, the two top data bytes on the stack are loaded back into the program counter. The RETI instruction does not set IFF1 to a logical 1 and interrupts must be re-enabled with the EI opcode.

Note that a RET instruction may also be used to force a return to the initial routine. However, certain components are capable of detecting that the Z-80 has fetched a RETI instruction from memory. This capability may be used to remove the initial interrupt request.

Mode 1 interrupt : Enabled by the IM 1 instruction.

The initialisation sequence performed by an MSX computer on power up selects this mode. The sequence executed after an enabled request has been made

is as follows:

1. The IFF1 and IFF2 flip-flops are set to a logical 0.

2. The contents of the PC are placed on the stack.

3. Program execution continues from the instruction at &H38.

4. When a RETI opcode is encountered, the top two bytes of the stack are pulled into the program counter, and execution of the interrupted routine re-commences.

Note that to re-enable maskable interrupts, IFF1 must be set to a logical 1 with the EI instruction.

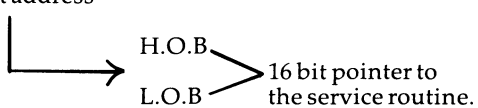
Mode 2 Interrupt : Enabled by the IM 2 instruction.

Mode 2 is the most flexible of the three INT modes. It allows a component to access upto 128 separate routines. After the interrupt is acknowledged by the processor, the initiating component places a single byte onto the data bus. This is joined to the byte in the I register to give a 16 bit address:

I Register byte / Component byte
b b b b b b b b b b b b b b b b

The two byte pointer at this address gives the location of the routine that is to be executed:

16 bit address



The sequence executed is:

1. IFF1 is set to a logical 0.

2. The Z-80 acknowledges the interrupt by asserting the IORQ and M1 lines.

3. The component places a data byte on the data bus.

4. The current execution address held by the progam counter is placed on the stack.

5. The routine given by the pointer at the computed location is executed.

6. When a RETI instruction is encountered, the top two bytes of the stack are pulled into the program counter register, and the interrupted routine is continued.

Input/Output operations.

This group has two sets of instructions for the input or output of data to or from an 8 bit port.

1. Single byte I/O:

The two mnemonics are `IN` and `OUT`. Each has two forms, the first may only use the A register and the second, any of the following: A,B,C,D,E,H or L.

If the accumulator specific form is used, the port must be given in brackets: `IN A,(port)`. The second form uses the C register to select the port: `OUT (C), register`.

We briefly discussed the 'mechanics' of the I/O operations earlier in the chapter. The Z-80 outputs the port number onto the least significant eight lines of the address bus prior to placing the data byte onto the data bus.

However, it is noted that both the `IN A,(port)` and `OUT (port),A` instructions also place the contents of the accumulator onto the address lines A15 to A8. Similarly, the `IN register,(C)` and `OUT(C),register` operations, place the contents of the B register onto the upper address lines. This allows for more than 256 ports to be utilized - if the relevant decoding hardware is available.

2. Block I/O instructions:

The block of data may contain from one to 256 bytes. This group contains four pairs of two instructions, and is similar in style to the block transfer and search groups:

i. `INIR` and `OTIR` : The initialisation data required is:

a. Port number in the C register.
b. First data location in HL.
c. Block size in the B register.

The HL register pair is incremented after each transfer. If the B register has been decremented to zero the Z flag is set.

ii. INDR and OTDR : As for INIR and OTIR except that HL is decremented after each transfer.

iii. INI and OUTI : As for INIR and OTIR except that only one transfer occurs. HL is incremented and the B register is decremented.

iv. IND and OUTD : As for INI and OUTI except that the HL register is decremented after each transfer.

Addressing Modes

As we have seen, each instruction mnemonic may specify the location of the

target data in one or more ways. Although the Z-80 has ten address modes, it is unnecessary for each type of instruction to have a variant for each mode. The reader will find the majority of formats - many of which have already been introduced - to be straightforward:

1. Implied : No address is necessary - it is implicit to the instruction.

Example: `NEG`

2. Immediate : The actual 8 bit data is given rather than an address.

Example: `LD A,2`

3. Extended immediate : The actual 16 bit data is given rather than an address.

Example: `LD HL,20000`

4. Relative : This mode is only used by the jump relative and DJNZ instructions. The byte following the instruction contains a signed value in the inclusive range –128 to 127. This displacement is added to the address in the program counter to give the new execution address.

Example `JR 20`

5. Register : The target data is held by the specified register.

Example `LD A,B`

6. Register indirect : An index register or register pair holds the address of the target data. The register or register pair is given in brackets.

Example: `LD A,(DE)`

7. Extended : The address of the data is given in brackets.

Example: `LD A,(4000)`

8. Indexed : Essentially the register indirect form with a displacement to be added to the address held by the index register.

Example: `LD A,(IY+5)`

9. Modified page zero : Only the RST instructions use this format. A call is made to the location specified. This must be a multiple of eight in the inclusive range 0 to 56

Example: `RST 8`

10. Bit : The bit mode is unusual in that it specifies a bit and not a byte. The bit mode is used by three types of instruction: `BIT`, `SET` and `RES`. The

bits are numbered as follows:

```
X   X   X   X   X   X   X   X
7   6   5   4   3   2   1   Ø
```

Example: `SET 4,A`

The stage has now been reached when the readers may wish to try a few of their own routines. The following guidelines may be helpful:

1. Always save a machine code program before a trial run. If it contains a bug, the machine will probably hang up and need to be switched off before it may be re-used.

2. A routine is unlikely to operate correctly straight away. If you are lucky, one or two sections may perform as expected. Patience is the keyword.

3. If combining a M/C routine with a BASIC program, it is easier to use a BASIC loop to poke the code into position, rather than loading & saving two programmes separately.

If values are to be passed between the two, then unless the resident F.P. routines are to be utilized, peeks and pokes are to be used in preference to the USR function.

4. Generally, M/C routines require more debugging than their BASIC counterparts, and it is sensible to structure the program into small sections, each of which is called by a core program.

Chapter 5

The MSX Configuration

The basic MSX specification calls for a Z80 CPU (or equivalent), clocked at 3.5 MHz, supported by a Texas Instruments TMS-9929A Video Display Processor, (TMS-9918A in America and Japan) and a General Instuments AY-3-8910 Sound Chip.

In addition to this the specification also calls for an Intel 8255 Programmable Peripheral Interface, which handles keyboard scanning and manages the memory paging system. The specification is completed by 32 Kbytes of MSX BASIC and operating system ROM and by a minimum of 8 Kbytes of user RAM (although for the European market it would seem very unlikely that any machine will be released with less than 32 Kbytes of RAM). Further the VDP is provided with 16 Kbytes of dedicated Video RAM which it uses for storage of the screen memory, colour, and sprite definitions. The usage of this 'VRAM' will be fully discussed in chapter 6.

MSX Memory Management

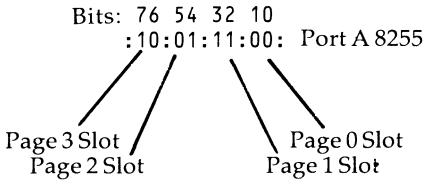
Central to the understanding of the MSX systems memory management is the concept of 'slots' which I shall now briefly describe.

Since the Z80 is only an eight bit processor its maximum address space is only 64 Kbytes; however the MSX system is designed to be able to access up to 1024 Kbytes and so the system of 'slots' is used.

A 'slot' is one set of 64Kbytes address space of which the MSX system may address four. Each of these four 'primary slots' may be further expanded into four 'secondary slots' thus giving a total addressable space of 4*4*64 =1024 Kbytes. Each slot (primary or secondary) is further subdivided into four 16Kbyte pages which are the basic units that the MSX system uses to create its address space. Whatever slot they may happen to be in the four pages start at addresses 0000H, 4000H, 8000H, and 0C000H. Pages may only ever be addressed at their respective page addresses (page 0 at 0000H, page 1 at 4000H et seq.). Thus the MSX system may select any page from any slot but must always select page 0 to be addressed from 0000H, and page 3 to be addressed from 0C000H.

Primary slot selection is performed by writing to port A of the 8255 PPI in

the following format;



Since each slot allocation is allowed two bits of the register it follows that each allocation may vary numerically between 0 and 3 (00B and 11B) thus allowing selection of any of the four primary slots for each of the four pages. Selection of the secondary slot is performed by writing, in the same format, to location 0FFFFH of the primary slot. When reading this register the value returned is the complement of its actual value.

On power up or Cold Start the MSX system (which is resident in slot 0 and is therefore the first slot read by the CPU) searches for the largest contiguous area of RAM from 0FFFFH downwards and selects tht area for pages 3 and 2. Under normal circumstances therefore it will not be necessary for the programmer to perform slot selection, except in the case of a 64 Kbyte or greater RAM system where RAM will be mapped over ROM, or where cartridge based software is being written to run in slots other than 0.

Accessing the Sound Chip, VDP and PPI

The Z80 in addition to being able to address 64 Kbytes of memory is also provided with the ability to address 256 i/o ports. These are used to address the VDP etc. (including the RS232 and other system expansions). However, in order to allow hardware modifications, BIOS (Basic Input/Output System) calls have been provided in the MSX system ROM. These allow access to the MSX system chips in such a way that if in one MSX unit one of the chips occupies a different position this may be compensated for in firmware. The difference is transparent to the programmer and ensures complete software compatibility. The only exception to this rule is the VDP which will be fully discussed later.

The VDP a General Introduction

The Texas Instruments TMS-9929A Video Display Processor is a microprocessor in its own right. It is responsible for all screen handling and accesses 16 Kbytes of dedicated Video RAM. This RAM holds all character definitions, the screen data, colour data, sprite definitions aand sprite attributes. This frees system

RAM for use in programming (the penalty paid being a reduction in the speed at which Video RAM may be accessed). Under normal circumstances, howeve, this reduction is not critical. In critical situations various tricks, which are discussed in chapter 6 may be used.

The VDP has four display modes: Graphics I, Graphics II, Multicolour and Text mode. The text mode provides twenty-four 40-character rows in two colours and was designed to maximise the text display capabilities of the domestic television. The multicolour mode provides an unrestricted 64*48 colour dot display using fifteen colours plus transparent. Graphics I mode provides a 256*192 pixel display for generating pattern graphics in 15 colours plus transparent. Graphics II mode is an enhancement of Graphics I mode allowing more complex colour and pattern displays.

The video display consists of 34 planes stacked vertically. The innermost plane is the backdrop plane, the next innermost is the pattern plane (containing Graphics I and II patterns) and over these lie the 32 sprite planes.

In addition to the video handling the VDP also keeps user RAM refeshed and provides an Interrupt Request every 20ms. For more complete details of the VDP and its programming refer to chapter 6.

The General Instruments AY-3-8910 Sound Chip

The GI AY-3-8910 is a three voice, eight octave sound chip incorporating a simple hardwaare volume envelope and the ability to mix white noise into any or all of the three voices. The chip also incorporates 2 i/o ports which are used by the system to read the joysticks, games paddles, and touchpads. For a description of programming this chip refer to chapter 7.

The Intel 8255 PPI

The 8255 PPI is a very powerful general purpose i/o chip used in the MSX system for several purposes. It is responsible for handling the keyboard (including any extension keyboards), the caps lamp, various cassette i/o functions, the memory management system and provides a 1 bit sound port.

A more complete description of this chip and its varying uses follows in chapter 8.

Interrupt Handling and 'RAM Hooks'

Many MSX ROM routines are indirected through RAM by means of 'hooks'. In the MSX system a 'Hook' consists of 5 bytes of RAM initilised to contain a Z80 'RET' instruction (0C9H). Indirection is performed by the relevant ROM routine by means of a Z80 'CALL' to the hook. It is therefore a simple matter to redirect the CALL by inserting a Z80 'JP nnnn' into the first three bytes of the hook.

The MSX system runs under interrupt mode 1 and provides two hooks for handling IRQs.

Firstly a hook is provided at 0FD9AH for handling IRQs which originate in devices other than the VDP timer. In the basic MSX system this hook is of little use, but is provided for later eexpansion. Secondly the hook at 0FD9FH is provided to handle interrupts generated at frame flyback by the VDP. This is the only interrupt available in the basic system. The VDP interrupt also handles the keyboard scan and various other operating system functions. It is essential that any user routine return to the calling operating system routine in order to cleanly handle the interrupt.

The operating system pushes all registers (including the alternate set) onto the stack before calling the hook at 0FD9FH, so all registers may be used with impunity. However the operating system also fetches the contents of the VDP status register in the accumulator before calling the hook and so it is essential that any user routine not modify the AF pair. On return from the hook the operating system stores the value of the accumulator in location 0F3E7H and it is preferable to read this location, rather than the VDP status register direct, when checking any of the VDP status bits in any user routine which is not interrupt driven.

In addition to these hooks the system also provides a hook at 0FDD6H for handling NMIs. This is of little readily apparent use since the basic system provides no NMI and in the disc system the NMI entry vector at 66H is occupied by File Control Block data for the Disc Operating System.

In user applications NMI handling may be necessary and so this hook is provided for flexibility.

The following program will set up an interrupt driven real-time clock and is provided as an example of user IRQ handling.

Example Program Real Time Clock

```
WRTVDP EQU 47H      ;TABLE OF BIOS CALLS
RDVRM  EQU 4AH
WRTVRM EQU 4DH
FILVRM EQU 56H
LDIRVM EQU 5CH
LDIRMV EQU 59H
CHGET  EQU 9FH
CHPUT  EQU ØA2H
GTSTCK EQU ØD5H
GTTRIG EQU ØD8H
RDVDP  EQU 13EH
SNSMAT EQU 141H     ;END OF BIOS TABLE
VDPRGS EQU ØF3DFH   ;MSX SAVE AREA FOR VDP REGISTERS
BASNOS EQU ØF3B3H   ;START LOCATIONS FOR VDP TABLES
INTHOK EQU ØFD9FH
```

```
        ORG ØE000H

START:  LD HL,MESS          ;SET HL TO POINT TO INPUT PROMPT
MPLP:   LD A,(HL)
        CP '$'
        JR Z,GETIN
        CALL CHPUT          ; PRINT PROMPT
        INC HL
        JR MPLP
GETIN:  LD HL,HHMMSS
        LD B,3
GTINOL: CALL GETNUM         ;GET 2 NIBBLE NUMERIC
        LD (HL),A           ;STORE IN HHMMSS
GTINIL: CALL CHGET          ;GET CHARACTER
        CP ':'              ;IS IT THE SEPARATOR
        JR NZ,GTINIL        ;IF NOT TRY AGAIN
        INC HL              ;POINT TO NEXT IN HHMMSS
        DJNZ GTINOL         ;DO IT THREE TIMES
        LD A,LOW CLOCK      ;SET UP...
        LD (INTHOK+1),A     ;JP ADDRESS
        LD A,HIGH CLOCK
        LD (INTHOK+2),A
        LD A,ØC3H           ;=Z8Ø 'JP'
        LD(INTHOK),A
        RET                 ;TO BASIC, HAVING REDIRECTED THE HOOK

GETNUM: CALL CHGET          ;GET CHARACTER
        CP 'Ø'
        JR C,GETNUM
        CP ':'              ;IS IT NUMERIC?
        CALL CHPUT          ;PRINT IT
        SUB 'Ø'             ;CONVERT TO NUMERIC
        SLA A
        SLA A
        SLA A               ;MULTIPLY BY 16
        SLA A
        LD B,A              ;PRESERVEE IT IN B
GTNUM1: CALL CHGET          ;GET CHARACTER
        CP 'Ø'
        JR C,GTNUM1
        CP ':'              ;IS IT NUMERIC?
        JR NC,GTNUM1        ;IF NOT TRY AGAIN
        CALL CHPUT          ;PRINT IT
        SUB 'Ø'             ;CONVERT NUMERIC
        ADD A,B             ;ADD IT TO B
        RET                 ;RETURN WITH NUMBER IN A
MESS:   DEFB 'ENTER TIME HH:MM:SS:$'
HHMMSS: DEFB Ø,Ø,Ø
CTR:    DEFB Ø
CLOCK:  PUSH AF             ;PRESERVE VDP STATUS
        LD A,(CTR)
        DEC A
```

```
        LD (CTR),A          ;ONLY DO CLOCK EVERY 50...
        JP P,OUT            ;INTERRUPTS
        LD A,49
        LD (CTR),A          ;RESET COUNTER
        LD IX,HHMMSS
        LD A,(IX+2)
        INC A               ;INC SECONDS
        DAA
        LD (IX+2),A
        CP 60H              ;IF NECESSARY...
        JR NZ,CLKPNT
        XOR A               ;RESET SECONDS
        LD (IX+2),A
        LD A,(IX+1)
        INC A               ;AND INC MINUTES
        DAA
        LD (IX+1),A
        CP 60H              ;IF NECESSARY...
        JR NZ,CLKPNT
        XOR A               ;RESET MINUTES
        LD (IX+1),A
        LD A,(IX+0)
        INC A               ;AND INC HOURS
        DAA
        LD (IX+0),A
        CP 24H
        JR NZ,CLKPNT
        XOR A
        LD (IX+0),A
CLKPNT: LD HL,VDPRGS        ;ROUTINE TO PRINT UPDATED...
        LD A,(HL)           ;CLOCK.
        LD DE,10            ;READ SYSTEM RAM COPIES...
        AND 2               ;OF VDP WRITE ONLY...
        JR NZ,GNAMTB        ;REGISTERS.
        INC HL              ;TO FIND SCREEN MODE...
        LD A,(HL)           ;AND CALCULATE OFFSET...
        AND 8               ;FROM START OF BASE(N)...
        JP NZ,OUT           ;VARIABLE TABLE.
        LD A,(HL)           ;TO FIND BASE ADDRESS
        AND 16              ;OF NAME TABLE.
        LD DE,0
        JR NZ,NGAMTB
        LD DE,5
GNAMTB: LD IX,BASNOS
        ADD IX,DE
        LD L,(IX+0)         ;GET NAME TABLE ADDRESS
        LD H,(IX+1)
        LD DE,24            ;AND ADD 24(OFFSET)
        ADD HL,DE
        LD IX,HHMMSS        ;GET ADDRESS OF TIME
        LD A,(IX+0)         ;GET FIRST VALUE (HOURS)
        CALL NPNT           ;PRINT IT
```

```
        LD A,':'            ;GET COLON
        CALL WRTVRM         ;PRINT THAT
        INC HL              ;INC PRINT POSITION
        LD A,(IX+1)         ;DO MINUTES
        CALL NPNT
        LD A,':'
        CALL WRTVRM
        INC HL
        LD A,(IX+2)         ;DO SECONDS
        CALL NPNT
OUT:    POP AF              ;RESTORE VDP STATUS...
        RET                 ;AND RETURN TO OS ROUTINE
NPNT:   PUSH AF             ;SAVE NUMERIC
        SRL A               ;GET TOP NIBBLE
        SRL A
        SRL A
        SRL A
        ADD A,30H           ;CONVERT ASCII
        CALL WRTVRM         ;PRINT IT
        INC HL              ;INC PRINT POSITION
        POP AF              ;RESTORE NUMERIC
        AND 15              ;GET LOW NIBBLE
        ADD A,30H           ;CONVERT ASCII
        CALL WRTVRM         ;PRINT IT
        INC HL              ;INC PRINT POSITION
        RET



        END
```

MSX System RAM Usage

In the MSX system the locations from 0F380H to 0FFFFH are used by BASIC and the operating system for various house-keeping functions. Some of those locations most useful to the assembly language programmer are described below.

Interslot read/write and call routines are provided at the start of system RAM.

At 0F380H is a routine for reading from any primary slot. This accepts the value for slot selection (in the format required by the 8255) in the accumulator, the old slot status in D and the address in the HL register pair. The routine returns the value read in the E register, leaving all other registers preserved.

The counterpart to this routine at 0F385H expects the same setup but performs a write operation using the data in E.

Finally a routine at 0F38CH performs interslot calls. This routine expects to find the old slot status on the stack, the new slot status in the accumulator, any values to be passed in the alternate AF pair and the location to call in IX. Any values to be returned should be passed in the alternate AF pair.

The system stores call addresses for the BASIC `USR` function in locations 0F39AH to 0F3ADH in the order `USRØ` to `USR9`, allowing two bytes for each. Locations 0F3B3H to 0F3D9H contain the values accessed by the BASIC `BASE(N)` pseudo-variable,two bytes being allowed for each value.

The key click toggle is stored at 0F3DBH. Writing a zero to this location disables the key click, while any non-zero value will enable it.

The x,y position of the screen cursor is stored at locations 0F3DCH (Y) and 0F3DDH (X).

The eight locations from 0F3DFH onwards are used by the operating system to store the present values of the eight VDP write only registers. The present value of the VDP status register is stored during every vertical blanking interrupt in location 0F3E7H.

Locations 0F3E9H to 0F3EBH are used to store the current foreground, background and border colours as set by the BASIC `COLOR` command.

The system stores the address of the highest location it can find in RAM at addresses 0F672H and 0F673H. These locations may be altered in order to conceal areas of RAM from BASIC and the operating system, thus securing any user code stored in these areas. The two bytes following these locations define the highest location to be used by the stack.

The 26 locations starting at 0F6CAH are used to store the current default variable types for variables beginning with A to Z. This table is modified by `DEFINT`, `DEFSTR`, `DEFSNG` etc. and is read when any variable is encountered that is not accompanied by a postfixed variable type declaration.(ie. $, %, !). Variable types are;

Integer	:table value is 2
String	:table value is 3
Single Precision	:table value is 4
Double Precision	:table value is 8

Please note that table entries will always default to double precision.

Finally, from 0FD9AH to 0FFCAH is the block of RAM indirection 'hooks' each of which consists of five bytes initialised to contain Z80 '`RET`' instructions (0C9H). Most of these are set aside for future system expansion and are of little use to the general programmer. Those which are of immediate use will be discussed as they arise.

Using Machine Code Subroutines From BASIC

The key to successfully interfacing machine code routines with BASIC lies in two areas: firstly hiding the routine to avoid BASIC overwriting the code and secondly passing parameters to and reading results from the routine.

The simplest method of assigning space for code routines is by use of the BASIC `CLEAR n1,n2` statement. The first parameter of this sets the amount of space for string storage and the second sets the highest memory location to be used by BASIC. Thus to free the top 16 Kbytes of RAM for machine code routines and data and to set up 200 bytes of string space the command `CLEAR 200,&HBFFF` should be given. Note however that under normal circumstances the system work area will remain from 0F380H upwards.

Calling machine code routines from BASIC requires the BASIC USRn function. The syntax of this statement is:

```
Var= USRn(Var/Const), or

PRINT USRn(Var/Const)
```

Where n=user routine number set by `DEFUSR` and Var or Var/Const may be of any type.

Thus if we wish to pass a string to a routine which is intended to return an integer value we may call with:

```
A%=USR1("abcde")
```

The `USR` call passes values in the following manner:

Integer: location 0F663H contains 2, and the value is stored in 0F7F8H and 0F7F9H (low byte first).

String: location 0F663H contains 3, and 0F7F8H and 0F7F9H contain the address of the string descriptor. A string descriptor consists of three bytes, the length of the string followed by its address.

Single precision: location 0F663H contains 4, and the value is stored from 0F7F6H to 0F7F9H.

Double precision: location 0F663H contains 8, and the value is stored from 0F7F6H to 0F7FDH.

Parameters may be returned to BASIC in a directly analogous manner. The following program illustrates this by accepting a numeric string in base three, and returning the decimal value, as an integer.

```
VARTYP  EQU  ØF663H
VARPTR  EQU  ØF7F8H
```

```
        ORG 09000H


START:  LD A,(VARTYP)
        CP 3                ;IS VAR A STRING?
        RET NZ              ;RETURN IF NOT
        LD HL,(VARPTR)      ;GET ADDR OF DESCRIPTOR
        LD B,(HL)           ;LEN OF STRING IN B
        INC HL

        LD E,(HL)           ;ADDR OF STRING IN DE
        INC HL
        LD D,(HL)
        LD HL,0             ;CLEAR HL
ICLP:   PUSH DE
        PUSH HL             ;SAVE REGS
        ADD HL,HL
        POP DE
        ADD HL,DE           ;BINVAL=BINVAL*3
        POP DE              ;GET CURR STRING PTR
        EX DE,HL            ;INTO HL AND BINVAL IN DE
        LD A,(HL)           ;GET NEXT CHARACTER FROM STRING
        SUB '0'             ;CONVERT NUMERIC
        INC HL              ;INC STRING PTR
        PUSH HL             ;SAVE IT
        LD L,A
        LD H,0              ;PUT NUM IN HL
        ADD HL,DE           ;BINVAL=BINVAL+NUM
        POP DE              ;STRING PTR IN DE
        DJNZ ICLP           ;IF MORE STRING DO IT AGAIN
        LD (VARPTR),HL      ;BINVAL IN VARPTR
        LD A,2
        LD (VARTYP),A       ;VARTYP=INTEGER
        RET                 ;TO BASIC


        END
```

Chapter 6

The Video Display Processor

The VDP interfaces with the CPU via an eight bit bidirectional data bus, three control lines and an interrupt. Four operations may be performed: writing data to VRAM, reading data from VRAM, writing to the VDP control registers and reading from the VDP status register. Each of these operations requires one or more data transfers over the VDP/CPU data bus interface with the interpretation of these transfers being dependant on the status of the three control lines. The CPU may communicate with the VDP asynchronously with the television raster scan, the VDP allowing access to VRAM at times even during a raster scan period.

The Control Lines

The three control lines (CSW, CSR and MODE) control the VDP's interpretation of data transfers. CSW is active (low) when a CPU to VDP data transfer is to take place. CSR is active (low) when a CPU read from the VDP is to be effected. CSW and CSR should never be active simultaneously.

MODE determines the source or destination of a data read or write operation, and in the MSX system is usually tied to CPU address line 0. Refer to figure 6.1 for a summary of CPU/VDP interface operations. These operations are described below.

CPU write to VDP register
The VDP has eight write only control registers and a read only status register.

Each of the eight write only registers may be loaded by only two eight bit data transfers from the CPU (the control line status necessary for this and all other operations may be found in Table 6.1). The first byte to be written is the data to transfer, and the second byte determines the destination register (0-7 decimal, with the most significant bit set to distinguish the operation as a register write as opposed to a VRAM address setup operation).

To rewrite data in internal registers after a data byte has already been transferred it is necessary to read the VDP status register in order to re-initialise the interface logic. This situation is often encountered in an interrupt driven environment such as the MSX system. In general, whenever the status of VDP read/write parameters is in doubt, this procedure should be used.

Table 6.1 CPU/VDP Data Transfers

OPERATION	BIT 7	6	5	4	3	2	1	0	$\overline{CSW}$	$\overline{CSR}$	MODE
Write to VDP register											
Byte 1: Data write	D7	D6	D5	D4	D3	D2	D1	D0	0	1	1
Byte 2: Register select	1	0	0	0	0	RS2	RS1	RS0	0	1	1
Write to VRAM											
Byte 1: Address setup	A7	A6	A5	A4	A3	A2	A1	A0	0	1	1
Byte 2: Address setup	0	1	A13	A12	A11	A10	A9	A8	0	1	1
Byte 3: Data write	D7	D6	D5	D4	D3	D2	D1	D0	0	1	0
Read VDP register											
Byte 1: Data read	D7	D6	D5	D4	D3	D2	D1	D0	1	0	1
Read VRAM											
Byte 1: Address setup	A7	A6	A5	A4	A3	A2	A1	A0	0	1	1
Byte 2: Address setup	0	0	A13	A12	A11	A10	A9	A8	0	1	1
Byte 3: Data read	D7	D6	D5	D4	D3	D2	D1	D0	1	0	0

CPU write to VRAM
CPU to VRAM data transfers use a fourteen bit auto- incrementing address register. The first two byte transfers of write to VRAM are used to set up this register and sequential writes thereafter require only one byte transfer, since this register has already been set up and increments after every write operation.

CPU read from VDP status register
The CPU may read the VDP status register using only one byte transfer. MODE is active (high) for the transfer and CSR is used to signal that a read operation is required.

CPU read from VRAM
The CPU reads from VRAM in a directly analogous way to that in which it writes to VRAM, using the auto-incrementing address register.

Timing
Since the CPU interacts with VRAM through the VDP it is obvious that data transfers may only occur when the VDP is not busy with handling video output; thus the time necessary for the CPU to transfer a byte of data to or from VRAM varies between 2 and 8 microseconds, dependant on whether the VDP is busy with memory refresh or screen display. Approximate timings are given in table 6.2.

The VDP registers

Of the eight write only registers numbers 0 and 1 contain flags controlling various VDP functions and modes. Registers 2 to 6 contain values specifying start addresses for the various sub-blocks of VRAM, and register7 is used to define backdrop colour in all modes and text colours in the 40 column mode. Detailed descriptions of these registers now follow.

Register 0
The two least significant bits of this register contain control bits. All other bits are reserved for future expansion and must be reset.

Bit 0: External VDP enable/disable (1=enable).

Bit 1: Mode bit 3, see below for details.

Register 1 (8 VDP control bits)
Bit 7: 4/16K RAM selection.(always 1 in the MSX system.)

Bit 6: Blank enable/disable (1=enable active display). Blanking causes the screen to blank to backdrop colour.

Bit 5: Interrupt enable (1=enable, 0=disable).

Bit 4: Mode bit 1.

Table 6.2. VRAM access timings

CONDITION	*MODE*	*VDP DELAY*	*TIME WAITING FOR AN ACCESS WINDOW*	*TOTAL TIME*
Active display	Text	2 us	0- 1.1 μs	2- 3.1 μs
Active display	Graphics I, 11	2 μs	0- 5.95 μs	2- 8 μs
Active display	Multicolour	2 μs	0- 1.5 μs	2- 3.5 μs
Register 1 blank bit 0	All 2	μs	0 μs	2 μs
4300 μs after interrupt	All	2 μs	0 μs	2 μs

Bit 3: Mode bit 2, with mode bit three in register 0 these bits define the screen mode, according to the following table;

M1	*M2*	*M3*	
0	*0*	*0*	*GRAPHICS I MODE*
0	0	1	*GRAPHICS II MODE*
0	1	0	*MULTICOLOUR MODE*
1	0	0	*TEXT MODE*

Bit 2: Reserved for future expansion, must be 0.

Bit 1: Sprite size select: 0 selects 8*8 sprites, 1 selects 16*16 sprites.

Bit 0: Selects the magnification option for sprites: 0 selects normal sprites, 1 selects magnified spites.

Register 2
The least significant four bits of register 2 form the upper four bits of the fourteen bit Name Table address; thus the name table start address is equal to (Register 2)*400H.

Register 3
Register 3 defines the start address in VRAM of the colour table. The contents of this register form the upper eight bits of the fourteen bit address. Thus the address may be calculated by (Register 3)*40H.

Register 4
The least significant three bits of register 4 define the start of the Pattern, Text or Multicolour Generator sub-block, forming the top three bits of that address. Thus the generator sub-block address is given by (Register 4)*800H

Register 5
The lower seven bits of this register form the upper seven bits of the Sprite Attribute table; thus the base address is equal to (Register 5)*80H.

Register 6
Register 6 defines the start address in VRAM of the Sprite Pattern generator table. The address is equal to (Register 6)*800H.

Register 7
The most significant four bits of register 7 contain the colour code for colour 1 in text mode and the lower four bits form the colour code for colour 0 in text mode and the backdrop in all modes.

The Status Register
The VDP has a single eight bit status register which may be accessed by the

CPU. This register contains the interrupt flag, the sprite collision flag, the fifth sprite flag and the fifth sprite number (should one exist). The format of the status register is described below.

The status register may be read at any time. However, reading the status register asynchronously with the frame flyback interrupt will reset the frame flag and may cause interrupts to be missed. To avoid this problem it is advisable to read the copy of this register that the operating system keeps in system RAM (see chapter 5).

Bit 7: The Interrupt Flag.
This flag is set to 1 at the end of the raster scan of the last line of the active display. Setting this flag will also cause the interrupt pin to go active if the interrupt enable flag in register 0 is set. The flag is reset by reading the status register; please note that it is necessary to read this register at each frame flyback in order to reset the interrupt flag and re-enable it for the next frame.

Bit 5: The Sprite Coincidence Flag
This flag is set whenever two sprites are in collision (ie. when they have one or more overlapping pixels). Transparent sprites are considered as are those which are partially or completely off the screen. Sprites beyond the sprite attribute table terminator (0D0H), however, are not considered. The flag is cleared when the status register is read.

Bit 6: The Fifth Sprite Flag.
Bit 6 of the status register is set whenever there are five or more sprites on a horizontal line. It is cleared whenever the status register is read. Whenever the flag is set the number of the offending fifth sprite is placed in the lower five bits of the status register, allowing the user to move the sprite before the next frame to ensure that all sprites are properly displayed.

Video Display Modes.

The VDP generates an image which may be considered as a number of planes sandwiched together. The order of priority of these planes is from front to back, thus if two objects occur in the same position on different planes the object on the higher priority plane will occlude objects on any lower priority plane. It may be seen therefore that a sprite on plane 0 will appear to be in front of the rest of the display. From front to back the planes are: The 32 sprite planes, followed by the pattern plane, followed by the backdrop plane, followed by the planes of any external VDP which is connected. I know however of no intentions of the MSX companies to introduce machines with more than one VDP.

The backdrop plane is a solid colour used to display the border areas and is the default colour for the background of the pattern plane. When switched to transparent it automatically defaults to black unless an external VDP is active.

The 32 sprite planes lie on top of the pattern plane, sprite 31 having lowest priority and sprite 0 the highest. These are inactive in text mode. Sprites are transparent by default and their positions may be defined pixel by pixel thus

allowing smooth movement. The appearance and storage of the pattern plane tends to differ in the various screen modes and will therefore be discussed separately.

Fifteen colours are available in all modes, as follows:-

CODE	COLOUR
0	Transparent
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

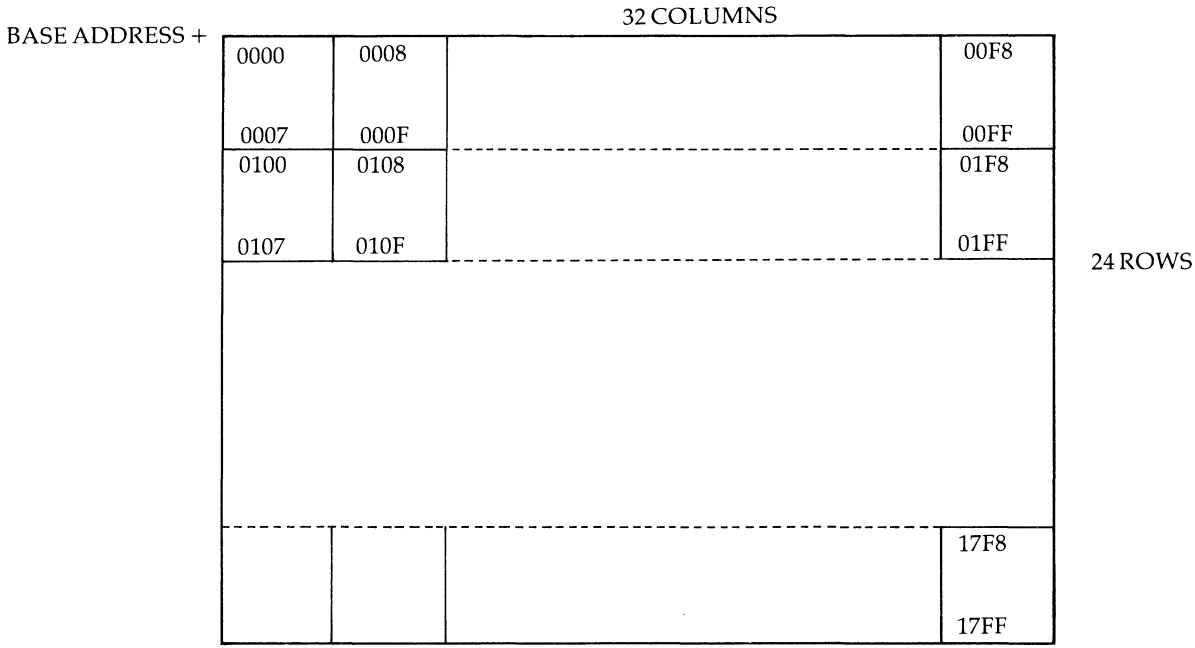
Graphics Mode I

In Graphics Mode I the Name Table sub-block of VRAM is comprised of 768 bytes, arranged as 24 rows of 32 characters. Thus the first 32 values in the table map to the first row of the display, the next 32 map to the second row and so on. Each entry in the table refers to 1 of 256 pattern definitions stored in the pattern generator table. Each of the 256 entries in this table consists of eight bytes thus giving 8*8 bits, and requires 2048 bytes of VRAM in total. In each eight byte entry in the pattern generator table a set bit is displayed as colour 1, and a reset bit is displayed as colour 0. The colours for each pattern are taken from the colour table (which in Graphics Mode I is 32 bytes long). Each entry defines a foreground (colour 1) and background (colour 0) colour for eight patterns in the pattern generator table. The first entry in the colour table defines the colours for patterns 0 to 7 in the pattern generator table, the second entry defines patterns 8 to 15 and so on. The entries in the colour table are considered as two nibbles, the most significant four bits defining the foreground colour and the least significant nibble defining the background colour. From the above it may be seen that the full implementation of this mode requires 2848 bytes of VRAM however if the full 256 pattern definitions are not required it is possible to overlap the tables.

Graphics Mode II

In Graphics Mode II the name table is mapped in the same manner as in Graphics Mode I, except that each entry in the name table may have its own

Figure 6.1 Graphics II bit mapped screen map



unique definition in the pattern generator table. This is achieved by splitting the pattern generator table into three blocks of 256 definitions; entries in the first 256 bytes of the name table (ie. the top third of the display) refer to definitions 0 to 255 in the pattern generator table, and values in the second and third 256 byte blocks of the name table refer to definitions 256 to 511 and 512 to 767 respectively.

It is also possible to define foreground and background colours for each horizontal line of each pattern definition. The colour table consists of 768 eight byte entries. Each of these bytes corresponds to one byte of the pattern generator table. The most significant four bits of each byte of the colour table defines the foreground colour of the corresponding byte in the pattern generator table and the least significant bits the background colour.

It can be seen that by careful arrangement of values in the name table Graphics Mode II can be considered to be bit-mapped; or by copying the same values into the three blocks of the pattern generator and colour tables it may be used in the same manner as the Graphics I Mode but with better colour definition.

Multicolour Mode.

The Multicolour Mode provides an unrestricted 64*48 colour square display, each colour square containing a 4*4 block of pixels. Each colour square may be in any of the 15 colours available in the VDP.

The Multicolour Mode name table is mapped similarly to the two graphics modes, although the name no longer points to an entry in the colour table. Colour is now derived from the entries in the pattern generator table.

Each value in the name table points to an eight byte block in the pattern generator table. Only two bytes of this block are used to define the screen image. These two bytes, considered as four nibbles, define an 8*8 pixel multicolour pattern. The high order nibble of the first byte defines the colour of the 4*4 pixel square in the top left of the multicolour pattern, the lower nibble defines the colour of the square in the top right and the second byte performs the same function for the colour squares in the bottom left and right corners of the multicolour pattern. Entries on rows 0,4,8,12,16 and 20 of the name table use the first two bytes of the eight byte block in the pattern generator table, entries on rows 1,5,9,13,17 and 21 use the second two bytes and so on.

Text Mode

In text mode the screen is considered as a grid 40 characters across by 24 lines down. Each entry in this grid is 6 pixels across by 8 pixels down. The tables used to generate the pattern plane are the name table and the pattern generator table and there may be up to 256 different patterns defined at any one time. The name table maps the definitions into each of the 960 pattern cells on the pattern plane. Sprites are not availble in this mode. The pattern generator table is identical to that in Graphics Mode I except that since each pattern cell is only six pixels across the two least significant bits of each byte

in the pattern generator table are ignored. The foreground and background colours are set by the value of VDP register 7, the high nibble defining the foreground colour and the low nibble defining the background and border colours.

Sprites

The video display may have up to 32 sprites on the highest priority planes. Their positions are defined from the top left hand corner of the sprite pattern, and since this position may be defined pixel by pixel it is easy to swiftly and smoothly move these patterns around. Each of the sprite planes is fully transparent outside of the sprite itself. The sprites are not active in the 40 column text mode.

The tables in VRAM which are relevant to sprite usage are the sprite attribute table and the sprite pattern generator table. These tables are similar to their equivalents on the pattern plane since the sprite attribute table defines the location of the sprite, while the sprite pattern table contains a definition of what it looks like.

Each of the 32 entries in the sprite attribute table consists of four bytes. The number of pixels down the screen of its top left hand corner, (its vertical position) followed by its horizontal position (the number of pixels from the left hand edge of the screen), followed by its name (i.e. its definition in the sprite pattern generator table), and finally its 'tag', the least significant four bits of which define its colour, and the most significant bit of which is the 'early clock' bit. This causes the sprite to appear 32 pixels to the left of its position as defined by byte two of the entry in the sprite attribute table.

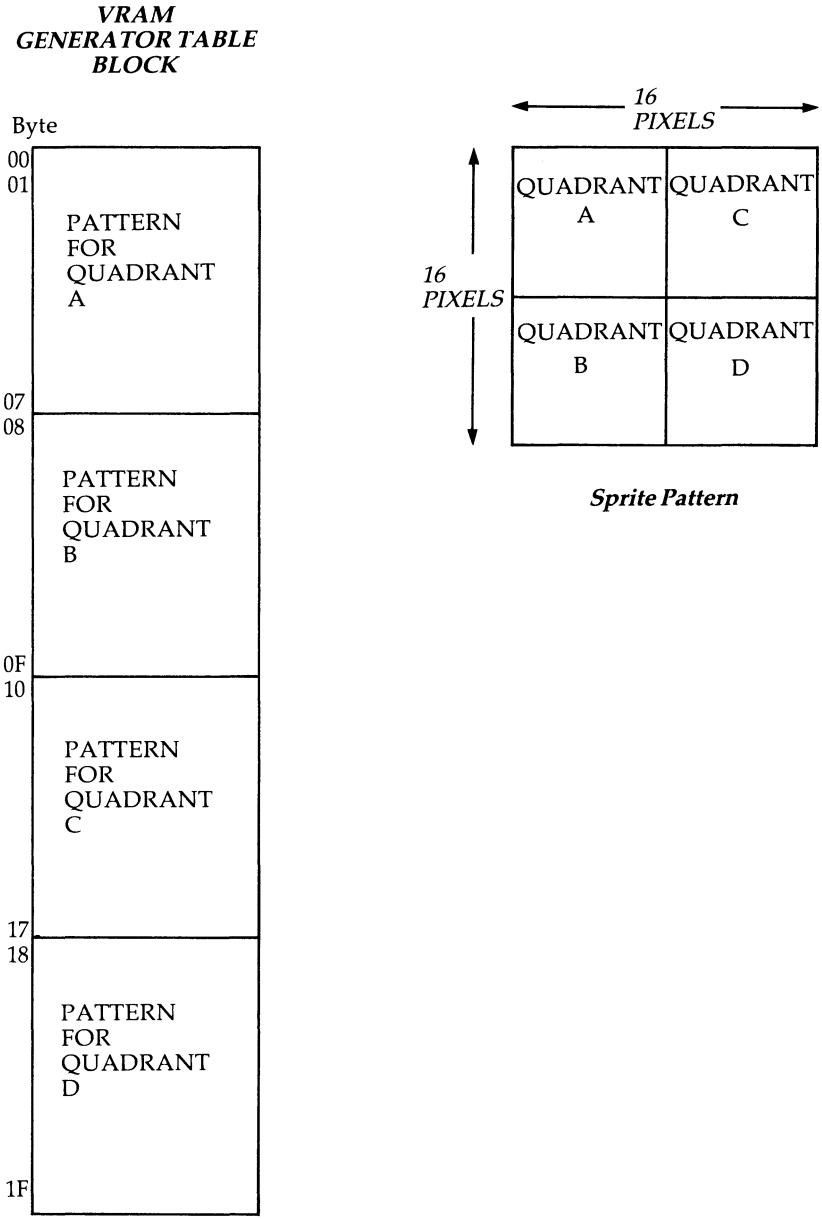
When a sprite is positioned at 0,0 it is butted up against the top left edge of the border. In many applications, however, it is necessary to bleed a sprite in from the edges of the screen. To effect this a different method is provided in each axis, as follows:-

Firstly the vertical position of the sprite is considered to be partially signed in that values between –31 and 0 (in two's complement) allow the sprite to bleed in from the top edge. It can readily be seen however that since the horizontal resolution of the display is 256 pixels this method cannot be used to bleed in sprites from the left hand edge; it is for this purpose that the 'early clock' bit is provided in byte four of each entry in the sprite attribute table.

The sprite generator table consists of a maximum of 256 eight byte entries. The third byte of each entry in the sprite attribute table then defines which block of the sprite generator table is to be used for the sprite on that plane. Note that when 16*16 sprites are in use this value is taken div 4, thus pointing to a 32 byte block of the sprite generator table.

Finally it must be noted that a maximum of four sprites may be displayed on any horizontal line. Should this rule be violated only the four highest priority sprites will be displayed; also the fifth sprite flag will be set in the status

Figure 6.2. 16x16 Sprite Mapping



register and the number of the offending fifth sprite will be loaded into the lower five bits.

The VDP in the MSX environment

In the MSX system the VDP may be accessed in two ways. Firstly the MSX system ROM provides routines to perform most of the basic functions that the user may require. In some cases however when these routines may be inadequate in their function or in their speed of execution, it is possible to the VDP directly since locations 6 and 7 of the system ROM hold read and write addresses for the VDP.

It may at first be less than obvious why there are two addresses provided, but on further inspection it may be seen that the reason lies in the MODE VDP control line.

Referring to Table 6.1 we may see that for a write to a VDP register the MODE line must be active (high), whereas for a read from or write to VRAM the MODE line is held low. Since, as previously noted, the VDP MODE line is usually tied to one the CPU address lines, we might expect that the addresses given in locations 6 and 7 would differ by the presence or absence of a single bit. Examining the ROM we find that location 6 contains 98H and location 7 contains 99H, thus we may deduce that the MODE line is tied to address line 0. The other two VDP control lines, CSR and CSW are gated to the Z80 control lines IORQ, RD and WR such that their control is performed automatically. The following routine will write to a location in VRAM specified by a fourteen bit address in HL, the data in the accumulator, and illustrates the above points:-

```
WRTVRM:  PUSH AF              ;SAVE DATA
         LD A,(6)             ;GET VDP ADDRESS WITH MODE HIGH
         LD C,A               ;INTO C
         INC C                ;MODE HIGH
         DI                   ;DISABLE INTERRUPTS
         OUT (C),L            ;OUTPUT LOW BYTE OF ADDRESS
         SET 6,H              ;SET BIT 6 OF HIGH BYTE OF ADDRESS
         OUT (C),H            ;OUTPUT IT
         DEC C                ;MODE LOW
         POP AF               ;RESTORE DATA
         OUT (C),A            ;OUTPUT IT
         EI                   ;RE-ENABLE INTERRUPTS
         RET                  ;AND RETURN
```

The operating system provides well-written and speed-optimised routines for all the functions which are likely to be needed and these routines will now be discussed.

ADDRESS	FUNCTION
0041H	Disables the screen display. This routine requires no parameters

to be passed, but modifies the AF and BC pairs.

0044H — Enables the screen display, requiring no parameters to be passed and modifying the AF and BC pairs.

0047H — Writes the data passed in B to the register specified by C, again modifying the AF and BC pairs.

004AH — Reads a byte of VRAM specified by an address in the HL pair returning the value read in the accumulator, and modifying only the AF pair.

004DH — Writes the data in the accumulator to the address in VRAM specified by the contents of the HL pair, modifying only the AF pair.

0050H — Sets up the VDP for a read from VRAM, it accepts the address for the first read in HL and modifies the AF pair.

0053H — This routine is the analogue of the above routine but sets up the VDP for a write operation instead of a read.

0056H — This routine fills the number of bytes of VRAM specified by the BC pair, and starting from the address specified by the HL pair with the data passed in the accumulator. It modifies the BC and AF pairs.

0059H — Moves a block of memory from VRAM to RAM, its operation is directly analogous to its corresponding routine below.

005CH — Moves a block of memory from CPU memory to VRAM. It accepts the address of the source in HL the destination in DE and the length of the block to be moved in BC, modifying all registers.

005FH — Sets the VDP to the display mode defined by location 00CAFH.(0 Text 40, 1 Graphics I, 2 Graphics II, 3 Multicolour.) This routine modifies all registers.

0062H — Changes the screen colour, accepting its parameters in 0F3E9H (foreground colour), 0F3EAH (background colour) and 0F3EBH (border colour). This routine modifies all registers.

0069H — Initialises all the sprites, sprite patterns are set to blanks, sprite names are set to sprite plane numbers, colours are set to foreground colour and vertical positions are set to 209. This routine requires the VDP display mode to be passed in 0FCAFH and modifies all registers.

006CH — Initialises the VDP for the 40 column text mode. This routine uses the values to be found in the system RAM area from 0F3B3H onwards and modifies all registers.

006FH Initialises the VDP for Graphics I mode, it and the following two routines are in all ways analogous to the above routine.

0072H Initialises the VDP for Graphics II mode.

0075H Initialises the VDP for Multicolour mode.

Under normal circumstances direct access to the VDP by the user is unlikely to be necessary.

Programming the VDP: hints and tips.

The Pattern Plane

In Graphics Modes I and II, which are the VDP modes of greatest interest to the general programmer, the appearance of the pattern plane is completely at the whim of the programmer since all character definitions are taken from RAM. It can readily be seen however that it would be advisable, in the production of one's own character definitions, to follow certain guidelines. for instance, in work on the MSX machines it has become clear that it is advisable to locate at least the upper case character set and the numerics in locations in VRAM such that it is possible to write the ASCII value of any number or letter to the name table, in order to display it. For example the definition of 'A' (ASCII value 65) would lie at an address in VRAM calculable by: (base address of pattern generator table)+(65*8).

I have found that the simplest way to achieve this is to fetch the basic definitions from their location in ROM and then redefine those which one is likely to need. This is done by the operating system in Graphics I mode when a `SCREEN 1` command is given, however when the `SCREEN` command is used to switch to Graphics Mode II the operating system sets up the VDP with all of the 768 definitions available blank. It then becomes necessary to locate the standard character definitions in the MSX system ROM. This location may differ from machine to machine so the following short program will locate the letter 'A' in the MSX ROM;

```
10 FOR N%=&H0000 TO &H7FFF
20 A%=PEEK(N%):READ B%
30 IF A%<>B% THEN RESTORE:Q%=0 ELSE Q%=Q%+1
40 IF Q%=8 THEN PRINT HEX$(N%-8):END
50 NEXT N%
60 END
70 DATA 32,80,136,136,248,136,136,0
```

The following machine code and BASIC programs illustrate many of the points to be considered when using the VDP. They also provide a powerful character and sprite definition program for use primarily in Graphics Mode II (but which may also be used in other modes).

```
        WRTVDP EQU 47H      ;TABLE OF BIOS CALLS
        RDVRM  EQU 4AH
        WRTVRM EQU 4DH
        FILVRM EQU 56H
        LDIRVM EQU 5CH
        LDIRMV EQU 59H
        CHGET  EQU 9FH
        GTSTCK EQU ØD5H
        GTTRIG EQU ØD8H
        RDVDP  EQU 13EH
        SNSMAT EQU 141H     ;END OF BIOS TABLE

;********GRAPHICS MODE II CHARACTER/COLOUR DEFINER*********


        NAMTAB EQU 18ØØH
        COLTAB EQU 2ØØØH    ;VRAM ADDRESSES FOR GM2
        PATTAB EQU ØØØØH
        SATTAB EQU 1BØØH
        SPTTAB EQU 38ØØH    ;END VRAM ADDRESSES
        RNAMTB EQU Ø6H      ;VDP REGISTER VALUES
        RCOLTB EQU ØFFH     ;FOR GRAPHICS II MODE
        RPTTAB EQU Ø3H
        RSATAB EQU 36H
        RSPTAB EQU Ø7H      ;END REGVALS
        CHARDT EQU 1CBFH    ;LOCATION OF CHARS IN ROM
                            ;MAY BE DIFFERENT

        ORG 9ØØØH


REGDAT: DEFB 2,ØE0H,RNAMTB,RCOLTB,RPTTAB,RSATAB,RSPTAB,6


START:  LD E,8
        LD IX,REGDAT        ;DATA POINTER IN IX
        LD D,Ø              ;LOAD REGISTERS
SVDPLP: LD B,(IX+Ø)         ;DATA IN B
        LD C,D              ;REGISTER NUMBER IN C
        CALL WRTVDP         ;TO VDP
        INC IX              ;INCREMENT DATA POINTER
        INC D               ;INCREMENT REGISTER NUMBER
        DEC E
        JR NZ,SVDPLP        ;DO 8 REGISTERS
        LD HHL,CHARDT       ;GET ADDRRESS OF CHARACTERS
        LD DE,PATTAB+1100H  ;DE IS VRAM ADDRESS
        LD BC,60*8          ;DO 60 CHARACTERS
        CALL LDIRVM
        LD BC,800H
        LD HL,COLTAB+1000H
        LD A,ØF1H           ;SET UP COLOURS
```

```
        CALL FILVRM
        LD BC,800H
        LD HL,COLTAB
        LD A,1FH
        CALL FILVRM          ;SET UP COLOURS
        LD BC,800H
        LD HL,COLTAB+800H
        LD A,81H
        CALL FILVRM          ;MORE COLOURS
        LD BC,768
        LD HL,NAMTAB
        LD A,32
        CALL FILVRM          ;CLEAR THE SCREEN
        LD HL,P2DAT
        LD DE,PATTAB+800H
        LD BC,24
        CALL LDIRVM          ;DEFINE CHARS 0..2 IN BLOCK 2
        LD HL,P2DAT+16
        LD DE,SPTTAB
        LD BC,8
        CALL LDIRVM          ;DEFINE SPRITE 0
        CALL FUNCPT          ;PRINT FUNCTION COLUMNS
        CALL FTTXRM          ;TRANSFER VRAM DATA TO RAM
MLP:    CALL ARRYPT          ;PRINT DEFINITION MATRIX
        CALL SPON            ;PUT ON CURSOR
        CALL JOY             ;READ JOYSTICK
        CALL FUNC            ;PRINT FUNCTION MESSAGES
        HALT                 ;WAIT FOR FLYBACK
        CALL DOFUNC          ;DO FUNCTIONS
NDFNC:  HALT                 ;WAIT FOR FLYBACK
        HALT
        CALL CHART           ;TRANSFER DEFINITION FROM
        CALL CSPT            ;MATRIX TO RAM
        CALL COLPT           ;AND PRINT STATUS MESSAGES
        CALL BANKPT
        CALL LPTXRM          ;TRANSFER VRAM DATA TO RAM
        HALT                 ;WAIT FOR FLYBACK
        JR MLP               ;GO BACK AND DO IT AGAIN
        RET
SPY:    DEFB 0               ;CURSOR Y POSITION
SPX:    DEFB 0               ;CURSOR X POSITION


ARRYPT: LD C,8               ;ROUTINE TO PRINT DEF MATRIX
        LD HL,NAMTAB+259     ;NAMETABLE ADDRESS+OFFSET
        LD IX,HARRY1         ;ARRAY ADDRESS IN IX
APTOLP: LD B,8               ;INNER LOOP OF 8
APTILP: LD A,(IX+0)          ;GET ARRAY DATA
        PUSH BC              ;SAVE COUNTERS
        CALL WRTVRM          ;WRITE ARRAY DATA TO SCREEN
        POP BC               ;RESTORE COUNTERS
        INC HL               ;INCREMENT SCREEN POINTER
```

```
        INC IX              ;INCREMENT ARRAY POINTER
        DJNZ APTILP         ;END OF INNER LOOP
        LD DE,24            ;LINE LENGTH-8
        ADD HL,DE           ;ADDED TO SCREEN POINTER
        DEC C               ;DEC OUTER LOOP COUNT
        JR NZ,APTOLP        ;DO IT 8 TIMES
        RET
FUNCPT: LD HL,NAMTAB+258    ;ADDRESS OF COLUMN LEFT OF ARRAY
        CALL ROWOF8         ;PRINT COLUMN
        LD HL,NAMTAB+267    ;RIGHT OF ARRAY
        CALL ROWOF8         ;PRINT COLUMN
        RET
ROWOF8: LD B,8              ;COUNT OF 8
RO8LP:  PUSH BC             ;SAVE COUNTER
        PUSH HL             ;SAVE ADDRESS
        LD A,2              ;CHARACTER 2
        CALL WRTVRM         ;TO SCREEN ADDRESS
        POP HL              ;RESTORE SCREEN ADDRESS
        LD DE,32            ;ADD LINE LENGTH...
        ADD HL,DE           ;TO SCREEN ADDRSS
        POP BC              ;RESTORE COUNTER
        DJNZ RO8LP          ;DO FOR COUNT OF 8
        RET
SPON:   LD HL,SATTAB        ;ROUTINE TO PLACE CURSOR
        LD A,(SPY)          ;GET CURSOR Y CO-ORD
        SLA A
        SLA A               ;MULTIPLY BY 8
        SLA A               ;AS SPRITE POS IS CHARPOS*8
        ADD A,63            ;TOP LEFT OF MATRIX, OFFSET
        CALL WRTVRM         ;WRITE TO SPRITE ATTRIBUTE TABLE
        INC HL              ;INCREMENT VRAM POINTER
        LD A,(SPX)          ;GET SPRITE X
        SLA A
        SLA A
        SLA A               ;*8
        ADD A,16            ;ADD X OFFSET
        CALL WRTVRM         ;TO SPRITE ATTRIBUTE TABLE
        INC HL              ;POINT TO SPRITE NAME
        LD A,Ø              ;NAME IS Ø
        CALL WRTVRM         ;WRITE SPRITE ATTRIBUTES
        INC HL              ;INC POINTER
        LD A,15             ;SPRITE IS WHITE
        CALL WRTVRM         ;TO ATTRIBUTES
        RET
P2DAT:  DEFB 255,129,129,129,129,129,129,255   ;DEFINITIONS ...
        DEFB 255,255,255,255,255,255,255,255   ;OF CHARACTERS
        DEFB 255,195,165,153,153,165,195,255   ;Ø-2 AND SPRITE Ø

HARRY1: DEFB Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
        DEFB Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
        DEFB Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
        DEFB Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
```

```
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0;CHARACTER ARRAY
          DEFB 0,0,0,0,0,0,0,0
HARRY2:   DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0
          DEFB 0,0,0,0,0,0,0,0;TEMPORARY ARRAY FOR
          DEFB 0,0,0,0,0,0,0,0;TRANSFORMATIONS
JOY:      LD A,0                ;TO READ JOY(0) IE.CURSOR KEYS...
          CALL GTSTCK           ;CHANGE AD LIB TO READ JOYSTICKS.
          LD HL,(SPY)           ;CURSOR X,Y IN HL
          CP 1                  ;JOYSTICK FORWARDS ?
          CALL Z,DEY            ;IF SO DEC Y
          CP 3                  ;JOYSTICK RIGHT ?
          CALL Z,INX            ;IF SO INC X
          CP 5                  ;JOYSTICK DOWN ?
          CALL Z,INY            ;IF SO INC Y
          CP 7                  ;JOYSTICK LEFT ?
          CALL Z,DEX            ;IF SO DEC X
          LD A,L                ;GET Y
          CP 8
          JR NZ,JOY1
          LD L,0                ;IF NECESSARY WRAP IT AROUND
JOY1:     CP 255
          JR NZ,JOY2
          LD L,7
JOY2:     LD A,H                ;MORE WRAPAROUND
          CP 10
          JR NZ,JOY3
          LD H,0                ;IF NECESSARY WRAP X
JOY3:     CP 255
          JR NZ,JOY4
          LD H,9
JOY4:     LD (SPY),HL           ;PUT CURSOR X Y TO STORE
          RET                   ;RETURN
INY:      INC L
          RET
DEY:      DEC L
          RET
INX:      INC H
          RET
DEX:      DEC H
          RET
EXIT:     LD A,0                ;EXIT ROUTINE
          LD (BASFNC),A         ;EXIT IS FUNCTION 0
          POP BC                ;REMOVE RETURN ADDRESS
          RET                   ;RETURN TO BASIC
GETMGS:   LD A,(MENU)           ;MENU 0 OR 1
```

```
        LD IY,MGSTAB        ;IY POINTS TO MESSAGES
        SLA A               ;MENU BECOMES Ø OR 2
        LD C,A
        LD B,Ø
        ADD IY,BC           ;IY IS IY OR IY PLUS 2
        LD A,(IY+Ø)         ;LOW ORDER OF MESSAGES ADDRESS
        LD C,A              ;INTO C
        LD A,(IY+1)         ;HIGH ORDER
        LD B,A              ;INTO B
        PUSH BC             ;GET MESSAGE TABLE ADDRESS
        POP IX              ;INTO IX
        RET
DFUNC:  DEFB Ø
MGSTAB: DEFW FNMSGS,FNMSG1  ;HOLDS ADDRESSES OF MESSAGE TABLES
MENU:   DEFB Ø              ;MENU NUMBER
FUNC:   CALL SPPNT          ;CLEAR OLD FUNCTION MESSAGE
        LD A,(SPX)          ;GET SPRITE X
        CP Ø                ;RIGHT HAND FUNCTION COLUMN ?
        JR Z,FUNC1          ;IF SO GO DO FUNC MESS PRINT
        CP 9                ;LEFT HAND COLUMN?
        RET NZ              ;IF NOT RETURN
        LD A,8
FUNC1:  LD B,A
        LD A,(SPY)
        ADD A,B
        LD B,A
        LD (DFUNC),A        ;CALCULATE AND STORE FUNCTION NUMBER
        PUSH BC             ;SAVE BC
        CALL GETMGS         ;GET MESSAGE ADDRESS
        POP BC              ;RESTORE BC
        CALL MSGFND         ;FINDS RELEVANT MESSAGE IN TABLE
FPNT:   LD HL,NAMTAB+515
FNPNT1: LD A,(IX+Ø)
        CP '$'
        RET Z
        CALL WRTVRM
        INC HL
        INC IX
        JR FNPNT1           ;PRINT MESSAGE
SPPNT:  PUSH AF             ;ROUTINE PRINTS SPACES
        LD IX,NOFUNC
        CALL FNPNT
        POP AF
        RET
MSGFND: DEC B               ;FINDS MESSAGE B IN STRING...
        RET M
MSGFN1: LD ,(IX+Ø)          ;ADDRESSED BY IX
        INC IX
        CP '$'
        JR NZ,MSGFN1
        JR MSGFND
NOFUNC: DEFB '                $'   ;NO FUNCTION DUMMY STRING
```

```
FNMSGS:  DEFB 'EXIT$LOAD$SAVE$CLEAR$';MENU 1 MESSAGE STRING
         DEFB 'CHARACTER$SPRITE$COLOUR$"
         DEFB 'SCROLL LEFT$SCROLL RIGHT$'
         DEFB 'SCROLL UP$SCROLL DOWN$'
         DEFB 'COPY$INVERT$FLIP VERTICAL$'
         DEFB 'FLIP HORIZONTAL$MENU 2$'
FNMSG1:  DEFB 'POSITION$SWAP BANK$MAGNIFY SPRITE$';MENU 2 STRING
         DEFB 'DEMAGNIFY SPRITE$16*16 SPRITE$'
         DEFB '8*8 SPRITE$NOT IMPLEMENTED$'
         DEFB 'NOT IMPLEMENTED$NOT IMPLEMENTED$'
         DEFB 'NOT IMPLEMENTED$NOT IMPLEMENTED$'
         DEFB 'NOT IMPLEMENTED$NOT IMPLEMENTED$'
         DEFB 'NOT IMPLEMENTED$NOT IMPLEMENTED$'
         DEFB 'MENU 1$'
EDIT:    LD IX,HARRY1          ;IX IS ARRAY POINTER
         LD A,(SPX)
         DEC A
         LD C,A
         LD B,Ø
         ADD IX,BC             ;ADD X OFFSET
         LD A,(SPY)            ;GET Y OFFSET
         SLA A
         SLA A
         SLA A                 ;TIMES 8
         LD C,A
         ADD IX,BC             ;ADD Y OFFSET
         LD A,(IX+Ø)
         INC A
         AND 1
         LD (IX+Ø),A           ;TOGGLE ARRAY LOCATION
         RET
DOFUNC:  LD A,Ø
         CALL GTTRIG           ;IS THE SPACE BAR PRESSED ?
         CP 255
         RET NZ                ;IF NOT RETURN
         LD A,(SPX)
         CP Ø
         JR Z,DOFN1
         CP 9
         JR NZ,EDIT            ;IF NOT IN A FUNCTION COLUMN
DOFN1:   LD IX,DFUNC
         LD A,(MENU)
         SLA A
         SLA A
         SLA A
         SLA A
         ADD A,(IX+Ø)
         SLA A
         LD C,A
         LD B,Ø
         LD IX,FNTABL
         ADD IX,BC
```

```
          LD A,(IX+Ø)          ;GET ADDRESS FROM
          LD L,A               ;JUMP TABLE
          LD A,(IX+1)          ;INTO
          LD H,A               ;HL
          JP (HL)              ;AND GO DO IT
FNTABL:   DEFW EXIT,LOAD,SAVE,CLR,CHAR,SPRIT  ;FUNCTION JUMP TABLE

          DEFW COLR,LSCRL,RSCRL,USCRL
          DEFW DSCRL,COPY,INVT,FVER,FHOR,NXTMNU
          DEFW POSIT,BANKSW,MSP
          DEFW DMSP,SP16,SP8,NIMP,NIMP,NIMP,NIMP
          DEFW NIMP,NIMP,NIMP,NIMP,NIMP,NXTMNU
NXTMNU:   LD A,(MENU)          ;TOGGLE MENU
          INC A
          AND 1
          LD (MENU),A
NIMP:     RET                  ;DUMMY ROUTINE FOR NON FUNCTIONS
SPRIT:    LD HL,SPTTAB         ;ROUTINE SELECTS SPRITE
          LD (CHSP),HL         ;TO DEFINE
          JR CHAR1             ;BY SETTING SPRITE ADDRESS
CHARF:    DEFB Ø               ;AND DOING CHARACTER SELECT
CHAR:     LD HL,PATTAB         ;CHARACTER SELECTION ROUTINE
          LD (CHSP),HL         ;CHARACTER OR SPRITE TABLE ADDRESS
CHAR1:    CALL GETNUM          ;NUMERIC INPUT ROUTINE
          LD (CHARF),A         ;STORE CHARACTER NUMBER
          CALL TCHAR           ;GET  IT TO DEFINE MATRIX
YOBBO:    LD A,Ø
          CALL GTTRIG
          CP Ø
          JR NZ,YOBBO
          RET
GETNUM:   CALL SPPNT           ;PRINT SPACES
GN1:      LD HL,NAMTAB+519
          CALL NUMLOP
          INC HL
          LD B,A
          CALL NUMLOP
          SLA B
          SLA B
          SLA B
          SLA B
          ADD A,B              ;GET TWO DIGIT HEX NUMERIC
          RET                  ;INTO A AND RETURN
NUMLOP:   PUSH HL
NUMLAP:   LD C,Ø
          LD A,Ø
          CALL GTTRIG
          CP Ø
          JR NZ,NUMLAP         ;WAIT FOR RELEASE OF SPACE
NUMLLP:   POP HL
SGLOP:    CALL NPNT            ;PRINT DIGIT
          PUSH HL
```

```
        LD A,Ø
        CALL GTSTCK
        CP Ø
        JR Z,NUMELP
        INC C
        LD A,C
        CP 16
        JR NZ,NUMELP
        LD C,Ø              ;IF A CURSOR KEY IS PRESSED
NUMELP: HALT                ;INCREMENT AND WRAP DIGIT
        HALT
        HALT
        HALT                ;WAIT A BIT
        LD A,Ø
        CALL GTTRIG         ;DONE THIS DIGIT?
        CP 255
        JR NZ,UMLLP         ;IF NOT GO DO IT AGAIN
        POP HL              ;ELSE RESTORE HL
        LD A,C              ;DIGIT IN A
        RET                 ;RETURN
NPNT:   LD A,C              ;NUMBER IN A
        ADD A,48            ;CONVERT ASCII
        CP 58               ;IF HEX A TO F
        JR C,NPNT2
        ADD A,7             ;THEN FURTHER ASCII CONVERT
NPNT2:  CALL WRTVRM         ;PRINT IT
        RET
TCHAR:  LD A,(CHARF)        ;THIS ROUTINE TRANSFERS
        LD L,A              ;VRAM CHARACTER OR SPRITE
        LD H,Ø              ;DEFINITION TO THE DEFINITION MATRIX
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        PUSH HL
        POP DE
        LD IX,(CHSP)
        ADD IX,DE
        PUSH IX
        POP HL
        LD C,8
        LD IY,HARRY1
TCOLP:  LD B,8
        CALL RDVRM
TCILP:  LD D,Ø
        RLC A               ;SHIFTS DEFINITION BITS
        JR NC,TCSKP         ;INTO CARRY
        LD D,1              ;AND WRITES Ø AND 1
TCSKP:  LD (IY+Ø),D         ;TO ARRAY AS NECESSARY
        INC IY              ;INCREMENT ARRAY POINTER
        DJNZ TCILP          ;AND DO IT AGAIN
        INC HL              :INC VRAM POINTER
        DEC C
```

```
        JR NZ,TCOLP         ;AND DO IT FOR 8 ROWS
        RET
CHSP:   DEFW PATTAB
CHART:  LD A,(CHARF)        ;THIS ROUTINE IS THE
        LD L,A              ;INVERSE OF THE PREVIOUS
        LD H,Ø              ;ROUTINE
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        PUSH HL
        POP DE
        LD IX,(CHSP)
        ADD IX,DE
        PUSH IX
        POP  HL
        LD C,8
        LD IY,HARRY1
CTLO:   XOR A
        LD D,A
        LD B,8
CTLI:   LD A,(IY+Ø)
        CP 1
        CCF
        RL D
        INC IY
        DJNZ CTLI
        LD A,D
        CALL WRTVRM
        INC HL
        DEC C
        JR NZ,CTLO
        RET
CLR:    LD IX,HARRY1        ;MATRIX CLEAR ROUTINE
        LD B,64             ;ARRAY IS 64 BYTES
        LD A,Ø              ;Ø CLEARS
CLRLP:  LD (IX+Ø),A         ;THE MATRIX
        INC IX              ;INC MATRIX POINTER
        DJNZ CLRLP          ;64 TIMES
        RET                 ;RETURN
INVT:   LD IX,HARRY1        ;IX IS MATRIX POINTER
        LD B,64             ;MATRIX IS 64 BYTES LONG
IVTLP:  LD A,(IX+Ø)         ;GET MATRIX CONTENTS
        INC A               ;Ø BECOMES 1
        AND 1               ;AND 1 BECOMES Ø
        LD (IX+Ø),A         ;PUT IT BACK
        INC IX              ;INCREMENT THE POINTER
        DJNZ IVTLP          ;DO IT 64 TIMES
        RET
POSIT:  CALL CURPOS         ;POSITION CURSOR
        LD HL,(CHSP)        ;CHARACTER OR SPRITE ?
        XOR A               ;CLEAR A AND CARRY FLAG
        LD DE,PATTAB        ;ADDRESS OF PATTERN GENERATOR TABLE
```

```
        SBC HL,DE           ;ARE WE DOING A CHARACTER?
        JR NZ,SPOSIT        ;NO, GO TO SPRITE POSITIONING
        LD A,(CYP)
        AND ØF8H
        SLA A
        SLA A               ;DIVIDE CURSOR X AND Y POS BY 8
        LD B,A
        LD A,(CXP)
        SRL A
        SRL A
        SRL A
        ADD A,B
        LD HL,NAMTAB
        LD E,A
        LD D,Ø
        ADD HL,DE           ;CALCULATE NAME TABLE ADDRESS
        LD A,(CHARF)
        CALL WRTVRM         ;AND WRITE PRESENT CHARACTER
        RET
SPOSIT: LD HL,SATTAB        ;ROUTINE TO POSITION SPRITE
        LD A,(CHARF)        ;SPRITE NUMBER
        AND 31              ;MOD 32
        LD B,A
SPOSTØ: INC HL
        INC HL
        INC HL
        INC HL
        DJNZ SPOSTØ         ;IS SPRITE PLANE TO USE
SPOST1: LD A,(CYP)
        CALL WRTVRM
        INC HL
        LD A,(CXP)
        CALL WRTVRM
        INC HL
        LD A,(CHARF)
        CALL WRTVRM
        LD A,(SPCOL)
        INC HL
        CALL WRTVRM         ;WRITE SPRITE ATTRIBUTES
        RET
SPCOL:  DEFB 3              ;HOLDS  SPRITE COLOUR
CHCOL:  DEFB 31H,31H,31H,31H,31H,31H,31H,31H  ;CHARACTER COLOURS
CXP:    DEFB Ø
CYP:    DEFB Ø
CURPOS: LD A,16             ;STARTING POSITION FOR...
        LD (CXP),A          ;POSITIONING CURSOR
        LD (CYP),A
CUP1:   LD A,Ø
        CALL GTTRIG         ;HAVE WE RELEASED...
        CP 255              ;THE SPACE BAR
        JR Z,CUP1
CUPOLP: LD HL,(CXP)         ;GET CURSOR X,Y COORDINATES
```

```
        LD A,Ø
        PUSH HL
        CALL GTSTCK
        POP HL
        CP 1
        JR NZ,CUP2          ;MOVE THE CURSOR AROUND
        DEC H
CUP2:   CP 3
        JR NZ,CUP3
        INC L
CUP3:   CP 5
        JR NZ,CUP4
        INC H
CUP4:   CP 7
        JR NZ,CUP5
        DEC L
CUP5:   LD A,H
        CP 57
        JR C,CUP6
        LD H,Ø
CUP6:   LD (CXP),HL
        LD HL,SATTAB
        LD A,(CYP)
        CALL WRTVRM
        INC HL
        LD A,(CXP)
        CALL WRTVRM
        INC HL
        INC HL
        LD A,12
        CALL WRTVRM         ;WRITE NEW SPRITE POSITION
        LD A,Ø
        PUSH HL
        CALL GTTRIG         ;HAVE WE FINISHED MOVING?
        POP HL
        HALT
        HALT
        HALT
        HALT
        CP Ø
        JR Z,CUPOLP         ;IF NOT GO MOVE SOME MORE
        RET
CSPT:   LD B,2Ø
        LD HL,NAMTAB+547
CSPTØ:  LD A,32
        CALL WRTVRM
        INC HL
        DJNZ CSPTØ
        LD HL(CHSP)
        LD DE,PATTAB        ;THIS IS ALL ANOTHER...
        LD IX,CMESS         ;SPACE AND MESSAGE...
        SBC HL,DE           ;PRINTING ROUTINE
```

```
        JR Z,CSPT1
        LD IX,SMESS
CSPT1:  LD HL,NAMTAB+547
CSPT2:  LD A,(IX+Ø)
        CP '$'
        JR Z,CSPT3
        CALL WRTVM
        INC HL
        INC IX
        JR CSPT2
CSPT3:  LD A,(CHARF)
        SRL A
        SRL A
        SRL A
        SRL A
        LD C,A
        CALL NPNT
        INC HL
        LD A,(CHARF)
        AND 15
        LD C,A
        CALL NPNT           ;THESE ROUTINES KEEP THE...
        RET                 ;STATUS AREA OF THE SCREEN...
CMESS:  DEFB 'CHAR:$'       ;UP TO DATE
SMESS:  DEFB 'SPRITE:$'
COLMES: DEFB 'SPRITECOL:$'
COLMS1: DEFB 'CHARCOLS$'
COLPT:  LD HL,NAMTAB+579
        LD IX,COLMES
COLPT1: LD A,(IX+Ø)
        CP '$'
        JR Z,COLPT2
        CALL WRTVRM
        INC HL
        INC IX
        JR COLPT1
COLPT2: LD A,(SPCOL)
        SRL A
        SRL A
        SRL A
        SRL A
        LD ,A
        CALL NPNT
        LD A,(SPCOL)
        AND 15
        LD C,A
        INC HL
        CALL NPNT
        LD HL,NAMTAB+532
        LD IX,COLMS1
COLPT3: LD A,(IX+Ø)
        CP '$'
```

```
        JR Z,COLPT4
        CALL WRTVRM
        INC HL
        INC IX
        JR COLPT3
COLPT4: INC HL
        PUSH HL
        LD A,(CHARF)
        LD L,A
        LD H,Ø
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        PUSH HL
        POP DE
        LD IX,COLTAB
        ADD IX,DE
        PUSH IX
        POP HL
        LD IX,CHCOL
        LD B,8
COLPT5: CALL RDVRM
        LD (IX+Ø),A
        INC IX
        INC HL
        DJNZ COLPT5
        POP HL
        LD B,8
        LD DE,31
        LD IX,CHCOL
COLPT6: LD A,(IX+Ø)
        PUSH AF
        SRL A
        SRL A
        SRL A
        SRL A
        LD C,A
        CALL NPNT
        POP AF
        AND 15
        INC HL
        LD C,A
        CALL NPNT
        ADD HL,DE
        INC IX
        DJNZ COLPT6
        RET
BANK:   DEFB Ø
BPMESS: DEFB 'BANK:'
BANKPT: LD HL,NAMTAB+611
        LD IX,BPMESS
BKPT1:  LD A,(IX+Ø)
```

```
        CP '$'
        JR Z,BKPT2
        CALL WRTVRM
        INC HL
        INC IX
        JR BKPT1
BKPT2:  LD A,(BANK)
        AND 15
        LD C,A
        CALL NPNT
        RET
FTTXRM: LD HL,PATTAB
        LD DE,ØBØØØH
        LD BC,18ØØH
        CALL LDIRMV
        LD HL,COLTAB
        LD DE,ØC8ØØH
        LD BC,18ØØH
        CALL LDIRMV
SPTXRM: LD HL,SPTTAB
        LD DE,ØEØØØH
        LD BC,8ØØH
        CALL LDIRMV
        RET
LPTXRM: LD A,(BANK)
        LD H,A
        LD L,Ø
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        PUSH HL
        LD DE,ØBØØØH
        ADD HL,DE
        EX DE,HL
        LD HL,PATTAB
        LD BC,8ØØH
        CALL LDIRMV
        POP HL
        LD DE,ØC8ØØH
        ADD HL,DE
        EX DE,HL
        LD HL,COLTAB
        LD BC,8ØØH
        CALL LDIRMV
        JR SPTXRM
COLR:   CALL SPPNT
        LD A,Ø
        CALL GTTRIG
        CP 255
        JR Z,COLR
        LD HL,(CHSP)
        XOR A
```

```
          LD DE,PATTAB
          SBC HL,DE
          JR NZ,COLSP
          LD IX,CHCOL
          LD B,8
COLLYP:   LD A,8
          SUB B
          ADD A,48
          PUSH BC
          LD HL,NAMTAB+517
          CALL WRTVRM
          INC HL
          LD A,':'
          CALL WRTVRM
          INC HL
          LD A,(IX+Ø)
          SRL A
          SRL A
          SRL A
          SRL A
          LD C,A
HNGON1:   LD A,Ø
          CALL GTTRIG
          CP 255
          JR Z,HNGON1
          PUSH IX
          CALL SGLOP
          POP IX
          SLA A
          SLA A
          SLA A
          SLA A
          LD B,A
          LD A,(IX+Ø)
          AND 15
          OR B
          LD (IX+Ø),A
          AND 15
          LD C,A
          INC HL
HANGON:   LD A,Ø
          CALL GTTRIG
          CP 255
          JR Z,HANGON
          PUSH IX
          CALL SGLOP
          POP IX
          AND 15
          LD B,A
          LD A,(IX+Ø)
          AND ØFØH
          OR B
```

```
        LD (IX+Ø),A
        INC IX
        POP BC
        DJNZ COLLYP
        LD A,(CHARF)
        LD L,A
        LD H,Ø
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        LD DE,COLTAB
        ADD HL,DE
        EX DE,HL
        LD HL,CHCOL
        LD BC,8
        CALL LDIRVM
        RET
COLSP:  CALL GETNUM
        LD (SPCOL),A
        RET
MSP:    LD A,(9ØØ1H)        ;GET PRESENT VALUE VDP(1)
        OR 1                ;SET THE MAG FLAG
        LD (9ØØ1H),A        ;PUT IT BACK
        LD B,A
        LD C,1
        CALL WRTVRM         ;WRITE IT TO VDP
        RET
DMSP:   LD A,(9ØØ1H)
        AND ØFEH            ;RESET THE MAG FLAG
        LD (9ØØ1H),A
        LD B,A
        LD C,1
        CALL WRTVDP         ;AND WRITE IT TO VDP
        RET
SP8:    LD A,(9ØØ1H)
        AND ØFDH            ;RESET 16*16 SPRITE FLAG
        LD (9ØØ1H),A
        LD B,A
        LD C,1
        CALL WRTVDP         ; AND WRITE IT TO VDP
        RET
SP16:   LD A,(9ØØ1H)
        OR 2                ;SET 16*16 SPRITE FLAG
        LD (9ØØ1H),A
        LD B,A
        LD C,1
        CALL WRTVDP         ;AND WRITE TO VDP
        RET
LSCRL:  LD IX,HARRY1        ;SCROLLS LEFT FROM ARRAY1
        LD IY,HARRY2        ;TO ARRAY 2
        LD C,8
LSLP:   LD A,(IX+Ø)
```

```
          LD (IY+7),A
          LD B,7
          INC IX
LSLP1:    LD A,(IX+Ø)
          LD (IY+Ø),A
          INC IX
          INC IY
          DJNZ LSLP1
          INC IY
          DEC C
          JR NZ,LSLP
          LD IX,HARRY1          ;PUT ARRAY2 TO ARRAY1
          LD IY,HARRY2
          LD B,64
LSLP2:    LD A,(IY+Ø)
          LD (IX+Ø),A
          INC IX
          INC IY
          DJNZ LSLP2            ;DONE 64 BYTES?
LSLP3:    LD A,Ø
          CALL GTTRIG           ;WAIT FOR SPACE BAR RELEASE
          CP 255
          JR Z,LSLP3
          RET
RSCRL:    LD B,7                ;RIGHT SCROLL DOES LEFT
RSCRLP:   PUSH BC               ;SCROLL 7 TIMES
          CALL LSCRL
          POP BC
          DJNZ RSCRLP
          RET
USCRL:    LD DE,8               ;UPWARDS SCROLL IS SIMILAR...
          LD IX,HARRY1          ;TO LEFT SCROLL
          LD IY,HARRY2
          LD C,8
USRLP1:   PUSH IX
          PUSH IY
          LD A,(IX+Ø)
          LD (IY+56),A
          ADD IX,DE
          LD B,7
USRLP2:   LD A,(IX+Ø)
          LD (IY+Ø),A
          ADD IX,DE
          ADD IY,DE
          DJNZ USRLP2
          POP IY
          POP IX
          INC IY
          INC IX
          DEC C
          JR NZ,USRLP1
          LD IX,HARRY1
```

```
        LD IY,HARRY2
        LD B,64
USRLP3: LD A,(IY+Ø)
        LD (IX+Ø),A
        INC IX
        INC IY
        DJNZ USRLP3
USRLP4: LD A,Ø
        CALL GTTRIG
        CP 255
        JR Z,USRLP4
        RET
DSCRL:  LD B,7              ;DOWNSCROLL DOES UP..
DCRLP:  PUSH BC
        CALL USCRL          ;7 TIMES.
        POP BC
        DJNZ DCRLP
        RET
FVER:   LD IX,HARRY1        ;VERTICAL MIRROR FROM ARRAY 1
        LD IY,HARRY2+56     ;TO ARRAY2
        LD C,8
        LD DE,8
FVER1:  PUSH IX
        PUSH IY
        LD B,8
FVER2:  LD A,(IX+Ø)
        LD (IY+Ø),A
        INC IX
        INC IY
        DJNZ FVER2
        POP HL
        XOR A
        SBC HL,DE
        PUSH HL
        POP IY
        POP IX
        ADD IX,DE
        DEC C
        JR NZ,FVER1
        LD B,64
        LD IX,HARRY1        ;THEN PUT TRANSFORMED..
        LD IY,HARRY2        ;ARRAY2 BACK TO ARRAY 1
FVER3:  LD A,(IY+Ø)
        LD (IX+Ø),A
        INC IX
        INC IY
        DJNZ FVER3
        JP USRLP4
FHOR:   LD IX,HARRY1        ;HORIZONTAL MIRROR...
        LD IY,HARRY2+7      ; IS SIMILAR TO VERTICAL
        LD C,8
FHOR1:  PUSH IY
```

```
          LD DE,8
          LD B,8
FHOR2:    LD A,(IX+Ø)
          LD (IY+Ø),A
          INC IX
          DEC IY
          DJNZ FHOR2
          POP IY
          XOR A
          ADD IY,DE
          DEC C
          JR NZ,FHOR1
          LD B,64
          LD IX,HARRY1
          LD IY,HARRY2
FHOR3:    LD A,(IY+Ø)
          LD (IX+Ø),A
          INC IX
          INC IY
          DJNZ FHOR3
          JP USRLP4
BANKSW:   CALL SPPNT
          LD HL,NAMTAB+157
          CALL NUMLOP
          CP 3
          JR NC,BANKSW
BSW1:     LD (BANK),A
          LD BC,8ØØH
          LD DE,ØB ØØØH
          LD H,A
          LD L,Ø
          ADD HL,HL
          ADD HL,HL
          ADD HL,HL
          PUSH HL
          LD HL,DE
          LD DE,PATTAB
          CALL LDIRVM
          POP HL
          LD DE,ØC8ØØH
          ADD HL,DE
          LD DE,COLTAB
          LD BC,8ØØH
          CALL LDIRVM
          CALL TCHAR
          RET
COPBNK:   DEFB Ø
COPCHA:   DEFB Ø
COPY:     CALL SPPNT          ;ROUTINE TO COPY A DEFINITION
          LD IX,BPMESS
          LD HL,NAMTAB+517
COPYLP:   LD A,(IX+Ø)
```

```
        CP '$'
        JR Z,COPY1
        CALL WRTVRM
        INC HL
        INC IX
        JR COPYLP           ;PRINT MESSAGE
COPY1:  CALL NUMLOP         ;GET BANK NUMBER
        LD (COPBNK),A       ;SAVE BANK TO COPY FROM
        CALL GETNUM         ;GET CHARACTER/SPRITE NUMBER
        LD (COPCHA),A
        LD A,(COPBNK)
        CP 4
        JR NC,COPY
        CP 3
        JP Z,SPCOPY         ;COPY CHAR OR SPRITE
        LD A,(COPCHA)
        LD L,A
        LD A,(COPBNK)
        LD H,A
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL           ;CALCULATE CHARACTER ADDRESS
        PUSH HL
        LD DE,ØBØØØH
        ADD HL,DE           ;IN RAM
        PUSH HL
        LD A,(CHARF)
        LD L,A
        LD H,Ø
        ADD HL,HL
        ADD HL,HL
        ADD HL,HL
        LD DE,(CHSP)
        ADD HL,DE
        EX DE,HL            ;GET CHAR OR SPRITE VRAM ADDR
        POP HL
        LD BC,8
        CALL LDIRVM         ;TRANSFER IT FROM RAM TO VRAM
        POP BC
        LD DE,(CHSP)
        LD HL,PATTAB
        XOR A
        SBC HL,DE
        JR NZ,COPSKP
        PUSH BC
        POP HL
        LD DE,ØC8ØØH        ;IF TRANSFERRING A CHARACTER
        ADD HL,DE
        PUSH HL
        LD DE,COLTAB
        LD A,(CHARF)
        LD L,A
```

```
          LD H,Ø
          ADD HL,HL
          ADD HL,HL
          ADD HL,HL
          ADD HL,HL
          EX DE,HL
          POP HL
          LD BC,8
          CALL LDIRVM          ;THEN GET COLOUR TABLE ENTRY
COPSKP:   CALL TCHAR           ;PUT CHAR TO MATRIX
          RET
SPCOPY:   LD A,(COPCHA)        ;SUBROUTINE COPIES SPRITES
          LD L,A
          LD H,Ø
          ADD HL,HL
          ADD HL,HL
          ADD HL,HL
          LD DE,ØE000H
          ADD HL,DE
          PUSH HL
          LD A,(CHARF)
          LD L,A
          LD H,Ø
          ADD HL,HL
          ADD HL,HL
          ADD HL,HL
          LD DE,(CHSP)
          ADD HL,DE
          EX DE,HL
          POP HL
          LD BC,8
          CALL LDIRVM
          CALL TCHAR
          RET
LOAD:     LD A,1               ;LOAD IS BASIC FUNCTION 1
L1:       LD (BASFNC),A
          POP HL               ;GET RETURN ADDRESS OFF STACK
          LD HL,L2             ;GET  NEW RETURN ADDRESS
          LD (RTADDR),HL       ;SAVE IT FOR BASIC
          RET                  ;TO BASIC
L2:       LD A,(BANK)
          CALL GTX
          CALL TCHAR
          JP NDFNC
SAVE:     LD A,2               ;SAVE IS BASIC FUNCTION 2
          JR L1                ;GOTO LOAD ROUTINE
GTX:      LD HL,ØB000H
          LD DE,PATTAB
          LD BC,1800H
          CALL LDIRVM
          LD HL,ØC800H
          LD DE,COLTAB
```

```
        LD BC,1800H
        CALL LDIRVM
        LD HL,0E000H
        LD DE,SPTTAB
        LD BC,800H
        CALL LDIRVM
        RET
BASFNC: DEFB 0              ;STORE FOR FUNCTION NUMBER...
RTADDR: DEFW 0              ;AND RETURN ADDRESS FOR...

                            ;BASIC


        END
```

The above program should be assembled, and saved to tape as 'GM2'. The following BASIC program should then be typed in and saved. The program will load and run the machine code.

```
10 CLEAR 200,&H8FFF
20 PRINT "LOADING..."
30 BLOAD"CAS:GM2"
40 DEFUSR=&H9008
50 A=USR(A)
60 DEFUSR=PEEK(&H9AED)+256*(PEEK(&H9AEE)):REM RTADDR
70 IF PEEK(&H9AEC)=0 THEN SCREEN1,0,0,1:END
80 IF PEEK(&H9AEC)=1 THEN BLOAD"CAS:CHARS"
90 IF PEEK(&H9AEC)=2 THEN BSAVE"CAS:CHARS",&HB000,&HE801
100 GOTO 50
```

Using the Definer.
On running the program you will find that the screen is divided into three zones. The top zone is black, the middle zone contains a 10 by 8 grid (the extreme left and right columns of which are marked with a line of crosses) and the third which displays the functions which the program will perform. By moving the cursor over the left and right columns of the second zone you will see messages appearing under the grid, in the third section of the screen. The cursor is moved by use of the cursor control keys and the space bar has two functions; when the cursor is the main 8 by 8 body of the definition matrix pressing the space bar will toggle the cursor position in the matrix from set to unset and vice versa. When the cursor is situated in one of the function columns, however, pressing the space bar will perform the function displayed below the character matrix.

In addition to the function messages the lower third of the screen also displays certain status messages, most of which are self-explanatory.

The functions which the program will perform are listed below;

EXIT	Exits the program.
LOAD	Loads a file called "CHARS" from tape which contains

	character and sprite definitions and the character colours.
SAVE	Saves the file "CHARS" to tape.
CLEAR	Clears the definition matrix.
CHARACTER	Selects the character to be modified and transfers its definition from VRAM to the character definition matrix.
SPRITE	Selects the sprite to be defined.
COLOUR	When defining a character this allows the colours for the character to be defined row by row. When defining a sprite this sets up the colour for the POSITION function.
SCROLL LEFT RIGHT UP DOWN	Performs a scroll with wraparound.
COPY	Copies a character or sprite definition to the present character or sprite.
INVERT	Sets and resets all pixels in the current definition.
FLIP HORIZONTAL VERTICAL	Mirrors the definition across either a horizontal or vertical line drawn across the centre of the definition.
MENU 1/2	Toggles the function menus accessible by the function columns.
POSITION	Allows the positioning of sprites or characters in the top third of the display.
SWAP BANK	As we have previously seen the pattern table in Graphics Mode II consists of three sets of 256 character definitions. The bank number refers to which of those three sets of definitions the characters being defined and those appearing in the third of the screen belong. (Note that when prompted for a bank number in the COPY function banks 0 to 2 are the three character banks and bank 3 is the sprite definitions).
MAGNIFY DEMAGNIFY SPRITES 8*8/16*16 SPRITES	These two functions set up the VDP registers for sprite magnification and size.

Numeric input is always considered to be two hexadecimal digits. These digits are selected by the cursor keys, high order followed by low order.

The character data is left on exit from the program in CPU RAM at the following locations:-

0B000H to 0C800H	The pattern generator table, (ie. character definitions).
0C800H to 0E000H	The colour table.
0E000H to 0E800H	Sprite definitions.

The definer in modes other than Graphics II

In Graphics Mode I there are only 256 possible character definitions and so only the first 'bank' need be defined. These definitions will then lie from

0B000H to 0B800H (the colour table may be ignored) and the sprite definitions remain useful.

To use the definer in text mode the above notes remain in force, except that the sprites are not available and therefore their definitions are irrelevant. It must be remembered that in text mode only the left hand 6 columns are used so it is advisable to leave the right hand 2 columns of any definition blank.

Dynamic pattern definition.

Since the VDP keeps all of its character definitions in VRAM, it is possible to achieve some effective animation on the pattern plane by redefining a character while the program is running. When this is done it can be seen that the new definition will appear corresponding with all references in the name table. Thus large areas of the screen may be changed by updating a relatively small number of entries in the pattern generator table. As an example, let us consider the production of a ladder 7 characters high and 1 wide. This will appear as an elevator with the rungs travelling from the bottom to the top and will be achieved by the use of only one character.

Firstly we must produce the basic character definition. A character is needed which will appear to be part of a ladder, and will tesselate vertically. The following pattern definition will serve:

```
10000001  81H
11111111  FFH
10000001  81H
10000001  81H
10000001  81H
11111111  FFH
10000001  81H
10000001  81H
```

Having decided on the appearance of our basic character it must be inserted into VRAM; we will make it character 0.

```
CHRDEF:  DEFB 081H,0FFH,081H,081H,081H,0FFH,081H,081H
CHARIN:  LD B,8              ;NUMBER OF ENTRIES IN CHRDEF
         LD IX,CHRDEF        ;GET ADDRESS OF DEFINITION IN IX
         LD HL,PGTAB         ;GET ADDR OF PGT IN HL
CILP:    LD A,(IX+0)         ;GET BYTE OF DATA
         CALL 004DH          ;WRITE (HL) IN VRAM
         INC HL
         INC IX              ;INCREMENT POINTERS
         DJNZ CILP           ;DO IT 8 TIMES
         RET
```

We now have a definition in VRAM which we may alter to achieve the desired effect. The manipulation required is a vertical scroll of the eight bytes of the character definition, wrapping the top byte around to become the bottom.

The following routine will achieve this:-

```
        RDVRM  EQU 00H4AH
        WRTVRM EQU 00H4DH

DSCROL: LD B,7              ;SET UP LOOP COUNTER
        LD HL,PGTAB         ;SET UP VRAM ADDR OF DEF
        CALL RDVRM          ;GET FIRST VALUE
        LD C,A              ;AND SAVE IT FOR LATER
DSCLP:  INC HL              ;INCREMENT VRAM POINTER
        CALL RDVRM          ;GET DEFINTION LINE
        DEC HL              ;AND STORE IT UP ONE...
        CALL WRTVRM         ;LINE OF DEFINITION
        INC HL              ;RESET VRAM POINTER
        DJNZ DSCLP          ;DO IT 7 TIMES
        LD A,C              ;OLD TOP LINE...
        CALL WRTVRM         ;BECOMES NEW BOTTOM LINE
        RET
```

By calling the above routine again and again our ladder character will appear to have upward-moving rungs. Ladders of any size may be built. The following program is also needed:-

```
START:  CALL CHARIN         ;DEFINE CHARACTER
        LD HL,NAMTAB+8      ;GET NAME TABLE ADDR+8
        LD DE,32            ;LINE LENGTH IN DE
        LD B,7              ;LADDER IS 7 CHARACTERS LONG
        LD A,0              ;LADDER CHARACTER IS 0
PLOOP:  CALL WRTVRM         ;PRINT LADDER CHARACTER
        ADD HL,DE           ;PRINT POSITION DOWN 1 LINE
        DJNZ PLOOP          ;DO 7 CHARACTERS
LOOP2:  CALL DSCROL         ;SCROLL DEFINITION
        HALT
        HALT                ;WAIT 2/50THS OF A SECOND
        JR LOOP2            ;GO ROUND AND DO IT AGAIN
```

As may be seen from this example the principle of dynamic character definition is a very powerful tool; its use is only limited by the imagination. In fact it is the principle of dynamic character definition which (in the next section) will allow us to consider the Graphics II mode as a bit mapped mode.

Graphics II mode as a bit mapped mode.

If in Graphics II mode we set up the name table such that the 768 bytes of the table are consecutively numbered 0 to 255 three times it follows that each character cell will have its own definition in the pattern generator table, and as such may be considered to be bit mapped as shown in figure 6.1. Having set up the name table in this fashion we have now, in effect, a bit mapped 256 by 192 pixel high resolution mode with 15 colours available (although unfortunately the horizontal colour resolution is only 32 eight pixel blocks).

If we consider the origin to be the lower left hand corner of the display the formula for calculating which byte in the pattern generator table contains the pixel referred to by any X,Y coordinate is:-

(Pattern table base address)+((((191-Y)div 8)*256)+((191-Y)mod 8)+(X AND 0F8H))

and the bit to be set within that byte may be found by,

7–(X AND 7)

Using the above formulae, the following routine will set the pixel specified by the coordinates in HL (L contains the XX coordinate, and H contains the Y coordinate) to the colour specified by the accumulator:

```
SETXY:  PUSH AF             ;SAVE COLOUR CODE
        LD A,191            ;MAKE Y EQUAL...
        SUB H               ;TO 191-Y
        LD H,A              ;PUT IT BACK IN H
        PUSH HL             ;SAVE X,Y
        SRL H
        SRL H
        SRL H               ;Y BECOMES Y DIV 8
        LD D,H              ;MAKING IT HIGH ORDER OF DE=*256
        POP HL              ;RESTORE X,Y
        LD A,H              ;GET Y
        AND 7               ;MOD 8
        LD E,A              ;MAKE IT LOW ORDER OF DE
        XOR A               ;CLEAR CARRY FLAG FOR 16 BIT ADDITION
        LD A,L              ;GET X
        AND ØF8H            ;
        LD E,A              ; ADD IT TO DE
        LD A,Ø
        ADC A,D             ;FINISH 16 BIT ADDITION
        LD D,A              ;DE NOW CONTAINS THE OFFSET FROM
                            ;PATTERN GENERATOR, AND COLOUR TABLE
                            ;BASE ADDRESSES.
        LD B,L              ;SAVE X
        LD HL,COLTAB        ;GET COLOUR TABLE BASE ADDRESS
        ADD HL,DE           ;ADD THE OFFSET
        CALL ØØ4AH          ;GET PRESENT COLOUR VALUE
        AND 15              ;MASK OUT BACKGROUND COLOUR
        LD C,A              ;SAVE IT IN C
        POP AF              ;RESTORE COLOUR
        SLA A               ;SHIFT COLOUR CODE
        SLA A
        SLA A
        SLA A               ;TO HIGH ORDER NIBBLE
        ADD A,C             ;ADD BACKGROUND COLOUR
        CALL ØØ4DH          ;AND PUT IT BACK
        LD A,7
        AND B
```

```
        LD B,A
        XOR A
        INC B
        CCF
MLP:    RRA                 ;SHIFT CARRY FLAG INTO A TO FORM
        DJNZ MLP            ;PLOT MASK
        LD HL,PGTAB         ;ADDRESS OF PAT GEN TAB IN HL
        ADD HL,DE           ;ADD OFFSET
        CALL 004AH          ;GET OLD VALUE
        XOR B               ;MASK WITH B
        CALL 004DH          ;PUT IT BACK
        RET                 ;DONE
```

By considering the basics of the Graphics II mode it may easily be seen that it is a simple matter, by manipulation of the entries in the name table, to set up many different apparent screen memory maps. In practice, however, the memory map described above and in figure 6.7 is as easy to use as any and in many ways is more convenient.

It must also be remembered that although the above technique allows us to treat the Graphics II mode screen as bit mapped, we may still perform operations as though it were character mapped. Thus the whole screen may be scrolled, in 8 pixel steps, by the manipulation of only 768 bytes of VRAM. Alternatively, we may define the last block of character definitions (those which correspond with references in the lower third of the name table) to be a fairly standard character set and treat the top two thirds of the screen as if it were bit mapped. Further possibilities will doubtless occur to the reader.

More from sprites: interrupt switching techniques.

There are two obvious problems with sprite handling in the TMS-9929A VDP. Firstly sprites may only be of a single colour, and secondly all sprites must at any one time be the same size and magnification. It is however possible to produce sprites which are apparently of two colours, and it is also possible to maintain sprites on screen in two different sizes or magnifications. This may be achieved by interrupt switching of the sprite tables. Note that when accessing the VDP during an interrupt, it is essential that the interrupt not occur during VDP access by the background program. This may be achieved either by putting all background VDP access after a Z80 'HALT' instruction, or by flagging that VDP access is occurring.

Two colour sprites.

By switching the sprite name and sprite colour entries in the sprite attribute table at every frame flyback interrupt, it is possible to produce sprites which appear to be in two different colours. The following method is suggested:-

Firstly define two definitions for the sprite: one definition for each colour. The definitions should lie in VRAM conseecutively, such that sprite definitions 0 and 1 define one multicolour sprite, definitions 2 and 3 another and so on.

Secondly reserve in memory a 'sprite switching table'. Each entry of this will contain one byte ccontaining the colour code for the even-numbered sprite definition (in its high order nibble) and the colour code for the odd-numbered sprite definition (in its low order nibble). This table should bee written to whenever a new sprite is placed in the sprite attribute table. Lastly an interrupt driven routine should be written to switch between the two definitions and the two colours at each frame flyback interrupt. The following routine illustrates this:-

```
SPSWTB:  DEFS 32              ;SWITCH COLOUR TABLE


SPSWIT:  LD HL,SATTAB         ;BASE ADDR FOR SPR ATT TBL IN HL
         LD DE,SPSWTB         ;ADDR OF COLOUR SWITCH TBL IN DE
         LD B,32              ;MAX NO. OF ENTRIES TO SWITCH
SPSWLP:  CALL 004AH           ;READ VERT POS OF SPRITE
         CP 0D0H              ;IS IT THE 'END OF SPRITES' MARKER
         RET Z                ;IF SO QUIT
         INC HL               ;ELSE INC SPRT ATTR TBL PTR
         INC HL               ;TWICE, TO POINT TO SPRITE NAME
         CALL 004AH           ;READ PATTERN NO.
         XOR 1                ;PATTERN BECOMES PATTERN + OR - 1
         LD C,A               ;SAVE NEW PATTERN NUMBER
         CALL 004DH           ;AND WRITE IT TO SPRITE ATTRIBUTES
         INC HL               ;POINT TO SPRITE TAG
         LD A,(DE)            ;GET SWITCH COLOURS
         AND A                ;CLEAR CARRY FLAG
         RR C                 ;ROTATE BIT 0 OF PAT NO. INTO CARRY
         JR C,SKP             ;IF PAT IS EVEN NUMBERED  DONT...
         SLA A                ;SHIFT TOP NIBBLE OF COLOUR...
         SLA A
         SLA A
         SLA A                ;INTO BOTTOM NIBBLE
SKP:     AND 15               ;MASK OUT TOP NIBBLE
         LD C,A               ;STORE NEW COLOUR CODE IN C
         CALL 004AH           ;READ OLD TAG
         AND 80H              ;MASK OUT EARLY CLOCK BIT
         OR C                 ;ADD NEW COLOUR
         CALL 004DH           ;WRITE NEW TAG TO SPRITE ATTRIBUTES
         INC HL               ;POINT NEXT ENTRY IN SPRT ATTR TBL
         INC DE               ;POINT NEXT COL SWITCH TBL ENTRY
         DJNZ SPSWLP          ;DO IT ALL 32 TIMES
         RET                  ;AND RETURN TO O.S. INTERRUPT
```

Thus, two sprites of different colours and definitions have been made to share the same entry in the sprite attribute table, each being displayed on every second frame of the television picture. It can also be seen that a small amount

of flicker will be seen because of this; under normal circumstances the persistance of the average display tube is great enough to make the flicker insignificant.

Different sized sprites: interrupt switching of VDP registers.

The maintenance of sprites of different sizes or magnifications on screen at the same time requires two sprite attribute tables to be maintained in VRAM at the same time - one for each size or magnification. The VDP is then made to access them one after another for each frame.

In the Multicolour mode and in Graphics Mode I it is usually possible to design a VRAM memory map to allow this without overlapping the other VDP sub-blocks. In Graphics Mode II, however, it is necessary to overlap the second sprite attribute table over one of the other tables. It is most satisfactory to steal the last 16 sprite definitions for this purpose, and the following memory map illustrates this:-

0000H Character generator table
1800H Pattern name table
1B00H Sprite attribute table number 1
2000H Character colour table
3800H Sprite pattern generator table
3F80H Sprite attribute table number 2

Having designed a suitable memory map it is then a simple matter to write an interrupt routine which will switch the size and/or magnification bits in register 1, and the sprite attribute table base address bits in register 5. The following routine illustrates this by maintaining two attribute tables, one for 16*16 unmagnified sprites, and the other for 16*16 magnified sprites:-

```
          VAL1A   EQU 0E2H      ;REG 1 VALUE SETS 16*16 SPRTS UNMAG
          VAL1B   EQU 0E3H      ;AS ABOVE BUT WITH MAG FLAG SET
          VAL5A   EQU 036H      ;VALUE FOR REG 5 SETS ATTR ADDR TO
                                ;BE 1B00H.
          VAL5B   EQU 07FH      ;SETS ATTR TBL AT 3F80H


CTR:      DEFB 0                ;SWITCH COUNTER


SPATSW:   LD A,(CTR)            ;GET SWITCH COUNTER
          INC A                 ;INCREMENT IT
          LD (CTR),A            ;PUT IT BACK
          AND 1                 ;TEST BIT 0
          JR NZ,UNMAG           ;IF SET SET UP UNMAG SPRITES
          LD B,VAL1B            ;VAL FOR REG1 (MAG SPRITES)
          LD C,1                ;REGISTER NO.
          CALL 0047H            ;WRITE B TO REG C
```

```
        LD B,VAL5B          ;VAL FOR REG 5, SPRT ATTR AT 1B00H
        LD C,5
        CALL 0047H
        RET
UNMAG:  LD B,VAL1A
        LD C,1
        CALL 0047H
        LD B,VAL5A
        LD C,5
        CALL 0047H
        RET
```

Not only does the above technique allow us to maintain sprites of two different sizes on screen at the same time, but since we now have two (switched) sprite attribute tables we may have twice the normal number of sprites on screen at any one time to aa maximum of 64. Since only one attribute table is active at aany one time we may now have a maximum of eight sprites on any one horizontal line, provided of course that only four on the same line occur in any one attribute table.

It must be borne in mind that in video systems where the tube persistence is particularly short unacceptable flicker may be introduced - unfortunately you can't win them all.

Quick VDP access: avoiding time problems.

Although the method of access to VRAM provided for by the VDP has the advantage of not taking up valuable CPU address space, it has the drawback that it is slow. It is never possible to completely evade this fact, although its worst effects may usually be avoided. With respect to VRAM access there is one cardinal rule, which whenever possible you should follow:-

MAKE USE OF THE VDP ADDRESS REGISTER AND ACCESS VRAM SEQUENTIALLY

Since sequential VRAM access only requires one data transfer, whereas non-sequential access requires two data transfers to set up the address register every time, sequential access is significantly faster. For example, when writing a routine to scroll the screen it is advantageous to read the entire screen (ie. the name table) into a buffer in CPU RAM and then put it back into VRAM en masse.

Using this method the scroll could be achieved in considerably less than a 50th of a second, whereas a routine using non- sequential VRAM access could not achieve a scroll in less than 3/50ths of a second.

If large amounts of VRAM data are to be manipulated at any one time you should read the data into RAM and perform the manipulation there, before loading it back into VRAM sequentially.

By reference to figure 6.2 it may also be seen that access to the VDP is a great deal faster during the 4,300 microseconds immediately following the frame flyback interrupt. In cases where timing is extremely critical therefore it is advantageous to perform all or most of the necessary VRAM access in an interrupt driven routine.

In summary, fast access to VRAM requires optimisation along the following lines:-

Firstly you should access VRAM as little as possible. Keep a copy of VRAM data which you are likely to need to modify in CPU-addressable RAM.

Secondly you should whenever possible access VRAM sequentially, non-sequential access being time-consuming and with careful thought usually unnecessary.

Finally, if things are really desperate try to access VRAM only during the frame flyback period, since at this time access to VRAM is never held up while the VDP finishes whatever it was last doing.

Chapter 7

The Programmable Sound Generator

The Data Registers

The sound generator chosen for the MSX system is the General Instruments AY-3-8910 (or equivalent). This chip was briefly discussed in chapter 5. The device contains 16 read/write registers which allow the user to produce tone and noise output on any of three separate channels. The sound chip data registers are as follows:-

Register 0: Channel A tone period fine tune.
Register 1: Channel A tone period coarse tune.
Register 2: Channel B tone period fine tune.
Register 3: Channel B tone period coarse tune.
Register 4: Channel C fine tune.
Register 5: Channel C coarse tune.
Register 6: Noise period.
Register 7: Enables and i/o direction.
Register 8: Channel A amplitude and envelope enable.
Register 9: Channel B amplitude and envelope enable.
Register 10: Channel C amplitude and envelope enable.
Register 11: Envelope period fine tune.
Register 12: Envelope period coarse tune.
Register 13: Envelope shape.
Register 14: Data store for port A.
Register 15: Data store for port B.

The Tone Generators (registers 0..5)
Each channel has two tone period registers associated with it. These set the period of the sound to be generated (in units of 8 microseconds). The fine tune registers contain the least significaant eight bits of that period while the coarse tune registers store the most significant four bits. To instruct the chip to mix in the output of a given channel the appropriate bit in the enables register (register 7) must be cleared to 0.

The Noise Generator (register 6)
This is the register which controls the single pseudo-random noise generator with which the chip is provided. The output from this can be mixed into any of the three sound channels, as required, by resetting the relevant bit in register 7. The period of the noise generator is set by the least significant five bits of this register.

The Enables register (register 7).
Register 7 specifies whether tone or noise or both are to be included in the

output of the three channels. It also determines whether the two input/output ports are to be used in input or output mode. The bits are allocated as follows:

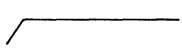
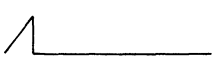
Bit 0: If set disables tone production on channel A.
Bit 1: If set disables tone production on channel B.
Bit 2: If set disables tone production on channel C.
Bit 3: Channel A noise disable.
Bit 4: Channel B noise disable.
Bit 5: Channel C noise disable.
Bit 6: If set determines port A output mode.
Bit 7: If reset determines port B input mode.

Please note that in the MSX system the input/output ports are used to read the joystick ports, and hence should always be set as inputs.

Amplitude control (registers 8..10).
Each channel has its own amplitude control register. Bit 4 of this register specifies whether hardware enveloping is to be used for the channel. Setting bit 4 places the amplitude of the channel under the control of the hardware envelope generator. If the bit is cleared the amplitude is set by software using bits 0-3, 0 being no volume and 15 setting maximum volume.

Envelope generator (registers 11..13).
The sound chip has a single hardware volume envelope generator which may be used by any or all of the three channels, as determined by the channel's

Figure 7.1 Envelope shapes

VALUE	SHAPE
8	
9	
10	
11	
12	
13	
14	
15	

amplitude control register. Bits 0-3 of register 13 control the shape of the envelope in a rather arcane manner. Figure 7.1 gives the values required to generate the eight hardware envelope shapes available. Any other values will duplicate either envelope 9 or 15.

The length of each of the ramps, upwards or downwards, is set by the envelope period.

The envelope period is a full 16-bit value set by register 11 (LSB) and register 12 (MSB). The period is given in 128 microsecond units and is the time between steps in the ramp. As the ramp has 16 steps (volume settings 0-15) the total time taken for the ramp is 1024 microseconds times the envelope period. Thus the envelope period sets the length of the ramp in milliseconds, approximately.

The I/O ports (registers 14..15).
Since the input/output registers are used by the MSX system to read the joystick ports, it is best not to attempt to use them: also they have been buffered strangely to aid their use in the MSX setup.

Notes and tone periods.
The required tone period can be calculated from the frequency required by the formula:-

Period = 125000 / Frequency

And the frequency of any note in the even-tempered scale in the full eight octave range is calculated from the International Standard A as below:-

Frequency = 440*(2^(Octave+(N-10)/12))

Where:

> Octave is the octave number. O is the octave containing middle C, -1 is the octave below, 1 is the octave above and so on.

N is the note number, 1 is C, 2 is C#, 3 is D, etc.

Since the period is an integer value, it follows that values calculated from the above formulae will not produce exactly the required frequency. The errors are however very small and are not usually noticeable.

The table below gives the values for Octave 0 calculated from the above formulae with the error factor given for each note as a percentage of the required frequency.

Note	Frequency	Period	Error
C	261.626	478	0.046%
C#	277.183	451	0.007%
D	293.665	426	0.081%
D#	311.127	402	0.058%
E	329.628	379	0.057%
F	349.228	358	0.019%
F#	369.994	338	0.046%
G	391.995	319	0.037%
G#	415.305	301	0.005%
A	440.000	284	0.032%
A#	466.164	268	0.055%
B	493.883	253	0.038%

Given the above formulae the following BASIC routine will calculate tone periods for notes in the entire eight octave range from string input in the format 'ON'. However to simply string parsing the octave number (the first character in the string) varies from 0 to 7 (low octave to high octave) rather than from –3 to 4.

The right hand side of the string (the note) may be either one or two characters in length, and if two must have the # character as its rightmost character.

```
 10 INPUT A$
 20 GOSUB 1000
 30 PRINT A
 40 GOTO 10
1000 O=VAL(LEFT$(A$,1))
1010 A$=MID$(A$,2)
1020 FOR NC=1 TO 12
1030 READ B$:IF B$=A$ THEN NO=NC
1040 NEXT NC
1050 RESTORE
1060 P=125000/(440*(2^((O-3)+(NO-10)/12))
1070 A=INT(P)
1080 RETURN
1090 DATA C,C#,D,D#,E,F,F#,G,G#,A,A#,B
```

Using an extension of the above routine it is a simple matter to produce a program that will provide tone period data for use in machine code music routines.

Accessing the PSG in the MSX environment.

Since the MSX designers reserve the right to alter the hardware specification of the MSX series computers, as long as they maintain software compatability, it is only really possible to program the PSG by use of the operating system calls provided in the MSX system ROM. A description of these routines follows:-

ADDRESS	FUNCTION
0090H	This routine initialises the PSG, it requires no parameters to be passed and may modify all registers.
0093H	This routine writes the data in E to the PSG register specified by the accumulator, it returns no values and leaves all registers unchanged.
0096H	This routine reads a value from the register specified by the accumulator, returning the data in the accumulator, and leaving all other registers unmodified.

Programming the PSG.

Programming the PSG requires a series of write operations to the chip to specify the tone period for a given channel, the noise period if required, any envelope shape which the user may wish to use, the envelope period and (if envelopes are not to be used) the amplitude of the relevant channel. These actions should be performed with the channel currently being written disabled (by setting the relevant ENABLE bit in register 7). As an example of this the following routine will set up channel A to play middle C with envelope shape 8:-

```
MIDC:   LD A,7
        CALL 0096H          ;READ ENABLES REGISTER
        OR 9                ;SET CHANNEL A TONE & NOISE DISABLE
        LD E,A
        LD A,7
        CALL 0093H          ;PUT IT BACK
        LD E,1
        LD A,1
        CALL 0093H          ;SET UP CHAN A PERIOD COARSE TUNE
        LD E,0DEH
        LD A,0
        CALL 0093H          ;AND FINE TUNE FOR MIDDLE C
        LD E,8
        LD A,13
        CALL 0093H          ;SET UP ENV SHAPE
        LD E,15
        LD A,11
        CALL 0093H          ;FINE TUNE ENV PERIOD
        LD E,0
        LD A,12
        CALL 0093H          ;COARSE TUNE ENV PERIOD
        LD E,16
```

```
        LD A,8
        CALL 0093H          ;ENABLE ENV CONTROL OF CHAN A
        LD A,7
        CALL 0096H          ;READ ENABLES REGISTER
        AND 0FEH            ;RESET CHAN A TONE DISABLE FLAG
        LD E,A
        LD A,7
        CALL 0093H          ;PUT BACK NEW ENABLES VALUE
        RET
```

Although the above routine is rather trivial, it does quite effectively illustrate the techniques of programming the PSG. In the case of a chip such as the AY-3-8910 which has such enormous capabilities the only way to really know it is to experiment. The rest of this chapter will present a series of examples intended to aid in gaining a familiarity with this chip.

Three channel music: the computer as performer.

The following routine is interrupt driven to allow the playing of the three music channels regardless of any other program which may be running. The data for the three music chnnels should be stored at the addresses assigned to the symbols C1DAT, C2DAT and C3DAT as a series of three byte entries. Byte one of each entry is the fine tune value for the note period, byte two is the coarse tune value, and byte three is the duration of the note in units of 20 milliseconds. The end of the tune is signified by writing 255 in the coarse tune byte of the last entry of the data for channel A. While the program is running, and hence the tune is playing, the volume of each channel can be altered by writing to the addresses assigned to the symbols VOL1, VOL2 and VOL3. If you wish to experiment with the hardware envelope generator this may be achieved by setting it up using the BASIC SOUND command and then writing a value of 16 to the volume address of the relevant channel.

```
        INTHOK EQU 0F9DFH   ;INTERRUPT HOOK ADDRESS
        PSGINI EQU 00090H   ;O.S. ROUTINE FOR PSG INITIALISATION
        WRTPSG EQU 00093H   ;DITTO TO WRITE DATA TO PSG
        RDPSG  EQU 00096H   ;AS ABOVE TO READ DATA
        C1DAT  EQU 0B000H   ;START OF CHANNEL 1 TUNE DATA
        C2DAT  EQU 0B400H   ;DITTO CHANNEL 2
        C3DAT  EQU 0B800H   ;AND CHANNEL 3. ALL MAY BE CHANGED
                            ;AS REQUIRED
        ORG 9000H


START:  LD HL,C1DAT
        LD (C1PTR),HL
        LD HL,C2DAT
        LD (C2PTR),HL
        LD HL,C3DAT
        LD (C3PTR),HL       ;SET UP DATA POINTERS
        LD A,1
        LD (C1CTR),A
```

```
        LD (C2CTR),A
        LD (C3CTR),A        ;SET UP NOTE DURATION COUNTERS
        CALL PSGINI         ;INITIALISE PSG
        LD A,LOW MUSROT     ;GET LOW ORDER OF ADDRESS OF MUSIC
                            ;ROUTINE
        LD (INTHOK+1),A     ;WRITE TO INTERRUPT HOOK
        LD A,HIGH MUSROT
        LD (INTHOK+2),A     ;WRITE HIGH ORDER TO INT HOOK
        LD A,ØC3H           ;GET Z8Ø 'JP' INSTRUCTION
        LD (INTHOK),A       ;INTO INTERRUPT HOOK
        RET                 ;INTERRUPT IS NOW SET UP TO DO
                            ;MUSIC ROUTINE

C1PTR:  DEFW Ø
C2PTR:  DEFW Ø
C3PTR:  DEFW Ø
C1CTR:  DEFB Ø
C2CTR:  DEFB Ø
C3CTR:  DEFB Ø
VOL1:   DEFB 15
VOL2:   DEFB 15
VOL3:   DEFB 15             ;SET UP VARIABLE TABLE


MUSROT: PUSH AF             ;SAVE VDP STATUS
        LD A,(C1CTR)        ;BEGINNING OF MUSIC PLAYING ROUTINE
        DEC A               ;DEC NOTE DURATION
        LD (C1CTR),A
        OR A                ;IS DURATION NOW Ø IE. IS THE NOTE
                            ;FINISHED
        JR NZ,CHANB         ;IF NOT DO  CHANNEL 2
        LD A,7              ;IF NOTE COMPLETED
        CALL RDPSG
        OR 1
        LD E,A
        LD A,7
        CALL WRTPSG         ;THEN TURN OFF CHANNEL 1
        LD IX,(C1PTR)       ;GET DATA POINTER IN IX
        LD E,(IX+Ø)         ;GET FINE TUNE  VALUE
        LD A,Ø
        CALL WRTPSG         ;AND WRITE TO REGISTER Ø
        LD A,(IX+1)         ;GET COARSE TUNE VALUE
        CP 255              ;IS IT THE END OF TUNE MARKER
        JR Z,START          ;IF SO REINITIALISE
        LD E,A
        LD A,1
        CALL WRTPSG         ;ELSE WRITE IT TO REGISTER 1
        LD A,(IX+2)         ;SET  UP NOTE DURATION COUNTER
        LD (C1CTR),A
        LD A,(VOL1)         ;GET CHANNEL 1 VOLUME
        LD E,A
```

```
        LD A,8
        CALL WRTPSG          ;WRITE IT TO CHANNEL 1 VOLUME REG
        LD A,7
        CALL RDPSG
        AND ØFEH
        LD E,A
        LD A,7
        CALL WRTPSG          ;AND REENABLE CHAN 1 TONE PRODUCTN
        INC IX
        INC IX
        INC IX               ;POINT TO NEXT  NOTE ENTRY
        LD (C1PTR),IX        ;SAVE POINTER
CHANB:  LD A,(C2CTR)         ;DO CHANNEL 2
        DEC A
        LD (C2CTR),A
        OR A
        JR NZ,CHANC
        LD A,7
        CALL RDPSG
        OR 2
        LD E,A
        LD A,7
        CALL WRTPSG
        LD IX,(C2PTR)
        LD E,(IX+Ø)
        LD A,2
        CALL WRTPSG
        LD E,(IX+1)
        LD A,3
        CALL WRTPSG
        LD A,(VOL2)
        LD E,A
        LD A,9
        CALL WRTPSG
        LD A,7
        CALL RDPSG
        AND ØFDH
        LD E,A
        LD A,7
        CALL WRTPSG
        LD A,(IX+2)
        LD (C2CTR),A
        INC IX
        INC IX
        INC IX
        LD (C2PTR),IX
CCHANC: LD A,(C3CTR)
        DEC A
        LD (C3CTR),A
        OR A
        JR NZ,ENDMUS
        LD A,7
```

```
        CALL RDPSG
        OR 4
        LD E,A
        LD A,7
        CALL WRTPSG
        LD IX,(C3PTR)
        LD E,(IX+Ø)
        LD A,4
        CALL WRTPSG
        LD E,(IX+1)
        LD A,5
        CALL WRTPSG
        LD A,(IX+2)
        LD (C3CTR),A
        LD A,(VOL3)
        LD E,A
        LD A,1Ø
        CALL WRTPSG
        LD A,7
        CALL RDPSG
        AND ØFBH
        LD E,A
        LD A,7
        CALL WRTPSG
        INC IX
        INC IX
        INC IX
        LD (C3PTR),IX
ENDMUS: POP AF              ;RESTORE VDP STATUS

        RET
```

Sound effects on the AY-3-8910.

The production of sound effects is one area where there is absolutely no substitute for experimentation. While it is true that the PSG can produce virtually any sound you could wish for, it may be necessary to spend hours trying various PSG register values before achieving even a close match to the sound required. This procedure is probably best performed in BASIC using the SOUND command, whereby one may quickly and simply change any PSG register. Having found the required register values one may then simply code up the necessary routines to load values into the PSG.

For example the following BASIC program produces the sound of a gunshot:-

```
1Ø SOUND 6,15:SOUND 7,7
2Ø SOUND 8,16:SOUND 9,16:SOUND 1Ø,16
3Ø SOUND 11,Ø:SOUND 12,16:SOUND 13,Ø
```

This may written in Z80 code as follows:-

```
GUN:     LD B,6
         LD HL,GUNTBL
GUNLP:   LD E,(HL)
         LD A,B
         CALL 0093H
         INC HL
         INC B
         LD A,B
         CP 14
         JR NZ,GUNLP
         RET
GUNTBL:  DEFB 15,7,16,16,16,0,16,0
```

Sound generation in software: the one bit sound port.

In addition to the generation of sound in hardware by the PSG the MSX system also sets aside one bit of port C of the 8255 PPI for the generation of sound in software. An operating system call located at 0135H accepts a value in the accumulator, and if the value is 0 turns off the one bit sound port, else turns it on. By strobing this bit on and off rapidly in software sound may be produced; try the following routine:-

```
SFTSND:  LD A,0
         CALL 0135H
         LD A,1
         CALL 0135H
         JR SFTSND
```

On running this routine a high pitched tone should be heard. Try experimenting with turning the port on and off at different rates to see what happens.

Chapter 8

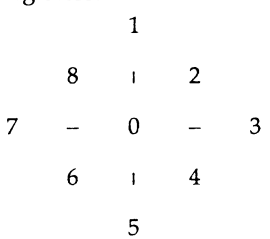
Input-Output: the computer's window on the world.

The MSX system provides a comprehensive set of routines to provide such functions as keyboard reading and scanning, reading the joysticks and game paddles and printing on the screen. This chapter deals firstly with reading the joysticks and finally with general console input/output.

Game I/O: joysticks, paddles and touchpads.

The MSX system supports a maximum of two joysticks, and considers the cursor keys and space bar to be a third. The following routines are used to read the joysticks and trigger buttons.

0D5H Reads the current status of the joystick specified by the accumulator, (0 to 2 with 0 referring to the cursor keys). It returns a value in the accumulator which varies between 0 and 8. See diagram below. A value of 0 indicates that the joystick is centralised. This routine may modify all registers.



0D8H This routine reads the joystick trigger specified by the joystick I.D. in the accumulator. It returns 0 in the accumulator if the trigger is not pressed and 255 if it is. Only the AF register pair is modified. (Note that if this routine is called with a joystick I.D. of 0 then the space bar is checked).

The joystick ports are also designed to be able read game paddles. It is possible in the MSX system to connect six game paddles to each joystick port, making a total of twelve in all. These are numbered from 1 to 12. Odd numbered paddles are connected to joystick port 1, even numbered paddles to port 2.

0DEH This routine reads the game paddle specified by the accumulator,

and returns a value between 0 and 255 (dependent on the position of the relevant paddle) in the acccumulator. It may modify all registers.

Finally in this section the joystick ports may also be used to read NEC PC-6051 compatible touchpads. The following routine is used to read the touchpad status:-

0DBH This routine accepts an ID in the accumulator, and returns a value from the touchpad, modifying all registers. The IDs and values returned are as follows:

ID	*Returns*
0	255 if the touchpad in port 1 is pressed, 0 otherwise.
1	Returns the X coordinate of the point pressed on pad 1.
2	As above but the Y coordinate.
3	Returns 255 if the switch on pad 1 is pressed, or 0 if it is not.
4 to 7	IDs 4 to 7 are the same as above but for the touchpad in port 2.

Console input/output.

The following routines are provided to access the keyboard and CRT;

09CH This routine checks the status of the keyboard buffer. It returns with the zero flag set if the buffer is empty, and modifies the AF pair.

09FH This routine fetches a character from the input buffer, or if the buffer is empty waits for a character to be typed in. It returns the ASCII value of the character typed in the accumulator and modifies the AF pair.

0A2H Prints a character, the code of which is passed in the accumulator, at the present cursor position, leaving all registers unchanged.

0C0H Produces a beep (equivalent of CHR$(7)). It may modify all registers.

0C3H Clears the screen, modifying the AF, BC and DE pairs.

0C6H Positions the screen cursor at column H row L modifying the AF pair.

0CCH Erases the function key display, modifying all registers.

0CFH Displays the function key display, modifying all registers.

The use of these routines is fairly self-evident and the construction of user applications should pose no problems, as they provide all the basic functions necessary for console inputs/outputs; for example a simple input and display routine may be constructed as follows:-

```
        LD HL,Ø
        CALL ØC6H           ;HOME THE CURSOR
        CALL ØC3H           ;CLS
INLP:   CALL Ø9FH           ;GET CHARACTER
        CALL ØA2H           ;PRINT IT
        CP 13
        JR NZ,INLP          ;IF NOT CARRIAGE RETURN GO DO IT
                            ;AGAIN
```

Or a routine might be written to request a password to be typed in before allowing execution to be continued, along the following lines:-

```
PSSWD:  LD HL,PSWORD        ;POINT HL TO PASSWORD STRING
PWLP:   CALL Ø9FH           ;GET A CHARACTER
        CP (HL)             ;DOES IT MATCH THE PASSWORD?
        JR Z,PWSKP          ;IF SO SKIP TO END OF LOOP
        LD A,(HL)
        CP '&'              ;END OF PASSWORD?
        JR Z,GO             ;YES GO TO OTHER ROUTINES
        JP ØØØØH            ;OTHERWISE DO COLD START
PWSKP:  INC HL              ;INCREMENT PASSWORD POINTER
        CALL ØA2H           ;PRINT PREVIOUS CHARACTER
        JR PWLP             ;CHECK NEXT CHARACTER
PSWORD: DEFB 'ANY PASSWORD YOU WANT TERMINATED BY&'

GO:     ;INSERT REST OF ROUTINES HERE
```

Slot Selection.

Since the 8255 PPI need not be addressed at the same location in any two different MSX computers two routines have been made available to facilitate the reading and writing of port A in the 8255 (i.e. the primary slot select register):

0138H — This reads the current value of the slot selection register, returning the value in the accumulator and leaving all other registers unchanged.

013BH — Writes the value in the accumulator to the primary slot selection register, leaving all other registers unchanged.

Thus if one wished to select pages 2 and 3 from slot 2 while leaving pages 0 and 1 unchanged one could perform the following operations:

```
        CALL 138H           ;READ SLOT SELECT
        AND 15              ;MASK OUT BITS FOR PAGES Ø AND 1
        OR 1Ø1ØØØØØB        ;MASK TO SELECT SLOT 2 FOR PAGES 2
                            ;AND 3
        CALL 13BH           ;PUT BACK NEW VALUE
```

There are two other operating system routines dealing with the 8255 PPI, firstly

the routine at 0132H which accepts a value in the accumulator (either zero or non-zero) and if the value is 0 turns off the CAPS lamp, else turning it on. The second routine is used to scan the keyboard.

Keyboard scanning: checking individual keys.

0141H The routine at this location scans the row of the keyboard matrix (see figure 8.1) specified by the value in the accumulator (0 to 9) and returns with a value in the accumulator such that the bit corresponding to a column of the keyboard matrix is reset if the key in that column of the row scanned is pressed. If no keys in the row are pressed, the routine will return 255.

Figure 8.1 The Keyboard Matrix

COLUMNS

ROWS	7	6	5	4	3	2	1	0
0	7	6	5	4	3	2	1	0
1	+	[		¥	–	=	9	8
2	B	A	–	/	›	‹	]	*
3	J	I	H	G	F	E	D	C
4	R	Q	P	O	N	M	L	K
5	Z	Y	X	W	V	U	T	S
6	F3	F2	F1	CODE	CAP	GRAPH	CTRL	SHIFT
7	RETURN	SELECT	BS	STOP	TAB	ESC	FS	F4
8	→	↓	↑	←	DEL	INS	HOME	SPACE
9								

As a clarification of the above consider the following routine, which will scan the keyboard until the 'Z' and 'X' keys are pressed together:

```
ZXCHK:  LD A,5              ;SCAN ROW 5  OF THE KEYBD MATRIX
        CALL 0141H
        AND 10100000B       ;CHECK BITS 7 AND 5
                            ;ARE THEY BOTH RESET
        JR NZ,ZXCHK         ;IF NOT TRY AGAIN
        RET
```

APPENDICES

A .... Character codes

B .... Colour assignments

C .... Video RAM table

D .... Z-80 Instructions

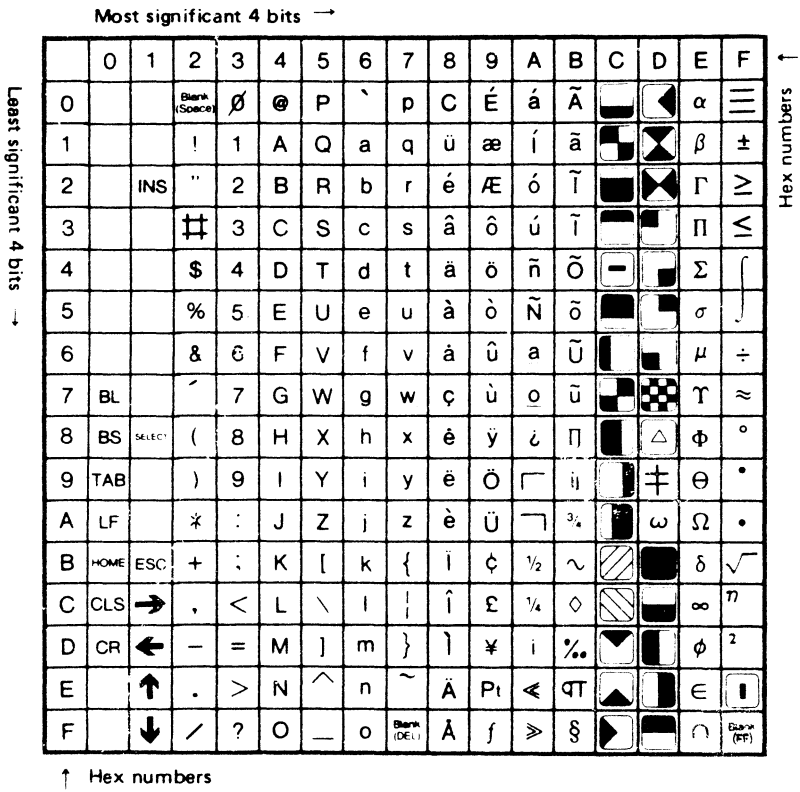
E .... TI 9929A VDP

F .... GI AY-3-8910 PSG

APPENDIX A

CHARACTER CODE TABLE

Reproduced by kind permission of Toshiba UK Limited.



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0			Blank (Space)	Ø	@	P	`	p	C	É	á	Ã	▄	◢	α	═
1			!	1	A	Q	a	q	ü	æ	í	ã	▚	▞	β	±
2		INS	"	2	B	R	b	r	é	Æ	ó	Ĩ	▆	▞	Γ	≥
3			#	3	C	S	c	s	â	ô	ú	Ĩ	▀	▟	Π	≤
4			$	4	D	T	d	t	ä	ö	ñ	Õ	▬	▙	Σ	⌠
5			%	5	E	U	e	u	à	ò	Ñ	õ	▆	▛	σ	⌡
6			&	6	F	V	f	v	å	û	a	Ũ	▌	▙	μ	÷
7	BL		´	7	G	W	g	w	ç	ù	º	ũ	▚	▩	τ	≈
8	BS	SELECT	(	8	H	X	h	x	ê	ÿ	¿	∏	▌	△	Φ	°
9	TAB		)	9	I	Y	i	y	ë	Ö	⌐	ij	▐	╪	Θ	•
A	LF		*	:	J	Z	j	z	è	Ü	¬	¾	▐	ω	Ω	•
B	HOME	ESC	+	;	K	[	k	{	ï	¢	½	~	▨	▇	δ	√
C	CLS	→	,	<	L	\	l	¦	î	£	¼	◊	▧	▃	∞	n
D	CR	←	−	=	M	]	m	}	ì	¥	¡	‰	▼	▌	φ	2
E		↑	.	>	N	^	n	~	Ä	Pt	≪	¶T	▲	▙	∈	▮
F		↓	/	?	O	_	o	Blank (DEL)	Å	ƒ	≫	§	▶	▃	∩	Blank (FF)

APPENDIX B

COLOUR TABLE

0 .... Transparent

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

APPENDIX C

Video RAM Tables

	40 Column Text	32 Column Text	HRG	Multicolour
Name table	0	6144	6144	2048
Pattern table	2048	0	0	0
Colour table	-	8192	8192	-
Sprite Attribute table	-	6912	6912	6912
Sprite Pattern table	-	14336	14336	14336

APPENDIX D

Z-80 Instructions

The entries under each flag have the following meanings:
R: the flag is updated as a result of the operation.
0: the flag is re-set.
1: the flag is set.
C: the status of the carry flag is copied into the H flag.
Other abbreviations are listed at the end of this appendix.

	FLAGS								FLAGS							
	S	Z	H	N	P	C			S	Z	H	N	P	C		
ADC HL,ss	R	R	R	0	R	R		CPI	R	R	R	1	R	-		
ADC A,s	R	R	R	0	R	R		CPIR	R	R	R	1	R	-		
ADD A,n	R	R	R	0	R	R		CPL	-	-	1	1	-	-		
ADD A,r	R	R	R	0	R	R		DAA	R	R	R	-	R	R		
ADD A,(HL)	R	R	R	0	R	R		DEC m	R	R	R	1	R	-		
ADD A,(IX+d)	R	R	R	0	R	R		DEC IX	-	-	-	-	-	-		
ADD A,(IY+d)	R	R	R	0	R	R		DEC IY	-	-	-	-	-	-		
ADD HL,ss	-	-	R	0	-	R		DEC ss	-	-	-	-	-	-		
ADD IX,pp	-	-	R	0	-	R		DI	-	-	-	-	-	-		
ADD IY,rr	-	-	R	0	-	R		DJNZ e	-	-	-	-	-	-		
AND s	R	R	1	0	R	0		EI	-	-	-	-	-	-		
BIT b,(HL)	-	R	1	0	-	-		EX (SP),HL	-	-	-	-	-	-		
BIT b,(IX+d)	-	R	1	0	-	-		EX (SP),IX	-	-	-	-	-	-		
BIT b,(IY+d)	-	R	1	0	-	-		EX (SP),IY	-	-	-	-	-	-		
BIT b,r	-	R	1	0	-	-		EX AF,AF'	R	R	R	R	R	R		
CALL cc,nn	-	-	-	-	-	-		EX DE,HL	-	-	-	-	-	-		
CALL nn	-	-	-	-	-	-		EXX	-	-	-	-	-	-		
CCF	-	-	C	0	-	R		HALT	-	-	-	-	-	-		
CP s	R	R	R	1	R	R		IM 0	-	-	-	-	-	-		
CPD	R	R	R	1	R	-		IM 1	-	-	-	-	-	-		
CPDR	R	R	R	1	R	-		IM 2	-	-	-	-	-	-		

IN A,(N)	-	-	-	-	-	-
IN R,(C)	R	R	R	0	R	-
INC (HL)	R	R	R	0	R	-
INC IX	-	-	-	-	-	-
INC (IX+d)	R	R	R	0	R	-
INC IY	-	-	-	-	-	-
INC (IY+d)	R	R	R	0	R	-
INC r	R	R	R	0	R	-
INC ss	-	-	-	-	-	-
IND	-	R	-	1	-	-
INDR	-	1	-	1	-	-
INI	-	R	-	1	-	-
INIR	-	1	-	1	-	-
JP (HL)	-	-	-	-	-	-
JP (IX)	-	-	-	-	-	-
JP (IY)	-	-	-	-	-	-
JP cc,nn	-	-	-	-	-	-
JP nn	-	-	-	-	-	-
JR C,e	-	-	-	-	-	-
JR e	-	-	-	-	-	-
JR NC,e	-	-	-	-	-	-
JR NZ,e	-	-	-	-	-	-
JR Z,e	-	-	-	-	-	-
LD A,(BC)	-	-	-	-	-	-
LD A,(DE)	-	-	-	-	-	-
LD A,I	R	R	0	0	R	-
LD A,(nn)	-	-	-	-	-	-
LD A,R	R	R	0	0	R	-
LD (BC),A	-	-	-	-	-	-
LD (DE),A	-	-	-	-	-	-
LD (HL),n	-	-	-	-	-	-
LD dd,nn	-	-	-	-	-	-
LD HL,(nn)	-	-	-	-	-	-
LD (HL),r	-	-	-	-	-	-
LD I,A	-	-	-	-	-	-
LD IX,nn	-	-	-	-	-	-
LD IX,(nn)	-	-	-	-	-	-
LD (IX+d),n	-	-	-	-	-	-
LD (IX+d),r	-	-	-	-	-	-
LD IY,nn	-	-	-	-	-	-
LD IY,(nn)	-	-	-	-	-	-
LD (IY+d),n	-	-	-	-	-	-
LD (IY+d),r	-	-	-	-	-	-
LD (nn),A	-	-	-	-	-	-
LD (nn),dd	-	-	-	-	-	-
LD (nn),HL	-	-	-	-	-	-
LD (nn),IX	-	-	-	-	-	-
LD (nn),IY	-	-	-	-	-	-
LD R,A	-	-	-	-	-	-
LD r,(HL)	-	-	-	-	-	-
LD r,(IX+d)	-	-	-	-	-	-
LD r,(IY+d)	-	-	-	-	-	-
LD r,n	-	-	-	-	-	-
LD r,r'	-	-	-	-	-	-
LD SP,HL	-	-	-	-	-	-
LD SP,IX	-	-	-	-	-	-
LD SP,IY	-	-	-	-	-	-
LDD	-	-	0	0	R	-
LDDR	-	-	0	0	R	-
LDI	-	-	0	0	R	-
LDIR	-	-	0	0	R	-
NEG	R	R	R	1	R	R
NOP	-	-	-	-	-	-
OR s	R	R	0	0	R	0
OTDR	-	1	-	1	-	-
OTIR	-	1	-	1	-	-
OUT (C),r	-	-	-	-	-	-
OUT (n),A	-	-	-	-	-	-
OUTD	-	R	-	1	-	-
OUTI	-	R	-	1	-	-
POP IX	-	-	-	-	-	-
POP IY	-	-	-	-	-	-
POP qq	-	-	-	-	-	-
PUSH IX	-	-	-	-	-	-
PUSH IY	-	-	-	-	-	-
PUSH qq	-	-	-	-	-	-
RES b,m	-	-	-	-	-	-
RET	-	-	-	-	-	-
RET cc	-	-	-	-	-	-
RETI	-	-	-	-	-	-
RETN	-	-	-	-	-	-
RL m	R	R	0	0	R	R

RLA	-	-	0	0	-	R
RLC (HL)	R	R	0	0	R	R
RLC (IX+d)	R	R	0	0	R	R
RLC (IY+d)	R	R	0	0	R	R
RLC,r	R	R	0	0	R	R
RLCA	-	-	0	0	-	R
RLD	R	R	0	0	R	R
RR m	R	R	0	0	R	R
RRA	-	-	0	0	-	R
RRC m	R	R	0	0	R	R
RRCA	-	-	0	0	-	R
RRD	R	R	0	0	R	R
RST p	-	-	-	-	-	-
SBC A,s	R	R	R	1	R	R
SBC HL,ss	R	R	R	1	R	R
SCF	-	-	0	0	-	1

SET b,(HL)	-	-	-	-	-	-
SET b,(IY+d)	-	-	-	-	-	-
SET b,r	-	-	-	-	-	-
SLA m	R	R	0	0	R	R
SRA m	R	R	0	0	R	R
SRL m	R	R	0	0	R	R
SUB s	R	R	R	1	R	R
XOR s	R	R	R	1	R	R
SET b,(IX+d)	-	-	-	-	-	-

ABBREVIATIONS:

r — Register: A, B, C, D, E, H or L

n — A value in the inclusive integer range 0 to 255 (unsigned).

s — r, n, (HL), (IX+d) or (IY+d).

m — A, B, C, E, H, L, (HL), (IX+d), (IY+d)

dd — BC, DE, HL, SP [LD HL,(dd) LD (nn),dd]

nn — A value in the inclusive integer range 0 to 65535.

qq — BC, DE, HL, AF [POP qq PUSH qq]

ss — BC, DE, HL, SP [ADC HL,ss ADD HL,ss DEC ss INC ss SBC HL,ss]

pp — BC, DE, IX, SP [ADD IX,pp]

rr — BC, DE, IY, SP [ADD IY,rr]

b — bit 0 to 7

e — A signed displacement byte in the range -126 to 129.

p — A zero page address given with the RESTART instruction. Must be a multiple of eight in the range 0 to 56 inclusive.

cc — Condition code: C, NC, Z, NZ, P, M, PE, PO

APPENDIX E

Extract from the TMS 9118/9128/9128 Data Manual

(reproduced by kind permission of Texas Istruments Ltd.)

2. ARCHITECTURE

The TMS9118 Video Display Processor (VDP) is designed to provide a simple interface between a microprocessor and a raster-scanned color monitor or television. The TMS9128/9129 VDPs are designed as simple interfaces between a microprocessor and a R-G-B monitor or a video encoder which produces the video signal to drive the video monitor. Figure 2-1 is a block diagram of the major portions of the VDP architecture interfaces to the CPU, VRAM, and color television.

2.1 CPU INTERFACE

The VDP interfaces to the CPU using an 8-bit bidirectional data bus, three control lines, and an interrupt. Through these interfaces the CPU can conduct four operations:

1) Write to one of the eight VDP write-only registers
2) Read the VDP Status Register
3) Write display data bytes to VRAM
4) Read display data bytes from VRAM

Each of these operations requires that one or more data transfers take place over the CPU/VDP data bus interface. The interpretation of the data transfer is determined by the three control lines of the VDP.

NOTE

The CPU can communicate with the VDP simultaneously and asynchronously, with respect to the VDP's screen refresh operations. The VDP performs memory management and allows periodic intervals of CPU access to VRAM, even in the middle of a raster scan.

2.1.1 CPU/VDP Data Bus

The CPU transfers data between itself and the VDP via an 8-bit bidirectional data bus. CD0 (MSB) through CD7 (LSB) represent the eight bits of this bus as shown in Table 2-1 and Table 2-2.

NOTE

Throughout this manual, the CPU/VDP interface is described assuming the use of TI chips. Other CPU manufacturers assign data lines D0 to be the LSB and D7 to be the MSB. The proper hookup is shown in Figure 2-2. If not connected properly, there will be no video display.

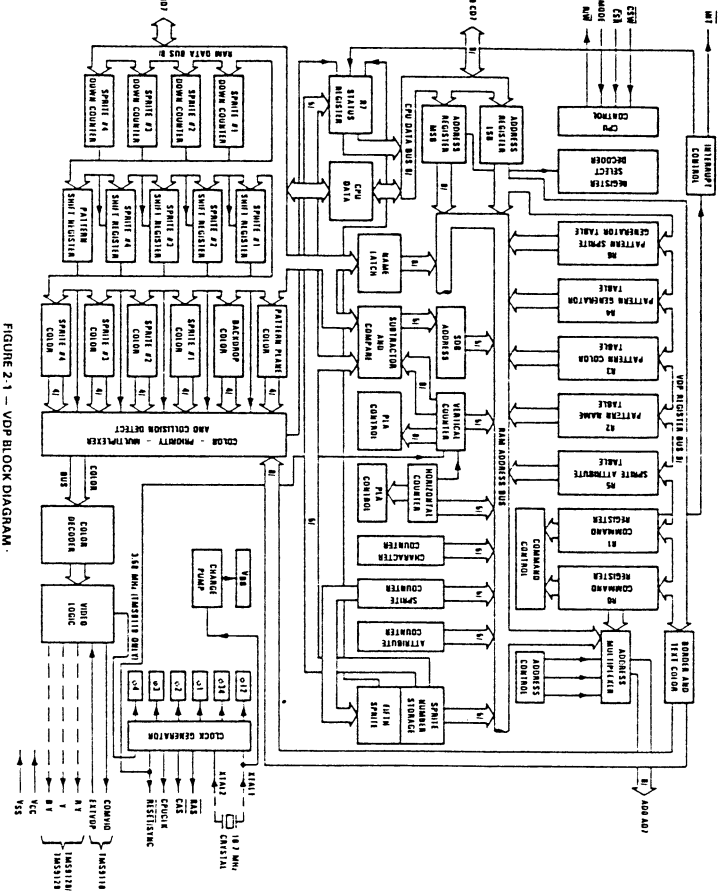


FIGURE 2-1 — VDP BLOCK DIAGRAM

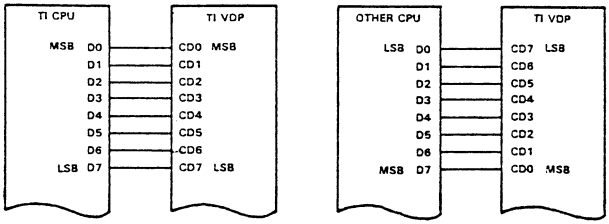


FIGURE 2-2 — CPU TO VDP CONNECTION

2.1.2 CPU Interface Control Signals

The $\overline{CSW}$, $\overline{CSR}$, and MODE inputs control the type and direction of data transfers. $\overline{CSW}$ is the CPU-to-VDP write select. When it is active (low), the eight bits on CD0-CD7 are strobed into the VDP. $\overline{CSR}$ is the CPU-from-VDP read select. When it is active (low), the VDP outputs eight bits on CD0-CD7 to the CPU. $\overline{CSW}$ and $\overline{CSR}$ should never be simultaneously low. If both are low, the VDP outputs invalid data on CD0-CD7 and latches in invalid data.

The MODE input determines the source or destination of a read or write data transfer. This input is normally tied to a CPU low order address line.

2.1.3 VDP Registers

The VDP has eight write-only registers and one read-only status register. The write-only registers control the VDP operation and determine the allocation of the VRAM. The status register contains the interrupt pending, sprite coincidence, and fifth sprite status flags. Section 2.2 and Section 2.3 contain more detailed descriptions of the write-only registers and status register, respectively.

Each of the eight VDP write-only registers can be loaded using two 8-bit data transfers from the CPU. Table 2-3 describes the required format for the two bytes. The first byte transferred is the data byte, and the second byte transferred controls the destination. The MSB of the second byte must be a 1.

The next four bits are 0s, and the lowest three bits make up the destination register number (from 0 to 7). The MODE input is high for both byte transfers.

NOTE

If the two MSBs of the second byte indicate that a write to register is in process, the first byte is not interpreted as a CPU address byte.

2.1.4 CPU Write to VRAM

The CPU transfers data to the VRAM through the VDP using a 14-bit autoincrementing address register. The address register setup requires 2-byte transfers. Each address setup takes two microseconds for a total of four microseconds (see Table 2-3). A 1-byte transfer is then required to write the data to the addressed VRAM byte. The time required is dependent on the MODE. The address register is then autoincremented. Sequential VRAM writes require only 1-byte transfers, since the address register is already set up. During setup of the address register, the two MSBs of the second address byte must be 0 and 1, respectively. MODE is high for both address transfers and low for the data transfer. $\overline{CSW}$ is used in all transfers to strobe the 8 bits into the VDP (see Table 2-3).

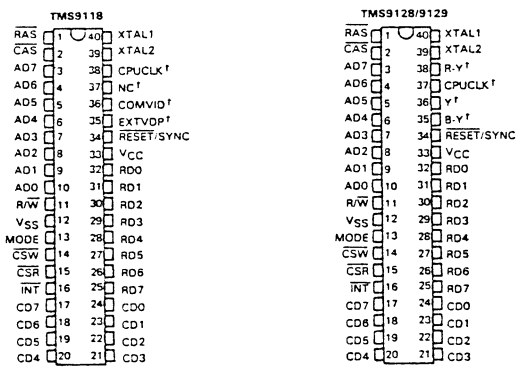


FIGURE 2-3 – TMS9118/9128/9129 VDP PIN ASSIGNMENTS

TABLE 2-3 — CPU/VDP DATA TRANSFERS AND SETUP SEQUENCES

OPERATION	BIT 0	1	2	3	4	5	6	7	$\overline{CSW}$	$\overline{CSR}$	MODE	REQ TIME
WRITE TO VDP REGISTER												
BYTE 1 DATA WRITE	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	0	1	1	2 μs
BYTE 2 REGISTER SELECT	1	0	0	0	0	RS_0	RS_1	RS_2	0	1	1	2 μs
WRITE TO VRAM												
BYTE 1 ADDRESS SETUP	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	0	1	1	2 μs
BYTE 2 ADDRESS SETUP	0	1	A_0	A_1	A_2	A_3	A_4	A_5	0	1	1	2 μs
BYTE 3 DATA WRITE	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	0	1	0	–
READ FROM STATUS REGISTER												
BYTE 1 DATA READ	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	1	0	1	2 μs
READ FROM VRAM												
BYTE 1 ADDRESS SETUP	A_6	A_7	A_8	A_9	A_{10}	A_{11}	A_{12}	A_{13}	0	1	1	2 μs
BYTE 2 ADDRESS SETUP	0	0	A_0	A_1	A_2	A_3	A_4	A_5	0	1	1	–
BYTE 3 DATA READ	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	1	0	0	–

2.1.5 CPU Read from VDP Status Register

As shown in Table 2-3, the CPU can read the contents of the status register with a single-byte transfer. MODE is high for the transfer. $\overline{CSR}$ is used to signal the VDP that a read operation is required. Each status register read requires a two microsecond minimum.

2.1.6 CPU Read from VRAM

The CPU reads from the VRAM through the VDP using the autoincrementing address register. Once the address is set up, a 1-byte transfer is all that is required to read the addressed VRAM byte. The address register is then autoincremented. Sequential VRAM data reads require only a 1-byte transfer since the address register is already set up. During setup of the address register, the two MSBs of the second address byte must be 0. By setting up the address in this way, a read cycle to VRAM is initiated, and read data will be available for the first data transfer to the CPU (see Table 2-3). MODE is high for the address byte transfers and low for the data transfers.

The CPU interacts with VRAM memory through the VDP. It takes four microseconds for the write address setup and two microseconds for the read address setup. The total time necessary for the CPU to transfer a byte of data to or from VRAM memory can vary from two to eight microseconds. Once the VDP has been told to write or read a byte of data to or from VRAM, it takes approximately two microseconds until the VDP is ready to make the data transfer. This time is measured from the leading edge of $\overline{CSW}$ of byte 3 for a write and from the leading edge of CSW of byte 2 for a read. In addition to this two-microsecond delay, the VDP must wait for a CPU access window (i.e., the period of time when the VDP is not occupied with memory refresh or screen display, and is available) to read or write data.

The worst-case time between windows occurs during the Graphics I or Graphics II mode when sprites are being used (refer to Table 2-4). During the active display, CPU windows occur once every 16 memory cycles, giving a maximum delay of six microseconds (a memory cycle takes about 372 nanoseconds). In the Text mode, the CPU windows occur at least once every three memory cycles or a worst-case delay of about 1.1 microseconds. Finally, in the Multicolor mode, CPU windows occur at least once every four memory cycles.

If the user needs to access memory in two microseconds, there are two situations where the time waiting for an access window is effectively zero. Both are independent of the display mode being used.

The first situation occurs when the blank bit of register 1 is 0. With this bit low, the entire screen will show only border color, and the VDP need not wait for a CPU access window.

The second situation occurs during the vertical retrace interval. The VDP issues an interrupt output at the end of each active area. This signal indicates that the VDP is entering the vertical-retrace mode, and that for the next 4.3 milliseconds there is no waiting for an access window. If the user wants the CPU to access memory during this interval, the controlling CPU must monitor the interrupt output of the VDP (the CPU can either poll this output, or use it as an interrupt input).

The program that monitors the interrupt output must allow for its own delays in responding to the interrupt signal, and recognize how much time remains of the 4.3-millisecond refresh period. The CPU must write a 1 to the interrupt-enable bit of Register 1 at initialization, to enable the interrupt for each frame. It must then read the status register each time an inter-

TABLE 2-4 — MEMORY ACCESS DELAY TIMES

CONDITION	MODE	VDP DELAY	TIME WAITING FOR AN ACCESS WINDOW	TOTAL TIME
Active Display Area	Text	2 μs	0 - 1.1 μs	2 - 3.1 μs
Active Display Area	Graphics I, II	2 μs	0 - 5.95 μs	2 - 8 μs
4300 μs after Vertical Interrupt Signal	All	2 μs	0 μs	2 μs
Register I Blank Bit 0	All	2 μs	0 μs	2 μs
Active Display Area	Multicolor	2 μs	0 - 1.5 μs	2 - 3.5 μs

2.1.7 Interrupt Driven CPUs

In an interrupt-driven environment (CPUs accepting interrupts), it is possible for an interrupt to occur before any one of the sequences in Table 2-3 is finished. For example, an interrupt may occur immediately after loading address byte 1 or byte 2 during a write to VRAM operation. In this case, the interrupt service routine does not know where the interrupt occurred within the sequence. Therefore, it is necessary to disable and enable interrupts before and after every setup sequence. This action prevents loss of continuity between the CPU and the VDP.

If this continuity is not important, the status register may be read. The internal CPU interface logic is then reinitialized and will accept the next byte as the first event in another CPU-to-VDP interface. If neither technique is acceptable, polling may be used.

2.1.8 VDP Interrupt

The VDP's $\overline{\text{INT}}$ output pin is used to generate an interrupt at the end of each active display scan (approximately every 1/60 second for the TMS9118/9128, and 1/50 second for the TMS9129). The $\overline{\text{INT}}$ output is active when the Interrupt Enable bit (IE) in VDP Register 1 is a 1 and the F status flag of the status register is set to a 1. Interrupts are cleared when the register is read.

The VDP interrupt occurs at the end of the active pattern plane display, before the last few lines of the backdrop are displayed. This interrupt can be used for moving sprites and updating the pattern display, but is not useful for quickly changing the backdrop color.

2.1.9 $\overline{\text{RESET}}$/SYNC

The VDP is externally initialized whenever the $\overline{\text{RESET}}$/SYNC input is active (low) and must be held low for a minimum of three microseconds. The external reset synchronizes all clocks with its falling edge, sets the horizontal and vertical counters to known states, and clears VDP registers 0 and 1. The video display is automatically blanked since the BLANK bit in VDP register 1 becomes a 0. However, the VDP, continues to refresh the VRAM even though the display is blanked. To restore the picture, write the correct values to VDP registers 0 and 1. While the $\overline{\text{RESET}}$/SYNC line is active, the VDP does not refresh the VRAM.

2.2 WRITE-ONLY REGISTERS

The eight VDP write-only registers are shown in Figure 2-4. They are loaded by the CPU as described in Section 2.1.3. Registers 0 and 1 contain flags to enable or disable various VDP features and modes. Registers 2 through 6 define the base addresses for several sub-blocks within VRAM. These sub-blocks form tables which are used to produce the desired image on the screen. The contents of these tables must be provided by the microprocessor. Register 7 is used to define backdrop and text colors.

Each register is described in the following sections.

2.2.1 Register 0

Register 0 contains two VDP option control bits. All other bits are reserved for future use and must be 0s.

BIT 6 M3 (mode bit 3) (see Section 2.2.2 for description)

BIT 7 External VDP Plane enable/disable (see Section 3.7 for explanation)

0 disables External VDP Plane 1 enables External VDP Plane

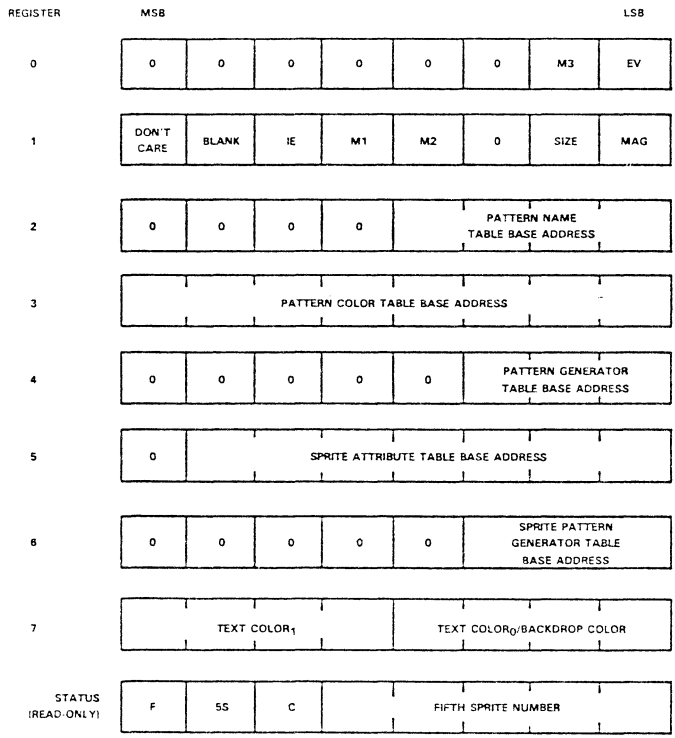


FIGURE 2-4 -- VDP REGISTERS

2.2.2 Register 1 (contains 6 VDP option control bits)

BIT 0 Unused bit

BIT 1 BLANK enable/disable

0 causes the active display area to blank, leaving only the border color to be displayed on the screen.
1 enables the active display

BIT 2 IE (Interrupt Enable)

0 disables VDP interrupt
1 enables VDP interrupt

BIT 3,4 M1, M2 (mode bits 1 and 2)

M1, M2 and M3 determine the operating mode of the VDP:

M1	M2	M3	
0	0	0	Graphics I mode
0	0	1	Graphics II mode
0	1	0	Multicolor mode
1	0	0	Text mode

BIT 5 Reserved

BIT 6 Size (sprite size select)

0 selects Size 0 sprites (8 x 8 bit)
1 selects Size 1 sprites (16 x 16 bits)

BIT 7 MAG (Magnification option for sprites)

0 selects MAG0 sprites (1X)
1 selects MAG1 sprites (2X)

2.2.3 Register 2

Register 2 defines the base address of the Pattern Name Table sub-block. The Pattern Name Table contains the name of or the pointer to a pattern definition in the Pattern Generator Table. The range of Register 2 is from 0 to 15. The contents of the register form the upper 4 bits of the 14-bit Pattern Name Table addresses; thus the Pattern Name Table base address is equal to (Register 2) *400(hex). Table 2-5 shows the possible starting addresses for the Pattern Name Table sub-block.

TABLE 2-5 — REGISTER 2 ADDRESSING

$R2 * 400_{(16)}$ = START ADDRESS

R2	ADDRESS	
00	0000	
01	0400	
02	0800	
03	0C00	– MAXIMUM NUMBER FOR 4K RAMS
04	1000	
05	1400	
06	1800	
07	1C00	
08	2000	
09	2400	
0A	2800	
0B	2C00	
0C	3000	
0D	3400	
0E	3800	
0F	3C00	– MAXIMUM NUMBER

2.2.4 Register 3

Register 3 defines the base address of the Pattern Color Table sub-block. The Pattern Color Table defines the color of the 1s and 0s. The range of Register 3 is from 0 to 255. The contents of the register form the upper 8 bits of the 14-bit Pattern Color Table addresses; thus the Pattern Color Table base address is equal to (Register 3) *40(hex). Table 2-6 shows the possible starting addresses for the Pattern Color Table except for Graphics II mode.

NOTE

Register 3 functions differently when the VDP is in Graphics II mode. In this mode, the Pattern Color Table can only be located in one of two places in VRAM, either >0000 or >2000. If >0000 is the Pattern Color Table location, then the MSB in register 3 must be 0. If >2000 is the location choice, then the MSB in register 3 must be one. Bits 1 to 7 in register 3 must be set to 1. Therefore, in Graphics II mode, the only two values that work correctly in register 3 are >7F and >FF.

2.2.5 Register 4

Register 4 defines the base address of the Pattern Generator Table sub-block. The Pattern Generator Table contains a library of patterns that can be displayed on the screen. The range of Register 4 is 0 through 7. The contents of the register form the upper 3 bits of the 14-bit Generator addresses; thus the Pattern Generator Table base address is equal to (Register 4) *800(hex). Table 2-7 shows the possible starting addresses for the Pattern Generator Table sub-block except for Graphics II mode.

NOTE

Register 4 functions differently when the VDP is in Graphics II mode. In this mode, the Pattern Generator Table can only be located in one of two places in

VRAM, either >0000 or >2000. If >0000 is where the Pattern Generator Table is to be located, then bit 5 in register 4 must be 0. If >2000 is the location choice, then bit 5 in register 4 must be one. In either case, bits 6 and 7 in register 4 must be set to 1s. Therefore, in Graphics II mode, the only two values that work correctly in register 4 are >03 and >07.

CAUTION

Bit 5 of Register 4 MUST have the opposite value of bit 0 of Register 3 so that the two 8K areas are separate. Otherwise, Graphics II mode will not work.

TABLE 2-7 — REGISTER 4 ADDRESSING

(R4) * $800_{(16)}$ = START ADDRESS

R4	START ADDRESS	
00	0000	
01	0800	– Max = for 4K RAMS
02	1000	
03	1800	
04	2000	
05	2800	
06	3000	
07	3800	– Max = for 16K RAMS

2.2.6 Register 5

Register 5 defines the base address of the Sprite Attribute Table sub-block. The Sprite Attribute Table specifies where the sprite goes on the screen. The range of Register 5 is from 0 through 127. The contents of the register form the upper 7 bits of the 14-bit Sprite Attribute Table addresses; thus the base address is equal to (Register 5) *80(hex).

2.2.7 Register 6

Register 6 defines the base address of the Sprite Pattern Generator sub-block. The Sprite Pattern Generator Table describes what the sprite looks like. The range of Register 6 is 0 through 7. The contents of the register form the upper 3 bits of the 14-bit Sprite Pattern Generator addresses; thus the Sprite Pattern Generator base address is equal to (Register 6) *800(hex). Table 2-9 shows the possible starting addresses for the Sprite Pattern Generator sub-block.

TABLE 2-9 — REGISTER 6 ADDRESSING

STARTING ADDRESS = R6 *<800

R6	START ADDRESS	
00	0000	
01	0800	– Max = for 4K DRAMS
02	1000	
03	1800	
04	2000	
05	2800	
06	3000	
07	3800	– Max = for 16K RAMS

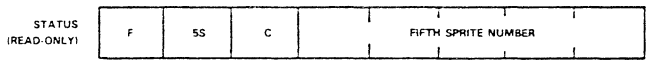
2.2.8 Register 7

The upper 4 bits of Register 7 contain the color code of color 1 in the Text mode. The lower 4 bits contain the color code for color 0 in the Text mode, backdrop color, and border color in all other modes.

2.3 STATUS REGISTER

The VDP has a single 8-bit read-only status register that can be accessed by the CPU. The status register contains the interrupt flag (F), the sprite coincidence flag (C), the fifth sprite flag (5S), and the fifth sprite number (if one exists). The format of the status register is

shown below and is discussed in the following paragraphs.



The status register may be read at any time to test the F, C, and 5S status flags. Reading the status register will clear the interrupt flag, F.

CAUTION

Asynchronous reads will cause the interrupt flag (F) to be reset and missed. Therefore, the status register should be read only when the VDP interrupt is pending.

2.3.1 Interrupt Flag (F)

The F interrupt flag in the status register is set to 1 at the end of the raster scan of the last line of the active display. It is reset to a 0 after the status register is read or when the VDP is externally reset. If the Interrupt Enable bit (IE) in VDP Register 1 is active (1), the VDP interrupt output (INT) will be active (low) whenever the F interrupt flag is a 1.

NOTE

The status register needs to be read frame by frame in order to clear the interrupt and receive the new interrupt of the next frame.

2.3.2 Sprite Coincidence Flag (C)

The Sprite Coincidence Flag C in the status register is set to a 1 if two or more sprites coincide. Coincidence occurs if any two sprites on the screen have one overlapping pixel. Transparent colored sprites, as well as those that are partially or completely off the screen, are also considered. To make an individual sprite invisible, it is necessary to select transparent as its color. The C flag is cleared to a 0 after the status register is read or the VDP is externally reset. The setting of the C flag will not cause an interrupt.

Sprites beyond the Sprite Attribute Table terminator (>D0) are not considered. For example, if a >D0 terminator is put in the vertical position byte of Sprite 3 in the Sprite Attribute Table, Sprites 4 through 31 will be invisible.

CAUTION

The status register should be read immediately upon powerup to ensure that the Coincidence flag is reset.

The VDP checks each pixel position for coincidence during the generation of the pixel regardless of where it is located on the screen. This occurs every 1/60th of a second for the TMS9118 and TMS9128, and every 1/50th of a second for the TMS9129. Thus, when moving sprites more than one pixel position during these intervals, it is possible for the sprites to have multiple pixels overlapping or even to have passed completely over one another when the VDP checks for coincidence.

2.3.3 Fifth Sprite Flag (5S) and Number

The 5S Fifth Sprite Flag in the status register is set to a 1 when there are five or more sprites on the same horizontal line (lines 1 to 192) and the interrupt flag is equal to a 0. The fifth sprite flag is set even if sprites are positioned off-screen. The 5S flag is cleared to a 0 after the status register is read, or the VDP is externally reset. The number of the fifth sprite is placed into the lower 5 bits of the status register when the 5S flag is set, and is valid whenever the 5S flag is 1. The setting of the fifth sprite flag will not cause an interrupt.

2.3.4 Oscillator and Clock Generation

The VDP is designed to operate with a 10.738635 (+/- 0.0005) MHz crystal input to generate the required internal clock signals. A fundamental-frequency parallel-mode crystal is required for the internal clock oscillator, which is the master time base for all system operations. This master clock is divided by two to generate the pixel clock (5.3 MHz) and by three to provide the CPUCLK (3.58 MHz for TMS9118 only).

NOTE

Crystals for the TMS9118/9128/9129 may be purchased from one of the following companies or their authorized distributors:

NDK
10080 North Wolfe Road
Suite 220
Cupertino, California 95014
Telephone: 408-255-0831
Telex: 352057

CTS Knight Inc.
400 Reimann Avenue
Sandwich, Illinois 60548
Telephone: 815-786-8411

APPENDIX F

Extract from the AY-3-8910 Programmable Sound Generator Manual

(reproduced by kind permission of General Istruments Microelectronics Ltd.)

1.2 Features

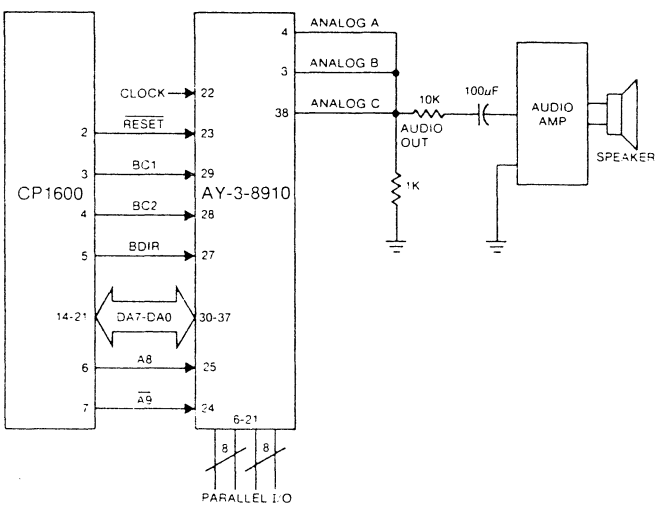
- ☐ Full software control of sound generation.
- ☐ Interfaces to most 8-bit and 16-bit microprocessors.
- ☐ Three independently programmed analog outputs.
- ☐ Two 8-bit general purpose I/O ports (AY-3-8910).
- ☐ One 8-bit general purpose I/O port (AY-3-8912).
- ☐ Single +5 Volt Supply.

1.3 Scope

This Data Manual is intended to introduce the techniques needed to cause the AY-3-8910/8912 Programmable Sound Generator to perform in its intended fashion. All of the programs, programming, and hardware designs have been tested to ensure that the methods are practical rather than purely theoretical.

Although the techniques described will produce powerful results, the range of sounds to be synthesized is so vast and the PSG capabilities so varied that this guide should be viewed merely as an introduction to the applications possibilities of the PSG.

Fig. 1 TYPICAL SYSTEM DIAGRAM



2.2
Pin Assignments

The AY-3-8910 is supplied in a 40 lead dual in-line package with the pin assignments as shown in Fig. 4. The AY-3-8912 is supplied in a 28 lead dual in-line package with the pin assignments as shown in Fig. 5.

Fig. 4 AY-3-8910 PIN ASSIGNMENTS

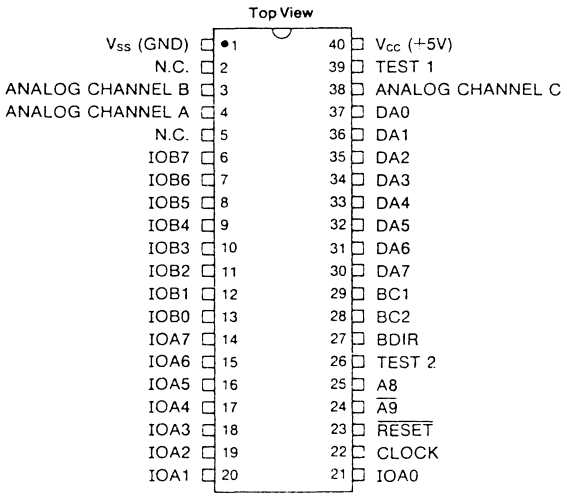
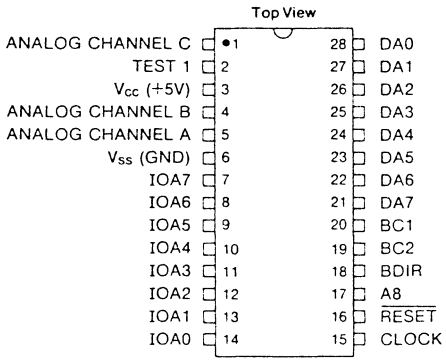


Fig. 5 AY-3-8912 PIN ASSIGNMENTS



2.3
Pin Functions

DA7--DA0 (input/output/high impedance): pins 30--37 (AY-3-8910)
Data/Address 7--0: pins 21--28 (AY-3-8912)

These 8 lines comprise the 8-bit bidirectional bus used by the microprocessor to send both data and addresses to the PSG and to receive data from the PSG. In the data mode, DA7--DA0 correspond to Register Array bits B7--B0. In the address mode, DA3--DA0 select the register # ($0--17_8$) and DA7--DA4 in conjunction with address inputs $\overline{A9}$ and A8 form the high order address (chip select).

A8 (input): pin 25 (AY-3-8910)
pin 17 (AY-3-8912)

$\overline{A9}$ (input): pin 24 (AY-3-8910)
(not provided on AY-3-8912)

Address 9, Address 8

These "extra" address bits are made available to enable the positioning of the PSG (assigning a 16 word memory space) in a total 1,024 word memory area rather than in a 256 word memory area as defined by address bits DA7--DA0 alone. If the memory size does not require the use of these extra address lines they may be left unconnected as each is provided with either an on-chip pull down ($\overline{A9}$) or pull-up (A8) resistor. In "noisy" environments, however, it is recommended that $\overline{A9}$ and A8 be tied to an external ground and +5V, respectively, if they are not to be used.

$\overline{\text{RESET}}$ (input): pin 23 (AY-3-8910)
pin 16 (AY-3-8912)

For initialization/power-on purposes, applying a logic "0" (ground) to the $\overline{\text{Reset}}$ pin will reset all registers to "0". The $\overline{\text{Reset}}$ pin is provided with an on-chip pull-up resistor.

CLOCK (input): pin 22 (AY-3-8910)
pin 15 (AY-3-8912)

This TTL-compatible input supplies the timing reference for the Tone, Noise and Envelope Generators.

BDIR, BC2, BC1 (inputs): pins 27,28,29 (AY-3-8910)
pins 18,19,20 (AY-3-8912)

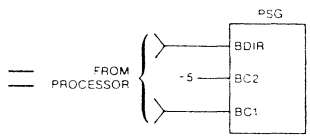
Bus DIRection, Bus Control 2,1

These bus control signals are generated directly by GI's CP1600 series of microprocessors to control all external and internal bus operations in the PSG. When using a processor other than the CP1600, these signals can be provided either by comparable bus signals or by simulating the signals on I/O lines of the processor. The PSG decodes these signals as illustrated in the following:

BDIR	BC2	BC1	CP1600 FUNCTION	PSG FUNCTION
0	0	0	NACT	INACTIVE. See 010 (IAB) below.
0	0	1	ADAR	LATCH ADDRESS. See 111 (INTAK) below.
0	1	0	IAB	INACTIVE. The PSG/CPU bus is inactive. DA7--DA0 are in a high impedance state.
0	1	1	DTB	READ FROM PSG. This signal causes the contents of the register which is currently addressed to appear on the PSG/CPU bus. DA7--DA0 are in the output mode.
1	0	0	BAR	LATCH ADDRESS. See 111 (INTAK) below.
1	0	1	DW	INACTIVE. See 010 (IAB) above.
1	1	0	DWS	WRITE TO PSG. This signal indicates that the bus contains register data which should be latched into the currently addressed register. DA7--DA0 are in the input mode.
1	1	1	INTAK	LATCH ADDRESS. This signal indicates that the bus contains a register address which should be latched in the PSG. DA7--DA0 are in the input mode.

While interfacing to a processor other than the CP1600 would simply require simulating the above decoding, the redundancies in the PSG functions vs. bus control signals can be used to advantage in that only four of the eight possible decoded bus functions are required by the PSG. This could simplify the programming of the bus control signals to the following, which would only require that the processor generate two bus control signals (BDIR and BC1, with BC2 tied to +5V):

BDIR	BC2	BC1	PSG FUNCTION
0	1	0	INACTIVE.
0	1	1	READ FROM PSG.
1	1	0	WRITE TO PSG.
1	1	1	LATCH ADDRESS.



ANALOG CHANNEL A, B, C (outputs): pins 4, 3, 38 (AY-3-8910)
pins 5, 4, 1 (AY-3-8912)

Each of these signals is the output of its corresponding D/A Converter, and provides an up to 1V peak-peak signal representing the complex sound waveshape generated by the PSG.

IOA7--IOA0 (input/output): pins 14--21 (AY-3-8910)
pins 7--14 (AY-3-8912)

IOB7--IOB0 (input/output): pins 6--13 (AY-3-8910)
(not provided on AY-3-8912)

Input/Output A7--A0, B7--B0

Each of these two parallel input/output ports provides 8 bits of parallel data to/from the PSG/CPU bus from/to any external devices connected to the IOA or IOB pins. Each pin is provided with an on-chip pull-up resistor, so that when in the "input" mode, all pins will read normally high. Therefore, the recommended method for scanning external switches, for example, would be to ground the input bit.

TEST 1: pin 39 (AY-3-8910)
pin 2 (AY-3-8912)
TEST 2: pin 26 (AY-3-8910)
(not connected on AY-3-8912)

These pins are for GI test purposes only and should be left open—do not use as tie-points.

V_{CC}: pin 40 (AY-3-8910)
pin 3 (AY-3-8912)

Nominal +5Volt power supply to the PSG.

V_{SS}: pin 1 (AY-3-8910)
pin 6 (AY-3-8912)

Ground reference for the PSG.

2.4 Bus Timing

Since the PSG functions are controlled by commands from the system processor, the common data/address bus (DA7--DA0) requires definition as to its function at any particluar time. This is accomplished by the processor issuing bus control signals, previously described, defining the state of the bus; the PSG then decodes these signals to perform the requested task.

The conditioning of these bus control signals by the processor is the same as if the processor were interacting with RAM: (1) the processor outputs a memory address; and (2) the processor either outputs or inputs data to/from the memory. The "memory" in this case is the PSG's array of 16 read/write control registers.

The timing relationships in issuing the bus control signals relative to the data or address signals on the bus are reviewed in general in the following section, and in detail in Section 7, Electrical Specifications.

Fig. 3 PSG REGISTER ARRAY

REGISTER \ BIT		B7	B6	B5	B4	B3	B2	B1	B0
R0	Channel A Tone Period	8-BIT Fine Tune A							
R1						4-BIT Coarse Tune A			
R2	Channel B Tone Period	8-BIT Fine Tune B							
R3						4-BIT Coarse Tune B			
R4	Channel C Tone Period	8-BIT Fine Tune C							
R5						4-BIT Coarse Tune C			
R6	Noise Period				5-BIT Period Control				
R7	Enable	IN/OUT		Noise			Tone		
		IOB	IOA	C	B	A	C	B	A
R10	Channel A Amplitude				M	L3	L2	L1	L0
R11	Channel B Amplitude				M	L3	L2	L1	L0
R12	Channel C Amplitude				M	L3	L2	L1	L0
R13	Envelope Period	8-BIT Fine Tune E							
R14		8-BIT Coarse Tune E							
R15	Envelope Shape/Cycle					CONT	ATT	ALT	HOLD
R16	I/O Port A Data Store	8-BIT PARALLEL I/O on Port A							
R17	I/O Port B Data Store	8-BIT PARALLEL I/O Port B							

NOTE	OCTAVE	IDEAL FREQUENCY	ACTUAL FREQUENCY	12-BIT REGISTER VALUE IN OCTAL			
C	1	32.703	32.698	6	5	3	5
C#	1	34.648	34.653	6	2	3	4
D	1	36.708	36.712	5	7	4	7
D#	1	38.891	38.895	5	4	7	4
E	1	41.203	41.201	5	2	3	3
F	1	43.654	43.662	5	0	0	2
F#	1	46.249	46.243	4	5	6	3
G	1	48.999	48.997	4	3	5	3
G#	1	51.913	51.908	4	1	5	3
A	1	55.000	54.995	3	7	6	2
A#	1	58.270	58.261	3	6	0	0
B	1	61.735	61.733	3	4	2	4
C	2	65.406	65.416	3	2	5	6
C#	2	69.296	69.307	3	1	1	6
D	2	73.416	73.399	2	7	6	4
D#	2	77.782	77.789	2	6	3	6
E	2	82.406	82.432	2	5	1	5
F	2	87.308	87.323	2	4	0	1
F#	2	92.498	92.523	2	2	7	1
G	2	97.998	98.037	2	1	6	5
G#	2	103.826	103.863	2	0	6	5
A	2	110.000	109.991	1	7	7	1
A#	2	116.540	116.522	1	7	0	0
B	2	123.470	123.467	1	6	1	2
C	3	130.812	130.831	1	5	2	7
C#	3	138.592	138.613	1	4	4	7
D	3	146.832	146.799	1	3	7	2
D#	3	155.564	155.578	1	3	1	7
E	3	164.812	164.743	1	2	4	7
F	3	174.616	174.510	1	2	0	1
F#	3	184.996	184.894	1	1	3	5
G	3	195.996	195.903	1	0	7	3
G#	3	207.652	207.534	1	0	3	3
A	3	220.000	220.198	0	7	7	4
A#	3	233.080	233.043	0	7	4	0
B	3	246.940	246.933	0	7	0	5
C	4	261.624	261.357	0	6	5	4
C#	4	277.184	276.883	0	6	2	4
D	4	293.664	293.598	0	5	7	5
D#	4	311.128	310.724	0	5	5	0
E	4	329.624	329.973	0	5	2	3
F	4	349.232	349.565	0	5	0	0
F#	4	369.992	370.400	0	4	5	6
G	4	391.992	392.494	0	4	3	5
G#	4	415.304	415.839	0	4	1	5
A	4	440.000	440.397	0	3	7	6
A#	4	466.160	466.087	0	3	6	0
B	4	493.880	494.959	0	3	4	2
C	5	523.248	522.714	0	3	2	6
C#	5	554.368	553.766	0	3	1	2
D	5	587.328	588.741	0	2	7	6
D#	5	622.256	621.449	0	2	6	4
E	5	659.248	658.005	0	2	5	2
F	5	698.464	699.130	0	2	4	0
F#	5	739.984	740.800	0	2	2	7
G	5	783.984	782.243	0	2	1	7
G#	5	830.608	828.598	0	2	0	7
A	5	880.000	880.794	0	1	7	7
A#	5	932.320	932.173	0	1	7	0
B	5	987.760	989.918	0	1	6	1
C	6	1046.496	1045.428	0	1	5	3
C#	6	1108.736	1107.532	0	1	4	5
D	6	1174.656	1177.482	0	1	3	7
D#	6	1244.512	1242.898	0	1	3	2
E	6	1318.496	1316.009	0	1	2	5
F	6	1396.928	1398.260	0	1	2	0
F#	6	1479.968	1471.852	0	1	1	4
G	6	1567.968	1575.504	0	1	0	7
G#	6	1661.216	1669.564	0	1	0	3
A	6	1760.000	1747.825	0	1	0	0
A#	6	1864.640	1864.346	0	0	7	4
B	6	1975.520	1962.470	0	0	7	1
C	7	2092.992	2110.581	0	0	6	5
C#	7	2217.472	2237.216	0	0	6	2
D	7	2349.312	2330.433	0	0	6	0
D#	7	2489.024	2485.795	0	0	5	5
E	7	2636.992	2663.352	0	0	5	2
F	7	2793.856	2796.520	0	0	5	0
F#	7	2959.936	2943.705	0	0	4	6
G	7	3135.936	3107.244	0	0	4	4
G#	7	3322.432	3290.023	0	0	4	2
A	7	3520.000	3495.649	0	0	4	0
A#	7	3729.280	3728.693	0	0	3	6
B	7	3951.040	3995.028	0	0	3	4
C	8	4185.984	4142.992	0	0	3	3
C#	8	4434.944	4474.431	0	0	3	1
D	8	4698.624	4660.866	0	0	3	0
D#	8	4978.048	5084.581	0	0	2	6
E	8	5273.984	5326.704	0	0	2	5
F	8	5587.712	5593.039	0	0	2	4
F#	8	5919.872	5887.410	0	0	2	3
G	8	6271.872	6214.488	0	0	2	2
G#	8	6644.864	6580.046	0	0	2	1
A	8	7040.000	6991.299	0	0	2	0
A#	8	7458.560	7457.385	0	0	1	7
B	8	7902.080	7990.056	0	0	1	6

Fig. 23 EQUAL TEMPERED CHROMATIC SCALE (f_{CLOCK}=1.78977MHz)

3 OPERATION

Since all functions of the PSG are controlled by the host processor via a series of register loads, a detailed description of the PSG operation can best be accomplished by relating each PSG function to the control of its corresponding register. The function of creating or programming a specific sound or sound effect logically follows the control sequence listed:

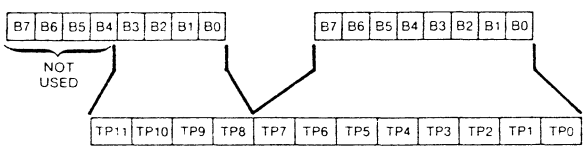
Section	Operation	Registers	Function
3.1	Tone Generator Control	R0--R5	Program tone periods.
3.2	Noise Generator Control	R6	Program noise period.
3.3	Mixer Control	R7	Enable tone and/or noise on selected channels.
3.4	Amplitude Control	R10--R12	Select "fixed" or "envelope-variable" amplitudes.
3.5	Envelope Generator Control	R13--R15	Program envelope period and select envelope pattern.

3.1 Tone Generator Control

(Registers R0, R1, R2, R3, R4, R5)

The frequency of each square wave generated by the three Tone Generators (one each for Channels A, B, and C) is obtained in the PSG by first counting down the input clock by 16, then by further counting down the result by the programmed 12-bit Tone Period value. Each 12-bit value is obtained in the PSG by combining the contents of the relative Coarse and Fine Tune registers, as illustrated in the following:

Coarse Tune Register	Channel	Fine Tune Register
R1	A	R0
R3	B	R2
R5	C	R4



12-bit Tone Period (TP) to Tone Generator

Note that the 12-bit value programmed in the combined Coarse and Fine Tune registers is a period value—the higher the value in the registers, the lower the resultant tone frequency.

Note also that due to the design technique used in the Tone Period count-down, the lowest period value is 000000000001 (divide by 1) and the highest period value is 111111111111 (divide by $4{,}095_{10}$).

The equations describing the relationship between the desired output tone frequency and the input clock frequency and Tone Period value are:

$$\text{(a) } f_T = \frac{f_{CLOCK}}{16TP_{10}} \qquad \text{(b) } TP_{10} = 256CT_{10} + FT_{10}$$

Where: f_T = desired tone frequency
f_{CLOCK} = input clock frequency
TP_{10} = decimal equivalent of the Tone Period bits TP11--TP0.
CT_{10} = decimal equivalent of the Coarse Tune register bits B3--B0 (TP11--TP8)
FT_{10} = decimal equivalent of the Fine Tune register bits B7--B0 (TP7--TP0)

From the above equations it can be seen that the tone frequency can range from a low of $\frac{f_{CLOCK}}{65,520}$ (wherein: TP_{10}=4,095$_{10}$) to a high of $\frac{f_{CLOCK}}{16}$ (wherein: TP_{10}=1). Using a 2 MHz input clock, for example, would produce a range of tone frequencies from 30.5 Hz to 125 kHz.

To calculate the values for the contents of the Tone Period Coarse and Fine Tune registers, given the input clock and the desired output tone frequencies, we simply rearrange the above equations, yielding:

$$\text{(a) } TP_{10} = \frac{f_{CLOCK}}{16f_T} \qquad \text{(b) } CT_{10} + \frac{FT_{10}}{256} = \frac{TP_{10}}{256}$$

Example 1: $f_T = 1kHz$
$f_{CLOCK} = 2MHz$

$$TP_{10} = \frac{2x10^6}{16(1x10^3)} = 125$$

Substituting this result into equation (b):

$$CT_{10} + \frac{FT_{10}}{256} = \frac{125}{256}$$

$\therefore CT_{10} = 0 \quad = 0000$ (B3--B0)
$FT_{10} = 125_{10} = 01111101$ (B7--B0)

Example 2: $f_T = 100Hz$
$f_{CLOCK} = 2MHz$

$$TP_{10} = \frac{2x10^6}{16(1x10^2)} = 1250$$

Substituting this result into equation (b):

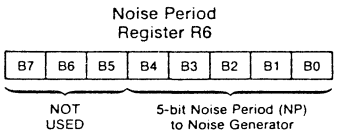
$$CT_{10} + \frac{FT_{10}}{256} = \frac{1250}{256} = 4 + \frac{226}{256}$$

$\therefore CT_{10} = 4_{10} = 0100$ (B3--B0)
$FT_{10} = 226_{10} = 11100010$ (B7--B0)

3.2
Noise Generator Control

(Register R6)

The frequency of the noise source is obtained in the PSG by first counting down the input clock by 16, then by further counting down the result by the programmed 5-bit Noise Period value. This 5-bit value consists of the lower 5 bits (B4--B0) of register R6, as illustrated in the following:



Note that the 5-bit value in R11 is a period value—the higher the value in the register, the lower the resultant noise frequency. Note also that, as with the Tone Period, the lowest period value is 00001 (divide by 1); the highest period value is 11111 (divide by 31_{10}).

The noise frequency equation is:

$$f_N = \frac{f_{CLOCK}}{16\ NP_{10}}$$

Where: f_N = desired noise frequency
f_{CLOCK} = input clock frequency
NP_{10} = decimal equivalent of the Noise Period register bits B4--B0.

From the above equation it can be seen that the noise frequency can range from a low of $\frac{f_{CLOCK}}{496}$ (wherein: $NP_{10} = 31_{10}$) to a high of $\frac{f_{CLOCK}}{16}$ (wherein: $NP_{10} = 1$). Using a 2 MHz input clock, for example, would produce a range of noise frequencies from 4 kHz to 125 kHz.

To calculate the value for the contents of the Noise Period register, given the input clock and the desired output noise frequencies, we simply rearrange the above equation, yielding:

$$NP_{10} = \frac{f_{CLOCK}}{16\ f_N}$$

3.3
Mixer Control-I/O Enable

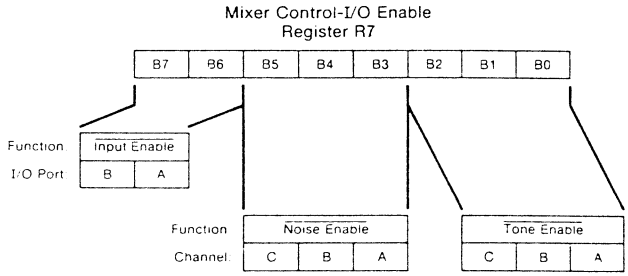
(Register R7)

Register 7 is a multi-function $\overline{\text{Enable}}$ register which controls the three Noise/Tone Mixers and the two general purpose I/O Ports.

The Mixers, as previously described, combine the noise and tone frequencies for each of the three channels. The determination of combining neither/either/both noise and tone frequencies on each channel is made by the state of bits B5--B0 of R7.

The direction (input or output) of the two general purpose I/O Ports (IOA and IOB) is determined by the state of bits B7 and B6 of R7.

These functions are illustrated in the following:



Noise Enable Truth Table:

R7 Bits B5	B4	B3	Noise Enabled on Channel		
0	0	0	C	B	A
0	0	1	C	B	—
0	1	0	C	—	A
0	1	1	C	—	—
1	0	0	—	B	A
1	0	1	—	B	—
1	1	0	—	—	A
1	1	1	—	—	—

Tone Enable Truth Table:

R7 Bits B2	B1	B0	Tone Enabled on Channel		
0	0	0	C	B	A
0	0	1	C	B	—
0	1	0	C	—	A
0	1	1	C	—	—
1	0	0	—	B	A
1	0	1	—	B	—
1	1	0	—	—	A
1	1	1	—	—	—

I/O Port Truth Table:

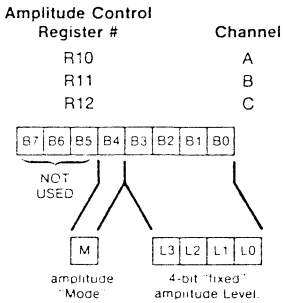
R7 Bits B7	B6	I/O Port Status IOB	IOA
0	0	Input	Input
0	1	Input	Output
1	0	Output	Input
1	1	Output	Output

NOTE: Disabling noise and tone does not turn off a channel. Turning a channel off can only be accomplished by writing all zeroes into the corresponding Amplitude Control register, R10, R11, or R12 (see Section 3.4).

3.4 Amplitude Control

(Registers R10, R11, R12)

The amplitudes of the signals generated by each of the three D/A Converters (one each for Channels A, B, and C) is determined by the contents of the lower 5 bits (B4--B0) of registers R10, R11, and R12 as illustrated in the following:



The amplitude "mode" (bit M) selects either fixed level amplitude (M=0) or variable level amplitude (M=1). It follows then that bits L3--L0, defining the value of a "fixed" level amplitude, are only active when M=0. When fixed level amplitude is selected, it is "fixed" only in the sense that the amplitude level is under the direct control of the system processor (via bits D3--D0). Varying the amplitude when in this "fixed" amplitude mode requires in each instance the direct intervention of the system processor via an address latch/write data sequence to modify the D3--D0 data.

When M=1 (select "variable" level amplitudes), the amplitude of each channel is determined by the envelope pattern as defined by the Envelope Generator's 4-bit output E3 E2 E1 E0.

The amplitude "mode" (bit M) can also be thought of as an "envelope enable" bit; i.e., when M=0 the envelope is not used, and when M=1 the envelope is enabled. (A full description of the Envelope Generator function follows in Section 3.5).

The full chart describing all combinations of the 5-bit Amplitude Control is as follows:

Amplitude Control Register #	Channel
R10	A
R11	B
R12	C

B7	B6	B5	B4	B3	B2	B1	B0
NOT USED							

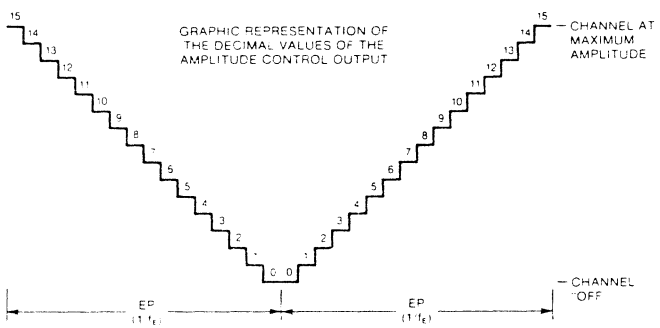
M	L3	L2	L1	L0	Amplitude Control Output	
0	0	0	0	0	* 0 0 0 0	The amplitude is fixed at 1 of 16 levels as determined by L3 L2 L1 L0.
.	.	.	.	.	. . . .	
.	.	.	.	.	. . . .	
.	.	.	.	.	. . . .	
0	1	1	1	1	1 1 1 1	
1	X	X	X	X	E3 E2 E1 E0	The amplitude is variable at 16 levels as determined by the output of the Envelope Generator.

(X=Don't Care)

*The all zeroes code is used to turn a channel "off".

Fig. 6 graphically illustrates a selection of variable level (envelope-controlled) amplitude where the 16 levels directly reflect the output of the Envelope Generator. A fixed level amplitude would correspond to only one of the levels shown, with the level directly determined by the decimal equivalent of bits L3 L2 L1 L0.

Fig. 6 VARIABLE AMPLITUDE CONTROL (M=1)



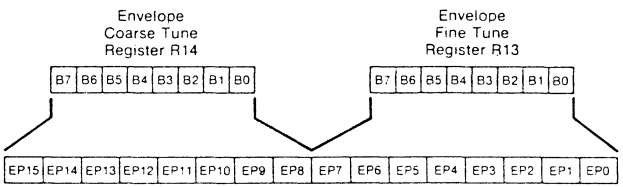
3.5 Envelope Generator Control

(Registers R13, R14, R15)

To accomplish the generation of fairly complex envelope patterns, two independent methods of control are provided in the PSG: first, it is possible to vary the frequency of the envelope using registers R13 and R14; and second, the relative shape and cycle pattern of the envelope can be varied using register R15. The following paragraphs explain the details of the envelope control functions, describing first the envelope period control and then the envelope shape/cycle control.

3.5.1 ENVELOPE PERIOD CONTROL (Registers R13, R14)

The frequency of the envelope is obtained in the PSG by first counting down the input clock by 256, then by further counting down the result by the programmed 16-bit Envelope Period value. This 16-bit value is obtained in the PSG by combining the contents of the Envelope Coarse and Fine Tune registers, as illustrated in the following:



16-bit Envelope Period (EP)
to Envelope Generator

Note that the 16-bit value programmed in the combined Coarse and Fine Tune registers is a period value—the higher the value in the registers, the lower the resultant envelope frequency.

Note also, that as with the Tone Period, the lowest period value is 0000000000000001 (divide by 1); the highest period value is 1111111111111111 (divide by $65{,}535_{10}$).

The envelope frequency equations are:

$$\text{(a) } f_E = \frac{f_{CLOCK}}{256EP_{10}} \qquad \text{(b) } EP_{10} = 256CT_{10} + FT_{10}$$

Where:
f_E = desired envelope frequency
f_{CLOCK} = input clock frequency
EP_{10} = decimal equivalent of the Envelope Period bits EP15--EP0
CT_{10} = decimal equivalent of the Coarse Tune register bits B7--B0 (EP15--EP8)
FT_{10} = decimal equivalent of the Fine Tune register bits B7--B0 (EP7--EP0)

From the above equation it can bee seen that the envelope frequency can range from a low of $\frac{f_{CLOCK}}{16{,}776{,}960_{10}}$ (wherein: $EP_{10}=65{,}535_{10}$) to a high of $\frac{f_{CLOCK}}{256}$ (wherein: $EP_{10}=1$). Using a 2 MHz clock, for example, would produce a range of envelope frequencies from 0.12 Hz to 7812.5 Hz.

To calculate the values for the contents of the Envelope Period Coarse and Fine Tune registers, given the input clock and the desired envelope frequencies, we rearrange the above equations, yielding:

$$\text{(a) } EP_{10} = \frac{f_{CLOCK}}{256 f_E} \qquad \text{(b) } CT_{10} + \frac{FT_{10}}{256} = \frac{EP_{10}}{256}$$

Example: $f_E = 0.5$ Hz
$f_{CLOCK} = 2$ MHz

$$EP_{10} = \frac{2 \times 10^6}{256(0.5)} = 15.625$$

Substituting this result into equation (b):

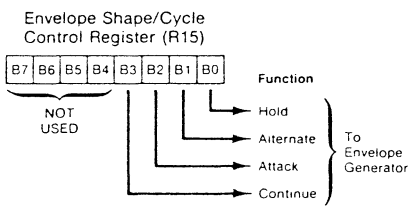
$$CT_{10} + \frac{FT_{10}}{256} = \frac{15.625}{256} = 61 + \frac{9}{256}$$

$CT_{10} = 61_{10} = 00111101$ (B7--B0)
$FT_{10} = 9_{10} = 00001001$ (B7--B0)

3.5.2 ENVELOPE SHAPE/CYCLE CONTROL (Register R15)

The Envelope Generator further counts down the envelope frequency by 16, producing a 16-state per cycle envelope pattern as defined by its 4-bit counter output, E3 E2 E1 E0. The particular shape and cycle pattern of any desired envelope is accomplished by controlling the count pattern (count up/count down) of the 4-bit counter and by defining a single-cycle or repeat-cycle pattern.

This envelope shape/cycle control is contained in the lower 4 bits (B3--B0) of register R15. Each of these 4 bits controls a function in the envelope generator, as illustrated in the following:



The definition of each function is as follows:

Hold — when set to logic "1", limits the envelope to one cycle, holding the last count of the envelope counter (E3--E0=0000 or 1111, depending on whether the envelope counter was in a count-down or count-up mode, respectively).

Alternate — when set to logic "1", the envelope counter reverses count direction (up-down) after each cycle.

NOTE: When both the Hold bit and the Alternate bit are ones, the envelope counter is reset to its initial count before holding

3.5 Envelope Generator Control (cont.)

Attack — when set to logic "1", the envelope counter will count up (attack) from E3 E2 E1 E0=0000 to E3 E2 E1 E0=1111; when set to logic "0", the envelope counter will count down (decay) from 1111 to 0000.

Continue — when set to logic "1", the cycle pattern will be as defined by the Hold bit; when set to logic "0", the envelope generator will reset to 0000 after one cycle and hold at that count.

To further describe the above functions could be accomplished by numerous charts of the binary count sequence of E3 E2 E1 E0 for each combination of Hold, Alternate, Attack and Continue. However, since these outputs are used (when selected by the Amplitude Control registers) to amplitude modulate the output of the Mixers, a better understanding of their effect can be accomplished via a graphic representation of their value for each condition selected, as illustrated in Figs. 7 and 8.

Fig. 7 ENVELOPE SHAPE/CYCLE CONTROL

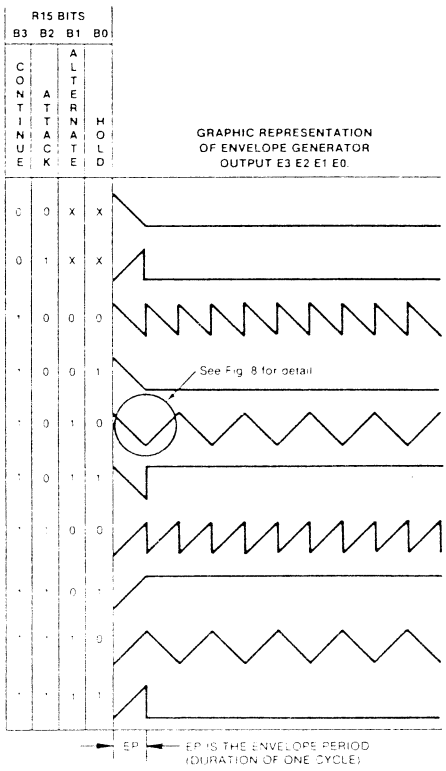
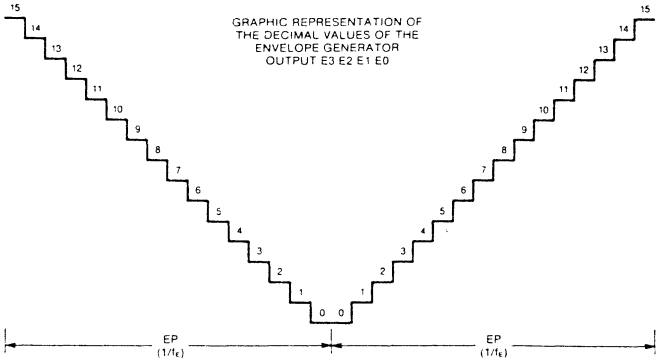


Fig. 8 DETAIL OF TWO CYCLES OF Fig. 7
(ref. waveform "1010" in Fig. 7)



6 SOUND EFFECTS GENERATION

One of the main uses of the PSG is to produce non-musical sound effects to accompany visual action or as a feature in itself. The following sections outline techniques and provide actual examples of some popular effects. All examples are based on a 1.78977MHz PSG clock.

6.1 Tone Only Effects

Many effects are possible using only the tone generation capability of the PSG without adding noise and without using the PSG's envelope generation capability. Examples of this type of effect would include telephone tone frequencies (two distinct frequencies produced simultaneously) or the European Siren effect listed in Fig. 27 (two distinct frequencies sequentially produced).

Fig. 27 EUROPEAN SIREN SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R0	376	Set Channel A Tone period to 2.27ms
R1	000	(440Hz).
R7	076	Enable Tone only on Channel A only.
R10	017	Select maximum amplitude on Channel A.
	(Wait approximately 350ms before continuing)	
R0	126	Set Channel A Tone period to 5.346ms
R1	001	(187Hz).
	(Wait approximately 350ms before continuing)	
R10	000	Turn off Channel A to end sound effect.

6.2 Noise Only Effects

Some of the more commonly required sounds require only the use of noise and the envelope generator (or processor control of channel envelope if other channels are using the envelope generator).

Examples of this, which can be seen in Figs. 28 and 29, are gunshot and explosion. In both cases pure noise is used with a decaying envelope. In the examples shown the only changes are in the length of the envelope as modified by the coarse tune register and in the noise period. Note that a significantly lower explosion can be obtained by using all three channels operating with the same parameters.

Fig. 28 GUNSHOT SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R6	017	Set Noise period to mid-value.
R7	007	Enable Noise only on Channels A,B,C.
R10	020	Select full amplitude range under direct control of Envelope Generator.
R11	020	
R12	020	
R14	020	Set Envelope period to 0.586 seconds.
R15	000	Select Envelope "decay", one cycle only.

Fig. 29 EXPLOSION SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R6	000	Set Noise period to max. value.
R7	007	Enable Noise only, on Channels A,B,C.
R10	020	Select full amplitude range under direct control of Envelope Generator.
R11	020	
R12	020	
R14	070	Set Envelope period to 2.05 seconds.
R15	000	Select Envelope "decay", one cycle only.

6.3 Frequency Sweep Effects

The Laser, Whistling Bomb, Wolf Whistle, and Race Car sounds in Figs. 30 thru 33 all utilize frequency sweeping effects. In all cases they involve the increasing or decreasing of the values in the tone period registers with variable start, end, and time between frequency changes. For example, the sweep speed of the Laser is much more rapid than the high gear accelerate in the race car, yet both use the same computer routine with differing parameters.

Other easily achievable results include "doppler" and noise sweep effects. The sweeping of the noise clocking register (R6) produces a "doppler" effect which seems well suited for "space war" type games.

Fig. 30 LASER SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R7	076	Enable Tone only on Channel A only.
R10	017	Select maximum amplitude on Channel A.
R0	060 (start)	Sweep effect for Channel A Tone period via a processor loop with approximately 3ms wait time between each step from 060 to 160 (0.429ms/2330Hz to 1.0ms/1000Hz).
R0	160 (end)	
R10	000	Turn off Channel A to end sound effect.

Fig. 31 WHISTLING BOMB SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R7	076	Enable Tone only on Channel A only.
R10	017	Select maximum amplitude on Channel A.
R0	060 (start)	Sweep effect for Channel A Tone period via a processor loop with approximately 25ms wait time between each step from 060 to 300 (0.429ms/2330Hz to 1.72ms/582Hz).
R0	300 (end)	

After above loop is completed, follow with sequence in Fig. 28.

6.4 Multi-Channel Effects

Because of the independent architecture of the PSG, many rather complex effects are possible without burdening the processor. For example, the Wolf Whistle effect in Fig. 32 shows two channels in use to add constant breath hissing noise to the three concentrated frequency sweeps of the whistle. Once the noise is put on the channel, the processor only need be concerned with the frequency sweep operation.

Fig. 32 WOLF WHISTLE SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R6	001	Set Noise period to minimum value.
R7	056	Enable Tone on Channel A, Noise on Channel B.
R10	017	Select maximum amplitude on Channel A.
R11	011	Select lower amplitude on Channel B.
R0	100 (start)	Sweep effect for Channel A Tone period via a processor loop with approximately 12ms wait time between each step from 100 to 040 (0.572ms/1748Hz to 0.286ms/3496Hz).
R0	040 (end)	
(Wait approximately 150ms before continuing)		
R0	100 (start)	A processor loop with approximately 25ms wait time between each step from 100 to 060 (0.572ms/1748Hz to 0.429ms/2331Hz).
R0	060 (end)	
R0	060 (start)	A processor loop with approximately 6ms wait time between each step from 060 to 150 (0.429ms/2331Hz to 0.930ms/1075Hz).
R0	150 (end)	
R10	000	Turn off Channels A and B to end effect.
R11	000	

Fig. 33 RACE CAR SOUND EFFECT CHART

Register #	Octal Load Value	Explanation
Any not specified	000	—
R3	017	Set Channel B Tone period to 34.33ms (29Hz).
R7	074	Enable Tones only on Channels A and B.
R10	017	Select maximum amplitude on Channel A.
R11	012	Select lower amplitude on Channel B.
*R1/R0	013/000 (start)	Sweep effect for Channel A Tone period via a processor loop with approximately 3ms wait time between each step from 013/000 to 004/000 (25.17ms/39.7Hz to 9.15ms/109.3Hz).
*R1/R0	004/000 (end)	
R1/R0	011/000 (start)	A processor loop with approximately 3ms wait time between each step from 011/000 to 003/000 (20.6ms/48.5Hz to 6.87ms/145.6Hz).
R1/R0	003/000 (end)	
R1/R0	006/000 (start)	A processor loop with approximately 6ms wait time between each step from 006/000 to 001/000 (13.73ms/72.8Hz to 2.29ms/436.7Hz).
R1/R0	001/000 (end)	
R10	000	Turn off Channels A and B to end effect.
R11	000	

* Decrement R1/R0 as a whole number, e.g. start at 013/000, then 012/377, then 012/376, etc.

INDEX

For MSX Programmers – your constant companion

This is not a book for the raw beginner - rather, it takes the MSX programmer "inside" the computer to see exactly how it works and how to get the most from it.

Chris Burkinshaw is a programmer with immense experience, and has already completed one book for Sigma "Beyond BASIC on Your Commodore 64". Likewise, **Ross Goodley** is well respected in computing circles and is well-known for the successful games he has written for Alligata Software Ltd.

These authors have divided their book into two parts.

The first part covers system design, how the BASIC vocabulary relates to MSX machines, and an introduction to machine code. You will find detailed explanations of memory organisation, display modes, and the VDP and sound chips.

The second part is concerned with the use of assembly language in the MSX environment. Major sections include:

* The Video Display Processor
* The AY-3-8910 Sound Chip
* Input/Output

Whether you need to write high-performance programs, or just want to know how your computer works, this book will be invaluable. We feel that it will rapidly become the standard text for MSX programmers who want to do more than just write BASIC programs.

Sigma Press produce the widest range of stimulating books for the computer professional – and we're always happy to evaluate new proposals.

Contact Sigma at:

5 Alton Road
Wilmslow
Cheshire
SK9 5DY

GB £ NET +007.95

ISBN 1-85058-015-4

00795

9 781850 580157